Listing of the basic C+− files

Ulrich Mutze
www.ulrichmutze.de

October 20, 2023

Contents

1 Introduction 4

2 Credit 5

3 Legal Matters 5

4 cpmangle.h 6

5 cpmangle.cpp 12

6 cpmbas.h 20

7 cpmbas.cpp 21

8 cpmbasicinterfaces.h 22

9 cpmbasictypes.h 24

10 cpmc.h 26

11 cpmc.cpp 34

12 cpmcompdef.h 41

13 cpmdefinitions.h 44

14 cpmf.h 45

15 cpmfa.h 55

16 cpmfl.h **65**

17 cpmfo.h **88**

18 cpmfr.h **92**

19 cpmgreg.h **100**

20 cpmgreg.cpp **106**

21 cpminterfaces.h **114**

22 cpmm.h **124**

23 cpmm2.h **139**

24 cpmmacros.h **150**

25 cpmmpi.h **152**

26 cpmmpi.cpp **159**

27 cpmnumbers.h **164**

28 cpmnumbers.cpp **199**

29 cpmp.h **207**

30 cpms.h **215**

31 cpmsr.h **222**

32 cpmsystem.h **228**

33 cpmsystem.cpp **248**

34 cpmsystemdependencies.h **266**

35 cpmtests.h **268**

36 cpmtypes.h **304**

37 cpmtypes.cpp **314**

38 cpmuc.h **323**

39 cpmuc.cpp **328**

40 cpmv.h **329**

41 cpmv.cpp **383**

42 cpmva.h **385**

43 cpmvcow.h **400**

44 cpmviewport.h **455**

45 cpmviewport.cpp **466**

46 cpmvlin.h **478**

47 cpmvm.h **482**

48 cpmvo.h **485**

49 cpmvr.h **501**

50 cpmvsp.h **506**

51 cpmvuc.h **510**

52 cpmword.h **563**

53 cpmword.cpp **578**

54 cpmx.h **589**

55 cpmzinterval.h **611**

56 cpmzinterval.cpp **620**

57 testcpm0.cpp **626**

58 tut1.cpp **639**

59 tut1experimenta.cpp **653**

60 tut1experimenta.cpp **655**

61 tut1old1.cpp **657**

62 tut1old1.cpp **668**

63 tut1old2.cpp **679**

64 tut1old2.cpp **701**

65 survey.txt **723**

66 headerdependencies.txt **735**

1 Introduction

This is a listing of the basic C+− source files. This is the the part which is not related to physics and is not concerned with creating graphical representations of the objects under consideraton. It is very similar in scope to the part of the C++ Standard Libary which was once the Standard Template Library (STL). See file *cpmProject.pdf* for motivation and details concerning the C+− project.

The C++ files with names beginning with `cpm` constitute a library (i.e. a set of programming tools) whereas the other files — in the present case this is only the single file testcpm0.cpp — are taken from my C+− - based applications. These are included to provide examples for usage of the tool classes. The search capability of the *Adobe Reader* program allows to find places where specified classes are employed. These search capabilities also allow us to extract from a verbatim code listing all informations that one may obtain from the graphically more appealing documentations generated by the *doxygen* program.

In the following part of the document, there is one section (in the sense of LaTex) for each file and the files are in alphabetic order with the exception that header files (*.h) precede implementation files (*.cpp) of the same primary name. Since there is table of content, that shows these sections, it is not difficult to navigate within this moderately large files system.

In trying to get an overview of the content of this file collection one should also consult the final files survey.txt and headerdependencies.txt. As is mentioned in the header of these files they are auto-generated by means of *Ruby* programs. I would not have been able to carry out all the testing and documentation of C+− files without my rich tool box of Ruby programs. In filling this tool box piece by piece, I became a Ruby enthusiast, wheres in the beginning it was more or less by chance that I turned to just this language. I was happy enough that, when the need of extended scripting occurred to me, Ruby was available. Those of my colleagues for whom this need arose earlier, had their personal toolbox yet full of *Perl* scripts and for me — an enthusiast of C++ and an admirer of Stroustrup's writing — it was hard to understand their enthusiasm for Perl. So I tried Ruby and found it to be in some (restricted) sense 'a better C++'. For some of C+− programming style decisions, Ruby was the model, and for others, for

which agreement with Ruby strategies became clear after their introduction to C+− , this backed up my confidence to be on a viable track.

2 Credit

Many of the more complex algoritms contained in the present code are modified from sources in the wonderful book *Numerical Recipies in C* by William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, of which I own and use a copy of the second edition. In all these uses I made sustantial modifications by introducing the C+− data and array types, promoting C-functions to member functions of suitable C++ classes, and by iserting diagnostic message generators. Searching for '*Press*' in the present document shows all instances of such uses. For the programming language related matters my sources of advice and inspiration are indicated in file *cpmProject.pdf*.

3 Legal Matters

4 cpmangle.h

```
//? cpmangle.h
//? Status of work 2023-10-20.
//?
//?  ...


#ifndef CPM_ANGLE_H_
#define CPM_ANGLE_H_
//////////////////////////////////////////////////////////////////////////
// Description: A class describing angles and the 1-dimensional
//   torus
//////////////////////////////////////////////////////////////////////////
#include <cpmc.h>
#include <cpmtypes.h>
#include <cpmv.h>

namespace CpmGeo{

   using CpmRoot::R;
   using CpmRoot::Z;
   using CpmRoot::Word;
   using CpmRoot::C;
   using CpmArrays::V;

enum AngleUnit { CYCLE, HOUR, DEG, RAD};
   // cycle (full angle), hour, degree, radian
enum AngleRange { POS, NEG};
   // always positive, negative allowed
enum AngleHex {DEC, MIN, SEC};
   // full decimal, minutes, minutes and seconds
// friend functions need to be declared as belonging to the
// present namespace
class Angle;
Angle mean(const CpmArrays::V<Angle>& phi);
Angle mean(const CpmArrays::V<Angle>& angles,
   const CpmArrays::V<R>& weights);
Angle RealToAngle(R x, AngleUnit u, AngleHex h);
Angle realToAngle(R c);

class Angle{ // angles
   // An instance of Angle is an angle, the manifold of all angles is
   // a 1-dimensional torus. There are two interpretations.
   // (i) Angle as a coordinate on a circle (as a curve) and
   // (ii) angle as a descriptor of a rotation arround a given axis.
   // Both are closely related due to the fact, that the rotations act
   // transitively and effectively on a circle. For defining a proper set
   // of methods we have not to make a choice in favor of one of these
```

```
   // interpretations.
   // Interpretation (i) endows Angle most directly with the structure of
   // an oriented metric space, and (ii) with the structure of an
   // Abelian group.

   static void normalizeAngle(R& a0);
      // changes a0 into a value in the interval (-Pi,Pi] by adding
      // a multiples of 2*Pi

   R a_; // is read as alpha, is the measure of the angle as arc length
      // on the unit circle (for effective usage of built-in
      // trigonometric functions. The range is by definition (-Pi,Pi].

   void normalize(void){ normalizeAngle(a_);}

public:
   typedef Angle Type;
   typedef R ScalarType;

   CPM_IO
   CPM_ORDER

   CPM_SUM_PLAIN
   CPM_DIFFERENCE_PLAIN
      // Addition and subtraction among angles as borrowed from the
      // mathematical model by residue classes in R 'modulo 2*Pi'.
      // A further mathematical model is given by
      // { z \in C : |z|=1 }. In  this model addition of angles corresponds
      // to multiplication of complex numbers and subtraction of angles
      // corresponds  to division of numbers. If we are simply given a
      // torus (e.g. as a thin ring of polished metal) then we need to mark
      // on it a point as origin and an arrow as direction in order map it
      // to any of the mathematical models considered here.
      // This implies that the algebraic structure of an abelian group
      // exhibited by both mathematical models is not inherent in the
      // geometry of the pure torus but depends on the selection of an
      // origin and a direction (orientation) as indicated above. Obviously
      // the selected origin plays the role of the neutral element of the
      // abelian group. Notice that the pure torus is a metric space
      // where the distance between two points is the length of the
      // curve along the torus (i.e. the shortest arc) which connects the
      // points.

   CPM_SCALAR_M
   // The meaning of multiplication with real numbers derives by
   // continuity from the meaning of multiplication by rational
   // numbers. Let us consider first multiplication by a natural
   // number n. The proper definition follows from the group
   // structure: alpha*n := alpha + ... + alpha (n terms).
   // Multiplication by 1/m for natural number:
```

```
   // alpha*(1/m) := smallest angle beta for which beta*m=Angle(),
   // where Angle() is the zero angle which is the unit element of
   // the group. Smallness, obviously, refers to the distance to
   // the unit element. Obviously, one may add (2*Pi)/m to beta
   // (let the result be called beta') and one also has
   // beta'*m=Angle().

   CPM_CONJUGATION
      // An idempotent conjugation operation. If the angle is interpreted
      // as the phase angle of a complex number, this corresponds to
      // the complex conjugation of the number. This is a natural
      // geometric operations for angles, resulting from the distinction
      // of one angle as the zero-angle (origin on torus).

   Word nameOf()const{ return Word("Angle");}

   R operator /(const Angle&)const;
      // alpha/beta is the number r for which beta*r=alpha.
      // Notice the comment to ScalarType on multiplying
      // Angles by numbers. This is not defined in residue
      // classes modulo 2*Pi.
      // Products of angles don't give angles; they are
      // omitted here.

   Angle operator+(R shift)const{ return Angle(a_+shift,false);}
      // Returning the angle which results from *this by adding an arc
      // which may be any length (e.g. much larger than 2*Pi) and sign.
      // This gives Angle the structure of an affine space.

// constructors

   Angle(void):a_(0.){}
      // angle zero

   explicit Angle(R angle, bool deg=true);
      // Constructor for Angles out of real numbers. The first argument
      // is always interpreted as an angle in degrees, unless deg==0
      // Never enable automatic conversion. Would result in ambiguous
      // arithmetics. Input needs not to be normalized.

   Angle(R vAngle, AngleUnit au);
      // creates an angle which is vAngle*au

   explicit Angle(C const& z):a_(z.arg()){}
      // Constructs the phase angle (polar angle) of z

   C expi()const{ return C(1.,a_,"polar");}
      // Returns the complex number exp(i*a). Thus Angle now has a
      // bi-directional efficient interface to C.
```

```
// automatic conversion in one direction Angle-->R is OK

   operator R(void)const{return a_;}
   // cast to R for making trigonometric functions defined for angles
   // No longer used in the implementation of Angle.

   R toR()const{ return a_;}
   // Explicit conversion in a standard manner. The implementation of
   // Angle now relies on this.

   R cut( R c)const;
   //: cut
   // Returns a value in the range (c*degree-2*Pi,c*degree] which from
   // a_ is derived by adding or subtracting a multiple of 2*Pi.
   // To have such a facility turned out to be very useful in
   // dealing with angular motion of stepper motors.

   void qun_(Z n);
      //: quantize
      // Changes angle *this into the nearest of n equi-spaced
      // angles which are equi-distributed over the whole range.
      // For n<1, no action.

   R toDeg(Z i=0)const;
      //: to degrees
      // Returns the angle *this in degrees. For the default
      // value 0 of the argument, the range is the standard range
      // (-180,180], and for 1 the range is [0,360).

   Word toWord(AngleUnit u, AngleRange r, AngleHex h, Z digits=1)const;
      // Returns a string which gives the angle in the units
      // given by u, in the range indicated by r, in a hexagesimal
      // subdivision characterized by h and with a number of figures
      // after the decimal point is given by digits. If digits is -1,
      // there is one figure behind the decimal point and this is rounded
      // to 0 or 5 corresponding to the accuracy with which the hour
      // circle of my C5 telescope can be read. The meaning of the
      // parameters follows from the explanation to the
      // enumerations AngleUnit, AngleRange, and AngleHex.

   friend Word AngleToWord(const Angle& alpha,
      AngleUnit u, AngleRange r, AngleHex h, Z digits=1)
      { return alpha.toWord(u,r,h,digits);}

// modifying an angle
   void setArc(R alpha);
      // input is an arc (i.e. something like a, but need not be
      // normalized)

// absolute value
```

```
   R abs(void)const{ return (a_>=0. ?  a_ : -a_);}
      // absolute value: positive arc as a number (distance from zero
      // angle)

   R dis(const Angle& phi)const;
      //: distance
      // Lets angles form a metric space. Is symmetric in the arguments.
      // Implementation corrected 2016-03-25 (symmetry was not guaranteed
      // before)

   bool isPos()const{ return a_>=0;}
      // returns true if the angle is in quadrant 1 or 2 (by definition,
      // 0 belongs to quadrant 1, Pi to quadrant 2 ) and false else

   void pos(){ if (a_<0) a_=-a_;}
      // makes the angle positive by conjugation

// mean values taking into account the embedding in the complex numbers
   friend Angle mean(const CpmArrays::V<Angle>& phi);
      // mean of an array of angles.
      // Polar angle of exp(i*phi[1])+...+exp(i*phi[n]).
      // There seems to be no easy way to define this on R/~
      // where a1 ~ a2 <==> a1 = a2 (mod 2Pi)

   friend Angle mean(const CpmArrays::V<Angle>& angles,
      const CpmArrays::V<R>& weights);
      // weighted mean of an array of angles
      // Polar angle of
      // weights[1]*exp(i*phi[1])+...+weights[n]*exp(i*phi[n])

   friend Angle RealToAngle(R x, AngleUnit u, AngleHex h);
      // Returns an angle alpha which is given by a real number
      // according the format indicated by the unit u and the
      // subdivision schema indicated by h.

   friend Angle realToAngle(R c){ return Angle(c);}
      // Converts a real number, interpreted as an arc into a
      // instance of class Angle.

   static V<R> smoothAngleList(const V<Angle>& va);
      // Given a list va of Angles, we return a list res of corresponding
      // R's such that
      //    res[i]==va[i] mod(2*Pi)
      // where adding or subtracting of 2*Pi is done such that the
      // resulting list is as smooth as possible. More precisely we
      // assure that for each i
      // | res[i+i]-res[i]| < |res[i+1]+2*Pi-res[i]|
      // and
      // | res[i+i]-res[i]| < |res[i+1]-2*Pi-res[i]|
      // As a result of this the res[i] may grow to much larger values
```

```
      // than 2*Pi as in a list of successive angular positions of a
      // rotating wheel.

    static V<R> smoothList(const V<R>& vr);
      // Similar to previous function. The argument list is not a list
      // of angles but a list of their conversions to numbers (based
      // on the RAD unit).

    static Angle deg1, deg90, deg180, deg270, deg360;
};

} // namespace

#endif
```

5 cpmangle.cpp

```
//? cpmangle.cpp
//? Status of work 2023-10-20.
//?
//?  ...


//Description: see cpmangle.h

#include <cpmangle.h>

using namespace CpmStd;

using namespace CpmGeo;
using CpmRoot::R;
using CpmRoot::Z;
using CpmRoot::C;
using CpmRoot::Word;

namespace{
   const R fullAngle=cpmpi*2;
   const R iFullAngle=R(1.)/fullAngle;
   const R hour=fullAngle/24;
}

Angle Angle::deg1(1,DEG);
Angle Angle::deg90(90,DEG);
Angle Angle::deg180(180,DEG);
Angle Angle::deg270(270,DEG);
Angle Angle::deg360(360,DEG);

void Angle::normalizeAngle(R& a)
{
   if (a>-cpmpi && a<=cpmpi) return;
      // then no normalization needed. Introduced 2006-03-29
      // after it was observed that normalizing the zero-angle
      // caused difficulties
   a*=iFullAngle;
   a-=cpmfloor(a);
   if (a>0.5) a-=1;
   a*=fullAngle;
}

Angle::Angle(R angle, bool deg):a_(angle)
{
   if (deg) a_*=cpmdeg;
   normalize();
}
```

```
Angle::Angle(R angle, AngleUnit au):a_(angle)
{
   if (au==RAD){
      ;
   }
   else if (au==DEG){
      a_*=cpmdeg;
   }
   else if (au==CYCLE){
      a_*=fullAngle;
   }
   else if (au==HOUR){
      a_*=hour;
   }
   else{
      cpmerror("Angle(R, AngleUnit): unvalid AngleUnit");
   }
   normalize();
}

void Angle::setArc(R alpha)
{
   a_=alpha;
   normalize();
}

namespace{
void quantize(R& x, Z n)
{
   R y=x*n;
   y=cpmfloor(y+0.5);
   x=y/n;
}

}
Angle& Angle::operator +=(const R& s)
{
   a_+=s;
   normalize();
   return *this;
}

Angle& Angle::operator *=(const R& s)
{
   a_*=s;
   normalize();
   return *this;
}
```

```
Angle Angle::operator -(void)const
{
   Angle res;
   res.a_=-a_;
   res.normalize();
   return res;
}

Angle& Angle::operator +=(const Angle& s )
{
   a_+=(s.a_);
   normalize();
   return *this;
}

Angle& Angle::operator -=(const Angle& a1 )
{
   a_-=a1.a_;
   normalize();
   return *this;
}

Angle Angle::operator +(const Angle& a1)const
{
   return Angle(a_+a1.a_,RAD);
}

Angle Angle::operator -(const Angle& a1)const
{
   return Angle(a_-a1.a_,RAD);
}

R Angle::operator /(const Angle& a1)const
{
   R a1Inv=cpminv(a1.a_);
   return a_*a1Inv;
}

Angle Angle::con(void)const
{
   if (a_==cpmpi) return *this;
   else return Angle(-a_,RAD);
}

R Angle::cut(R c)const
{
   R lim=c*cpmdeg;
   R ar=a_+fullAngle;
   if (ar>lim) ar-=fullAngle;
   return ar;
```

```
}

bool Angle::prnOn(ostream& out)const
{
   return CpmRoot::write(a_,out);
}

bool Angle::scanFrom(istream& in)
{
   bool res=CpmRoot::read(a_,in);
   normalize();
   return res;
}

Angle CpmGeo::mean(const CpmArrays::V<Angle>& angles)
{
   Z i,n=angles.dim();
   Word loc("Angle CpmGeo::mean(const CpmArrays::V<Angle>& angles)");
   cpmassert(n>0,loc);
   C z,sum(0,0);
   for (i=1;i<=n;i++){
      z.polar(1.,angles[i].toR());
      sum+=z;
   }
   R r,phi;
   sum.toPolar(r,phi);
   return Angle(phi,RAD);
}

Angle CpmGeo::mean(const CpmArrays::V<Angle>& angles,
   const CpmArrays::V<R>& weights)
{
   Word loc("Angle CpmGeo::mean(...)");
   Z i,n=angles.dim(),n1=weights.dim();
   cpmassert(n>0,loc);
   cpmassert(n1>=n,loc);
   C z,sum(0,0);
   for (i=1;i<=n;i++){
      z.polar(1.,angles[i].toR());
      sum+=z*weights[i];
   }
   R r,phi;
   sum.toPolar(r,phi);
   return Angle(phi,RAD);
}

Z Angle::com(const Angle& a1)const
{
   if (a_<a1.a_) return 1;
   else if (a_>a1.a_) return -1;
```

```
   else return 0;
}

R Angle::dis(const Angle& phi)const
{
   Angle a1=*this - phi;
   Angle a2= phi - *this;
   R d1=cpmabs(a1.toR());
   R d2=cpmabs(a2.toR());
   return cpminf(d1,d2)*iFullAngle;
}

R Angle::toDeg(Z i)const // 'to degrees'
   {
      static R iDegree=R(180)/cpmpi;
      R b=a_*iDegree;
      if (i==1){
         if (b<0.) b+=360.;
      }
      return b;
   }

Word Angle::toWord(AngleUnit u, AngleRange r, AngleHex h, Z digits)const
{
   const R iDegree=R(180)/cpmpi;
   const R ihour=R(12)/cpmpi;
   R y=a_;
   if (r==POS && y<0) y+=fullAngle;
    Z flag=0;
   Word res, sep=":";
   const char* ff="%-#3.1f";
   const char* fff="%2d";
   if (digits==0)  ff="%-#2.0f";
   if (digits==1)  ff="%-#3.1f";
   if (digits==-1){ff="%-#3.1f"; flag=1;}
   if (digits==2)  ff="%-#4.2f";
   if (digits==3)  ff="%-#5.3f";
   if (digits==4)  ff="%-#6.4f";
   if (digits==5)  ff="%-#7.5f";
   if (digits==6)  ff="%-#8.6f";

   if (y<0){
      res="-";  // this is the first part of the result
                // further parts will be appended later
      y=-y;
   }
   else{
      res=" ";
   }
```

```
   if (u==CYCLE){
      y*=iFullAngle; // positive angle in full angles (turns)
   }
    else if (u==HOUR){
      y*=ihour; // positive angle in hours
   }
   else if (u==DEG){
      y*=iDegree; // positive angle in degrees
   }

   if (h==DEC){
      if (flag) quantize(y,2);
      res=res&CpmRoot::toWord(y,ff);
      return res;
   }

   Z ip=cpmtoz(y); // integer part (degrees or hours depending on u)
   res=res&CpmRoot::toWord(ip,fff);
   y-=ip;
   y*=60;  // y=minutes (time or angle depending on u)
   if (h==MIN){
      if (flag) quantize(y,2);
      res=res&sep&CpmRoot::toWord(y,ff);
      return res;
   }

   ip=cpmtoz(y); // integer part (minutes of time or angle)
   res=res&sep&CpmRoot::toWord(ip,fff);
   y-=ip;
   y*=60; // y=seconds (time or angle depending on u)
   if (h==SEC){
      if (flag) quantize(y,2);
      res=res&sep&CpmRoot::toWord(y,ff);
      return res;
   }
   return res;  // this should not be reached, for symmetry and for
      // avoiding a warning
}

Angle CpmGeo::RealToAngle(R x, AngleUnit u, AngleHex h)
{
   const R i60=1./60.;
   Z sign;
   Angle res;
   if (h==DEC){
      if (u==CYCLE) x*=fullAngle;
      if (u==HOUR) x*=hour;
      if (u==DEG) x*=cpmdeg;
      Angle::normalizeAngle(x);
      res.a_=x;
```

```
      return res;
   }
   if (x<0.){
      sign=-1; x=-x;
   }
   else{
      sign=1;
   } // now x is positive
   R ix=cpmfloor(x);
   R y=(x-ix)*100;
   if (h==MIN){
      x=ix+y*i60;
      if (u==CYCLE) x*=fullAngle;
      if (u==HOUR) x*=hour;
      if (u==DEG) x*=cpmdeg;
        x*=sign;
      Angle::normalizeAngle(x);
      res.a_=x;
      return res;
   }

   R iy=cpmfloor(y);
   R z=(y-iy)*100;
   if (h==SEC){
      x=ix+(iy+z*i60)*i60;
      if (u==CYCLE) x*=fullAngle;
      if (u==HOUR) x*=hour;
      if (u==DEG) x*=cpmdeg;
      x*=sign;
      Angle::normalizeAngle(x);
      res.a_=x;
      return res;
   }
   return res;
}

V<R> Angle::smoothAngleList(const V<Angle>& va)
{
   Z i,n=va.dim();
   V<R> res(n);
   for (i=1;i<=n;i++) res[i]=va[i].toR();
   for (i=2;i<=n;i++){
      R diff=res[i]-res[i-1];
      R dz=cpmabs(diff);
      if (dz<cpmpi) continue;
      R dp=cpmabs(diff+fullAngle);
      R dm=cpmabs(diff-fullAngle); //p,z,m stands for plus, zero, minus
      R dMin=cpminf(dp,dz,dm);
      if (dp==dMin) res[i]+=fullAngle;
      else if (dm==dMin) res[i]-=fullAngle;
```

```
      else ;
   }
   return res;
}

V<R> Angle::smoothList(const V<R>& vr)
{
   Z i,n=vr.dim();
   V<R> res=vr;
   for (i=2;i<=n;i++){
      R ri=res[i];
      R rim=res[i-1];
      R diff=ri-rim;
      R dz=cpmabs(diff);
      if (dz<cpmpi) continue;
      R dp=cpmabs(diff+fullAngle);
      R dm=cpmabs(diff-fullAngle); //p,z,m stands for plus, zero, minus
      R dMin=cpminf(dp,dz,dm);
      if (dp==dMin){
         res[i]+=fullAngle;
      }
      else if (dm==dMin){
         res[i]-=fullAngle;
      }
      else ;
   }
   return res;
}

void Angle::qun_(Z n)
{
   if (n<1) return; // do nothing for meaningless input
   R a=(a_+cpmpi)*iFullAngle*n; // ia a value between 0 and n
   a=cpmrnd(a);
   a*=(fullAngle/n);
   a_=a-cpmpi;
}
```

6 cpmbas.h

```
//? cpmbas.h
//? Status of work 2023-10-20.
//?
//?  ...


#ifndef CPM_BAS_H_
#define CPM_BAS_H_
/*
   Description: Collects header files from cpm0/include in one file
   in order to make working with basic C+- easier
*/
   #include <cpmtypes.h>
   #include <cpmfr.h>
   #include <cpmvr.h>
   #include <cpmsr.h>
   #include <cpmm.h>
   #include <cpmp.h>
   #include <cpmc.h>
   #include <cpmangle.h>
   #include <cpmgreg.h>
#endif
```

7 cpmbas.cpp

```
//? cpmbas.cpp
//? Status of work 2023-10-20.
//?
//?  ...


// see cpmbas.h

#include "cpmc.cpp"
#include "cpmangle.cpp"
#include "cpmv.cpp"
#include "cpmgreg.cpp"
#include "cpmsystem.cpp"
#include "cpmtypes.cpp"
#include "cpmuc.cpp"
#include "cpmzinterval.cpp"
#include "cpmnumbers.cpp"
#include "cpmword.cpp"
#include "cpmviewport.cpp"
```

8 cpmbasicinterfaces.h

```
//? cpmbasicinterfaces.h
//? Status of work 2023-10-20.
//?
//?  ...


#ifndef CPM_BASIC_INTERFACES_H_
#define CPM_BASIC_INTERFACES_H_
/*

   Description: see cpminterfaces.h

*/

// The following makes sure that no client class will use Type in a way
// that would need copy constructor and assignment.
// Should be placed in the private section

#include <cpmbasictypes.h>
   // for Z in CPM_ORDER

#define CPM_INVAR(TypeName)\
   Type& operator = (Type const&);\
   TypeName(Type const&);

// example for usage
/*
   template <class X, class Y>
   class SUMFUO: public ...{
      typedef SUMFUO<X,Y> Type;
      CPM_INVAR(SUMFUO)
   public:
      ...
   };
*/

// order related stuff

#define CPM_ORDER_PLAIN\
   bool operator == ( Type const& x)const;\
   bool operator != (Type const& x)const;\
   bool operator < (Type const& x)const;\
   bool operator > (Type const& x)const;\
   bool operator <= (Type const& x)const;\
   bool operator >= (Type const& x)const;

// consistent generation of all six order-related operators from a
```

```
// single member functions. Notice that it would be useless to make com
// virtual since any useful re-definition probably differs in the
// type of the second argument too.

// see cpmnumers.h for explanations to 'com'

#define CPM_ORDER\
   CpmRoot::Z com(Type const&)const;\
   bool equalTo(Type const& x)const{ return com(x)==0;}\
   bool priorTo(Type const& x)const{ return com(x)==1;}\
   bool operator == ( Type const& x)const\
      { return com(x)==0;}\
   bool operator != (Type const& x)const\
      { return com(x)!=0;}\
   bool operator < (Type const& x)const\
      { return com(x)>0;}\
   bool operator > (Type const& x)const\
      { return com(x)<0;}\
   bool operator <= (Type const& x)const\
      { return com(x)>=0;}\
   bool operator >= (Type const& x)const\
      { return com(x)<=0;}

#endif
```

9 cpmbasictypes.h

```
//? cpmbasictypes.h
//? Status of work 2023-10-20.
//?
//?  ...


#ifndef CPM_BASIC_TYPES_H_
#define CPM_BASIC_TYPES_H_
/*
   Description: Introduces mathematics style named
      aliases for basic integer number types.
*/
#include <cstddef> // for std::ptrdiff_t and std::size_t
#include <cpmdefinitions.h> // for CPM_FLOAT, CPM_LONG, CPM_QUAD,
   // and CPM_MP

namespace CpmRoot{
   // Basic number-related C+- types and functions.

// This defines what types are to be used to represent
// C+-'s notion of integer numbers. Floating point types
// will be treated in cpmnumbers.h

// Integer types L, N, Z.
// Remember that N and Z are standard names for the sets of
// natural numbers and integer numbers in mathematics.

typedef std::size_t N;
      // natural numbers
      // provides cyclic definition of addition, without overflow.
      // Also needed as the native index type of std::vector.
      // Type N will be used only in implementation code,
      // where it is important that addition is a cyclic operation
      // which never creates overflow. As type of function
      // arguments I don't use N. The reason is the following:
      // Function arguments for which Z or N would be appropriate types
      // occur frequently. Often, in cases in which N seemed the natural
      // choice when defining a function in the first place, it has to
      // be discovered that an extension to Z could be useful. So one
      // would be tempted so switch (or oscillate!) between using Z and N.
      // This is to be avoided!

#if defined(CPM_LONG) || defined(CPM_QUAD) || defined(CPM_MP)
   // take the largest possible data types
   typedef long int Z;
      // integer numbers
```

```
#else // take normal int and unsigned
   typedef int Z;
      // integer numbers
#endif

   typedef unsigned char L;
      // 'L' for 'letter' represents a byte which after 'integral
      // promotion' is a value between 0 and 255.
      // Mainly needed for storing pixel values in image matrices.
      // Gets written to files as a number \in {0,1,...,255}

   inline N toN(Z const& i){ return static_cast<N>(i);}



   N operator""_N(unsigned long long int);
   Z operator""_Z(unsigned long long int);

} // CpmRoot


#endif // CPM_BASIC_TYPES_H_
```

10 cpmc.h

```
//? cpmc.h
//? Status of work 2023-10-20.
//?
//?  ...


#ifndef CPM_C_H_
#define CPM_C_H_
/*
   Description: Declaration of a class of complex numbers
       using the system provided type <complex> for implementation was
       tried. But this class has only very limited ammount of
       functions and my component access via [] can't be implemented
       here since the components are private.
       Division by zero will write a message, return 0 and continue.
       Test_c<C> d(10,1,1) gave a result which ~10-7 (including read
       write difference which is OK.
*/

#include <cpmword.h>
#include <complex>
   // connection with std complex numbers is useful when interfacing
   // with the Eigen library.

namespace CpmRoot{

class C;
// These are declarations of functions in scope CpmRoot
// most of them have a friend declaration within class C
// which enables a more efficient implementation.
R arg(C const& z);
C log10(C const& z);
C pow(C const& z, Z const& n);
C pow(C const& z, C const& n);
C sin(C const& z);
C cos(C const& z);
C tan(C const& z);
C cot(C const& z);
C sinh(C const& z);
C cosh(C const& z);
C tanh(C const& z);
C coth(C const& z);
C arcsin(C const& z);
C arccos(C const& z);
C arctan(C const& z);
C arccot(C const& z);
C arcsinh(C const& z);
```

```
C arccosh(C const& z);
C arctanh(C const& z);
C arccoth(C const& z);
C operator /(C const& z, R const& s);
bool isVal(C const& z);

class C { // complex numbers

   R re, im;
   typedef C Type;

public:

//// general infrastructure, in part with inline implementation
// This is essentially the general interface of templates Vr<> and Fr<>
// appart from the fact that here no function is virtual since we don't
// need to derive from this 'ultimate' class.

// order and comparison
   CPM_ORDER

// I/O operations
   CPM_IO

// test devices
   C ran(Z j=0)const;
// value of res.re is in the open interval (-|re|,|re|)
// value of res.im is in the open interval (-|im|,|im|)

   R dis(C const& y)const;
   C test(Z)const;
// descriptors
   Word nameOf(void)const{ return Word("C");}

   Word toWord()const;
      //: to word
      // for a nice printable representation

   Z hash(void) const;
      //: hash value

   C net(Z i=0)const{ if (i==1) return C(1_R,0_R); else return C();}
      //: neutrals
      // net(0): neutral element of addition,
      // net(1): neutral element of multiplication

   C neg()const{ return C(-re,-im);}
      //: negative

   C inv(void)const;
```

```
      //: inversion
      // Returns C(0,0) upon division by zero, but creates
      // warnings on cpmcerr if this happens. The total
      // number of warnings that can arrise in this way is limited
      // so that this file can't grow too much by this mechanism.

   C operator !(void)const{ return inv();}

// complex conjugation
   C con()const{ return C(re,-im);}
   void con_(){im=-im;}

   C operator~(void)const{ return C(re,-im);}
   //: ~
   // instead of z.con() we may write ~z

   C operator|(C const& z)const{ return con()*z;}
   //: (|) scalar product
   // The combination ~z1*z2 can also be written as (z1|z2)

// more specific topics

   static C I;  // 'imaginary unit'
   static C one; // number one = C(1,0)
   static C Ibar; // conjugate of I = - I = C(0,-1)

   C():re(0_R),im(0_R){}

   explicit C(R const& u, R const& v=0. ):re(u),im(v){}
      // constructor from real and imaginary part
      // no automatic conversion from R to C

   explicit C(std::complex<R> const& z):re(z.real()),im(z.imag()){}
      // construction from std::complex

   C(C const& z):re(z.re),im(z.im){}
      // standard construction operator

   C(R r, R phi, Word w):
   re(r*cpmcos(phi)),im(r*cpmsin(phi)){}
      // constructor from polar coordinates
      // In calling the constructor it is helpful to
      // use w="polar"

   R real(void)const{ return re;}

   R imag(void)const{ return im;}

   std::complex<R> std()const{ return std::complex<R>{re,im};}
```

```
  C& operator=(C const& c){ re=c.re; im=c.im; return *this;}
     // standard assignment operator

  C& operator=(std::complex<R> const& c)
  { re=c.real(); im=c.imag(); return *this;}

  Z dim(void)const{ return 2;} // dimension as a real linear space
     //: dimension

  R arg(void)const{ return cpmarg(re,im);}
     //: argument
     // The value is in the interval (-Pi,Pi] independent
     // of the convention followed by the system's ::atan2 function

//  R abs(bool careful=true)const
// Warning: not clear whether this works for choices of
// R different from float, double, long double.
R abs(bool careful=true)const
  { return  careful ? std::abs(std::complex<R>(re,im)) :
     cpmsqrt(re*re+im*im);}
     //: rho
     // common symbol for the polar radius

  R absFast(void)const
  { return cpmsqrt(re*re+im*im);}
     //: rho fast
     // No provision against overflow by sqaring potentially large
     // numbers.

  R absSqr(void)const{ return re*re+im*im;}
     //: absolute (value) squared

  R abs2(void)const{ return re*re+im*im;}
     //: absolute (value) squared

  bool isVal(void)const{ return cpmiva(re)&&cpmiva(im);}
     //: is valid

  C scaleTo(R const& r)const;
     // returns a complex number of absolute value |r| and the direction
     // of *this for positive r and antidirection for negative r.

  C timesI(void)const{ return C(-im,re); }

  C dividedByI(void)const{ return C(im,-re);}

  C dividedBy2I(void)const{ return C(im*0.5_R,re*(-0.5_R));}

  void pol_(R r, R phi);
     //: polar
```

```
      // changes *this into r*exp(i*phi)
      // 'modern' indication of mutating nature of the function

   static C pol(R r, R phi){C z; z.pol_(r,phi); return z;}
      //: polar
      // construction from polar coordinates as a static
      // function instead of a constructor

   void pol_1_2(R& r, R& phi)const{ r=abs(); phi=arg();}
      //: polar
      // 'modern' indication of reference nature of first
      // and second argument

// heritage versions of functions related to polar coordinates
   void polar(R const& r, R const& phi){ pol_(r,phi);}
      // changes *this into r*exp(i*phi)
      // not conforming to the C+- naming convention

   void toPolar(R& r, R& phi)const;
      // after call r and phi have the values radius() and arg()

// end of heritage versions of functions related to polar coordinates

   R nor_(R r=1_R)
      //: normalize
   {
      R a=abs();
      if(a!=0_R){R fac=r/a;re*=fac;im*=fac;}
      else{re=r;}
      return a;
   }
      // turns *this into a C of length r and returns the norm of
      // the original in order not to loose information. If *this was
      // zero, it will be turned into  a 'standard C'
      // of length r.

   C sqrt()const;
   C sqr()const;
   C exp()const;
   C ln()const;

   C pow(C const& p)const;
   C sin()const;
   C cos()const;
   C tan()const;
   C cot()const;
   C sinh()const;
   C cosh()const;
   C tanh()const;
   C coth()const;
```

```
   C arcsin()const;
   C arccos()const;
   C arctan()const;
   C arccot()const;
   C arcsinh()const;
   C arccosh()const;
   C arctanh()const;
   C arccoth()const; // these added 2005-08-23

   friend R arg(C const& z);
   friend R rho(C const& z);
   static C expi(R phi){ return C(cpmcos(phi),cpmsin(phi));}
      // t instead of phi would also be suggestive
   friend C log10(C const& z);
   friend C pow(C const& z, const Z& n);
   friend C pow(C const& z, C const& n);
   friend C sin(C const& z);
   friend C cos(C const& z);
   friend C tan(C const& z);
   friend C cot(C const& z);
   friend C sinh(C const& z);
   friend C cosh(C const& z);
   friend C tanh(C const& z);
   friend C coth(C const& z);
   friend C arcsin(C const& z);
   friend C arccos(C const& z);
   friend C arctan(C const& z);
   friend C arccot(C const& z);
   friend C arcsinh(C const& z);
   friend C arccosh(C const& z);
   friend C arctanh(C const& z);
   friend C arccoth(C const& z);

// components for uniformity with arrays:  z.re=z[1], z.im=z[2]
// no exceptions !!!

   R& operator[](int i){ return i==2 ? im : re;}

   R const& operator[](int i)const{ return i==2 ? im : re;}

// mutating arithmetics
// There are a few more operations to be defined (such as C+R)
// which are not prepared in the interfaces; so everything
// is done explicitely
   C& operator +=(C const& x){re+=x.re; im+=x.im; return *this;}
   C& operator +=(R const& s){re+=s; return *this;}

   C& operator -=(C const& x){re-=x.re; im-=x.im; return *this;}
   C& operator -=(R const& s){re-=s; return *this;}
```

```
   C& operator *=(C const& x)
     {R re0=re; re=re*x.re-im*x.im; im=im*x.re+re0*x.im; return *this;}

   C& operator /=(C const& x){ return operator *=(x.inv());}

   C& operator *=(R const& s){re*=s; im*=s; return *this;}
   C& operator /=(R const& s);

// generating arithmetics
   C operator -(void)const{ return C(-re,-im);}
   C operator +(C const& z)const
      { return C(re+z.re,im+z.im);}
   friend C operator +(R const& s, C const& z2);
   C operator +(R const& s)const
      { return C(re+s,im);}
   C operator -(C const& z)const
      { return C(re-z.re,im-z.im);}
   friend C operator -(R const& s, C const& z2);
   C operator -(R const& s)const
      { return C(re-s,im);}
   C operator *(C const& z)const
      { return C(re*z.re-im*z.im,im*z.re+re*z.im);}
   friend C operator *(R const& s, C const& z2);
   C operator *(R const& s)const
       { return C(s*re,s*im);}
   C operator /(C const& z)const
      { return (*this)*z.inv();}
   friend C operator /(C const& z, R const& s);
};

// These are definitions of functions in scope CpmRoot, declared friend
// in class C.
inline C operator +(R const& s, C const& z2)
      { return C(z2.re+s,z2.im);}
inline C operator -(R const& s, C const& z2)
      { return C(s-z2.re,-z2.im);}
inline C operator *(R const& s, C const& z2)
       { return C(s*z2.re,s*z2.im);}
// The following functions need not to be friends of C.
// their argument in ways that need them to be declared friends of C.
inline bool isVal(C const& x){ return x.isVal();}
inline C ran(C const& x, Z j){ return x.ran(j);}
inline R dis(C const& x1, C const& x2){ return x1.dis(x2);}
inline C test(C const& x, Z cpl){ return x.test(cpl);}
inline Z hash(C const& x){ return x.hash();}
inline R abs(C const& x){ return x.abs();}
inline R rho(C const& x){ return x.abs();}
inline C net(C const& x, Z i=0){ return x.net(i);}
inline C inv(C const& x){ return x.inv();}
inline C con(C const& x){ return x.con();}
```

```
inline C sqrt(C const& z){ return z.sqrt();}
inline R Re(C const& z){return z.real();}
inline R Im(C const& z){return z.imag();}
inline C exp(C const& z){return z.exp();}
inline C expI(R phi){ return C::expi(phi);}
      // even for Angle alpha, we can write directly expI(alpha) since
      // CpmGeo::Angle gets automatically converted to an R-valued arc
      // So, instead of saying
      // C z; z.polar(r,phi);
      // we may, much nicer, say:
      // C z=r*expI(phi);
inline C ln(C const& z){ return z.ln();}
inline C log(C const& z){ return z.ln();}
inline C asin(C const& z){ return CpmRoot::arcsin(z);}
inline C acos(C const& z){ return CpmRoot::arccos(z);}
inline C atan(C const& z){ return CpmRoot::arctan(z);}
inline C acot(C const& z){ return CpmRoot::arccot(z);}
inline C asinh(C const& z){ return CpmRoot::arcsinh(z);}
inline C acosh(C const& z){ return CpmRoot::arccosh(z);}
inline C atanh(C const& z){ return CpmRoot::arctanh(z);}
inline C acoth(C const& z){ return CpmRoot::arccoth(z);}

} // namespace
#endif
```

11 cpmc.cpp

```
//? cpmc.cpp
//? Status of work 2023-10-20.
//?
//?  ...


#include <cpmc.h>
#include <cpmtypes.h>

using namespace CpmRoot;
using namespace CpmSystem;

C CpmRoot::C::I=C(0_R,1_R);
C CpmRoot::C::Ibar=C(0_R,-1_R);
C CpmRoot::C::one=C(1_R,0_R);

Word C::toWord()const
      // for a nice printable representation
{
   Word sep_i=" "; // initial
   Word sep_f=" "; // final separator
   Word res;
   if (im==0.) res=cpmwrite(re);
   else if (re==0.){
      if (im>0.) res="i*"&cpmwrite(im);
      else res="-i*"&cpmwrite(-im);
   }
   else if (im>0.){
      res=cpmwrite(re)&"+i*"&cpmwrite(im);
   }
   else{
      res=cpmwrite(re)&"-i*"&cpmwrite(-im);
   }
   return sep_i&res&sep_f;
}

bool C::prnOn(ostream& str )const
{
   //if (!CpmRoot::writeTitle("C",str)) return false;
   cpmwat;
   cpmp(re);
   cpmp(im);
   return true;
}

bool C::scanFrom(istream& str )
{
```

```
   cpms(re);
   cpms(im);
   return true;
}

// inversions

C C::inv(void)const
// same logic as inv(R) for error handling
// see Press et al. p. 177 (5.4.5) for the algorithm
{
   static const Z maxMes=100_Z;
   static Z mes=1_Z;
   R c_=(re>=0_R ? re : -re);
   R d_=(im>=0_R ? im : -im);
   if (c_>=d_){
      if (c_==0_R) goto DIVBYZERO;
      R y=im/re; R z=1_R/(re+im*y); return C(z,-y*z);
   }
   else{
      if (d_==0_R) goto DIVBYZERO;
      R y=re/im; R z=1_R/(re*y+im); return C(y*z,-z);
   }

DIVBYZERO:

      if (mes==maxMes){
         mes++; // needed !
         Message::message(
            "C::inv(): argument is (0,0) ... messages discontinued");
      }
      if (mes<maxMes){
         mes++;
         Message::warning(
            "C::inv(): argument is (0,0);  (0,0) returned");
      }
      return C();
}

C& C::operator /=(R const& s)
{
   R si=CpmRoot::inv(s);
   re*=si;
   im*=si;
   return *this;
}

C CpmRoot::operator /(C const& z, R const& s)
{
   R si=cpminv(s);
```

```
   return z*si;
}

// Functions related to polar representation

R CpmRoot::arg(C const& x)
{
   return x.arg();
}

void C::pol_(R r, R phi)
{
   re=r*cpmcos(phi);
   im=r*cpmsin(phi);
}

void C::toPolar(R& r, R& phi)const
{
   r=abs();
   phi=arg();
}

// Square

C C::sqr(void)const
{
   C res(*this);
   return res*res;
}

// Square root

C C::sqrt(void)const
{
   R r=abs(), phi=arg();
   r=cpmsqrt(r);
   phi*=0.5;
   return C(r*cpmcos(phi),r*cpmsin(phi));
}

// Elementary transcendental functions. We need the following real
// functions to be provided by the compiler:

//      exp, log, sin, cos, sqrt, atan. If complex numbers Ca are to be
//      defined based on a class Ra of numbers of arbitrary precission,
//      we have to implement these functions also for Ra.

C C::exp()const
{
   R r=cpmexp(re);
```

```
   return C(r*cpmcos(im),r*cpmsin(im));
}

C C::ln()const
{
   R r=abs(), phi=arg();
   r=cpmlog(r);  // library function
   return C(r,phi);
}

C CpmRoot::log10(C const& z)
// logarithm to basis 10
{
   static const R log10_=1./cpmlog(10.);
   return log10_*ln(z);
}

C CpmRoot::pow(C const& z, const Z& n)
{
   C res(1.,0);
   if (n==0) return res;

   C z_; Z n_;
   if (n>0){
      z_=z;
      n_=n;
   }
   else{
      z_=!z;
      n_=-n;
    }
   for (Z k=1; k<=n_; k++) res*=z_;
   return res;
}

C CpmRoot::pow(C const& z, C const& w)
{
   return exp(w*ln(z));
}

C CpmRoot::sin(C const& z)
{
   return (exp(z.timesI())-exp(z.dividedByI())).dividedBy2I();
}

C CpmRoot::cos(C const& z)
{
   return (exp(z.timesI())+exp(z.dividedByI()))*0.5;
}
```

```
C CpmRoot::tan(C const& z){ return sin(z)/cos(z);}

C CpmRoot::cot(C const& z){ return cos(z)/sin(z);}

C CpmRoot::sinh(C const& z){ return 0.5*(exp(z)-exp(-z));}

C CpmRoot::cosh(C const& z){ return 0.5*(exp(z)+exp(-z));}

C CpmRoot::tanh(C const& z){ return sinh(z)/cosh(z);}

C CpmRoot::coth(C const& z){ return cosh(z)/sinh(z);}

C CpmRoot::arcsinh(C const& z)
   // Gradshteyn Ryzhik 1.622 5.
{ return ln(z+(z*z+1).sqrt());}

C CpmRoot::arccosh(C const& z)
   // Gradshteyn Ryzhik 1.622 6.
{ return ln(z+sqrt(z*z-1));}

C CpmRoot::arctanh(C const& z)
   // Gradshteyn Ryzhik 1.622 7.
{ return ln((z+C(1))/(C(1)-z))*0.5;}

C CpmRoot::arccoth(C const& z)
   // Gradshteyn Ryzhik 1.622 8.
{ return ln((z+C(1))/(z-C(1)))*0.5;}

C CpmRoot::arcsin(C const& z)
   // Gradshteyn Ryzhik 1.622 1.
{ return (arcsinh(z.timesI())).dividedByI();}

C CpmRoot::arccos(C const& z)
   // Gradshteyn Ryzhik 1.622 2.
{ return (arccosh(z)).dividedByI();}

C CpmRoot::arctan(C const& z)
   // Gradshteyn Ryzhik 1.622 3.
{ return (arctanh(z.timesI())).dividedByI();}

C CpmRoot::arccot(C const& z)
   // Gradshteyn Ryzhik 1.622 4.
{ return (arccoth(z.timesI())).timesI();}

C C::pow(C const& p)const{ return CpmRoot::pow(*this,p);}
C C::sin()const{ return CpmRoot::sin(*this);}
C C::cos()const{ return CpmRoot::cos(*this);}
C C::tan()const{ return CpmRoot::tan(*this);}
C C::cot()const{ return CpmRoot::cot(*this);}
C C::sinh()const{ return CpmRoot::sinh(*this);}
```

```
C C::cosh()const{ return CpmRoot::cosh(*this);}
C C::tanh()const{ return CpmRoot::tanh(*this);}
C C::coth()const{ return CpmRoot::coth(*this);}
C C::arcsin()const{ return CpmRoot::arcsin(*this);}
C C::arccos()const{ return CpmRoot::arccos(*this);}
C C::arctan()const{ return CpmRoot::arctan(*this);}
C C::arccot()const{ return CpmRoot::arccot(*this);}
C C::arcsinh()const{ return CpmRoot::arcsinh(*this);}
C C::arccosh()const{ return CpmRoot::arccosh(*this);}
C C::arctanh()const{ return CpmRoot::arctanh(*this);}
C C::arccoth()const{ return CpmRoot::arccoth(*this);}
   // these added 2005-08-23

// Indicators

/*************** utility functions ****************************/

C C::test(Z cpl)const
{
   const R red=0.5;
      // complex functions like exp() should be included in tests and
      // don't behave uncritical for large arguments
   R a=0.;
   R ar=CpmRoot::test(a,cpl);
   R ai=ar*red;
   return C(ar,ai);
}

C C::ran(Z j)const
{
   R a,b;
   if (j==0){
      a=CpmRoot::ran(re,0);
      b=CpmRoot::ran(im,0);
   }
   else{
      Z j2=j*2;
      a=CpmRoot::ran(re,j2);
      b=CpmRoot::ran(im,j2+1);
   }
   return C(a,b);
}

R C::dis(C const& z)const
{
   return CpmRoot::disDefFun(abs(),z.abs(),(*this-z).abs());
}

Z C::com(C const& s)const
{
```

```
   if (re<s.re) return 1;
   if (re>s.re) return -1;
   if (im<s.im) return 1;
   if (im>s.im) return -1;
   return 0;
}

Z C::hash()const
{
   Z ix=CpmRoot::hash(re);
   Z iy=CpmRoot::hash(im);

   return ix^iy;
}

C C::scaleTo(R const& r)const
      // returns a complex number of absolute value |r| and the direction
      // of *this for positive r and antidirection for negative r.
{
   R rOrig=abs();
   if (rOrig==0.) return C(r,0);
   C res=*this;
   return res*(r/rOrig);
}
```

12 cpmcompdef.h

```
//? cpmcompdef.h
//? Status of work 2023-10-20.
//?
//?  ...


// no double include guard needed since this occurs only as
// '#include <cpmcompdef.h>' in cpmdefinitions.h and the letter file
// is guarded against double inclusion.
/*************************************************************************
   cpmcompdef.h
   Description: In this file one defines how compiler options of the group
   -D influence the compilation directives defined in cpmdefinition.h
*************************************************************************/
// ways to define R by compiler options of group -D
// one line in a makefile could for instance be
// defines = -D_CONSOLE -DCPM_RMP_C
// compilation than would result in an executable which uses a
// representation with 64 decimal places (actually the computed equivalent
// of binary places) for each number of type R.
// Since C+- is made compatible with C++20 it only supports a single way
// to implement multiple precision arithmetics: based on
// boost::multiprecision . Unfortunately this is not fully in harmony with
// using Eigen library. It only works if compilation is done under
// the -fpermissive option. If one likes not be overoaded with warnings one
// also needs the -w option. Also one can't use expression templates in Eigen
// (the flag which regulates this is set in the C+- code).

#if defined(CPM_RFLOAT)  // R float
   #undef CPM_FLOAT
   #undef CPM_DOUBLE
   #undef CPM_LONG
   #undef CPM_QUAD
   #undef CPM_MP
   #define CPM_FLOAT

#elif defined(CPM_R)  // R double
   #undef CPM_FLOAT
   #undef CPM_DOUBLE
   #undef CPM_LONG
   #undef CPM_QUAD
   #undef CPM_MP
   #define CPM_DOUBLE

#elif defined(CPM_RDOUBLE)  // R double (same as above, for uniformity)
   #undef CPM_FLOAT
   #undef CPM_DOUBLE
```

```
   #undef CPM_LONG
   #undef CPM_QUAD
   #undef CPM_MP
   #define CPM_DOUBLE

#elif defined(CPM_RLONG)  // R long double
   #undef CPM_FLOAT
   #undef CPM_DOUBLE
   #undef CPM_LONG
   #undef CPM_QUAD
   #undef CPM_MP
   #define CPM_LONG

#elif defined(CPM_RQUAD)  // R quad
   #undef CPM_FLOAT
   #undef CPM_DOUBLE
   #undef CPM_LONG
   #undef CPM_QUAD
   #undef CPM_MP
   #define CPM_QUAD

#elif defined(CPM_RMP) // R multiprecision, here comparable with double
   #undef CPM_FLOAT
   #undef CPM_DOUBLE
   #undef CPM_LONG
   #undef CPM_QUAD
   #undef CPM_MP
   #define CPM_MP 16

#elif defined(CPM_RMP_A)
   #undef CPM_FLOAT
   #undef CPM_DOUBLE
   #undef CPM_LONG
   #undef CPM_QUAD
   #undef CPM_MP
   #define CPM_MP 24

#elif defined(CPM_RMP_B) // R multiprecision, here comparable with quad
   #undef CPM_FLOAT
   #undef CPM_DOUBLE
   #undef CPM_LONG
   #undef CPM_QUAD
   #undef CPM_MP
   #define CPM_MP 32

#elif defined(CPM_RMP_C)
   #undef CPM_FLOAT
   #undef CPM_DOUBLE
   #undef CPM_LONG
   #undef CPM_QUAD
```

```
   #undef CPM_MP
   #define CPM_MP 64

#elif defined(CPM_RMP_D)
   #undef CPM_FLOAT
   #undef CPM_DOUBLE
   #undef CPM_LONG
   #undef CPM_QUAD
   #undef CPM_MP
   #define CPM_MP 128

#elif defined(CPM_RMP_E)
   #undef CPM_FLOAT
   #undef CPM_DOUBLE
   #undef CPM_LONG
   #undef CPM_QUAD
   #undef CPM_MP
   #define CPM_MP 256

#endif
```

13 cpmdefinitions.h

```
//? cpmdefinitions.h
//? Status of work 2023-10-20.
//?
//?  ...


#ifndef CPM_DEFINITIONS_H_
#define CPM_DEFINITIONS_H_
/***************************************************************************
   cpmdefinitions.h
   Description: In this file one can place defines that are valid in all
   translation units working with Cpm classes
   Cpm = Classes for Physics and Mathematics || C+- (C plus minus)
         -           -           -                   - -    -
***************************************************************************/

//#define CPM_LONG

#define CPM_NAMEOF

#define CPM_RANGE_CHECK

#define CPM_USECOUNT

#define CPM_TEMPLATE_TESTS
   // having this defined increases code size
   // it helps to detect inconsistencies in interfaces of template
   // classes
   // should be disabled for release compilation
//#define CPM_MP 32
#include <cpmcompdef.h>
#endif
```

14 cpmf.h

```
//? cpmf.h
//? Status of work 2023-10-20.
//?
//?  ...


#ifndef CPM_F_H_
#define CPM_F_H_
/*
   Purpose: see cpmfl.h

*/
#include <cpmv.h>
   // includes <cpmfl.h>, where the essential template F<X,Y>
   // is defined

namespace CpmFunctions{

   using CpmArrays::V;
   using CpmArrays::IvZ;

   template <class X, class Y>
   V<Y> apply(F<X,Y>  const& f, V<X>  const& xs)
      // action on sequences: applying f to xs gives the return value of
      // the present function
   {
      IvZ d=xs.dom(); // general indexing
      V<Y> res(d);
      for (Z i=xs.b();i<=xs.e();i++) res[i]=f(xs[i]);
      return res;
   }

namespace aux{
#ifdef CPM_Fn
template
   <class X, class P1, class P2, class P3, class P4, class P5, class Y>
class Par5FncObj : public FncObj<X,Y>{

   const P1 p1;
   const P2 p2;
   const P3 p3;
   const P4 p4;
   const P5 p5;
   Y (* const fp)
      (X const&, P1 const&, P2 const&, P3 const&, P4 const&, P5 const&);
public:
   Y operator()(X const& x)const
```

```
      { return (*fp)(x,p1,p2,p3,p4,p5);}
   Par5FncObj(
      Y (*g)(X const&,P1 const&,P2 const&,P3 const&,P4 const&,P5 const&),
      P1 const& q1,P2 const& q2,P3 const& q3,P4 const& q4, P5 const& q5):
      p1(q1),p2(q2),p3(q3),p4(q4),p5(q5),fp(g){}
};

template
   <class X, class P1, class P2, class P3, class P4,
   class P5, class P6, class Y>
class Par6FncObj : public FncObj<X,Y>{

   const P1 p1;
   const P2 p2;
   const P3 p3;
   const P4 p4;
   const P5 p5;
   const P6 p6;
   Y (* const fp)(X const&, P1 const&, P2 const&, P3 const&,
      P4 const&, P5 const&, P6 const&);
public:
   Y operator()(X const& x)const
      { return (*fp)(x,p1,p2,p3,p4,p5,p6);}
   Par6FncObj(
      Y (*g)(X const&,P1 const&,P2 const&,P3 const&,P4 const&,
         P5 const&, P6 const&),
         P1 const& q1,P2 const& q2,P3 const& q3,
         P4 const& q4, P5 const& q5, P6 const& q6):
         p1(q1),p2(q2),p3(q3),p4(q4),p5(q5),p6(q6),fp(g){}
};

#endif // CPM_Fn

} // aux

// For F5 and F6, we do not introduce the corresponding F5_1,...F6_6
// which would enable using the parameters as function arguments. Having
// this capability for up to four parameters is sufficient. See cpmfl.h for // // this matter.

#ifdef CPM_Fn
////////////////////// class F5<> ////////////////////////////////////

template <class X, class P1, class P2, class P3, class P4,
   class P5, class Y>
class F5{ // functions with five parameters
    const P1 p1;
    const P2 p2;
    const P3 p3;
    const P4 p4;
    const P5 p5;
```

```
    typedef F5<X,P1,P2,P3,P4,P5,Y> Type;
    CPM_INVAR(F5)
public:
    F5( P1 const& p1_,P2 const& p2_,P3 const& p3_,P4 const& p4_,
       P5 const& p5_):
    p1(p1_),p2(p2_),p3(p3_),p4(p4_),p5(p5_){}
    F<X,Y> operator()(Y (*f)(X const&,P1 const&,P2 const&,
       P3 const&,P4 const&,P5 const&))const;
};

template <class X, class P1, class P2, class P3, class P4,
   class P5,  class Y>

F<X,Y> F5<X,P1,P2,P3,P4,P5,Y>::operator()(Y (*f)(X const&,P1 const&,
   P2 const&, P3 const&,P4 const&,P5 const&))const
{
   using namespace aux;
   return
   F<X,Y>(new Par5FncObj<X,P1,P2,P3,P4,P5,Y>(f,p1,p2,p3,p4,p5));
}

//////////////////////// class F6<> ////////////////////////////////////

template <class X, class P1, class P2, class P3, class P4,
   class P5, class P6, class Y>
class F6{ // functions with six parameters
    const P1 p1;
    const P2 p2;
    const P3 p3;
    const P4 p4;
    const P5 p5;
    const P6 p6;
    typedef F6<X,P1,P2,P3,P4,P5,P6,Y> Type;
    CPM_INVAR(F6)
public:
    F6( P1 const& p1_,P2 const& p2_,P3 const& p3_,P4 const& p4_,
       P5 const& p5_, P6 const& p6_):
    p1(p1_),p2(p2_),p3(p3_),p4(p4_),p5(p5_),p6(p6_){}
    F<X,Y> operator()(Y (*f)(X const&,P1 const&,P2 const&,
       P3 const&,P4 const&,P5 const&, P6 const&))const;
};

template <class X, class P1, class P2, class P3, class P4,
   class P5, class P6, class Y>

F<X,Y> F6<X,P1,P2,P3,P4,P5,P6,Y>::operator()(Y (*f)(X const&,P1 const&,
   P2 const&, P3 const&,P4 const&,P5 const&, P6 const&))const
{
   using namespace aux;
   return
```

```
   F<X,Y>(new Par6FncObj<X,P1,P2,P3,P4,P5,P6,Y>
      (f,p1,p2,p3,p4,p5,p6));
}

#endif // CPM_Fn

namespace aux{

template <class X1, class X2, class Y1, class Y2>
class cart: public FncObj<cpmX2<X1,X2>,cpmX2<Y1,Y2> >{ // CART stands for
    // Cartesian

   const F<X1,Y1> f1;
   const F<X2,Y2> f2;

public:
   cpmX2<Y1,Y2> operator()(const cpmX2<X1,X2>& x)const
   { return cpmX2<Y1,Y2>(f1(x.first),f2(x.second));}

   cart(const F<X1,Y1>& f1_,const F<X2,Y2>& f2_):f1(f1_),f2(f2_){}
};

// pairing of functions which are defined on the same class

////////////////////////// class cart1<> ////////////////////////////////

template <class X, class Y1, class Y2>
class cart1: public FncObj<X,cpmX2<Y1,Y2> >{

   const F<X,Y1> f1;
   const F<X,Y2> f2;
public:
   cpmX2<Y1,Y2> operator()(X const& x)const
   { return cpmX2<Y1,Y2>(f1(x),f2(x));}

   cart1(const F<X,Y1>& f1_,const F<X,Y2>& f2_):f1(f1_),f2(f2_){}
};

} // aux

////////////////////// function pairOf<> ////////////////////////////////

template <class X1, class X2, class Y1, class Y2>
F<cpmX2<X1,X2>,cpmX2<Y1,Y2> > pairOf(const F<X1,Y1>& f1,
   const F<X2,Y2>& f2)
   // Cartesian product of functions:
   // cpmX2(f1,f2)(x1,x2)=(f1(x1),f2(x2)).
   // name changed from pair to pairOf to avoid clash with std name
   // (99-5-31)
{
```

```
   using namespace aux;
   return F<cpmX2<X1,X2>,cpmX2<Y1,Y2> >(new cart<X1,X2,Y1,Y2>(f1,f2));
}

////////////////////////// operator&&<> ////////////////////////////////

template <class X, class Y1, class Y2>
F<X,cpmX2<Y1,Y2> > operator &&(const F<X,Y1>& f1,const F<X,Y2>& f2)
   // Cartesian product of functions which are defined on the same class.
    // (f1&&f2)(x)=(f1(x),f2(x)).
    // Elegant way for flexible definition of binary operators by
    // concatenation
    // with a mapping cpmX2<Y1,Y2> ---> Y3
{
   using namespace aux;
   return F<X,cpmX2<Y1,Y2> >(new cart1<X,Y1,Y2>(f1,f2));
}

// 'changing the number of variables from 2 to 1'

// Bind1 and Bind2 as classes of the same design as FuncPar.
// Assume:
// class X1;
// class X2;
// class Y;
// F<cpmX2<X1,X2>,Y> f(...);
// X1 x1=...;
// F<X2,Y> f1=Bind1<X1,X2,Y>(x1)(f);
// X2 x2=...;
// Y ya=f(cpmX2<X1,X2>(x1,x2));
// Y yb=f1(x2);
// then ya==yb

namespace aux{

template <class X1, class X2, class Y>
class bind1: public FncObj<X2,Y>{
   // bind1 stands for bind first variable
   const X1 x1;
   const F<cpmX2<X1,X2>,Y> f;
public:
   Y operator()(const X2& x2)const{ return f( cpmX2<X1,X2>(x1,x2));}
   bind1( const F<cpmX2<X1,X2>,Y> & f_, const X1& x1_):f(f_),x1(x1_){}
};

template <class X1, class X2, class Y>
class bind2: public FncObj<X1,Y>{
   // bind2 stands for bind second variable
   const X2 x2;
   const F< cpmX2<X1,X2>,Y>  f;
```

```
public:
   Y operator()(const X1& x1)const{ return f( cpmX2<X1,X2>(x1,x2));}
   bind2( const F< cpmX2<X1,X2>,Y>& f_, const X2& x2_):f(f_),x2(x2_){}
};

} // aux

////////////////////////// class Bind1<> ///////////////////////////////

template <class X1, class X2, class Y>
class Bind1 { // binding the first parameter
    const X1 x1;
public:
    Bind1(const X1& x1_):x1(x1_){}
    F<X2,Y> operator()(const F<cpmX2<X1,X2>,Y>& f)const
    { return F<X2,Y>(new aux::bind1<X1,X2,Y>(f,x1));}
};

////////////////////////// class Bind2<> ///////////////////////////////

template <class X1, class X2, class Y>
class Bind2 { // binding the second parameter
    const X2 x2;
public:
    Bind2(const X2& x2_):x2(x2_){}
    F<X1,Y> operator()(const F<cpmX2<X1,X2>,Y>& f)const
    { return F<X1,Y>(new aux::bind2<X1,X2,Y>(f,x2));}
};

// range and domain conversion

namespace aux{

template <class X1, class X2, class Y>
class convertDomain: public FncObj<X2,Y>{
   // means to transform a F<X,Y> in a F<X',Y> if there is an
   // automatic conversion from X to X'

   const F<X1,Y> f;
public:
   Y operator()(const X2& x)const{ return f(x);}
   convertDomain( const F<X1,Y>& f_):f(f_){}
};

template <class X, class Y1, class Y2>
class convertRange: public FncObj<X,Y2>{
   // means to transform a F<X,Y> in a F<X,Y'> if there is an
   // automatic conversion from Y to Y'

   const F<X,Y1> f;
```

```
public:
   Y2 operator()(X const& x)const{ return f(x);}
   convertRange( const F<X,Y1>& f_):f(f_){}
};

} // auxiliatry

////////////////////// class ConvertDomain<> ////////////////////////

template <class X1, class X2, class Y>
// usage e.g. :
//   F<R,C> f=.....
//   ConvertDomain<R,N,C> converter;
//   F<N,C> f_N=converter(f);
// The same converter object can be used for converting other functions
// of the same domain and range classes.
class ConvertDomain{ // converting the function domain
public:
   ConvertDomain(void){}
   F<X2,Y> operator()(const F<X1,Y>& f)const
      // transforms a F<X1,Y> in a F<X2,Y> if there is an
      // automatic conversion from X1 to X2.
   {
      return F<X2,Y>(new aux::convertDomain<X1,X2,Y>(f));
   }
};

////////////////////// class ConvertRange ////////////////////////

template <class X, class Y1, class Y2>
// Usage:
//   F<R,C> f=.....
//   ConvertRange<R,C,Cd> converter;
//   F<R,Cd> f_Cd=converter(f);
// The same converter object can be used for converting other functions
// of the same domain and range classes.
class ConvertRange{ // converting the function range
public:
   ConvertRange(void){}
   F<X,Y2> operator()(const F<X,Y1>& f)const
      // transforms a F<X,Y1> in a F<X,Y2> if there is an
      // automatic conversion from Y1 to Y2.
   {
      return F<X,Y2>(new aux::convertRange<X,Y1,Y2>(f));
   }
};

/// functions of more than one variable
```

```
namespace aux{

template <class Y1, class Y2, class Y3>
class Arg2FncObj : public FncObj<cpmX2<Y1,Y2>, Y3>{

   const Y3 (*fp)(const Y1&, const Y2& );

public:
   Y3 operator()(const cpmX2<Y1,Y2>& x)const
   { return fp(x.c1(),x.c2());}

   Arg2FncObj( Y3 (*g)(const Y1&, const Y2&)):fp(g){}
};

} // aux

#ifdef CPM_Fn
//////////////////// class F_2<> ////////////////////////////////////////

template <class Y1, class Y2, class Y3>
class F_2{ // functions of two variables

   const F<cpmX2<Y1,Y2>,Y3> f;

public:

   F_2(Y3 (*f_)(const Y1&, const Y2&)):
      f(new aux::Arg2FncObj<Y1,Y2,Y3>(f_)){}
      // construction from function pointer

   F_2(const F<CpmArrays::X2<Y1,Y2>,Y3>& f_):f(f_){}
      // construction from 'smart function object'

   F<CpmArrays::X2<Y1,Y2>,Y3> operator()(void)const{ return f;}
      // evaluation to 'smart function object'

   Y3 operator()(const Y1& y1, const Y2& y2)const
      // evaluation to value
   { return f(CpmArrays::X2<Y1,Y2>(y1,y2));}

   F<Y2,Y3> c1(const Y1& y1)const
      // evaluation to a 'smart function object'
      // representing a function of one variable;
      // obtained by binding the first of the two variables
   { return Bind1<Y1,Y2,Y3>(y1)(f);}

    F<Y1,Y3> c2(const Y2& y2)const
     // ...  binding the second of the two variables
   { return Bind2<Y1,Y2,Y3>(y2)(f);}
```

```
};

#endif // CPM_Fn

// define a function by selecting one function value from two functions

namespace aux{

template <class X, class Y>
class sel: public FncObj<X,Y>{

   const F<X,Y>  f1;
   const F<X,Y>  f2;
   const F<X,Z>  s;

public:

   sel(const F<X,Y> & f1_, F<X,Y>  const& f2_, const F<X,Z>& s_):
      f1(f1_),f2(f2_),s(s_){}

   Y operator()(X const& )const;
};

template <class X, class Y>
Y sel<X,Y>::operator()(X const& x)const
{
   Z i=s(x);
   if (i==1) return f1(x);
   else if (i==2) return f2(x);
   else {
      cpmerror
         ("select::operator(): unvalid value of selection function");
      return f1(x); // never done
   }
}

////////////////////// class vselect<> ////////////////////////////////////

// define a function by selecting one function value from a sequence of
// functions

template <class X, class Y>
class vselect: public FncObj<X,Y>{

   const V< F<X,Y> > f;
   const F<X,Z>  sel;

public:

   vselect(const V< F<X,Y> >& f_, const F<X,Z> & sel_):
```

```
      f(f_),sel(sel_){}

   Y operator()(X const& )const;

};

template <class X, class Y>
Y vselect<X,Y>::operator()(X const& x)const
{
   Z i=sel(x);
   return (f[i])(x);
}

} // aux

////////////////// function select<> ////////////////////////////////

template <class X, class Y>
F<X,Y> select(F<X,Y>  const& f, F<X,Y>  const& g, const F<X,Z>& h)
// The returned function res has the property res(x)==f(x) if h(x)==1,
// res(x)== g(x) if h(x)==2, cpmerror() else
{
   return F<X,Y>(new aux::sel<X,Y>(f,g,h));
}

template <class X, class Y>
F<X,Y> select(const V<F<X,Y> >& f, const F<X,Z>& h)
// The returned function res has the property res(x)=(f[g(x)])(x);
{
   return F<X,Y>(new aux::vselect<X,Y>(f,h));
}

} // namespace

#endif
```

15 cpmfa.h

```
//? cpmfa.h
//? Status of work 2023-10-20.
//?
//?  ...


#ifndef CPM_FA_H_
#define CPM_FA_H_
////////////////////////////////////////////////////////////////////////
//
// Purpose: Define classes describing function-like objects with
//   arithmetic operations
//
////////////////////////////////// class FUOBJ<> ///////////////////////////
#include <cpmfo.h>
#include <cpmtypes.h>

namespace CpmFunctions{

  // using std::ostream;
  // using std::istream;
   using namespace CpmStd;
   using CpmRoot::Z;

   namespace auxiliary { // is within a Cpm... namespace !

//////////////////// tools for defining class Fa<X,Y> ///////////////////

// Advanced arithmetics of functions based on handles to FncObjs

// Arithmetic operations: Here it is assumed that Y is a ring i.e. that
// Y+Y, Y-Y, Y*Y, -Y are defined. Also division Y/Y is assumed to be
// defined. Also template functions/operators will be implemented which
// do not depend only on X and Y.

// class SUMFUO<>

template <class X, class Y>
class SUMFUO: public BINOPFUO<X,Y>{
   typedef SUMFUO<X,Y> Type;
   CPM_INVAR(SUMFUO)
public:
   Y operator()(const X& x)const{ return this->f1(x)+this->f2(x);}
   SUMFUO( const F<X,Y> & f1In, const F<X,Y>& f2In)
   :BINOPFUO<X,Y>(f1In,f2In){}
};
```

```
// class DIFFFUO<>

template <class X, class Y>
class DIFFFUO: public BINOPFUO<X,Y>{

public:
   Y operator()(const X& x)const{ return this->f1(x)-this->f2(x);}
   DIFFFUO( const F<X,Y> & f1In, const F<X,Y> & f2In)
   :BINOPFUO<X,Y>(f1In,f2In){}
};

// class NEGFUO<>

template <class X, class Y>
class NEGFUO: public MONOPFUO<X,Y>{

public:
   Y operator()(X const& x)const{ return -this->f1(x);}
   NEGFUO(const F<X,Y> & f1In):MONOPFUO<X,Y>(f1In){}
};

// class PRODFUO<>

template <class X, class Y>
class PRODFUO: public BINOPFUO<X,Y>{

public:
   Y operator()(const X& x)const{ return this->f1(x)*this->f2(x);}
   PRODFUO( const F<X,Y> & f1In, const F<X,Y> & f2In)
   :BINOPFUO<X,Y>(f1In,f2In){}
};

// class DIVIFUO<>

template <class X, class Y>
class DIVIFUO: public BINOPFUO<X,Y>{

public:
   Y operator()(const X& x)const{ return  this->f1(x)/this->f2(x);}
   DIVIFUO( const F<X,Y> & f1In, const F<X,Y> & f2In)
   :BINOPFUO<X,Y>(f1In,f2In){}
};

template <class X, class Y>
class INVFUO : public FncObj<X,Y>{
   const F<X,Y> f;
public:
   INVFUO(const F<X,Y>& f_):f(f_){}
   Y operator()(const X& x)const
   { return CpmRoot::invT<Y>(f(x));}
```

```
};

// treating operations which need more than 2 template arguments

template <class X1, class X2, class X3>
class TENSFUO: public FncObj< cpmX2<X1,X2>, X3 >{
   // TENS stands for Tensor Product

   const F<X1,X3>  f1;
   const F<X2,X3>  f2;

public:
   X3 operator()(const cpmX2<X1,X2>& x)const
      { return f1(x.get1())*f2(x.get2());}
   TENSFUO( const F<X1,X3> & f1In, const F<X2,X3> & f2In):
      f1(f1In),f2(f2In){}
};

template <class X1, class X2, class X12, class Y1, class Y2, class Y12>
class GENPRODFUO: public FncObj< X12,Y12 >{
   // GENPROD stands for general product. We assume that X12 is a class
   // which defines functions get1() and get2() such that for each x12 in
   // X12 the values of x12.get1() and x12.get2() allow unique automatic
   // conversion to X1 and X2 respectively. Further, we assume that for
   // each y1 in Y1 and y2 in Y2 the value of Y12(y1,y2) is an element in
   // Y12. Notice that the normal tensor product
   // TENSFUO<X1,X2,X3>(f1,f2)
   // would be obtained as
   // GENPRODFUO<X1,X2,cpmX2<X1,X2>,Y3,Y3,Triv<Y3> > (f1,f2)
   // where
   //    template <class X>
   //    class Triv: public X{
   //    public:
   //       Triv(const X& x):X(x){}
   //       Triv(const X& x1, const X& x2):X(x1*x2){}
   //   };
   //

   const F<X1,Y1>  f1;
   const F<X2,Y2>  f2;

public:
   Y12 operator()(const X12& x)const{ return Y12(f1(x.c1()),f2(x.c2()));}
   GENPRODFUO( const F<X1,Y1> & f1In, const F<X2,Y2>& f2In):
      f1(f1In),f2(f2In){}
};

template <class X, class Y, class S>
   // assumption S*Y defined, in typical situations, Y is a linear space
   // over S
```

```
class MULTFUO: public FncObj<X,Y>{

   const F<X,Y>  f1;
   const S s;

public:

   Y operator()(const X& x)const{ return s*(f1(x));}

   MULTFUO(const S& sIn, const F<X,Y> & f1In):f1(f1In),s(sIn){}

};

template <class X, class Y, class S>
   // assumption Y*S defined, in typical situations, Y is a linear space
   // over S

class MULTFUOR: public FncObj<X,Y>{

   const F<X,Y>  f1;
   const S s;

public:

   Y operator()(const X& x)const{ return (f1(x))*s;}

   MULTFUOR(const S& sIn, const F<X,Y> & f1In):f1(f1In),s(sIn){}

};

template <class X, class Y>
   // assumption Y+Y defined

class Y_SUMFUO: public FncObj<X,Y>{

   const F<X,Y>  f1;
   const Y y;

public:

   Y operator()(const X& x)const{ return (f1(x))+y;}

   Y_SUMFUO(const F<X,Y> & f1In, const Y& yIn):f1(f1In),y(yIn){}

};

template <class X, class Y>
   // assumption Y-Y defined
```

```
class Y_DIFFFUO: public FncObj<X,Y>{

   const F<X,Y> f1;
   const Y y;

public:

   Y operator()(const X& x)const{ return f1(x)-y;}

   Y_DIFFFUO(const F<X,Y> & f1In, const Y& yIn):f1(f1In),y(yIn){}

};

template <class X, class Y>
   // assumption Y*Y defined

class Y_PRODFUO: public FncObj<X,Y>{

   const F<X,Y>  f1;
   const Y y;

public:

   Y operator()(const X& x)const{ return f1(x)*y;}

   Y_PRODFUO(const F<X,Y> & f1In, const Y& yIn):f1(f1In),y(yIn){}

};

template <class X, class Y>

class Y_DIVIFUO: public FncObj<X,Y>{

protected:

   const F<X,Y>  f1;
   const Y y;

public:

   Y operator()(const X& x)const{ return f1(x)/y;}

   Y_DIVIFUO(const F<X,Y> & f1In,const Y& yIn):f1(f1In),y(yIn){}

};

} // auxiliary

////////////////////////////////// class Fa<X,Y> ////////////////////////
```

```
// Arithmetics and some additional functionality for Fo<X,Y>

// Now arithmetics is introduced. Just as in class Va, we assume that
// -Y, Y+Y, Y-Y, Y*Y, Y/Y , Y < Y, Y > Y, are defined
// Note 02-05-24:
// Notice that e.g. addition of functions is not implemented by really
// 'adding something together' but arranging pointers to the code
// of all functions in the sum in a manner that when the sum function is
// asked to be evaluated for some value of the argument, all functions
// needed in the sum are ready for evaluation and addition of the
// output-values. This is sometimes called 'deferred evaluation'.
// I found that code of type:
// Fa<X,Y> f;
// Z i,n=3300;
// V< Fa<X,Y> > vf(n);
// for (i=1;i<=n;i++) vf[i]=...;
// for (i=1;i<=n;i++) f+=vf[i];
// caused stack overflow in the last line

template <class X, class Y>
class Fa: public Fo<X,Y> { // version of Fo with arithmetics operations

public:

   typedef Fa<X,Y> Type;
   typedef Y ScalarType;

   Fa(void):Fo<X,Y>(){}
      // default constructor

   Fa(const F<X,Y>& g):Fo<X,Y>(g){}
      // downcast constructor

   Fa(const Fo<X,Y>& g):Fo<X,Y>(g){}
      // downcast constructor

   Fa(const Fa<X,Y>& g):Fo<X,Y>(g){}
      // copy constructor

   Fa(FncObj<X,Y>* fop):Fo<X,Y>(fop){}
      // constructor from pointers to FncObj

// Construction from function pointers

   Fa( Y (*f)(const X& )):Fo<X,Y>(f){}
   Fa( Y (*f)(X)):Fo<X,Y>(f){}

   Fa( std::function<Y(X)> f):Fo<X,Y>(f){}

// Construtor for constant function
```

```
   explicit Fa(const Y& y1):Fo<X,Y>(y1){}

   virtual F<X,Y>* clone()const{ return new Fa(*this);}

   virtual Word nameOf()const
   {
      Word wi="Fa<";
      Word wx=CpmRoot::Name<X>()(X());
      Word wy=CpmRoot::Name<Y>()(Y());
      return wi&wx&","&wy&">";
   }

   CPM_SUM_C
   CPM_PRODUCT_C
   CPM_DIFFERENCE
   CPM_DIVISION
   CPM_SCALAR_C
   CPM_IO_V

   void scl_(X const& fac, X const& x0=X());
      //: scale
      // changes the function f into x|-->f(x0+(x-x0)*fac)
      // assumes that X defines addition, subtraction, and multiplication,
      // typically that X is a ring. Similar to function stretch_ in
      // class R_Func. For X=R this operation applied with fac>1
      // lets features of function *this (such as a peak width) shrink and
      // fac<1 lets them grow.
};

namespace{// anonymus
   template <class X, class Y>
   Y scaleFunc(X const& x, X const& x0, X const& fac, F<X,Y> const& f)
   { return f(x0+(x-x0)*fac);}
}// anonymus

template <class X, class Y>
void Fa<X,Y>::scl_(X const& fac, X const& x0)
// elegant implementation, avoids using operator new directly.
{
#ifdef CPM_Fn
   *this=F3<X,X,X,F<X,Y>,Y>(x0,fac,*this)(scaleFunc);
#else
   *this=F<X,Y>(bind(scaleFunc<X,Y>,_1,x0,fac,*this));
#endif
}

// functions(operators) outside this class. Here additional template
// arguments are allowed, which creates the flexibility needed for
// treating composition and tensor product.
```

```
template <class X1, class X2, class X3>

Fa<cpmX2<X1,X2>,X3> operator || (const Fa<X1,X3>& f13,
         const Fa<X2,X3>& f23)
   // Tensor product of functions: Assumption is that X3*X3 is defined.
   // This creates functions of two variables out of two functions of one
   // variable. Symbol || should remind to the old symbol )( for the
   // 'dyadic product' which is essentially the same construction
{
   using namespace auxiliary;
   return Fa<cpmX2<X1,X2>,X3>(new TENSFUO<X1,X2,X3>(f13,f23));
}

template <class X, class Y, class S>
   // assumption S*Y defined, in typical situations,
   // Y is a linear space over S

Fa<X,Y> mult(const S& sIn, const Fa<X,Y>& fIn)
   // Function values are multiplied by s, operator * instead of
   // mult results in compiler errors caused by mistaking this as
   // another operator *
{
   using namespace auxiliary;
   return Fa<X,Y>(new MULTFUO<X,Y,S>(sIn,fIn));
}

template <class X, class Y, class S>
   // assumption Y*S defined, in typical situations,
   // Y is a linear space over S
Fa<X,Y> multR(const S& sIn, const Fa<X,Y>& fIn)
   // Function values are multiplied by s, operator * instead of
   // mult results in compiler errors caused by mistaking this as
   // another operator *
{
   using namespace auxiliary;
   return Fa<X,Y>(new MULTFUOR<X,Y,S>(sIn,fIn));
}

template <class X, class Y>
Fa<X,Y> Fa<X,Y>::neg(void)const
{
   using namespace auxiliary;
   return Fa<X,Y>(new NEGFUO<X,Y>(*this));
}

template <class X, class Y>
Fa<X,Y> Fa<X,Y>::inv(void)const
{
   return Fa<X,Y>(new auxiliary::INVFUO<X,Y>(*this));
```

```
}

template <class X, class Y>
Fa<X,Y> Fa<X,Y>::net(CpmRoot::Z i)const
{
   Y y0;
   Y yn=CpmRoot::netT<Y>(y0,i);
   return Fa<X,Y>(yn);
}

template <class X, class Y>
Fa<X,Y> Fa<X,Y>::operator +(const Fa<X,Y>& f )const
{
   using namespace auxiliary;
   return Fa<X,Y>(new SUMFUO<X,Y>(*this,f));
}

template <class X, class Y>
Fa<X,Y> Fa<X,Y>::operator *(const Fa<X,Y>& f )const
{
   using namespace auxiliary;
   return Fa<X,Y>(new PRODFUO<X,Y>(*this,f));
}

template <class X, class Y>
Fa<X,Y> Fa<X,Y>::operator *(const Y& y )const
{
   using namespace auxiliary;
   return Fa<X,Y>(new Y_PRODFUO<X,Y>(*this,y));
}

template <class X, class Y>
Fa<X,Y> Fa<X,Y>::operator +(const Y& y )const
{
   using namespace auxiliary;
   return Fa<X,Y>(new Y_SUMFUO<X,Y>(*this,y));
}

template <class X, class Y>
Fa<X,Y> operator *( const Y& y, const Fa<X,Y>& f1 )
{
//   using namespace auxiliary;
   return (f1*y);
}

// I/O interface trivial. No assumtions on X,Y

template <class X, class Y>
bool Fa<X,Y>::prnOn(ostream& out)const
{
```

```
   return CpmRoot::writeTitle(" a function",out);
}

template <class X, class Y>
bool Fa<X,Y>::scanFrom(istream& in)
{
   in;
  // return (in!=0);
   if(!in) return false;
   else return true;
}

} // namespace

#endif
```

16 cpmfl.h

```
//? cpmfl.h
//? Status of work 2023-10-20.
//?
//?  ...


#ifndef CPM_FL_H_
#define CPM_FL_H_
/*
Purpose: Define class F<X,Y> describing 'functions' from
   X to Y. Later a derived class Fa<X,Y> will define arithmetic
   operations. This description also has in mind this class.
   The classes have default constructor, copy constructor, and assignment
   operator. Thus these 'functions' behave like the most conventional
   arithmetic quantities (they have value semantics, moreover they
   satisfy the strict value interface, see preamble of cpmv.h)

   Example:

      F<R,R> f1(cos);
      F<R,R> f2(sin);
      F<R,R> f3=f1&f2;
      F<R,R> f4=f1+f2;
      R x=1.2345;
      R eps1=f3(x)-sin(cos(x));
      R eps2=f4(x)-sin(x)-cos(x);

   will result in eps1=eps2=0. I.e. the concatenation of functions sin()
   and cos() can be created as a F object simply by a binary operator
   acting on the F objects corresponding to sin() and cos() (notice the
   order convention for & which is not intuitive to everybody).
   F<>-instances are allowed as arguments and return values of functions:

      F<C,C> power3(const F<C,C>& g)
      {
         return g*g*g;
      }

   creates the third power of a complex function, i.e.
      power3(g)(x)=g(x)*g(x)*g(x).

   In a nutshell, these new functions behave just like normal variables.
   They can also serve as data members in class definitions. This is
   important in physics and engineering applications where e.g.
   electrical fields now can be directly represented as functions instead
   of discrete value samples. They also can form components of vectors.
```

```
For instance, a family of functions consisting of the first 101
Chebyshev polynomials
      T0(x),....T100(x)
can be defined by verbally coding the (numerically not optimal)
definition

      Tn(x):=cos(n*arccos(x)):
Z n=100;
IvZ iv(0,n);
V<Fa<R,R> > Chebyshev(iv);
Fa<R,R> ACos(acos);
Fa<R,R> Cos(cos);
for (Z i=0; i<=n; i++) Chebyshev[i]=(ACos*i)&Cos ;

The value of the 27-th Chebyshev polynomial for x=0.5 then will be
written as Chebyshev[27](0.5)

In defining new classes, data members of type F<X,Y> are allowed
and for the classes defined this way the default copy constructor
and default assignment operator are OK.

Compared to all other classes that I dealt with, F<X,Y> shows
the particularity that it is not possible to implement
CPM_IO and CPM_ORDER (see cpminterfaces.h) in a useful manner.
Later Fr<X,Y> tries to do the best one can in this respect.

Remarks on 'functions as values':

Consider

   R sum=0;
   for (Z i=1;i<=1000;++i) sum+=(result of some complicated
      computation with i);
   R x=sum;

and the similiar construct

   F<R,R> f;
   for (Z i=1;i<=1000;++i) f+=(Sin*i); // (*)
   F<R,R> g=f;

There is a remarkable difference between the two:
Whenever x will be used in subsequent parts of the program it is
simply a number and the potential pain of getting to it is forgotten.
On the contrary, when using g by evaluating it for specific
values of the argument, e.g.
   R y=g(137)-g(1./137);
each evaluation has to go through the huge expression which g is
defined in (*) to be. What was achieved by defining g compactly
was to compose the code defining the Sin*i into the code defining g.
```

```
   But this code was composed verbatim without any intent (and chance)
   of simplification or compactification. So the normal idea  associated
   with evaluation is misleading in this case.
   In a time-stepping simulation one could be tempted
   to replace in each step a data member F<X,X> f of the evolving system
   by f&g with some transformation g characteristic for the step under
   consideration. This would make f more expensive to compute with each
   step (actually we would run into stack overflow sometime).
   Here the right way is to parameterize the mappings and express their
   composition in terms of a computation done with these parameters.
   Sophus Lie has forseen the importance of this kind of computational
   treatment of the combination of transformations. With group theory,
   Lie groups of transformations, and their linear representations, there
   has a huge body of knowledge grown on this topic.
*/

#include <cpmuc.h>
   // for reference counting handle template CpmArrays::P<X>
#include <cpmword.h>
#include <functional>

//#undef CPM_Fn
//#define CPM_Fn CPM_Fn has to be set in cpmdefinitions.
// The next version of C+- will very probably have the form which
// corresponds here to #undef CPM_Fn.
// No longer so sure. The simple basic run of the my pala program
// first ran 1% faster with the old method. But a new run did fiffer only
// by 0.2 s of 95 s. So,there seems to be no significant speed difference.

namespace CpmFunctions{

   using CpmRoot::Z; // for order relation
   using CpmRoot::Word;
   using CpmArrays::P;
  // using std::function;

////////////////////////////// class FncObj<> ////////////////////////

template <class X, class Y>
class  FncObj{ // function objects
   // FncObj stands for function object
   // A class which is derived from some FncObj<X,Y> is said to be a
   // function class and all instances of function classes are called
   // function objects. A function object can do everything (and more)
   // than a function can do. Unlike a function, a function class can be
   // defined within a function block. Let G be a function class and g an
   // object belonging to it. The value of g at x can be written as
   // g(x) (see. operator()). If in deriving G from
   // FncObj<X,Y>, no further non-static data members are added (then
   // G is said to be a pure function class), G has just one instance
```

```
   // g; this is selected by a statement as simple as
   //      G g;
   // Despite this close relationship between function objects and their
   // classes, there is an important syntactic difference between these
   // entities. As already mentioned, function values are accessed via
   // instances, e.g. Y y=g(x). Function arithmetics can be implemented
   // via template classes, among the arguments of which are
   // function classes. We will use such templates only as a transitory
   // step in implementing value-like function objects in F<X,Y>,
   // Fa<X,Y>.

   // Notice that FncObj<X,Y> is an abstract class. This is a usefull
   // prevention against invalid attempts to re-define operator() (e.g.
   // by forgetting the final 'const' attribute.
   //
   // Example: function class describing the sin function:
   //
   //    class Sin: public FncObj<R,R>{
   //       public:
   //       R operator()(const R& x)const{ return sin(x);}
   //    };
   // classes derived from FncObj may implement a more flexible and
   // effective evaluation scheme than mere function: certain
   // preparatory actions (e.g. computation of auxiliary quantities that
   // are held as data members) can be done by constructor code, so that
   // evaluation of operator()(X const&) gets simpler.

   typedef FncObj<X,Y> Type;
   CPM_INVAR(FncObj)
      // makes sure that no client class will use FncObj in a way that
      // makes use of copy or assignment

public:

   FncObj(void){}

   virtual ~FncObj(void){}

   virtual   Y operator()(X const& x)const=0;

};

namespace aux{
// prevention against polluting namespace CpmFunctions
// The classes to be defined in this namespace aux are not intended
// for direct usage. They will be used internally for building
// classes which have a more expressive interface: the
// classes F<>,F1<>, .... F6<>
// and
// F1_1<>, F2_1<>, F2_2<>, F3_1<>, F3_2<>, F3_3<>, F4_1<>, F4_2<>,
```

```
// F4_3<>, F4_4<>

////////////////// ParFncObj ////////////////////////////////////////////

template <class X, class Y>
class ParFncObj : public FncObj<X,Y>{

   Y (* const fp)(X);
public:
   Y operator()(X const& x)const{ return (*fp)(x);}
   ParFncObj( Y (*g)(X)):fp(g){}
};

////////////////// Par0FncObj ////////////////////////////////////////////
template <class X, class Y>
class Par0FncObj : public FncObj<X,Y>{

   Y (* const fp)(X const&);

public:
   Y operator()(X const& x)const{ return (*fp)(x);}
   Par0FncObj( Y (*g)(X const&)):fp(g){}
};

////////////////// StdFunctionObj ////////////////////////////////////////
template <class X, class Y>
// Addition 2020-02-13. Means to use also std::function<Y(X)> objects
// as a source for F<X,Y>-objects. Requires C++11 or higher.
class StdFunctionObj : public FncObj<X,Y>{

   const std::function<Y(X)> f_;

public:
   Y operator()(X const& x)const{ return f_(x);}
   StdFunctionObj( std::function<Y(X)> f):f_(f){}
};

////////////////// ParcFncObj ////////////////////////////////////////////
// Addition 2010-01-13. Means to use also functions with
// constant return value as a source for F<X,Y>-objects.
template <class X, class Y>
class ParcFncObj : public FncObj<X,Y>{

   const Y (* const fp)(X const&); // This is the prototype of
   // many mathematical functions in mpreal have (unfortunately
   // up to an additional default argument)

public:
   Y operator()(X const& x)const{ return Y((*fp)(x));}
   ParcFncObj( const Y (*g)(X const&)):fp(g){}
```

```
};

//////////////////// Par1FncObj ////////////////////////////////////////
template <class X, class P1, class Y>
class Par1FncObj : public FncObj<X,Y>{

   const P1 p_;
   Y (* const f_)(X const&, P1 const& );

public:
   Y operator()(X const& x)const{ return (*f_)(x,p_);}
 //  P1 const& getP(void)const{return p_;} // experiment
   Par1FncObj( Y (*g)(X const&, P1 const&), P1 const& p):p_(p),f_(g){}
};

/////////////// Par1_1FncObj ////////////////////////////////////////
template <class X, class P1, class Y>
class Par1_1FncObj : public FncObj<X,Y>{

   const X x_;
   Y (* const f_)(X const&, P1 const& );

public:
   Y operator()(P1 const& p)const{ return (*f_)(x_,p);}
   Par1_1FncObj( Y (*g)(X const&, P1 const&), X const& x):x_(x),f_(g){}
};

#ifdef CPM_Fn
/////////////// Par2FncObj ////////////////////////////////////////
// See F2, F2_1, F2_2 for the reasons to introduce these class templates

template <class X, class P1, class P2, class Y>
class Par2FncObj : public FncObj<X,Y>{

   const P1 p1_;
   const P2 p2_;
   Y (* const f_)(X const&, P1 const&, P2 const& );

public:
   Y operator()(X const& x)const{ return (*f_)(x,p1_,p2_);}
   Par2FncObj(Y (*g)(X const&, P1 const&, P2 const&), P1 const& p1,
      P2 const& p2)
   :p1_(p1),p2_(p2),f_(g){}
};

/////////////// Par2_1FncObj ////////////////////////////////////////

template <class X, class P1, class P2, class Y>
class Par2_1FncObj : public FncObj<X,Y>{
```

```
   const X x_;
   const P2 p2_;
   Y (* const f_)(X const&, P1 const&, P2 const& );

public:
   Y operator()(P1 const& p1)const{ return (*f_)(x_,p1,p2_);}
   Par2_1FncObj(Y (*g)(X const&, P1 const&, P2 const&), X const& x,
      P2 const& p2)
   :x_(x),p2_(p2),f_(g){}
};

/////////////// Par2_2FncObj ////////////////////////////////////////

template <class X, class P1, class P2, class Y>
class Par2_2FncObj : public FncObj<X,Y>{

   const X x_;
   const P1 p1_;
   Y (* const f_)(X const&, P1 const&, P2 const& );

public:
   Y operator()(P2 const& p2)const{ return (*f_)(x_,p1_,p2);}
   Par2_2FncObj(Y (*g)(X const&, P1 const&, P2 const&), X const& x,
      P1 const& p1)
   :x_(x),p1_(p1),f_(g){}
};

/////////////// Par3FncObj ////////////////////////////////////////

template <class X, class P1, class P2, class P3, class Y>
class Par3FncObj : public FncObj<X,Y>{

   const P1 p1_;
   const P2 p2_;
   const P3 p3_;
   Y (* const f_)(X const&,P1 const&,P2 const&,P3 const&);

public:
   Y operator()(X const& x)const{ return (*f_)(x,p1_,p2_,p3_);}
   Par3FncObj(Y (*g)(X const&,P1 const&,P2 const&,P3 const&),
   P1 const& p1, P2 const& p2, P3 const& p3)
   :p1_(p1),p2_(p2),p3_(p3),f_(g){}
};

/////////////// Par3_1FncObj ////////////////////////////////////////

template <class X, class P1, class P2, class P3, class Y>
class Par3_1FncObj : public FncObj<X,Y>{

   const X x_;
```

```
   const P2 p2_;
   const P3 p3_;
   Y (* const f_)(X const&,P1 const&,P2 const&,P3 const&);

public:
   Y operator()(P1 const& p1)const{ return (*f_)(x_,p1,p2_,p3_);}
   Par3_1FncObj(Y (*g)(X const&,P1 const&,P2 const&,P3 const&),
   X const& x, P2 const& p2, P3 const& p3)
   :x_(x),p2_(p2),p3_(p3),f_(g){}
};

/////////////// Par3_2FncObj ////////////////////////////////////////

template <class X, class P1, class P2, class P3, class Y>
class Par3_2FncObj : public FncObj<X,Y>{

   const X x_;
   const P1 p1_;
   const P3 p3_;
   Y (* const f_)(X const&,P1 const&,P2 const&,P3 const&);

public:
   Y operator()(P2 const& p2)const{ return (*f_)(x_,p1_,p2,p3_);}
   Par3_2FncObj(Y (*g)(X const&,P1 const&,P2 const&,P3 const&),
   X const& x, P1 const& p1, P3 const& p3)
   :x_(x),p1_(p1),p3_(p3),f_(g){}
};

/////////////// Par3_3FncObj ////////////////////////////////////////

template <class X, class P1, class P2, class P3, class Y>
class Par3_3FncObj : public FncObj<X,Y>{

   const X x_;
   const P1 p1_;
   const P2 p2_;
   Y (* const f_)(X const&,P1 const&,P2 const&,P3 const&);

public:
   Y operator()(P3 const& p3)const{ return (*f_)(x_,p1_,p2_,p3);}
   Par3_3FncObj(Y (*g)(X const&,P1 const&,P2 const&,P3 const&),
   X const& x, P1 const& p1, P2 const& p2)
   :x_(x),p1_(p1),p2_(p2),f_(g){}
};

/////////////// Par4FncObj ////////////////////////////////////////

template <class X, class P1, class P2, class P3, class P4, class Y>
class Par4FncObj : public FncObj<X,Y>{
```

```
   const P1 p1_;
   const P2 p2_;
   const P3 p3_;
   const P4 p4_;
   Y (* const f_)(X const&, P1 const&, P2 const& ,P3 const& ,P4 const&);

public:
   Y operator()(X const& x)const{ return (*f_)(x,p1_,p2_,p3_,p4_);}
   Par4FncObj(Y (*g)(X const&,P1 const&,P2 const&,P3 const&,P4 const&),
   P1 const& p1,P2 const& p2,P3 const& p3,P4 const& p4)
   :p1_(p1),p2_(p2),p3_(p3),p4_(p4),f_(g){}
};

/////////////// Par4_1FncObj ////////////////////////////////////////

template <class X, class P1, class P2, class P3, class P4, class Y>
class Par4_1FncObj : public FncObj<X,Y>{

   const X x_;
   const P2 p2_;
   const P3 p3_;
   const P4 p4_;
   Y (* const f_)(X const&, P1 const&, P2 const& ,P3 const& ,P4 const&);

public:
   Y operator()(P1 const& p1)const{ return (*f_)(x_,p1,p2_,p3_,p4_);}
   Par4_1FncObj(Y (*g)(X const&,P1 const&,P2 const&,P3 const&,P4 const&),
   X const& x,P2 const& p2,P3 const& p3,P4 const& p4)
   :x_(x),p2_(p2),p3_(p3),p4_(p4),f_(g){}
};

/////////////// Par4_2FncObj ////////////////////////////////////////

template <class X, class P1, class P2, class P3, class P4, class Y>
class Par4_2FncObj : public FncObj<X,Y>{

   const X x_;
   const P1 p1_;
   const P3 p3_;
   const P4 p4_;
   Y (* const f_)(X const&, P1 const&, P2 const& ,P3 const& ,P4 const&);

public:
   Y operator()(P2 const& p2)const{ return (*f_)(x_,p1_,p2,p3_,p4_);}
   Par4_2FncObj(Y (*g)(X const&,P1 const&,P2 const&,P3 const&,P4 const&),
   X const& x,P1 const& p1,P3 const& p3,P4 const& p4)
   :x_(x),p1_(p1),p3_(p3),p4_(p4),f_(g){}
};

/////////////// Par4_3FncObj ////////////////////////////////////////
```

```
template <class X, class P1, class P2, class P3, class P4, class Y>
class Par4_3FncObj : public FncObj<X,Y>{

   const X x_;
   const P1 p1_;
   const P2 p2_;
   const P4 p4_;
   Y (* const f_)(X const&, P1 const&, P2 const& ,P3 const& ,P4 const&);

public:
   Y operator()(P3 const& p3)const{ return (*f_)(x_,p1_,p2_,p3,p4_);}
   Par4_3FncObj(Y (*g)(X const&,P1 const&,P2 const&,P3 const&,P4 const&),
   X const& x,P1 const& p1,P2 const& p2,P4 const& p4)
   :x_(x),p1_(p1),p2_(p2),p4_(p4),f_(g){}
};

/////////////// Par4_4FncObj ////////////////////////////////////////

template <class X, class P1, class P2, class P3, class P4, class Y>
class Par4_4FncObj : public FncObj<X,Y>{

   const X x_;
   const P1 p1_;
   const P2 p2_;
   const P3 p3_;
   Y (* const f_)(X const&, P1 const&, P2 const& ,P3 const& ,P4 const&);

public:
   Y operator()(P4 const& p4)const{ return (*f_)(x_,p1_,p2_,p3_,p4);}
   Par4_4FncObj(Y (*g)(X const&,P1 const&,P2 const&,P3 const&,P4 const&),
   X const& x,P1 const& p1,P2 const& p2,P3 const& p3)
   :x_(x),p1_(p1),p2_(p2),p3_(p3),f_(g){}
};

#endif

template <class X1, class X2, class X3> class comp;

} // aux

//////////////////////////// class F<X,Y> ////////////////////////////////

// Defining a template class implementing the value interface out of the
// 'poor' class FncObj<X,Y> via the handle template CpmArrays::P<>

// No assumption on X
// Y has to define Y()

template <class X, class Y>
```

```
class F: public P< FncObj<X,Y> >{ // functions as a class
   // F stands for function

   typedef F<X,Y> Type;

   typedef P< FncObj<X,Y> > base;

   static Y f_const(X const& x, const Y& y0){x; return y0;}
   static Y f_default(X const& x){ x; return Y();}

public:
   virtual F<X,Y>* clone()const{ return new F(*this);}
   F<X,Y> toClnBase()const{ return *this;}
   CPM_ORDER
      // order is here a simple bookkeeping device
      // (based on position in memory space not on
      // function values, so no order relations in X
      // or Y are needed. This is sufficient for the set
      // template  S< F<X,Y> > to work.
      // Operations that deal with order and equality in Y
      // will be introduced in Fo and Fr.
   CPM_IO_V

      // neeeded for using V< <F<X,Y> >

// destructor, and assignment inherited from P<...>
// constructors
   F(FncObj<X,Y>* fop):base(fop){}
      // constructor from pointers to FncObj's. All deeper constructions
      // utilize this way. F<X,Y>(new DC(...)) where DC is a class
      // derived from FncObj<X,Y> and (...) are suitable constructor
      // arguments, is thus a correct construction of a F<X,Y> object.
      // The class DC should be immutable, i.e. all its
      // data are declared const.
      // That this is also directly possible for classes derived from
      // F<X,Y> it is essential to have this as automatic conversion
      // i.e. without an 'explicit' in front

   F(base const& g):base(g){}
      // upcast constructor
      // conversion needed that F<X,Y> is just a new name for
      // P< FncObj<X,Y> >

   F(void):base(new aux::Par0FncObj<X,Y>(f_default)){}
      // default constructor

   explicit F(Y const& y1):base(
      new aux::Par1FncObj<X,Y,Y>(f_const,y1)){}
      // constructor for constant function
```

```
// Construction from function pointers

   explicit F( Y (*f)(X const& )):
   base(new aux::Par0FncObj<X,Y>(f)){}

   explicit F( const Y (*f)(X const& )):
   base(new aux::ParcFncObj<X,Y>(f)){}

   explicit F( Y (*f)(X)):
   base(new aux::ParFncObj<X,Y>(f)){}

   explicit F( std::function<Y(X)> f):
   base(new aux::StdFunctionObj<X,Y>(f)){}
      // This C++11-facility allows the definition of F<X,Y> instances
      // from functional expressions in many variables in a convenient
      // manner: consider
      // Y f(X1 const& x1, X2 const& x2, X3 const& x3)
      // we would like to define a function X3 --> Y, x3 |--> f(a,b,x3)
      // for some a in X1 and b in X2.
      // We set:
      // #include <functional>
      // using std::bind;
      // using std::function;
      // using namespace std::placeholders;
      // function<Y(X3)> g(bind(f,a,b,_1))
      // and F<X3,Y> f3(g) or less verbose F<X3,Y> f3(bind(f,a,b,_1)).
      // In expression bind(f,a,b,_1) the placeholder is _1 and not _3
      // since it becomes the single first argument of function f3.
      // Making use of this method, we never need the classes
      // F<>,F1<>, .... F6<>
      // and
      // F1_1<>, F2_1<>, F2_2<>, F3_1<>, F3_2<>, F3_3<>, F4_1<>, F4_2<>,
      // F4_3<>, F4_4<>. Actually, the macro CPM_Fn allows to purge those
      // from the active code. With my present system the larger generality
      // of the method based on std::function and std::bind results in
      // slightly longer compilation time (10%) and longer run time (1%-2%)
      // a single test with program pala.

   //F<X,Y>& operator=( F<X,Y> const& f){ return *this=f;}
      // should not be needed
   F<X,Y>& operator=( Y (*f)(X) ){ return *this=F<X,Y>(f);}
      // added 2005-04-16. Now we can write also
      // F<R,R> sinCpm; sinCpm=sin;
      // instead of F<R,R> sinCpm(sin);
      // The = syntax is particularly convenient in arrays:
      // V< F<R,R> > fList(3);
      // fList[1]=sin; fList[2]=cos; fList[3]=exp;
      // Beware of F<R,R> sinCpm=sin; which can't be valid in one's
      // right mind
   F<X,Y>& operator=( Y (*f)(X const&)){ return *this=F<X,Y>(f);}
```

```
      // see previous function

// enabling 'function value taking' via '()' for the underlying FncObj
   virtual Y operator()(X const& x)const
   {
      return (base::operator()())(x);
      // this complicated-looking syntax is the price for using
      // operator()(void) for 'de-referencing' in 'pointer'-class P<>,
      // since the function to be defined itself is operator(), a fully
      // scope resolved name is to be used for an expression which in a
      // different scope could be simply (*this)()(x)
   }

   Y evl(X const& x)const{ return (this->base::operator()())(x);}
      // evl stands for evaluation

// turning *this into a constant-valued approximation (the most simple
// easy-to-evaluate approximating function)

   void makeConstant(X const& x0){ Y y0=(*this)(x0);*this=F<X,Y>(y0);}
      // the modified object is a constant-valued function taking
      // everywhere the value y0, which the un-modified one takes at
      // x=x0.

   void makeConstant_(X const& x0){ makeConstant(x0);}
      // proper naming for a mutating function.

// concatenation (composition) of functions.

   template< class T>
   F<X,T> operator&(F<Y,T> const& g)const
      // x |--> y=(*this)(x) |--> t=g(y)=g((*this)(x))
      // thus (f&g)(x)=g(f(x))
      // notice that f & g : x |--> f(x) |--> g(f(x))
      // first action by f, second action by g thus n o t
      // (f&g)(x) = f(g(x))    wrong  !!!!
      // but
      // (f&g)(x) = g(f(x))     right !!!!
      // If one would write x.f for f(x) one had
      // x.(f&g) = x.f.g which looks much better
      // In the literature one finds f;g for f&g
   { return F<X,T>(new aux::comp<X,Y,T>(*this,g));}

   F<X,Y>& operator&=(F<Y,Y> const& g)
   { *this=operator&(g); return *this;}
      // mutating form of &, but type should not change, thus
      // T == Y needed

   template< class T>
   F<T,Y> circ(F<T,X> const& g)const
```

```
      // t|-->x=g(t)|-->y=(*this)(x)=(*this)(g(t))
      // thus (f.circ(g))(x)=f(g(x))
      // \circ is the LATEX name for the usual function
      // composition symbol
   { return F<T,Y>(new aux::comp<T,X,Y>(g,*this));}

   F<X,Y>& circ_(F<X,X> const& g){ *this=circ(g); return *this;}
      // mutating form of circ, but type should not change, thus
      // T == X needed

   void leftCombine(F<X,X> const& fL){ *this=fL&(*this);}
      // *this is changed into fL & *this

   void rightCombine(F<Y,Y> const& fR){ *this=operator&(fR);}
      // *this is changed into *this & fR

   virtual Word nameOf()const
   {
      Word wi="F<";
      Word wx=CpmRoot::Name<X>()(X());
      Word wy=CpmRoot::Name<Y>()(Y());
      return wi&wx&","&wy&">";
   }
};

template <class X, class Y>
bool F<X,Y>::prnOn(std::ostream&)const{return false;}

template <class X, class Y>
bool F<X,Y>::scanFrom(std::istream&){return false;}

template <class X, class Y>
Z F<X,Y>::com(F<X,Y> const& f)const
{
   return base::com(f);
}

namespace aux{

template <class X1, class X2, class X3>
class comp: public FncObj<X1,X3>{
   const F<X1,X2> f1;
   const F<X2,X3> f2;
public:
   comp(const F<X1,X2>& f1_, const F<X2,X3>& f2_):f1(f1_),f2(f2_){}
   X3 operator ()(const X1& x1)const{ return f2(f1(x1));}
};

} // aux
```

```
#ifdef CPM_Fn
//////////////////////// class F1<> ////////////////////////////////////

template <class X, class P1, class Y>
class F1{ // functions with one parameter
    const P1 p1_;
    typedef F1<X,P1,Y> Type;
    CPM_INVAR(F1)
public:
explicit F1( P1 const& p):p1_(p){}
    F<X,Y> operator()( Y (*f)(X const&, P1 const&) )const;
};

// We explain the usage of F2<X,P1,P2,Y> which is representative
// for F1,F2,F3,..., F6 (see cpmf.h for F5 and F6).
// F2<X,P1,P2,Y> is a tool for constructing instances of
// F<X,Y> in a flexible manner from classical C-functions
// of prototype Y (*f)(X const&, P1 const&, P2 const&). The
// syntax is as follows:
// Assume definitions
// P1 p1(...);
// P2 p2(...);
// Y fc(X const& x, P1 const& p1, P2 const& p2){....}
// Then
// F<X,Y> f=F2<X,P1,P2,Y>(p1,p2)(fc);
// ( or F<X,Y> f(F2<X,P1,P2,Y>(p1,p2)(fc));
// defines a function f for which
// f(x)==fc(x,p1,p2)
// Thus f is 'the function fc with the parameters of types
// P1 and P2 given fixed values'.
// If the fixed values p1, p2 of these parameters are to be used for
// several classical functions of identical prototype,
// say fc, gc, hc, the the following syntax is economic:
// F2<X,P1,P2,Y> f2(p1,p2);
// F<X,Y> f=f2(fc);
// F<X,Y> g=f2(gc);
// F<X,Y> h=f2(hc);

// Addition 2011-03-10
// Tools to let any of the parameter play the role of the function
// argument:
// Consider the function fc as defined above. Sometimes one needs to
// consider it as, say, a function of the parameter p1 for fixed values
// of parameters x and p2. In such a case it is inconvenient to be forced
// to define a new function which has p1 in the first argument slot such
// as by the definition
//    Y fc1(P1 const& p1, X const& x, P2 const& p2)
//    {
//       return fc(x,p1,p2);
//    }
```

```
// To avoid this need we define a class template F2_1 which considers the
// first parameter slot as providing the function argument. Of course, the
// truly first slot is taken as a parameter slot in turn. Further, we
// define a class template F2_2 which takes the second parameter slot as
// the argument slot, and the other slots as parameter slots.
// This should be usable as a pattern to explain also the class templates
// F3_1, F3_2, F3_3, F4_1, F4_2, F4_3, F4_4.
// We don't define F5_{1,...,5} and F6_{1,...,6} since already 4 parameters
// is more than I ever needed in a context in which the Fn_m were useful.

// Addition 2015-06-11
// Having in mind a general function
// Y f(X1 const& x1, ... Xn const& xn)
// one could be interested in partitioning the index set {1,...,n} in an
// arbitrary way into two disjoint sets S1 and S2 and take the xi indexed
// by the second set as parameters and those indexed by the first set as
// arguments of a 'function of many variables'. This turned out to be very
// and even after replacing my caresian products by std::tupel I found no
// solution. The problem is that in these heterotypic arrays one has no
// for-loops over all components. Only costant expressions are allowed
// to access the components. If all types X1,...Xn had a common base class
// one could use pointers for the loops and one would have no problem.
// Taking this difficulties into account, one can appreciate the fact that
// the special case card(S1)=1 which we solved  by means of the functions
// Fn_m can be treated in a clear and elementary manner.

template <class X, class P1, class Y>
F<X,Y> F1<X,P1,Y>::operator()( Y (*f)(X const&, P1 const&) )const
{
   return F<X,Y>(new aux::Par1FncObj<X,P1,Y>(f,p1_));
}

////////////////////// class F1_1<> ////////////////////////////////////

template <class X, class P1, class Y>
class F1_1{
// as F1, but the first parameter is the function argument
    const X x_;
    typedef F1_1<X,P1,Y> Type;
    CPM_INVAR(F1_1)
public:
explicit F1_1( X const& x):x_(x){}
    F<P1,Y> operator()( Y (*f)(X const&, P1 const&) )const;
};

template <class X, class P1, class Y>
F<P1,Y> F1_1<X,P1,Y>::operator()( Y (*f)(X const&, P1 const&) )const
{
   return F<P1,Y>(new aux::Par1_1FncObj<X,P1,Y>(f,x_));
}
```

```
//////////////////////// class F2<> ////////////////////////////////////

template <class X, class P1, class P2, class Y>
class F2{ // functions with two parameters
    const P1 p1_;
    const P2 p2_;
    typedef F2<X,P1,P2,Y> Type;
    CPM_INVAR(F2)
public:
    F2( P1 const& p1, P2 const& p2):p1_(p1),p2_(p2){}
    F<X,Y> operator()(Y (*f)(X const&, P1 const&, P2 const&))const;
};

template <class X, class P1, class P2, class Y>
F<X,Y> F2<X,P1,P2,Y>::operator()(Y (*f)(X const&, P1 const&,
   P2 const&) )const
{
   using namespace aux;
   return F<X,Y>(new Par2FncObj<X,P1,P2,Y>(f,p1_,p2_));
}

//////////////////////// class F2_1<> ////////////////////////////////////

template <class X, class P1, class P2, class Y>
class F2_1{
// as F2, but the first parameter is the function argument
    const X x_;
    const P2 p2_;
    typedef F2_1<X,P1,P2,Y> Type;
    CPM_INVAR(F2_1)
public:
    F2_1( X const& x, P2 const& p2):x_(x),p2_(p2){}
    F<P1,Y> operator()(Y (*f)(X const&, P1 const&, P2 const&))const;
};

template <class X, class P1, class P2, class Y>
F<P1,Y> F2_1<X,P1,P2,Y>::operator()(Y (*f)(X const&, P1 const&,
   P2 const&) )const
{
   using namespace aux;
   return F<P1,Y>(new Par2_1FncObj<X,P1,P2,Y>(f,x_,p2_));
}

//////////////////////// class F2_2<> ////////////////////////////////////

template <class X, class P1, class P2, class Y>
class F2_2{
// as F2, but the second parameter is the function argument
    const X x_;
```

```
    const P1 p1_;
    typedef F2_2<X,P1,P2,Y> Type;
    CPM_INVAR(F2_2)
public:
    F2_2( X const& x, P1 const& p1):x_(x),p1_(p1){}
    F<P2,Y> operator()(Y (*f)(X const&, P1 const&, P2 const&))const;
};

template <class X, class P1, class P2, class Y>
F<P2,Y> F2_2<X,P1,P2,Y>::operator()(Y (*f)(X const&, P1 const&,
   P2 const&) )const
{
   using namespace aux;
   return F<P2,Y>(new Par2_2FncObj<X,P1,P2,Y>(f,x_,p1_));
}

////////////////////// class F3<> ////////////////////////////////////

template <class X, class P1, class P2, class P3, class Y>
class F3{ // functions with three parameters
    const P1 p1_;
    const P2 p2_;
    const P3 p3_;
    typedef F3<X,P1,P2,P3,Y> Type;
    CPM_INVAR(F3)
public:
    F3( P1 const& p1, P2 const& p2, P3 const& p3):
    p1_(p1),p2_(p2),p3_(p3){}
    F<X,Y> operator()(Y (*f)(X const&, P1 const&, P2 const&,
       P3 const&) )const;
};

template <class X, class P1, class P2, class P3, class Y>
F<X,Y> F3<X,P1,P2,P3,Y>::operator()(Y (*f)(X const&, P1 const&,
   P2 const&, P3 const&) )const
{
   using namespace aux;
   return F<X,Y>(new Par3FncObj<X,P1,P2,P3,Y>(f,p1_,p2_,p3_));
}
////////////////////// class F3_1<> ////////////////////////////////////

template <class X, class P1, class P2, class P3, class Y>
class F3_1{
// as F3, but the first parameter is the function argument
    const X x_;
    const P2 p2_;
    const P3 p3_;
    typedef F3_1<X,P1,P2,P3,Y> Type;
    CPM_INVAR(F3_1)
public:
```

```
    F3_1( X const& x, P2 const& p2, P3 const& p3):
    x_(x),p2_(p2),p3_(p3){}
    F<P1,Y> operator()(Y (*f)(X const&, P1 const&, P2 const&,
       P3 const&) )const;
};

template <class X, class P1, class P2, class P3, class Y>
F<P1,Y> F3_1<X,P1,P2,P3,Y>::operator()(Y (*f)(X const&, P1 const&,
   P2 const&, P3 const&) )const
{
   using namespace aux;
   return F<P1,Y>(new Par3_1FncObj<X,P1,P2,P3,Y>(f,x_,p2_,p3_));
}

////////////////////// class F3_2<> ////////////////////////////////////

template <class X, class P1, class P2, class P3, class Y>
class F3_2{
// as F3, but the second parameter is the function argument
    const X x_;
    const P1 p1_;
    const P3 p3_;
    typedef F3_2<X,P1,P2,P3,Y> Type;
    CPM_INVAR(F3_2)
public:
    F3_2( X const& x, P1 const& p1, P3 const& p3):
    x_(x),p1_(p1),p3_(p3){}
    F<P2,Y> operator()(Y (*f)(X const&, P1 const&, P2 const&,
       P3 const&) )const;
};

template <class X, class P1, class P2, class P3, class Y>
F<P2,Y> F3_2<X,P1,P2,P3,Y>::operator()(Y (*f)(X const&, P1 const&,
   P2 const&, P3 const&) )const
{
   using namespace aux;
   return F<P2,Y>(new Par3_2FncObj<X,P1,P2,P3,Y>(f,x_,p1_,p3_));
}

////////////////////// class F3_3<> ////////////////////////////////////

template <class X, class P1, class P2, class P3, class Y>
class F3_3{
// as F3, but the third parameter is the function argument
    const X x_;
    const P1 p1_;
    const P2 p2_;
    typedef F3_3<X,P1,P2,P3,Y> Type;
    CPM_INVAR(F3_3)
public:
```

```
    F3_3( X const& x, P1 const& p1, P2 const& p2):
    x_(x),p1_(p1),p2_(p2){}
    F<P3,Y> operator()(Y (*f)(X const&, P1 const&, P2 const&,
       P3 const&) )const;
};

template <class X, class P1, class P2, class P3, class Y>
F<P3,Y> F3_3<X,P1,P2,P3,Y>::operator()(Y (*f)(X const&, P1 const&,
   P2 const&, P3 const&) )const
{
   using namespace aux;
   return F<P3,Y>(new Par3_3FncObj<X,P1,P2,P3,Y>(f,x_,p1_,p2_));
}

////////////////////// class F4<> ////////////////////////////////////

template <class X, class P1, class P2, class P3, class P4, class Y>
class F4{ // functions with four parameters
    const P1 p1_;
    const P2 p2_;
    const P3 p3_;
    const P4 p4_;
    typedef F4<X,P1,P2,P3,P4,Y> Type;
    CPM_INVAR(F4)
public:
    F4( P1 const& p1, P2 const& p2, P3 const& p3, P4 const& p4):
    p1_(p1),p2_(p2),p3_(p3),p4_(p4){}
    F<X,Y> operator()(Y (*f)(X const&, P1 const&,
       P2 const&, P3 const&, P4 const&))const;
};

template <class X, class P1, class P2, class P3, class P4,
     class Y>

F<X,Y> F4<X,P1,P2,P3,P4,Y>::operator()(Y (*f)(X const&, P1 const&,
   P2 const&, P3 const&, P4 const&))const
{
   using namespace aux;
   return
   F<X,Y>(new Par4FncObj<X,P1,P2,P3,P4,Y>(f,p1_,p2_,p3_,p4_));
}

////////////////////// class F4_1<> ////////////////////////////////////

template <class X, class P1, class P2, class P3, class P4, class Y>
class F4_1{
// as F4, but the first parameter is the function argument
    const X x_;
    const P2 p2_;
    const P3 p3_;
```

```
    const P4 p4_;
    typedef F4_1<X,P1,P2,P3,P4,Y> Type;
    CPM_INVAR(F4_1)
public:
    F4_1( X const& x, P2 const& p2, P3 const& p3, P4 const& p4):
    x_(x),p2_(p2),p3_(p3),p4_(p4){}
    F<P1,Y> operator()(Y (*f)(X const&, P1 const&,
       P2 const&, P3 const&, P4 const&))const;
};

template <class X, class P1, class P2, class P3, class P4,
     class Y>

F<P1,Y> F4_1<X,P1,P2,P3,P4,Y>::operator()(Y (*f)(X const&, P1 const&,
   P2 const&, P3 const&, P4 const&))const
{
   using namespace aux;
   return
   F<P1,Y>(new Par4_1FncObj<X,P1,P2,P3,P4,Y>(f,x_,p2_,p3_,p4_));
}

//////////////////////// class F4_2<> ////////////////////////////////////

template <class X, class P1, class P2, class P3, class P4, class Y>
class F4_2{
// as F4, but the second parameter is the function argument
    const X x_;
    const P1 p1_;
    const P3 p3_;
    const P4 p4_;
    typedef F4_2<X,P1,P2,P3,P4,Y> Type;
    CPM_INVAR(F4_2)
public:
    F4_2( X const& x, P1 const& p1, P3 const& p3, P4 const& p4):
    x_(x),p1_(p1),p3_(p3),p4_(p4){}
    F<P2,Y> operator()(Y (*f)(X const&, P1 const&,
       P2 const&, P3 const&, P4 const&))const;
};

template <class X, class P1, class P2, class P3, class P4,
     class Y>

F<P2,Y> F4_2<X,P1,P2,P3,P4,Y>::operator()(Y (*f)(X const&, P1 const&,
   P2 const&, P3 const&, P4 const&))const
{
   using namespace aux;
   return
   F<P2,Y>(new Par4_2FncObj<X,P1,P2,P3,P4,Y>(f,x_,p1_,p3_,p4_));
}
```

```
//////////////////////// class F4_3<> ////////////////////////////////////

template <class X, class P1, class P2, class P3, class P4, class Y>
class F4_3{
// as F4, but the third parameter is the function argument
    const X x_;
    const P1 p1_;
    const P2 p2_;
    const P4 p4_;
    typedef F4_3<X,P1,P2,P3,P4,Y> Type;
    CPM_INVAR(F4_3)
public:
    F4_3( X const& x, P1 const& p1, P2 const& p2, P4 const& p4):
    x_(x),p1_(p1),p2_(p2),p4_(p4){}
    F<P3,Y> operator()(Y (*f)(X const&, P1 const&,
       P2 const&, P3 const&, P4 const&))const;
};

template <class X, class P1, class P2, class P3, class P4,
     class Y>

F<P3,Y> F4_3<X,P1,P2,P3,P4,Y>::operator()(Y (*f)(X const&, P1 const&,
   P2 const&, P3 const&, P4 const&))const
{
   using namespace aux;
   return
   F<P3,Y>(new Par4_3FncObj<X,P1,P2,P3,P4,Y>(f,x_,p1_,p2_,p4_));
}

//////////////////////// class F4_4<> ////////////////////////////////////

template <class X, class P1, class P2, class P3, class P4, class Y>
class F4_4{
// as F4, but the fourth parameter is the function argument
    const X x_;
    const P1 p1_;
    const P2 p2_;
    const P3 p3_;
    typedef F4_4<X,P1,P2,P3,P4,Y> Type;
    CPM_INVAR(F4_4)
public:
    F4_4( X const& x, P1 const& p1, P2 const& p2, P3 const& p3):
    x_(x),p1_(p1),p2_(p2),p3_(p3){}
    F<P4,Y> operator()(Y (*f)(X const&, P1 const&,
       P2 const&, P3 const&, P4 const&))const;
};

template <class X, class P1, class P2, class P3, class P4,
     class Y>
```

```
F<P4,Y> F4_4<X,P1,P2,P3,P4,Y>::operator()(Y (*f)(X const&, P1 const&,
   P2 const&, P3 const&, P4 const&))const
{
   using namespace aux;
   return
   F<P4,Y>(new Par4_4FncObj<X,P1,P2,P3,P4,Y>(f,x_,p1_,p2_,p3_));
}

#endif // CPM_Fn

} // namespace

#endif
```

17 cpmfo.h

```
//? cpmfo.h
//? Status of work 2023-10-20.
//?
//?  ...


#ifndef CPM_FO_H_
#define CPM_FO_H_
/*

Purpose: Define classes describing function-like objects with
   order operations

*/
#include <cpmf.h>

namespace CpmFunctions{

// Base class for classes describing binary operations

namespace auxiliary{

template <class X, class Y>
class BINOPFUO: public FncObj<X,Y>{ // BINOP stands for Binary
// Operation
protected:
   const F<X,Y>  f1;
   const F<X,Y>  f2;

public:
   BINOPFUO(  const F<X,Y> & f1In, const F<X,Y> & f2In ):
   f1(f1In),f2(f2In){}
};

// Base class for classes describing monary operations

template <class X, class Y>
class MONOPFUO: public FncObj<X,Y>{

protected:
   const F<X,Y>  f1;

public:

    MONOPFUO( const F<X,Y> &  f1In): f1(f1In){}
};
```

```
// order-related operations
// Now it will be assumed that Y is ordered ( Y < Y and Y > Y)

// class MINFUO<>

template <class X, class Y>
class MINFUO: public BINOPFUO<X,Y>{

public:
   Y operator()(const X& x)const
   {
      Y y1=f1(x);
      Y y2=f2(x);
      return (y1 < y2 ? y1 : y2);
   }

   MINFUO(const F<X,Y> & f1In, const F<X,Y> & f2In):
      BINOPFUO<X,Y>(f1In,f2In){}
};

// class MAXFUO<>

template <class X, class Y>
class MAXFUO: public BINOPFUO<X,Y>{

public:
   Y operator()(const X& x)const
   {
      Y y1=f1(x);
      Y y2=f2(x);
      return (y1 > y2 ? y1 : y2);
   }

   MAXFUO(const F<X,Y>& f1In, const F<X,Y>& f2In):
      BINOPFUO<X,Y>(f1In,f2In){}
};

// setting function values

template <class X, class Y>
class SETVALUEFUO : public FncObj<X,Y>{   // changing function values
   // assumes that '==' is defined in X

   const F<X,Y>  f1;
   const X xs;
   const Y ys;

public:
   Y operator()(const X& x)const
   {
```

```
      if (x==xs) return ys; else return f1(x);
   }

   SETVALUEFUO(const F<X,Y>& g1, const X& x1, const Y& y1):
   f1(g1),xs(x1),ys(y1){}
};

} // auxiliary

////////////////////////////// class Fo<X,Y> ////////////////////////////////

template <class X, class Y>
class Fo: public F<X,Y> { // version of F with order-related operations

   typedef Fo<X,Y> Type;

public:

   Fo(const F<X,Y>& g):F<X,Y>(g){}
      // upcast constructor

   Fo(const Fo<X,Y>& g):F<X,Y>(g){}
      // copy constructor

   Fo(void):F<X,Y>(){}
      // default constructor

   Fo(FncObj<X,Y>* fop):F<X,Y>(fop){}
      // constructor from pointers to FncObj

// Construction from function pointers

   Fo( Y (*f)(const X& )):F<X,Y>(f){}
   Fo( Y (*f)(X)):F<X,Y>(f){}

   Fo( std::function<Y(X)> f):F<X,Y>(f){}

// Construtor for constant function

   explicit Fo(const Y& y1):F<X,Y>(y1){}
      // Functionality:
      // Y y=...; F<X,Y> f(y); X x=...; f(x)==y;
      // last statement always evaluates to 1, independent of x

   void setValue(const X& x0, const Y& y0);
      // non-constant member function which modifies the function value
      // at x=x0 to become y0
   virtual F<X,Y>* clone()const{ return new Fo(*this);}
   virtual Word nameOf()const
   {
```

```
      Word wi="Fo<";
      Word wx=CpmRoot::Name<X>()(X());
      Word wy=CpmRoot::Name<Y>()(Y());
      return wi&wx&","&wy&">";
   }
   CPM_ORDER
};

template <class X, class Y>
   // assumption Y < Y defined

Fo<X,Y> inf( const Fo<X,Y>& f1, const Fo<X,Y>& f2)
{
   using namespace auxiliary;
   return Fo<X,Y>(new MINFUO<X,Y>(f1,f2));
}

template <class X, class Y>
   // assumption Y < Y defined

Fo<X,Y> sup( const Fo<X,Y>& f1, const Fo<X,Y>& f2)
{
   using namespace auxiliary;
   return Fo<X,Y>(new MAXFUO<X,Y>(f1,f2));
}

// Order interface consistent but in a sense trivial.
// No assumtions on X,Y
// Does not take into account the function values but
// only the storage position.

template <class X, class Y>
Z Fo<X,Y>::com(const Fo<X,Y>& f)const
{
   return F<X,Y>::com(f);
}

} // namespace

#endif
```

18 cpmfr.h

```
//? cpmfr.h
//? Status of work 2023-10-20.
//?
//?  ...


#ifndef CPM_FR_H_
#define CPM_FR_H_
/*
   Purpose: Define classes describing function-like objects with
      arithmetic operations

*/

#include <cpmfa.h>
#include <cpmvo.h>

namespace CpmFunctions{

   using namespace CpmStd;

   using CpmRoot::R;
   using CpmRoot::Word;

template <class X, class Y>

class Fr: public Fa<X,Y> { // version of Fa with rich interface

   static Z imax;
   static Z cpl;
      // means for implementing order and equality
      // by comparison of function values ( actually all
      // values would be needed for being mathematically exact)
      // Another way of looking at the matter is that imax and cpl define
      // an equivalence relation on Fr<X,Y> and order and equality are
      // defined exactly on the set of equivalunce classes. The larger
      // imax and cpl are, the smaller the equivalence classes

public:

   typedef Fr<X,Y> Type;

   Fr(void):Fa<X,Y>(){}
      // default constructor

   Fr(const F<X,Y>& g):Fa<X,Y>(g){}
      // upcast constructor
```

```
   Fr(const Fo<X,Y>& g):Fa<X,Y>(g){}
      // upcast constructor

   Fr(const Fa<X,Y>& g):Fa<X,Y>(g){}
      // upcast constructor, note Fa = Fa

   Fr(const Fr<X,Y>& g):Fa<X,Y>(g){}
      // copy constructor

   Fr(FncObj<X,Y>* fop):Fa<X,Y>(fop){}
      // constructor from pointers to FncObj

// Construction from function pointers

   Fr( Y (*f)(const X& )):Fa<X,Y>(f){}
   Fr( Y (*f)(X)):Fa<X,Y>(f){}

// Construtor for constant function

   Fr(const Y& y1):Fa<X,Y>(y1){}
      // takes always the value y1, irrespective of x

   Fr(CpmArrays::V<X> const& xl, CpmArrays::V<Y> const& yl);
      // Construction from a list of x-values and a list of corresponding
      // y-values. Makes a staircase-function since real interpolation
      // assumes R-related operations in Y which we don't want to assme
      // here.

   virtual F<X,Y>* clone()const{ return new Fr(*this);}

   CPM_IO
   CPM_TEST_X
   CPM_DESCRIPTORS
   CPM_CONJUGATION

   Y operator()(X const& x)const;
      // redefinition with better error handling

   R compare(Word top, Fr<X,Y> const& rhs)const;

};

template <class X, class Y>
Y Fr<X,Y>::operator()(const X& x)const
      // notice the * operator from class P<>
{
   using CpmArrays::P;
   try{
      return (P< FncObj<X,Y> >::operator()())(x);
```

```
   }
   catch(...){
      CpmRoot::Word wx=CpmRoot::Name<X>()(X());
      Y y=Y();
      CpmRoot::Word wy=CpmRoot::Name<Y>()(y);
      cpmerror("trouble in Y Fa<X,Y>::operator()(const X& x)const\
      for X type "&wx&" and Y type "&wy);
      return y; // to avoid warning
   }
}

// helper classes for implementation

namespace aux{
template <class X, class Y>
class CONJFUO : public FncObj<X,Y>{
   const Fr<X,Y> f;
public:
   CONJFUO(const Fr<X,Y>& f_):f(f_){}
   Y operator()(const X& x)const
   { return CpmRoot::conT<Y>(f(x));}
};

template <class X, class Y>
class RANDFUO : public FncObj<X,Y>{
   const Z i;
   const Z tvs;
   const Z memory;
public:
   RANDFUO(const Z& i_, Z tvs_, Z mem_ ):i(i_),tvs(tvs_),
      memory(mem_){}
   Y operator()(X const& x)const
      // never invoke ran() since then the random functions
      // depend on history
   {
   // would make operator() a mutating operator
   // this would have rather opaque implications
      Z iAct=(i==0 ? memory : i);
      Y ys;
      ys=CpmRoot::testT<Y>(ys,tvs);
      ys=CpmRoot::ranT<Y>(ys,iAct);
      Z j=CpmRoot::hashT<X>(x);
      if (j==0)
         return ys;
      else
         return CpmRoot::ranT<Y>(ys,j);
         // notice that for j==0 ran behaves not as a function
         // (value depends on how often the routine was called)
   }
};
```

```
template <class X, class Y>
class IPOLFUO: public FncObj<X,Y>{
   // IPOL stands for interpolation. However, here we don't really
   // interpolate, we use staircase approximation instead.
   // This needs no assumptions concerning Y.
   const CpmArrays::V<X> xl;
   const CpmArrays::V<Y> yl;
public:
   Y operator()(const X& x)const;
   IPOLFUO( const CpmArrays::V<X>& xlIn, const CpmArrays::V<Y>& ylIn ):
      xl(xlIn),yl(ylIn){}
};

template <class X, class Y>
Y IPOLFUO<X,Y>::operator()(X const& x)const
{
   Z i,n=xl.dim();
   Word loc("IPOLFUO<X,Y>::operator()");
   cpmassert(yl.dim()>=n,loc);
   CpmArrays::Vo<R> dis(n);
   for (i=1;i<=n;i++){
      X xd=x-xl[i];
      dis[i]=CpmRoot::absT<X>(xd);
   }
   Z j=dis.indInf();
   return yl[j];
}

} // aux

// end of helper classes

template <class X, class Y>
Z Fr<X,Y>::imax=8;

template <class X, class Y>
Z Fr<X,Y>::cpl=8;

template <class X, class Y>
Fr<X,Y>::Fr(const CpmArrays::V<X>& xl,
   const CpmArrays::V<Y>& yl)
   // a rather crude interpolation based on weighted means
   // of function values from the list
{
   *this=Fr(new aux::IPOLFUO<X,Y>(xl,yl));
}

template <class X, class Y>
Fr<X,Y> Fr<X,Y>::con(void)const
```

```
{
   return Fr<X,Y>(new aux::CONJFUO<X,Y>(*this));
}

template <class X, class Y>
Fr<X,Y> Fr<X,Y>::ran(Z i)const
{
   static Z mem=37;
   if (i==0) mem++;
   return Fr<X,Y>(new aux::RANDFUO<X,Y>(i,cpl,mem));
}

template <class X, class Y>
Word Fr<X,Y>::nameOf(void)const
{
   Word wi="Fr<";
   X x; Y y;
   Word wx=CpmRoot::Name<X>()(x);
   Word wy=CpmRoot::Name<Y>()(y);
   return wi&wx&","&wy&">";
}

template <class X, class Y>
Word Fr<X,Y>::toWord(void)const
{
   ostringstream ost;
   prnOn(ost);
   return Word(ost.str());
}

template <class X, class Y>
Fr<X,Y> Fr<X,Y>::test(Z i)const
{
   Y y0;
   Y yt=CpmRoot::testT<Y>(y0,i);
   Fr<X,Y> res(yt);
   imax=i;
   cpl=i;
   return res;
}

template <class X, class Y>
bool Fr<X,Y>::prnOn(ostream& out)const
{
   out<<endl<<"// printing of "<<nameOf()<<endl;
   out<<endl<<"// Number of (x,y)-pairs: "<<endl;
   out<<imax<<endl;
   X x;
   Y y;
   X xs=CpmRoot::testT<X>(x,cpl);
```

```
   for (Z i=1;i<=imax;i++){
      x=CpmRoot::ranT<X>(xs,i);
      y=(*this)(x);
      bool b;
      out<<endl<<"// value pair number "<<i<<" of "<<imax;
      out<<endl<<"// x = "<<endl;
      b=CpmRoot::prnT<X>(x,out);
      if (!b) return false;
      out<<endl<<"// y = "<<endl;
      b=CpmRoot::prnT<Y>(y,out);
      if (!b) return false;
   }
   return true;
}

template <class X, class Y>
bool Fr<X,Y>::scanFrom(istream& in)
// reading value pairs and creating an interrpolated function
// from a file that may contain comments
{
   static const Z readLimit=512;
   Z nList;
   if (!CpmRoot::read(nList,in)) return false;
   cpmassert(nList>0,"Fr<X,Y>::scanFrom");
   cpmassert(nList<=readLimit,"Fr<X,Y>::scanFrom");
   CpmArrays::V<X> xl(nList);
   CpmArrays::V<Y> yl(nList);
   for (Z i=1;i<=nList;i++){
      if (!CpmRoot::scanT<X>(xl[i],in)) return false;
      if (!CpmRoot::scanT<Y>(yl[i],in)) return false;
   }
   *this=Fr<X,Y>(xl,yl);
   return true;
}

template <class X, class Y>
Z Fr<X,Y>::hash(void)const
// returns a key value to *this. This should be not
// expensive to calculate. Therfore the number of function
// evaluation is restricted by imax
{
   Z h=0;
   X x,xs;
   Y y;
   xs=CpmRoot::testT<X>(x,cpl);
   for (Z i=1;i<=imax;i++){
      x=CpmRoot::ranT<X>(xs,i);
      y=(*this)(x);
      h^=CpmRoot::hashT<Y>(y);
         // bit operation since += may give overflow
```

```
   }
   return h;
}

template <class X, class Y>
R Fr<X,Y>::absSqr(void)const
// returns as an absolute value a mean absolute value of function values
// where the arguments are chosen as a random sequence.
{
try{
   X x,xs;
   Y y;
   xs=CpmRoot::testT<X>(x,cpl);
   R res=0;
   for (Z i=1;i<=imax;i++){
      x=CpmRoot::ranT<X>(xs,i);
      y=(*this)(x);
      R a=CpmRoot::absT<Y>(y);
      res+=a*a;
   }
   cpmassert(imax>0,"Fr<X,Y>::absSqr");
   res/=imax;
   return cpmsqrt(res);
}
catch(...){
   cpmerror("trouble in R Fr<X,Y>::abs(void)const");
   return 0; // to avoid warning
}
}

template <class X, class Y>
R Fr<X,Y>::abs(void)const
{
   return cpmsqrt(absSqr());
}

template <class X, class Y>
R Fr<X,Y>::dis(const Fr<X,Y>& x)const
{
   Fr<X,Y> diffVector=*this;
   diffVector-=x;
   R da=diffVector.abs();
   R xa=x.abs();
   R ya=abs();
   return CpmRoot::disDefFun(xa,ya,da);
}

} // namespace

#endif
```

19 cpmgreg.h

```
//? cpmgreg.h
//? Status of work 2023-10-20.
//?
//?  ...


#ifndef CPM_GREG_H_
#define CPM_GREG_H_

/*

    Functions for handling Gregorian calendar dates

*/

#include <cpmangle.h>

namespace CpmTime{ // Quantities related to calendar, date, time, clock

using namespace CpmStd;
//using CpmSystem::R_;
using CpmRoot::Z;
using CpmRoot::R;
using CpmRoot::Word;
using CpmGeo::Angle;
using CpmSystem::OFileStream;

//////////////////////////// enum TimeScale ////////////////////////////////

enum TimeScale {TD, UT}; // presenly not used in this general-purpose
   // (non-astronomical) version of Greg in order not to need the more
   // astronomical files.

   // TD = Temps dynamique, UT = universal time

////////////////////////////// struct TimeStyle ////////////////////////////

struct TimeStyle{ // how to interprete time with respect to the globe
   TimeScale tsc;
   Angle lambdaOfTimeMeridian;
   TimeStyle(void):tsc(UT),lambdaOfTimeMeridian(Angle()){}
   TimeStyle(TimeScale s, const Angle& a):tsc(s),
      lambdaOfTimeMeridian(a){}
};

class Greg;
R operator -(const Greg& d1, const Greg& d2);
```

```
////////////////////////////// class Greg ////////////////////////////////

class Greg{  // time, date, and Gregorian Calendar
   // An instance of class Greg determines a point in time
   // coded according to the Gregorgian calendar (and according to the
   // Julian calendar for dates prior to the introduction of the
   // Gregorian calendar). No reference to a particular time scale (UT
   // or TD) or to a time zone (e.g. MEZ or MESZ ) is implied

private:

   static Z yearOffset;
      // The 'century' which is understood for 2 decimal year input
      // e.g. 2000 for dates in the 21th century
      // May be changed via setCentury(). Presently initialized as 2000.

   static const R mjdOffset;
      // JD=MJD+mjdOffset, mjdOffset=2400000.5

protected:

// primary (independent) information
   Z y;         // year
   Z m;         // month
   Z d;         // day
   Z h;         // hour
   Z min;       // minute
   R s;         // second

// secondary (dependent) information (function of the primary
// information)
   R mjd;         // analog to the MJD without reference to a concrete
               // time scale such as UT or TD
   void update(void);
      // updates dependent information (i.e. mjd)

   void makeDate(void);
      // adjusts the rest of data members to present value of mjd

public:

   typedef Greg Type;
   CPM_IO
   CPM_NAM_V(Greg)
   virtual Greg* clone()const{ return new Greg(*this);}

// parts of the day (the unit) which are often needed

   static const R hour;
```

```
   static const R minute;
   static const R second;

   static R mjd_(Z y, Z m, Z d);
      // Modified from Montenbruck, Pfleger: Astronomy on the Personal
      // Computer,
      // 4th edition, p. 15
      // Notice that mjd_==0 <==> y=1858 m=11 d=17

   static Z trunc(R x);
      // See O. Montenbruck, Grundlagen der Ephemeridenrechnung S. 49

   static void caldat(Z mjd, Z& yy, Z& mm, Z& dd);
      // From Montenbruck, Pfleger: Astronomie mit dem Personal Computer
      // p. 13
      // Translated from PASCAL to C

   void setCentury( Z century)
      // setCentury(20) sets the value to be added to a short
      // year, such as 95, to 1900, thus resulting in 1995.
      // Notice that this coincides with the conventional enumeration
      // of centuries.
      // Century should be positive, we are not interested in using
      // Gregorian Calendar in times much prior to its introduction !
      {
         yearOffset=(century-1)*100;
      }

   Greg(void);
      // Default constructor, creates 2000-1-1 0:0:0

   Greg(Z year, Z mm, Z dd);
      // mm has to be in the range [1,12]
      // dd is arbitrary (e.g. 94-12-31 = 95-1-0 = 95-2-(-31) )
      // if year<99 it will be understood as year+yearOffset.

   explicit Greg(R date){ setDate(date,false);}
     // date is assumed to be of the form year.mmdd
     // where year is the integer year number e.g. 2006 or
     // 1905 (no shortened numbers like 50 for 1950
     // or 6 for 2006 !) Only the first 4 decimal places get
     // used (no fractions of a day).

   Greg(R date, R watch){ setDate(date,false);setWatch(watch);}
      // Creates a Greg instance from date=YYYY.MMDD and
      // watch=HH.MMSSsss... (see setWatch() for details).

// getting calendar data, always clear from code

   Z getYear(void)const{ return y;}
```

```
   Z getMonth(void)const{ return m;}

   Z getDay(void)const{ return d;}

   Z getHour(void)const{ return h;}

   Z getMinute(void)const{ return min;}

   R getSecond(void)const{ return s;}

   R getMJD(void)const{ return mjd;}

   R getJD(void)const{ return mjd+mjdOffset;}

   Word getWeekDay(void)const;

   R yearsSince1900(void)const{ return (mjd-33281.)/R(365.2422)+50.;}
   R yearsSince2000(void)const{ return (mjd-51544.)/365.2422;}
   R years(void)const{ return yearsSince2000()+2000;}
      // Convenient unit for time axes which go over longer spans of time
      // and which are related to history.

   Word getDate(bool verbose=true)const;
      // "yyyy-mm-ddThh:mm:ssZ" ( ISO6801 time stamp )
      // a compact representation of a date in which the newline commands
      // of printOut() would cause trouble

   Word getCode(Z n=14)const;
      // returns a characterization of a point in time which
      // is compact enough to be used as an appendix to file names.
      // It consists of the n trailing characters of the string
      // yyyymmddhhmmss. So the default value for n means that
      // a condensed form of the date will be given. For
      // n=8 one gets a characterisation of the time within the
      // present day.

   Word getCodeWord(Z n, Z p=5);
      // returns a sequence of n digits which is determined by the
      // instance of call and may be used as random signature
      // method: multiply mjd by 10^p, form the integer part
      // and take the last n digits of the result. Thus, the larger
      // p, the faster the result changes with time. The frequency
      // of change is 10^p per day. Thus p=5 means 10^5 changes
      // in 86400 s, i.e. 1.157 changes per second. This is a
      // reasonable time resolution for differentiating between
      // susequent runs of a program.

// setting Greg according to numbered days input (Julian date and
//   modified Julian date
```

```
   void setMJD(R mjd){ this->mjd=mjd; makeDate();}
      // adjusts *this according to MJD input.
      // In constructor form, this function would enable automatic
      // conversion from Rd to Greg and finally would lead to
      // Greg + Greg being defined (since Greg + R is defined). This
      // is not desirable, only Greg - Greg is OK.

   void setMJD(R mjd, const TimeStyle& ts);
      // adjusts *this according to MJD input. The correspondence
      // between what the function does and the values
      // UT or TD of the component ts.tsc is logical if we
      // consider mjd as referring to TD. Then ts.tsc=UT means that
      // a deltaT subtraction has to be done to get a result in
      // UT or the civil time of a time zone specified by
      // ts.lambdaOfTimeMeridian

   void setJD(R jd){ mjd=jd-mjdOffset; makeDate();}
      // adjusts *this according to JD input.

   R toMJDTD(const TimeStyle& ts)const;
      // returns MJD (as TD) from the Greg instance *this where ts
      // tells how to interprete *this
      // (e.g. as civil time of a particular time zone)

   void now(void);
      // sets *this according to the computer's idea of Greenwich
      // mean time (Z, GMT0) at the moment of function call. Notice that
      // shifting this point in time according to needs can easily be done
      // by the arithmetics of Greg, so that no shift argument is needed
      // for this function.
      // e.g.
      // TimeStyle ts=...;
      // Greg g;
      // g.now();
      // g+=10*Greg::minute;
      // R t=g.toMJDTD(ts);
      // Earth e;
      // e.update(t);
      // will represent the state of the earth 10 minutes after the
      // function call if ts properly reflects the characteristics of the
      // computer clock

   void now_(void){ now();}
      //: now
      // See function now(), for which this is the proper naming

friend R operator -(const Greg& d1, const Greg& d2);
      // time span in days, notice that a corresponding + operator
        // makes no sense
```

```
   Greg& operator +=(R t);
      // shifting a point in time by t days

   Greg& nextMonth_();
      // shifting a point in time to the same day, minute, second
      // but one month in calander later (shift may be 28 days,
      // 30 days or 31 days according to the calender month we are
      // in). Name ends in '_' since this is a mutating function.

   Greg& nextDay_(){ mjd+=1.; makeDate(); return *this;}
      // shifting a point in time to by one day
      // in). Name ends in '_' since this is a mutating function.

   Greg& operator -=(R t);
      // shifting a point in time by -t days

   Greg operator +(R t){ Greg res=*this; return res+=t;}

   Greg operator -(R t){ Greg res=*this; return res-=t;}

   void setDate(R date, bool yShort=true);
      // Sets the values of y, m, d from date=Year.MMDD .
      // If yShort=true and Year<99 the integer number
      // yearOffset will be added to y.

   void setWatch(R watch);
      // Sets the values of h, min, s from watch=HH.MMSSsss...
      // where SS.sss... is a number of seconds possibly together with
      // decimal fractions of a second.

   bool readFormatted(istream& inStream);
   bool writeFormatted(ostream& outStream)const;
   bool writeForm(OFileStream& ofs)const{ return writeFormatted(ofs());}
};

} // namespace

#endif
```

20 cpmgreg.cpp

```
//? cpmgreg.cpp
//? Status of work 2023-10-20.
//?
//?  ...


// Inserting diagnostic messages in function bodies of this file may causes
// problems.
// Functions may be called in a situation where the log file streams are
// not yet created.

#include <time.h>
#include <iomanip>

#if !defined(WIN32)
   #include <stdlib.h>
#endif

#include <cpmgreg.h>

using CpmRoot::Z;
using CpmRoot::R;
using CpmRoot::Word;
using CpmRoot::toWord;
using CpmRoot::toDouble;

using CpmTime::Greg;
using CpmTime::TimeStyle;

using namespace CpmStd;

R Greg::mjd_(Z y, Z m, Z d)
{
   if (m<=2){
      m+=12;
      y-=1;
   }
   R a=10000.*y+100.*m+d;
   Z b;
   if (a<=15821004.1){
      b=-2+(y+4716)/4-1179;
   }
   else{
      b=y/400-y/100+y/4;
   }
   a=365.0*y-679004.0;
   R res=a+b+floor(30.6001*(m+1))+d;
```

```
   return res;
}

Z Greg::trunc(R x)
{
   return CpmRoot::toZ(x,true);
}

void Greg::caldat(Z mjd, Z& yy, Z& mm, Z& dd)
{
   Z b,d,f;
   R jd,jd0,c,e;
   jd=2400000.5+mjd;
   jd0=trunc(jd+0.5);
   if (jd0<2299161.0){
      b=0;
      c=jd0+1524.0;
   }
   else{
      b=trunc((jd0-1867216.25)/36524.25);
      c=jd0+(b-trunc(b/4))+1525.0;
   }
   d=trunc((c-122.1)/365.25);
   e=365.0*d+trunc(d/4);
   f=trunc((c-e)/30.6001);
   dd=trunc(c-e+0.5)-trunc(30.6001*f);
   mm=f-1-12*trunc(f/14);
   yy=d-4715-trunc((7+mm)/10);
}

Z Greg::yearOffset=2000;

const R Greg::mjdOffset=2400000.5;

const R Greg::hour=1./24;

const R Greg::minute=Greg::hour/60;

const R Greg::second=Greg::minute/60;

void Greg::update(void)
{
   mjd=mjd_(y,m,d)+R(h)*hour+R(min)*minute+s*second;
}

void Greg::makeDate(void)
{
   Z mjdInt=cpmtoz(mjd);
   R rest=mjd-mjdInt;
   caldat(mjdInt,y,m,d); // now y,m,d are valid
```

```
   R h1=rest*24;
   h=trunc(h1);
   R min1=(h1-h)*60;
   min=trunc(min1);
   s=(min1-min)*60;
}

Greg::Greg(void)
{
   y=2000;
   m=1;
   d=1;
   h=0;
   min=0;
   s=0.;
   mjd=51544;
}

Greg::Greg(Z yy, Z mm, Z dd):y(yy),m(mm),d(dd),h(0),min(0),s(0)
{
   if (m<1 || m>12) cpmerror("Greg","month must be 1,...,12");
   update();
   if (d<1 || d>28) makeDate();
}

Greg& Greg::nextMonth_()
{
   m++;
   if (m>12){
      y++;
      m=1;
   }
   update();
   if (d>28) makeDate(); // this makes a valid date even when called
   // e.g. for March 31, where April 31 would not be a correct
   // date.
   return *this;
}

bool Greg::prnOn(ostream& str)const
{
   cpmwat;
   cpmp(y);
   cpmp(m);
   cpmp(d);
   cpmp(h);
   cpmp(min);
   cpmp(s);
   return true;
}
```

```
bool Greg::scanFrom(istream& str)
{
   cpms(y);
   cpms(m);
   cpms(d);
   cpms(h);
   cpms(min);
   cpms(s);
   update();
   return true;
}

Word Greg::getDate(bool verbose)const
{
   using std::setw;
   using std::setfill;
   ostringstream ost;
   if (verbose) ost<<"yyyy-mm-ddThh:mm:ss ";
   ost<<setw(4)<<setfill('0')<<y<<"-"
      <<setw(2)<<setfill('0')<<m<<"-"
      <<setw(2)<<setfill('0')<<d<<"T"
      <<setw(2)<<setfill('0')<<h<<":"
      <<setw(2)<<setfill('0')<<min<<":"
      <<setw(2)<<setfill('0')<<toDouble(s)<<"Z";
   return Word(ost.str());
}

Word Greg::getCode(Z n)const
{
   using std::setw;
   using std::setfill;
   ostringstream ost;
   ost<<setw(4)<<setfill('0')<<y
      <<setw(2)<<setfill('0')<<m
      <<setw(2)<<setfill('0')<<d
      <<setw(2)<<setfill('0')<<h
      <<setw(2)<<setfill('0')<<min
      <<setw(2)<<setfill('0')<<toDouble(s);
   return Word(ost.str()).tail(n);
}

bool Greg::readFormatted(istream& inStream)
{
   cout<<endl<<
   "Entering a point in time in terms of the Gregorian calendar";
   R d,h;
   cout<<"\n Enter the date in the format yy.mmdd or yyyy.mmdd:  ";
   inStream>>d;
   cout<<"\n Enter the time in the format hh.mmss:  "; inStream>>h;
```

```
   setDate(d);
   setWatch(h);
   update();
   return true;
}

bool Greg::writeFormatted(ostream& str)const
{
   Word yS=Word::write(y,4);
   Word mS=Word::write(m,2);
   Word dS=Word::write(d,2);
   Word GregS=yS&"-"&mS&"-"&dS&" ";
   Word hS=Word::write(h,2);
   Word minS=Word::write(min,2);
   Word sS=toWord(s,"%2.1f");
   Word watchS=hS&":"&minS&":"&sS&" Z";
   Word res=GregS&watchS;
   str<<res;
   //return (str!=0);
   if (!str) return false;
   else return true;
}

R CpmTime::operator - (const Greg& g1, const Greg& g2)
{
   return g1.mjd-g2.mjd;
}

Greg& Greg::operator +=(R t)
{
   mjd+=t;
   makeDate();
   return *this;
}

Greg& Greg::operator -=(R t)
{
   mjd-=t;
   makeDate();
   return *this;
}

void Greg::setDate(R date, bool yShort)
{
   const R eps=0.0001;
   Z sig=(date >=0. ? 1 : -1);
   date*=sig; // now date is positive
   y=cpmtoz(date);
   date-=y; // fractional part of the absolute value of date
   y*=sig; // now the year carries its sign
```

```
   if (yShort){
      if (y<=99) y+=yearOffset;
   }
   date*=100;
   m=cpmtoz(date+eps);
   date-=m;
   date*=100;
   d=cpmtoz(date+eps);
   h=0;
   min=0;
   s=0.;
   update(); // added 99-11-27
}

void Greg::setWatch(R watch)
{
   const R eps=0.0001;
   if (watch<0) cpmerror("Greg::setWatch(R_): argument should be >=0");
   h=cpmtoz(watch+eps);
   watch-=h; // fractional part of the absolute value of watch
   watch*=100;
   min=cpmtoz(watch+eps);
   watch-=min;
   watch*=100;
   s=watch;
   update(); // added 99-11-27
}

Word Greg::getWeekDay(void)const
{
   Z weekDay=cpmtoz(getMJD());
   weekDay=weekDay%7;
   Word res;
   switch (weekDay){
      case 0: res="Wendsday";break;
      case 1: res="Thursday";break;
      case 2: res="Friday";break;
      case 3: res="Saturday";break;
      case 4: res="Sunday";break;
      case 5: res="Monday";break;
      case 6: res="Tuesday";break;
   }
   return res;
}

void Greg::now( void)
{
   static bool firstrun=true;
   if (firstrun){
   #if defined(WIN32)
```

```
      _putenv("TZ=GMT0");
      _tzset();
   #else
      char tz[7]={'T','Z','=','G','M','T','0'};
      putenv(tz);
      tzset(); // needs stdlib.h
   #endif
      firstrun=false;
   }
   time_t t;
   time(&t);
   tm* all;
   all=gmtime(&t);
   // due to putenv() this gives the normal conversion
      // without a time zone shift
   s=all->tm_sec;
   min=all->tm_min;
   h=all->tm_hour;
   d=all->tm_mday;
   m=1+all->tm_mon;
   y=all->tm_year; // comes out as 99 for 1999, and 100 for 2000

   if (y>=90 /* && y<=99 */) y+=1900; // Windows NT gives 100 in 2000
      // this means that the rule to add 1900 is not changed
    update();
    if (cpmverbose>=4){
        cout<<endl<<"s="<<s;
        cout<<endl<<"min="<<min;
        cout<<endl<<"h="<<h;
        cout<<endl<<"d="<<d;
        cout<<endl<<"m="<<m;
        cout<<endl<<"y="<<y;
        cout<<endl<<"mjd="<<mjd;
    }
}

R Greg::toMJDTD(const TimeStyle& ts)const
{

   R shift=ts.lambdaOfTimeMeridian.toDeg(0);
    shift/=360.; // now it is in days
    R res=mjd;
    res+=shift; // now res can be compared with MJD of UT or TD
        // i.e. there is no longer a time zone contribution
    return res;
}

void Greg::setMJD(R mjdI, const TimeStyle& ts)
{
    mjd=mjdI;
```

```
   R shift=ts.lambdaOfTimeMeridian.toDeg(0);
    shift/=360.; // now shift is in days
    mjd-=shift; // now referring to the local meridian
    makeDate();
}

Word Greg::getCodeWord(Z n, Z p)
{
   now();
   R f=pow(10.,p);
   Z cod=cpmtoz(mjd*f);
   Word res=cpmwrite(cod);
   return res.tail(n);
}
```

21 cpminterfaces.h

```
//? cpminterfaces.h
//? Status of work 2023-10-20.
//?
//?  ...


#ifndef CPM_INTERFACES_H_
#define CPM_INTERFACES_H_
/*

   Description: Declaration macros for interfaces for
      CPM classes. No macro argument, assuming that the
      client class will typedef her name as Type; some macros
      assume that the client class also defines ScalarType
      and CloneBase

*/

#include <cpmnumbers.h>
   // includes cpmdefinitions.h
#include <cpmbasicinterfaces.h>
#include <cpmmpi.h>
   // contains valid (though trivial) declarations even for
   // CPM_USE_MPI not defined

// Notice that using types Z and R in macros is only safe as
// CpmRoot::Z and CpmRoot::R since they might be used in a context
// where a corresponding using directive is not available.

// defining member functions send and rec(ieve) for communication between
// processes if the message passing interface library is being used
// (for parallel programming)
//using CpmMPI::Com;
// Definition of send builds on function prnOn, definition of rec builds
// on function scanFrom. The macros CPM_MPI_... will always be used after
// macros which declare these functions prnOn and scanFrom.

#if defined(CPM_USE_MPI)

   #define CPM_MPI_0\
   bool send(Z dest, CpmRoot::Z tag, CpmMPI::Com comm=CpmMPI::Com()\
   )const\
      { \
         std::ostringstream ost;\
         CpmRoot::Z wrtPrcMem=CpmRoot::wrtPrc;\
         CpmRoot::wrtPrc=18;\
         bool b=prnOn(ost);\
```

```
         CpmRoot::wrtPrc=wrtPrcMem;\
         return b ? comm.sendStr(ost.str(),dest,tag): false;\
      }

   #define CPM_MPI_I\
      bool rec(CpmRoot::Z source, CpmRoot::Z tag, CpmMPI::Com\
       comm=CpmMPI::Com())\
      {\
         std::string s;\
         bool res=comm.recStr(s,source,tag);\
         std::istringstream ist(s);\
         bool b=scanFrom(ist);\
         return b ? res : false;\
      }

#else // !defined(CPM_USE_MPI)

   #define CPM_MPI_O\
   bool send(CpmRoot::Z dest, CpmRoot::Z tag, CpmMPI::Com comm=\
   CpmMPI::Com())const{ dest; tag; comm; return true;}

   #define CPM_MPI_I\
      bool rec(CpmRoot::Z source, CpmRoot::Z tag, CpmMPI::Com\
       comm=CpmMPI::Com()){ source; tag; comm; return true;}

#endif

// input and output functions; the function read is the common interface
// to both built-in types and Cpm classes to read from commented files
// ( which does not work for R and Z via >>)

// all IMPL-MACROS get completed by their corresponding
// MPI-functionality.
// Then it is guaranteed that all classes define proper inter-process
// communication.
// Notice that for non-class types like Z and R, there is no re-definition
// of operators >> an <<.

#define CPM_IO_IMPL\
   friend std::ostream& operator<<(std::ostream& out, Type const& x)\
      { CpmRoot::IO<Type>().o(x,out); return out;}\
   friend std::istream& operator>>(std::istream& in, Type& x)\
      { CpmRoot::IO<Type>().i(x,in); return in;}\
      CPM_MPI_O\
      CPM_MPI_I

// declaration macros without public qualifications
// with and without virtual qualification.

#define CPM_IO_V\
```

```
   virtual bool prnOn(std::ostream&)const;\
   virtual bool scanFrom(std::istream&);\
   CPM_IO_IMPL

#define CPM_IO\
   bool prnOn(std::ostream&)const;\
   bool scanFrom(std::istream&);\
   CPM_IO_IMPL

// sometimes one needs the O-part (= output part) in isolation:

#define CPM_O_IMPL\
   friend std::ostream& operator<<(std::ostream& out, Type const& x)\
      { CpmRoot::IO<Type>().o(x,out); return out;}\
   CPM_MPI_O

#define CPM_O_V\
   virtual bool prnOn(std::ostream&)const;\
   CPM_O_IMPL

#define CPM_O\
   bool prnOn(std::ostream&)const;\
   CPM_O_IMPL

// help to unify defining member functions
//    bool prnOn(ostream& str)const;
//    bool scanFrom(istream& str);
// in classes with many data members that define prnOn and scanFrom.
// Notice that the ; is missing in the definition of cpmp and cpms,
// so that the call of those macros has to provide it.
// These lests these macros behave just as if they were functions.
// E.g.
/*
   bool prnOn(ostream& str)const
   {
      cpmp(x1);
      cpmp(x2);
      cpmp(x3);
      return true;
   }

   bool scanFrom(istream& str)
   {
      cpms(x1);
      cpms(x2);
      cpms(x3);
      return true;
   }
*/
// Defining the nameOf function conveniently, no ';' at the end.
```

```
#define CPM_NAM(X) Word nameOf()const{ return Word(#X);}
#define CPM_NAM_V(X) virtual Word nameOf()const{ return Word(#X);}
// Declarations that need a ';' behind. The rule is that any CPM_...
// needs no ; behind it, but cpm... needs it.
#define cpmwt(X) if (!CpmRoot::writeTitle((X),str)) return false
   // wt for 'write title'
#define cpmwat if (!CpmRoot::writeTitle(nameOf().str(),str)) return false
   // wt for 'write automatic title', needs no argument.
#define cpmwbt if (!CpmRoot::writeTitle((nameOf()&" begin").str(),str))\
 return false
   // wbt for 'write begin title'
#define cpmwet if (!CpmRoot::writeTitle((nameOf()&" end").str(),str))\
 return false
   // wet for 'write end title', needs no argument.
#define cpmp(X) if (!CpmRoot::prnT((X),str)) return false
   // p for print
#define cpms(X) if (!CpmRoot::scanT((X),str)) return false
   // s for scan
#define cpmrh(X) rch.read(sec,#X,X)
   // Regularly occurring idiom for reading from a RecordHandler
   // read handler. Stops the program if the quantity cannot be
   // read.
#define cpmrhf(X) rch.read(sec,#X,X,1)
   // Regularly occurring idiom for reading from a RecordHandler
   // read handler, option pedantic = 1, so that only a warning
   // gets issued if the quantity cannot be read.
   // 'f' for 'floppy'
#define cpmr(X) rc.get(sec,#X,X,1)
   // Regularly occurring idiom for reading from a read handler
   // of type Record. Never stops the program if the quantity
   // cannot be read. Issues a warning however.
   // Introduced 2010-08-30.
#define cpmrf(X) rc.get(sec,#X,X,0)
   // Regularly occurring idiom for reading from a read handler
   // of type Record. Never stops the program if the quantity cannot be
   // read. Does not even warn. Introduced 2010-08-30.
#define cpmwh(X) rch.write(sec,#X,X)
   // Regularly occurring idiom for writing to a RecordHandler
   // write handler.
#define cpmrc(X) rc_.get(sec,#X,X,1)
   // regularly occurring idiom for reading from the attribute
   // Recordable::rc_

#define cpmc(X) if (X < obj.X) return 1;\
 if (X > obj.X) return -1
   // c for compare, helps to implement member function com
   // declared in CPM_ORDER (2009-10-02). As in the macros cpmp and
   // cpms the name 'str' for the stream argument was implied, we
   // require here the argument to be named obj'. Works fine and
   // simplifies the implementation of CPM_ORDER considerably.
```

```
#define CPM_WORD\
   CpmRoot::Word toWord()const\
   { std::ostringstream ost; prnOn(ost);\
   return CpmRoot::Word(ost.str());}

// order-related stuff now in basicinterfaces

// multiplicative structure with respect to scalars of type ScalarType.
// version where mutating member functions (actually operator *=)  are
// primary
#define CPM_PRODUCT_SCALAR_M\
   Type& operator *=(ScalarType const& s);\
   Type& operator /=(ScalarType const& s)\
   {return operator *=(CpmRoot::invT<ScalarType>(s));}\
   Type operator *(ScalarType const& s)const\
   { Type res=*this; return res*=s;}\
   friend Type operator *(ScalarType const& s, Type const& x)\
   { Type res=x; return res*=s;}\
   Type operator /(ScalarType const& s)const\
   { Type res=*this; return res/=s;}

// version where constant member functions (actually operators *) are
// primary
#define CPM_PRODUCT_SCALAR_C\
   Type operator *(ScalarType const& s)const;\
   Type operator /(ScalarType const& s)const\
   { return *this*CpmRoot::invT<ScalarType>(s);}\
   friend Type operator *(ScalarType const& s, Type const& x)\
    { return x*s;}\
   Type& operator *=(ScalarType const& s){ return *this=*this*s;}\
   Type& operator /=(ScalarType const& s){ return *this=*this/s;}

// linear structure with respect to scalars of type ScalarType.
// version where mutating member functions (actually operator *=, +=) are
// primary
#define CPM_SCALAR_M\
   Type& operator *=(ScalarType const& s);\
   Type& operator +=(ScalarType const& s);\
   Type& operator -=(ScalarType const& s){return operator +=(-s);}\
   Type& operator /=(ScalarType const& s)\
   {return operator *=(CpmRoot::invT<ScalarType>(s));}\
   Type operator *(ScalarType const& s)const\
   { Type res=*this; return res*=s;}\
   friend Type operator *(ScalarType const& s, Type const& x)\
   { Type res=x; return res*=s;}\
   Type operator +(ScalarType const& s)const\
   { Type res=*this; return res+=s;}\
   Type operator -(ScalarType const& s)const\
   { Type res=*this; return res-=s;}\
```

```
   Type operator /(ScalarType const& s)const\
   { Type res=*this; return res/=s;}

// version where constant member functions (actually operators *,+) are
// primary
#define CPM_SCALAR_C\
   Type operator *(ScalarType const& s)const;\
   Type operator +(ScalarType const& s)const;\
   Type operator -(ScalarType const& s)const{ return *this+(-s);}\
   Type operator /(ScalarType const& s)const\
   { return *this*CpmRoot::invT<ScalarType>(s);}\
   Type& operator *=(ScalarType const& s){ return *this=*this*s;}\
   Type& operator +=(ScalarType const& s){ return *this=*this+s;}\
   Type& operator -=(ScalarType const& s){ return *this=*this-s;}\
   Type& operator /=(ScalarType const& s){ return *this=*this/s;}

// product, where the mutating member function *= is primary
#define CPM_PRODUCT_M\
   Type& operator *=(Type const& x);\
   Type operator * (Type const& x)const\
   { Type res=*this; return res*=x;}

// product, where the constant member function operator * is primary
#define CPM_PRODUCT_C\
   Type operator * (Type const& x)const;\
   Type& operator *=(Type const& x){ return *this=(*this)*x;}

// inversion
// lean inversion
#define CPM_INVERSION\
   Type operator !(void)const;\
   Type& operator /=(Type const& x){ return operator*=(!x);}\
   Type operator /(Type const& x)const\
   { return *this*!x;}

// more complex version of inversion which also brings the zero into the
// game (which in a group needs not to be a defined concept)
#define CPM_DIVISION\
   Type net(CpmRoot::Z i=0)const;\
   Type inv(void)const;\
   Type operator !(void)const{ return inv();}\
   Type& operator /=(Type const& x){ return operator*=(x.inv());}\
   Type operator /(Type const& x)const\
   { return *this*x.inv();}\
   friend Type net(Type const& x, CpmRoot::Z i=0)\
   { return x.net(i);}\
   friend Type inv(Type const& x){ return x.inv();}

// inversion and neutral elements; no multiplication involved.
#define CPM_NET\
```

```
   Type net(CpmRoot::Z i=0)const;\
   friend Type net(Type const& x, CpmRoot::Z i=0)\
   { return x.net(i);}

#define CPM_INV\
   CPM_NET\
   Type inv(void)const;\
   Type operator !(void)const{ return inv();}\
   friend Type inv(Type const& x){ return x.inv();}

// minimum infra structure for programming unbiased test for the validity
// of expected mathematical identities

#define CPM_TEST\
   Type ran(CpmRoot::Z j=0)const;\
   Type test(CpmRoot::Z)const;\
   CpmRoot::Z hash(void)const;

#define CPM_TEST_X\
   CPM_TEST\
   CpmRoot::R dis(Type const&)const;\
   CpmRoot::R abs(void)const;\
   CpmRoot::R absSqr(void)const;\
   CpmRoot::R abs2(void)const{ return absSqr();}

// CPM_TEST_ALL, comprises all infra-structure functions
// which are defined in cpmnumbers.h for the bb-types.
#define CPM_TEST_ALL\
   CPM_TEST_X\
   CPM_INV\
   Type con()const;

// CPM_TEST_MOD is a modification of CPM_TEST which turned out
// to be needed in a proper implementation of class Q where it would
// be unnatural to have abs R-valued instead of Q-valued. Also
// implementing abs in terms of absSqure would not be suitable in
// this case.
#define CPM_TEST_MOD\
   CPM_TEST\
   CpmRoot::R dis(Type const&)const;\
   Type abs(void)const;\
   Type absSqr(void)const;

// descriptors
#define CPM_DESCRIPTORS\
   virtual Word nameOf(void) const;\
   virtual Word toWord(void)const;

// arithmetic
#define CPM_DIFFERENCE\
```

```
   Type neg(void)const;\
   Type& operator -=(Type const& x){ return operator+=(x.neg());}\
   Type operator -(void)const{ return neg();}\
   Type operator - (Type const& x)const\
   { return *this+x.neg();}

#define CPM_SUM_PLAIN\
   Type& operator +=(Type const& x);\
   Type operator + (Type const& x)const;

#define CPM_DIFFERENCE_PLAIN\
   Type& operator -=(Type const& x);\
   Type operator -(void)const;\
   Type operator - (Type const& x)const;

// sum, where the mutating member function += is primary
#define CPM_SUM_M\
   Type& operator +=(Type const& x);\
   Type operator + (Type const& x)const\
   { Type res=*this; return res+=x;}

// sum, where the constant member function + is primary
#define CPM_SUM_C\
   Type operator + (Type const& x2)const;\
   Type& operator +=(Type const& x){ return *this=(*this)+x;}

// sum, where the constant member function + is primary
#define CPM_DIFF_C\
   Type operator - (Type const& x2)const;\
   Type& operator -=(Type const& x){ return *this=(*this)-x;}

// complex conjugation
// for matrix types it is hermitian conjugation
// in star-algegras it is the adjoint (i.e.the star)
#define CPM_CONJUGATION\
   Type con(void)const;\
   friend Type con(Type const& x){ return x.con();}\
   Type operator~(void)const{ return con();}

// lean complex conjugation without friend function
// (friend functions are better avoided since they cause
// problems with todays MS compilers when templates are
// heavily involved)
#define CPM_CONJ\
   Type con(void)const;\
   Type operator~(void)const{ return con();}

// Scalar Product and real valued dependents

#define CPM_DOT_PRODUCT_LEAN\
```

```
   ScalarType operator |(Type const&)const;\
   CpmRoot::R absSqr()const\
   { return CpmRoot::absT<ScalarType>(*this|*this);}\
   CpmRoot::R abs()const\
   { return cpmsqrt(absSqr());}\
   CpmRoot::R abs2()const\
   { return absSqr();}

#define CPM_DOT_PRODUCT\
   CPM_DOT_PRODUCT_LEAN\
   CpmRoot::R dis(Type const& x)const\
   {\
      CpmRoot::R a=abs();\
      CpmRoot::R b=x.abs();\
      CpmRoot::R d=(*this - x).abs();\
      return CpmRoot::disDefFun(a,b,d);\
   }

#define CPM_NORMALIZE \
CpmRoot::R nor_()\
{\
   CpmRoot::R nOrig=abs();\
   if (nOrig>0){\
      CpmRoot::R nInv=R(1.)/nOrig;\
      operator*=(ScalarType(nInv));\
   }\
   return nOrig;\
}

#define CPM_RFD \
 Type (void)=default;\
 Type(const Type&)=default;\
 Type(Type&&)=default;\
 Type& operator=(const Type&)=default;\
 Type& operator=(Type&&)=default;

// linear structure with respect to scalars of type ScalarType.
// Inversion in ScalarType is not considered; has to be added in a class
// where this is needed (e.g in C)

#define CPM_LINEAR\
   CpmRoot::Z dim()const;\
   Type& operator +=(Type const& a);\
   Type& operator -=(Type const& a);\
   Type& operator *=(ScalarType const& r);\
   Type& neg(void);\
   friend CpmRoot::Z dim(Type const& x){ return x.dim();}\
   friend Type operator +(Type const& x1, Type const& x2)\
      { Type res=x1; return res+=x2;}\
   friend Type operator -(Type const& x1, Type const& x2)\
```

```
      { Type res=x1; return res-=x2;}\
   friend Type operator -(Type const& x)\
      { Type res=x; return res.neg();}\
   friend Type operator *(Type const& x, ScalarType const& s)\
      { Type res=x; return res*=s;}\
   friend Type operator *(ScalarType const& s, Type const& x)\
      { Type res=x; return res*=s;}

// a lean and collection of arithmetic operations for linear spaces with
// scalar product. This is now the basic interface of
// VVa<>, VVVa<>, and VVVVa<>.
#define CPM_LIN\
   CPM_SUM_C\
   CPM_DIFF_C\
   Type operator-(void)const;\
   Type operator*(ScalarType const& r)const;\
   Type& operator *=(ScalarType const& r)\
      { return *this=*this*r;}\
   CPM_DOT_PRODUCT\
   CPM_CONJ
#endif
```

22 cpmm.h

```
//? cpmm.h
//? Status of work 2023-10-20.
//?
//?  ...


#ifndef CPM_M_H_
#define CPM_M_H_
/*

   Description: Defines a template class M<X,Y> of mappings
      X-->Y comparable to std::map<X,Y>
*/

#include <map>
#include <cpmv.h>
#include <cpmsr.h>

////////////////////// class M<X,Y> ////////////////////////////////////

namespace CpmArrays{

   using namespace CpmStd;
   using namespace CpmRoot;
   using CpmFunctions::F;
   using CpmArrays::S;

template <class X, class Y>
   class M{ // map, associative array, dictionary, hash
   // it is assumed that X<X and X>X is defined

   Y y0_{};
   typedef M<X,Y> Type;

public:
   std::map<X,Y> p_;
 //  Z locate(X const& x)const{ return rep_.locate(aux::Pair01<X,Y>(x));}

   auto begin(){return p_.begin();}
   auto end(){return p_.end();}
   auto cbegin(){return p_.cbegin();}
   auto cend(){return p_.cend();}
   CPM_IO
   CPM_ORDER
   M():p_{}{}
      // constructor for the void set of X,Y pairs.
      // Non-trivial instances can be obtained by applying
```

```
      // mutating methods, such as function set(...) on the
      // default instance
   M<X,Y> test(Z i)const;
   M<X,Y> ran(Z i=0)const;
   R dis(M<X,Y> const& m)const;
   R absSqr()const;
   R abs()const;

    X const& inv(Y const& y, bool& found)const;
      //:inverse
      // Returns a value x in X such that (*this)[x]==y if such an x
      // exists. Otherwise we return the dfault element from X
      // an set 'found' equal to false.


   M(S<X> const& sx, F<X,Y> const& f):p_{},y0_{}
      // Setting the values by an algorithm.
   {
      S<X> s{sx};
      for (auto const& x:s){
         p_[x]=f(x);
      }
   }

   template <class T>
   M<X,T> operator&(M<Y,T> const& m)const;
   // concatenation corresponds to composition of functions
/*
   M(S<X> const& sx, Y const& y);
      // Setting a single value y for all elements of sx.

   void merge(V<Type> const& mapList, bool reverse=false);
      // combining *this and a list of maps into a single one.
      // This constructs a map that is formed out of all
      // (x,y)-pairs of *this and the components of mapList.
      // If for a x, there are more pairs (x,y), (x,y'), ...
      // The pair coming from mapList[i] with highest i
      // will actually be taken (if we would take more than one
      // we would not get a map). Thus we work through the
      // mapList in ascending order of their indexes and use it
      // in an overwriting mode. If the second argument differs
      // from the default value, we use reversed order so that
      // mapList[i] has highest priority for smallest i.
*/
   Y operator()(X const& x)const{ return (*this)[x];}
      // in analogy to F<X,Y>; returns the 'value of *this at x'

 //  S<Y> operator()(S<X> const& sx)const;
      // The common extension of functions from single argument values
      // to sets of argument values
```

```
// cardinality access: number of (x,y) pairs stored in *this
Z car()const{ return p_.size();}
   //: cardinality
Z dim()const{ return p_.size();}
   //: dimension
   // for uniformity with other array types
Z size()const{ return p_.size();}
   // for uniformity with STL code
bool isVoid(void)const{ return p_.size()==0;}
   //: is void

M<X,Y> select(F<X,bool> const& sel)const;
   // returns the M<X,Y> objects consisting of all those
   // (x,y)-pairs of *this for which sel(x)==true

Y& operator[](X const& x){ return p_[x];}
   // setting values in array-style (which also is the STL-style)
   // for M<X,Y> f, X x, Y y the statement
   // f[x]=y has the same effect as f.set(x,y)

Y const&  operator[](X const& x)const{
   auto q=p_.find(x);
   return q!=p_.end() ? q->second : y0_;}
   // getting to the values as references.
   // Let Z g(const& Y), M<X,Y> f, X x, Y y . Then
   // g(f[x]) does the same as
   // Y y1; f.get(x,y1); g(y1);
   // So, operator[] provides slightly more convenient grammar than
   // get(...) in this case.
   // This is the most efficient non-mutating access function

bool defined(X const& x)const{ return p_.find(x) != p_.end();}
   // f.defined(x) iff before a statement f[x]=... or f.set(x,...)
   // happened.

void set(X const& x, Y const& y){(*this)[x]=y;}
   //: set
   //  M<X,Y> f, X x, Y y ==>
   // ( f.set(x,y) ==> (y==f(x) evaluates to true)). Interesting:
   // it is possible to express this mixture of logic notation
   // and C++ much clearer in pure C++:
   // template <class X, class Y>
   // bool setTest(X x, Y y, M<X,Y> f){ f.set(x,y); return y==f(x);}
   // returns true for all arguments.

bool get(X const& x, Y& y)const;
   //: get
   // template <class X, class Y>
   // bool getTest(X x, M<X,Y> f){ Y y; f.get(x,y); return y==f(x);}
```

```
      // returns true for all arguments.

   V<Y> const& toV()const;

   S<X> dom()const{
      S<X> res;
      for (auto const& [x,y]:p_){
         res.add_(x);
      }
      return res;
   }

   S<Y> rng()const{
      S<Y> res;
      for (auto const& [x,y]:p_){
         res.add_(y);
      }
      return res;
   }


   Word toWord()const;
      //: to word

   Word nameOf()const;
      //: name of
};


////////////////////////// Implementation ////////////////////////////////

//{
//   Vr<T> res(Base::dom());
//   T tempX;
//   if (j==0){
//      for (Z i=Base::b(); i<=Base::e(); i++){
//         tempX=(*this)[i];
//         res[i]=Root<T>(tempX).ran(j);
//      }
//   }
//   else{
//      Z j0=1000; // this is done to avoid an obvious correlation
//         // among the coordinates of x.ran(i)
//         // and x.ran(i+1)
//      Z jIncr=Root<Z>(j0).ran(j);
//      Z jc=j;
//      for (Z i=Base::b(); i<=Base::e(); i++){
//         tempX=(*this)[i];
//         jc+=jIncr;
//         res[i]=Root<T>(tempX).ran(jc);
```

```
//      }
//   }
//   return res;
//}

template <class X, class Y>
M<X,Y> M<X,Y>::test(Z i)const
{  Sr<X> s;
   s=s.test(i);
   Y y0;
   Y yi=Root<Y>(y0).test(i);
   M<X,Y> res;
   for (auto const& x : s){
     res[x]=Root<Y>(yi).ran(++i);
   }
   return res;
}

template <class X, class Y>
M<X,Y> M<X,Y>::ran(Z i)const
{
   M<X,Y> res;
   if (i==0){
      for (auto const& [x,y] : p_ ){
         res[x]=Root<Y>(y).ran(i);
      }
   }
   else{
      Z i0{1000};
      Z ic=i;
      Z iIncr=Root<Z>(i0).ran(i);
      for (auto const& [x,y] : p_ ){
         ic+=iIncr;
         res[x]=Root<Y>(y).ran(ic);
      }
   }
   return res;
}

template <class X, class Y>
R M<X,Y>::dis(M<X,Y> const& m)const
{
   R res{0.};
   for (auto const& [x,y] : p_){
      Y y2=m[x];
      R d=CpmRoot::Root<Y>(y).dis(y2);
      if (d>res) res=d;
   }
   return res;
}
```

```
template <class X, class Y>
R M<X,Y>::absSqr()const
{
   R res{0.};
   for (auto const& [x,y] : p_){
      res+=CpmRoot::Root<Y>(y).absSqr();
   }
   return res;
}

template <class X, class Y>
R M<X,Y>::abs()const
{
   return cpmsqrt(absSqr());
}

template <class X, class Y>
V<Y> const& M<X,Y>::toV()const
{
   Z n=size(),i=0;
   V<Y> res(n);
   for (auto const& [key, value] : p_){
      i++;
      res[i]=value;
   }
   res.sort_();
   return res;
}

template <class X, class Y>
template <class T>
M<X,T> M<X,Y>::operator&(M<Y,T> const& m)const
{
   M<X,T> res;
      for (auto const& [x,y] : p_){
      bool bi=m.defined(y);
      if (bi) res.set(x,m[y]);
   }
   return res;
}

template <class X, class Y>
bool M<X,Y>::get(X const& x, Y& y)const
{
   auto q=p_.find(x);
   if (q==p_.end()){
      return false;
   }
   y=q->second;
```

```
   return true;
}

template <class X, class Y>
X const& M<X,Y>::inv(Y const& y, bool& found)const
{
  X res{};
  found=false;
  for (auto const& [key, value] : p_){
      if (value==y){ res=key; found=true; break;}
  }
  return res;
}

template <class X, class Y>
M<X,Y>  M<X,Y>::select(F<X,bool> const& sel)const
{
   M<X,Y> res;
   for (auto const& [key,value] : p_){
      X2<X,Y> xy(key,value);
      X x=xy.c1();
      if (sel(x)) res[x]=xy.c2();
   }
   return res;
}

template <class X, class Y>
Word M<X,Y>::nameOf()const
{
   Word wi="M<";
   Word wx=CpmRoot::Name<X>()(X());
   Word wy=CpmRoot::Name<Y>()(Y());
   return wi&wx&","&wy&">";
}

template <class X, class Y>
Word M<X,Y>::toWord()const
{
   Word res="{ ";
   bool first=true;
   for (const auto& [key, value] : p_){
      Word wx=CpmRoot::toWord<X>(key);
      Word wy=CpmRoot::toWord<Y>(value);
      Word wb="("&wx&", "&wy&")"; // b for begining
      Word wf=", "&wb; // f for following
      first ? res&=wb : res&=wf;
      first=false;
   }
   res&=" }";
   return res;
```

```
}

template <class X, class Y>
bool M<X,Y>::prnOn(ostream& str)const
{
   Z mL=3;
   static Word loc("M<>::prnOn(ostream&)");
   CPM_MA
   Z n=size();
   cpmwat;
   cpmp(n);
   if (n==0){
      cpmmessage(mL, "no values there");
      goto label;
   }
   for (const auto& [key, value] : p_) {
      X2<X,Y> val(key,value);
      if (!val.prnOn(str)){
         CPM_MZ
         return false;
      }
    }
label:
   CPM_MZ
   return true;
}

template <class X, class Y>
bool M<X,Y>::scanFrom(istream& str)
{
   Z mL=3;
   Z mL2=5;
   static Word loc("M<>::scanFrom(istream&)");
   CPM_MA
   Z n;
   if (!CpmRoot::scanT<Z>(n,str)){
      cpmwarning(loc&": can't read dimension");
      CPM_MZ
      return false;
   }
   else if (n<0){
      cpmwarning(loc&": negative dimension: n="&cpm(n));
      CPM_MZ
      return false;
   }
   else if (n>dimMax)
      cpmwarning(loc&": dimension read as "&cpm(n)&
      ": probably too large");
   else if (n==0){
      cpmwarning(loc&" dimension read as 0");
```

```
      if (cpmverbose>mL2){ // copying the stream beyond the dubious point
         // can be useful
         char c;
         while (str.get(c)) cpmdata.put(c);
      }
   }
   else{
      cpmmessage(mL,loc&": dimension read as "&cpm(n));
   }
   M<X,Y> res;
   X2<X,Y> val;
   //for (const auto& [key, value] : p_) {
   for (Z i=1;i<=n;i++){
      if (!val.scanFrom(str)){
         cpmwarning(loc&": reading failed for i="&cpm(i)&" of "&cpm(n));
         CPM_MZ
         return false;
      }
      res[val.c1()]=val.c2();
   }
   *this=res;
   CPM_MZ
   return true;
}

template <class X, class Y >
Z M<X,Y>::com(M<X,Y> const& m)const
{
   Z d1=size(), d2=m.size();
   if (d1<d2) return 1;
   else if (d1>d2) return -1;
   else{ // then d1==d2
      for (auto const& [key, value] : p_){
         if (value < m[key])   return 1;
         if (value > m[key])   return -1;
      }
      return 0;
   }
}

/*****************************
template <class X, class Y>
template<class T>
M<X,T> M<X,Y>::operator()(F<Y,T> const& g)const
{
   M<X,T> res();
   for (Z i=b();i<=e();++i){
      X2<X,Y> xyi=xy(i);
      res.set(xyi.c1(),g(xyi.c2()));
   }
```

```
   return res;
}

template <class X, class Y>
template <class T>
M<X,T> M<X,Y>::operator&(M<Y,T> const& m)const
{
   M<X,T> res;
   for (Z i=b();i<=e();++i){
      X2<X,Y> xyi=xy(i);
      bool bi=m.defined(xyi.c2());
      if (bi) res.set(xyi.c1(),m[xyi.c2()]);
   }
   return res;
}

template <class X, class Y>
Word M<X,Y>::nameOf()const
{
   Word wi="M<";
   Word wx=CpmRoot::Name<X>()(X());
   Word wy=CpmRoot::Name<Y>()(Y());
   return wi&wx&","&wy&">";
}

template <class X, class Y>
Word M<X,Y>::toWord()const
{
   Word res="{ ";
   for (Z i=b();i<=e();++i){
      X2<X,Y> xyi=xy(i);
      Word wxi=CpmRoot::toWord<X>(xyi.c1());
      Word wyi=CpmRoot::toWord<Y>(xyi.c2());
      res&=" ( "&wxi&" , "&wyi&" ) ";
      if (i!=e()) res&=" , ";
   }
   res&=" }";
   return res;
}

template <class X, class Y>
M<X,Y>::M(S<X> const& sx, F<X,Y> const& f):y0_(),rep_()
{
   for (Z i=sx.b();i<=sx.e();++i){
      X xi=sx[i];
      set(xi,f(xi));
   }
}

template <class X, class Y>
```

```
M<X,Y>::M(S<X> const& sx, Y const& y):y0_(),rep_()
{
   for (Z i=sx.b();i<=sx.e();++i){
      X xi=sx[i];
      set(xi,y);
   }
}

template <class X, class Y>
Y M<X,Y>::operator()(X const& x)const
{
   Z i=locate(x);
   if (i==0) return Y();
   else return rep_[i].c2();
}

template <class X, class Y>
Y& M<X,Y>::operator[](X const& x)
{
   aux::PairO1<X,Y> fx(x);
   rep_<<fx;
   Z i=rep_.locate(fx);
   return rep_.ref(i).c2();
}

template <class X, class Y>
Y const& M<X,Y>::operator[](X const& x)const
{
   aux::PairO1<X,Y> fx(x);
   Z i=rep_.locate(fx);
   if (i<=0){
      return y0_;
   }
   else{
      return rep_[i].c2();
   }
}

template <class X, class Y>
bool M<X,Y>::get(X const& x, Y& y)const
{
   Z i=locate(x);
   if (i==0){
      return false;
   }
   else{
      y=rep_[i].c2();
      return true;
   }
}
```

```
template <class X, class Y>
S<Y> M<X,Y>::operator()(S<X> const& sx)const
{
   Z n=sx.car();
   V<Y> val(n);
   for (Z i=1;i<=n;i++){
      val[i]=y((sx[i]));
   }
   return S<Y>(val);
}

template <class X, class Y>
void M<X,Y>::merge(V<Type> const& mapList, bool reversed)
{
   Z ni,i,j,d=mapList.dim();
   if (!reversed){
      for (i=1;i<=d;i++){
         ni=mapList[i].car();
         for (j=1;j<=ni;j++){
            X x=mapList[i].x(j);
            Y y=mapList[i].y(j);
            set(x,y);
         }
      }
   }
   else{
      for (i=d;i>=1;i--){
         ni=mapList[i].car();
         for (j=1;j<=ni;j++){
            X x=mapList[i].x(j);
            Y y=mapList[i].y(j);
            set(x,y);
         }
      }
   }
}

template <class X, class Y>
S<X> M<X,Y>::dom()const
{
   S<X> res;
   for (Z i=1;i<=dim();++i) res.add(x(i));
   return res;
}

template <class X, class Y>
S<Y> M<X,Y>::ran()const
{
   S<Y> res;
```

```
   for (Z i=1;i<=dim();++i) res.add(y(i));
   return res;
}

template <class X, class Y>
M<X,Y>  M<X,Y>::select(F<X,bool> const& sel)const
{
   M<X,Y> res;
   for (Z i=1;i<=dim();++i){
      X2<X,Y> xyi=xy(i);
      X xi=xyi.c1();
      if (sel(xi)) res[xi]=xyi.c2();
   }
   return res;
}

template <class X, class Y>
bool M<X,Y>::prnOn(ostream& str)const
{
   Z mL=3;
   static Word loc("M<>::prnOn(ostream&)");
   CPM_MA
   Z n=car();
   cpmwat;
   cpmp(n);
   if (n==0){
      cpmmessage(mL, "no values there");
      goto label;
   }
   for (Z i=1;i<=n;i++){
      if (CpmRoot::wrtTit){
         Word wi("// i=");
         wi&=cpm(i);
         bool bi=wi.prnOn(str);
         cpmassert(bi==true,loc);
      }
      X2<X,Y> val(x(i),y(i));
      if (!val.prnOn(str)){
         CPM_MZ
         return false;
      }
   }
label:
   cpmwet;
   CPM_MZ
   return true;
}

template <class X, class Y>
bool M<X,Y>::scanFrom(istream& str)
```

```
{
   Z mL=3;
   Z mL2=5;
   static Word loc("M<>::scanFrom(istream&)");
   CPM_MA
   Z n;
   if (!CpmRoot::scanT<Z>(n,str)){
      cpmwarning(loc&": can't read dimension");
      CPM_MZ
      return false;
   }
   else if (n<0){
      cpmwarning(loc&": negative dimension: n="&cpm(n));
      CPM_MZ
      return false;
   }
   else if (n>dimMax)
      cpmwarning(loc&": dimension read as "&cpm(n)&
      ": probably too large");
   else if (n==0){
      cpmwarning(loc&" dimension read as 0");
      if (cpmverbose>mL2){ // copying the stream beyond the dubious point
         // can be useful
         char c;
         while (str.get(c)) cpmdata.put(c);
      }
   }
   else{
      cpmmessage(mL,loc&": dimension read as "&cpm(n));
   }
   M<X,Y> res;
   X2<X,Y> val;
   for (Z i=1;i<=n;i++){
      if (!val.scanFrom(str)){
         cpmwarning(loc&": reading failed for i="&cpm(i)&" of "&cpm(n));
         CPM_MZ
         return false;
      }
      res.set(val.c1(),val.c2());
   }
   *this=res;
   CPM_MZ
   return true;
}

template <class X, class Y >
Z M<X,Y>::com(M<X,Y> const& s)const
{
   Z d1=dim(), d2=s.dim();
   if (d1<d2) return 1;
```

```
   else if (d1>d2) return -1;
   else{ // then d1==d2
      for (Z i=1;i<=d1;i++){
         if (xy(i)<s.xy(i))   return 1;
         if (xy(i)>s.xy(i))   return -1;
      }
      return 0;
   }
}
*****************************************/
} // CpmArrays

#endif
```

23 cpmm2.h

```
//? cpmm2.h
//? Status of work 2023-10-20.
//?
//?  ...


#ifndef CPM_M_H_
#define CPM_M_H_
/*

   Description: Defines a template class M<X,Y> of mappings
      X-->Y comparable to std::map<X,Y>
*/

#include <cpms.h>

////////////////////// class M<X,Y> ////////////////////////////////////

namespace CpmArrays{

   using namespace CpmStd;
   using CpmFunctions::F;
   using CpmArrays::S;

namespace aux{ // a technical construct

template <class X, class Y>
class  Pair01{  // Pair with order according to first element only.
   // Especially all pairs differing in only the second component
   // are equal by ==. This allows an elegant access to the x-values
   // of (x,y) pairs held in M<X,Y>::rep_. Since this is a 'heavy-duty
   // technical class' it is no longer implemented as derived from
   // X2<X,Y>.
   X x_;
   Y y_;
   typedef Pair01<X,Y> Type;
public:
   Word nameOf()const
   {
      return Word("Pair01< "&
          CpmRoot::Name<X>()(X())&
          ","&
          CpmRoot::Name<Y>()(Y())&
          " >");
   }
   CPM_ORDER
   CPM_IO
```

```
   Pair01(X const& x, Y const& y):x_(x),y_(y){}
   explicit Pair01(X const& x):x_(x),y_(){}
   Pair01():x_(),y_(){}
   X& c1(void){ return x_;}
   Y& c2(void){ return y_;}
   X const& c1(void)const{ return x_;}
   Y const& c2(void)const{ return y_;}
};

template <class X, class Y>
Z Pair01<X,Y>::com(Pair01<X,Y> const& s)const
{
   if (x_<s.x_) return 1;
   else if (x_>s.x_) return -1;
   else return 0;
}

template <class X, class Y>
bool Pair01<X,Y>::prnOn(ostream& str)const
{
   cpmp(x_);
   cpmp(y_);
   return true;
}

template <class X, class Y>
bool Pair01<X,Y>::scanFrom(istream& str)
{
   cpms(x_);
   cpms(y_);
   return true;
}

} // aux

template <class X, class Y>
   class M{ // map, associative array, dictionary, hash
   // it is assumed that X<X and X>X is defined
   Y y0_;
      // default Y, allows to define
      // Y const& operator[](const X const& x)const;
      // which STL is omitting
   S< aux::Pair01<X,Y> > rep_;
   Z locate(X const& x)const{ return rep_.locate(aux::Pair01<X,Y>(x));}

public:
   typedef M<X,Y> Type;
   typedef aux::Pair01<X,Y> RepType;
   CPM_IO
   CPM_ORDER
```

```
M(void):y0_(),rep_(){}
   // constructor for the void set of X,Y pairs.
   // Non-trivial instances can be obtained by applying
   // mutating methods, such as function set(...) on the
   // default intance

RepType const& operator[](Z i)const{ return rep_[i];}

M(S<X> const& sx, F<X,Y> const& f);
   // Setting the values by an algorithm.

M(S<X> const& sx, Y const& y);
   // Setting a single value y for all elements of sx.

void merge(V<Type> const& mapList, bool reverse=false);
   // combining *this and a list of maps into a single one.
   // This constructs a map that is formed out of all
   // (x,y)-pairs of *this and the components of mapList.
   // If for a x, there are more pairs (x,y), (x,y'), ...
   // The pair coming from mapList[i] with highest i
   // will actually be taken (if we would take more than one
   // we would not get a map). Thus we work through the
   // mapList in ascending order of their indexes and use it
   // in an overwriting mode. If the second argument differs
   // from the default value, we use reversed order so that
   // mapList[i] has highest priority for smallest i.

Y operator()(X const& x)const;
   // in analogy to F<X,Y>; returns the 'value of *this at x'

S<Y> operator()(S<X> const& sx)const;
   // The common extension of functions from single argument values
   // to sets of argument values

Y& operator[](X const& x);
   // setting values in array-style (which also is the STL-style)
   // for M<X,Y> f, X x, Y y the statement
   // f[x]=y has the same effect as f.set(x,y)

Y const& operator[](X const& x)const;
   // getting to the values as references.
   // Let Z g(const& Y), M<X,Y> f, X x, Y y . Then
   // g(f[x]) does the same as
   // Y y1; f.get(x,y1); g(y1);
   // So, operator[] provides slightly more convenient grammar than
   // get(...) in this case.
   // This is the most efficient non-mutating access function

void set(X const& x, Y const& y){(*this)[x]=y;}
   //: set
```

```
      //  M<X,Y> f, X x, Y y ==>
      // ( f.set(x,y) ==> (y==f(x) evaluates to true)). Interesting:
      // it is possible to express this mixture of logic notation
      // and C++ much clearer in pure C++:
      // template <class X, class Y>
      // bool setTest(X x, Y y, M<X,Y> f){ f.set(x,y); return y==f(x);}
      // returns true for all arguments.

   bool get(X const& x, Y& y)const;
      //: get
      // template <class X, class Y>
      // bool getTest(X x, M<X,Y> f){ Y y; f.get(x,y); return y==f(x);}
      // returns true for all arguments.

// cardinality access: number of (x,y) pairs stored in *this
   Z car()const{ return rep_.dim();}
      //: cardinality
   Z dim()const{ return rep_.dim();} // for uniformity with other array
      //: dimension
   // types
   Z size()const{ return rep_.dim();} // for uniformity with STL code
   bool isVoid(void)const{ return rep_.isVoid();}
      //: is void
   Z b()const{ return rep_.b();}
      //. begin
   Z e()const{ return rep_.e();}
      //. end

// accessing arguments, values, and (x,y)-pairs by an index ranging from
// 1 to car()

   X x(Z i)const{ return rep_(i).c1();}
   Y y(Z i)const{ return rep_(i).c2();}
   X2<X,Y> xy(Z i)const{ return X2<X,Y>(x(i),y(i));}
      // template <class X, class Y>
      // bool xyTest(Z i, M<X,Y> f){ return y(i)==f(x(i));}
      // returns true for all arguments.
   bool defined(X const& x)const{ return locate(x)!=0;}
      // f.defined(x) iff before a statement f[x]=... or f.set(x,...)
      // happened.
   void clear_(X const& x){ Z i=locate(x); if (i>0) rep_.eliminate(i);}
      // after f.clear_(x), we have f.defined(x)==false

   S<X> dom()const;
      //: domain
      // domain of the function X-->Y associated with object *this
      // returns {x \in X | defined(x)==true}

   S<Y> ran()const;
      //: range
```

```
      // range of the function X-->Y associated with object *this
      // returns {y \in Y | y==operator[](x) for some x \in X}

   M<X,Y> select(F<X,bool> const& sel)const;
      // returns the M<X,Y> objects consisting of all those
      // (x,y)-pairs of *this for which sel(x)==true

   template <class T>
   M<X,T> operator()(F<Y,T> const& g)const;

   template <class T>
   M<X,T> operator&(M<Y,T> const& m)const;

   template< class T>
   M<T,Y> circ(M<T,X> const& m)const{ return m&(*this);}
      //: circ (LaTeX-command)
      // concatenation of mappings

   Word toWord()const;
      //: to word

   Word nameOf()const;
      //: name of
};

////////////////////////// Implementation ////////////////////////////////

template <class X, class Y>
template<class T>
M<X,T> M<X,Y>::operator()(F<Y,T> const& g)const
{
   M<X,T> res();
   for (Z i=b();i<=e();++i){
      X2<X,Y> xyi=xy(i);
      res.set(xyi.c1(),g(xyi.c2()));
   }
   return res;
}

template <class X, class Y>
template <class T>
M<X,T> M<X,Y>::operator&(M<Y,T> const& m)const
{
   M<X,T> res;
   for (Z i=b();i<=e();++i){
      X2<X,Y> xyi=xy(i);
      bool bi=m.defined(xyi.c2());
      if (bi) res.set(xyi.c1(),m[xyi.c2()]);
   }
   return res;
```

```
}

template <class X, class Y>
Word M<X,Y>::nameOf()const
{
   Word wi="M<";
   Word wx=CpmRoot::Name<X>()(X());
   Word wy=CpmRoot::Name<Y>()(Y());
   return wi&wx&","&wy&">";
}

template <class X, class Y>
Word M<X,Y>::toWord()const
{
   Word res="{ ";
   for (Z i=b();i<=e();++i){
      X2<X,Y> xyi=xy(i);
      Word wxi=CpmRoot::toWord<X>(xyi.c1());
      Word wyi=CpmRoot::toWord<Y>(xyi.c2());
      res&=" ( "&wxi&" , "&wyi&" ) ";
      if (i!=e()) res&=" , ";
   }
   res&=" }";
   return res;
}

template <class X, class Y>
M<X,Y>::M(S<X> const& sx, F<X,Y> const& f):y0_(),rep_()
{
   for (Z i=sx.b();i<=sx.e();++i){
      X xi=sx[i];
      set(xi,f(xi));
   }
}

template <class X, class Y>
M<X,Y>::M(S<X> const& sx, Y const& y):y0_(),rep_()
{
   for (Z i=sx.b();i<=sx.e();++i){
      X xi=sx[i];
      set(xi,y);
   }
}

template <class X, class Y>
Y M<X,Y>::operator()(X const& x)const
{
   Z i=locate(x);
   if (i==0) return Y();
   else return rep_[i].c2();
```

```
}

/*****************************
template <class X, class Y>
Y& M<X,Y>::operator[](X const& x)
{
   cpmcerr<<"Y& M<X,Y>::operator[](X const& x) calling"<<endl;
   aux::Pair01<X,Y> fx(x);
   cpmcerr<<" aux::Pair01<X,Y> fx(x); done"<<endl;
 //  Z i=rep_.loc3(fx);

   cpmcerr<<" first i = "<<i<<endl;
   if (i==0){
      cpmcerr<<"rep_.add(fx) calling"<<endl;
      rep_.add(fx);
   }
   i=rep_.locate(fx);
   cpmcerr<<" second i = "<<i<<endl;

   if (i<=0) cpmerror("Y& M<X,Y>::operator[](X const& x): i<=0");
   cpmcerr<<"Y& M<X,Y>::operator[](X const& x) nearly done"<<endl;

   rep_.enq_(fx);
   Z i=rep_.loc3(fx).c1();
   return rep_.ref(i).c2();
      // since the ordering of rep_ only relies on rep_.ref(i).c1()
      // there is no danger in exposing rep_.ref(i).c2() to
      // changes from outside
}
*********************************/

template <class X, class Y>
Y& M<X,Y>::operator[](X const& x)
{
   aux::Pair01<X,Y> fx(x);
   rep_<<fx;
   Z i=rep_.locate(fx);
   return rep_.ref(i).c2();
}

template <class X, class Y>
Y const& M<X,Y>::operator[](X const& x)const
{
   aux::Pair01<X,Y> fx(x);
   Z i=rep_.locate(fx);
   if (i<=0){
      return y0_;
   }
   else{
      return rep_[i].c2();
```

```
   }
}

template <class X, class Y>
bool M<X,Y>::get(X const& x, Y& y)const
{
   Z i=locate(x);
   if (i==0){
      return false;
   }
   else{
      y=rep_[i].c2();
      return true;
   }
}

template <class X, class Y>
S<Y> M<X,Y>::operator()(S<X> const& sx)const
{
   Z n=sx.car();
   V<Y> val(n);
   for (Z i=1;i<=n;i++){
      val[i]=y((sx[i]));
   }
   return S<Y>(val);
}

template <class X, class Y>
void M<X,Y>::merge(V<Type> const& mapList, bool reversed)
{
   Z ni,i,j,d=mapList.dim();
   if (!reversed){
      for (i=1;i<=d;i++){
         ni=mapList[i].car();
         for (j=1;j<=ni;j++){
            X x=mapList[i].x(j);
            Y y=mapList[i].y(j);
            set(x,y);
         }
      }
   }
   else{
      for (i=d;i>=1;i--){
         ni=mapList[i].car();
         for (j=1;j<=ni;j++){
            X x=mapList[i].x(j);
            Y y=mapList[i].y(j);
            set(x,y);
         }
      }
```

```
   }
}

template <class X, class Y>
S<X> M<X,Y>::dom()const
{
   S<X> res;
   for (Z i=1;i<=dim();++i) res.add(x(i));
   return res;
}

template <class X, class Y>
S<Y> M<X,Y>::ran()const
{
   S<Y> res;
   for (Z i=1;i<=dim();++i) res.add(y(i));
   return res;
}

template <class X, class Y>
M<X,Y>  M<X,Y>::select(F<X,bool> const& sel)const
{
   M<X,Y> res;
   for (Z i=1;i<=dim();++i){
      X2<X,Y> xyi=xy(i);
      X xi=xyi.c1();
      if (sel(xi)) res[xi]=xyi.c2();
   }
   return res;
}

template <class X, class Y>
bool M<X,Y>::prnOn(ostream& str)const
{
   Z mL=3;
   static Word loc("M<>::prnOn(ostream&)");
   CPM_MA
   Z n=car();
   cpmwat;
   cpmp(n);
   if (n==0){
      cpmmessage(mL, "no values there");
      goto label;
   }
   for (Z i=1;i<=n;i++){
      if (CpmRoot::wrtTit){
         Word wi("// i=");
         wi&=cpm(i);
         bool bi=wi.prnOn(str);
         cpmassert(bi==true,loc);
```

```
      }
      X2<X,Y> val(x(i),y(i));
      if (!val.prnOn(str)){
         CPM_MZ
         return false;
      }
   }
label:
   cpmwet;
   CPM_MZ
   return true;
}

template <class X, class Y>
bool M<X,Y>::scanFrom(istream& str)
{
   Z mL=3;
   Z mL2=5;
   static Word loc("M<>::scanFrom(istream&)");
   CPM_MA
   Z n;
   if (!CpmRoot::scanT<Z>(n,str)){
      cpmwarning(loc&": can't read dimension");
      CPM_MZ
      return false;
   }
   else if (n<0){
      cpmwarning(loc&": negative dimension: n="&cpm(n));
      CPM_MZ
      return false;
   }
   else if (n>dimMax)
      cpmwarning(loc&": dimension read as "&cpm(n)&
      ": probably too large");
   else if (n==0){
      cpmwarning(loc&" dimension read as 0");
      if (cpmverbose>mL2){ // copying the stream beyond the dubious point
         // can be useful
         char c;
         while (str.get(c)) cpmdata.put(c);
      }
   }
   else{
      cpmmessage(mL,loc&": dimension read as "&cpm(n));
   }
   M<X,Y> res;
   X2<X,Y> val;
   for (Z i=1;i<=n;i++){
      if (!val.scanFrom(str)){
         cpmwarning(loc&": reading failed for i="&cpm(i)&" of "&cpm(n));
```

```
         CPM_MZ
         return false;
      }
      res.set(val.c1(),val.c2());
   }
   *this=res;
   CPM_MZ
   return true;
}

template <class X, class Y >
Z M<X,Y>::com(M<X,Y> const& s)const
{
   Z d1=dim(), d2=s.dim();
   if (d1<d2) return 1;
   else if (d1>d2) return -1;
   else{ // then d1==d2
      for (Z i=1;i<=d1;i++){
         if (xy(i)<s.xy(i))   return 1;
         if (xy(i)>s.xy(i))   return -1;
      }
      return 0;
   }
}

} // CpmArrays

#endif
```

24 cpmmacros.h

```
//? cpmmacros.h
//? Status of work 2023-10-20.
//?
//?  ...


#ifndef CPM_MACROS_H_
#define CPM_MACROS_H_
/*

   Description: macros for uniform implementation of
      functions. Too often I wrote the starting cpmmessage
      and the closing one in function blocks. The abbreviation by macros
      is not impressive in this case. For the R-macro the savings are
      more convincing.

*/

////////////////////////// CPM_MAZ ///////////////////////////////////////

// M stands for messaging
//              -
// A for beginning, Z for end (from the
// role of A and Z in the alphabet)

// needs
// Z mL=...;
// Word loc(".....");
// see the example following CPM_RAZ

#define CPM_MA\
   cpmmessage(mL,loc&" started");

#define CPM_MZ\
   cpmmessage(mL,loc&" done");

////////////////////////// CPM_RAZ ///////////////////////////////////////

// R stands for report
//              -
// reports the computation time for the code between CPM_RA and
// CPM_RZ. If loadMes (load measure) is assigned a positive value
// also the computation time divided by that quantity is returned
// under the name of 'normalized computation time'

// needs
// Word loc(".....");
```

```
#define CPM_RA\
   R tcStart=cpmtime();\
   R loadMes=0;

#define CPM_RZ\
   R tcFinal=cpmtime();\
   R tcTotal=tcFinal-tcStart;\
   std::ostringstream ost;\
   ost<<std::endl<<"Performance data for "<<loc.toStr()<<std::endl;\
   ost<<"computing time ="<<tcTotal;\
   if (loadMes>0){\
      R tcNorm=tcTotal/loadMes;\
      ost<<std::endl<<"normalized computing time="<<tcNorm;\
   }\
   cpmmessage(1,Word(ost.str()),-1);
      // not writing to the status bar due to '-1'

//////////////////// usage of CPM_MAZ and CPM_RAZ //////////////////////
/*
   a typical usage is

  void aFunction()
  {
      Z mL=2;
      Word loc("aFunction()");
      CPM_MA
      CPM_RA
      ...
      loadMes=...;
      ...
      CPM_RZ
      CPM_MZ
   }

   for a function that notifies start, end, computation time,
   and an efficiency measure on cpmcerr.txt. Notice that A and Z
   resemble opening and closing brackets.
*/
#endif
```

25 cpmmpi.h

```
//? cpmmpi.h
//? Status of work 2023-10-20.
//?
//?  ...


#ifndef CPM_MPI_H_
#define CPM_MPI_H_

/*
   Description: Functions to send and receive objects of C++
   type string via MPI functions. The length of the strings
   is handled automatically. This header file will be included in
   cpminterfaces.h and, therefore, should not include files like
   cpmsystem.h, cpmtypes.h, cpmword.h
*/

#include <cpmdefinitions.h>
   // only to make known CPM_USE_MPI
   // no additional C+- stuff needed here

#include <string>

#if defined(CPM_USE_MPI)
   #include <mpi.h>
      // MPIPro
      // including within the namespace caused trouble!
#endif

namespace CpmMPI{

using std::string;

extern int mpiverbose;

#if defined(CPM_USE_MPI)

////////////////////////////// class Com ////////////////////////////////

class Com{
// class version of MPI's communicator concept for parallel computing
   // for which rank varies from 1 to size in order to cooperate with
   // CpmArrays. Note that, due to their template character, the
   // main communication functions can be used to send arround objects
   // of any type which implements the basic C+- marshalling scheme
   // provided by macro CPM_IO.
```

```
protected:
   MPI_Comm mc;
   int size;
   int rank; // starts with 1
public:
   static int tagFromToStatic(int f, int t,int size_)
   // returned simply f*t till 2004-07-07. The footnote to p. 254 of GLS
   // (Gropp,Lusk,Skjellum: using MPI) indicates that using the same tag
   // several times in succession might be not so desastrous as one could
   // expect it to be. Nevertheless, one should ensure that the function
   // is injective (appart from tagFromToStatic(f, t, size_)==
   // tagFromToStatic(t, f, size_) )
   {
      int i,j;
      if (f<=t){ i=f-1; j=t;}
      else{ i=t-1; j=f;}
      return size_*i+j;
   }
   int tagFromTo(int f, int t)const{ return tagFromToStatic(f, t, size);}
   Com(MPI_Comm comm=MPI_COMM_WORLD);
   int getSize()const{ return size;}
      // number of processes between which one can communicate via
      // sendStr and recStr
   int getRank()const{ return rank;}
   bool sendStr(const string& s, int dest, int tag)const;
      // sending C++-strings without having to input their length
      // return value true means success.
      // I use string as a universal data type singe the C++ class
      // stringstream offers efficient read and write.
      // 1<=dest<=size
   bool bcastStr(string& s)const;
      // broadcasts a string from the process with rank 1 to
      // all other processes
   bool recStr(string& s, int source, int tag)const;
      // receiving strings
      // 1<=source<=size
   int cyc(int i)const;
      // the return value k is equal i modulo size
      // also 1<=k<=size. Useful tool although one could argue that this
      // function belongs to the topic of 'communicator topologies'
      // and not to the communicator itself

// communication patterns:

// The aim of the following functions is to ensure a pairing
// of send and receive activities that avoids deadlock. Although
// it is not difficult to gain an intuitive insight saying that
// the combinations as implemented in these functions are safe against
// deadlock, I have no formal proof for this so far.
// Supporting points are
```

```
// 1. empirical: no exception found so far
// 2. philosophical: no simpler expression conceivable except
//    of obviously wrong ones.

// template member functions are very helpful here.
   template <class T, class V>
      // T has to define send() and rec(), V has to have
      // the basic properties of V<T> (see cpmv.h)
   void sendAndRec(const T& t, V& v)const
/*
   Typical application in a SPMD (single program, multiple data )
   situation: (using C+- classes)
   Com com;
   Z size=com.getSize();
   T t(...); // the contribution of com.getRank()
   V<T> v(size); // container for the contributions of others
   com.sendAndRec(t,v); // communication
   R sum=0; // starting to use the communicated stuff
   for (Z i=1;i<=size;i++) sum+=g(v[i]); // R g(const T&)
       // A cummulative effect of all processes is thus available for
       // each process.
*/
   {
      for (int i=1;i<=size;i++){
         if (i==rank){ // OK for size==1, rank==1
            v[i]=t;
         }
         else if (rank<i){
            t.send(i,tagFromTo(rank,i));
            v[i].rec(i,tagFromTo(i,rank));
         }
         else{
            v[i].rec(i,tagFromTo(i,rank));
            t.send(i,tagFromTo(rank,i));
         }
      }
   }

   template <class V>
      // V has to have the basic properties of V<T>, where
      // T defines send() and rec().
   void exchange(V const& s, V& v)const
/*
   Typical application in a SPMD (single program, multiple data )
   situation: (using C+- classes)
   Com com;
   int size=com.getSize();
   int rank=com.getRank();
// the contribution of com.getRank()
   V<T> s(size);
```

```
   for (int i=1;i<=size;i++){
      for (i==rank) continue;
         // there is nothing to exchange from rank to rank
      s[i]=....; // this is what rank has to say to i
   }
   V<T> v(size); // container for the messages from others
   com.exchange(s,v); // communication
   R sum=0; // starting to use the communicated stuff
   for (int i=1;i<=size;i++) sum+=g(v[i]); // R g(const T&)
       // Process rank has available the messages from its
       // partners all at once
*/
   {
      for (int i=1;i<=size;i++){
         if (i==rank){
            v[i]=s[i]; // was continue till 2003-06-10
         }
         else if (rank<i){
            s[i].send(i,tagFromTo(rank,i));
            v[i].rec(i,tagFromTo(i,rank));
         }
         else{
            v[i].rec(i,tagFromTo(i,rank));
            s[i].send(i,tagFromTo(rank,i));
         }
      }
   }

//   export // not working with MS cl
   template <class T>
      // see sendAndRec() for the assumed properties of T
   void bcast(T& t)const
      // 'broadcast'
      // Workhorse method for distributing data from the master to
      // the public. See CpmRootX::RecordHandler::ini_ for the typical
      // usage.
      // Call to bcast has to be preceeded by initialization of t in rank
      // 1. In tis respect rank 1 plays a special role; the call to
      // bcast shoult not be spoilt by conditions on rank!
      // After call for each rank t is assured to have the value which
      // rank 1 had provided. Execution of this function involves each
      // rank with at least one send or receive activity. Since rank 1
      // sends to all processes, and send is not considered done before
      // data are received, rank 1 can only continue after all receives
      // are done in all other processes. Any other process also can
      // only continue after it received. Who received first may hurry
      // away not waiting till others are also ready with receiving. So
      // this is not a strict synchronization mechanism.
      // However, each process that hurries away must have completed its
      // receive and this can only be achieved if all messages sent
```

```
   // earlier have been received to. So the function should force
   // completion of all communication tasks that  were pending at the
   // moment of calling bcast. Similar considerations apply to the
   // other collective communication functions.

   // A definition of this template providing diagnostics would be
   // desirable. In order to have cpmmessage available one could in
   // principle include  cpmsystems.h into the present file and
   // implement messages much like those of sendStr and recStr. In our
   // special situation this is not working since cpmmpi.h has to be
   // included in cpminterfaces.h where cpmsystem.h should not yet be
   // known. So one had to move the implementation to cpmmpi.cpp which,
   // however, asks for the additional qualification 'export'
   // (see Vandevoorde, Josuttis: C++ Templates, p. 68) which MS cl
   // does not understand.
{
   if (size==1) return;
   if (rank==1){
      for (int i=2; i<=size;i++) t.send(i,tagFromTo(1,i));
   }
   else  t.rec(1,tagFromTo(1,rank));
}

template <class T, class V>
void gather(const T& t, V& v)const
// The out-commented messages made me aware of the actual timing of
// processes in my emulating MPICH runs: Writing of moviefiles for some
// processes may be in progress long before function step() is done for
// all tasks.
{
   if (size==1){ v[1]=t; return;}
   if (rank!=1){
      t.send(1,tagFromTo(rank,1));
   }
   else{
      v[1]=t;
      for (int i=2;i<=size;i++) v[i].rec(i,tagFromTo(i,1));
   }
}

template <class T, class V>
   // see sendAndRec() for the assumed properties of T and V
void gatherXL(const T& t, V& v, int n)const
   // clear from code
{
   if (size==1){ v[1]=t; return;}
   if (n<2) {
      gather(t,v);
      return;
   }
```

```
      if (rank!=1){
         t.sendXL(1,tagFromTo(rank,1),n);
      }
      else{
         v[1]=t;
         for (int i=2;i<=size;i++) v[i].rec(i,tagFromTo(i,1));
      }
   }

   template <class T, class V>
      // see sendAndRec() for the assumed properties of T and V
   void scatter(T& t, const V& v)const
      // clear from code, not yet used or tested
   {
      if (rank==1){ // directly OK for size==1
         t=v[1];
         for (int i=2;i<=size;i++) v[i].send(i,tagFromTo(1,i));
      }
      else{
         t.rec(1,tagFromTo(1,rank));
      }
   }

   template <class T, class V>
   void scatterXL(T& t, const V& v, int n)const
      // clear from code, not yet used or tested
   {
      if (rank==1){
         t=v[1];
         for (int i=2;i<=size;i++) v[i].sendXL(i,tagFromTo(1,i),n);
      }
      else{
         t.rec(1,tagFromTo(1,rank));
      }
   }
};

#else // not defined(CPM_USE_MPI)

class Com{ //trivial implementation of Com

public:
   static int tagFromToStatic(int f, int t,int size_)
   {f;t;size_;return 0;}
   static int tagFromTo(int f, int t){ f;t;return 0;}
   Com(){}
   int getSize()const{ return 1;}
   int getRank()const{ return 1;}
   bool sendStr(const string& s, int dest, int tag)const
   {s;dest;tag; return true;}
```

```
   bool bcastStr(string& s)const{s; return true;}
   bool recStr(string& s, int source, int tag)const
      {s; source; tag; return true;}
   int cyc(int i)const{ i;return 1;}

   template <class T, class V>
      void sendAndRec(const T& t, V& v)const{t;v;}

   template <class V>
      void exchange(const V& s, V& v)const
      { v[1]=s[1];}

   template <class T>
      void bcast(T& t)const{t;}

   template <class T, class V>
      void gather(const T& t, V& v)const{t;v;}

   template <class T, class V>
      void gatherXL(const T& t, V& v, int n)const{t;v;n;}

   template <class T, class V>
      void scatter(T& t, const V& v)const{t;v;}

   template <class T, class V>
      void scatterXL(T& t, const V& v, int n)const{t;v;n;}
};

#endif // #if defined(CPM_USE_MPI)

extern CpmMPI::Com Cpm_com;
void initialize(int* a, char*** b);
void finalize();

} // namespace

#endif // #if defined(CPM_MPI_H)
```

26 cpmmpi.cpp

```
//? cpmmpi.cpp
//? Status of work 2023-10-20.
//?
//?  ...


/*
   cpmmpi.cpp

   Description: see cpmmpi.h
      can be made part of the source file set even if
      CPM_USE_MPI is not defined
*/
#include <cpmmpi.h>

int CpmMPI::mpiverbose=1; // experiment, 1 is normal

namespace{
   const int mL1=10;
   const int mL2=20;
}

   // limits against which mpiverbose will be tested
CpmMPI::Com CpmMPI::Cpm_com;
   // initialized by default constructor

#if defined(CPM_USE_MPI)

#include <cpmsystem.h>

using namespace CpmMPI;

using CpmRoot::Word;

using std::ostringstream;
using std::endl;

void CpmMPI::initialize(int* a, char*** b){ MPI_Init(a,b);}

void CpmMPI::finalize(){ MPI_Finalize();}

CpmMPI::Com::Com(MPI_Comm comm):mc(comm)
{
   int mpiflag;
   MPI_Initialized(&mpiflag);
   if(mpiflag) { // 01-09-25: mpiflag introduced by John Zollweg,
      // Cornell university, to avoids warnings
```

```
      MPI_Comm_size(comm,&size);
      MPI_Comm_rank(comm,&rank);
      ++rank;
   }
}

int CpmMPI::Com::cyc(int i)const
{
   while(i>size) i-=size;
   while(i<1) i+=size;
   return i;
}

// not working code for non-blocking sends and receives is commentarized
bool CpmMPI::Com::sendStr(const string& s, int dest, int tag)const
{
   static int shL=100; // show length, 100 is normal
   static int counter=0;
   counter++;
   int n=1+(int)s.size();
   if (mpiverbose>mL1){
      ostringstream mes;
      mes<<"sendStr( dest="<<dest<<" tag="<<tag<<") started n="
         <<n<<"; send counter="<<counter;
  //    if (counter==21) mes<<" the string to send is "<<s; // experiment
  // ad hoc
      cpmmessage(Word(mes.str()));
   }
   int tag1=tag;
   int tag2=tag+1;
   int* np=new int(n);
 //  MPI_Request request;
 //  MPI_Isend(np,1, MPI_INT, dest-1, tag1, mc,&request);
   MPI_Ssend(np,1, MPI_INT, dest-1, tag1, mc);
   if (mpiverbose>mL2){
      ostringstream mes;
      mes<<"sending the number done, tag="<<tag1;
      mes<<" next send will use tag="<<tag2;
      cpmmessage(Word(mes.str()));
   }
   delete np;
   if (mpiverbose>mL2){
      ostringstream mes;
      if (n<=shL)
         mes<<endl<<"string to send is: "<<s;
      else
         mes<<endl<<"string to send not shown since longer than "<<shL;
      mes<<endl<<"end of string to send";
      cpmmessage(Word(mes.str()));
   }
```

```
  int res= MPI_Ssend((void*)s.c_str(), n, MPI_CHAR, dest-1, tag2, mc);
  // int res= MPI_Isend((void*)s.c_str(), n, MPI_CHAR, dest-1, tag2,
  // mc,&request);

   if (mpiverbose>mL2){
      ostringstream mes;
      mes<<"sending the string done";
      cpmmessage(Word(mes.str()));
   }

   bool b=(res==MPI_SUCCESS);
   if (mpiverbose>mL1){
      ostringstream mes;
      mes<<"sendStr( dest="<<dest<<" tag="<<tag<<"), n="<<n<<
         " done; send counter="<<counter;
      cpmmessage(Word(mes.str()));
   }
   if (!b) cpmerror("error in CpmMPI::sendStr");
   return b;
}

bool CpmMPI::Com::recStr(string& s, int source, int tag)const
{
   static int counter=0;
   counter++;
   if (mpiverbose>mL1){
      ostringstream mes;
      mes<<"recStr( source="<<source<<" tag="<<tag<<") started"
         <<"; receive counter="<<counter;
      cpmmessage(Word(mes.str()));
   }
   int tag1=tag;
   int tag2=tag+1;
   MPI_Status status;
  // MPI_Request request;
   int* np=new int;
   MPI_Recv(np,1,MPI_INT,source-1,tag1,mc,&status);
 //  MPI_Irecv(np,1,MPI_INT,source-1,tag1,mc,&request);
   // The non-blocking version does not work in my program, not clear why
   int n=*np;
   if (mpiverbose>=mL1){ // >= intentionally to have a control mode
      // in which only the most important events get reported
      // for all other tests we have mpiverbose>mL
      cpmmessage("string received of length n="&cpmwrite(n));
   }
   char* data=new char[n];
 //  int out=MPI_Irecv(data,n,MPI_CHAR,source-1,tag2,mc,&request);
   int out=MPI_Recv(data,n,MPI_CHAR,source-1,tag2,mc,&status);
   ostringstream ost;
```

```
   ost<<data;
   s=ost.str();
   if (mpiverbose>mL1){
      ostringstream mes;
      mes<<"recStr( source="<<source<<" tag="<<tag<<") done; n="
         <<n<<"; receive counter="<<counter;
      if (mpiverbose>mL2){
         mes<<endl<<"begin of string received "<<endl;
         mes<<s;
         mes<<endl<<"end of string received";
      }
      cpmmessage(Word(mes.str()));
   }
   bool b=(out==MPI_SUCCESS);
   delete[] data;
   delete np;
   if (!b) cpmerror("error in CpmMPI::recStr");
   return b;
}

//////// some collective communication ////////////////////////////////
bool CpmMPI::Com::bcastStr(string& s)const
{
   if (mpiverbose>mL1){
      ostringstream mes;
      mes<<"bcastStr() started";
      cpmmessage(Word(mes.str()));
   }
   bool b=true;
   if (rank==1){
      for (int i=2; i<=size;i++){
         b=b&&sendStr(s,i,size+i);
      }
   }
   else{
      b=b&&recStr(s,1,size+rank);
   }
   if (!b) cpmerror("error in CpmMPI::bcastStr");
   if (mpiverbose>mL1){
      cpmmessage("bcastStr() done");
   }
   return b;
}

#else // !defined(CPM_USE_MPI)

void CpmMPI::initialize(int* a, char*** b){a;b;}

void CpmMPI::finalize(){}
```

```
#endif  // #if defined(CPM_USE_MPI)
```

27 cpmnumbers.h

```
//? cpmnumbers.h
//? Status of work 2023-10-20.
//?
//?  ...


#ifndef CPM_NUMBERS_H_
#define CPM_NUMBERS_H_
/*
   Description: The C+- class system needs some basic types from C++.
      These are in the first place: (long) int, unsigned (long) int,
      unsigned char, bool, string, and (long) double.
      With the availability of convenient and efficient C++ types for
      floating point numbers of 'multiple precision' a new degree of
      freedom can be added to the C+- class system. We therefore add
      the multiple precision class mpfr::mpreal (by Pavel Holoborodko)
      which is based on the multiple precision libraries gmp and mpfr.
      This class can be used if the macro CPM_MP is defined in the header
      file cpmdefinitions.h.

      Introduces a common infrastructure for some basic types.
      There are two topics of interest here:
      1. It is a common situation that a class is to be defined in a way
      that some (or all) its data members have types which belong to these
      basic types. Then the methods should be in place which allow to
      carry over some basic infra structure from the members to the
      class to be defined. Most frequently used in present C+- code is
      the implementation of stream I/O ('Marshalling') according to the
      following example:

      class X{
         B1 x1; // B1,B2,...,Bn basic types
         B2 x2;
         ...
         Bn xn;
         typedef X Type;
      public:
         CPM_IO
            // declaration macro for Input/Output
         Word nameOf()const{ return "X";}
            //: name of
            ...
      };

      bool X::prnOn(str)const
      {
         cpmwat; // uses the nameOf - function to write an
```

```
      // automatic 'title'
   cpmp(x1); // 'p' for 'print'
   cpmp(x2);
   ...
   cpmp(xn);
   return true;
}

bool X::scanFrom(str)
{
   cpms(x1); // 's' for 'scan'
   cpms(x2);long
   ...
   cpms(xn);
   return true;
}

The declaration macro CPM_IO and the implementation macros
cpmwat, cpmp, cpms are defined in cpminterfaces.h.
The main achievement is the recursive nature of the scheme:
If X is the type of a member of a class Y the same mechanism
works:

class Y{
   X x1;
   ...
   typedef Y Type;
public:
   CPM_IO
   Word nameOf()const{ return "Y";}
   ...
};

bool Y::prnOn(str)const
{
   cpmwat; // uses the nameOf - function to write an
      // automatic 'title'
   cpmp(x1); // 'p' for 'print'. According to the
      // definition of X this makes sense as a part of
      // the implementation of Y's stream interface.
      ...
   return true;
}

bool Y::scanFrom(str)
{
   cpms(x1); // 's' for 'scan'
     ...
   return true;
}
```

```
      The details of the scheme will be given later in this file.

      2. Functions which enable unbiased testing of mathematical
      properties especially of class templates.

         ... not yet finished

      Z, N, R, L, bool, string (summerized as bb-types:
      basic built-in types). Thus no longer restricted to numbers.
      Provides most of the content of CpmRoot, cpmword adds the
      rest.

      We try to avoid the unsigned number types and thus don't introduce
      Nh, Nd as in earlier versions of this class library.
      (See. Bjarne Stroustroup: The C++ Programming Language,
      Third Edition, Addison-Wesley p. 70-71 last and first few lines.
      This book will referred to as BS3, the second edition as BS2.
      At present we have a fourth edition BS4.
      Certainly it is highly desirably to have the basic number
      types defined as classes. These seems however be possible only by
      reducing speed by factor 8, or so, in numerics intensive
      programs. This is not considered acceptable here.
      More recent experiments (in 2010) gave a speed loss of only 20% upon
      speed-optimized compilation. So the previous judgement seems no
      longer to have a very firm basis.

      We do not yet make use of the class CpmRoot::Word in this file.
*/

// making available the basic C++ standard library facilities
#include <string>
#include <iostream>
#include <fstream>  // for ifstream, ofstream
#include <sstream>
#include <cmath>
   // all math functions in namespace std ???
   // They can be also used as if they were in the global namespace.
   // This is important that we don't need a directive
   // 'using namespace std' in this file (as I had prior to 2016-08-29
   // and got a clash with a probably newly introduced function
   // std::ignore.
#include <limits>
#include <functional>

//#include <cpmbasictypes.h>
   // Defines integer types Z, N , L.
   // Includes cpmdefinitions.h from which defines (or does not
   // define) the macro CPM_MP which controls the usage of
   // 'multiple precision' floating-point numbers. Now in
```

```
   // cpmbasicinterfaces.h

#include <cpmbasicinterfaces.h>
   // defines macros CPM_IO and CPM_ORDER
   // includes <cpmbasictypes.h> since CPM_ORDER uses Z

#if defined(CPM_MP)
   #if defined(CPM_USE_EIGEN)
      #include<boost/multiprecision/eigen.hpp>
   #endif
   #include <boost/multiprecision/gmp.hpp>
   #include <boost/math/special_functions/round.hpp>
   #include <boost/math/special_functions.hpp>
   #include <boost/math/special_functions/bessel.hpp>
   #include <boost/math/constants/constants.hpp>
#elif defined(CPM_QUAD)
   #include <boost/multiprecision/float128.hpp>
#else
   #include <boost/math/special_functions/round.hpp>
   #include <boost/math/special_functions.hpp>
   #include <boost/math/special_functions/bessel.hpp>
   #include <boost/math/constants/constants.hpp>
#endif

namespace CpmStd{
// This is the part of the standard library which we use frequently.
// With C++11 we replace F1,F2,... with the replace of std::bind and
// namespace std::placeholders.
   using std::string;
   using std::ostream;
   using std::istream;
   using std::iostream;
   using std::ofstream;
   using std::ifstream;
   using std::fstream;
   using std::ostringstream;
   using std::istringstream;
   using std::stringstream;
   using std::endl;
   using std::cout;
   using std::cin;
   using std::bind;
   using namespace std::placeholders;
 } // CpmStd

namespace CpmRoot{
   using namespace CpmStd;

// Defining type R, the real type with which present C+- code uses
// in all places where floating point values are needed. No longer
```

```
// I use floating point types of multiple and fixed precision (such
// as R together with Rh or R_) in parallel.
#if defined(CPM_MP) // MP stands here for 'multiple precision'.
   // a typical line to appear in cpmdefinitions.h is
   // #define CPM_MP 32
   // where the number gives the number of decimal digits.
   // Here we use the tools from boost/multiprecision. These interface
   // well with boost/math but show strange behavior with respect to
   // C++ conversion operators and create a lot of warnings which at
   // the end of building are reported as errors by the Code::blocks
   // IDE. Nevertheless the result of building is valid and works.
   // Standard functions such as sqrt are also in this namespace.
   // No longer any warnings with g++-10.

    using namespace boost::multiprecision;
    using namespace boost::math;

    typedef number<gmp_float<CPM_MP>, et_on > R;
      //typedef number<gmp_float<CPM_MP>, et_on > R;
      // Notice the et_off-switch which disables expression-templates.
      // If these would left active compilation of nearly all of my C+-
      // code basis would fail. Since the Eigen library is said to build
      // heavly on expression templates disabling expression templates
      // probably compromises the efficiency of Eigen. Should be clarified!
      // Notice here numerical precision is set at compile time and can
      // not be changed by reading input from cpmconfig.ini.
#else
   // Then we don't need the multiple precision libraries and get
   // more speed. This was the only mode of C+- till March 2009. We have
   // the choice between double (64 bit) and long double (80 bit on most
   // systems) as a definition of type R.
   #if defined(CPM_FLOAT) //
      typedef float R;
   #elif defined(CPM_LONG) // then: typedef long int Z
      typedef long double R;
   #elif defined(CPM_QUAD)
      using namespace boost::multiprecision;
      using namespace boost::math;
      typedef number<float128_backend, et_on>  R;
   #else
      typedef double R;
   #endif
  // using namespace std; // experiment
#endif

   R operator""_R(long double);
   R operator""_R(unsigned long long int);

// Remark on using CPM_LONG:
// C+- often overloads functions on the basis of argument type R or Z.
```

```
// As an example consider the following constructors in cpmviewport.h:
// CpmGraphics::rgb(Z r_, Z g_, Z b_, bool norm=true);
// CpmGraphics::rgb(R r_, R g_, R b_, bool norm=true);
// For R=double one can distinguish the two by calling e.g.
// rgb c1(255.,128.,10.); for the R-version, and
// rgb c2(0,0,255); for the Z-version.
// For R=long double the code will become ambiguous since interpreting
// 255 as long int is not considered better than interpreting it as long
// double. Better one avoids numerical literals as function arguments
// and writes
// R r=255, g=128, b=10; Z z0=0,z1=255;
// rgb c1(r,g,b); rgb c2(z0,z0,z1);
// This gives all quantities their matching type and is as readable as
// rgb c1(R(255),R(128),R(10)); rgb c2(Z(0),Z(0),Z(255));

// Two implementation tools:

inline R disDefFun(R a, R b, R d)
   //: distance defining function
   // Function for implementing function dis.
   // Needs also be known to cpminterfaces.h.
   // Continuous function, but not reasonable from a geometrical
   // point of view, since the distance dis(a,b):=disDefFun(|a|,|b|,|a-b|)
   // between points based naturally on this function is not translation
   // invariant.
{
   R s=a+b;
   if (s==0.) return R(0.);
   R d1=d/s;
   return (d < d1 ? d  : d1);
}

template <class T>
T posVal(T t){ return t<T() ? -t : t;}
   //: positive value
   // added 2005-09-10. Similar to abs but not necessarily
   // R-valued. Not intended to be used for aggregated data types

// Defining the basic R-related functions as class-less
// functions in namespace CpmRoot.

void prec(Z p);
   // Setting precision; a typical numerical error is ~ 10^(-p).
   // Thus p is a number of decimal digits.
   // Always defined, but active only if
   //    defined(CPM_MP)

Z getPrec(void);
   // Returns the active value of precision.
   // If !defined(CPM_MP) the corresponding value for type
```

```
   // double or long double (i.e. the actual type of R) is returned.
R inv(R const& r);
   // returns r==0. ? 0 : 1./r with warnings in the case r==0.
R randomR(Z j=0);
   // Returns uniformely distributed random values in [0,1) if
   // no argument (or argument 0) is used. For j != 0 the return
   // value is a 'chaotic' function of j, so that for any sequence
   // of argument values we get a random sequence. Since the value
   // is determined by j, one gets reproducible results which sometimes
   // is useful. In March 2013 I tested {0,1}-valued sequences of length
   // 1000000 with the NIST Random Number Test Suite and found all tests
   // passed convincingly. I used the NIST suite as translated to
   // Mathematica by Ilja Gerhardt in 2010 and I translated randomR also
   // to Mathematica.
R test(R const& r, Z i);
   // Implementation tool for class Test<>, see this file.
R ran(R const& r, Z i=0);
   // Implementation tool for class Ran<>, see this file.
   // The return value x satisfies -|r| <= x < |r|
Z hash(R const& r);
   // Implementation tool for class Hash<>, see this file.
R dis(R const& r1, R const& r2);
   // Implementation tool for class Dis<>, see this file.
Z getPrec(R const& r);
   // Returns the number of decimal digits used to hold the value of r.
bool isVal(R const& r);
   // Returns 'false' if r is not a regular number (infinite or NaN).
Z toZ(R const& r, bool toZero=false);
   // For positive r we return floor(r) as a Z instead of a R.
   // For negative r, the second argument determines whether the next
   // integer in minus-direction (toZero=false) or in plus-direction
   // (toZero=true) is returned. So, for toZero==false, the result is
   // floor(r) also for negative r.
// functions with inline implementation
inline Z sgn(R const& r)
{
   if (r>0.) return 1;
   if (r<0.) return -1;
   return 0;
}

inline R hypot(R const& x, R const& y)
// hypotenuse, a numerically careful version of sqrt(x*x + y*y).
{
   R tiny=1.e-12;
   R xa=posVal<R>(x),ya=posVal<R>(y),xya=xa+ya;
   if (xya==R(0.)) return R(0.);
   xya=(xya<tiny ? tiny : xya);
   R xyaInv=CpmRoot::inv(xya);
   R xr=xa*xyaInv, yr=ya*xyaInv;
```

```
   return R(xya*sqrt(xr*xr+yr*yr));
}

inline R con(R const& r){ return r;}
   // Implementation tool for class Conj<>, see this file.

inline R net(R const& r, Z i){ if (i==1) return R(1.); else return R(0.);}
   // Implementation tool for class Neutrals<>, see this file.

inline Z rnd(R const& r)
   // Returns the nearest integer value (rounding to integer).
{
#if defined(CPM_MP)
      return lround(r);
#else
   return (Z)floor(r+0.5);
#endif
}

//: division (of integers with remainder)
inline std::pair<Z,Z> div(Z x, Z a)
{
   auto q=std::div(x,a);
   return  std::pair<Z,Z>(q.quot, q.rem);
}

//: division (of unsigned integers with remainder)
inline std::pair<N,N> div(N x, N a)
{
   N quot=x/a;
   N rem=x%a;
   return  std::pair<N,N>(quot,rem);
}

//. remainder
// a assumed to be positive. This function matches the behavior of Mod in
// Wolfram Language.
inline Z modWL(Z x, Z a)
{
   if (x>=0_Z){
      return x%a;
   }else{
      Z q=x/a;
      Z qL=q-1_Z;
      return x-qL*a;
   }
}

inline bool isZero(R const& r){ return r==R(0.);}
   // Returns 'true' if r==0, and 'false' else.
```

```
inline double toDouble(R const& r){
   // to double, helps to program uniformly formatted screen output
   return static_cast<double>(r);
}

inline R absSqr(R const& r){ return r*r;}
   //: absolute (value) squared
   // Returns the square of |r| (which happens to be the square of r)

inline R arg(R const& x, R const& y)
   //: argument
   // Returns the polar angle (in (-Pi,Pi]) of point (x,y).
   { return atan2(y,x);}

// Split implementation for circle related matters.
#if !defined(CPM_MP)
   const R Pi=asin(0.5)*6.0;
      // Number pi=3.14159..., pi = 30*deg * 6, sin(30*deg) = 0.5.
      // Faster convergence expected then for Pi = atan(1.)*4.
      // This is certainly not an important issue in this place.
   const R Pi2=Pi*2;
      // 2 * pi
   const R AngDeg= Pi/180.;
      //: angular degree, pi/180
#endif

   R operator""_R(long double r);
   R operator""_R(unsigned long long int r);

} // CpmRoot closed, will be re-opened soon



// Convenient abbreviations for constants and service functions.
// See also corresponding abbreviations
// cpmnam, cpmtow in cpmword.h and
// cpmswp, cpmord, cpminf, cpmsup, cpmtin, cpmhug in cpmtypes.h.

#if defined CPM_MP
   #define cpmpi     R(boost::math::constants::pi<CpmRoot::R>())
   #define cpmpiInv  R(R(1.)/boost::math::constants::pi<CpmRoot::R>())
   #define cpm2pi    R(boost::math::constants::pi<CpmRoot::R>()*R(2.))
   #define cpmdeg    R(boost::math::constants::pi<CpmRoot::R>()/R(180.))
   #define cpmrho    CpmRoot::hypot
#else
   #define cpmpi     CpmRoot::Pi
   #define cpmpiInv  1./CpmRoot::Pi
   #define cpm2pi    CpmRoot::Pi2
   #define cpmdeg    CpmRoot::AngDeg
```

```
   #define cpmrho    CpmRoot::hypot
#endif

// easy access to C+- utility functions
// Notice that we are here not in CpmRoot.
#define cpmrnd    CpmRoot::rnd
#define cpmtoz    CpmRoot::toZ
#define cpmtod    CpmRoot::toDouble
#define cpmpos    CpmRoot::posVal
#define cpmprn    CpmRoot::prnOn
#define cpmscn    CpmRoot::scanFrom
#define cpmsgn    CpmRoot::sgn
#define cpmcom    CpmRoot::com
#define cpmcon    CpmRoot::con
#define cpmdis    CpmRoot::dis
#define cpmran    CpmRoot::ran
#define cpmtes    CpmRoot::test
#define cpmhas    CpmRoot::hash
#define cpmnet    CpmRoot::net
#define cpmiva    CpmRoot::isVal
#define cpminv    CpmRoot::inv
#define cpmab2    CpmRoot::absSqr
#define cpmarg    CpmRoot::arg
#define cpmprec    CpmRoot::prec
#define cpmgetprec CpmRoot::getPrec

// Easy access to mathematical functions of a real argument.
// CPM_ is either std or boost::multiprecision depending on CPM_MP.
// Since the function-names begin with 'cpm' it should not harm that they
// are defined here in the global namespace.
// Since these function names are defined already in namespace std it would
// be not convenient to define them first in CpmRoot and that use, for
// instance,
// #define cpmsin    CpmRoot::sin
// Instead, we define a function with name cpmsin.

#if defined(CPM_MP)||defined(CPM_QUAD)
   inline CpmRoot::R cpmabs(CpmRoot::R const& x)
      { return boost::multiprecision::abs(x);}
   inline CpmRoot::R cpmfloor(CpmRoot::R const& x)
      { return boost::multiprecision::floor(x);}
   inline CpmRoot::R cpmceil(CpmRoot::R const& x)
      { return boost::multiprecision::ceil(x);}
   inline CpmRoot::R cpmsin(CpmRoot::R const& x)
      { return boost::multiprecision::sin(x);}
   inline CpmRoot::R cpmasin(CpmRoot::R const& x)
      { return boost::multiprecision::asin(x);}
   inline CpmRoot::R cpmsinh(CpmRoot::R const& x)
      { return boost::multiprecision::sinh(x);}
   inline CpmRoot::R cpmcos(CpmRoot::R const& x)
```

```
      { return boost::multiprecision::cos(x);}
   inline CpmRoot::R cpmacos(CpmRoot::R const& x)
      { return boost::multiprecision::acos(x);}
   inline CpmRoot::R cpmcosh(CpmRoot::R const& x)
      { return boost::multiprecision::cosh(x);}
   inline CpmRoot::R cpmtan(CpmRoot::R const& x)
      { return boost::multiprecision::tan(x);}
   inline CpmRoot::R cpmtanh(CpmRoot::R const& x)
      { return boost::multiprecision::tanh(x);}
   inline CpmRoot::R cpmatan(CpmRoot::R const& x)
      { return atan(x);}
   inline CpmRoot::R cpmatan2(CpmRoot::R const& y, CpmRoot::R const& x)
      { return boost::multiprecision::atan2(y,x);}
   inline CpmRoot::R cpmexp(CpmRoot::R const& x)
      { return boost::multiprecision::exp(x);}
   inline CpmRoot::R cpmsqrt(CpmRoot::R const& x)
      { return boost::multiprecision::sqrt(x);}
   inline CpmRoot::R cpmlog(CpmRoot::R const& x)
      { return boost::multiprecision::log(x);}
   inline CpmRoot::R cpmlog10(CpmRoot::R const& x)
      { return boost::multiprecision::log10(x);}
   inline CpmRoot::R cpmpow(CpmRoot::R const& a, CpmRoot::R const& b)
      { return boost::multiprecision::pow(a,b);}
#if !defined(CPM_QUAD)
   inline CpmRoot::R cpmbessel(CpmRoot::Z const& n, CpmRoot::R const& x)
      { return boost::math::cyl_bessel_j(n,x);}
#else
    inline CpmRoot::R cpmbessel(CpmRoot::Z const& n, CpmRoot::R const& x){
   //    cpmwarning("cpmbessel(Z,R): not implemented, 0 returned ");
      using namespace CpmRoot;
      return 0_R;
   }
#endif

#else // double or float
   inline CpmRoot::R cpmabs(CpmRoot::R const& x)
      { return std::abs(x);}
   inline CpmRoot::R cpmfloor(CpmRoot::R const& x)
      { return std::floor(x);}
   inline CpmRoot::R cpmceil(CpmRoot::R const& x)
      { return std::ceil(x);}
   inline CpmRoot::R cpmsin(CpmRoot::R const& x)
      { return std::sin(x);}
   inline CpmRoot::R cpmasin(CpmRoot::R const& x)
      { return std::asin(x);}
   inline CpmRoot::R cpmsinh(CpmRoot::R const& x)
      { return std::sinh(x);}
   inline CpmRoot::R cpmcos(CpmRoot::R const& x)
      { return std::cos(x);}
   inline CpmRoot::R cpmacos(CpmRoot::R const& x)
```

```
      { return std::acos(x);}
   inline CpmRoot::R cpmcosh(CpmRoot::R const& x)
      { return std::cosh(x);}
   inline CpmRoot::R cpmtan(CpmRoot::R const& x)
      { return std::tan(x);}
   inline CpmRoot::R cpmtanh(CpmRoot::R const& x)
      { return std::tanh(x);}
   inline CpmRoot::R cpmatan(CpmRoot::R const& x)
      { return std::atan(x);}
   inline CpmRoot::R cpmatan2(CpmRoot::R const& y, CpmRoot::R const& x)
      { return std::atan2(y,x);}
   inline CpmRoot::R cpmexp(CpmRoot::R const& x)
      { return std::exp(x);}
   inline CpmRoot::R cpmsqrt(CpmRoot::R const& x)
      { return std::sqrt(x);}
   inline CpmRoot::R cpmsqr(CpmRoot::R const& x)
      { return x*x;}
   inline CpmRoot::R cpmlog(CpmRoot::R const& x)
      { return std::log(x);}
   inline CpmRoot::R cpmlog10(CpmRoot::R const& x){
      if (x==CpmRoot::R(0.)) return CpmRoot::R(-16.);
      else if(x<CpmRoot::R(0.)) return std::log10(-x);
      else return std::log10(x);
   }
   inline CpmRoot::R cpmpow(CpmRoot::R const& a, CpmRoot::R const& b)
      { return std::pow(a,b);}
   inline CpmRoot::R cpmbessel(CpmRoot::Z const& n, CpmRoot::R const& x)
      { return std::cyl_bessel_j(n,x);}
#endif

// namespace re-opened
namespace CpmRoot{

// Numerical precision matters.
   extern Z numPrc;
      //: numerical precision
      // value for decimals used for the internal representation
      // of real numbers of type R.
// Formatting matters.

   extern Z wrtPrc;
      //: writing precision
      // value for decimals written to strings for real
      // numbers of type R. Notice that all data communicated
      // between processes in my MPI-based parallel computing
      // functions use writing on streams; so numerical errors in
      // communication are ignorable only if this number is large
      // e.g. 16. Now these functions set wrtPrc high and reset
      // it afterwards to a normal value (automatically!)
```

```
   bool writeTitle(const string& , ostream&);
      // write title
      // Uses the string argument to write a title which is precceced
      // by a comment line indicator. Is under the control of wrtTit.

   bool startsComment(char c);
      // starts comment
      // Returns true if c is a character which gives a line the
      // meaning of a comment if it is its first non-whitespace
      // character.
      // Presently startsComment(c)==true for
      // c \in { ';','/','#','*'}

   bool eatComments(istream& in);
      // eat comments
      // Returns false if no character not belonging to a comment
      // was found. If return is true, the stream is now in a
      // state that the next reading access will find something
      // not belonging to a comment.

// Unified interface to types Z,N,R,L,bool,string,
// which we refer to as bb-types (basic built-in types). Notice that the
// types Z, N, and L are defined via typedef and thus are no classes.
// R may be a class - depending on the macro CPM_MP.
// The types bool and string from standard C++ are useful as they stand.
// Nevertheless they will be wrapped into classes - CpmRootX::B in
// cpmtypes.h and class CpmRoot::Word in file cpmword.h.
// --------

   bool strVal(istream& str);
      //: stream validity
      // returns stream status

   bool strVal(ostream& str);
      //: stream validity
      // returns stream status

// Topic 1: reading instances of b-types from streams that may contain
// comments.

   template <class T>
   bool read(T& t, istream& in)
   //: read
   // reads a T from a istream; in doing so it looks out for characters
   // having the meaning of 'comment indicators'. After having identified
   // such a character, the reading action will jump to the beginning of
   // the next line (thus not reading the rest of the previous line,
   // which is supposed to be a comment not containing data for reading).
   // Of course, a comment indicator there, will result in skipping that
   // line too. See function startsComment for the characters that are
```

```
   // considered comment indicators.
   // If no T can be read the function returns
   // false otherwise true. Providing a program with numerical data from
   // a text file created by the programmer is useful only if the file is
   // structured by comments explaining the meaning of the data. Thus
   // reading from files containing comment lines is crucial.
   // On ENUMERATIONS:
   // Not only quantities of type Z but also enumerations will be
   // read by this function. Then the syntax of writing and reading
   // should be as follows:
/*
   enum Corners{LL,UL,LR,UR};
   Corners c;
   write((Z)c,cout);
   ...
   Z temp;
   read(temp,cin);
   Corners c2=(Corners)temp;
*/
   {
      if  (!eatComments(in)) return false;
      if (in>>t) return true; else return false;
   }

   template <>
   bool read(L& x, istream& in);

   bool readLine(string& str, istream&);
   // reads a non-comment line from a istream into str. Trailing
   // whitespace such as the most annoying such species CR (carriage
   // return) will not find a way to spoil str. Leading whitespace
   // will not be ignored here since it is essential for copying
   // texts in a readable form by reading and printing lines

// Topic 2: writing instances of b-types to streams
// in a similar syntax, to make large prnOn and scanFrom functions of
// classes more alike

   template <class T>
   bool write(T const& t, ostream& out)
   {
      out<<t<<endl; return strVal(out);
   }

   template <>
   bool write(L const& t, ostream& out);

   template <>
   bool write(R const& t, ostream& out);
      // special treatment to avoid problems with small
```

```
      // numbers (which should not exist)

// Miscellanea

   bool isVal(L x);
   bool isVal(Z x);
   bool isVal(N x);
      // No modes of invalidity of types L, bool, or string are
      // known to me. The real types are already treated.

   L getByte(Z z, Z n);
      // z is an (typically 32 bit)-integer and n varies over 1,2,3,4
      // returned is the first, second, third, forth byte (counted from
      // left to right)

   string skipLeadingWhitespace(string const& str);
      // clear from name

   string skipTrailingWhitespace(string const& str);
      // clear from name

///////////////// interface service classes //////////////////////////////
// Unified interface to the bb-types
//    Z, N, R, L, bool, string
// and most of the other C+- classes.
// This uniform interface is given by the interface service class
// templates
//    IO<T>, Comp<T>, Inv<T>, AbsSqr<T>, Abs<T>, Dis<T>,
//    Test<T>, Ran<T>, Hash<T>, Conj<T>, Neutrals<T>,
//    ToWord<T>, Name<T> (these two to be added later in cpmword.h)
// For a class T to implement this interface means to define the following
// member functions ( here an T as argument may also mean T const&):
//    bool prnOn(ostream& str)const;         // print on
//    bool scanFrom(istream& str);           // scan from
//    Z com(T)const;                         // compare
//    T inv()const;                          // inverse
//    R absSqr()const;                       // absolute (value) squared
//    R abs()const;                          // absolute value
//    R dis(T)const;                         // distance value
//    T test(Z)const;                        // test value
//    T ran(Z)const;                         // random value
//    Z hash()const;                         // hash value
//    T con()const;                          // conjugate
//    Z net(Z)const;                         // neutrals ( = 0 and 1)
//    Word nameOf()const;                   // class name (see cpmword.h)
//    Word toWord()const;                   // string representation of
                                            // the value  (see cpmword.h)
// For the non-class b-types, the interface service class templates are
// subsequently defined as specializations.
```

```
////////////////////////// class IO<> ////////////////////////////////////
// Here we describe for the important example of stream IO
// (marshalling) the method by which C+- succeeds to treat
// the non-class b-types and C+- classes on equal
// footing when these are used as template arguments of
// C+- array classes.
// For a C+- class T it is normal to define the member functions
// (1) bool T::prnOn(ostream& str)const;
// (2) bool T::scanFrom(istream& str);
// Their declaration (together with the definition of some
// related non-member functions) is provided by using the
// macro CPM_IO after a statement 'typedef T Type;'
// For a C+- class template, e.g. CpmArrays::V<T> it is  n o t
// natural to assume that T defines the member functions
// (1) and (2) since T could be Z or R and thus not a class at all.
// Instead, all C+- class templates which implement
// interaction with streams assume that the template argument T
// defines functions
// (3) bool CpmRoot::IO<T>().o(T const& t, ostream& str)const;
// (4) bool CmRoot::IO<T>().i(T& t, istream& str)const;
// There are two possibilities for these functions to get defined:
// (i) It could be that functions (1),(2) are defined for T
//    e.g. since T is a C+- class implementing CPM_IO.
// Then the following un-specialized version of the class template IO<T>
// does the job.
// If, however, T is not a C+- class and cannot be modified by
// adding member functions (1),(2) then
// (ii) a specialization of the class template
//    IO has to make IO<T> defined: Such specializations
//    are given in the following for the  bb-types.
//    Partial specializations (in which T is a class template)
//    can be defined where needed. An example is the definition
//    of IO< std::vector<T> > in cpmv.h.
// Finally there are class-less function templates (near the end of the
// file)
// bool prnT(T const& t, ostream& str)
// bool scanT(T& t, istream& str)
// which will be called in the covenient read and write macros
// #define cpms(X) if (!CpmRoot::scanT((X),str)) return false
// #define cpmp(X) if (!CpmRoot::prnT((X),str)) return false
// fom file cpminterfaces.h. These functions can also be used in
// template code instead of the more clumpsy functions (3),(4).
// It should be noted that none of the interface service classes
// IO, Comp, Inv, AbsSqr, Abs, Dis, Test, Ran,
// Hash, Conj, Neutrals contains its own data, so that no transport
// of data is involved in their use. Notice that there are two more
// such service classes: Name and ToWord in cpmword.h.
// On Name the C+- type naming system is based, and ToName provides
// convenient short output string representation of values.
// The send and receive functions for MPI usage are based on the
```

```
// present IO-class (see macros CPM_MPI_O and #define CPM_MPI_I
// in cpminterfaces.h).
// For all such service classes except IO there is only one member
// function (not two, namely i and o, in IO) and this is chosen to
// be operator().

template<class T>
class IO{ // input output class template
public:
   IO(){}
   bool o(T const& t, ostream& str)const
      { return t.prnOn(str);}
   bool i(T& t, istream& str)const
      { return t.scanFrom(str);}
};

template<>
class IO<R>{ // specialization of IO
public:
   IO(){}
   bool o(R const& t, ostream& str)const
      { return CpmRoot::write(t,str);}
   bool i(R& t, istream& str)const
      { return CpmRoot::read(t,str);}
};

template<>
class IO<Z>{ // specialization of IO
public:
   IO(){}
   bool o(Z const& t, ostream& str)const
      { return CpmRoot::write(t,str);}
   bool i(Z& t, istream& str)const
      { return CpmRoot::read(t,str);}
};

template<>
class IO<N>{ // specialization of IO
public:
   IO(){}
   bool o(N const& t, ostream& str)const
      { return CpmRoot::write(t,str);}
   bool i(N& t, istream& str)const
      { return CpmRoot::read(t,str);}
};

template<>
class IO<L>{ // specialization of IO
public:
   IO(){}
```

```
   bool o(L const& t, ostream& str)const
      { return CpmRoot::write(t,str);}
   bool i(L& t, istream& str)const
      { return CpmRoot::read(t,str);}
};

template<>
class IO<bool>{ // specialization of IO
public:
   IO(){}
   bool o(bool const& t, ostream& str)const
      { return CpmRoot::write(t,str);}
   bool i(bool& t, istream& str)const
      { return CpmRoot::read(t,str);}
};

template<>
class IO<string>{ // specialization of IO
public:
   IO(){}
   bool o(string const& t, ostream& str)const
      { return CpmRoot::write(t,str);}
   bool i(string& t, istream& str)const
      { return CpmRoot::read(t,str);}
};

////////////////////////// class Comp<> ///////////////////////////////////
// class is named Comp instead of Com in order to avoid confusion
// with CpmMPI::Com
// Ruby style comparison
// com = 'compared' corresponds to the negative of
// Ruby's intelligent operator <=> (only the symbol '<=>' together
// with the ordered list -1,0,1 suggests Ruby's choice; if one is not
// bound by a symbol it is more natural to associate a<b (odered) with
// a.com(b)==1 and a>b (converse order) with a.com(b)==-1.
// Thus our normation:
// a.com(b) ==  0  for  a == b
// a.com(b) ==  1  for  a <  b
// a.com(b) == -1  for  a >  b

template<class T>
class Comp{ // compare
public:
   Comp(){}
   Z operator()(T const& x, T const& y){ return x.com(y);}
};

// Specializations building on > and < :
#define CPM_SC\
 if (x<y) return 1; if (x>y) return -1; return 0
```

```
template<>
class Comp<R>{
public:
   Comp(){}
   Z operator()(R const& x, R const& y){CPM_SC;}
};

template<>
class Comp<Z>{
public:
   Comp(){}
   Z operator()(Z const& x, Z const& y){CPM_SC;}
};

template<>
class Comp<N>{
public:
   Comp(){}
   Z operator()(N const& x, N const& y){CPM_SC;}
};

template<>
class Comp<L>{
public:
   Comp(){}
   Z operator()(L const& x, L const& y){CPM_SC;}
};

template<>
class Comp<bool>{
public:
   Comp(){}
   Z operator()(bool const& x, bool const& y){CPM_SC;}
};

template<>
class Comp<string>{
public:
   Comp(){}
   Z operator()(string const& x, string const& y){CPM_SC;}
};

#undef CPM_SC

////////////////////////// class Inv<> ////////////////////////////////////

template<class T>
class Inv{ // inverse
public:
```

```
   Inv(){}
   T operator()(T const& t)const{ return t.inv();}
};

template<>
class Inv<R>{ // inverse
public:
   Inv(){}
   R operator()(R const& t)const{ return cpminv(t);}
};

template<>
class Inv<Z>{ // inverse
public:
   Inv(){}
   Z operator()(Z const& t)const{ t; return Z(0);}
};

template<>
class Inv<N>{ // inverse
public:
   Inv(){}
   N operator()(N const& t)const{ t; return N(0);}
};

template<>
class Inv<L>{ // inverse, defined in an arbitrary manner
public:
   Inv(){}
   L operator()(L const& t)const{ return t;}
};

template<>
class Inv<bool>{ // inverse
public:
   Inv(){}
   bool operator()(bool const& t)const{ return !t;}
};

template<>
class Inv<string>{ // reverse string
public:
   Inv(){}
   string operator()(string const& t)const
   {
      string::const_reverse_iterator p1=t.rbegin(), p2=t.rend();
      return string(p1,p2);
   }
};
```

```
////////////////////////// class Test<> ////////////////////////////////
// The intented usage is as follows:
//    using CpmRoot;
//    T t;
//    Z complexity=3; // for example
//    T tc=Test<T>()(t,complexity);
//    T t1=Ran<T>()(tc,1);
//    T t2=Ran<T>()(tc,2);
//    T t3=Ran<T>()(tc,3);
// which is an unbiased way to create an arbitrarily large set of T-values.

template<class T>
class Test{ // test value
public:
   Test(){}
   T operator()(T const& t, Z complexity)const
   { return t.test(complexity);}
};

template<>
class Test<R>{
public:
   Test(){}
   R operator()(R const& x, Z complexity)const
   { return cpmtes(x,complexity);}
};

template<>
class Test<Z>{
public:
   Test(){}
   Z operator()(Z const& x, Z complexity)const
   { /*x;*/ return complexity;}
};

template<>
class Test<N>{
public:
   Test(){}
   L operator()(N const& x, Z complexity)const
   { x; Z cp=posVal(complexity); return N(cp);}
};

template<>
class Test<L>{
public:
   Test(){}
   L operator()(L const& x, Z complexity)const
   { x; Z cp=posVal(complexity); return L(cp%256);}
};
```

```
template<>
class Test<bool>{
public:
   Test(){}
   bool operator()(bool const& x, Z complexity)const
   { x; Z cp=posVal(complexity); return cp%2==1;}
};

template<>
class Test<string>{
public:
   Test(){}
   string operator()(string const& x, Z complexity)const
   {
      x;
      if (complexity<=1)
         return "az";
      else if (complexity==2)
         return "abcdefghijklmnopqrstuvwxyz";
      else if (complexity==3)
         return "abcdefghijklmnopqrstuvwxyz0123456789";
      else return
         "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789";
   }
};

////////////////////////// class Ran<> ////////////////////////////////////

Z randomZ(Z n, Z j=0);
   // returns uniformly distributed integers in the natural
   // index range [1,n]. Thus stops with error if n<1.
   // Function IvZ::ran allows to set lower limit and upper limit
   // arbitrarly.

template<class T>
class Ran{ // random value
public:
   Ran(){}
   T operator()(T const& t, Z j=0)const
   { return t.ran(j);}
};

template<>
class Ran<R>{
public:
   Ran(){}
   R operator()(R const& x, Z j=0)const
   {
     return cpmran(x,j);
```

```
   }
};

template<>
class Ran<Z>{
public:
   Ran(){}
   Z operator()(Z const& i, Z j=0)const
   {
      R y=randomR(j);
      Z ia=(i>=0 ? i : -i);
      if (ia==0) ia=1;
      // without this modification we would get 0,0,0,... for i=0. This
      // 'random' sequence will probably never be needed in a serious
      // context.
      // More probably i itself will result from a random process and
      // nevertheless ran(i,i2) will be expected to be random in i2.
      // Thus we increase ia in order to get the random sequence with
      // values in {-1,0,1}.
      Z n=ia*2+1;
      y*=n; // belongs to [0,2|i|+1)
      Z res=cpmtoz(y);
      if (res==n) res--; // should not happen
      return res-ia;
   }
};

template<>
class Ran<N>{
public:
   Ran(){}
   N operator()(N const& i, Z j=0)const
   {
      R y=randomR(j);
      Z ip=(i==0 ? 1 : i);
      // without this modification we would get 0,0,0,... for i=0. This
      // 'random' sequence will probably never be needed in a serious
      // context.
      // More probably i itself will result from a random process and
      // nevertheless ran(i,i2) will be expected to be random in i2.
      // Thus we increase ip in order to get the random sequence with
      // values in {0,1}.
      y*=ip; // belongs to [0,ip)
      Z res=cpmtoz(y);
      if (res==ip) res--; // should not happen
      return N(res); // belongs to {0,1} for i==0 and to {0,...i-1} else.
   }
};

template<>
```

```
class Ran<L>{
public:
   Ran(){}
   L operator()(L const& x, Z j=0)const
   {
      x;
      R y=randomR(j);
      y*=256; // belongs to [0,256)
      Z zy=cpmtoz(y); // belongs to {0,1,..., 255}=
      if (zy==256) zy--; // should not happen
      return (L)zy;
   }
};

template<>
class Ran<bool>{
public:
   Ran(){}
   bool operator()(bool const& b, Z j=0)const
   {
      R y=randomR(j);
      if (y<=0.5) return b; else return !b;
   }
};

// Here, all random strings have only printable ascii characters. See the
// definition of myChars for details.
template<>
class Ran<string>{
public:
   Ran(){}
   string operator()(string const& a, Z i=0)const
   {
      string res=a;
      Z l=i;
      string::const_iterator j;
      string::iterator k;
      const char myChars[] {'a','b','c','d','e','f','g','h','i','j','k',
      'l','m','n','o','p','q','r','s','t','u','v','w','x','y','z',
      'A','B','C','D','E','F','G','H','I','J','K',
      'L','M','N','O','P','Q','R','S','T','U','V','W','X','Y','Z',
      '0','1','2','3','4','5','6','7','8','9','_','.'};
      const int nc=64;
      for (j=a.begin(),k=res.begin();j!=a.end(),k!=res.end();++j,++k){
         int z = *j;
         int r = z%nc + 1;
         int ii = i;
         if (ii!=0){
            l++;
            ii+=l;
```

```
         }
         int q = randomZ(nc,ii) + r;
         *k = myChars[q%nc];
      }
      return res;
   }
};

//////////////////// class Hash<> ////////////////////////////////////
// If for T t1,t2 one has hash(t1)==hash(t2) there
// should be a overwhelming probability that t1 is equal
// to t2 (unless t2 is obtained from t1 by only a minor
// modification, or by a modification which is aware
// of the algorithm for hash).

template<class T>
class Hash{ // hash value
public:
   Hash(){}
   Z operator()(T const& t)const
   { return t.hash();}
};

template<>
class Hash<R>{
public:
   Hash(){}
   Z operator()(R const& x)const
   {
      return cpmhas(x);
   }
};

template<>
class Hash<Z>{
public:
   Hash(){}
   Z operator()(Z const& x)const
   {  return x-x%137+(x/13)*7;}
};

template<>
class Hash<N>{
public:
   Hash(){}
   N operator()(N const& x)const
   {  return x+x%137+(x/13)*7;}
};

template<>
```

```
class Hash<L>{
public:
   Hash(){}
   Z operator()(L const& x)const
   {
      Z y=(Z)x+117;
      return Hash<Z>()(y);
   }
};

template<>
class Hash<bool>{
public:
   Hash(){}
   Z operator()(bool const& x)const
   { return x ? 137 : 13;}
};

template<>
class Hash<string>{
public:
   Hash(){}
   Z operator()(string const& x)const
   {
      Z sum=137;
      string::const_iterator i;
      for (i=x.begin();i!=x.end();++i) sum+=(Z)(*i);
      return Hash<Z>()(sum);
   }
};

//////////////////// class Conj<> ////////////////////////////////////////
// This describes a concept that does not convincingly arrises
// from the types under present consideration. It will
// represent complex conjugation of complex numbers, transposition
// of real matrices, and Hermitean conjugation of complex
// matrices.

template<class T>
class Conj{ // conjugation
public:
   Conj(){}
   T operator()(T const& t)const
   { return t.con();}
};

template<>
class Conj<R>{
public:
   Conj(){}
```

```
   R operator()(R const& t)const
   { return cpmcon(t);}
};

template<>
class Conj<Z>{
public:
   Conj(){}
   Z operator()(Z const& t)const
   { return t;}
};

template<>
class Conj<N>{
public:
   Conj(){}
   N operator()(N const& t)const
   { return t;}
};

template<>
class Conj<L>{
public:
   Conj(){}
   L operator()(L const& t)const
   { return t;}
};

template<>
class Conj<bool>{
public:
   Conj(){}
   bool operator()(bool const& t)const
   { return t;}
};

template<>
class Conj<string>{
public:
   Conj(){}
   string operator()(string const& t)const
   { return t;}
};

////////////////////// class Neutrals<> ////////////////////////////////////
// for T t; the quantity Neutrals<T>(t,0) is the
// instance of T that is zero or comes as close as possible
// to realising the concept of a zero of type T.
// Correspondingly Neutrals<T>(t,1) with zero
// replaced by unity. The name comes from the fact that
```

```
// zero is a neutral element for addition and
// unity is a neutral element for multiplication.
// This renders the concept ambiguous for T==string since
// here we do refer only to addition (concatenation)
// but not to multiplication.

template<class T>
class Neutrals{ // neutral elements
public:
   Neutrals(){}
   T operator()(T const& t, Z i)const
   { return t.net(i);}
};

template<>
class Neutrals<R>{
public:
   Neutrals(){}
   R operator()(R const& t, Z i)const
   { return cpmnet(t,i);}
};

template<>
class Neutrals<Z>{
public:
   Neutrals(){}
   Z operator()(Z const& t, Z i)const
   { t; return i==1 ? 1 : 0;}
};

template<>
class Neutrals<N>{
public:
   Neutrals(){}
   N operator()(N const& t, Z i)const
   { t; return i==1 ? N(1) : N(0);}
};

template<>
class Neutrals<L>{
public:
   Neutrals(){}
   L operator()(L const& t, Z i)const
   { t; return i==1 ? L(1) : L(0);}
};

template<>
class Neutrals<bool>{
public:
   Neutrals(){}
```

```
   bool operator()(bool const& t, Z i)const
   { t; return i==1 ? true : false;}
};

template<>
class Neutrals<string>{
public:
   Neutrals(){}
   string operator()(string const& t, Z i)const
   { t; return i==1 ? "1" : "";}
};

// Now some classes will be defined which use in their interface
// a common real type, which is chosen as R. This type appears here
// as a return value of operator() of classes
// AbsSqr<>, Abs<>, and Dis<>. The definition of class Root<> in
// cpmword.h puts this role into perspective.
////////////////// class AbsSqr<> ////////////////////////////////////////

template<class T>
class AbsSqr{ // absolute (value) squared
public:
   AbsSqr(){}
   R operator()(T const& t)const{ return t.absSqr();}
};

template<>
class AbsSqr<R>{ // absolute (value) squared
public:
   AbsSqr(){}
   R operator()(R const& x)const{ return cpmab2(x);}
};

template<>
class AbsSqr<Z>{ // absolute (value) squared
public:
   AbsSqr(){}
   R operator()(Z const& x)const{ return R(x)*R(x);}
};

template<>
class AbsSqr<N>{ // absolute (value) squared
public:
   AbsSqr(){}
   R operator()(N const& x)const{ return R(x)*R(x);}
};

template<>
class AbsSqr<L>{ // absolute (value) squared
public:
```

```
   AbsSqr(){}
   R operator()(L const& x)const{
      Z y = static_cast<Z>(x);
      R r = y;
      return r * r;
   }
};

template<>
class AbsSqr<bool>{ // absolute (value) squared
public:
   AbsSqr(){}
   R operator()(bool const& x)const
      { return x==false ? R(0) : R(1);}
};

template<>
class AbsSqr<string>{
   // here string is interpreted as an array of L's
public:
   AbsSqr(){}
   R operator()(string const& x)const
   { // string components are cased to type L
      R res=0;
      string::const_iterator i;
      for (i=x.begin();i!=x.end();++i) res+=AbsSqr<L>()((L)(*i));
      return res;
   }
};

////////////////// class Abs<> ////////////////////////////////////

template<class T>
class Abs{ // absolute value
public:
   Abs(){}
   R operator()(T const& t)const{ return t.abs();}
};

template<>
class Abs<R>{ // absolute value
public:
   Abs(){}
   R operator()(R const& x)const{ return cpmabs(x);}
};

template<>
class Abs<Z>{ // absolute value
public:
   Abs(){}
```

```
   R operator()(Z const& x)const{ return x<0 ? R(-x) : R(x) ;}
};

template<>
class Abs<N>{ // absolute value
public:
   Abs(){}
   R operator()(N const& x)const{ return R(x);}
};

template<>
class Abs<L>{ // absolute value
public:
   Abs(){}
   R operator()(L const& x)const{ return R(x);}
};

template<>
class Abs<bool>{ // absolute value
public:
   Abs(){}
   R operator()(bool const& x)const{ return x==false ? R(0) : R(1) ;}
};

template<>
class Abs<string>{
   // here string is interpreted as an array of L's
public:
   Abs(){}
   R operator()(string const& x)const
   { return sqrt(AbsSqr<string>()(x)) ;}
};

////////////////////// class Dis<> ////////////////////////////////////
// For T t1,t2 the real number Dis<T>()(t1,t2) belongs to [0,1]
// and tries to assess whether t1 and t2 are significantly different
// or differ only by numerical noise. Thus for discrete types we
// set 0 for t1==t2 and 1 else. For 'continuous types' we use
// the natural construct dis(x,y)=Min( |x-y|, |x-y|/(|x|+|y|) )
// = distFunc(|x|,|y|,|x-y|)
// based on the concepts of absolute value and difference.

template<class T>
class Dis{
public:
   Dis(){}
   R operator()(T const& x, T const& y){ return x.dis(y);}
};

template<>
```

```
class Dis<R>{
public:
   Dis(){}
   R operator()(R const& x, R const& y)
   { return cpmdis(x,y);}
};

template<>
class Dis<Z>{
public:
   Dis(){}
   R operator()(Z const& x, Z const& y)
   { return x!=y ?  1. : 0.;}
};

template<>
class Dis<N>{
public:
   Dis(){}
   R operator()(N const& x, N const& y)
   { return x!=y ?  1. : 0.;}
};

template<>
class Dis<L>{
public:
   Dis(){}
   R operator()(L const& x, L const& y)
   { return x!=y ?  1. : 0.;}
};

template<>
class Dis<bool>{
public:
   Dis(){}
   R operator()(bool const& x, bool const& y)
   { return x!=y ?  1. : 0.;}
};

template<>
class Dis<string>{
public:
   Dis(){}
   R operator()(string const& x, string const& y)
   { return x!=y ?  1. : 0.;}
};

bool apprInt(R x, R tol);
   // returns true if the distance between x and the nearest
   // integer is <= tol (if tol<0, this is never the case)
```

```
//////////////////////// class Conv ////////////////////////////////////
template<class Ti, class Tf>
class Conv{
// conversion
// Allows to define conversion between basic types in a manner
// which differes from that provided by the compiler.
   Ti x_;
public:
   Conv(Ti const& t):x_(t){}
   Tf operator()(void){ return (Tf)x_;}
};

template<>
class Conv<R,L>{
   // The image classes Image<T> and MultiImage<T> have to convert
   // between T and R in some functions. By far the most important
   // case is T=R so that no real conversion is involved. However
   // also T = L occurs. This case is handled here.
   R x_;
public:
   Conv(R const& t):x_(t){}
   L operator()(void){ return (L)cpmrnd(x_);}
};

template<>
class Conv<L,R>{
   L x_;
public:
   Conv(L const& t):x_(t){}
   R operator()(void){ return (R)(Z)x_;}
};

////////////////// class-less function templates ////////////////////////
// This is a convenient form for using the service functions:
// The final T in the name is to suggest "template" and avoids name
// clashes with functions already defined for T = R with names inv, abs,
// hash, ran, ...
// See the comments on template <class T> class Root at the end of this
// file. Calling these functions is possible without mentioning the type
// of the argument, e.g. prnT(t,cout) which makes them usable in macros
// such as cpmp(X) and cpms(X).

// implementation of CPM_IO
template<class T>
bool prnT(T const& t, ostream& str)
   // print on
{ return CpmRoot::IO<T>().o(t,str);}

template<class T>
```

```
bool scanT(T& t, istream& str)
   // scan from
{ return CpmRoot::IO<T>().i(t,str);}

// implementation of CPM_ORDER
template<class T>
Z comT(T const& t1, T const& t2){ return Comp<T>()(t1,t2);}
   //: compare

// implementation of CPM_TEST_ALL
template<class T>
T invT(T const& t){ return Inv<T>()(t);}
   //: inverse

template<class T>
R absSqrT(T const& x){ return AbsSqr<T>()(x);}
   // absolute (value) squared

template<class T>
R absT(T const& x){ return Abs<T>()(x);}
   // absolute value

template<class T>
R disT(T const& t1, T const& t2){ return Dis<T>()(t1,t2);}
   // distance value

template<class T>
T testT(T const& t, Z complexity){ return Test<T>()(t,complexity);}
   // test value

template<class T>
T ranT(T const& t, Z j=0){ return Ran<T>()(t,j);}
   // random value

template<class T>
Z hashT(T const& t){ return Hash<T>()(t);}
   //: hash value

template<class T>
T conT(T const& t){ return Conj<T>()(t);}
   //: conjugate

template<class T>
T netT(T const& t, Z i){ return Neutrals<T>()(t,i);}
   // neutrals
} // CpmRoot

// A probably more convenient collection of these service functions
// is given by the class template CpmRoot::Root<> defined in cpmword.h.
// The class Root<T> has for each of the previous class-less functions
```

```
// a member function of the same functionality. For instance:
// T t1 = ... ;
// T t2 = ... ;
// R d = dis(t1,t2); ( = cpmdis(t1,t2), see the defines above )
// could also be written
// R d = Root<T>(t1).dis(t2);
// Here the Root-template needs the template argument T, which for the
// function template is optional (we could also write
// R d = disT<T>(t1,t2);).
// This explicit indication of the template argument helps more the reader
// than the compiler (who should know what to do anyway).
// Notice that definition of the I/O macro cpmp(X) and cpms(X) is
// easily based on functions prnOnT and scanFromT. As defined in
// cpminterfaces.h:
// #define cpmp(X) if (!prnOnT((X),str)) return false
// Probably also
// Root<typeid((X))>((X)).prnOn(str) would work. But I want to avoid
// the overhead incured by using typeid.

#endif
```

28 cpmnumbers.cpp

```
//? cpmnumbers.cpp
//? Status of work 2023-10-20.
//?
//?  ...


#include <cpmnumbers.h>
#include <cpmsystem.h>
#include <cpmsystemdependencies.h>
   // defines wrtPrc, wrtTit
#include <cpmmacros.h>
   // CPM_MA, CPM_MZ

using namespace CpmStd;
using CpmRoot::R;
using CpmRoot::Z;
using CpmRoot::N;
using CpmRoot::L;
using CpmRoot::Word;

N CpmRoot::operator""_N(unsigned long long int n){ return (N)n; }
Z CpmRoot::operator""_Z(unsigned long long int z){ return (Z)z; }
R CpmRoot::operator""_R(long double r){ return (R)r; }
R CpmRoot::operator""_R(unsigned long long int r){ return (R)r; }

#if defined(CPM_MP)
   Z CpmRoot::numPrc=CPM_MP;
#else
   Z CpmRoot::numPrc=0;
#endif

Z tenToTwo(Z p){ return (Z)(p*3.3219281 + 0.5);}
Z twoToTen(Z d){ return (Z)(d/3.3219281 + 0.5);}

void CpmRoot::prec(Z p)
{
;
}

Z CpmRoot::getPrec()
{
#if defined(CPM_MP)
   return CPM_MP;
#else
   return twoToTen(8*sizeof(R));
#endif
}
```

```
Z CpmRoot::getPrec(R const& r)
{
#if defined(CPM_MP)
   return CPM_MP;
#else
   return twoToTen(8*sizeof(R));
#endif
}

namespace{
// template tool for implementation
template <class T>
inline bool isValFunc(T x) // inline is needed
{
   if (cpmdbg>0){ // so calling this function can be switched
   // to idle from a central place.
      ostringstream ost;
      ost<<x;
      string s=ost.str();
      bool b1=s.find("I")==string::npos; // true if not found!
      bool b2=s.find("N")==string::npos;
      bool b3=s.find("n")==string::npos;
      //return b1||b2||b3; // corrected 2014-01-13
      return b1&&b2&&b3; // needs to give true for all options
   }
   else return true;
}

} // namespace

bool CpmRoot::isVal(L x){return true;}
bool CpmRoot::isVal(Z x){return isValFunc(x);}
bool CpmRoot::isVal(N x){return isValFunc(x);}

bool CpmRoot::isVal(R const& x)
{
#if defined(CPM_MP)
   if (cpmdbg>1){
      return !(isnan(x)||isinf(x));
   }
   else return true;
#else
   return isValFunc(x);
#endif
}

R CpmRoot::inv(R const& t)
{
   const Z maxMes=100;
```

```
   static Z mes=1;
   R z(0.);
   Word loc("inv(R): argument is 0");
   if (t==z){
      if (mes==maxMes){
         mes++;
         cpmmessage(loc&" ... messages discontinued");
      }
      if (mes<maxMes){
         mes++;
         cpmwarning(loc&", 0 returned");
      }
      return z;
   }
   R res=R(1.)/t;
   if (isVal(res)) return res;
   else {
      loc="inv(R): 1/argument is not valid";
      if (mes==maxMes){
         mes++;
         cpmmessage(loc&" ... messages discontinued");
      }
      if (mes<maxMes){
         mes++;
         cpmwarning(loc&", 0 returned");
      }
      return z;
   }
}

R CpmRoot::randomR(Z j)
{
   static N j0{137};
   static const R ranFac{1.e6};
   R x = j!=0 ? R(j) : R(++j0); // no overflow possible due to
      // using unsigned integers as N
   R y = cpmsin(x)*ranFac;
   return y -= cpmfloor(y);
}

R CpmRoot::ran(R const& r, Z j)
{
   R fac = randomR(j);
   return r*(fac*2.0 - 1.);
}

Z CpmRoot::hash(R const& r)
{
   const R reg=100000;
   R a=cpmabs(r);
```

```
   return cpmrnd(r+reg/(a*reg+1));
}

R CpmRoot::test(R const& r, Z complexity)
{ return R(1.123456789)*complexity;}

Z CpmRoot::toZ(R const& r, bool toZero)
{
   if (toZero){
      Z s = cpmsgn(r);
      R x = cpmfloor(cpmabs(r));
      Z xi;
#if defined(CPM_MP)
      xi=lround(x);
#else
      xi=(Z)x;
#endif
      return s*xi;
   }
   else{
#if defined(CPM_MP)
       return (Z)lround(cpmfloor(r));
#else
   return (Z)cpmfloor(r);
#endif
   }
}

R CpmRoot::dis(R const& r1, R const& r2)
{
   R a1 = cpmabs(r1);
   R a2 = cpmabs(r2);
   R a3 = cpmabs(r1-r2);
   return disDefFun(a1,a2,a3);
}

Z CpmRoot::randomZ(Z n, Z j)
{
   if (n<1) cpmerror("n<1 in CpmRoot::randomZ(Z n, Z j)");
   R y=n*randomR(j);
   Z res=toZ(y);
   return ((res==n) /* should not happen */ ? n : res+1);
}

bool CpmRoot::apprInt(R x, R tol)
{
   R d=x-rnd(x);
   return ( d>=0 ? d<=tol : -d<=tol);
}
```

```
namespace{
   L getbits(N x, N p, unsigned n)
   // Kernighan Ritchie: Programming in C, section 2.9
   { return (L)((x>>(p+1-n))& ~(~0<<n));}
}

L CpmRoot::getByte(Z z, Z n)
{
   N x=(N)z;
   return getbits(x, (n-1)*4,4);
}

#if defined(CPM_WRITE_PRECISION)
   Z CpmRoot::wrtPrc=CPM_WRITE_PRECISION;
#else
   Z CpmRoot::wrtPrc=10;
#endif

bool CpmRoot::writeTitle(string const& s , ostream& out)
{
   if (wrtTit) CpmRoot::write(string("//")+s,out);
   //return (out!=0);
   return !out ? false : true;
}

string CpmRoot::skipLeadingWhitespace(string const& str)
{
   Z n=(Z)str.size();
   if (n==0) return str;
   char cTest=str[0];
   if ( !isspace(cTest)) return str;
      // fast part of the process, no leading whitespace there
// now the first character is space
   Z i=0;
   while(i<n && isspace(str[i])) i++;
// eliminate the i leading characters
   string res=str;
   res.erase(0,i);
   return res;
}

string CpmRoot::skipTrailingWhitespace(string const& str)
   // made 2004-10-12 in analogy to the previous function
{
   Z n=(Z)str.size();
   if (n==0) return str;
   char cTest=str[n-1];
   if ( !isspace(cTest)) return str;
      // fast part of the process, no trailing whitespace there
// now the last character is space
```

```
   Z iTest=n-1;
   Z found=0;
   while(iTest>=0 && isspace(str[iTest])){
      iTest--;
      found++;
   }
   // now iTest is -1 or the index of a character
   // so iTest++ is the insex of the first of the trailing
   // whitespaces
   iTest++;
// eliminate the found trailing characters
   string res=str;
   res.erase(iTest,found);
   return res;
}

bool CpmRoot::startsComment(char c)
{ return c=='/' || c==';'|| c=='*'|| c=='#';}

bool CpmRoot::eatComments(istream& in)
{
   Z mL=3;
   Word loc("CpmRoot::eatComments(istream&)");
   CPM_MA
   char c=0;
   string buff;
   if (!in){
      cpmmessage(mL,loc&": bad stream before reading");
      CPM_MZ
      return false;
   }
LOOP:
   if (!(in>>c)){
      ostringstream ost;
      ost<<loc<<":  bad stream after reading character c="<<(Z)c;
      cpmmessage(mL,Word(ost.str()));
      CPM_MZ
      return false;
   }
   if (startsComment(c)){
      std::getline(in,buff); // reads the line till end of line
         // including the '\n' into the self-adjusting buff
      goto LOOP;
   }  // now c is no longer the potential beginning of a comment
   in.putback(c);
   if (!in){
      cpmmessage(mL,loc&": bad stream after reading");
      CPM_MZ
      return false;
   }
```

```
   CPM_MZ
   return true;
}

bool CpmRoot::strVal(istream& str)
{ if (!str||str.bad()) return false; else return str.good(); }

bool CpmRoot::strVal(ostream& str)
{ if (!str||str.bad()) return false; else return str.good(); }

bool CpmRoot::readLine(string& line , istream& in)
{
   if  (!eatComments(in)) return false;
   std::getline(in,line);
   line=skipTrailingWhitespace(line);
   return in ? true : false;
}

template <>
bool CpmRoot::read(L& x, istream& in)
{
   Z z;
   bool suc=CpmRoot::read<Z>(z,in);
   x=(L)z;
   return suc;
}

template <>
bool CpmRoot::write(L const& x, ostream& out)
{
   out<<(Z)x<<endl;
   return strVal(out);
}

template <>
bool CpmRoot::write(R const& r, ostream& out)
// experimental avoidance of writing too small numbers
// which caused trouble under Windows
{
   const R wrTiny=1e-304;
      // 1e-305 works, 1e-310 works not
   out.precision(wrtPrc);
   R x=r;
   if (x>0){
      if (x<wrTiny) x=wrTiny;
   }
   else if (x<0){
      if (x>-wrTiny) x=-wrTiny;
   }
   else{
```

```
      x=0;
   }
   out<<x<<endl; return strVal(out);
}
```

29 cpmp.h

```
//? cpmp.h
//? Status of work 2023-10-20.
//?
//?  ...


#ifndef CPM_P_H_
#define CPM_P_H_
/*

Purpose: defining classes of smart pointers, and polymorphic arrays.
classes Pp<>, Po<>, Vp<>.
class P<> is defined in cpmuc.h

Notice usage of the new macro definition (2017-05-23)
 #define const& const&
 from cpmbasictypes.h

*/

#include <cpmv.h>

namespace CpmArrays{

////////////////////////  class Pp< >  ///////////////////////////////////
//
// class of non-constant smart pointers implementing 'copy on write' (see
// cpmv.h) and polymorphism (indicated by the 'p'). Let us first discuss
// the 'mono-morphic' aspects, i.e. we discuss the properties of Pp<T>
// for arbitrary but fixed T and not for a variety of T's:
//
// Let T be any non-abstact class that defines a member function
//
//       Tb* T::clone(void)const{ return new T(*this);}  (ii)
// (Notice that T has to define a copy constructor for this to work)
//
// where Tb is either T itself or a base class of T which declares
//
//       virtual *Tb Tb::clone(void)const{ return new Tb(*this);}
// or
//       virtual *Tb Tb::clone(void)const=0;
//
// Then, let us say that T is cloneable, with clone base Tb. Then the
// main property of the Pb<> template is:
//
//    cloneable T ==> value class Pb<T>
//
```

```
// If the source code of a non-abstract T is open to modifications, it
// can be made cloneable by inserting the definition
//    T* T::clone()const{ return new T(*this);}
// (lets ignore the complication that T may already use the name
//  clone in a conflicting manner).
//
// Properties of Pp<T> as a 'polymorphic container':
// -------------------------------------------------
//
// We consider a finite set L of cloneable classes which all have the
// same clone base T0. Let T1 be one of those and let L(T1) denote the
// classes in L which are derived from T1. Then Pp<T1> is a polymorphic
// container for L(T1) in the following sense:
// Let pp be an instance of Pp<T1>, created e.g. as
//    Pp<T1> pp;
// and let T2 be a class belonging to L(T1), and let t2 be an instance of
// T2.
//
// Then t2 can be put into the container pp by the
// statement
//
//       pp=t2; (or also pp.set(t2);) (**)
//
// Now pp carries just the information that was carried before (and is
// carried still) by t2. Warning: using to the right of = a ternary
// expression (... ? ... : ...) may destroy polymorphism. This is no real
// problem since the ternary can always replaced by if (...) ... else ...
// .
// If t2 gets changed or destroyed (e.g. as a local variable going out of
// scope) pp, will conserve the information and can return it in two
// ways.
// 1. Reconstruction as object: T2 t2_=pp(T2()) will create an object t2_
//    which is equivalent to t2.
// 2. Reconstruction by action: Let
//
//       virtual void T1::show(Media&)const;
//
// be a function which creates a detailed textual or graphical
// description of the calling instance by interacting with a class Media
// that provides the basis for that. Then 'Media m; pp().show(m);' will
// leave m in a state that carries the same record that we would get from
// 'Media m; t2.show(m)'. Technically, pp() is a quantity that is typed
// T1&; the object it refers to is of type T2 however.
// Let T3 be another class derived from T1, then pp, after pp.set(t3)
// behaves as T3 object. Thus pp behaves changing its type depending on
// initialization.
// Pp<> is closely related to the surrogate classes as discussed in
// Andrew Koenig: Ruminations on C++, AT&T 1997 Chapter 5. As stated
// there, the main benefit of such a 'polymorphic class' is that its
// instances can be put into a strictly typed container. This will be
```

```
// carried out in defining the extremely useful class Vp<>.
//
// Note that that in line (**) a constuction
// Vp<T1> pp=t2
// for a not yet existing pp does not work. The safe mode of operation is:
// 1. create the container; 2. fill the container.

template <class T>

class  Pp: public P<T>{ // polymorphic smart pointers
   typedef P<T> Base;
public:
    Pp(T* p=0):P<T>(p){}
      // carrying over the constructor from P. Notice that the
      // primary act of construction was to initialize p_. This can be by
      // T* p_=new Td(....) where Td is some derived class (or T itself)

    explicit Pp(T const& t):P<T>(static_cast<T*>(t.clone())){}
    // 2004-09-27: finally a pointer-less interface of Pp<> enabled!
    // The queer casting is needed in the general situation that
    // T0 and T1 are different (in the description above). This general
    // situation is probably not really important (I had it
    // unintentionally for T0=RigidObject and T1=ExtSys, but
    // simplified it to T0=T1=ExtSys without encountering problems)
    // Notice, however, that t.clone() involves call to the copy
    // constructor of T which might be expensive.

    Pp(const P<T>& x):P<T>(x){}
      // down-cast constructor

   T& operator()(void){ return *Base::p_;}
      // Pp<T> pp;
      // T t;
      // pp=t;
      // makes t.aVirtualFunction(...);
      // and
      // pp().aVirtualFunction(...);
      // calling the same function with the same data.

   T const& operator()(void)const{ return *Base::p_;}
      // not clear why the MS-compiler cannot infer this definition from
      // the base class.
      // Pp<T> pp;
      // T t=pp()
      // is the usual way to retreive the value of pp. If the value is
      // known to belong to a derived class Td, use the next function:
      //    Td t=pp(Td()).
      // If we don't want to retreive t but only want to call a
      // (potentially virtual) member function, one always proceeds as in
      // the following example:
```

```
      // Word w = pp().nameOf(); // see also comment to previous function

   template <class X>
      X operator()(X const& x)const
      // retreiving an element from the container as a copy of a
      // prescribed type.
      // The value of x has no influence, only the type matters.
      // T has to be a clone base of X.
      { X* px=static_cast<X*>(Base::p_->clone()); return X(*px);}

   template <class X>
   bool get(X& x)const
      // retreiving the element from the container as a reference
      // to the prescribed type. The value of x at input has no
      // influence, only the type matters. T has to be a clone base of X.
   { X* px=static_cast<X*>(Base::p_->clone()); x=X(*px); return true;}

   void set(T const& x){ cow(); Base::p_=static_cast<T*>(x.clone());}
      // loading an object of type T or of a derived type into Pp<T>
      // no longer needed since assignment works now

   Pp<T>& operator=(T const& x){ cow(); return *this=Pp<T>(x);}
      // 2004-09-27 together with a redefinition of
      // Vp<>::operator[]() this allows the intialization
      // of components of polymorphic arrays by assignment
      // so that the less idiomatic set(i,T $)-functions
      // should no longer be needed. The problems with the old
      // attempts to set values of components via assignment are
      // now understood (and eliminated)
   Word nameOf()const
   {
      return Word("Pp< "&CpmRoot::Name<T>()(T())&" >");
   }

protected:
   void cow(void);
      // 'copy on write'. The first '*this-changing' statement in the
      // body of a non-constant member function has to be cow();
};

template <class T>
void  Pp<T>::cow(void)
{
   if (Base::u_.makeOnly()){
      if (Base::p_!=0) Base::p_=static_cast<T*>(Base::p_->clone());
         // let Tb be the class closest to the root of the class
         // hierarchy where a virtual ... clone(void)const is defined,
         // then present clone is Tb* T::clone()const. Notice that
         // modern compilers sometimes accept T* T::clone()const.
         // Microsoft Visual C++ 5.0 is not of this kind,
```

```
         // so in defining clone() in a class, we have always to
         // remember the base. This is only a minor inconvenience.
      Base::u_.startNew_();
   }
}

//////////////////////////// class Po< > ////////////////////////////////

template <class T>
class  Po: public  Pp<T>{  // Adding order operations to Pp<T>
     // Allows to form, for instance, std::vector<Po<T> >
     // and thus allows to combine STL functionality with polymorphism
     // and banning of pointers
   typedef Pp<T> Base;
public:
   Po(T* p=0): Pp<T>(p){}
   Po(const Pp<T>& x): Pp<T>(x){}
   bool operator  <(const  Po& x)const{ return *Base::p_ <  *(x.p_);}
   bool operator ==(const  Po& x)const{ return *Base::p_ == *(x.p_);}
   bool operator  >(const  Po& x)const{ return *Base::p_ >  *(x.p_);}
   bool operator !=(const  Po& x)const{ return *Base::p_ != *(x.p_);}
   bool operator <=(const  Po& x)const{ return *Base::p_ <= *(x.p_);}
   bool operator >=(const  Po& x)const{ return *Base::p_ >= *(x.p_);}
};

////////////////////////////  class Vp< >  ////////////////////////////////
// This template behaves just as Pp<T> except that access is indexed
// which allows to store a number of instances and to retreive them:
// Let T,T1,T2,T3 as in the explanation of Pp<>. Then a typical usage is
//
// T1 t1(...);
// T2 t2(...);
//
// V<T3> t3(2);
// t3[1]=T3(...);
// t3[2]=T3(...);
//
// Vp<T1> vp(4);
// vp[1]=t1;
// vp[2]=t2;
// vp[3]=t3[1];
// vp[4]=t3[2];

// A detailed description (see Pp<>) of the the vp-components can be
// created like
// that:
//
// Z nItems=4;
// V<Media> m(nItems);
// (for Z i=1;i<=nItems;i++) vp(i).show(m[i]);
```

```
//
// Thus, despite their differing types, the t1, t2, t3[1], t3[2] get
// processed in a single loop; this capability is indispensible for
// polymorphism to be useful in non-trivial situations. The most
// expressive application is to use Vp<T1>-typed data members in classes:
//
// class ParticleMix{ // assume that Particle is clone base for both
//    // ParticleA and ParticleB
//    Vp<Particle> p;
// public:
//    ParticleMix(Z nA, Z nB):p(nA+nB)
//       // creates a mix of nA particles of type A and
//       // and nB particles of type B
//    {
//       Z i;
//       V<ParticleA> pA(nA); for (i=1;i<=nA;i++) p[i]=pA[i];
//       V<ParticleB> pB(nB); for (i=1;i<=nB;i++) p[i+nA]=pB[i];
//    }
//    void moveIndependently(R timeStep)
//    {
//       for (Z i=1;i<=p.dim();i++) p(i).moveIndependently(timeStep);
//             // applies the law of motion for A-particles if p(i) is
//             // a A-particle, and for B-particles correspondingly
//    }
// };
//
// Whereas in a 'stand alone'-control structure like the 'Media-loop'
// given above it would not be much more complicated to use a T1 pointer
// for implementing the loop, in the class example we gain more: Since
// Vp<Particle> is a value class, the compiler will properly and
// automatically define copy constructor, assignment operator, and
// destructor for us, whereas using pointers would leave this burden on
// us. Programmers who did this pointer implementation several times
// successfully might begin to love the procedure. I consider it wasted
// time and energy, and a lost opportunity to deal with complexity in a
// state-of-the-art manner.

template <class T>
class  Vp{ // polymorphic V template

   CpmArrays::V< Pp<T> > rep;

   explicit Vp(V< Pp<T> > const& vp):rep(vp){}

public:

    Vp(){}
      // notice that  both Pp<T> and V define a default constructor

   explicit Vp(Z n):rep(CpmArrays::V<Pp<T> >(n,Pp<T>())){}
```

```
     // No pre-setting of values occurs. If needed, this has to be
     // done by function void set(T const& x).

//  Vp(Z n, Pp<T> const& p):rep(CpmArrays::V<Pp<T> >(n,p)){}
     // 'Making a series of copies of a single polymorphic object'.

//  Pp<T> at(Z i)const{ return rep(i);}

  Vp<T>& operator=(Vp<T> const&)=default;



  Pp<T>& operator[](Z i){ return rep[i];}
     // We use [] for access to the true components, which
     // are instances of the true value class Pp<T>.
     // Typical usage:
     // Vp<T> vp(2); T x=..., y=...; vp[1]=x; vp[2]=y;
  Pp<T> const& operator[](Z i)const{ return rep[i];}

  T& operator()(Z i){ return rep[i]();}
  T const& operator()(Z i)const{ return rep[i]();}
     // We use () for access to the T& that will be in
     // most cases the objects needed for further
     // processes, as for usage as function arguments.
     // Typical usage:
     // Vp<T> vp=...; Word w1=vp(1).nameOf(), w2=vp(2).nameOf();

   //  void operator<<(Pp<T> const& p){rep<<p;}

  template <class X>
  X operator()(Z i, const X& x)const
     // retreiving the i-th component from the container as a copy of a
     // prescribed type. The value of x has no influence, only the type
     // matters. T has to be a clone base of X.
     { X* px=static_cast<X*>(rep[i]().clone()); return X(*px);}

  template <class X>
  bool get(Z i, X& x)const
     // retreiving the i-th component from the container as a referemnce
     // to the prescribed type. The value of x at input has no
     // influence, only the type matters. T has to be a clone base of X.
  {
     if (i<1 || i>rep.dim()) return false;
     X* px=static_cast<X*>(rep[i]().clone()); x=X(*px);
     return true;
  }

  void set(Z i, const T& x){ rep[i]=x;}
     // no longer essential since the defining statement
     // is natural enough not to need a wrapper
```

```
   void set(T const& x){ for (Z i=1;i<=rep.dim();i++) rep[i]=x;}

   Z dim()const{ return rep.dim();}
   Z b()const{ return rep.b();}
   Z e()const{ return rep.e();}
   bool valInd(Z i)const{ return rep.valInd(i);}

   Vp<T> rev()const
      //: reversed
   {
      return Vp<T>(rep.rev());
   }

   Word nameOf()const{
      Word nt=CpmRoot::Name<T>()(T());
      Word wi="Vp<";
      return wi&nt&">";
   }
};

} // namespace

#endif
```

30 cpms.h

```
//? cpms.h
//? Status of work 2023-10-20.
//?
//?  ...


#ifndef CPM_S_H_
#define CPM_S_H_
/*

   Description: Defines a template class S<T> of sets
       the elements of which are of type T.
      Intented as a basis for a mathematics style definition of
           associative lists.

*/

#include <cpmv.h>
#include <cpmtypes.h>
#include <set>

//////////////////////////// class S<> ///////////////////////////////////

namespace CpmArrays{

   using CpmFunctions::F;
   using CpmRoot::Word;

template <class T>
class S{ // sets of homotypic elements
   // Represents sets, all the elements of which are of type T.
   // It is assumed that T<T and T>T is defined.

public:

   std::set<T> p_;

   typedef S<T> Type;

//public:

   CPM_IO_V
      // CPM_ORDER comes later in this class declaration/definition
   CPM_ORDER

// constructors
```

```
   S():p_(){}
      // constructor for the void set

   S(std::set<T> const& p):p_{p}{}

   void merge(V<Type> const& setList){
      for (Z i=setList.b();i<=setList.e();++i) *this << setList[i];
   }
      // changes *this into the union of *this and the sets in the list

   explicit S(V<T> const& vec){
      S<T> s;
      for (Z i=vec.b();i<=vec.e();++i) s << vec[i];
      *this = s;
   }

      // constructor from an array of T-values (V<T> sufficient, not
      // needing Vo<T>; T, however, has to support ordering (which is
      // clear if S<T> is under consideration). Clearly, multiple
      // appearing of components in vec does not prevent their
      // single appearence in the result (which, otherwise, would not be
      // a set)

    explicit S(std::initializer_list<T> il ){ *this = S<T>(V<T>(il));}

// cardinality access

   Z car()const{ return p_.size();}
      //: cardinality
   Z dim()const{ return p_.size();} // for uniformity with other array types
      //: dimension
   Z size()const{ return p_.size();} // for uniformity with STL code
      //: size

 // means for forming loops over sets without using indexing

   auto begin(){ return p_.begin();}
      // allows to write code such as
      // S<Z> s{-1,1,-2,2,-3,3};
      // for (auto const& x : s) cout<<x<<endl;
      // for 'looping over the range of s'.
      // also possible but less elegant:
      // for (auto q=s.begin(); q!=s.end();++q){ cout<<*q<<endl;}

   auto end(){ return p_.end();}
   auto cbegin(){ return p_.cbegin();}
   auto cend(){ return p_.cend();}

// set formation operations
   bool addAct_(T const& t){
```

```
   auto q=p_.insert(t);
   return q.second;
}
   // changes *this into the union of (*this) and {t}
   // and returns true if t was not already there
   // so that it had to be added 'actually'

void operator<<(T const& t){
   p_.insert(t);
}

void add_(T const& t){ p_.insert(t);}
   // changes *this into the union of (*this) and {t}
   // return value of addAct not used

void add_(Type const& s){ for (auto const& x : s.p_){ p_.insert(x);} }
   // changes *this into the union of (*this) and s

void operator<<(Type const& s){
   for (auto const& x : s.p_){ p_.insert(x);}
}
// same as add_(Type const& s)

bool rem_(T const&  t){
   auto q=p_.find(t);
   if (q!=p_.end()){
      p_.erase(q);
      return true;
   }
   else return false;
}
   //: remove
   // *this will be changed into *this\{t}
   // If this reduces the cardinality of *this, the return
   // value is 'true' and 'false' else

bool remove(T const&  t){ return rem_(t);} // heritage

Type operator |(Type const&  s)const
   // forming the union of *this and s (corresponds to the
   // 'or' operation for indicator functions)
{
   S<T> res(p_);
   for (auto const& x : s.p_){ res<<x;}
   return res;
}

Type operator &(Type const&  s)const
   // forming the section of *this and s (corresponds to the
   // 'and' operation for indicator functions)
```

```
{
   S<T> res;
   for (auto const& x : s.p_){
      if (hasElm(x)){ res<<x;}
   }
   return res;
}

Type operator -(Type const&  s)const
   // forming *this\s i.e. the set of all those elements of
   // *this which don't belong to s (relative complement)
{
   S<T> res(p_);
   for (auto const& x : s.p_){
      if (hasElm(x)){ res.rem_(x);}
   }
   return res;
}

Type operator ^(Type const&  s)const{ return (*this-s)|(s-*this);}
   // forming the symmetric difference of s1 and s2
   // (corresponds to 'exclusive or' for indicator functions)

T  operator[](Z i)const
{
   Z count=0;
   for(auto x: p_){
      ++count;
      if (count==i) return x;
   }
   return T();
}
   // Means of accessing an elements of the set by index.
   // The specific indexing is gouverned by the < order.
   // As a means of looping over all elements of larger sets it is very
   // inefficient. Use the technique described in a comment to
   //    auto begin()
   // for efficient looping.


T const&  fir()const{ return *(p_.begin());}
   //: first
   // first element of *this according to the ordering on
   // which the construction of Type is based.
   // causes error if *this is void

T const&  last()const{ return *(p_.end()--);}
   // last element of *this according to the ordering on
   // which the construction of Type is based.
   // causes error if *this is void
```

```
// boolean operations

   bool isVoid(void)const{ return p_.empty();}

   bool isEmpty(void)const{ return p_.empty();}

   bool hasElm(T const& t)const{
      return p_.contains(t);
   }
      //: has element
      // returns true iff t is an element of *this

   bool ni(T const&  t)const{ return hasElm(t);}
      // 'in' reversed, from LATEX symbol \ni for the mirror image of
      // the epsilon-like 'is element'-symbol
      // Notice that the 'in' function would have as first argument a T
      // and as second argument a Type. This cannot be made a member
      // function. Introducing a non-member friend function would offend
      // the C+- coding style.

   void select(V<B> const& sel){
      V<T> temp=toV();
      temp=temp.select(sel);
      *this=S<T>(temp);
   }
      // eliminates from *this all components x[i] for which sel[i]
      // is defined and statisfies sel[i]==false.
      // (see corresonding (non-mutating) function of V<T>)

// relations to other classes:

   V<T> toV(void)const{
      std::vector<T> p;
      for (auto const& x : p_) p.push_back(x);
    //  std::sort(p.begin(),p.end());
    // tests showed that res comes out as increasingly ordered
    // In BSC++20: p. 170 Advice [24]: use ordered containers (e.g. map and set)
    // if you need to iterate over their elements in order
      V<T> res(1,p);
      return res;
    }
      // getting the representing vector as a new entity. It may be
      // useful to manipulate this vector by, for instance,
      // the V<T>::condense() function and then form a new set via the
      // constructor from vectors

// Infrastructure functions as declaration macros  known from including
// <cpmvo.h>
```

```
      // since T has to allow ordering, we implement the order
      // operations. This allows to form S< Type >.

   virtual S<T>* clone()const{ return new S(*this);}

   Word toWord()const;

   virtual Word nameOf()const{
      Word wi="S< ";
      Word wt=CpmRoot::Name<T>()(T());
      return wi&wt&" >";
   }
};

template <class T>
Z S<T>::com(Type const& s)const{ return toV().com(s.toV());}

template <class T>
bool S<T>::prnOn(ostream& str)const
{
   Z mL=3;
   static Word loc("S<>::prnOn(ostream&)");
   CPM_MA
   cpmwbt;
   bool b;
   for (auto const& x : p_){
      b=CpmRoot::IO<T>().o(x,str);
      cpmassert(b==true,loc);
      if (b==false){
         cpmdebug(x);
         CPM_MZ
         return false;
      }
   }
   cpmwet;
   CPM_MZ
   return true;
}

template <class T>
bool S<T>::scanFrom(istream& str)
{
   Z mL=3;
   static Word loc("S<>::scanFrom(istream&)");
   CPM_MA
   S<T> res;
   T t;
   while (CpmRoot::IO<T>().i(t,str)){ res.add_(t);}
   *this=res;
   CPM_MZ
```

```
   return true;
}

} // CpmArrays

#endif
```

31 cpmsr.h

```
//? cpmsr.h
//? Status of work 2023-10-20.
//?
//?  ...


#ifndef CPM_SR_H_
#define CPM_SR_H_
/*
   Description: Defines a class template Sr<T> of sets
       the elements of which are of type T.

   History: Started 95-3-13
      95-3-15 transformed in a class derived from Obj
      97-9-5 transformed as to fit the CTL framework
      99-7-27 transformed as to fit the CPM framework
      02-5-24 implementation changed to STL, Test_set carried out
      2005-09-10 implementation purged from io_clone and r_clone
*/

#include <cpms.h>
#include <cpmvr.h>

namespace CpmArrays{

   using CpmRoot::Word;

template <class T>
class Sr: public S<T>{  // S with a rich interface
   // represents sets, all the elements of which are of type T.
   typedef Sr<T> Type;
   typedef S<T> Base;
public:
// constructors.

   Sr(void):S<T>(){}
         // constructor for the void set (n=0 added 95-11-24)

   Sr(const T& t):S<T>(t){}
         // constructor for the set {t}

   Sr(const S<T>& s):S<T>(s){}
      // down-cast constructor

   Sr<T> condense(void)const;
      // forms a new set obtained by shrinking to a single representative
      // element every equivalence class with respect to function equiv,
```

```
      // which itself is controlled by the static element acc

// Infrastructure functions

   virtual S<T>* clone()const{ return new Sr(*this);}

   CPM_IO
      // prnOn, scanFrom
   CPM_TEST_X
      // ran, test, hash, dis, abs, absSqr
   CPM_DESCRIPTORS
      // toWord, nameOf
      // The missing 3 from Root are: net, com, con

   static R acc;

protected:

   static bool equiv(const T& t1, const T& t2)
   { return CpmRoot::disT<T>(t1,t2)<acc;}

};

// test functions based on set theoretical identities

 template <class T>
   void idem1(Sr<T>& lhs, Sr<T>& rhs, const Sr<T>&s)
      { lhs=(s&s); rhs=s;}

   template <class T>
   void idem2(Sr<T>& lhs, Sr<T>& rhs, const Sr<T>& s)
      { lhs=(s|s); rhs=s;}

   template <class T>
   void comm1(Sr<T>& lhs, Sr<T>& rhs, const Sr<T>& s1, const Sr<T>& s2)
      { lhs= (s1&s2); rhs=(s2&s1);}

   template <class T>
   void comm2(Sr<T>& lhs, Sr<T>& rhs, const Sr<T>& s1, const Sr<T>& s2)
      {   lhs= (s1|s2); rhs=(s2|s1);}

   template <class T>
   void assoc1(Sr<T>& lhs, Sr<T>& rhs, const Sr<T>& s1, const Sr<T>& s2,
    const Sr<T>& s3)
      { lhs= ((s1&s2)&s3); rhs=(s1&(s2&s3));}

   template <class T>
   void assoc2(Sr<T>& lhs, Sr<T>& rhs, const Sr<T>& s1, const Sr<T>& s2,
    const Sr<T>& s3)
      { lhs= ((s1|s2)|s3); rhs=(s1|(s2|s3));}
```

```
   template <class T>
   void distr1(Sr<T>& lhs, Sr<T>& rhs, const Sr<T>& s1, const Sr<T>& s2,
    const Sr<T>& s3)
      { lhs= (s1|(s2&s3)); rhs=((s1|s2)&(s1|s3)) ;}

   template <class T>
   void distr2(Sr<T>& lhs, Sr<T>& rhs, const Sr<T>& s1, const Sr<T>& s2,
    const Sr<T>& s3)
      { lhs= (s1&(s2|s3)); rhs=((s1&s2)|(s1&s3)) ;}

   template <class T>
   void diffdistr1(Sr<T>& lhs, Sr<T>& rhs, const Sr<T>& s1, const Sr<T>&
    s2, const Sr<T>& s3)
      { lhs= (s1-(s2|s3)); rhs=((s1-s2)&(s1-s3)) ;}

   template <class T>
   void diffdistr2(Sr<T>& lhs, Sr<T>& rhs, const Sr<T>& s1, const Sr<T>&
    s2, const Sr<T>& s3)
      { lhs= (s1-(s2&s3)); rhs=((s1-s2)|(s1-s3)) ;}

// more technical tests

   template <class T>
   int addElements(const Sr<T>& s, const T& t);
      // adding and finding an element

   template <class T>
   R writeRead(const Sr<T>& s);
      // read write test (to and from file)

// Implementation of Sr<T>

template <class T>
R Sr<T>::acc=1e-3;

template <class T>
Sr<T> Sr<T>::condense(void)const
{
   Vo<T> xr=Base::toV();
   Vo<T> x1=(static_cast< V<T> >(xr).condense(equiv)).first();
   return S<T>(x1);
}

template <class T>
Word Sr<T>::nameOf(void)const
{
   Word a="Sr<";
   Word b=CpmRoot::Name<T>()(T());
   return a&b&">";
```

```
}

template <class T>
R Sr<T>::absSqr(void)const
//
{
   R s{0.};
   for (auto const& x : Base::p_){
      s+=CpmRoot::AbsSqr<T>()(x);
   }
   return s;
}

template <class T>
R Sr<T>::abs(void)const
//
{
   return cpmsqrt(absSqr());
}

/*
// looks natural but behaves not well in comparing two sets where the
// second is a noisy copy of the first (e.g. ofter writing to file with
// limited numerical accuracy and reading back. In this case the next
// version is superior. This illustrates the point that the notion of sets
// depends strongly on the notation of equality (and vice versa).

template <class T>
R Sr<T>::dis(const Sr<T>& s)const
{
   Sr<T> sa=(*this).condense();
   Sr<T> sb=s.condense();
   Sr<T> sc=sa^sb;

   R s1=sa.abs();
   R s2=sb.abs();
   R s3=sc.abs();
   return CpmRootX::distFunc(s1,s2,s3);
}
*/

template <class T>
R Sr<T>::dis(Sr<T> const& s)const
{
   if ( Base::car() != s.car()) return 1;
   if ((*this^s).car()>0) return 1; else return 0;
}

template <class T>
Sr<T> Sr<T>::ran(Z j)const
```

```
{
   Vr<T> temp=Base::toV();
   auto v=temp.ran(j);
   return S(v);
}

template <class T>
Sr<T> Sr<T>::test(Z tsz)const
{
   Z nt=CpmRoot::Root<Z>(Z{0}).test(tsz);
   T tTest;
   tTest=CpmRoot::Root<T>(tTest).test(tsz);
   V<T> xr(nt);
   for (Z i=1;i<=nt;i++){
      xr[i]=CpmRoot::Root<T>(tTest).ran(i);
   }
   S<T> res(xr);
   return res;
}

template <class T>
Z Sr<T>::hash(void)const
{
   Z h{0};
   h+=Base::dim();
   for (auto const& x : Base::rep()){
      h+=CpmRoot::Root<T>(x).hash();
   }
   return h;
}

template <class T>
bool Sr<T>::prnOn(ostream& str)const
{
   return Base::prnOn(str);
}

template <class T>
bool Sr<T>::scanFrom(istream& str)
{
   return Base::scanFrom(str);
}

template <class T>
Word Sr<T>::toWord()const
{
   Word res="{ ";
   for (auto const& x : Base::p_){
      Word wi=CpmRoot::toWord<T>(x);
      wi&=", ";
```

```
      res&=wi;
   }
   res&=CpmRoot::toWord<Z>(Base::car());
   res&=" }";
   return res;
}

template <class T>
int addElements(const Sr<T>& s, const T& t)
   // Tests whether an element added to a set is retrieved as an element
   {
      Sr<T> s1=s;
      s1.add(t);
      return s1.hasElem(t);
   }

template <class T>
R writeRead(const Sr<T>& s)
   // Tests whether a set can be read again when written to file
   {
      Word filename=s.nameOf()+".dat";
      Word name=s.nameOf();
      ofstream out(-filename);

      if (!out) cpmerror(name,"test: output file cannot be opened");
      out<<s;
      out.close();
      Sr<T> sr;
      R res;
      ifstream in(-filename);
      if (!in) cpmerror(name, "test: input file cannot be opened");
      in>>sr;
      in.close();
      res=s.dis(sr);
      return res;
    }
} // CpmArrays

#endif
```

32 cpmsystem.h

```
//? cpmsystem.h
//? Status of work 2023-10-20.
//?
//?  ...


#ifndef CPM_SYSTEM_H
#define CPM_SYSTEM_H
/*
   Purpose: Declaring basic facilities for a C+- based program to
   communicate with the user. These facilities are available whenever
   the file 'cpmsystem.cpp' is among the source file set of the program
   and the programs include search path finds 'cpm0/include'.

*/

#include <cpmword.h>
#include <filesystem> // needs C++20

namespace CpmRoot{
/////////////////////////// class B ////////////////////////////////////////
// a wrapper for type bool
// We need in class Message some quantities of a bool-like type and need
// this type T also in the form V<T> and hence also in the form
// std::vector<T>. Due to the wellknown irregularites of std::vector<bool>
// we use a wrapper class for bool as that T.

class B{ // boolean values as a class, for which V<B> behaves regularly
   // unlike V<bool>
   bool x1;
public:
   typedef B Type;
// constructors
   B():x1(false){}
      // default is false !!!
      // corresponds to default 0 in Z1,R1,...
   explicit B(bool a):x1(a){} // no automatic conversion wanted
  // explicit B(Z a):x1(a!=0){} // no automatic conversion wanted
   operator bool()const{ return x1;}

   void setF_(){ x1=false;}

   void setT_(){ x1=true;}

   B& operator=(bool a){ x1=a; return *this;}
  // B& operator=(Z a){ x1=(a!=0); return *this;}
```

```
   CPM_IO
   CPM_ORDER
   Word nameOf()const{ return "B";}
   Word toWord()const{ return (x1 ? "true" : "false");}
   B operator!()const{ return B(!x1);}
   B operator&(const B& b)const{ return B(x1&&b.x1);}
   B operator|(const B& b)const{ return B(x1||b.x1);}
// escaping to Latin since 'not', 'and', and 'or' now
// are restricted in usage very much like C++ key words
   B non()const{ return B(!x1);}
   B et(const B& b)const{ return B(x1&&b.x1);}
   B vel(const B& b)const{ return B(x1||b.x1);}
   B ran(Z j=0){ return B(CpmRoot::ranT(x1,j));}
};

extern B wrtTit;
      //: write title
      // if this is true, all but the elementary tyes get written on
      // stream with a title preceeded by a comment indicator such as
      // '//'

   B operator""_B(unsigned long long int);
}

namespace CpmSystem{ // C+- system basic facilities

   using namespace CpmStd;

   using CpmRoot::Word;
   using CpmRoot::Z;
   using CpmRoot::operator""_Z;
   using CpmRoot::R;
   using CpmRoot::operator""_R;
   using CpmRoot::B;

   class Message;
   void updFromConfigFile();
   class Error;
   void error(Word const& w);
   void error(std::ostringstream const& ost);
   void error(Word const& w1, Word const& w2);

////////////////////////// class OFileStream ////////////////////////////////

const bool binDef=true;
//const bool binDef=false;
   // true is proven, false for experimentation
   // default value for setting the argument 'binary'
   // for opening streams. Introduced 2004-04-29
```

```
class OFileStream{ // output file stream

   ofstream* fs_;
   const Word name_;
   typedef OFileStream Type;
   CPM_INVAR(OFileStream)
public:

   explicit OFileStream(Word const& fileName,
      bool safe=true, bool binary=binDef);

   OFileStream():fs_(0){}

  // virtual ~OFileStream(){ fs_->close();delete fs_;}
   virtual ~OFileStream()=default;

    bool isVal()const
       //: is valid
       // returns stream status
    { if (!*fs_||fs_->bad()) return false; else return fs_->good(); }

   Word getName()const{ return name_;}
      //: get name

   ofstream& operator()(void){return *fs_;}
      // returns the object for direct access,
      // this is all one needs to write to the object
};


////////////////////////// function ppm2mp4 ////////////////////////////////////

  void ppm2mp4(Word const& movieName, Z frameRate=8_Z);
      // Makes a mp4-video from a series of ppm-image files.
      // The image files are assumed to be placed in the same directory as
      // the executable. If the ppm images are created by the function
      // CmpGraphics::Frame::vis(bool,bool, bool) they will be created if the
      // first of the boolean arguments is true, and they will end up in the
      // right place (for being processed by ppm2mp4) if the last argument is
      // true. The video file created by function ppm2mp4(Word,Z) will finally
      // moved into the requested output directory Message::outDir2.
      // The ppm-files from which the video-file was created will not be moved;
      // they will be deleted since otherwise they would too easily overload
      // the hard disk.

  void ppm2jpg(Word const& imgFileName);
      // We assume that in the directory from which the function ppm2jpg is
      // called there is a ppm-image file with name given by the argument of the
      // function (this name thus has to end in .ppm).
      // As a result of calling ppm2jpg in the same directory there will be
```

```
      // an jpg-file with a name as the input but with .ppm replaced by .jpg.
      //
      // If one creates a simulation video there it may sometimes look
      // advantageous to repeat one image several times to get the impression
      // of a still image that invites inspection. When writing a comment to
      // the video as a html-file one would like to refer to these faked still
      // images by a href to an actual still image. Creation of the pseudo-still
      // video sequences is the task of function
      // CmpGraphics::Frame::visRep(bool,bool, bool).
      // If the first and the last of the boolean arguments is 'true' a
      // jpg-file of the first of the repeated frames will be created in the
      // same directory as the ppm images.

void ppm2png(Word const& imgFileName);
      // We assume that in the directory from which the function ppm2png is
      // called there is a ppm-image file with name given by the argument of the
      // function (this name thus has to end in .ppm).
      // As a result of calling ppm2png in the same directory there will be
      // an png-file with a name as the input but with .ppm replaced by .png.
      //
      // If one creates a simulation video there it may sometimes look
      // advantageous to repeat one image several times to get the impression
      // of a still image that invites inspection. When writing a comment to
      // the video as a html-file one would like to refer to these faked still
      // images by a href to an actual still image. Creation of the pseudo-still
      // video sequences is the task of function
      // CmpGraphics::Frame::visRep(bool,bool, bool).
      // If the first and the last of the boolean arguments is 'true' a
      // jpg-file of the first of the repeated frames will be created in the
      // same directory as the ppm images.


////////////////////// function glue ////////////////////////////////////////

Word glue(Word const& path, Word const& file);
   //: glue
   // Connects a (partial)directory name and a file name
   // to give a full file name. 'glues' path and file together.
   // Define bool extended = startSep(file) (see cpmsystem.cpp)
   // For extended==false, the result is simply
   // path&Word('/')&file. This is the correct behavior if file is a
   // normal filename which does not contain directory separators.
   // if extended==true we take into account that file may start
   // with ./ or ../. Then these dots will be eliminated.
   // Actually all all dots will be eliminated except the one
   // preceeding the file extension and

//////////////////////////// class Message //////////////////////////////////

class Message{ // provides basic output (unidirectional communication)
```

```
   // channels for a program. Two output files (cpmcerr and cpmdata
   // according to their define alias, see comments to data member
   // 'out' for these important entities) and a status bar featuring
   // 4 addressable segments are made available.
   // If we are running a console application, messages also go to the
   // console.
   // There is also limited communication in the other direction:
   // A file cpmconfig.ini, if it can be opened, will be used to
   // modify some static data members of the present class. This
   // excludes the data used to create the files cpmcerr and cpmdata.
   //
   // This class only has static data and methods, thus all instances
   // are identical. The information for initializing the static data
   // comes from the file cpmsystemdependencies.h.
   // A statement
   //    Cpmsystem::updFromConfigFile();
   // allows to change most static data by replacing them by values
   // read from the ini-file cpmconfig.ini. Since updFromConfigFile is a
   // friend of Message, it can do this replacement of private data of
   // class Message.

friend void updFromConfigFile();
   // this allows to manipulate the private data of Message, by a
   // function which can be implemented only later when class
   // RecordHandler is available. Will be implemented in
   // cpmapplication.cpp. A project which does not call
   // function updFromConfigFile() needs not to have cpmapplication.cpp
   // in its file set.

static bool streamInitialized; // not needing B

static Z vpw_, vph_, fsw_, fsh_;
   // viewport width, viewport height,
   // framestore width, framestore height,
static Word sttmes1;
static Word sttmes2;
static Word sttmes3;
static Word sttmes4;

static std::filesystem::path wd_ ;
   //. wd: 'working directory'
   // To be initialized in function Messsage::ini(void)

static Z sttBarWid1;
   // status bar width 1
   // the status bar has 4 'panes' and the width of each is
   // characterized by a number; sttBarWidi is this number for the
   // i-th pane
static Z sttBarWid2;
static Z sttBarWid3;
```

```
   static Z sttBarWid4;

   static Word relPos1;
   static Word relPos2;
   static Word relPos3;
   static Word relPos4;
      //: relative position
      // Relative directory positions, presently used only to locate
      // basic input file  cpmconfig.ini starting from the location
      // of the executable when starting from the command line or
      // from the working directory as seen by an IDE one is working
      // with. The search for cpmconfig.ini will be done for relPos1,
      // relPos2, relPos3, relPos4 in succession. Values for these
      // 4 static variables get set by reading cpmsystemdependencies
      // during compilation. An example is
      /*
         #define CPM_REL_POS1 "."
         #define CPM_REL_POS2 "./control"
         #define CPM_REL_POS3 ".."
         #define CPM_REL_POS4 "../control"
      */
      // The file cpmconfig.ini will be read at run-time and thus allows to
      // set quantities the change of which should not require
      // recompilation. Examples for this are size of the application
      // window, input directory (e.g. ./webControl), and output
      // directories (e.g. ./r20220404a). This flexible handling of input
      // directories allows to feed several programs with identical input
      // for speed and accuracy benchmarking.
      // Notice that searching for cpmconfig.ini is automated only for
      // applications that have cpmapplication.cpp among its source
      // files.


   static bool initialized(void){ return streamInitialized;}
      //

protected: // for usage by derived class Error

   static Z maxNumMes;
      // number of 'unenforced' messages to be written on out (helps to
      // avoid creating unintentionally huge text files and to slow
      // program run down

   static B silent;
      // if this is true, the basic function communication runs idle
      // If it is set to true initially no log files will be created
      // and no possibility exists to swich it to false.
      // Thus setting CPM_NOLOG reliably says that there will no log
      // files be created during program run.
```

```
   static void communication(Word const&, Word const&, Word const&,
      bool, Z barSeg=1_Z);

public:

   static Word inpDir;
      //: input directory
      // directory root read from cpmconfig.ini that is available for
      // defining interaction of a program with the file system of the
      // machine
      // typically inpDir = "./control"

   static Word outDir1;
      //: output directory 1
      // directory to which cpmcerr.txt and cpmdata.txt get written.
      // Can't be changed by reading from config.ini.

   static Word outDir2;
      //: output directory 2
      // Directory to which output data of a program run will be written.
      // For projects that have cpmapplication.cpp among their source files
      // this can be set to a value different from outDir1 by reading from
      // config.ini. If the directory set here does not exist it will be
      // created. Changing this setting doesn't require recompilation.

   static Word program;
      // e.g. "classicalcrossway", "pala", "tut2"

   static Word subProgram;

   static Word sffd1;
      // source file for documentation
      // typically e.g. sffd1 = ../cpm3/include/quantumcrossway.h

   static Word sffd2;
      // typically e.g. sffd2 = ../cpm3/source/quantumcrossway.cpp


   static Word iniFile;
      //: initialization file
      // Name of an input data file which is often used to define the behavior
      // of a program of the same name (without an optional file extension,
      // which in most cases will be ".ini").
      // typically iniFile = program&".ini"

   static Word srcDir;
      //. source directory
      // typically dir ="./source"

   static Word srcDir2;
```

```
      //. source directory
      // typically dir ="../cpm3"

    static Word fontDir;
      //: font directory
      // font location  read from cpmconfig.ini that is available for
      // defining interaction of a program with the file system of the
      // machine. So a C+- user is completely free where to place
      // the C+- font file on his computer.

   static Word runName;
      // a descriptive (may be abbreviated) name for the intent or
      // some property of the run. Trivial examples: sr for short run and
      // lr for long run.

    static Word runId;
      //: run ID
      // Is created prior to the creation of the log files.

    static B appRunId;
      //: append run id
      // If this is true, the run ID will be appended to the names of
      // auto-created image files made by class CpmGraphics::Frame
      // and documentation files made by
      // class CpmApplication::IniFileBasedApplication.
      // In the latter case there are provisions in place
      // to prescribe name appendices in ini-files, for instance
/**************************************************************************
         selection
         W runName=090616x
**************************************************************************/
      // This mechanism alone
      // does not make evident what version of the log files describes
      // their generation. So if appRunId is true, one appends runID
      // to this runName so that documentation files and log files which
      // belong togeher always have runID as a common name fragment.
      // One may change appRunId from cpmconfig.ini, where there is an
      // entry
/**************************************************************************
         append run id
         B val=false // or true
**************************************************************************/
    // calling void CpmSystem::updFromConfigFile() reads cpmconfig.ini and
    // may change appRunId.

   static Z runIdLength;


// data
   static Z verbose;
```

```
      // controls the ammount of messages to be written to file in a way
      // that every message that writes under some value of
      // Message::verbose, will also be written under all higher values.

   static Z debug;
      // controls the behaviour of the cpmassert macro.
      // For debug==1 Message::warning will be executed
      // For debug==2 Message::error instead
      // In all other cases the asserted condition will not be tested

   static Z trigger;
      // Device for debugging, should be used only in code inserted for
      // debugging or temporary analysis. For instance one may
      // add
      //    cpmtrigger=1;
      // just before a dubious function call, and add to the block
      // of this function the statements:
      // if (cpmtrigger>0){
      //    cpmdebug(<argument 1>);
      //    cpmdebug(<argument 2>);
      //       ...
      //    cpmtrigger=0;}
      // This allows to make sure that the function one had in mind was
      // actually called, and it shows the arguments used for the call.
      // Who runs a debugger has probably never to do such tricks.

// functions
   static void ini(void);
      // one has not to call this function, but it can do no harm either,
      // helps experimentation in non-standard situations
   static void fini(void){ out.close(); outData.close();}
      // only after being sure that the communication files have been
      // closed one can copy them for final documentation purposes
   static Z pane1(){ return sttBarWid1;}
      //: pane 1
   static Z pane2(){ return sttBarWid2;}
      //: pane 2
   static Z pane3(){ return sttBarWid3;}
      //: pane 3
   static Z pane4(){ return sttBarWid4;}
      //: pane 4

   static Z w(){ return vpw_;}
      // tendative frame width in pixel
   static Z h(){ return vph_;}
      // tendative frame height in pixel
      // Intended is a call
      // Z nx=Message::w(), ny=Message::h();
      // CpmGraphics::Viewport vp(nx,ny,title);
      // Making a statusbar within this field is the task of Viewport.
```

```
 static Z ws(){ return fsw_;}
    // tendative frame store width in pixel
 static Z hs(){ return fsh_;}
    // tendative frame store height in pixel

 //static OFileStream out;
 //static OFileStream out;
 static ofstream out;
    // writing via '>>' to Message::out is the primarily provided
    // (unidirectional) communication facility of the system.
    // After program run, one will find the text file cpmcerr.txt in
    // the directory Message::outDir1 that contains all what has been
    // was written to stream Message::out.

    // If CPM_USE_MPI is defined and the number np of processes is > 1
    // the program is aware of its rank and this will be appended
    // to the filenames cpmcerr and cpmdata in a way that
    // alphabetic order is the natural order. E.g. for np=64
    // we have cpmcerr_01.txt,...cpmcerr_09.txt, cpmcerr_10.txt, ...,
    // cpmcerr_64.txt.
    // The next program run will overwrite cpmcerr(_rank).txt. In most
    // cases this is OK in order not to pollute the working directory.
    // Sometimes it is better to retain the cpmcerr, cpmdata files
    // e.g. for comparing them for two related runs. Of course,
    // we may rename the files after creation in an informative way
    // to enable this. There is, however, the function
    // CpmSystem::docRun(bool) which does this using Message::runId
    // as as a name appendix and copies the 'name-mangeled' files to
    // directory Message::outDir2. If cpmapplication.cpp is among the
    // source files of the project, entries in the file cpmconfig.ini
    // determine whether CpmSystem::docRun(bool) will be called
    // automatically and which preexisting directory will be used as
    // Message::outDir2.

// static OFileStream outData;
 static ofstream outData;
    // writing via '>>' to Message::outData is a second
    // (unidirectional) communication facility of the system.
    // After program run, one will find the text file cpmdata.txt in
    // the execution directory that contains all what was written to
    // stream Message::outData and to the text file
    // text file cpmdata.txt in the directory Message::outDir1. Function
    // CpmSystem::docRun(bool) will also rename and copy this file if the
    // boolean argument is given the value true.

 static void onStatusBar(Word const& w, Z barSegment=1_Z);
    // stores w in the Word memory for 'pane'=barSegment
    // If possible for the system, it displays the Words
    // sttmes1,...sttmes4 on the status bar of the application
```

```
      // window. The status bar has several segments, numbered
      // 1,2,... and the second argument determines the field
      // on which the message shall be placed. Presently
      // there are 4 such fields. In CONSOLE applications, only
      // w is displayed on the console.
      // For barSegment<=0 no action is done

   static void setMaxMes(Z mes=1000_Z){maxNumMes=mes;}
      // setting the maximum number of (non-enforced) messages to be
      // written on out.

   // writing to the status bar

   static void progress(Word const& taskName, R frac, Z barSegment=1_Z,
      Z field=3_Z); // field =3 is usual
      // Displays task progress on the status bar.
      // for 0<=frac<=1 we get 000, 001, 002, ... 999, 1000
      // as a sufficient information on the degree of completion, only if
      // a digit changes there is change in the display. This is the
      // situation for field==3; for field=2 we go from
      // 00 to 100; etc. Only field=1,2,3,4,5 work this way; for any other
      // value one gets the same output as for field==4.

// writing to out

   static void report(Word const& text, Z val);
      // writes the number val on out if verbose>0

   static void report(Word const& text, R val);
      // writes the number val on out if verbose>0

// writing numerical values to both the status bar and to out (=cpmcerr)
// Seeing characteristic floating point values displayed on the status
// bar during program run is often useful, especially for diagnosis
// of unexpected behavior. Consider for instance a discrete time dynamical
// simulation which displays for any step the value of dt (the time step).
// If the algorithm for a time step is very fast (as one wants it to be)
// a diagnostic message 'value(" dt = ", dt, 2)' may slow the program run
// by a factor 1000. Further, if writing on the status bar is accompanied
// by writing on cpmcerr, this log file may grow to a size that slows down
// the program run even more. So setting verbose carefully might be
// necessary. The influence of verbose can be seen from the following
// shared code which is used in all functions value...
/*
#define CPM_SC\
   static const Z prec=4;\
   if (verbose){\
      ostringstream ost;\
      ost.precision(prec);\
      ost<<text<<" = ";\
```

```
      ost.width(8);\
      ost<<valx;\
      if (verbose>2){\
         message(Word(ost.str()),seg);\
      }\
      else{\
        if (verbose==2) cpmcerr<<text<<" = "<<valx<<endl;\
        if (verbose>0) onStatusBar(Word(ost.str()),seg);\
      }\
   }
*/

   static void value(Word const& text, R val, Z barSegment=1_Z);
      // writes the number val on the status bar and cpmcerr under
      // control of verbose.

   static void values2(Word const& text, R val1, R val2, Z barSegment=1_Z);
      // writes the numbers val1 and val2 on the status bar and out if
      // verbose>0

   static void values3(Word const& text, R val1, R val2, R val3,
      Z barSegment=1_Z);

   static void values4(Word const& text, R val1, R val2, R val3,
      R val4, Z barSegment=1_Z);

   static void value(Word const& text, Z val, Z barSegment=1_Z);
         // writes the number val on the status bar and out if verbose>0

   Message(void);
      // initializes Message::out (if not done earlier)

   explicit Message(Word const& w);
      // initializes out (if not done earlier) and writes w to it and
      // to cout

// Message and warning only differ in the wording of the output.
// Two Word arguments allow to specify the class in the scope of which
// the message is created. For barSeg<=0, no action on the status bar.
// Reason: Sometimes it is important to write some data
// to cpmcerr which would overload and disturb the status display
// on the status bar. If barSeg<0, the message goes to the
// console if we have one, even if CPM_NOGRAPHICS is not defined.

   static void warning(Word const& text, bool fileAlways=false);
      // A simple warning message function writing on out.
      // If fileAlways is false, a limitation of messages to
      // be written to file will take place to avoid slowing down program
      // execution too much.
```

```
static void warning(std::ostringstream const& ost,
   bool fileAlways=false)
{ Message::warning(Word(ost),fileAlways);}

static void message(Word const& text, Z barSeg=1_Z,
   bool fileAlways=false);

static void message(std::ostringstream const& ost, Z barSeg=1_Z,
   bool fileAlways=false)
{ Message::message(Word(ost),barSeg,fileAlways);}

static void message(Word const& className, Word const& text,
   bool fileAlways=false);

static void warning(Word const& className, Word const& text,
   bool fileAlways=false);

static void message(Z mL, Word const& text, Z barSeg=1_Z,
   bool fileAlways=false);
   // messaging takes place only if cpmverbose>=mL

static void urgentMessage(Word const& text)
   // a message that will always be given
{
   Z verbMem=verbose;
   verbose=1; message(text, 0, false); verbose=verbMem;
}

static Word getIniFile(){ return iniFile;}

static void setIniFile(Word const& ifi){ iniFile=ifi;}

static void setSil(bool sil){ silent=sil;}
   // set silent

static bool getSil(){ return silent;}

static Word getRunId(){ return runId;}
   //: get run ID

static void setRunIdLength(Z const& l){ runIdLength=l;}
   //: set run ID length

static Z getRunIdLength(){ return runIdLength;}
   //: set run ID length

static void setRunId(Word const& id){ runId=id;}
   //: set run ID

static void setRunName(Word const& rn){ runName=rn;}
```

```
static Word getRunName(){ return runName;}

static void augRunId(Word const& topic)
   //: augment run ID
{ if (topic.dim()>0) runId=topic&"_"&runId;}
   // allows making the ID more 'speaking'

static Z get_viewport_w(){ return vpw_;}
static Z get_viewport_h(){ return vph_;}

static bool getAppRunId(){ return appRunId;}
   //: get append run ID

static void setAppRunId(bool app){ appRunId=app;}
   //: set append run ID

static Word getInpDir(){ return inpDir;}
   //: get input directory

static Word getOutDir1(){ return outDir1;}
   //: get output directory 1

static Word getOutDir2(){ return outDir2;}
   //: get output directory 2
   // This is the directory to which the images created by
   // CpmGraphics::Frame::vis(true,bool) get stored.
   // Projects which have cpmapplication.cpp among its source files
   // allow to set

static Word get_sffd1(){ return sffd1;}
   //: get source file1 for docu

static Word get_sffd2(){ return sffd2;}
   //: get source file1 for docu


static Word getFontDir(){ return fontDir;}
   //: get font directory

static Word ext(Word const& w){ return glue(inpDir,w);}
   //: extend
   // Returns a file name which can be used to open a file.
   // For instance,
   //    IFileStream ifs(Message::ext("basics.txt"));
   //    Z sel; ifs>>sel;
   // reads an integer sel from file "basics.txt" which will be
   // searched in the directory determined by Message's conception
   // of a input directory. This allows different programs (with
   // different execution directories) access the same input data
   // for reliable performance comparisons. The different
```

```
      // programs need to mention in their respective files cpmconfig.ini
      // the same directory location as input directory.
      // It also allows to test a single program with various input
      // data sets (e.g. an experimental one and one already delivered
      // to a customer) by changing the content of the programs
      // cpmconfig.ini. This file may e.g. contain the lines
/*
      d:/cpm/pala
         // stable delivery data
      d:/cpm/palaX
         // experimental data
*/
   // Such entries between which one can switch by commentarization,
   // help to keep track of what one is doing.

   static Word extOut(Word const& w)
   {
      if (w.isVoid()) return Word(); else return glue(outDir2,w);
   }
      //: extend (for) output directory (2)
      // Notice that only outDir2 (not outDir1) can be set via config.ini
      // The void Word as filename as argument in some filing function
      // is always meant as the direction to ignore this function call.
      // So, it should not be transformed in a non-void Word by appending
      // a directory name to it. Changing the old and heavily used function
      // glue(...) to take this into account seems dangerous.

};

void error(Word const& w);
void error(std::ostringstream const& ost);
void error(Word const& w1, Word const& w2);
void lethalError(Word const& w);
   // does not write on cpmcerr.txt

//////////////////////////// class Error ////////////////////////////

class Error: public Message{ // used if C+- classes throw errors
    // No new data to Message, thus also a pure action class.
    // Since the action is determined by the constructor, the
    // functionality is new.
    // Inheritence allows to use the data and methods of Message to use
    // for implementation.

public:

explicit Error(Word const& messageText=Word("unspecified error"));
       // reporting a text in case of error, throws terminating exception

   Error(Word const& className, Word const& messageText);
```

```
       // reporting a class name and a text in case of error,
       // throws terminating exception
   friend void error(Word const& w);
      // terminates execution only if Message::debug>0
   friend void error(std::ostringstream const& ost);
   friend void error(Word const& w1, Word const& w2);

};

inline void error(std::ostringstream const& ost)
   { CpmSystem::error(Word(ost));}

struct Exception{ // simple debugging tool
// instead of saying cpmmessage(" reached point 3"); one simply says
// Exception(3);
   Z i_;
   Exception(Z i):i_(i){ Message::message("Exception "&Word::write(i_));}
};

///////////////// class Timer ////////////////////////////////////////

void wait(R t, Z pane=0_Z);
      // delays program run for t seconds (especially to have time to
      // look to some screen information in absense of getchar() interupt
      // in Win32s programs. When the thread of control reaches a wait
      // statement it is not guaranteed that all graphical effects
      // created earlier on video memory can actually be seen on screen!
      // If the last argument is equal to one of the valid pane indexes
      // 1,2,3,4 and cpmverbose>0 the remaining time in seconds will be
      // shown there.
      // Of course, for t <= 0 nothing is being done.

R time(void);
      // returns the time at evaluation instant in seconds with respect
      // to the first call to time(). Relies on system function
      // gettimeofday.

class Timer{ // implements e.g. getSecondsLeft()
   R tBegin,tEnd;
   bool act;
public:
   Timer():tBegin(0_Z),tEnd(0_Z),act(false){}
   explicit Timer(R sp): tBegin(CpmSystem::time()),tEnd(tBegin+sp),
      act(true){}
      // sets tBegin to the time of function call
      // and tEnd to tBegin+sp. All times in seconds
   R getSecondsLeft()const;
      // returns the remaining span to tEnd expressed in seconds
      // For the default counter this function always returns 137*3600
   R getHoursLeft()const;
```

```
      // returns getSecondsLeft()/3600
   bool timeOut()const{ return getSecondsLeft()<=0_R;}

};

////////////////////////// class IFileStream ////////////////////////////
//
// typical usage of file stream classes
// Word fIn("c:/d/cpm/myfile.txt");
// Word fOut("log.txt");
// IFileStream infi(fIn);
// OFileStream outfi(fOut);
// V<R> vr;
// vr.scanFrom(infi());
// vr.prnOn(outfi()); // notice '()' in using the streams
// In cpmfile.h there will be defined classes OFile and IFile which
// define assignment and copy.
// See also classes IFile and OFile in cpmfile.h for files which
// satisfy the strict value interface.


class IFileStream{ // input file stream

   ifstream* fs_;
   const Word name_;
   typedef IFileStream Type;
   CPM_INVAR(IFileStream)

public:

   explicit IFileStream(Word const& fileName,
      bool safe=true, bool binary=binDef);
      // If the second argument is true one gets an error if the
      // file could not be opened, otherwise only a warning will
      // be issued.

   IFileStream():fs_(0){}

   //virtual ~IFileStream(){ fs_->close(); delete fs_;}
   virtual ~IFileStream()=default;

   char readChar(){ char res('x'); *fs_>>res; return res;}
      //: read character
      // reading a char from stream

   string readWord(){ string res; *fs_>>res; return res;}
      //: read word
      // reads the next contiguous group of non-whitespace characters

   string readLine(){ string res; std::getline(*fs_,res); return res;}
```

```
      //: read line
      // reads the next line with all charcters also whitespace

   bool isVal()const
      //: is valid
      // returns stream status
   { if (!*fs_||fs_->bad()) return false; else return fs_->good(); }

   Word getName()const{ return name_;}
      //: get name

   ifstream& operator()(void){return *fs_;}
      // returns the object for direct access

   bool skipComments();
      // over-reads lines starting with '/', ';', '*', or '#'
};

////////////////////////// class OFileStream ////////////////////////////
//
//class OFileStream{ // output file stream
//
//   ofstream* fs_;
//   const Word name_;
//   typedef OFileStream Type;
//   CPM_INVAR(OFileStream)
//public:
//
//   explicit OFileStream(Word const& fileName,
//      bool safe=true, bool binary=binDef);
//
//   OFileStream():fs_(0){}
//
//  // virtual ~OFileStream(){ fs_->close();delete fs_;}
//   virtual ~OFileStream()=default;
//
//    bool isVal()const
//       //: is valid
//       // returns stream status
//    { if (!*fs_||fs_->bad()) return false; else return fs_->good(); }
//
//   Word getName()const{ return name_;}
//      //: get name
//
//   ofstream& operator()(void){return *fs_;}
//      // returns the object for direct access,
//      // this is all one needs to write to the object
//};

void copyFile(Word const& fileNameIn, Word const& fileNameOut);
```

```
   //: copy file
   // lean file copy function, closely following BS3, p. 638 function main
   // The first argument needs to be the name of a file that can be opened
   // for reading.
   // The output file will be created (and written to) and will have the
   // name given by the second argument. Since as the result of a
   // successful function call the output file exists in the file system
   // of the machine, the file needs not to be returned.
   // Any malfunction causes an exit from the program after having left a message
   // on cout. Since the raison d'etre of this function is to copy
   // the files cpmcerr.txt and cpmdata.txt, it is important that no writing
   // on the file 'fileNameIn' takes place.

void docRun(bool all=false);
   //: document run
   // creates a copy of cpmcerr.txt and, if all==true, also cpmdata.txt
   // with Message::getRunId() appended to the name in the directory
   // Message::outDir2. Notice that Message::outDir1 is the place where
   // the original cpmcerr.txt and cpmdata.txt are placed.
   // This not up to date. Much more files are get copied to Message::outDir2.
   // If video files were created they also end up in Message::outDir2.
   // Has the new feature which makes use of Message::subProgram.

} // namespace

// An assert macro which is fully under the control of the C+- class
// system. The first argument should be a piece of C++ code which
// evaluates to bool. (Thus no C++-function can do the job since there
// are no variables that hold pieces of code)
// Example of usage:
//    cpmassert(x>0,"norm in function eval should be positive");
// notice that the final ';' is not needed

#define cpmassert(X,Y)\
if (CpmSystem::Message::debug==1){\
if (!(X)) CpmSystem::Message::warning(CpmRoot::Word(#X)&\
   " violated: "&Y);}\
else if (CpmSystem::Message::debug==2)\
{ if (!(X)) CpmSystem::error(CpmRoot::Word(#X)&" violated: "&Y);}
   // example of usage: cpmassert(v.b()==1," index convention");

//////////////////// short names ////////////////////////////////////////
#define cpmcerr         CpmSystem::Message::out
#define cpmdata         CpmSystem::Message::outData
#define cpmverbose      CpmSystem::Message::verbose
#define cpmtrigger      CpmSystem::Message::trigger
#define cpmsilent       CpmSystem::Message::getSil
#define cpmdbg          CpmSystem::Message::debug
   // the name cpmdebug, which would be natural here, was defined
   // earlier, see cpmtypes.h
```

```
#define cpmmessage      CpmSystem::Message::message
#define cpmurgent       CpmSystem::Message::urgentMessage
#define cpmwarning      CpmSystem::Message::warning
#define cpmprogress     CpmSystem::Message::progress
#define cpmreport       CpmSystem::Message::report
#define cpmstatus       CpmSystem::Message::onStatusBar
#define cpmrunid        CpmSystem::Message::getRunId()
#define cpmerror        CpmSystem::error
#define cpmwait         CpmSystem::wait
#define cpmvalue        CpmSystem::Message::value
#define cpmvalues2      CpmSystem::Message::values2
#define cpmvalues3      CpmSystem::Message::values3
#define cpmvalues4      CpmSystem::Message::values4
#define cpmtime         CpmSystem::time
#define cpmexc          CpmSystem::Exception

#endif // guard
```

33 cpmsystem.cpp

```
//? cpmsystem.cpp
//? Status of work 2023-10-20.
//?
//?  ...


#include <sys/time.h> // <time.h> not sufficient
#include <iomanip>

#include <cpmgreg.h>
#include <cpmsystem.h>
#include <cpmsystemdependencies.h>

#if !defined(CPM_NOGRAPHICS)
   #include <cpmviewport.h>
#endif

using namespace CpmStd;
using CpmRoot::N;
using CpmRoot::Z;
using CpmRoot::R;
using CpmRoot::B;
using CpmRoot::Word;
using CpmRoot::toDouble;
using CpmArrays::V;


////////////////////////// class B ////////////////////////////////////////////

bool B::prnOn(ostream& out)const
{
   return CpmRoot::write(x1,out);
}

bool B::scanFrom(istream& in)
{
   return CpmRoot::read(x1,in);
}

Z B::com(const B& a)const
{
   if (x1<a.x1) return 1;
   if (x1>a.x1) return -1;
   return 0;
}

#if defined(CPM_WRITE_TITLE)
```

```
   B CpmRoot::wrtTit{true};
#else
   B CpmRoot::wrtTit{false};
#endif

B CpmRoot::operator""_B(unsigned long long int n)
{ bool b=(bool)n; return B(b);}

////////////////////// class Message ///////////////////////////////////////

namespace{
   const char sep='/';
   const char dot='.';

   bool startDot(Word const& file)
   {
      if (file.dim()==0) return false;
      return file[1]==dot;
   }

   bool absPath(Word const& file)
   // if file start with /home/... it is an absolute path
   // and should not be augmented by a leading dot
   {
      if (file.dim()<6) return false;
      return file[1]==sep && file[2]=='h' &&
      file[3]=='o' && file[4]=='m' && file[5]=='e'
      && file[6]==sep;
   }

   bool startSep(Word const& file)
   {
      if (file.dim()==0) return false;
      return file[1]==sep && !absPath(file);
   }
}

Word CpmSystem::glue(Word const& path, Word const& file)
{
// Do not insert diagnostic messages, like cpmdebug(temp); into
// this function block, since the function is used in Message::ini()
// before the messaging file cpmcerr exists.
   if (file.dim()==0) return path;
   if (path.dim()==0) return file;
   Word res= startSep(file) ? path&file : path&Word(sep)&file;
   if (startSep(res)) res=Word(dot)&res;
   return res;
}

void CpmSystem::ppm2mp4(Word const& movieName, Z frameRate)
```

```
{
   cpmcerr<<"CpmSystem::ppm2mp4(Word, Z) started"<<endl;
   if (movieName.dim()==0){
      cpmcerr<<" no movie name given, no movie written"<<endl;
   }
   Z vpw=Message::get_viewport_w();
   Z vph=Message::get_viewport_h();
   std::filesystem::path fcp(std::filesystem::current_path());
   cpmcerr<<"creating movie at "<<fcp<<endl;

   Word command("ffmpeg -f image2 -pattern_type glob -framerate ");
   command&=cpm(frameRate);
   command&=" -i \"*.ppm\" -s ";
   command&=cpm(vpw);
   command&="x";
   command&=cpm(vph);
   command&=" -pix_fmt yuv420p ";
   command&=movieName;
   command&="_"&Message::getRunName();
   command&="_"&Message::getRunId();
   command&=".mp4";
   Z sysRes1=system(-command);
   cpmcerr<<"A movie file has been written, purging of ppm-files ahead"<<endl;
   Word cd2("rm *.ppm");
   Z sysRes2=system(-cd2);
   cpmcerr<<"purging of ppm-files done"<<endl;
   cpmcerr<<"sysRes1 = "<<sysRes1<<", sysRes2 = "<<sysRes2<<endl;
   cpmcerr<<"CpmSystem::ppm2mp4(Word, Z) done"<<endl;
}

void CpmSystem::ppm2jpg(Word const& imgFileName)
{
   cpmcerr<<"CpmSystem::ppm2jpg(Word) started"<<endl;
   if (imgFileName.dim()==0){
      cpmcerr<<" no image file name given, no image written"<<endl;
   }
   Word imgFileName2=imgFileName.cut(4)&".jpg";
   Word command("pnmtojpeg --quality=97 ");
   command&=imgFileName;
   command&=" > ";
   command&=imgFileName2;
   Z sysRes=system(-command);
   //Word cd2("rm "&imgFileName);
  // Z sysRes2=system(-cd2);

   cpmcerr<<"sysRes = "<<sysRes<<endl;
   cpmcerr<<"In CpmSystem::ppm2jpg "<<imgFileName2<<" obtained from "
   <<imgFileName<<endl;
}
```

```
void CpmSystem::ppm2png(Word const& imgFileName)
{
   cpmcerr<<"CpmSystem::ppm2png(Word) started"<<endl;
   if (imgFileName.dim()==0){
      cpmcerr<<" no image file name given, no image written"<<endl;
   }
   Word imgFileName2=imgFileName.cut(4)&".png";
   Word command("pnmtopng ");
   command&=imgFileName;
   command&=" > ";
   command&=imgFileName2;
   Z sysRes=system(-command);
   cpmcerr<<"sysRes = "<<sysRes<<endl;
   cpmcerr<<"In CpmSystem::ppm2png "<<imgFileName2<<" obtained from "
   <<imgFileName<<endl;
}



////////////////////// class Message ///////////////////////////////////
// initialization by code
Z CpmSystem::Message::verbose=2;
Z CpmSystem::Message::debug=2;
Z CpmSystem::Message::trigger=0;
Z CpmSystem::Message::maxNumMes=100000;
B CpmSystem::Message::appRunId{true};
Word CpmSystem::Message::sttmes1="*";
Word CpmSystem::Message::sttmes2="**";
Word CpmSystem::Message::sttmes3="***";
Word CpmSystem::Message::sttmes4="****";
Word CpmSystem::Message::runId="";
Word CpmSystem::Message::runName="";
Word CpmSystem::Message::iniFile="";
Word CpmSystem::Message::srcDir="";
Word CpmSystem::Message::program="";
Word CpmSystem::Message::subProgram="";
Word CpmSystem::Message::sffd1="";
Word CpmSystem::Message::sffd2="";
ofstream CpmSystem::Message::out;
ofstream CpmSystem::Message::outData;

//initialization from cpmsystemdependencies.h
#if defined(CPM_RUN_ID_LENGTH)
   Z CpmSystem::Message::runIdLength=CPM_RUN_ID_LENGTH;
#else
   Z CpmSystem::Message::runIdLength=4;
#endif

#if defined(CPM_OUT_DIR) // this is the 'new style'
   // this style assumes an entry '#define CPM_OUT_DIR "."' in cpmsystemdependencies.h
```

```
   Word CpmSystem::Message::outDir1=CPM_OUT_DIR;
      // setting the directory location of cmcerr and cpmdata
   Word CpmSystem::Message::outDir2=CPM_OUT_DIR;
      // This directory will be used for storing a docu version of
      // cpmcerr.txt and cpmdat.txt. The new style is to set this docu-related
      // directory in the file cpmconfig.ini, the reason for this being, that
      // changes in cpmconfig.ini dont't ask for recompilation, whereas changes
      // in cpmsystemdependencies.h do. This is relevant since changes in outDir2
      // typically ocurr from program run to program run ( unless in phases of
      //  'wild experimentation')
#else // old style, outDir2 set in cpmsystemdependencies.h and thus
      // needs recompilation upon change
   Word CpmSystem::Message::outDir1=CPM_REL_POS2;
   Word CpmSystem::Message::outDir2=CPM_REL_POS2;
#endif

Word CpmSystem::Message::relPos1=CPM_REL_POS1;
Word CpmSystem::Message::relPos2=CPM_REL_POS2;
Word CpmSystem::Message::relPos3=CPM_REL_POS3;
Word CpmSystem::Message::relPos4=CPM_REL_POS4;
Word CpmSystem::Message::inpDir=CPM_INP_DIR;
Word CpmSystem::Message::fontDir=CPM_FONT_DIR;
//CpmSystem::OFileStream CpmSystem::Message::out(Word("xxx"));

#if defined(CPM_NOLOG)
   bool CpmSystem::Message::streamInitialized=true;
   B CpmSystem::Message::silent{true};
#else
   bool CpmSystem::Message::streamInitialized=false;
   B CpmSystem::Message::silent{false};
#endif
Z CpmSystem::Message::sttBarWid1=CPM_PANE1;
Z CpmSystem::Message::sttBarWid2=CPM_PANE2;
Z CpmSystem::Message::sttBarWid3=CPM_PANE3;
Z CpmSystem::Message::sttBarWid4=CPM_PANE4;

Z CpmSystem::Message::vph_=CPM_HEIGHT;
Z CpmSystem::Message::vpw_=CPM_WIDTH;
Z CpmSystem::Message::fsh_=CPM_HEIGHT;
Z CpmSystem::Message::fsw_=CPM_WIDTH;

// functions

void CpmSystem::Message::ini(void)
{
   if (initialized()) return;
      cout<<endl<<"Message::ini(void) started"<<endl;
      std::filesystem::path wd_(std::filesystem::current_path());
      cout<<"Current path (working directory) is "<<wd_<<endl;
      Word loc("CpmSystem::Message::ini(void)");
```

```
      Word logFile("cpmcerr");
      Word datFile("cpmdata");
      CpmTime::Greg date;
      date.now_();
      runId=date.getCode().tail(runIdLength);
      // runId will no longer be used to create a name appendix for
      // cpmcerr and cpmdata, since 'personalized' versions will be generated
      // in the same directory as the other output by function
      // CpmSystem::docRun(bool). runId can be changed by getting a new
      // value for runIdLength from cpmconfig.ini. In the C+- framework
      // the run Id gets used only after the deployment of cpmconfig.ini
      // (if there is no such file, of course the presently generated
      // runId will be employed.
      logFile&=".txt";
      datFile&=".txt";
// the folowing two lines were added 2014-01-19 after a new installation of
// Ubuntu as a conseqence of which the log files according to the unchanged
// logic were created always directly in the user directory
// (/home/mutze in my case).
      logFile=outDir1&"/"&logFile;
      datFile=outDir1&"/"&datFile;
      cout<<endl<<" trying to open "<<-logFile<<endl;
      //out=OFileStream(logFile);
      out.open(-logFile, std::ios_base::out);
      bool streamInitialized1=true;
      if (!out) {
         streamInitialized1=false;
         cout<<endl<<"stream out is not valid"<<endl;
      }
      else cout<<endl<<"stream out is valid"<<endl;

      cout<<endl<<" trying to open "<<-datFile<<endl;
      outData.open(-datFile, std::ios_base::out);
     // outData=OFileStream(datFile);
      bool streamInitialized2=true;
      if (!outData){
         streamInitialized2=false;
         cout<<endl<<"stream outData is not valid"<<endl;
      }
      else cout<<endl<<"stream outData is valid"<<endl;

      if (streamInitialized1&&streamInitialized2){
         streamInitialized=true;
         date.writeFormatted(out);
         date.writeFormatted(outData);
         Word mes=loc&
         ": files "&logFile&" and "&datFile&" successfully created,\n";
         cout<<endl<<-mes<<endl;
         cout<<"cpmmessage ahead"<<endl;
         cpmmessage(mes); // works
```

```
         cout<<"cpmmessage done"<<endl;
      // The files get created in the executable's working directory.
      // In MS Visual C++ the executable is automatically put (can be
      // changed in the 'project properties') into a Release or Debug
      // subdirectory of the project directory. cpmcerr.txt and
      // cpmdata.txt then get created just outside of these
      // subdirectories and thus directly in the project directory.
      }
      else{ // then we had trouble in creating the message streams
         if (streamInitialized1){ // since we really need only logFile, this
         // might be a situation not asking for stopping the program
            Word mes1=loc&": not able to create "&datFile;
            cout<<endl<<-mes1<<endl;
            cpmmessage(mes1);
         }
         else{
            Word mes2=loc&": not able to create "&logFile&
               ", stopping program";
            cout<<endl<<mes2<<endl;
            cpmmessage(mes2);
            throw;
            // something went fundamentally wrong, no message can be sent
         }
      }
   cout<<endl<<"Message::ini(void) done"<<endl;
}

CpmSystem::Message::Message(void)
{
   ini();
}

CpmSystem::Message::Message(Word const& w)
{
   ini();
   cout<<"Message::Message(Word) started)"<<endl;
   if (verbose>0) cpmcerr<<endl<<w<<endl;
   std::cout<<endl<<w<<endl;
   cout<<"Message::Message(Word) done)"<<endl;
  //  if (!out) cpmerror("CpmSystem::Message::Message(...): out==0");
}

#if !defined(CPM_NOGRAPHICS)
void CpmSystem::Message::onStatusBar(Word const& w, Z i)
{
   static Z firstrun_=1;
   if (i<=0) return; // was i<0 till 2006-06-30
   //in this case we can't write on the status bar before a graphical
   // window is available
   bool vii=CpmGraphics::Viewport::isInitialized();
```

```
   if (firstrun_ && vii){
      firstrun_=0;
      for (Z j=1; j<=4;j++) {
         CpmGraphics::Viewport::onStatusBar(" pane "&cpm(j)&" active", j);
      }
      cpmwait(0.5);
   }
   if (vii) CpmGraphics::Viewport::onStatusBar(w, i);
}
#else
   void CpmSystem::Message::onStatusBar(Word const& w, Z i){}
#endif

void CpmSystem::Message::communication(Word const& classText,
   Word const& commType, Word const& commText, bool fileAlways,
   Z barSeg)
// common building block for messages and warnings, not for errors
// Dependence on preprocessor directives:
// _CONSOLE: everything written on cpmcerr.txt will also be written on
// cout
{
   static R tRetard=0.;
   if (silent) return; // this allows us to use C+- functions which
      // contain cpmmessage or cpmwarning statements in situations
      // in which creation of the log streams did not yet happen.
   static Z messageCounter=0;
   ini(); // esuring that 'out' can be used, no work load if already
   // initialized. For silent==true we never come here by message
   // calling and thus we are sure never to create the log files
   // as long as silent==true. If silent gets switched to false
   // (by code or whatever, next call to the present function will
   // call ini() and thus create the log files.
   Z eowMem=Word::getEndOfWord();
   Word::setEndOfWord(0);
      // then writing Words to a stream has the same effect as writing
      // string literals
   if (verbose){ // only then, something has to be done
      messageCounter++;  // it is useful to count the messages
         // even if they are not restricted in number
      ostringstream ostStatus;
      ostStatus<<classText<<" "<<commType<<" "<<commText<<
      " # "<<messageCounter;
         // status bar should be written also if number of filed messages
         // is limited
      if (barSeg>0){
         onStatusBar(Word(ostStatus.str()),barSeg);
         if (messageCounter>110) cpmwait(tRetard,2);
      }
      ostringstream ostFile;
      if (fileAlways || messageCounter<maxNumMes){
```

```
         ostFile<<endl<<"counter = "<<messageCounter<<endl;
         ostFile<<classText<<" "<<commType<<" "<<commText<<endl;
         R t=time(); ostFile<<"  "<<cpmtod(t)<<" s after program start";
         cpmcerr<<ostFile.str()<<endl<<std::flush;
#if defined(_CONSOLE)
         //if (barSeg<0) std::cout<<ostFile.str()<<endl;
         std::cout<<ostFile.str()<<endl;
#endif
#if defined(CPM_NOGRAPHICS)&&defined(_CONSOLE)
         if (barSeg>0) std::cout<<ostFile.str()<<endl;
#endif
      }
      if (messageCounter==maxNumMes){
         ostFile<<endl<<
         "No more normal messages or warnings will be filed";
         cpmcerr<<ostFile.str()<<endl;
#if defined(CPM_NOGRAPHICS)&&defined(_CONSOLE)
         std::cout<<ostFile.str()<<endl;
#elif defined(_CONSOLE)
         if (barSeg<=0) std::cout<<ostFile.str()<<endl;
#endif
      }
   }
   Word::setEndOfWord(eowMem);
      // restoring the writing behavior of Word
}

void CpmSystem::Message::warning(Word const& errorText, bool fileAlways)
{
   static R waitTime=0.4;
   waitTime*=0.9;
      // if there are many warnings the run should not get slowed down
      // considerably
   communication("","Warning:",errorText, fileAlways);
   if (cpmverbose>=2) cpmwait(waitTime);
}

void CpmSystem::Message::message(Word const& text, Z barSeg,
   bool fileAlways)
{
   communication("","Message:",text,fileAlways,barSeg);
}

void CpmSystem::Message::message(Z mL, Word const& errorText,
   Z barSeg, bool fileAlways)
{
   if (cpmverbose>=mL)
   communication("","Message:",errorText, fileAlways,barSeg);
}
```

```
void CpmSystem::Message::warning(Word const& className,
   Word const& errorText, bool fileAlways)
{
   communication(className,"Warning:",errorText, fileAlways);
}

void CpmSystem::Message::message(Word const& className,
   Word const& errorText, bool fileAlways)
{
   communication(className,"Message:",errorText, fileAlways);
}

void CpmSystem::Message::progress(Word const& taskName, R frac, Z seg,
   Z field)
// new implementation (2005-04-17) asks for graphical action only if
// needed.
// This assumes that a specific segment on which progress display is
// active will not have to insert messages of different character.
// Actually the static memory elements should be associated with the
// segments directly and not only within a specific function such as
// 'progress'.
{
   if (silent) return;
   static Word mem1;
   static Word mem2;
   static Word mem3;
   static Word mem4;
   Word act1,act2,act3,act4;
   R fac2=1000;
  // R_ frac2=cpmtod(frac);
   if (field==3) ;
   else if (field==1) fac2=10;
   else if (field==2) fac2=100;
   else if (field==4) fac2=10000;
   else if (field==5) fac2=100000;
   else field=3;
   bool write=false;
   Word act=cpm(cpmtoz(frac*fac2),field);
   if (seg==1){
      if (act!=mem1){
         mem1=act;
         write=true;
      }
   }
   else if (seg==2){
      if (act!=mem2){
         mem2=act;
         write=true;
      }
   }
```

```
   else if (seg==3){
      if (act!=mem3){
         mem3=act;
         write=true;
      }
   }
   else if (seg==4){
      if (act!=mem4){
         mem4=act;
         write=true;
      }
   }
   if (write){ // graphical action only if needed
      write=false;
      Word mes=(verbose && seg==1) ? "Task: "&taskName&": Progress " :
         taskName&": ";
      mes&=act;
      if (verbose>2){
         message(mes,seg); // messages go always to the status bar
      }
      else{
         onStatusBar(mes,seg);
      }
   }
}

// shared code
// prec was 4 now experimental value 8

#define CPM_SC\
   static const Z prec=5;\
   if (verbose){\
      ostringstream ost;\
      ost<<std::setprecision(prec);\
      ost<<text<<" = ";\
      ost.width(8);\
      ost<<valx;\
      if (verbose>2){\
         message(Word(ost.str()),seg);\
      }\
      else{\
        if (verbose==2) cpmcerr<<text<<" = "<<valx<<endl;\
        if (verbose>0) onStatusBar(Word(ost.str()),seg);\
      }\
   }

void CpmSystem::Message::value(Word const& text, R val, Z seg)
{
   double valx=cpmtod(val);
   CPM_SC
```

```
}

void CpmSystem::Message::value(Word const& text, Z val, Z seg)
{
   Z valx=val;
   CPM_SC
}

#undef CPM_SC

void CpmSystem::Message::values2(Word const& text, R v1, R v2, Z seg)
{
   static const Z prec=4;
   if (verbose){
      ostringstream ost;
      ost.precision(prec);
      ost<<text<<": ";
      ost.width(10);
      double v1x=cpmtod(v1);
      double v2x=cpmtod(v2);
      ost<<v1x<<", "<<v2x<<endl;
      if (verbose>2){
         message(Word(ost.str()),seg);
      }
      else{
         onStatusBar(Word(ost.str()),seg);
      }
   }
}

void CpmSystem::Message::values3(Word const& text, R v1, R v2,
   R v3, Z seg)
{
   static const Z prec=3;
   if (verbose){
      ostringstream ost;
      ost.precision(prec);
      ost<<text<<": ";
      ost.width(7);
      double v1x=cpmtod(v1);
      double v2x=cpmtod(v2);
      double v3x=cpmtod(v3);
      ost<<v1x<<", "<<v2x<<", "<<v3x<<endl;
      if (verbose>2){
         message(Word(ost.str()),seg);
      }
      else{
         onStatusBar(Word(ost.str()),seg);
      }
   }
```

```
}

void CpmSystem::Message::values4(Word const& text, R v1, R v2,
   R v3, R v4, Z seg)
{
   static const Z prec=3;
   if (verbose){
      ostringstream ost;
      ost.precision(prec);
      ost.width(7);
      double v1x=cpmtod(v1);
      double v2x=cpmtod(v2);
      double v3x=cpmtod(v3);
      double v4x=cpmtod(v4);
      ost<<text<<": "<<v1x<<", "<<v2x<<", "<<v3x<<", "<<v4x;
      if (verbose>2){
         message(Word(ost.str()),seg);
      }
      else{
         onStatusBar(Word(ost.str()),seg);
      }
   }
}

#define CPM_SC\
   if (verbose){\
      ostringstream ost;\
      ost<<text<<"="<<valx;\
      message(Word(ost.str()));\
   }

void CpmSystem::Message::report(Word const& text, Z val)
{
   Z valx=val;
   CPM_SC
}

void CpmSystem::Message::report(Word const& text, R val)
{
   double valx=cpmtod(val);
   CPM_SC
}

#undef CPM_SC
//////////////////////////// class Error ////////////////////////////////

// notice: base class constructor Message() called anyway

CpmSystem::Error::Error(Word const& messageText)
{
```

```
#if !defined(CPM_NOLOG)
   silent=false; // errors have to be reported
   verbose=1;
   communication("C+-",messageText,"CpmError",true,-1);
      // -1 lets this write also to the console
#elif defined(_CONSOLE)
   cout<<endl<<"C+- "<<messageText<<" CpmError"<<endl;
#endif
   throw;  // errors should terminate
}

CpmSystem::Error::Error(Word const& className, Word const& messageText)
{
#if !defined(CPM_NOLOG)
   silent=false;
   verbose=1;
   communication(className,messageText,"CpmError",true,-1);
      // -1 lets this write also to the console
#elif defined(_CONSOLE)
   cout<<endl<<"C+- "<<className<<" "<<messageText<<" CpmError"<<endl;
#endif
   throw;  // errors should terminate
}

////////////////// class Timer ////////////////////////////////////////

R CpmSystem::Timer::getSecondsLeft()const
   {
      static const R future=137_R*3600_R;
      if (!act) return future;
      R tAct=CpmSystem::time();
      return tEnd-tAct;
   }

R CpmSystem::Timer::getHoursLeft()const
{
   const R inv_h=1_R/3600_R;
   return inv_h*getSecondsLeft();
}

////////////////////////// class-less functions //////////////////////////

void CpmSystem::lethalError(Word const& w)
{
   cout << "CpmSystem::letalError: " << -w << endl;
      // no service from Message needed. In particular,
      // cpmcerr.txt needs not to exist.
   exit(137);
}
```

```
void CpmSystem::error(Word const& w)
{
#if !defined(CPM_NOLOG)
   Message::ini();  // unclear whether needed
   Message::onStatusBar("Error",1);
   Message::onStatusBar(w,2);
#endif
   if (Message::debug>0) throw Error(w);
}

void CpmSystem::error(Word const& w1, Word const& w2)
{
#if !defined(CPM_NOLOG)
   Message::ini();
   Message::onStatusBar(w1,1);
   Message::onStatusBar(w2,2);
#endif
   throw Error(w1,w2);
}

R CpmSystem::time(void)
{
   static bool first=true;
   static timeval t1;
   if (first){
      gettimeofday(&t1, NULL);
      first=false;
   }
   timeval t2;
   gettimeofday(&t2, NULL);
   long sec  = t2.tv_sec  - t1.tv_sec;
   long usec = t2.tv_usec - t1.tv_usec;
   double t = sec + usec*1.e-6;
   return R(t);
}

void CpmSystem::wait(R t, Z pane)
{
   if (t<=0_R) return;
   R tStart=time();
   R tEnd=tStart+t;
   R tLeft=tEnd-tStart;
   Z sLeft=cpmrnd(tLeft);
   while (tLeft>0_R)
   {
      tLeft=tEnd-time();
      if (pane>=1_Z && pane<=4_Z){
         Z s0Left=cpmrnd(tLeft);
         if (s0Left<sLeft){
            sLeft=s0Left;
```

```
            cpmvalue(" tLeft",s0Left,pane);
               // visible only if cpmverbose>0
         }
      }
   }
   return;
}

////////////////// class IFileStream ////////////////////////
CpmSystem::IFileStream::IFileStream(Word const& fileName,
   bool safe, bool binary):name_(fileName)
{
   fs_=(binary? new ifstream(-fileName,std::ios_base::binary) :
      new ifstream(-fileName));
   if (!*fs_||fs_->bad()){  // till 2010-09-06 the test was fs==0
      Word mes="IFileStream(Word,bool,bool): unable to open "&fileName;
      if (safe) cpmerror(mes);
      else cpmwarning(mes);
   }
   else{
      cpmmessage("IFileStream(Word,bool,bool) did open "&fileName);
   }
}

bool CpmSystem::IFileStream::skipComments()
{
   return CpmRoot::eatComments(*fs_);
}

////////////////// class OFileStream ////////////////////////
CpmSystem::OFileStream::OFileStream(Word const& fileName,
   bool safe, bool binary):name_(fileName)
{
//   cout<<"entered OFileStream constructor code"<<endl;
   fs_=(binary? new ofstream(-fileName,std::ios_base::binary) :
      new ofstream(-fileName));
   if (!*fs_||fs_->bad()){
   //   cout<<"OFileStream(Word,bool,bool): unable to open "<<endl;
      Word mes="OFileStream(Word,bool,bool): unable to open "&fileName;
      if (safe) cpmerror(mes);
      else cpmwarning(mes);
   }
   else{
   //   cout<<"OFileStream(Word,bool,bool): did open "<<endl;
      cpmmessage("OFileStream(Word,bool,bool) did open "&fileName);
   }
}

void CpmSystem::copyFile(Word const& fileNameIn, Word const& fileNameOut)
{
```

```
   cout<<"try to copy "<<-fileNameIn<<" to "<<-fileNameOut<<endl;
   std::filesystem::copy(-fileNameIn,-fileNameOut);
   cout<<" copy done"<<endl;
}

void CpmSystem::docRun(bool all)
// has the new feature which makes use of Message::subProgram
{
   cout<<"CpmSystem::docRun(bool) started"<<endl;
      // cpmcerr is not available when function docRun gets called
      // in cpmapplication.cpp, CpmApplication::idleFunc()
   if (Message::getAppRunId()==false){
      cout<<"Message::getAppRunId()==false, docRun done trivially"<<endl;
      return;
   }
   Word appendix = Message::getRunName();
   appendix="_"&appendix&"_"; // only at this late
      // place in the call chain the final value of runId is available
   appendix&=Message::getRunId() ;
   cout<<"appendix = "<<-appendix<<endl;
   Word pre1=Message::outDir1;
   Word pre2=Message::outDir2;
   Word inpDir=Message::inpDir;
   Word program=Message::program;
   Word subProgram=Message::subProgram;
   Word s1=pre1&"/cpmcerr.txt";
   Word t1=pre2&"/cpmcerr"&appendix&".txt";
   Word s2=pre1&"/cpmdata.txt";
   Word t2=pre2&"/cpmdata"&appendix&".txt";
   Word s3=pre1&"/cpmconfig.ini";
   Word t3=pre2&"/cpmconfig"&appendix&".ini";
   Word s4=inpDir&"/"&program&".ini";
   Word t4=pre2&"/"&program&appendix&".ini";
   CpmSystem::copyFile(s1,t1);
   CpmSystem::copyFile(s2,t2);
   CpmSystem::copyFile(s3,t3);
   CpmSystem::copyFile(s4,t4);

   Word srcDir=Message::srcDir;
   cout<<"srcDir= "<<-srcDir<<endl;
   cout<<"subProgram.size() = "<<subProgram.size()<<endl;
   if (subProgram.size()>0)
   {
      Word s5=inpDir&"/"&subProgram&".ini";
      Word t5=pre2&"/"&subProgram&appendix&".ini";
      CpmSystem::copyFile(s5,t5);
      Word s6=srcDir&"/"&program&".cpp";
      Word t6=pre2&"/"&program&appendix&".cpp";
      CpmSystem::copyFile(s6,t6);
      Word s7=Message::sffd1;
```

```
      Word t7=pre2&"/"&Message::sffd1.baseName().appBefExt(appendix);
      cout<<"t7 = "<<-t7<<endl;
      CpmSystem::copyFile(s7,t7);
      Word s8=Message::sffd2;
      Word t8=pre2&"/"&Message::sffd2.baseName().appBefExt(appendix);
      cout<<"t8 = "<<-t8<<endl;
      CpmSystem::copyFile(s8,t8);
   }
   cout<<"CpmSystem::docRun(bool) done"<<endl;
}
```

34 cpmsystemdependencies.h

```
//? cpmsystemdependencies.h
//? Status of work 2023-10-20.
//?
//?  ...


#ifndef CPM_SYSTEMDEPENDENCIES_H_
#define CPM_SYSTEMDEPENDENCIES_H_
/*****************************************************************************
   cpmsystemdependencies.h
   Description: By this file one can place defines in implementation
    files in cases that the proper implementation is dependent on the
    computer system, e.g. an the access to a display or a file system.
    These defines may also modify the way who we use such facilities;
    e.g. using or not using the graphical display. Or, creating or
    not creating log files. Especially such changes of usage should
    not entail too much recompilation. So the present file should
    never be included in header files. Presently it is only included
    in six files: cpmnumbers.cpp, cpmsystem.cpp, cpmapplication.cpp
    cpmgraph.cpp, cpmimg24.cpp, cpminifilebasapp.cpp.

    Notice that changes in file cpmdefinitions.h trigger recompilation
    of virtually all C+- translation units.

    See also cpmconfig.ini, where most data can be overwritten.

*****************************************************************************/
// For the present project testcpm0 most of the following
// entries have no effect.
#define CPM_REL_POS1 "."
#define CPM_REL_POS2 "./control"
#define CPM_REL_POS3 ".."
#define CPM_REL_POS4 "../control"
   // four positions relative to the executables or the IDE's
   // idea of working directory. This is needed only for
   // finding cpmconfig.ini, on which the locations for further
   // input and for font files can be given

#define CPM_INP_DIR "./control"
   // default location of the directory for input
   // Typically not needed since the input directory can
   // comfortibly set in config.ini

#define CPM_FONT_DIR "../fonts"
   // location of the fonts file

#define CPM_HEIGHT 832
```

```
#define CPM_WIDTH  1432
   // screen data in pixels

#define CPM_PANE1 350
#define CPM_PANE2 250
#define CPM_PANE3 250
#define CPM_PANE4 100
   //relative size (arbitrary units) of the status bar subfields ('panes')

#define CPM_RUN_ID_LENGTH 4
   // number of digits of the run ID

#define CPM_WRITE_PRECISION 12
   // Numerical precision in writing floating point data to file

#define CPM_WRITE_TITLE
   // writing class instances to file will contain the
   // class name as 'title'

//#define CPM_CERR_EXTEND
   // if this is defined, the run ID will be appended
   // to the name of the log files

//#define CPM_NOLOG
   // if defined, no cpmcerr and cpmdata will be created

#define CPM_USE_EIGEN

#if defined(CPM_USE_EIGEN)
   // using the Eigen library for fast matrix and vector algorithms
   // for real and for complex quantities. The algorithms for
   // eigenvalues/eigenvectors and SVD do not cooperate perfectly with
   // multiple. The many resulting warnings can be disabled by "-w".
   // If multiple precision is based on boost instead of mpreal.h
   // one has even to compile with "-fpermissive" to avoid errors.
   #include <eigen3/Eigen/Dense>
#endif

#define CPM_NOGRAPHICS
   // if defined no graphical content will be shown on screen
   // and no graphics libraries (openGL and GLUT need to be linked

#endif
```

35 cpmtests.h

```
//? cpmtests.h
//? Status of work 2023-10-20.
//?
//?  ...


#ifndef CPM_TESTS_H_
#define CPM_TESTS_H_

/*
Purpose: Uniform test facility for finding out in an
unambiguous way whether a class satisfies a given
interface, such as the strict value interface or the
r-interface.
*/

#include <cpmtypes.h>
#include <cpmc.h>
#include <cpmvr.h>
#include <cpmsr.h>
#include <cpmfr.h>
#include <cpmm.h>
#include <cpmp.h> // for Vp
#include <iomanip> // for std::setprecision

namespace CpmTests{

   using namespace CpmStd;

   using CpmRoot::Z;
   using CpmRoot::R;
   using CpmRoot::Word;
   using CpmRoot::Root;
   using CpmArrays::V;
   using CpmArrays::Vo;
   using CpmRoot::B;
   using CpmSystem::IFileStream;
   using CpmSystem::OFileStream;
   using std::setprecision;

template <class T>
Z reg()
{
   bool a{false};
   Z res{0};
   T t{};
   Word nT="class "&t.nameOf();
```

```
   a=std::semiregular<T>;
   if (a) cout<<nT<<" is semiregular"<<endl;
   else {cout<<nT<<" is not semiregular"<<endl; res++;}
   a=std::regular<T>;
   if (a) cout<<nT<<" is regular"<<endl;
   else {cout<<nT<<" is not regular"<<endl; res++;}
   return res;
 }

// We assume that type T defines the operator  binary bool-valued
// operator ==
template <class T>
class Sym{ // testing symmetry of equality
public:
   Sym(){}
   bool operator()(T const& t1, T const& t2)
      // T == T is symmetric if this function returns true for all
      // input arguments (then we say: 'sym==true')
   {
      if (t1==t2) return (t2==t1); else return true;
   }
};
// Typical usage;
// T t1...;
// T t2...;
// Sym<T> s;
// bool b=s(t1,t2); // b ==true says that symmetry has been confirmed
// for the particular objects t1 and t2.


template <class T>
class Trans{ // testing transitivity of equality
public:
   Trans(){}
   bool operator()(T const& t1, T const& t2, T const& t3)
// T == T is an equivalence relation if sys==true and trans==true
   {
      if (t1==t2 && t2==t3) return (t1==t3); else return true;
   }
};

template <class T>
class DefaultConstructor{ // testing consistency of default constructor
public:
   DefaultConstructor(){}
   bool operator()(T const& t)
   // value of  t not used, carries type information
   {
      t; // to avoid warning concerning not making use of the argument
      T t1; T t2; return (t1==t2);
```

```
   }
};

template <class T>
class CopyConstructor{ // testing consistency of copy constructor
public:
   CopyConstructor(){}
   bool operator()(T const& t)
   {
      T t1(t); return t1==t;
   }
};

template <class T>
class Assignment{ // testing consistency of assignment
public:
   Assignment(){}
   bool operator()(T const& t1, T const& t2)
   {
      T tStart(t1);
      tStart=t2;
      return tStart==t2;
   }
};

template <class T>
class StrictAssignment{ // testing strict consistency of assignment
// Here we assume minimal infrastructure (only copy,assignement,equality).
public:
   StrictAssignment(){}

   bool operator()(T const& t1, T const& t2, T const& t3)
      // returns true if OK
   {
      T tStart(t1);
      T t2c(t2);
      tStart=t2c;
      T tMem(tStart);
      t2c=t3; // source for tStart's value re-defined
      return tStart==tMem; // detects change of tStart
         // (which should not happen).
   }

};

// Notice that so far we returned true for passing the test. The following
// two functions return the number of failures instead!

template <class T>
class StrictAssignment2{ // testing strict consistency of assignment
```

```
// for values created inside a function body. Needs more infrastructure
// of T than the previous class StrictAssignment<T>. Particularly we
// need a random generator and the distance function.
public:
   StrictAssignment2(){}
   static T gen(T const& t, Z j){ T res = Root<T>(t).ran(j); return res;}
      // studying return values which were primarily created inside a
      // function block
   R operator()(T const& t1)
      // returns an quantitative error varying between 0 (good) and 1
      // (bad)
   {
      T t2=gen(t1,7);
      T t3=gen(t2,6);
      T t4=gen(t3,5);
      T t4M(t4);
      // swapping t2 and t3
      T tTemp=t3;
      t3=t2;
      t2=tTemp;
      return Root<T>(t4).dis(t4M);
   }
};


template <class T>
class ValueBehavior{ // testing the property of a class T
   // to be a 'value class'. Relies on operator == and
   // three instances of T on input.
public:
   ValueBehavior(){}
   R operator()(T const& t1, T const& t2, T const& t3)
      // returns the number of errors
   {
      Z mL=3;
      Word loc("ValueBehavior::operator()(T,T,T)");
      CPM_MA
      DefaultConstructor<T> dc;
      CopyConstructor<T> cc;
      Assignment<T> a;
      StrictAssignment<T> sa;
      Z fc=0; // failure counter
      bool b1=dc(t1);
      if (!b1) {fc++;cpmmessage(loc&" failure in DefaultConstructor");}
      bool b2=cc(t1);
      if (!b2) {fc++;cpmmessage(loc&" failure in CopyConstructor");}
      bool b3=a(t1,t2);
      if (!b3) {fc++;cpmmessage(loc&" failure in Assignment");}
      bool b4=sa(t1,t2,t3);
      if (!b4) {fc++; cpmmessage(loc&" failure in StrictAssignment");}
```

```
   //   bool b5=sa(t1);
   //   if (!b5){fc++; cpmmessage(loc&" failure in StictAssignment II");}
      CPM_MZ
      return R(Z(fc));
   }
};

// Let T be a type for which the statement
// strictAssignment<T> sa; compiles then we say that the
// T implements the strict assignment on compilation.
// If we find sa(t1,t2,t3)==true for all T typed argument triplets
// based on the logic of the code we say that T implements the strict
// assignment.
// If we verify sa(t1,t2,t3)==true by running a program with sufficiently
// many arguments we may say that the strict assignment was tested with
// positive result.

// If for a fixed triplet of types T1, T2, C this
// function is true for all arguments, we says that C is a
// polymorphic single-slot-container for T1 and T2.

template <class T1, class T2, class C>
class PolymorphicSingle{ // testing polymorphic containers
public:
   PolymorphicSingle(){}
   bool operator()(const T1& t1, const T2& t2, const C& c )
   {
      C cc(c);
      cc()=t1;
      T1 tc1(cc());
      bool b1=(t1==tc1);
      cc()=t2;
      T2 tc2(cc());
      bool b2=(t2==tc2);
      return b1 && b2;
   }
};

template <class T1, class T2, class C>
class PolymorphicMulti{ // testing polymorphic containers
public:
   PolymorphicMulti(){}
   bool operator()(const T1& t1, const T2& t2, const C& c, Z i1, Z i2 )
   {
      if ( i1>=1 && i1<=c.dim() && i2>=1 && i2<=c.dim() && i1!=i2 ){
         C cc(c);
         cc[i1]=t1;
         cc[i2]=t2;
         T1 tc1(cc[i1]);
         T2 tc2(cc[i2]);
```

```
         return t1==tc1 && t2==tc2;
      }
      else return true;
   }
};

// If for a fixed triplet of types T1, T2, C this
// function is true for arguments, we says that c is a
// polymorphic multi-slot-container for T1 and T2.

// We will see that Vp<T> is a polymorphic multi-slot-container for all
// types T1, T2 equal to or derived from T.

template <class T>
class TestBase{ // base class for classes which test that class T
// implements a certain interface
protected:
   R errorSum;

   static R comp(Word top, T const& lhs, T const& rhs);
      // means for comparing results

   static R larger(Word top, R xL, R xS);
      // means for testing xL > xS. Returns 0 if the (expected)
      // inaquality is satisfied and xS-xU else.

   static R equal(Word top, R x1, R x2);
      // means for testing x1 == x2. Returns (x1-x2).abs()

   TestBase():errorSum(0.){}
public:
   static void wrt(ostringstream& ost){
      Word wd=Word(ost.str());
      wd.prnOn(cpmcerr);
      wd.prnOn(cout);
   }
   static void wrt(Word const& wd){
      wd.prnOn(cpmcerr);
      wd.prnOn(cout);
   }
   R val()const{return errorSum;}
};

template <class T>
R TestBase<T>::comp(Word top, T const& lhs, T const& rhs)
{
   using namespace CpmSystem;
   const R tiny=1e-4;

   R al=CpmRoot::absT<T>(lhs);
```

```
   R ar=CpmRoot::absT<T>(rhs);
   Word w1="Warning: trivial argument in ";
   Word w2=top;
   ostringstream ost;
   if (al==0. || ar==0.){
      ost<<endl<<(w1&w2);
   }
   R newError=Root<T>(lhs).dis(rhs);
   ost<<endl<<"Test of "
      <<top<<endl<<" error = "<<newError<<endl
      <<"cpmabs(lhs)= "<<al<<endl
      <<"cpmabs(rhs)= "<<ar<<endl
     ;
   if (newError>tiny || !cpmiva(newError)){
     ost<<endl<<" lhs="<<endl;
     CpmRoot::prnT<T>(lhs,ost);
     ost<<endl<<" rhs="<<endl;
     CpmRoot::prnT<T>(rhs,ost);
   }
   wrt(ost);
   return newError;
}

template <class T>
R TestBase<T>::larger(Word top, R xL, R xS)
{
   using namespace CpmSystem;
   Word w1="Warning: trivial argument in ";
   Word w2=top;
   ostringstream ost;
   if (xL==0. || xS==0.){
      ost<<endl<<(w1&w2);
   }
   R newError=xS>xL ? xS-xL : 0.;
   ost<<endl<<"Test of "
      <<top<<endl<<" error = "<<newError<<endl
      <<"xL= "<<xL<<endl
      <<"xS= "<<xS<<endl;
   wrt(ost);
   return newError;
}

template <class T>
R TestBase<T>::equal(Word top, R x1, R x2)
{
   using namespace CpmSystem;
   Word w1="Warning: trivial argument in ";
   Word w2=top;
   ostringstream ost;
   if (x1==0. || x2==0.){
```

```
      ost<<endl<<(w1&w2);
   }
   R newError=cpmabs(x1-x2);
   ost<<endl<<"Test of "
      <<top<<endl<<" error = "<<newError<<endl
      <<"x1= "<<x1<<endl
      <<"x2= "<<x2<<endl;
   wrt(ost);
   return newError;
}

////////////// testing the value interface ////////////////////////////

template <class T>
class Test_v: public TestBase<T>{ // testing the value interface

   typedef TestBase<T> Base;

public:
   Test_v(Z tvs=1, Z rep=4);
   // tvs: test value size ~ complexity of test
};

template <class T>
Test_v<T>::Test_v(Z tvs, Z rep):Base()
      // Tests all relations concerning the value interface and returns
      // an error sum
{
   using namespace CpmRoot;
   using namespace CpmRootX;

   const R dSmall=1e-6;
   const R dTiny=1e-12;

   Word top;
   Word loc("Test_v<T>::Test_v(Z tvs, Z rep)");
   R res=0;
   Z counter=0;
   T xd,x;
   T xt=xd.test(tvs);
   Z cc=137;
   cpmdbg=1; // experiment
   while (counter<rep){
      counter++;
      x=xt.ran(cc++);
     //  cpmassert(xt.dis(x)>dSmall,loc); // the random generator should
      // generate
      // new values, if it does so only after many iterations it is
      // probably not reliable, so we expect this to be valid for
      // sufficiently small
```

```
      // dSmall
      T y=xt.ran(cc++);
      cpmassert(y.dis(x)>dSmall,loc);
      T z=xt.ran(cc++);
      cpmassert(z.dis(x)>dSmall,loc);
      cpmassert(z.dis(y)>dSmall,loc);
      R r1=y.dis(z);
      R r2=z.dis(y);
      cpmassert(cpmabs(r1-r2)<dTiny,loc);

// now x,y,z are supposed to be sufficiently random to make test
// meaningful
      top="is copy equal to original ?";
      T xl(z);
      res+=Base::comp(top,xl,z);
      top="is assigned value equal to original ?";
      y=x;
      res+=Base::comp(top,y,x);
// This was all to be shown
   //   xt=z; // to start the new loop were the old one ended
   // this turned out not to be a good idea since the test objects
   // then become smaller and smaller so that dSmall and dTiny
   // become less and less adequate.
   }
   ostringstream ost;
   ost<<endl<<"  Test of Value Interface; errorsum= "<<res<<endl;
   Base::wrt(ost);
   Base::errorSum=res;
}

////////////// testing the strict value interface //////////////////////

template <class T>
class Test_sv: public TestBase<T>{ // testing the strict value interface

   typedef TestBase<T> Base;

public:
   Test_sv(Z tvs=1, Z rep=4);
   // tvs: test value size ~ complexity of test
};

template <class T>
Test_sv<T>::Test_sv(Z tvs, Z rep):Base()
      // Tests all relations concerning the strict value interface
      // repeatedly and returns an error sum.
{
   using namespace CpmRoot;
   using namespace CpmRootX;
```

```
   const R dSmall=1e-6;
   const R dTiny=1e-12;

   Word top;
   Word loc("Test_sv<T>::Test_sv(Z tvs, Z rep)");
   R res=0;
   Z counter=0;
   Z cc=137;
   T xd,x;
   T xt=xd.test(tvs);
   while (counter++ < rep){
      x=xt.ran(cc++);
      R d1=xt.dis(x);
   //   cout<<"d1 = "<<d1<<endl;
      cpmassert(d1>dSmall,loc); // the random generator should
      // generate new values, if it does so only after many iterations it
      // is probably not reliable, so we expect this to be valid for
      // sufficiently small dSmall
      T y=xt.ran(cc++);
      R d2=y.dis(x);
 //     cout<<"d2 = "<<d2<<endl;
      cpmassert(d2>dSmall,loc);

      T z=xt.ran(cc++);
      R d3=z.dis(x);
   //   cout<<"d3 = "<<d3<<endl;
      cpmassert(d3>dSmall,loc);

      R d4=z.dis(y);
    //  cout<<"d4 = "<<d4<<endl;
      cpmassert(d4>dSmall,loc);

      R r1=y.dis(z);
      R r2=z.dis(y);
      cpmassert(cpmabs(r1-r2)<dTiny,loc);
      T zr=z;

// now x,y,z are supposed to be sufficiently random to make test
// meaningful
      top="is copy equal to original ?";
      T xl(zr);
      z=z.ran(cc++);
      z=z.ran(cc++); // although zr was made from z, this change in z
         // should not affect zr, thus zr and xl have still to be equal
      res+=Base::comp(top,xl,zr);

      top="is assigned value equal to original ?";
      y=x;
      T x1=x;
      T x2=y;
```

```
// messing up the originals; x1 and x2 should remain unchanged
      x=x.ran(cc++);
      x=x.ran(cc++);
      y=y.ran(cc++);
      y=y.ran(cc++);
      res+=Base::comp(top,x1,x2);
// This was all to be shown

   }
   ostringstream ost;
   ost<<endl<<" Test of Strict Value Interface; errorsum="<<res<<endl;
   Base::wrt(ost);
   Base::errorSum=res;
}

//template <class T>
//Test_sv<T>::Test_sv(Z tvs, Z rep):Base()
//      // Tests all relations concerning the strict value interface
//      // repeatedly and returns an error sum.
//{
//   using namespace CpmRoot;
//   using namespace CpmRootX;
//
//   const R dSmall=1e-12;
//   const R dTiny=1e-16;
//
//   Word top;
//   Word loc("Test_sv<T>::Test_sv(Z tvs, Z rep)");
//   R res=0;
//   Z counter=0;
//   Z cc=137;
//   T xd,x;
//   T xt=xd.test(tvs);
//   while (counter++ < rep){
//      x=xt.ran();
//      R d1=xt.dis(x);
//      cout<<"d1 = "<<d1<<endl;
//      cpmassert(d1>dSmall,loc); // the random generator should
//      // generate new values, if it does so only after many iterations it
//      // is probably not reliable, so we expect this to be valid for
//      // sufficiently small dSmall
//      T y=xt.ran();
//      R d2=y.dis(x);
//      cout<<"d2 = "<<d2<<endl;
//      cpmassert(d2>dSmall,loc);
//
//      T z=xt.ran();
//      R d3=z.dis(x);
//      cout<<"d3 = "<<d3<<endl;
//      cpmassert(d3>dSmall,loc);
```

```
//
//      R d4=z.dis(y);
//      cout<<"d4 = "<<d4<<endl;
//      cpmassert(d4>dSmall,loc);
//
//      R r1=y.dis(z);
//      R r2=z.dis(y);
//      cpmassert(cpmabs(r1-r2)<dTiny,loc);
//      T zr=z;
//
//// now x,y,z are supposed to be sufficiently random to make test
//// meaningful
//      top="is copy equal to original ?";
//      T xl(zr);
//      z=z.ran();
//      z=z.ran(); // although zr was made from z, this change in z
//         // should not affect zr, thus zr and xl have still to be equal
//      res+=Base::comp(top,xl,zr);
//
//      top="is assigned value equal to original ?";
//      y=x;
//      T x1=x;
//      T x2=y;
//// messing up the originals; x1 and x2 should remain unchanged
//      x=x.ran();
//      x=x.ran();
//      y=y.ran();
//      y=y.ran();
//      res+=Base::comp(top,x1,x2);
//// This was all to be shown
//      //xt=z; // to start the new loop were the old one ended
//   }
//   ostringstream ost;
//   ost<<endl<<" Test of Strict Value Interface; errorsum="<<res<<endl;
//   Base::wrt(ost);
//   Base::errorSum=res;
//}


//////////// testing the r-interface ////////////////////////////////////

template <class T, class S>
class Test_r: public TestBase<T>{ // testing the r-interface
   // class for testing classes that
   // implement the r-interface
   // Testing e.g. Vr<R> is done e.g. by the following code
   // Test_r<Vr<R>,R> a(10);

   typedef TestBase<T> Base;
```

```
public:
   Test_r(Z tvs=1 , Z output=0, Z write=0);
   // tvs: test value size ~ complexity of test
};

template <class T, class S>
Test_r<T,S>::Test_r(Z tvs, Z output, Z write):Base()
      // Tests all relevant mathematical relations and returns
      // an error sum
      // output 0 : only end of test and error sum displayed
      // output 1 : output of every test. The potentially
      //      time consuming output of the test object is suppressed
      // output 2: only the first component of the test object is ??
      // displayed ??
      // output 3: the test object is displayed in full detail ??
      // This form only works for classes (not for the types R and Z. As
      // a compensation it shows the favorable property of not having to
      // know the namespace of T, since all access is intrinsic.??
      // If write is different from 0, a write/read test to and from file
      // will be executed. Only if write > 1,the difference between
      // written and read quantity will enter the error sum. This helps to
      // asses the error behavior for the newly included multiple
      // precision faclity, where otherwise the error would be dominated
      // by the write precision which I set independently from the number
      // precision.
{
   using namespace CpmSystem;
   using namespace CpmFunctions;
   using CpmRoot::nameOf;
   static const Z multOrder=4;
      // order test is made on an array of
      // multOrder*tvs elements
   //using Base::comp;

   Z mL=1;
   Word loc("Test_r(Z,Z,Z)");
   CPM_MA

   Z nOrd=multOrder*tvs;

   T t, v, x, y, z, xl, xr;
   Word top;

   R res=0.;
   ostringstream ost;
   ost<<endl<<" Test of class "<<t.nameOf()<<endl;
   Base::wrt(ost);

   if (output){
      cpmcerr<<endl<<
```

```
   " The distinguished objects associated with the class are: "<<endl;
}

T xd;
T xt=xd.test(tvs);
if (output){
   cpmcerr<<endl<<"test:";
   xt.toWord().prnOn(cpmcerr);
}
T xzero=xt.net(0);
if (output){
   cpmcerr<<endl<<"zero:";
   xzero.toWord().prnOn(cpmcerr);
}
T xunit=xt.net(1);
if (output){
   cpmcerr<<endl<<"unit:";
   xunit.toWord().prnOn(cpmcerr);
}

v=xt;
x=xt.ran(1);
y=xt.ran(2);
z=xt.ran(3);
T xWrite=z;

if (output){
   cpmcerr<<endl<<"first ran:";  x.toWord().prnOn(cpmcerr);
   cpmcerr<<endl<<"second ran:"; y.toWord().prnOn(cpmcerr);
   cpmcerr<<endl<<"third ran:";  z.toWord().prnOn(cpmcerr);
}
T v4=x;
T v5;
T v6=y;
v6=v5=x;
xl=v6;
xr=v4;
top="copy constructor, assignement, and equality";
  res+=Base::comp(top,xl,xr);

xl=x+y; xr=y+x; top="+ commutativity";
res+=Base::comp(top,xl,xr);
xl=(x+y)+z; xr=x+(y+z); top="+ associativity";
res+=Base::comp(top,xl,xr);
xl=(x*y)*z; xr=x*(y*z); top="* associativity";
res+=Base::comp(top,xl,xr);

T xZero=x.net(0);

xl=-x; xr=xZero-x; top="sign change";
```

```
  res+=Base::comp(top,xl,xr);
  xl=x*(y+z); xr=x*y+x*z; top="left-distributivity";
  res+=Base::comp(top,xl,xr);
  xl=(x+y)*z; xr=x*z+y*z; top="right-distributivity";
  res+=Base::comp(top,xl,xr);
  xl=x*(y-z); xr=x*y-x*z; top="left-distributivity of subtraction";
  res+=Base::comp(top,xl,xr);
  xl=(x-y)*z; xr=x*z-y*z; top="right-distributivity of subtraction";
  res+=Base::comp(top,xl,xr);
  // not clear why con(x*y) is not understood for T=Fr<Z,R>
  T xTemp=x*y;
  xl=xTemp.con(); xr=y.con()*x.con();
  top="conjugation commutativity";
  res+=Base::comp(top,xl,xr);
  x=z;
  T xu=x;

  xu=xu.net(1);
  if (output){
     cpmcerr<<endl<<"unit=";
     xu.toWord().prnOn(cpmcerr);
  }

  if (output){
     cpmcerr<<endl<<"x=";
     x.toWord().prnOn(cpmcerr);
  }

  T xi=x.inv();

  if (output){
     cpmcerr<<endl<<"xInv=";
     xi.toWord().prnOn(cpmcerr);
  }

  xl=x*xi; xr=xu; top="right inverse";
  res+=Base::comp(top,xl,xr);

  xl=xi*x; xr=xu; top="left inverse";
  res+=Base::comp(top,xl,xr);
// properties with scalars
  S s;
  s=CpmRoot::testT<S>(s,tvs);
  S sZero=CpmRoot::netT<S>(s,0);
  S sUnit=CpmRoot::netT<S>(s,1);
  S sInit=sUnit+sUnit+sUnit;
  sInit+=sInit;
  Z count=0;
  S s1=CpmRoot::ranT<S>(sInit,count);
  const Z countMax=100;
```

```
   while (s1==sZero || s1==sUnit){
      s1=CpmRoot::ranT<S>(sInit,++count);
      if (count>countMax){
         cpmwarning(
         "Test_r<T,S>: too many iterations (bad random generator of S)"
         );
         break;
      }
   }
   if (output){
      Word mes="iterations needed to get s1: "&cpmwrite(count);
      cpmcerr<<endl<<mes;
   }
   count=0;
   S s2=CpmRoot::ranT<S>(s1,++count);
   while (s2==sZero || s2==sUnit || s2==s1){
      s2=CpmRoot::ranT<S>(s1,++count);
      if (count>countMax){
         cpmwarning(
         "Test_r<T,S>: too many iterations (bad random generator of S)"
         );
         break;
      }
   }
   if (output){
      Word mes="iterations needed to get s2: "&cpmwrite(count);
      cpmcerr<<endl<<mes;
   }
   xl=s1*(x+y); xr=s1*x+s1*y; top="*s left-distributivity";
   res+=Base::comp(top,xl,xr);
   xl=(x+y)*s2; xr=x*s2+y*s2; top="*s right-distributivity";
   res+=Base::comp(top,xl,xr);
   xl=s1*(x-y); xr=s1*x-s1*y; top="*s left-distributivity of subtraction";
   res+=Base::comp(top,xl,xr);
   xl=(x-y)*s2; xr=x*s2-y*s2; top="*s right-distributivity of subtraction";
   res+=Base::comp(top,xl,xr);
   S s12=s1*s2; // workaround:
      // not clear why xl=(s1*s2)*x; gets not compiled in some cases
   xl=s12*x; xr=s1*(s2*x); top="s*s left-distributivity";
   res+=Base::comp(top,xl,xr);
   xl=x*(s1*s2); xr=(x*s1)*s2; top="s*s right distributivity";
   res+=Base::comp(top,xl,xr);

   xl=x; xl*=sUnit; xl=xl/sUnit; xr=(x*s1); xr/=s1;
   top="/s and *s compatibility";
   res+=Base::comp(top,xl,xr);

// order properties
// rather simple test of ordering consistency
```

```
   T xMax=CpmRootX::sup(x,y,z);
   T xMin=CpmRootX::inf(x,y,z);

   if ( xMin.priorTo(xMax) || xMin.equalTo(xMax)  ){
      cpmcerr<<endl<<"comparison test OK"<<endl;
      cout<<endl<<"comparison test OK"<<endl;
   }
   else{
      cpmcerr<<endl<<"comparison test failed"<<endl;
      cout<<endl<<"comparison test failed"<<endl;
      res+=1;
   }

// let us order a list of random values
   CpmArrays::Vo<T> list(nOrd);
   Z i;
   for (i=1;i<=nOrd;i++){
      list[i]=xt.ran(i);
   }
   list.order_();
   for (i=2;i<=nOrd;i++){
      if ( list[i].priorTo(list[i-1]) ){
         cpmmessage("comparison failure");
         cout<<endl<<"comparison failure";
         res+=1;
      }
   }

   if (write){
      cpmmessage("Writing path entered");
      Word filename(nameOf(xWrite));
      filename=filename.makeFileName();
      Word filename1="w-"+filename+".txt";
      OFileStream out1(filename1);
      if (!out1()) cpmerror(nameOf(xWrite),
         "test: output file cannot be opened");
      const int precWrite=12;
      out1()<<setprecision(precWrite);
      out1()<<endl<<"// I/O Test:"<<endl;
      out1()<<"// Object to be written: "<<endl;
      xWrite.prnOn(out1());
      cpmmessage("First writing done");
      if (!out1()) cpmerror(nameOf(xWrite),
         "out1 not OK before closing");
      IFileStream in(filename1);
      if (!in()) cpmerror(nameOf(xWrite),
         "test: input file cannot be opened");
      xr.scanFrom(in());
      cpmmessage("First reading done");
      if (!in()) cpmerror(nameOf(xWrite),
```

```
         "in not OK after reading and before closing");
      Word filename2="r-"+filename+".txt";
      OFileStream out2(filename2);
      if (!out2()) cpmerror(nameOf(xWrite), "test: out2 file cannot be\
       opened");
      out2()<<"// I/O Test. This is what we got back:"<<endl;
      xr.prnOn(out2());
      cpmmessage("Second writing done");
      if (!out2()) cpmerror(nameOf(xWrite),
         "out2 not OK before closing");
      xl=xWrite; top="file I/O";
      if (write>1) res+=Base::comp(top,xl,xr);
            // otherwise read write reproduction error will not be added
            // to the eror bilance, since it is not purely numerical in
            // nature.
      cpmmessage("Leaving writing path");
   }

   ostringstream ost2;
   ost2<<endl<<"  Test of "<<nameOf(xWrite)<<" errorsum= "<<res<<endl;
   Base::wrt(ost2);
   Base::errorSum=res;
   CPM_MZ
}

///// testing the interface of a commutative ring with unity ////////////
// Essentially Test_r with the test of inversion left out
// and some properties concerning relation of * and abs()
// optionally added.
// Test_rm : r modified

template <class T, class S>
class Test_rm: public TestBase<T>{
   // Testing the modified r-interface.
   // class for testing classes that
   // implement the cr-interface
   // Testing e.g. GaZ<R> is done e.g. by the following code
   // Test_rm<GaZ<R>,R> a(10);

   typedef TestBase<T> Base;

public:
   Test_rm(Z tvs=1 , Z output=0, Z write=0, Z prop=0);
   // tvs: test value size ~ complexity of test
};

template <class T, class S>
Test_rm<T,S>::Test_rm(Z tvs, Z output, Z write, Z prop):Base()
      // Tests all relevant mathematical relations and returns
      // an error sum
```

```
      // output 0 : only end of test and error sum displayed
      // output 1 : output of every test. The potentially
      //      time consuming output of the test object is suppressed
      // If write is different from 0, a write/read test to and from file
      // will be executed. Only if write > 1,the difference between
      // written and read quantity will enter the error sum. This helps to
      // asses the error behavior for the newly included multiple
      // precision faclity, where otherwise the error would be dominated
      // by the write precision which I set independently from the number
      // precision.
      // There are three relations of the norm which all are satisfied if
      // T is a C*-algebra. These are ignored for prop=0 and partially
      // or completely tested for larger values of prop.
{

   using namespace CpmSystem;
   using namespace CpmFunctions;
   using CpmRoot::nameOf;
   static const Z multOrder=4;
      // order test is made on an array of
      // multOrder*tvs elements

   Z mL=1;
   Word loc("Test_rm<T>(Z,Z,Z)");
   CPM_MA

   Z nOrd=multOrder*tvs;

   T t, v, x, y, z, xl, xr;
   Word top;

   R res=0.;
   ostringstream ost;
   ost<<endl<<" Test of class "<<t.nameOf()<<endl;
   Base::wrt(ost);

   if (output){
      cpmcerr<<endl<<
      " The distinguished objects associated with the class are: "<<endl;
   }

   T xd;
   T xt=xd.test(tvs);
   if (output){
      cpmcerr<<endl<<"test:";
      xt.toWord().prnOn(cpmcerr);
   }
   T xzero=xt.net(0);
   if (output){
      cpmcerr<<endl<<"zero:";
```

```
   xzero.toWord().prnOn(cpmcerr);
}
T xunit=xt.net(1);
if (output){
   cpmcerr<<endl<<"unit:";
   xunit.toWord().prnOn(cpmcerr);
}

v=xt;
x=xt.ran(1);
y=xt.ran(2);
z=xt.ran(3);
T xWrite=z;

if (output){
   cpmcerr<<endl<<"first ran:";  x.toWord().prnOn(cpmcerr);
   cpmcerr<<endl<<"second ran:"; y.toWord().prnOn(cpmcerr);
   cpmcerr<<endl<<"third ran:";  z.toWord().prnOn(cpmcerr);
}
T v4=x;
T v5;
T v6=y;
v6=v5=x;
xl=v6;
xr=v4;
top="copy constructor, assignement, and equality";
  res+=Base::comp(top,xl,xr);

xl=x+y; xr=y+x; top="+ commutativity";
res+=Base::comp(top,xl,xr);
xl=(x+y)+z; xr=x+(y+z); top="+ associativity";
res+=Base::comp(top,xl,xr);
xl=(x*y)*z; xr=x*(y*z); top="* associativity";
res+=Base::comp(top,xl,xr);

T xZero=x.net(0);

xl=-x; xr=xZero-x; top="sign change";
res+=Base::comp(top,xl,xr);
xl=x*(y+z); xr=x*y+x*z; top="left-distributivity";
res+=Base::comp(top,xl,xr);
xl=(x+y)*z; xr=x*z+y*z; top="right-distributivity";
res+=Base::comp(top,xl,xr);
T xy=x*y;
xl=xy.con(); xr=y.con()*x.con();
top="conjugation commutativity";
res+=Base::comp(top,xl,xr);
Z i;
if (prop>=1){
   // If the iequality is found to hold once,
```

```
      // this could be by chance
      Z nTrials=60;
      V<T> trials(nTrials);
      for (i=trials.b();i<=trials.e();++i){
         trials[i]=x.ran();
      }
      top="norm inequality";
      R errTopic=0;
      for (i=trials.b();i<trials.e();++i){
         T xa=trials[i];
         T xb=trials[i+1];
         T xab=xa*xb;
         R xLarger=xa.abs()*xb.abs();
         R xSmaller=xab.abs();
         if (xLarger<xSmaller) errTopic+=(xSmaller-xLarger);
      }
      Base::wrt("Test of "&top&": error="&cpm(errTopic));
      res+=errTopic;
   }
   if (prop>=2){
      top="conjugation norm equality";
      R x1=x.abs();
      R x2=x.con().abs();
      res+=Base::equal(top,x1,x2);
   }
   if (prop>=3){
      top="C* norm equality";
      R x1=x.abs()*x.con().abs();
      R x2=(x.con()*x).abs();
      res+=Base::equal(top,x1,x2);
   }
   x=z;
   T xu=x;
   xu=xu.net(1);
   if (output){
      cpmcerr<<endl<<"unit=";
      xu.toWord().prnOn(cpmcerr);
   }

   if (output){
      cpmcerr<<endl<<"x=";
      x.toWord().prnOn(cpmcerr);
   }
// properties with scalars
   S s;
   s=CpmRoot::testT<S>(s,tvs);
   S sZero=CpmRoot::netT<S>(s,0);
   S sUnit=CpmRoot::netT<S>(s,1);
   S sInit=sUnit+sUnit+sUnit;
   sInit+=sInit;
```

```
   Z count=0;
   S s1=CpmRoot::ranT<S>(sInit,count);
   const Z countMax=100;
   while (s1==sZero || s1==sUnit){
      s1=CpmRoot::ranT<S>(sInit,++count);
      if (count>countMax){
         cpmwarning(
         "Test_rm<T,S>: too many iterations (bad random generator of S)"
         );
         break;
      }
   }
   if (output){
      Word mes="iterations needed to get s1: "&cpmwrite(count);
      cpmcerr<<endl<<mes;
   }
   count=0;
   S s2=CpmRoot::ranT<S>(s1,++count);
   while (s2==sZero || s2==sUnit || s2==s1){
      s2=CpmRoot::ranT<S>(s1,++count);
      if (count>countMax){
         cpmwarning(
         "Test_rm<T,S>: too many iterations (bad random generator of S)"
         );
         break;
      }
   }
   if (output){
      Word mes="iterations needed to get s2: "&cpmwrite(count);
      cpmcerr<<endl<<mes;
   }
   xl=s1*(x+y); xr=s1*x+s1*y; top="*s left-distributivity";
   res+=Base::comp(top,xl,xr);
   xl=(x+y)*s2; xr=x*s2+y*s2; top="*s right-distributivity";
   res+=Base::comp(top,xl,xr);
   S s12=s1*s2; // workaround:
      // not clear why xl=(s1*s2)*x; gets not compiled in some cases
   xl=s12*x; xr=s1*(s2*x); top="s*s left-distributivity";
   res+=Base::comp(top,xl,xr);
   xl=x*(s1*s2); xr=(x*s1)*s2; top="s*s right distributivity";
   res+=Base::comp(top,xl,xr);

   xl=x; xl*=sUnit; xl=xl/sUnit; xr=(x*s1); xr/=s1;
   top="/s and *s compatibility";
   res+=Base::comp(top,xl,xr);

// order properties
// rather simple test of ordering consistency

   T xMax=CpmRootX::sup(x,y,z);
```

```
   T xMin=CpmRootX::inf(x,y,z);

   if ( xMin.priorTo(xMax) || xMin.equalTo(xMax)  ){
      cpmcerr<<endl<<"comparison test OK"<<endl;
      cout<<endl<<"comparison test OK"<<endl;
   }
   else{
      cpmcerr<<endl<<"comparison test failed"<<endl;
      cout<<endl<<"comparison test failed"<<endl;
      res+=1;
   }
// let us order a list of random values
   CpmArrays::Vo<T> list(nOrd);
   for (i=1;i<=nOrd;i++){
      list[i]=xt.ran(i);
   }
   list.order_();
   for (i=2;i<=nOrd;i++){
      if ( list[i].priorTo(list[i-1]) ){
         cpmmessage("comparison failure");
         cout<<endl<<"comparison failure";
         res+=1;
      }
   }
   if (write){
      cpmmessage("Writing path entered");
      Word filename(nameOf(xWrite));
      filename=filename.makeFileName();
      Word filename1="w-"+filename+".txt";
      OFileStream out1(filename1);
      if (!out1()) cpmerror(nameOf(xWrite),
         "test: output file cannot be opened");
      const int precWrite=12;
      out1()<<setprecision(precWrite);
      out1()<<endl<<"// I/O Test:"<<endl;
      out1()<<"// Object to be written: "<<endl;
      xWrite.prnOn(out1());
      cpmmessage("First writing done");
      if (!out1()) cpmerror(nameOf(xWrite),
         "out1 not OK before closing");
      IFileStream in(filename1);
      if (!in()) cpmerror(nameOf(xWrite),
         "test: input file cannot be opened");
      xr.scanFrom(in());
      cpmmessage("First reading done");
      if (!in()) cpmerror(nameOf(xWrite),
         "in not OK after reading and before closing");
      Word filename2="r-"+filename+".txt";
      OFileStream out2(filename2);
      if (!out2()) cpmerror(nameOf(xWrite), "test: out2 file cannot be\
```

```
       opened");
      out2()<<"// I/O Test. This is what we got back:"<<endl;
      xr.prnOn(out2());
      cpmmessage("Second writing done");
      if (!out2()) cpmerror(nameOf(xWrite),
         "out2 not OK before closing");
      xl=xWrite; top="file I/O";
      if (write>1) res+=Base::comp(top,xl,xr);
            // otherwise read write reproduction error will not be added
            // to the eror bilance, since it is not purely numerical in
            // nature.
      cpmmessage("Leaving writing path");
   }
   ostringstream ost2;
   ost2<<endl<<"  Test of "<<nameOf(xWrite)<<" errorsum= "<<res<<endl;
   Base::wrt(ost2);
   Base::errorSum=res;
   CPM_MZ
}

////////////// testing the complex-interface ////////////////////////////////
// better? testing complex types: T can be CpmRoot::C or
// std::complex(double), ...
template <class T>
class Test_c: public TestBase<T>{ // testing complex classes
   typedef TestBase<T> Base;
public:
   Test_c(Z tvs=1 , Z output=0, Z write=0);
   // tvs: test value size ~ complexity of test
};


template <class T>
Test_c<T>::Test_c(Z tvs, Z output, Z write):Base()
// The arguments have the same meaning as in Test_r. Look there.
{
   static const Z multOrder=4;
      // order test is made on an array of
      // multOrder*tvs elements

   Z nOrd=multOrder*tvs;

   T  x, y, z, xl, xr, xt;

   ostringstream ost;
   ost<<endl<<" Test of class "<<x.nameOf()<<endl;
   Base::wrt(ost);

   Word top;
   xt=x.test(tvs);
```

```
x=xt.ran(1);
y=xt.ran(2);
z=xt.ran(3);
T xWrite=z;

if (output){
   cpmcerr<<endl<<"test:"<<xt.toWord();
   cpmcerr<<endl<<"first ran:"<<x.toWord();
   cpmcerr<<endl<<"second ran:"<<y.toWord();
   cpmcerr<<endl<<"third ran:"<<z.toWord();
}

T v4=x;
T v5;
T v6=y;
v6=v5=x;
xl=v6;
xr=v4;

R res=0;
top="copy constructor, assignement, and equality";
  res+=Base::comp(top,xl,xr);
xl=x+y; xr=y+x; top="+ commutativity";
res+=Base::comp(top,xl,xr);
xl=x*y; xr=y*x; top="* commutativity";
res+=Base::comp(top,xl,xr);
xl=(x+y)+z; xr=x+(y+z); top="+ associativity";
res+=Base::comp(top,xl,xr);
xl=(x*y)*z; xr=x*(y*z); top="* associativity";
res+=Base::comp(top,xl,xr);

T xZero=net(x,0);

xl=-x; xr=xZero-x; top="sign change";
res+=Base::comp(top,xl,xr);
xl=x*(y+z); xr=x*y+x*z; top="left-distributivity";
res+=Base::comp(top,xl,xr);
xl=(x+y)*z; xr=x*z+y*z; top="right-distributivity";
res+=Base::comp(top,xl,xr);
// not clear why con(x*y) is not understood for T=Fr<Z,R>
T xTemp=x*y;
xl=con(xTemp); xr=con(y)*con(x); top="conjugation commutativity";
res+=Base::comp(top,xl,xr);

x=z;
T xu=x;

xu=net(xu,1);
T xi=inv(x);
```

```
   xl=x*xi; xr=xu; top="right inverse";
   res+=Base::comp(top,xl,xr);

   xl=xi*x; xr=xu; top="left inverse";
   res+=Base::comp(top,xl,xr);

// order properties
// rather simple test of ordering consistency

   T xMax=CpmRootX::sup(x,y,z);
   T xMin=CpmRootX::inf(x,y,z);

   if (xMin<=xMax){
      cpmcerr<<endl<<"comparison test OK"<<endl;
   }
   else{
      cpmcerr<<endl<<"comparison test failed"<<endl;
      res+=1;
   }

// let us order a list of random values
   Vo<T> list(nOrd);
   Z i;
   for (i=1;i<=nOrd;i++){
      list[i]=ran(xt,i);
   }
   list.order_();
   for (i=2;i<=nOrd;i++){
      if (list[i-1]>list[i]){
         cpmmessage("comparison failure");
         res+=1;
      }
   }
   xl=z;
   R r,phi;
   xl.toPolar(r,phi);
   xr.polar(r,phi);
   top="polar back and forth";
   res+=Base::comp(top,xl,xr);

   xl=x; xr=x.sqrt()*x.sqrt();
   top="sqrt and square";
   res+=Base::comp(top,xl,xr);

   xl=exp(x+y); xr=exp(x)*exp(y);
   top="exp addition law";
   res+=Base::comp(top,xl,xr);

   xl=x; xr=exp(ln(x));
   top="exp(ln(z))=z";
```

```
res+=Base::comp(top,xl,xr);

xl=sin(x+y); xr=sin(x)*cos(y)+cos(x)*sin(y);
top="sin addition law";
res+=Base::comp(top,xl,xr);

xl=y; xr=sin(arcsin(y));
top="sin(arcsin(z)=z";
res+=Base::comp(top,xl,xr);

xl=z; xr=cot(arccot(z));
top="cot(arccot(z)=z";
res+=Base::comp(top,xl,xr);

Z n=3;
xl=pow(cosh(x)+sinh(x),n);
xr=cosh(n*x)+sinh(n*x);
top="Moivre for cosh and sinh";
res+=Base::comp(top,xl,xr);
if (write){
   cpmmessage("Writing path entered");
   Word filename(nameOf(xWrite));
   Word filename1="w-"+filename+".txt";
   OFileStream out1(filename1);
   if (!out1()) cpmerror(nameOf(xWrite),
      "test: output file cannot be opened");
   const int precWrite=12;
   out1()<<setprecision(precWrite);
   out1()<<endl<<"// I/O Test:"<<endl;
   out1()<<"// Object to be written: "<<endl;
   out1()<<xWrite;
   cpmmessage("First writing done");
   if (!out1()) cpmerror(nameOf(xWrite),
      "out1 not OK before closing");
   IFileStream in(filename1);
   if (!in()) cpmerror(nameOf(xWrite),
      "test: input file cannot be opened");
   xr.scanFrom(in());
   cpmmessage("First reading done");
   if (!in()) cpmerror(nameOf(xWrite),
      "in not OK after reading and before closing");
   Word filename2="r-"+filename+".txt";
   OFileStream out2(filename2);
   if (!out2()) cpmerror(nameOf(xWrite),
      "test: out2 file cannot be opened");
   out2()<<"// I/O Test. This is what we got back:"<<endl;
   out2()<<xr;
   cpmmessage("Second writing done");
   if (!out2()) cpmerror(CpmRoot::nameOf(xWrite),
      "out2 not OK before closing");
```

```
      xl=xWrite; top="file I/O";
      if (write>1) res+=Base::comp(top,xl,xr);
            // otherwise read write reproduction error will not be added
            // to the eror bilance, since it is not purely numerical in
            // nature.
      cpmmessage("Leaving writing path");
   }
   Base::errorSum=res;
   ostringstream ost2;
   ost2<<endl<<"  Test of "<<nameOf(xWrite)<<" errorsum= "<<res<<endl;
   Base::wrt(ost2);
}

// class for testing consistent polymorphic behavior
template <class T, class Td, class Container> // assumes that Td is
// derived from T
//
class TestOfPolymorphism{ // testing consistent polymorphic behavior
   R eps_;
   public:
   TestOfPolymorphism(T const& t, const Td& td)
   {
       using namespace CpmSystem;
       Word loc("TestOfPolymorphism(T const& t, const Td& td)");
       Z tv=16;
       Container ct;
       T tMem=t;
       Td tdMem=td;
       ct.set(t);
       T t1=ct();
       cpmassert(t1==tMem,loc);
       ct.set(td);
       Td td1=ct();
       eps_=td1.dis(tdMem);
       cpmmessage("TestOfPolymorphism() successfully done");
       cout<<endl<<"TestOfPolymorphism() done with eps = "<<eps_<<endl;
    }
    R val()const{ return eps_;}
};

////////////////////////////// Test_Vp ////////////////////////////////
// testing polymorphicity and value interface of Vp<>

template <class T0, class T1, class T2, class T3>
class Test_Vp: public TestBase<T0>{// testing behavior of Vp<>

   typedef TestBase<T0> Base;

public:
   Test_Vp(Z tvs=1 , Z output=1);
```

```
};

template <class T0, class T1, class T2, class T3 >
Test_Vp<T0,T1,T2,T3>::Test_Vp(Z tvs , Z output):Base()
      // Tests all relations concerning polymorphic array Vp<> and
      // returns an error sum
{
   using namespace CpmRoot;
   using namespace CpmRootX;
   using namespace CpmArrays;

   Vp<T0> v0;

   ostringstream ost;
   ost<<endl<<" Test of class "<<v0.nameOf()<<endl;
   Base::wrt(ost);

   T3 t3;
   t3=t3.test(tvs);
   T3 t3_1=t3.ran(1);
   T3 t3_2=t3.ran(2);
   T3 t3_3=t3.ran(3);
   T3 t3_4=t3.ran(4);
   T3 t3_5=t3.ran(5);
   T3 t3_6=t3.ran(6);

   T0 t0_1=t3_1.toClnBase();
   T0 t0_2=t3_2.toClnBase();
   T0 t0_3=t3_3.toClnBase();
   T0 t0_4=t3_4.toClnBase();
   T0 t0_5=t3_5.toClnBase();
   T0 t0_6=t3_6.toClnBase();

   T1 t1_1=T1(t0_1);
   T1 t1_2=T1(t0_2);
   T1 t1_3=T1(t0_3);
   T1 t1_4=T1(t0_4);
   T1 t1_5=T1(t0_5);
   T1 t1_6=T1(t0_6);

   T2 t2_1=T2(t0_1);
   T2 t2_2=T2(t0_2);
   T2 t2_3=T2(t0_3);
   T2 t2_4=T2(t0_4);
   T2 t2_5=T2(t0_5);
   T2 t2_6=T2(t0_6);

   Z nvp=12;
   Vp<T0> vp(nvp);
   V<Word> wp(nvp);
```

```
   vp[1]=t0_1;
   vp[2]=t1_2;
   vp[3]=t2_3;
   vp[4]=t3_4;
   vp[5]=t0_5;
   vp[6]=t1_6;
   vp[7]=t2_1;
   vp[8]=t3_2;
   vp[9]=t0_3;
   vp[10]=t1_4;
   vp[11]=t2_5;
   vp[12]=t3_6;

   wp[1]=t0_1.nameOf();
   wp[2]=t1_2.nameOf();
   wp[3]=t2_3.nameOf();
   wp[4]=t3_4.nameOf();
   wp[5]=t0_5.nameOf();
   wp[6]=t1_6.nameOf();
   wp[7]=t2_1.nameOf();
   wp[8]=t3_2.nameOf();
   wp[9]=t0_3.nameOf();
   wp[10]=t1_4.nameOf();
   wp[11]=t2_5.nameOf();
   wp[12]=t3_6.nameOf();

// complicated process of creating an array Vp<T0> zp holding the same
// values as vp
   Vp<T0> xp; // default element
   xp=vp; // assignment to default element
   Vp<T0> yp(xp); // copy construction
   Vp<T0> zp(5001);
   zp.set(t0_6); // zp something different
   zp=yp; // assignment to something non-trivial

   R errSum=0;
   Z i;
   for (i=xp.b();i<=xp.e();++i){ // testing whether finally zp holds the
   // right objects. Here only the name gets tested
      Word wpi=zp(i).nameOf();
      if (output) cout<<endl<<"wpi="<<wpi<<" , wp[i]="<<wp[i];
      errSum+=wp[i].dis(wpi);
   }

   Base::errorSum=errSum;
   ostringstream ost2;
   ost2<<endl<<" Test of class "<<v0.nameOf()<<" done."
   <<endl<<" Errorsum= "<<errSum;
   Base::wrt(ost2);
}
```

```
////////////////////////////// Test_F ////////////////////////////////
// Testing the chaining operation & of F<>

template <class T1, class T2, class T3, class T4>
class Test_F: public TestBase<T1>{ // Testing the chaining operation of F
   typedef TestBase<T1> Base;
public:
   Test_F(Z tvs=1 );
   // tvs: test value size ~ complexity of test
};

template <class T1, class T2, class T3, class T4 >
Test_F<T1,T2,T3,T4>::Test_F(Z tvs):Base()
// tvs: test value size ~ complexity of test
{
   using namespace CpmRoot;
   using namespace CpmRootX;
   using namespace CpmFunctions;

   Fr<T1,T2> fa;
   Fr<T2,T3> fb;
   Fr<T3,T4> fc;

   Word mes(" Test of classes ");
   mes&=(fa.nameOf()&", "&fb.nameOf()&", and "&fc.nameOf());
   Base::wrt(mes);

   fa=fa.test(tvs);
   fb=fb.test(tvs);
   fc=fc.test(tvs);

   F<T1,T2> fa_1=fa.ran(1);
   F<T1,T2> fa_2=fa.ran(2);

   F<T2,T3> fb_1=fb.ran(3);
   F<T2,T3> fb_2=fb.ran(4);

   F<T3,T4> fc_1=fc.ran(5);
   F<T3,T4> fc_2=fc.ran(6);

   F<T1,T3> fab=fa_1&fb_2;
   F<T2,T4> fbc=fb_2&fc_1;

   Fr<T1,T4> fab_c=fab&fc_1;
   Fr<T1,T4> fa_bc=fa_1&fbc;

   Fr<T1,T4> fake=fa_2&fbc;

   R errSum=0;
```

```
   errSum+=fab_c.dis(fa_bc);
   R dis=fab_c.dis(fake); // should not be small
      // this detects programming bugs that let dis always return 0.
   R tiny=1e-6;
   if (dis<tiny) errSum+=1;

   Base::errorSum=errSum;
   mes&=(" done.\n Errorsum="&cpm(Base::errorSum));
   Base::wrt(mes);
}

////////////////////////////// Test_set ////////////////////////////

template <class S, class T> // S is the class of sets of type T;
// typically S=Sr<T>
class Test_set: public TestBase<S>{ // testing behavior of sets

   typedef TestBase<S> Base;

public:
   Test_set(Z tvs=1 , Z output=1, Z write=0);
};

template <class S, class T>
Test_set<S,T>::Test_set(Z tvs , Z output, Z write):Base()
   // Tests all relations concerning the value interface and returns
   // an error sum
{
   using namespace CpmRoot;
   using namespace CpmRootX;
   using namespace CpmArrays;

   const R dSmall=1e-4;
   const R dTiny=1e-12;

   T t;
   S s,s1,s2,s3,s4;
   S lhs, rhs;

   ostringstream ost;
   ost<<endl<<" Test of class "<<s.nameOf()<<endl;
   Base::wrt(ost);

   s=s.test(tvs); // test object, not performing a test.
   cpmcerr<<" s = "<<s<<endl;
   t=testT<T>(t,tvs);
   cpmcerr<<" t = "<<t<<endl;
   s1=s.ran(1);
   cpmcerr<<" s1 = "<<s1<<endl;
   s2=s.ran(2);
```

```
s3=s.ran(3);
s4=s.ran(4);
s1=s1|s4;
s2=s2|s4;
s3=s3|s4;
   // this achieves that not all sections become trivial
S xWrite=s3;
if (output){
   cpmcerr<<endl<<"test:";          s.prnOn(cpmcerr);
   cpmcerr<<endl<<"first ran:";  s1.prnOn(cpmcerr);
   cpmcerr<<endl<<"second ran:"; s2.prnOn(cpmcerr);
   cpmcerr<<endl<<"third ran:";  s3.prnOn(cpmcerr);
}

R res=0.;
Word top;

idem1(lhs,rhs,s1);
   // test template function defined in cpmsr.h
top="idem1";
res+=Base::comp(top,lhs,rhs);

idem2(lhs,rhs,s2);
top="idem2";
res+=Base::comp(top,lhs,rhs);

comm1(lhs,rhs,s1,s2);
top="comm1";
res+=Base::comp(top,lhs,rhs);

comm2(lhs,rhs,s1,s2);
top="comm2";
res+=Base::comp(top,lhs,rhs);

assoc1(lhs,rhs,s1,s2,s3);
top="assoc1";
res+=Base::comp(top,lhs,rhs);

assoc2(lhs,rhs,s1,s2,s3);
top="assoc2";
res+=Base::comp(top,lhs,rhs);

distr1(lhs,rhs,s1,s2,s3);
top="distr1";
res+=Base::comp(top,lhs,rhs);

distr2(lhs,rhs,s1,s2,s3);
top="distr2";
res+=Base::comp(top,lhs,rhs);
```

```
diffdistr1(lhs,rhs,s1,s2,s3);
top="diffdistr1";
res+=Base::comp(top,lhs,rhs);

diffdistr2(lhs,rhs,s1,s2,s3);
top="diffdistr2";
res+=Base::comp(top,lhs,rhs);

S xl,xr;

if (write){
   cpmmessage("Writing path entered");
   Word filename(nameOf(xWrite));
   filename=filename.makeFileName();
   Word filename1="w-"+filename+".txt";
   OFileStream out1(filename1);
   if (!out1()) cpmerror(nameOf(xWrite),
      "test: output file cannot be opened");
   const int precWrite=12;
   out1()<<std::setprecision(precWrite);
   out1()<<endl<<"// I/O Test:"<<endl;
   out1()<<"// Object to be written: "<<endl;
   xWrite.prnOn(out1());
   cpmmessage("First writing done");
   if (!out1()) cpmerror(nameOf(xWrite),
      "out1 not OK before closing");
   IFileStream in(filename1);
   if (!in()) cpmerror(nameOf(xWrite),
      "test: input file cannot be opened");
   xr.scanFrom(in());
   cpmmessage("First reading done");
   if (!in()) cpmerror(nameOf(xWrite),
      "in not OK after reading and before closing");
   Word filename2="r-"+filename+".txt";
   OFileStream out2(filename2);
   if (!out2()) cpmerror(nameOf(xWrite),
      "test: out2 file cannot be opened");
   out2()<<"// I/O Test. This is what we got back:"<<endl;
   xr.prnOn(out2());
   cpmmessage("Second writing done");
   if (!out2()) cpmerror(nameOf(xWrite),"out2 not OK before closing");
   xl=xWrite; top="file I/O";
   res+=Base::comp(top,xl,xr);
   S xlr=static_cast<S>(xl|xr);
   S xc=xlr.condense();
   cout<<endl<<"condensed union";
   xc.prnOn(cout);
   cpmmessage("Leaving writing path");
}
Base::errorSum=res;
```

```
   ostringstream ost2;
   ost2<<endl<<" Test of class "<<s.nameOf()<<" done."
   <<endl<<" Errorsum= "<<res;
   Base::wrt(ost2);

}

//////////////////// class Test_logic<> ////////////////////////////////
// Test of classes modeling propositional logic
// So far implemented: cpmlogic1.h, classes PropLog<>, PropLogic

template <class T>
class Test_logic: public TestBase<T>{//test models of propositional logic

   typedef TestBase<T> Base;

public:
   Test_logic(Z tvs=1 , Z output=1, Z write=0);

};

template <class T>
Test_logic<T>::Test_logic(Z n, Z output, Z write):Base()
   // Tests all relevant mathematical relations and returns
   // an error sum
   // output 0 : only end of test and error sum displayed
   // output 1 : output of every test. The potentially
   //      time consuming output of the test object is suppressed
   // output 2: only the first component of the test object is
   // displayed
   // output 3: the test object is displayed in full detail
{
   Z i;
   if (n<3) n=3; // we need as many variables to make indicative examples

   V<T> p=T::list(n);
   T p1=p[1];
   for (i=2;i<=n;i++) p1=p1.et(p[i]);
   p1=p1.non();
   T p2=p[1].non();
   for (i=2;i<=n;i++) p2=p2.vel(p[i].non());
   bool b=p1.iff(p2).taut();
   cpmmessage("b="&B(b).toWord()&"; true is correct");

// example to the commutativity of the diagram shown in
// cpmlogic1.h for class PropLog<X>

   T q1=p[1].non().et(p[2]).vel(p[3]);
   T q2=p[1].vel(p[2]);
   B b1(q1.eval());
```

```
   B b2(q2.eval());

   T q12a=q1.et(q2);
   B b12a=b1.et(b2);
   B b12a_(q12a.eval());
   B resa(b12a==b12a_);
   cpmmessage("b12a==b12a_ ?"&resa.toWord()&"; true is correct");

   T q12o=q1.vel(q2);
   B b12o=b1.vel(b2);
   B b12o_(q12o.eval());
   B reso(b12o==b12o_);
   cpmmessage("b12o==b12o_ ?"&reso.toWord()&"; true is correct");

   T q1n=q1.non();
   B b1n=b1.non();
   B b1n_(q1n.eval());
   B resn(b1n==b1n_);
   cpmmessage("b1n==b1n_ ?"&resn.toWord()&"; true is correct");

/************** takes too long so far ***********
   Z nc=3;
   V<T> q=T::list(nc);
   T xorq=T::xorV(q);
   V<T> known(2);
   known[1]=xorq;
   known[2]=q[nc];
   V<T> aux=T::list(nc-1);
   for (i=1;i<nc;i++) aux[i]=q[i].non();
   T hyp=T::andV(aux);
   B resx(hyp.implied(known));
   cpmmessage("resx ? "&resx.toWord()&"; true is correct");
*/
}

} // namespace

#endif
```

36 cpmtypes.h

```
//? cpmtypes.h
//? Status of work 2023-10-20.
//?
//?  ...


#ifndef CPM_TYPES_H_
#define CPM_TYPES_H_
/*

   Description: Declares basic infrastructure functions, macros,
      and ---most importantly--- the class versions B, Z1, and R1
      of bool, Z, and R.
*/
#include <cpmsystem.h> // includes cpmwords.h and
   // thus also cpmnumbers.h
#include <cpmmacros.h>
#include <cpmx.h>

#include <cmath>

namespace CpmRootX{ // CpmRoot extended
      // A statement
      //    using namespace CpmRoot;
      // makes life much easier when working with C+-. It is therefore
      // desirable that this does not inject to many names in the users
      // scope. Thus we separate the 'true root' from an 'extended root'
      // that hosts also names such as 'order', 'inf', and 'sup' which
      // are more likely to interfere with a user's names.

   using namespace CpmStd;

   using CpmRoot::Z;
   using CpmRoot::N;
   using CpmRoot::R;
   using CpmRoot::L;
   using CpmRoot::Word;

   const R hugeNumber=1e64;
   const R tinyNumber=R(1.)/hugeNumber;

#define cpmcheck(X)\
 if (!CpmRoot::isVal(X)){\
   ostringstream ost;\
   ost<<" bad value: "<<#X "="<< X;\
   cpmmessage(Word(ost.str()));}
   // checks whether a quantity of type Z,N,R,Rh is valid
```

```
   // (and stops the program) and gives a message

#define cpmdebug(X) cpmcerr<<endl<<"C+- debug: "<<#X "="<< X <<endl
   // useful for placing temporary debugging statements into one's code.
   // From Bruce Eckel: Thinking in C++, Prentice Hall 1995, p. 325
   // typical usage:
   //
   // R_Matrix y=....;
   // cpmdebug(y);          // ';' needed !
   // The part 'debug:' helps to find the debug info in the potentially
   // very large file cpmcerr.txt

#define cpmnote(X) cpmdata<<endl<<"C+- note: "<<#X "="<< X <<endl
   // Works as cpmdebug, but but writes on cpmdata. More suitable
   // for data which are of temporary interest, so that it would
   // not pay to provide a more elaborate documentation for them.
   // Notice that cpmdata is more easily surveyed than cpmcerr.

#define cpmis(X) std::string(#X)
   // 'is' stands for 'identifier to string'.
   // This can't be written as a C or C++ function. Such a function
   // always looks on the value of a variable, wheras the present
   // construct looks on the name of the variable and returns it as a
   // quantity of type string.
   // Consider:
   //    R x=3.14;
   //    cout<<cpmis(x);
   // This will write 'x' and not '3.14'

#define cpmiw(X) Word(#X)
   // much alike previous macro

typedef void (*fpWordToVoid)(const Word&);
   // function pointer for function Word to void as a type

typedef void (*fpVoidToVoid)(void);
   // function pointer for function void to void as a type

Z sig(Z i);
R sig(R i);
   // added 03-11-05, 0 for i==0, -1 for i<0, +1 for i>0

R krn(Z i, Z j);
   // Kronecker; defined as 1. for i==j, 0. else
   // added 2004-08-26

Z parity(Z k);
/*{
   if (k%2) return -1; else return 1;
}*/
```

```
Z fac(Z n);
//: factorial
// fac(n) = 1x2x3x...x(n-1)xn , n>0 (no other values allowed here!)
// Most basic implementation.

Z pow(Z const& x, Z p);

bool isPowOfTwo(Z const& x);

Z nextPowOfTwo(Z const& x);
// We return the smallest y \in P2 such that *this <= y
// Here P2 := {0,2,4,8,16,32,....}

Z nextLog2(Z const& x);
// We return the conventional dual logarithm of
// nextPowOfTwo() ( which then is always \in N)

// Auxiliar functions.

void copyTextFile(ifstream& from, ofstream& to);
   // here, the streams from and to have to exist already

void copyTextFile(const Word& nameOfSource, const Word& nameOfCopy );
   // a file of the name nameOfSource has to exist and the copy with the
   // name indicated by the second argument will exist after the function
   // body has been executed

string toString(ifstream& from);
   // copies the content of a text file into a standard library string
   // and thus opens up rich manipulation capabilities mainly by writing
   // these to stringstreams which finally can easily be filed back. In
   // this manner e.g. concatenation of text files becomes nearly trivial

// addition 97-6-5, notice that names min and max clash with
// macro names of Windows. Thus we choose presumably save naming.
// Version with const T& arguments is not always working. Sometimes one
// gets unability to resolve function overload

////////////////////////// sup ////////////////////////////////////////
   template <class T>
   T sup(T t1, T t2)
      // sup=supremum=maximum
   {
      if (t1<t2) return t2; else return t1;
   }

   template <class T>
   T sup(T t1, T t2, T t3)
   {
```

```
      T temp;
      if (t1<t2) temp=t2; else temp=t1;
      if (temp<t3) return t3; else return temp;
   }

   template <class T>
   T sup(T t1, T t2, T t3, T t4)
   {
      T temp1=sup<T>(t1,t2);
      T temp2=sup<T>(t3,t4);
      return sup<T>(temp1,temp2);
   }

////////////////////////// inf ////////////////////////////////////////

   template <class T>
   T inf(T t1, T t2)
      // inf=infimum=minimum
   {
      if (t1<t2) return t1; else return t2;
   }

   template <class T>
   T inf(T t1, T t2, T t3)
      // inf=infimum=minimum
   {
      T temp;
      if (t1<t2) temp=t1; else temp=t2;
      if (temp<t3) return temp; else return t3;
   }

   template <class T>
   T inf(T t1, T t2, T t3, T t4)
   {
      T temp1=inf<T>(t1,t2);
      T temp2=inf<T>(t3,t4);
      return inf<T>(temp1,temp2);
   }

   ////////////////////////// swap ////////////////////////////////////////

   template <class T>
   void swap(T& t1, T& t2)
   // After function call t1 has the value that t2 had before
   // and t2 has the value that t1 had before.
   // If T is a class (not built-in type) we assume that T defines
   // operator = and a copy constructor.
   {
      T temp=t2;
      t2=t1;
```

```
      t1=temp;
   }

////////////////////////// order ////////////////////////////////////////

   template <class T>
   void order(T& t1, T& t2)
      // after function call we have t1<=t2
      // If T is a class (not built-in type) we assume that T defines
      // a '<=' operation such that always at least one of t1<=t2 ,
      // t2<=t1 is valid. Also operator = and a copy constructor have to
      // be defined.
   {

      if (t1<=t2){
         return;
      }
      else{
         T temp=t2;
         t2=t1;
         t1=temp;
      }
   }

// Class equivalents of R , Z, bool.
// These can conveniently be used for class data members
// since they get initialized properly.
using CpmArrays::X2;

#if !defined(CPM_MP)
// Once I introduced an implementalion of floatingpoint reals and of
// integers as classes. This was the only way to provide for real and for
// integer data members a automatic initialization by means of the
// defined default constructor. With C++11 one can data members
// define as e.g. R x{0.}; for what was R1 x; before.
// We can now avoid the ugly mixture of R and R1 (and Z and Z1).
// Particularly inconvenient was the maintenance of R1, Z1 for the
// the multiprecision options. So the intent is to avoid any use of
// R1 and Z1 in regular C+- code. Nevertheless I conserve the definition
// of R1 and Z1 to enable experimentation, especially with the discrete
// lattice underlying the floating point numbers.

////////////////////////////// class R1 ////////////////////////////

class R1{  // real numbers as a class
   // since there is no need for 1-tupels, we
   // define this a class endowed with infratructure similar than R2,
   // R3, ... but with arithmetics gained from the non-class type R via
   // automatic conversion (in one direction only to avoid ambiguities)
   // Most functions use R-arguments and not R1-ones since R1's have to
```

```
   // be converted to R anyway in most situations.
public:
   static const R1 min;
   static const R1 max;
   static const R1 eps0;
   static const R1 eps1;
   static const R1 logmax;
   static const R1 log10max;
   static const R1 pi;
   static const R1 piInv;

   R x1;
   typedef R1 Type;
// constructors
   R1():x1(0){}
   explicit R1(R a):x1(a){} // no automatic conversion wanted
   operator R()const{ return x1;}
      // auto-conversion to R is wanted here
   R operator()(void)const{ return x1;}
      // natural output of the intrinsic value
   R1& operator=(R a){ x1=a; return *this;}
   R1& operator+=(R a){ x1+=a; return *this;}
   R1& operator-=(R a){ x1-=a; return *this;}
   R1& operator*=(R a){ x1*=a; return *this;}
   R1& operator/=(R a){ x1*=R(R1(a).inv()); return *this;}
   R1 operator - ()const{ return R1(-x1);}
 //  R operator*(R const& a)const{ return x1*a;}
 //  friend R operator*(R const& a, R1 const& b){ return a*b.x1;}

   R abs()const{ if (x1<0) return -x1; return x1;}
   R dis(const R1&)const;
   X2<R,R1> polDec()const;
      // polar decomposition: First component (res.first if res is the
      // return value) is the absolute value r and the second component
      // is sign(x1)
   CPM_IO
 //  CPM_ORDER
   friend bool operator == (R1 r1, R1 r2){ return r1.x1 == r2.x1;}
   friend bool operator != (R1 r1, R1 r2){ return r1.x1 != r2.x1;}
   friend bool operator >= (R1 r1, R1 r2){ return r1.x1 >= r2.x1;}
   friend bool operator <= (R1 r1, R1 r2){ return r1.x1 <= r2.x1;}
   friend bool operator > (R1 r1, R1 r2){ return r1.x1 > r2.x1;}
   friend bool operator < (R1 r1, R1 r2){ return r1.x1 < r2.x1;}
   Z com(R1 r)const; // Compare function as tool, since R itself
      // does not work with com.
   Z dim()const{ return 1;}
   R1 inf(R1 a)const{ return x1<=a.x1 ? R1(x1) : R1(a.x1);}
   R1 sup(R1 a)const{ return x1>=a.x1 ? R1(x1) : R1(a.x1);}
   R& operator[](Z i){ i; return x1;}  // defining components in analogy
      // to what is done for R2, R3, R4
```

```
   const R& operator[](Z i)const{ i; return x1;}
   void set(R t){ x1=t;}
   void makePos_(){ if (x1<0) x1=-x1;}
      //: make positive
   Word nameOf()const{ return "R1";}
   Word toWord()const{ return Word::write(x1);}
   bool isVal()const{ return CpmRoot::isVal(x1);}
// functions referring to R and R1 as a discrete sets ('lattice')
   R1& next_();
      // next (in lattice)
   R1& prv_();
      // previous (in lattice)
   Z dz(R a)const;
      // distance on the lattice of represented numbers
      // counts the points exactly and thus is very slow for
      // typical pairs of numbers. If two numbers arrose from
      // getting the same 'mathematical number' by different
      // numerical expressions, the values should be close enough
      // so that dz should be small (approximately the number of
      // operations in the algorithm ???). Function dr is a much
      // faster approximate representation of this discrete distance.
   R eps(Z dir=1)const;
      // distance to the next lattice point in positive direction
      // for dir>=0 and in negative direction for dir<0.
      // In this latter case also the result is returned as negative
      // Thus x+x.eps(1) is the next lattice neighbor of x in
      // positive direction and x+x.eps(-1) is that in negative
      // direction
   R dr(R a)const; // const?
      // approximate number of lattice intervals between a and *this
      // (always >=0)
   Z run(R xFinal, ostream& str, Z writePeriod=1000)const; //const?
      // runs from *this to xFinal in steps over all represented
      // numbers and prints out ( to stream o) each n'th of
      // them with n given by writePeriod
      // Returns the number of printed numbers.
// safe functions
   R1 inv()const;
      // inverse, returns R1::max.x1 for x1==0
   R exp()const;
      // Returns R1::max.x1 for all x1 that would normally give infinity.
      //
      // Example of usage: instead of exp(x) write R1(x).exp() if it is
      // important to get a valid, non-exceptionl, result
   R log()const;
   R log10()const;
      // these four functions are defined on whole R1.
      // logs are log of absolute value, a combination which
      // is common in mathematics.
```

```
   R sign()const{ return (x1> 0. ? 1. : (x1==0. ? 0. : -1.));}
         //: sign (English),
         // akin to Latin signum

   R sign(R b)const{ return x1>0 ? x1*R1(b).sign() : -x1*R1(b).sign();}
        // 'takes the modulus from *this and the sign from b'
};

////////////////////////////// class Z1 ////////////////////////////////

class Z1{ // integer numbers as a class
   // tuples of 1 elements, for formal
   // symmetry with others
public:
   Z x1;
   static const Z1 min;
   static const Z1 max;
   typedef Z1 Type;
// constructors
   Z1():x1(0){}

   explicit Z1(long int a):x1(a){} // no automatic conversion wanted
   explicit Z1(int a):x1(a){} // no automatic conversion wanted
   explicit Z1(R r):x1(cpmrnd(r)){} // no automatic conversion wanted
   operator Z()const{ return x1;}
   R toR()const{ return R(x1);}
   Z1& operator=(Z a){ x1=a; return *this;}
  // Z1& operator=(bool b){ if (b) x1=1; else x1=0; return *this;}
   Z1& operator+=(Z a){ x1+=a; return *this;}
   Z1& operator-=(Z a){ x1-=a; return *this;}
   Z1& operator*=(Z a){ x1*=a; return *this;}

   Z1 pow(Z p)const;
      // Z1(x1^p), cpmassert p>=0

   Z abs()const{ if (x1<0) return -x1; return x1;}
   R dis(const Z1&)const;
   X2<Z,Z1> polDec()const;
      // polar decomposition: First component (res.first if res is the
      // return value) is the absolute value r and the second component
      // is sign(x1)
   CPM_IO
   CPM_ORDER
   Z dim()const{ return 1;}
   Z1 inf(Z1 a)const{ return x1<=a.x1 ? Z1(x1) : Z1(a.x1);}
   Z1 sup(Z1 a)const{ return x1>=a.x1 ? Z1(x1) : Z1(a.x1);}
   Z& operator[](Z i){ i; return x1;}  // defining components in analogy
   // to what is done for Z2
   const Z& operator[](Z i)const{i; return x1;}
   Word nameOf()const{ return "Z1";}
```

```
   Word toWord()const{ return Word::write(x1);}
   bool isVal()const{ return CpmRoot::isVal(x1);}
   Z parity()const{ return CpmRootX::parity(x1);}

   bool isPowOfTwo()const;
      // Let N:={1,2,3,4,...} (no zero!) and
      // P2:={ 2,4,8,16,...}
      // Then we return true iff *this belongs to P2

   Z1 nextPowOfTwo()const;
      // We return the smallest y \in P2 such that *this <= y

   Z1 nextLog2()const;
      // We return the conventional dual logarithm of
      // nextPowOfTwo() ( which then is always \in N)
};
#endif

} //namespace CpmRootX

namespace CpmRoot{
// All similar declarations are in cpmword.h There, the present
// type is not yet defined. So we have to defer the declaration till
// here. That this works, is a triumph of C++'s templates.

   template<>
   class Name<CpmRootX::fpVoidToVoid>{ // also this type needs a name
   public:
      Name(){}
      Word operator()(CpmRootX::fpVoidToVoid const& t)const
      { t; return Word("fpVoidToVoid");}
   };

   template<>
   class Name<CpmRootX::fpWordToVoid>{ // also this type needs a name
   public:
      Name(){}
      Word operator()(CpmRootX::fpWordToVoid const& t)const
      { t; return Word("fpWordToVoid");}
   };

}// namespace

// new names
#define cpminf       CpmRootX::inf
#define cpmsup       CpmRootX::sup

#define cpmord      CpmRootX::order
#define cpmswp      CpmRootX::swap
#define cpmtin      CpmRootX::tinyNumber
```

```
#define cpmhug      CpmRootX::hugeNumber

#endif
```

37 cpmtypes.cpp

```
//? cpmtypes.cpp
//? Status of work 2023-10-20.
//?
//?  ...


#include <stdlib.h>
#include <sstream>

#include <cpmtypes.h>

using namespace CpmSystem;
using namespace CpmRoot;
using CpmArrays::X2;

////////////////////////////////////////////////////////////////

void CpmRootX::copyTextFile(ifstream& from, ofstream& to)
{
   char ch;
   while (from.get(ch)) to.put(ch);
   if (!from.eof() || to.bad())
   cpmerror("copyTextFile(const ifstream&, ofstream& to) failed");
     // Stroustrup Section 10.5 p. 351 top of page
}

void CpmRootX::copyTextFile(const Word& nameOfSource,
   const Word& nameOfCopy)
{
   Z mL=1;
   Word loc("copying "&nameOfSource&" to "&nameOfCopy);
   cpmassert(nameOfSource!="cpmcerr.txt",loc);
      // copying "cpmcerr.txt" caused problems, which are hard to fully
      // understand
   cpmmessage(mL,loc&" started");
   IFileStream from(nameOfSource);
   OFileStream to(nameOfCopy);
   CpmRootX::copyTextFile(from(),to());
   cpmmessage(mL,loc&" done");
}

string CpmRootX::toString(ifstream& from)
{
   if (!from) cpmerror("toString(): cannot open input file");
   ostringstream to;
   if (!to) cpmerror("toString(): cannot create output file stream");
   char ch;
```

```
   while(from.get(ch)) to.put(ch);
   if (!from.eof() || !to) cpmerror("failure in toString");
   return to.str();
}

////////////////////////// class Z1 ////////////////////////////////

const Z maxZ(std::numeric_limits<Z>::max());
const Z minZ(std::numeric_limits<Z>::min());

Z CpmRootX::sig(Z i)
{
   if (i==0) return Z(0);
   else if (i>0) return Z(1);
   else return Z(-1);
}

R CpmRootX::sig(R x)
{
   if (x==R(0.)) return R(0.);
   else if (x>R(0.)) return R(1.);
   else return R(-1.);
}

R CpmRootX::krn(Z i, Z j)
{
   return i==j ? R(1.) : R(0.);
}

Z CpmRootX::parity(Z k)
{
   return k%2==0 ? Z(1) : Z(-1);
}

Z CpmRootX::fac(Z n)
{
   cpmassert(n>0,"Z fac(Z)");
   Z res=1;
   for (Z i=1; i<=n; i++) res*=i;
   return res;
}

Z CpmRootX::pow(Z const& x, Z p)
{
   cpmassert(p>=0,"Z powerZ(Z,Z)");
   if (p==0) return Z(0);
   else if (p==1) return x;
   else if (p==2) return x*x;
   else{
      Z res=x;
```

```
      Z nMult=p-1;
      for (Z i=1;i<=nMult;i++) res*=x;
      return res;
   }
}

bool CpmRootX::isPowOfTwo(Z const& x)
{
   Z fac{2};
   if (x<fac) return false;
   Z f=fac;
   while (f<x) f*=fac;
   return f==x;
}

Z CpmRootX::nextPowOfTwo(Z const& x)
{
   if (x<2) return Z(2);
   Z f=2;
   while (f<x) f*=2;
   return Z(f);
}

Z CpmRootX::nextLog2(Z const& x)
{
   Z res{1};
   Z f{2};
   while (f<x){ // after the loop f>=x1
      res++;
      f*=2; // guarantees that f grows over all bounderies
   }
   return res;
}

#if !defined(CPM_MP) && !defined(CPM_QUAD)
////////////////////////// class R1 ////////////////////////////////

using CpmRootX::R1;
using CpmRootX::Z1;

const R1 R1::max(std::numeric_limits<R>::max());
const R1 R1::eps1(std::numeric_limits<R>::epsilon());
const R1 R1::eps0(std::numeric_limits<R>::min());

const R1 R1::min(-R1::max);
const R1 R1::logmax(::log(R1::max));
const R1 R1::log10max(::log10(R1::max));
const R1 R1::pi(cpmpi);
const R1 R1::piInv(R(1)/cpmpi);
```

```
bool R1::prnOn(ostream& out)const
{
   return CpmRoot::write(x1,out);
}

bool R1::scanFrom(istream& in)
{
   return CpmRoot::read(x1,in);
}

Z R1::com(R1 a)const
{
   if (x1<a.x1) return 1;
   if (x1>a.x1) return -1;
   return 0;
}

X2<R,R1 > R1::polDec()const
{
   if (x1>=0) return X2<R,R1 >(x1,R1(1.));
   else return X2<R,R1 >(-x1,R1(-1.));
}

R R1::dis(const R1& x)const
{
   return CpmRoot::dis(x1,x.x1);
}

// functions referring to R as a discrete 'lattice'

R1& R1::next_(void)
{
   R eps_=eps0.x1;
   R xMem=x1;
   while (x1==xMem){
      x1+=eps_;
      eps_*=2;
   }
   return *this;
}

R1& R1::prv_(void)
{
   R eps_=eps0.x1;
   R xMem=x1;
   while (x1==xMem){
      x1-=eps_;
      eps_*=2;
   }
   return *this;
```

```
}

namespace{

   void step_p(R& x, R& e)
   {
      R xMem=x;
      Label:
      x+=e;
      if (x==xMem){
         e*=2;
         goto Label;
      } // thus we end up with x!=xMem
   }

   void step_m(R& x, R& e)
   {
      R xMem=x;
      Label:
      x-=e;
      if (x==xMem){
         e*=2;
         goto Label;
      } // thus we end up with x!=xMem
   }
}

Z R1::dz(R a)const
{
   Z res=0;
   R e=R1::eps0.x1;
   if (x1>=a){
      while (x1>a){
         step_p(a,e);
         ++res;
      }
   }
   else{
      while (x1<a){
         step_m(a,e);
         --res;
      }
   }
   return res;
}

R R1::eps(Z dir)const
{
   R res= (x1>=1 || x1<=-1) ? eps1.x1 : eps0.x1;
   if (dir<0) res*=-1;
```

```
   R xc=x1;
   Label:
   xc+=res;
   if (xc==x1){
      res*=2;
      goto Label;
   }
   return res;
}

R R1::dr(R a)const
{
   if (a==x1) return 0;
   R dis=x1-a;
   if (dis<0) dis*=-1;
   if (x1>a){
      R ex=eps(-1);
      R ea=R1(a).eps(1);
      return R(2)*dis/(ea-ex);
   }
   else{
      R ex=eps(1);
      R ea=R1(a).eps(-1);
      return R(2)*dis/(ex-ea);
   }
}

Z R1::run(R final, ostream& str, Z writePeriod)const
{
   R xMem=x1;
   R xRun=x1;
   R e=R1::eps0.x1;
   Z wrt=0;
   Z count=0;
   if (final>xRun){
      while(xRun<final){
         step_p(xRun,e);
         count++;
         if (count==writePeriod){
            str<<"xRun-xMem="<<xRun-xMem<<" e="<<e<<endl;
            ++wrt;
            count=0;
         }
      }
   }
   else if (final<xRun){
      while(xRun>final){
         step_m(xRun,e);
         count++;
         if (count==writePeriod){
```

```
            str<<"xRun-xMem="<<xRun-xMem<<" e="<<e<<endl;
            ++wrt;
            count=0;
         }
      }
   }
   else;
   return wrt;
}

// save functions

R R1::log()const
{
   if (x1>0) return ::log(x1);
   else if (x1<0) return ::log(-x1);
   else return -logmax;
}

R R1::log10()const
{
   if (x1>0) return ::log10(x1);
   else if (x1<0) return ::log10(-x1);
   else return -log10max;
}

R R1::exp()const
{
   if (x1>logmax) return max;
   else if (x1<-logmax) return 0;
   else return ::exp(x1);
}

R1 R1::inv()const
{
   R e0=eps0;
   if (x1>e0 || x1<-e0) return R1(1./x1);
   else if (x1>=0) return R1(max);
   else return R1(min);
}

using CpmRootX::Z1;
/////////////////////////// class Z1 ///////////////////////////////

const Z1 Z1::max(std::numeric_limits<Z>::max());
const Z1 Z1::min(std::numeric_limits<Z>::min());

Z1 Z1::pow(Z p)const
{
   cpmassert(p>=0,"Z1 Z1::power(Z p)");
```

```
   if (p==0) return Z1();
   else if (p==1) return Z1(x1);
   else if (p==2) return Z1(x1*x1);
   else{
      Z res=x1;
      Z nMult=p-1;
      for (Z i=1;i<=nMult;i++) res*=x1;
      return Z1(res);
   }
}

bool Z1::prnOn(ostream& out)const
{
   return CpmRoot::write(x1,out);
}

bool Z1::scanFrom(istream& in)
{
   return CpmRoot::read(x1,in);
}

Z Z1::com(const Z1& a)const
{
   if (x1<a.x1) return 1;
   if (x1>a.x1) return -1;
   return 0;
}

X2<Z,Z1 > Z1::polDec()const
{
   if (x1>=0) return X2<Z,Z1 >(x1,Z1(1));
   else return X2<Z,Z1 >(-x1,Z1(-1));
}

R Z1::dis(const Z1& x)const
{
   return CpmRoot::disT<Z>(x1,x.x1);
}

bool Z1::isPowOfTwo()const
      // Let us write x for *this, and let N={1,2,3,...} (no zero!).
      // We return true iff x=2^n with some n \in N
{
   if (x1<2) return false;
   Z f=2;
   while (f<x1) f*=2;
   return f==x1;
}

Z1 Z1::nextPowOfTwo()const
```

```
{
   if (x1<2) return Z1(2);
   Z f=2;
   while (f<x1) f*=2;
   return Z1(f);
}

Z1 Z1::nextLog2()const
{
   Z res=1;
   Z f=2;
   while (f<x1){ // after the loop f>=x1
      res++;
      f*=2; // guarantees that f grows over all bounderies
   }
   return Z1(res);
}

#endif
```

38 cpmuc.h

```
//? cpmuc.h
//? Status of work 2023-10-20.
//?
//?  ...


#ifndef CPM_UC_H_
#define CPM_UC_H_
/*
Purpose: Defining class UseCount and P<>.
Class UseCount is a tool for implementing reference
counting.

Credit: Modified from Andrew Koenig: Ruminations on C++,
AT&T Ch. 7. Authoritative and enlightening treatment.

Recent history:
    2012-01-11 non-constant functions renamed to ending in '_'
*/


#include <cpmbasicinterfaces.h>

//////////////////////////// class UseCount ///////////////////////////////

namespace CpmArrays{

   using CpmRoot::Z;

class UseCount { // support for reference counting and 'copy on write'
   // class UseCount of Koenig. See explanations there.
   // Instances of this class are used as data members in client classes
   // which may describe the state of several instances by a common piece
   // of 'free store'. Typical example is class P<> within the present
   // file. Classes derived from P<> are defined in cpmp.h, cpmfl.h,
   // and in cpmfile.h
public:
   UseCount():p_(new Z(1)){}
   UseCount(UseCount const& u):p_(u.p_){++*p_;}
   ~UseCount(){ if (--*p_==0) delete p_;}
      // not virtual since no derivations will be defined
   bool only()const{return *p_==1;}
      //: only
      // to be called in the destructor of the client class
 //  bool makeOnly()const{if (*p_==1) return false; --*p_; return true;}
 bool makeOnly(){if (*p_==1) return false; --*p_; p_=new Z(1); return true;} // original Koenig/Moo
      //: make only
```

```
   // The name 'makeOnly' needs explanation: If *this is 'only' already
   // no 'making' is involved and 'makeOnly' returns 'false'.
   // If it is not 'only' we do something and thus return 'true'.
   // What is achieved in this doing is, however, not necessarily the
   // status 'only' one step in the direction of becoming 'only'.
   // Only if we had *p_==2 prior to calling makeOnly() we have
   // *p_==1, i.e. the status 'only' after the call.
   // To be called in the assignment operator of the client class.
   // Named unalias() by Bruce Eckel (p. 437). To me both names
   // don't provide a clear suggestion.
   // Consider a typical usage of this function as it appears in
   // the copy on write function (to be called in the assignment
   // operator) of the array class V<T> which has data
   // T* p_, UseCount u_, ... :
   // template <class T>
   // void V<T>::cow_(void){
   //    if (u_.makeonly()){ p_= (sz_==0 ? 0 : copy()); u_.startNew();}
   // }
   // This function thus calls u_.makeonly() once. Only if its return
   // value is true any action happens. This is the case only if
   // *(u_.p_) > 1 so that the piece of free store which u_'s client is
   // using (and to which p_ is pointing) is also being used by a
   // further client. Then action is needed: (*this) connects to a copy
   // of the original piece of free store so that this original piece
   // will have one user less. This is what u_.makeonly does in its
   // instruction --*p. The situation thus is that the old users of the
   // 'land' still use it and the individual *this who tries to change
   // things has to move to a new piece of 'land' which is made into
   // a copy of his old 'land' and on which the intended changes can be
   // made without interfering with the needs of the users of the
   // 'old land'. This is now a clear procedure. Earlier my
   // understanding was that the old users have move and it was not
   // clear how *this would enforce such a cooperation of the old
   // users.

bool reattach_(UseCount const&);
   //:: reattach
   //'::' means that the name is longer than the one formed according
   // to the CPM standard abbreviation scheme. In most cases no
   // abbreviation at all.
   // To be called in the assignment operator of the client class
   // See P<> and V<>.
   // Recall that the names of non-constant functions end in '_'.

void startNew_(){ p_=new Z(1);}
   //:: start new
   // my addition to be used in the definition of function cow_() in
   // CpmArrays::V<>
Z getCount(void)const{return *p_;}
   //:: get count
```

```
private:
   UseCount& operator=(UseCount const&);
      // not implemented,to make sure that this will never be used
   Z* p_;
};

//////////////////////  class P< >  ////////////////////////////////
// Class of constant smart pointers, no arrays (thus no indexing),
// reference counting, but no copy on write. Tool for implementation
// of function class F<X,Y>.
// See cpmp.h for a derived class which implements copy on write.

template <class T>
   // T is any non-abstract class or any built-in type.
   // If T is immutable, i.e. an instance of T cannot be modified after
   // it has been created, (see Andrew Koenig: Ruminations on C++, AT&T
   // 1997, p. 178, 190 for the relevance of immutable types) P<T>
   // implements the 'strict value interface' (see file cpmv.h).
   // In the notation of predicate logic:
   // non-abstract T & immutable T ==> value class P<T>
   //
   // Let T(A const& a) be a constructor of type T, then
   // the normal syntax for creating an instance of P<T> is
   // A a=...;
   // P<T> pt=new T(a)
   // or
   // P<T> pt;
   // pt=P<T>(new T(a));
   // or
   // P<T> pt(new T(a));

class P { // 'P' as in 'pointer', constant smart pointer.
   // Handle class the instances of which have to be initialized by
   // 'new'-generated pointers. The class takes the full responsibility
   // for deleting these pointers properly.
   typedef P<T> Type; // needed for declaration macro CPM_ORDER
public:
   CPM_ORDER
      // Will be implemented by comparison of adresses.
      // No order-related operators of T used.

   P(T *p=0):p_(p){}
      // Constructor, including default constructor.
      // important !!!: use this constructor only
      // for pointers initialized with a 'new' statement (on free store,
      // not static!) Important also: This allows to create P<T>'s from
      // pointers X* x where X is derived from T.
      // P<T>(new X(..arguments..)) will hold all information, that
      // the object X x(..arguments..) contains, although T does not know
```

```
      // about the additional members of X.
      // Also u_ gets initialized by its default constructor which sets
      // the use count to 1.
      // The piece of free store that thus comes under the control of P
      // is of course not associated with a use-counter already. The case
      // that he is, will be handled by the next function:

   P(P<T> const& h):u_(h.u_),p_(h.p_){}
      // copy constructor
      // Here the pointer h.p_ points to a piece of storage which is
      // already in use by h.u_.getCount() clients.
      // The copy constructor of UseCount is made such that it increases
      // the usecount by 1.

   virtual ~P(){ if (u_.only()) delete p_; }
      // destructor

   P<T>& operator=(P<T> const& h);
      // assignment

// access operator, which does not allow to change p
   T const& operator()(void)const{ return *p_;}
      // Let pt an instance of P<T>; then it is conventional to denote
      // the value as *pt. According to my aim to avoid pointers in
      // public interfaces, I also tend to avoid pointer notation.
      // So I denote the value of pt by pt().
      // This looks very natural since 'getting to the data content' of a
      // container-like quantity is much like evaluating a function.
      // Since no argument is needed (as for standard rand(), for example)
      // a void pair of brackets gives the right association.
      // Assume that T = V<X>, then we may form pt()[1].
      // In dereferencing notation we had to write (*pt)[1].

   bool isVal()const{ return !(p_==0);}
      //: is valid
protected:
   UseCount u_;
   T *p_;
};

////////////////////////////// Implementation ////////////////////////////

template <class T>
Z P<T>::com(P<T> const& obj)const
{
   if (p_<obj.p_) return 1;
   if (p_>obj.p_) return -1;
   return 0;
}
```

```
template <class T>
P<T>& P<T>::operator=(P<T> const& h)
{
   if (u_.reattach_(h.u_)) delete p_;
   p_=h.p_;
   return *this;
}

} // namespace

#endif
```

39 cpmuc.cpp

```
//? cpmuc.cpp
//? Status of work 2023-10-20.
//?
//?  ...


#include <cpmuc.h>   // see for explanations

bool CpmArrays::UseCount::reattach_(UseCount const& u)
{
   ++*u.p_;
      // no conflict with constantness of u !!
      // count of u becomes incremented whereas in the next statement
      // the count of *this becomes decremented
 //  if (--*p_==0){     // --p==0 in Koenig's book, works but is weird syntax
   if (--p_==nullptr){
      delete p_;
      p_=u.p_;
      return true;
   }
   else{
      p_=u.p_;
      return false;
   }
}
```

40 cpmv.h

```
//? cpmv.h
//? Status of work 2023-10-20.
//?
//?  ...

#ifndef CPM_V_H_
#define CPM_V_H_
/*

Purpose: Building on std::vector

   Basic array class with valid indexing ranging in a contiguous
   subset of Z. Two cases are of particular interest: That the lowest
   valid index is 0 (as for std::vector<>) or 1 (as e.g. in the functions
   in Press et al.). When dealing with FFT index ranges
   {-n,...,-1,0,1,...,n} are adequate (not yet implemented).
   Most constructors of V create instances of V<T> with valid indexes
   starting at 1. The two functions b_(Z) and e_(Z) provide means
   to arbitrarily shift the range of valid indexes. Access to components
   by means of indexing is range checked.

History: Till October 2010 there was a class template Vl<T> with
indexing starting at 0 and - based on that (not derived from that)
a class template V<T> with indexing starting at 1. This caused an
uncomfortable situation: Whenever dealing with a topic where I felt
that efficiency would be of utmost importance (e.g. font representation
and quantum dynamics) I was tempted to use Vl<> not only as an
internal device but also in the interface of public member functions.
This made the implementation code and even the classes dependent
on a decision that with a slight twist of emphasis could also have
been made differently. The new state of affairs is that Vl has been
eliminated (actually replaced by V) in all C+- code. If it should still
shine up in some comment, ignore it.

More recent history:
    2012-01-11 function cow() renamed to cow_()
    2012-01-18 function X2<Z,bool> findAsc(T const& t)const added
    2012-01-19 function prnOn modified so that indexes are printed
      (in commentarized form) together with the components.
    2017-02-18 Short report on a project which looked promising
      but was finally abandoned:
      The rather mature state that C++ has reached with C++11 triggered the
      desire to make more systematic use of the standard library
      facilities. Especially that now all standard containers are said to
      implement moving as a replacement of copying where appropriate
      promised to make it feasible to replace T* p_ by std::vector<T> p_
      and thus free the implementation of V<T> from defining copy, move,
```

```
      assignment operators. The first disappointing problem was that then
      V<bool>, since based on std::vector<bool>, did no longer work in my
      code which used operator[](Z) for V<bool> as in all other V<T>'s.
      Of course replacing V<bool> by V<B> would help. But re-writing all
      the many functions of V<T> in the new style looks disappointingly
      tedious to me. Before making a second attempt I need to understand
      whether the new move functionality is in fact a full replacement for
      the reference counting and copy on write functionality which is
      implemented in V<T> as of today. There is a project
      'codingexperiments' in ~/e/cpm/codingexperiments which illustrates
      the malfunction of vector<bool>, which also is well known to the
      WWW. Further there is ~/e/cpm0ExperimentBasedOnvector which
      contains the experimental version of cpmv.h. In this version by far
      not all uses of T* are replaced by vector<T>.

   2022-10-15 When revising the first tutorial on C+- felt the need
      base my abandoning of 'reference counting' and 'copy on write' in
      favour of the new panacea of 'move semantics' on hard facts.
      It turned out that my original test program gave for both
      std::vector and CpmArrays::V nearly the same speed. Where was the
      more than hunderedfold speed advantage of CpmArray::V that I saw with
      previous versions of V? Since V now relied on move semantics (imported
      by storing data not as T* but as std::vector<T> ) the similarity of
      speeds was no surprise, the surprise was that the speed was as low
      as was observed with a completely elementary beginners implementation.
      Only after having reintroduced reference counting the old advantage
      returned. I had to trivialize my test program to see std::vector
      switching to move semantics and getting to the speed V. Probably
      one could, by inserting move commands by hand, enforce acceleration.
      For C+-, however, only fully automatic methods are considered acceptable.
      As a consequence, reference counting and copy on write are back again.
      Although the new code does not mention 'move' and '&&' the obtained classes
      are diagnosed as 'movable', 'semiregular', and even 'regular'.
      Where in earlier versions V stored data as T* we now store them
      as shared_ptr<vector<T>> and implement copy on write along the lines
      explained in Stroustrups's C++11 book, p. 512. Using std::vector lets
      V's easily grow like vectors. And the default definitions of
      copy construction, assignement, and virtual destructor work fine.

This defines a template V<T> of T-valued lists or 'vectors with
T-valued components'. It is similar to std::vector<T>
but it differs from this in some respect. In describing
V<T> and these differences, I'll introduce some notions and notations
that will be used over and over in defining and stating properties of
other CPM classes
( CPM = 'C+-' or 'Classes for Physics and Mathematics').

1. Notice that V<> is not a 'polymorphic container'
   i.e. the components can hold only T typed objects and not the
   additional information content which instances of classes derived
```

```
   from T may carry. We may, however, form V<T*> for any type or class
   to support polymorphism in the old-fashioned pointer-based way.
   A safer way is to use the polymorphic vector template Vp mentioned
   in item 5.

2. V<> implements 'reference counting' and 'copy on write' a wellknown
   mechanism for avoiding making copies of temporary (potentially large)
   objects. The C+- tutorial project tut1 has examples of where the
   new strategy of 'move semantics' fails to bring about the speed
   advantage which 'copy on write' provides reliably.

3. Requirements on T in order to allow the formation of V<T>:
   These are the same requirements valid for std::vector<T>.
   'for all objects t1 and t2 of type T, the expression t1<t2 is defined'
   (compare David R. Musser, Atul Saini: STL Tutorial and Reference
   Guide, Addison-Wesley 1996, p. 246). Now the pertinent statement:
   V<T>  is well defined if (not iff!) T is a built-in type or a class
   which 'implements the value interface'. Then, V<T> also implements the
   value interface. If, moreover, T 'implements the strict value
   interface', then V<T> also implements the strict value interface.
   Here, a class T is said to implement the value interface if it
   publicly defines T(), T(const T&), and T& operator=(const T&).
   T is said to implement the strict value interface if, in addition, it
   has value semantics (as opposed to pointer semantics, see
   Andrew Koenig: Ruminations on C++, AT&T 1997, p. 62. and
   Bjarne Stroustrup: The C++ Programming Language, 3. edition,
   Addison-Wesley 1997, p. 294). Classes that implement the strict value
   are most convenient to use. Code only using such quantities is
   normally easy to understand. To have an even shorter denotation for
   this favorable species we call such a class a value class or a bit
   more general:

      T is a value class : <==> T is a built-in type or a class
      that implements the strict value interface.

   As was stated already, we have:

      value class T ==> value class V<T>

   The predicates 'T satisfies the value interface' and
   'T satisfies the strict value interface' can be evaluated for
   classes providing random generators by means of the test classes
   Test_v<T> and Test_sv<T> defined in file cpmtests.h. Although V<>
   provides no random generator (Vr<>, to be mentioned soon, does)
   the code of Test_sv<T> can be read as an operational definition
   of the concept 'strict value interface'. An interesting (or common-
   place ?) observation: syntactic aspects of a program are characterized
   by the program's interaction with the compiler; semantic aspects
   ('pointer sematics',...) are characterized by the data created during
   execution.
```

```
   Requiring order operators or even arithmetic operators in T, allows to
   define T-'valued' Vectors which themselves carry natural
   order/arithmetic operators.
   Therefore V is only the starting point in a series of vector
   templates with increasing functionality, accompanied by increasing
   assumptions on the structure of T. Presently this hierarchy looks as
   follows:

      V<T> , Vo<T> , Va<T> , Vr<T>

   Every such class is derived from its predecessor in this series
   by derivation without adding new data members. Thus there are
   unambiguous casts between all these classes.

      Vo : More order related methods added to V, basic order functions
         which are declared in CPM_ORDER now already here.
      Va : arithmetic operations added to Vo
      Vr : rich infrastructure supporting automated testing of
         internal consistency.

   This approach of integrating the functions (algorithms) into an
   inheritance tree of template classes is radically different from the
   STL approach. STL keeps algorithms as separate entities outside the
   classes and uses iterators and adaptors for making them work together.
   Both methods have their advantages and disadvantages. It is probably
   fair to say that the C+- approach is less universal but more
   convenient to use within the framework it fits.

4. On polymorphism: V<T*> may be formed, but doesn't implement the value
   interface for T and derived classes. However, with the smart pointer
   templates P<T>, Pp<T>, and Po<T> defined in file cpmp.h we can form
   e.g. V<Pp<T> > and get the functionality (together with some 'extras')
   which one would expect from  V<T*>. See file cpmp.h for details and
   the polymorphic version Vp of the vector templates mentioned so far.
   See test class PolymorphicMulti<*,*,*> in cpmtests.h for a operational
   definition of polymorphism of container classes.
*/

#include <cpmfl.h> // Includes <cpmuc.h> for std::size_t
   // (and std::ptrdiff_t, which is not being used so far).
   // Small and efficient function template with minimum
   // infra-structure requirements.
#include <cpmzinterval.h>
   // includes cpmsystem.h and cpmx.h
   // With using arrays something may go wrong and so messaging
   // capability is indispensible.
#include <cpmtypes.h>

#include <cpmmacros.h>
```

```
   // Provides help to write debugging-friendly function blocks.
#include <memory>
#include <vector>
#include <valarray>
#include <set>
////////////////////////////// class V<> ////////////////////////////////
// Array with index check, reference counting, and copy on write, and
// 'value semantics' (as opposed to 'pointer semantics')
// Generalized from Koenig's class Handle p. 72-73.
// Index range is now defined as an instance of class IvZ. So each
// 'vector' has its individual index range which may start with 1
// (following the convention of the Numerical Receipes) or with 0 as
// for into the 'vector' of the STL.
//  V< V<ColRef> > is the type of bitmap data in my graphical workhorse
// class Img24. This can be considered a proof for good performance of
// the class. Efficient conversion functions from and to std::vector<>
// are now provided.
// There is a nice way to iterate over the whole index range without
// mentioning the bounds as 0 and dim-1, or as 1 and dim:
//    V<T> v=....;
//    for (Z i=v.b();i<=v.e();++i) v[i]=...;

namespace CpmArrays{

   using namespace CpmStd;

   using CpmRoot::Word;
   using CpmRoot::Z;
   using CpmRoot::N;
   using CpmRoot::toN;
   using CpmRoot::R;
   using CpmRoot::B;
   using CpmRoot::Root;
   using CpmSystem::Error;
   using CpmFunctions::F;
   using std::vector;
   using std::valarray;
   using std::shared_ptr;

#ifdef CPM_Fn
   using CpmFunctions::F1;
#endif

// some infrastructure

   enum Begin { LEAN };
      //: begin
      // Never form V<Begin>

  // enum Outside { DEFAULT, CYCLIC, CONSTANT, ZERO };
```

```
enum Outside { ZERO, CYCLIC, DEFAULT, CONSTANT };
   //: outside
   // Controls the meaning of 'out of range indexes'.
   // DEFAULT: definition as the default value associated with
   // the type under consideration
   // CYCLIC: setting the meaning of out of range indexes by
   // cyclic repetition of value
   // CONSTANT: continuation as constant from the nearest value
   // ZERO: only used in R_Vector, R_Matrix, C_Vector, and C_Matrix
   //    stands for reflective boundary conditions

extern Z dimMax;
   // If the dimension of an array was either the result of a
   // calculation or of reading from a file, then the result may be
   // off the programmer's intent by orders of magnitude if something
   // went wrong. So it is helpful to exclude unnaturally large arrays
   // from becoming allocated.

extern Z firInd;
   // first index of arrays. Means to let C+-
   // cooperate with std containers.

extern bool ranChc;
   //: range check
   // Initialized as true.

extern bool ranChcAlw;
   //: range check always
   // If this is true, all access operators even those the name
   // of which suggests the opposite get checked --- with poor
   // diagnostics, though.
   // Initialized as true, since access to non-allocated memory,
   // as a rule, causes disaster. I had to discover in 2008-03-03
   // that such a case happened in the workhorse function
   // CpmGraphics::Graph::mark(V< V<C> >,...)
   // on a regular basis, due to an seemingly safe but actually
   // un-safe usage of V<>::cui().

extern bool signal;


void setDimMax(Z n);
   // sets dimMax=n unless n<0. In this case error with message

Z safeDim(Z n);
   // returns n for 0<=n<=dimMax, otherwise error with message

Z makeIndValNotMember(Z i, IvZ iv, Outside mode);
    //: make index valid
```

```
template <class T>
   // We assume that T provides (explicitly or implicitly)
   // copy constructor, and assignment or that T is a built-in type.
   // If the index operator [] is to be used and the variable ranChc
   // (which is initialized as 'true') was not set to 'false' an out
   // of range error will result in a runtime error which will be
   // documented on cpmcerr.txt. See ranChcAlw for an additional control.
   // The error message is particularly explicit (indicating the type of
   // T) when also the macro CPM_NAMEOF is defined. If this is the case,
   // the type T needs to define the member function
   // CpmRoot::Word nameOf()const. For all Cpm-classes this is the case
   // and for user classes, there should be no difficulty in adding such
   // a function. If one wants to make use of the function declared
   // by the declaration macros CPM_ORDER type T needs to define the
   // member function  CpmRoot::Z com(T const&)const;
   // If one wants to make use of the functions declared by the
   // declaration macro CPM_IO, type T needs to define the member
   // functions bool prnOn(ostream&)const and bool scanFrom(istream&).
   // These requirements do not apply to basic types:
   // For T = N, Z, R, Rh, L, bool, string one  may use all functions of
   // V<T> without further requirements.

class V{ // vector template, indexing starts with 1 by default.
   // The index range can however be shifted or even initially set
   // arbitrarily.
   // Although implementation details are inspired from handle classes,
   // the V class is a value array and not a handle class.
   // All allocations made with new T[] all de-allocations are  delete[]
   // All data are private, so the only access to data in
   // derived classes is over the public functions of the class. All
   // these incorporate reference counting internally where needed (only
   // if the preprocessing directive  is active). The user of
   // the class can't see this and has not to be aware of this.

   using Type=V<T>;
   using spv=std::shared_ptr<vector<T>>;

public:
   CPM_IO
   CPM_ORDER
   R dis(V<T> const& h)const;

//   V()=default; // surprisingly this does not work for:  V<Word> some;

//   V():iv_(),sz_(0),n_(0)
//   // default constructor, 0 components and not 1, as was inconvenieltly
//   // set in R_Vector, R_Matrix, C_Vector, C_Matrix.
//   // 2023-04-28: all default constructors of arrays now have 0 components.
//{
//   p_=spv(new vector<T>(0));
```

```
//}

explicit V(Z n=0):iv_(n), sz_(n), n_(sz_)
{
   iv_.b_(firInd);
   p_=spv(new vector<T>(sz_));
}
   // Has n components. Gives an error for n<0 .
   // The components are initialized by the default constructor
   // of T if T is a class for which such a constructor is defined
   // (explicitly or implicitly).
   // If T is a built-in type, initialization is done as 0.
   // See BS3, p. 131 for initialization of built-in types via
   // formal constructor calls.
   // If n>0, the first valid index is 1, which reflects the
   // normal behavior of class V.

   V(Z n, Begin bg);
      // Defined as the previous function. But the first valid index
      // is 0 (if n>0 so that there is at least one valid index).
      // This is a somewhat contrived construction: One has to make sure
      // that this does not interfere with the definition
      // V(Z n, T const& t, Z first=1) for some choice of T. Since we
      // agree on using V<T> only for C+- types T, we will never be
      // tempted to consider V<Begin>.
      // Typical usage:
      //    V<R> v(4,LEAN); // recall: enum Begin { LEAN }
      // lets v have valid indexes 0,1,2,3. v[0]=...=v[3]=0.
      //    V<R> w(4);
      // lets w have valid indexes 1,2,3,4. w[1]=...=w[4]=0.

explicit V(IvZ const& iv);
      // Notice that IvZ iv(3); // example
      // generates a set of 3 contiguous integers starting with 1, thus {1,2,3}
      // Has a component for each element of iv.
      // The comFponents are initialized by the default constructor
      // of T if T is a class for which such a constructor is defined
      // (explicitely or implicitely).
      // If T is a built-in type, initialization is done as 0.
      // See BS3, p. 131 for initialization of built-in types via
      // formal constructor calls.

   V(Z first, vector<T> const& v);
      // Construction from a standard library vector.

   V(Z n, T const& t, Z first=1) ;
      // Has n components all initialized as t.
      // The third argument gives the first valid index.

   V(Z n, T const& t, Begin bg);
```

```
      // Has n components all initialized as t.
      // The third argument (that can have only one value, namely LEAN)
      // says that the first valid index is 0.

   V(IvZ const& iv, T const& t);
      // Has iv.car() components all initialized as t.

   V(IvZ const& iv, vector<T> const& p):iv_{iv}, sz_{iv_.car()},
      n_{toN(sz_)}
   {
      cpmassert(n_ == p.size(), "size mismatch in V(IvZ,vector)");
      p_=spv(new vector<T>(p));
   }
      // Has iv.car() components all initialized with the components of p.

   V(Z n, F<Z,T> const& f);
      // construction from a function. Of course,
      // V<T> v(n,f);
      // implies v[i]==f(i) for all valid indexes i of v.

   V(IvZ const& iv, F<Z,T> const& f);
      // construction from a function. Of course,
      // V<T> v(iv,f);
      // implies v[i]==f(i) for all valid indexes i of v.

 // constructors from explicit lists
   explicit V(std::initializer_list<T> il );
      // requires C++11
      // constructors from explicit lists such as
      // V<Z> v{0,1,2,4,8};
      // The first valid index always is 1. Therefore
      // v[1]=0,...v[5]=8

//   V()=default;
// since we want to derive from V we have to follow the rule of five
// with defaults

   V(V<T> const&)=default;
      // copy construction

   V(V<T> &&)=default;
      // move construction

   V<T>& operator=(V<T> const&)=default;
      // assignment

   V<T>& operator=(V<T> &&)=default;
      // move assignment

   virtual ~V()=default;
```

```
      // destructor


   virtual V<T>* clone(void)const{ return new V(*this);}

   V<T> toClnBase()const{ return *this;}
      //: to clone base

   Z dim()const { return sz_;}
      //: dimension
      // Returns the number of components of the vector *this

// Returns the number of components of the vector represented by
// the built-in integer type std::siz_t (alias N). Thus the term
// vector<X>(nc()) is perfectly right without any type conversion.
   N nc()const{ return p_->size();}
      //. number (of) components

   Z size()const { return sz_;}
      //: size
      // for uniformity with STL

   IvZ dom()const { return iv_;}
      //: domain
      // Notice that with the array *this there is the
      // function f: {iv_b(),...,iv_.e()}-->T, i|-->(*this)[i]
      // associated in a natural manner.
      // For this function, dom() is just the domain.
      // The understanding of arrays as functions with domains of type
      // IvZ seems to be a good guide for defining some of the member
      // functions in a more natural manner by using arguments of type
      // IvZ. Present examples are the functions fa_ and valOn.

   bool isVoid()const { return iv_.isVoid();}
      //: is void
      //  short answer on whether the dimension is zero

   bool valInd(Z i)const{ return iv_.hasElm(i);}
      //: valid index
      // returns the validity of i as an index of *this

   Z makeIndVal(Z i,  Outside mode=CYCLIC )const
   {
      if (iv_.hasElm(i)) return i;
      if (mode==CYCLIC) return iv_.cyc(i);
      else return iv_.con(i);
   }
      //: make index valid
      // If i is a valid index we return i. If not, the result is
      // iv_.cyc(i) for mode=CYCLIC and iv_.con(i) else. Notice that the
```

```
      // normal treatment of mode==DEFAULT works not by replacing one
      // value of the index by another. It works on the value of vector
      // components and makes use of the default constructor T()

   bool sameDom(V<T> const& v)const{ return iv_==v.iv_;}
      //: same domain

   vector<T> std()const{ return *p_;}
      //: standard
      // Returns a STL-vector which holds all components of *this.

   vector<T> toV()const{ return *p_;}

   valarray<T> toValArr()const{
      N n=nc();
      valarray<T> res(n);
      for (N i=0;i<n;++i) res[i]=(*p_)[i];
      return res;
   }

   virtual Word nameOf()const;
      //: name of
      // returns a name of the type

// constant access functions

   const T& operator[](Z i)const;
      // Getting to the value of a component of a const instant of
      // V<T>. For instance
      // const V<T> v = ... ;
      // T t = v[3];
      // The actual process of going from v to v[3] depends on whether v is of
      // type V<T> or const V<T>. In the non-constant case the evaluation of []
      // may involve a copy action on v (and returning the component of
      // the copy). If we intend only to read the component, such an
      // copy action is not neededd and should be avoided by casting v
      // to type const V<T>.
      // Casting v from V<T> to const V<T> works this way:
      //    v = static_cast<const V<T>>(v);
      // Casting back to mutable seems to work only by using a new name:
      //    auto vMutable = const_cast<V<T>&>(v);
      //    vMutable[1] = T(); //( for instance)
      // or by applying mutating operations to an anonymous object as in:
      //    const_cast<V<T>&>(v)[1]=T();
      // Notice the '&' here which the compiler requires. By the way, the
      // effect of the 'anonymus action' actually is to change the state
      // of v: v[1]==T().
      // My analysis of the situation is in ~/codingexperiments/main.cpp
      // In the following there are many pairs of functions
      // T const& f(...)const;
```

```
   // T& f(...);
   // for which the above considerations apply mutandis mutatis.

T const& cui(Z i)const;
   //. component (with) unchecked index
   // getting to the value of a component for 'read', e.g.
   // T t=cui(i);
   // It helps to write efficient code in classes which use V<>-typed
   // data members.
   // Notice that writing loops by using b() and e() to define
   // the range is much safer than using limits like 1 and dim().
   // Index range check is missing only if variable ranChcAlw was
   // changed to 'false'.

T const& pi(N i)const{ return (*p_)[i];}
   //. plain index
   // index range not checked.
   // It helps to write efficient code in classes which use V<>-typed
   // data members.

T const& pr(Z i, Outside mode)const{ //. plain read, see below for more
   if (0 <= i && i < sz_) return (*p_)[i];
      // Notice that for i of type CpmRoot::Z the expression (*p_)[i]
      // is perfectly right. Of course, the same is true for i of
      // type std::size_t (for which CpmRoot::N is an alias).
   if (i == -1){
      if (mode==DEFAULT) return def_;
      if (mode==CYCLIC) return (*p_)[n_-1];
      if (mode==CONSTANT) return (*p_)[0];
   }
   if (i == sz_){
      if (mode==DEFAULT) return def_;
      if (mode==CYCLIC) return (*p_)[0];
      if (mode==CONSTANT) return (*p_)[n_-1];
   }
   return def_; // should never happen
}
   //. plain read
   // from an index which may be out of range by one, in which case
   // the return value is determined in an self-evident way by the
   // Outside-typed second argument.

T& pi(N i){ return (*p_)[i];}
   //. plain index
   // index range not checked.
   // It helps to write efficient code in classes which use V<>-typed
   // data members.

T const& cyc(Z i)const;
   //: cyclic
```

```
   // getting to the value of a component for 'read', e.g.
   // T t=cyc(i);
   // Here i is understood as cyclic (i.e. i modulo sz_). So no value
   // of the index has to be considered 'out of range' and for i
   // 'in range' cyc(i)==(*this)[i]

T const& li(Z i)const{ return (*p_)[i];}
   //. lean index
   // The valid range is for (Z i=0;i<sz_;++i) li(i)

T& li(Z i){ return (*p_)[i];}
   //. lean index

T const& con(Z i)const;
   //: constant
   // Same logic as cyc() but with constant continuation
   // instead of cyclic.

T operator()(Z i, Outside mode=DEFAULT)const;
   //: ()
   // Read access defined for all i.
   // By this definition, a vector becomes a mapping from Z to T.
   // For i's outside the proper range,
   // the default T is returned for mode==DEFAULT or if
   // sz_==0. If mode==CYCLIC cyc(i) gets returned.
   // For mode==CONSTANT we return (*p_)[0] for i<=0 and (*p_)[sz_-1]
   // for i>=sz_.
   // Notice that the return value is not a reference in
   // accordance with function behavior.

T const& read(Z i, Outside mode=DEFAULT)const;
   //: read
   // Same as previous function but returning a reference instead of a
   // T.
   // Reading components in an efficient (as &'s), safe and flexible
   // manner.
   // See T operator()(Z i, Outside mode=DEFAULT)const; for the
   // meaning of the second argument.

T const& r(Z i)const{ return (*p_)[iv_.ri(i)];}
   //. read
   // Unchecked and fast version of T const& oparator[](Z i)const

T& w(Z i)const{ return (*p_)[iv_.ri(i)];}
   //. write
   // Unchecked and fast version of T& oparator[](Z i)

F<Z,T> fnc(Outside meth=DEFAULT)const;
   // function
   // returns the function Z --> T, i |--> (*this)(i,meth)
```

```
 // non-constant access functions
   T& operator[](Z i);
      //: []
      // See T const& operator[](Z i)const for discussion of details
      // on the use count mechanism that comes into the play here

   T& cui(Z i);
      //. component (with) unchecked index
      // setting to the value of a component
      // T t=...;
      // V<T> x=...;
      // x.cui(i)=t;

    T& cyc(Z i);
      //: cyclic
      // setting to the value of a component
      // T t=...;
      // V<T> x=...;
      // x.cyc(i)=t; meaning of i is modulo sz_. So i is never out of
      // range

   T& con(Z i);
      //: constant
      // Same logic as cyc() but with constant continuation
      // instead of cyclic.

   V<T>& b_(Z i){ cow_();iv_.b_(i); return *this;}
      //. set b(), i.e. the first valid index.
      // For instance
      // V<R> v("",sqrt(2),sqrt(3),sqrt(4),sqrt(5));
      // for (Z i=1;i<=v.dim();++i) cout<<v[i]<<endl;
      // v.b_(0);
      // for (Z i=0;i<v.dim();++i) cout<<v[i]<<endl;
      // shows all components of the vector in both cases.

   V<T>& e_(Z i){ cow_();iv_.e_(i); return *this;}
      //. set e(),  i.e. the last valid index.

// accessing the first and the last element for reading
   T const& fir()const;
      //: first
   T const& last()const;
      // last
   T const& median()const;
      // median. If dim() is an even number, we take the smaller of the two
      // candidates

// interface to the modern index-free looping device 'span'
```

```
auto begin(){ return (*p_).begin();}
   // allows to write code such as
   // V<Z> v{-1,1,-2,2,-3,3};
   // for (auto const& x : v) cout<<x<<endl;
   // for 'looping over the range of v'.
   // also possible
   // for (auto q=v.begin(); q!=v.end();++q){ cout<<*q<<endl;}

auto end(){ return (*p_).end();}
auto cbegin(){ return (*p_).cbegin();}
auto cend(){ return (*p_).cend();}

// accessing the first and the last element for writing
   T& fir();
      //: first
   T& last();
      // last
   T& median();
      // median. If dim() is an even number, we take the smaller of the two
      // candidates

// getting the first and the last valid index
   Z b()const { return iv_.b();}
      //. begin
      // Returns the first valid index.

   Z e()const { return iv_.e();}
      //. end
      // Returns the last valid index.
      // Allows to write loops over all components as
      // for (Z i=v.b();i<=v.e();i++) ... v[i] ...;
      // Note that this is safe also for v.dim()==0, since then
      // v[e()] will never be called. Here one could replace
      // v[i] by v.cui(i) without danger.

   Z n()const { return iv_.n();}
      //. next
      // Returns the index next to the last valid one.
      // This allows to write loops over all components as
      // for (Z i=v.b();i<v.n();i++) ... v[i] ...;
      // Note that this is safe also for v.dim()==0, since then
      // v[e()] will never be called. Here one could replace
      // v[i] by v.cui(i) without danger.

   V<T> meet(IvZ const& iv)const;
      //: meet
      // Returns a vector which, when considered as a function is the
      // restriction of function *this to the domain iv_ & iv.

   V<T> join(IvZ const& iv)const;
```

```
   //: join
   // Returns a vector which, when considered as a function is the
   // extension of function *this to the domain iv_ | iv, where
   // all function values on iv\iv_ are T().

V<T> operator +(Z i)const{ return V<T>(iv_+i,p_,"");}
   //: operator +
   // Returns a vector res such that res.dom() is the shifted
   // domain dom()+i of *this and has the same components as
   // *this.

V<T> operator -(Z i)const{ return V<T>(iv_-i,p_,"");}
   //: operator -
   // Returns a vector res such that res.dom() is the shifted
   // domain dom()-i of *this and has the same components as
   // *this.

void set_(T const& t);
   //: set
   // non-constant function which sets all components of (*this)
   // equal to t

V<T> set(Z i, T const& t)const;
   //: set
   // Returns a vector that originates from *this by setting the
   // value of component i.
   // Regular behaviour:
   // if we say
   // Z i=...;
   // T t= ...;
   // V<T> v=...;
   // v=v.set(i,t);
   // then the i-th component (see eli() ) of v will get the value t
   // unless i<1.
   // If i is a value for which (*this)[i-1] is not yet defined,
   // the vector becomes enlarged to the necessary size and all
   // not specified components initialized with the default
   // constructor of T

T in_(T const& t, bool reversed=false);
   //. insert
   // This operations treats *this as a shift register:
   // All components get shifted by one position 'to the right' and
   // the total length (dim) of the vector remains the same. So what
   // was the last component prior to the operation has to be removed
   // from the vector, in order to not wasting information, this
   // removed component will shine up as the return value of the
   // function. After the operation, the first component of the vector
   // will be t.
   // If reversed==true, t gets inputted at the end, and all shift
```

```
      // operations go 'to the left'.


   Z locAsc(T const& t)const;
      //: locate ascending
      // Here it is assumed that *this is ascendingly strictly ordered:
      // If sz_>=2 we have (*p_)[i]<(*p_)[i+1] for all i \in {0,sz_-2}.
      // For sz_<2 there is no condition.
      // For sz_==0 we stop with error.
      // For t<fir() we return b()-1
      // For t>=last() we return e()
      // If none of the previous conditions was met we necessarily have
      // sz_>=2 and we return the uniquely determined j such that
      //    (*p_)[j]<=t<(*p_)[j+1].
      // The possible results from this regular part of the functionality
      // thus are b(),...,e()-1.
      // Notice that the very similar function locate of Press et al.
      // only guarantees (*p_)[j]<=t<=(*p_)[j+1] for its return value j.

   Z find(T const& t, bool ordered=true)const;
      //: find
      // If the return value is b()-1, there is no valid index i
      // such that (*this)[i]==t. Otherwise, the return a value i
      // such that (*this)[i]==t.
      // If ordered==true, this assumes that (*this) is in increasing
      // order. The method is by bisection and logarithically fast.
      // If no order is assumed the components are compared against t
      // starting from index b(). The first occurrence of t gets reported.

// building new objects from given ones (generative methods)

// concatenating vectors and appending components. These operations have
// always O(sz_) complexity. The corresponding combined assignements are
// less direct to implement are not considered useful in the present
// context (they would not be more efficient than the friend versions,
// since new memory has to be allocated anyway).

   V<T> app(T const& t)const;
      // returns a vector which is obtained from *this by appending t
      // as the last component
      // notice also the prepend functions to be introduced
      // after the insert-functions (in order to have the inline
      // definition available).

   V<T> app(V<T> const& h)const;
      // returns a vector which is obtained from *this by appending h
      // at the back end

   void push_back(T const& t) { *this=app(t);}
      // for uniformity with STL
```

```
   V<T>& operator<<(T const& t) { return *this&=t;}
      // appending an element in Ruby-style
      // Allows successive application as in
      // V<Word> w;
      // w<<"many"<<"words"<<"get"<<"easily"<<"stored";

   V<T>& operator<<(V<T> const& h) { return *this=app(h);}
      // appending an array in Ruby-style

// replaced 2023-04-30 by the following function, since caused severe
// malfunctions in project quantumcrossway. The replacement is conventional,
// so it looks not urgent to understand the reason for the malfunction
// of the non-conventional version.
//   V<T>& operator&=(T const& t){
//      iv_.n_();
//      sz_++;
//      p_->push_back(t);
//      return *this;
//   }

   V<T>& operator&=(T const& t) { return *this=app(t);}
      // appending an array in Ruby-style

   V<T>& operator&=(V<T> const& h) { return *this=app(h);}
   V<T> operator&(T const& t)const { return app(t);}
   V<T> operator&(V<T> const& h)const { return app(h);}
      // appending in my favorite style

   V<IvZ> valOn(F<T,bool> const& f)const;
      //: valid on
      // retuns the array of those sub-intervals of the
      // whole indexing interval dom() on which the function
      // dom()-->bool, i|-->f((*this)[i]) yields true.
      // Notice that the result res \in V<IvZ> is a convenient
      // representation of a subset of dom(). Considering
      // *this as a function g: dom()-->T, then the function
      // h:=g&f is of type dom()-->bool and res, as a subset
      // of dom(), is h^-1({true}).

// resizing

   V<T> resize(Z newDim)const;
      // Returns a vector res such that res.dim()=newDim. If newDim is
      // smaller than dim(), the end of *this will be cut away. If newDim
      // is larger, T()'s will be added.
      // For newDim<0 the action is as if newDim==0.

   V<T> cut(Z i)const{ return resize(sz_-i);}
      // returned is a V<T> which results from *this by removing i
```

```
      // components from the end. For exotic values of i, see code
      // and explanation of resize.

// elimination and insertion

   V<T> eli(IvZ const& iv)const;
      // eli stands for eliminate
      // Eliminating all components which belong to iv. No exceptions
      // can happen! Universal and convenient of elimination. All other
      // forms are superfluous. Moreover, they are to be considered as
      // obsolete.

   V<T> eli(Z i, Z nEli=1)const{ return eli(IvZ(nEli,i,0));}
      // eli stands for eliminate
      // returned is a vector which is obtained from *this by
      // eliminating the i-th component and the nEli-1 following ones
      // (thus nEli components are removed) and shifting all later
      // components (if there are such components left) 'to the left' to
      // close the gap.

  // V<T> eliFirst(Z nEli=1)const { return eli(1,nEli);}
   V<T> eliFirst(Z nEli=1)const { return eli(IvZ(nEli,b(),0));}
      // returned is a vector which is obtained from *this by
      // eliminating the nEli first components. If no
      // argument is provided, actually the first component
      // gets eliminated. Notice that in the three-argument constructor
      // IvZ(i,j,k) the first argument is the cardinality, the second
      // argument is the first element, and the third argument is dummy.

  // V<T> eliLast()const { return eli(sz_,1);}
   V<T> eliLast()const { return eli(IvZ(e(),e()));}
      // returned is a vector which is obtained from *this by
      // eliminating the last component.

   V<T> eli1(V<T>& h, Z i)const;
      // We return a vector which results from *this by eliminating
      // h.dim() components starting at the i'th component (for i<1,
      // i==1 is understood). After the call h will hold the
      // eliminated components in due order (and no more - even if h was
      // longer before).
      // The function name ends in '1' to indicate that the first
      // argument is a non-constant reference, used for communication
      // of a part of the result.

   // condensation (contracting equivalent components into one)

   X2< V<T>, V<Z> > condense( bool (*equi)(const T&, const T&))const;
      // given an equivalence relation equi on T (t1~t2 <==>
      // equi(t1,t2)==true )
      // we divide the components of *this into equivalence classes.
```

```
   // Let the resturned pair be written as (res1,res2) . Then
   // (i) (*this)[i]~res1[res2[i]]
   // (ii) there is a j such that  (*this)[j]==res1[res2[i]]
   // Actually, the construction is made such that this j is the
   // smallest j for which (*this)[j]~res1[res2[i]].

V<T> select(V<B> const& s)const;
   // returned is a list which is obtained from *this by eliminating
   // all components (*this)[i] for which s[i] is defined and
   // statisfies s[i]==false. Beside of this removal, the order of the
   // components in *this is retained. So if s is defined
   // by a condition s[i]=Condition((*this)[i]), for the vector
   // component, this condition has to express a property we like to
   // have fulfilled for the result-vector of the select-operation.
   // 's expresses the favorable condition' and  n o t  the one to be
   // eliminated. Notice, that the select operation makes sense for
   // all values of s.dim().

V<T> ins(Z i, T const& t)const;
   //: insert
   // returned is a vector which is obtained from *this by
   // inserting t as the i-th component and shifting all later
   // components 'to the right' to give room for t.
   // The phrase 'i-th component' refers to natural counting (thus
   // (*p_)[i-1] is the i-th component of *this).

V<T>& ins_(Z i, T const& t);
   //: insert
   // returned is a vector which is obtained from *this by
   // inserting t as the i-th component and shifting all later
   // components 'to the right' to give room for t.
   // The phrase 'i-th component' refers to natural counting (thus
   // (*p_)[i-1] is the i-th component of *this).


V<T> ins(Z i, V<T> const& h)const ;
   // returned is a vector which is obtained from *this by
   // inserting h as the i-th and following component,
   // and shifting all later components of *this
   // 'to the right' to give room for h.
   // The phrase 'i-th component' refers to natural counting (thus
   // (*p_)[i-1] is the i-th component of *this).

V<T> prepend(T const& t)const
   // returns a vector which is obtained from *this by appending t
   // as the first component
   { return ins(1,t);}

V<T>& prepend_(T const& t);
   // changes *this by appending t  as the first component
```

```
V<T> prepend(V<T> const& h)const
   // returns a vector which is obtained from *this by appending h
   // at the front end
   { return h.app(*this);}

V<T> rev()const;
   //: reversed
   // returned is the reversed vector (indexing in the opposite
   // direction)

V<T>& rev_(){ return *this = rev();}
   //: reversed
   // changes *this into a reversed version of itndexing in the
   // opposite direction

// re-indexing

V<T> rot(Z s, Outside mode=CYCLIC)const;
   //: rotate
   // Name as the list transformation function Rotate of Mathematica.
   // v.rot(s)[i] == v(i-s,mode)
   // This means that we shift the component i of the original vector
   // into the new position i+s

void rot_(Z s, Outside mode=CYCLIC){ *this=rot(s,mode);}
   //: rotate
   // Same as rot, but as a mutating operation.


V<T> compose(V<Z> const& w)const;
   //:: compose
   // v.compose(w)[i] = v[w[i]]

V<T> permute(V<Z> const& w)const{ return compose(w);}
   //:: permute

// transforming components

V<T> operator+(V<T> const& h)const{
   cout<<"operator + called"<<endl;
   V<T> res(sz_);
   for (Z i=0; i<sz_; ++i){ (res.p_)[i]=(*p_)[i]+(h.p_)[i];}
   return res;
}

template <class Y>
V<Y> operator()(F<T,Y> const& f)const { return V<Y>(iv_,fnc()&f);}
   // generating arrays of different type by a type changing function
```

```
   V<T> operator()(T (*f)(T const&))const{ return fa(f);}
      // generating arrays of the same type with components f(c) instead of
      // c.

   V<T> operator()(T (*f)(T))const{ return faa(f);}
      // generating arrays of the same type with components f(c) instead of
      // c.

   V<T> operator()(std::function<T(T)>(f))const{ return fsf(f);}
      // generating arrays of the same type with components f(c) instead of
      // c.

// defining generative laws and mutating laws for various expressions by
// function pointers. The idea behind is, that for T which allows
// particular operations, we can define those without using iteration via
// a operator[] since these all are defined directly in terms of
// pointers.
// See implementation of +=, -=, ... in Va<T> how this works

   T fAcc1(T (*f)(T const&), void (*acc)(T&, T const&))const;
      // returns an accumulated value of all values f(c), where c
      // are the components of *this. Accumulation is defined by
      // the argument acc:
      // T res=T(); 'for all components c' acc(res,c)

   T fAcc2(T (*f)(T const&, T const&), void (*acc)(T&, T const&),
      V<T> const& h )const ;
      // returns an accumulated value of all values f(c,ch), where c and
      // ch are the components of *this and h respectively. Accumulation
      // is defined by the argument acc:
      // T res; 'for all components c' acc(res,c)
      // useful in defining scalar products

   template <class Y>
   Y fAcc3(Y (*f)(T const&, T const&), void (*acc)(Y&, Y const&),
      V<T> const& h )const ;
      // returns an accumulated value of all values f(c,ch), where c and
      // ch are the components of *this and h respectively. Accumulation
      // is defined by the argument acc

   V<T> fAcc4(F<T2<T>,T> const& f)const;
      // The kind of accumulation needed for computing the forces in
      // particle systems with pair interaction. Here we assume that
      // forces and positions are of the same type, e.g. R2 or R3.
      // Let x[1],...x[n] be the components of *this. We first compute
      // the incomplete matrix f(x[i],x[j]) i=1,...n, j<i and complete it
      // assuming f(x[i],x[j]) = - f(x[j],x[i]). In application mentioned
      // earlier this is the collection of mutually forces. The total force
      // on particle i is sum over j of f(x[i],x[j]). It is the i-th
      // component of the V<T>-typed return value of the function.
```

```
   // The force type T thus needs to provide operations unary - and +=.

V<T> fAcc5(F<R,R> const& dPot)const;
   // The kind of accumulation needed for computing the forces in
   // particle systems with pair interaction derived from a distance-
   // dependent (radial symmetric) potential. The argument dPot is
   // the derivative with respect to r of the r-dependent radially
   // symmetric pair potential.

V<T> fAcc6(F<R,R> const& dPot)const;
   // forces from a space-dependent potential

V<T> fsf(std::function<T(T)> f)const;
   // transforming the components with std::function

V<T> fa(T (*f)(T const&))const ;
   // returns a vector defined by replacing the components c of *this
   // by f(c)

V<T> faa(T (*f)(T))const ;
   // returns a vector defined by replacing the components c of *this
   // by f(c)

V<T> fb(T (*f)(T const&, T const&), T const& t)const;
   // returns a vector defined by replacing the components c of *this
   // by f(c,t)

template <class Y>
V<T> fb2(T (*f)(T const&, Y const&), Y const& t)const;
   // returns a vector defined by replacing the components c of *this
   // by f(c,t)

template <class Y>
void fb2_(T (*f)(T const&, Y const&), Y const& y);
   // replaces *this by a vector defined by replacing the components
   // c of *this by f(c,y)

void fc_(T (*f)(T const&, T const&), T const& t) ;
   // replaces *this by a vector defined by replacing the components c
   // of *this by f(c,t)

void fd(T (*f)(T const&, T const&), T& t)const;
   // replaces t by f(c,t) for each component c
   // If, for instance, f(c,t)=t+c*c we replace t by t+sum of c*c

V<T> fe(T (*f)(T const&, T const&), V<T> const& h)const;
   // returns a vector defined by replacing the components c of *this
   // by f(c,c') where c' are the components of h.
   // Error if dim()!=h.dim()
```

```
template <class Y>
V<T> fet(T (*f)(T const&, Y const&), V<Y> const& h)const;
   // returns a vector defined by replacing the components c of *this
   // by f(c,c') where c' are the Y-typed components of h.
   // Error if dim()!=h.dim(). Template version of fe. Unfortunately
   // h.p_ is not accessible in the implementation of this function.
   // This enforced introducing the non-canonical access function rep()
   // in 2014-01-23.

V<T> fe2(T (*f)(T const&, T const&), V<T> const& h)const;
   // Very similar to fe. However, the dimension of the result
   // is the maximum of dim() and h.dim(). Non-existing components
   // of any of the operands are replaced by T().

void ff_(T (*f)(T const&, T const&), V<T> const& h, Z i=0);
   // Replaces the components of p_ starting from (*p_)[i]
   // by f(c,c') where c' are the components of h.p_. Thus for i=0
   // and h.dim()>=sz_ the whole array p_ is affected.

void ffl_(T (*f)(T const&, T const&), V<T> const& h);
   // lean form of ff_
   // Replaces the components of p_
   // by f(c,c') where c' are the components of h.p_
   // p_ and h.p_ are assumed to be of the same size


void fg_(T (*f)(T const&, T const&, T const&),
   V<T> const& h, T const& t);
   // replaces the components c of *this
   // by f(c,c',t) where c' are the components of h

void fa_(F<T,T> const& f, IvZ iv);
   // replaces the components c of *this with index in iv
   // by f(c); modern form of fa.

void fh_(T const& w1, T const& w2, T const& w3, Outside mode);
   // replaces the component c[i] of *this by
   // w1*read(i-1,mode)+w2*read(i,mode)+w3*read(i+1,mode)
   // Thus makes use of multiplication in T.

template <class Y>
void fht_(Y const& w1, Y const& w2, Y const& w3, Outside mode);
   // replaces the component c[i] of *this by
   // read(i-1,mode)*w1+read(i,mode)*w2+read(i+1,mode)*w3
   // Thus makes use of multiplication in T.

template <class Y>
V<T> fh(T (*f)(T const&, T const&, T const&, Y const&, Z),
   Y const&, Outside mode)const;
   // Returns a vector in which the component c[i] is replaced
```

```
      // by f(c[i-1],c[i],c[i+1],y,i),where the i-/+1 are understood
      // according to mode. The quantity y helps to provide parameters
      // required by a concrete situation. Worked for implementing
      // the Hamiltonian corresponding to Dirac's relativistic wave
      // equation.

   template <class Y>
   V<Y> fi(Y (*f)(T const&, T const&), void (*acc)(Y&, Y const&))const;
      // Returns a vector with Y-valued components y[i] which are obtained
      // by accumulating the terms f((*this)[i],(*this)[j]).
      // Accumulation is defined by the argument acc.

   template <class Y>
   V<Y> fj(Y (*f)(T const&, T const&), void (*acc)(Y&, Y const&))const;
      // Returns a vector with Y-valued components y[i] which are obtained
      // by accumulating the terms f((*this)[i],(*this)[j]).
      // Accumulation is defined by the argument acc.
      // We assume f((*this)[i],(*this)[j]) == - f((*this)[j],(*this)[i])
      // and use this for doing only one evaluation of f
      // for the two pairs (i,j) and (j,i) and no evaluation
      // of f for any pair (i,i).
      // We assume that for any Y-object y, -y is defined.

     vector<T> rep()const{ return p_;} // 20014-01-24
      // This is needed in the implementation of V<T>::fet<Y>. Here we need
      // direct access to the data member p_ of a function argument of type
      // V<Y>. Unfortunately (and unexpectedly) what is OK for V<T> does
      // not work for V<Y>. This function conflicts with my ambition to
      // ban pointers from the public interface of C+- classes.

private:
// static data
   static const T def_;
      // allows read function to return references
// static functions
   static T fCyc(Z const& i, V<T> const& v) { return v(i,CYCLIC);}
   static T fCon(Z const& i, V<T> const& v) { return v(i,CONSTANT);}
   static T fDef(Z const& i, V<T> const& v) { return v(i,DEFAULT);}
 //  static N toN(Z i) { return static_cast<N>(i);}
      // size type
      // Instead of new T[i], I now write new T[toN(i)] so that
      // allocation is always fed with a parameter which fits the system
      // (in 64 bit systems, size_t may be 64 bit wide). size_t is known
      // due to inclusion of <cpmfl.h> and it is assumed to take the
      // bit-width of the machine properly into account.
   void ini_(){
      sz_=iv_.car();n_=toN(sz_);
      p_=spv(new vector<T>(n_));
}
```

```
 protected:

   void cow_(){ // needed to get non-constant member function right
      if (p_.use_count()>1){
         p_=spv(new vector<T>(*p_));
      }
   }

// data members
   // These should be accessible to derived classes for enabling fast
   // component operations.
   IvZ iv_;
      // Set of valid indexes. Since (*this) can be considered a
      // function iv_ --> T, this object is also called the domain
      // of *this.

   Z sz_{0};
      // number of valid indexes (depends on iv_, equals iv_.car())

   N n_{0};

   shared_ptr<vector<T>> p_;
      // data member that holds the components as a container.
};

//////// using V for some special template arguments//////////////////

V<Z> IvZtoVofZ(IvZ const& iv);
   //: IvZ to V of Z
   // returns the Z's that make up the interval iv
   // as the components of an ordered array (increasing
   // order, of course)

V<Z> VofIvZtoVofZ(V<IvZ> const& viv);
   //: V of IvZ to V of Z
   // appends the results from applying the previous functions
   // to the viv[i] according to the obvious code
   //  { V<Z> res(0); Z n=viv.dim();
   //  for (Z i=0;i<n;++i) res&=IvZtoVofZ(viv[i]); return res;}

V<Word> comLine(int argc, char* argv[]);
   //: command line
   // converts the C-traditional command-line argument into a C+- array.

////////////////////// Implementation ////////////////////////////////

template <class T>
const T V<T>::def_=T();

template <class T>
```

```
Word V<T>::nameOf()const{
   Word nt=Root<T>(T()).nameOf();
   Word wi="V<";
   return wi&nt&">";
}

template <class T>
R V<T>::dis(V<T> const& h)const
{
   R res=0.;
   if (iv_!=h.iv_) return R(1);
   for (Z i=b();i<=e();++i){
      res+=Root<T>(cui(i)).dis(h.cui(i));
   }
   return res;
}

template <class T>
T const& V<T>::operator[](Z i)const
{
   if (ranChc){
      if (!iv_.hasElm(i)){
         cout<<"index access error at i="<<i<<endl;
         cpmerror(nameOf()&
         "::operator[]const: read-index out of  range: i= "&
         cpm(i)&" iMin= "&cpm(iv_.b())&" iMax= "&cpm(iv_.e()));
      }
   }
   if (signal) cout<<" const [] called"<<endl;
   return (*p_)[iv_.ri(i)];
}


template <class T>
T& V<T>::operator[](Z i)
{
   cow_();
   if (ranChc){
      if (!iv_.hasElm(i)){
         cout<<"index access error at i="<<i<<endl;
         cpmerror(nameOf()&"::operator[]: write-index out of  range: i= "&
         cpm(i)&" iMin= "&cpm(iv_.b())&" iMax= "&cpm(iv_.e()));
      }
   }
   if (signal) cout<<" mutating [] called"<<endl;
   return (*p_)[iv_.ri(i)];
}

// using intentionally previously defined []-indexing
// for common messaging and safeness.
```

```
template <class T>
T const& V<T>::fir()const{ return (*this)[iv_.b()];}

template <class T>
T& V<T>::fir(){ return (*this)[iv_.b()];}

template <class T>
T const& V<T>::last()const{ return (*this)[iv_.e()];}

template <class T>
T& V<T>::last(){ return (*this)[iv_.e()];}

template <class T>
T const& V<T>::median()const{ return (*this)[iv_.mean()];}

template <class T>
T& V<T>::median(){ return (*this)[iv_.mean()];}

template <class T>
inline T const& V<T>::cui(Z i)const
{
   if (ranChcAlw){
      if (!iv_.hasElm(i)) cpmerror("index out of range");
   }
   return (*p_)[iv_.ri(i)];
}

template <class T>
inline T& V<T>::cui(Z i)
{
   cow_();
   if (ranChcAlw){
      if (!iv_.hasElm(i)) cpmerror("index out of range");
   }
   return (*p_)[iv_.ri(i)];
}

template <class T>
T const& V<T>::cyc(Z i)const
{
   return (*this)[iv_.cyc(i)];
}

template <class T>
T& V<T>::cyc(Z i)
{
   cow_();
   return (*this)[iv_.cyc(i)];
}
```

```
template <class T>
T const& V<T>::con(Z i)const
{
   if (sz_==0) return def_;
   return (*this)[iv_.con(i)];
}

template <class T>
T& V<T>::con(Z i)
{
   cow_();
   if (sz_==0){
      cpmerror("V<T>::con(Z i): i="&cpm(i)&
         " is no valid index in void array");
      return (*p_)[0]; // never happens
   }
   return (*this)[iv_.con(i)];
}

template <class T>
T const& V<T>::read(Z i, Outside mode)const
{
   if (iv_.hasElm(i)) return (*p_)[iv_.ri(i)];
   else{
      if (mode==DEFAULT) return def_; // reference to it can be returned
      else if (mode==CYCLIC) return cyc(i);
      else return con(i);
   }
}

template <class T>
T V<T>::operator()(Z i, Outside mode)const
{
   if (sz_== 0){
       return T();
   }
   else if (iv_.hasElm(i)){
      return (*p_)[iv_.ri(i)];
   }
   else if (mode==DEFAULT){
      return T();
   }
   else if (mode==CYCLIC){
      return cyc(i);
   }
   else  return con(i);
}

template <class T>
F<Z,T> V<T>::fnc(Outside mode)const
```

```
{
#ifdef CPM_Fn
   if (mode==DEFAULT) return F1<Z,V<T>,T>(*this)(fDef);
   else if (mode==CYCLIC) return F1<Z,V<T>,T>(*this)(fCyc);
   else return F1<Z,V<T>,T>(*this)(fCon);
#else
   if (mode==DEFAULT) return F<Z,T>(bind(fDef,_1,*this));
   else if (mode==CYCLIC) return F<Z,T>(bind(fCyc,_1,*this));
   else return F<Z,T>(bind(fCon,_1,*this));
#endif
}

// constructors
// 137 is a dummy argument

/*
template <class T>
V<T>::V(Z n):iv_(safeDim(n),1,137)
{
   ini_();
}
*/

template <class T>
V<T>::V(Z n, Begin bg):iv_(safeDim(n),0,137)
{
   ini_();
}

template <class T>
V<T>::V(Z n, T const& t, Begin bg):iv_(safeDim(n),0,137),
   sz_(iv_.car()),n_(toN(sz_))
{
   p_= spv(new vector<T>(n_,t));
}


template <class T>
V<T>::V(Z n, T const& t, Z first):iv_(safeDim(n),first,137),
   sz_(iv_.car()),n_(toN(sz_))
{
   p_= spv(new vector<T>(n_,t));
}

template <class T>
V<T>::V(Z n, F<Z,T> const& f):iv_(safeDim(n),1,137),sz_(iv_.car()),
n_(toN(sz_)),p_(n_)
{
   N i=0;
   for (Z j=iv_.b();i<n_;++i,++j) (*p_)[i] = f(j);
```

```
}

template <class T>
V<T>::V(IvZ const& iv):iv_(iv), sz_(iv_.car()), n_(toN(sz_))
{
   p_=spv(new vector<T>(n_));
}


template <class T>
V<T>::V(IvZ const& iv, T const& t):iv_(iv), sz_(iv_.car()),
   n_(toN(sz_))
{
   p_=spv(new vector<T>(sz_,t));
}


template <class T>
V<T>::V(IvZ const& iv, F<Z,T> const& f):iv_(iv),sz_(iv_.car()),
   n_(toN(sz_))
{
   p_=spv(new vector<T>(sz_));
   N i=0;
   for (Z j=iv_.b();i<n_;++i,++j) (*p_)[i] = f(j);
}

template <class T>
V<T>::V(std::initializer_list<T> il ):
iv_(il.size()), sz_(iv_.car()), n_(toN(sz_))
   // notice that, in view of the construction of iv_, the first
   // valid index of the constructed vector is 1
   {
      p_=spv(new vector<T>({il}));
   }

template <class T>
V<T>::V(Z first, std::vector<T> const& v):
iv_(v.size(),first,137), sz_(iv_.car()), n_(toN(sz_))
{
   p_= spv(new vector<T>(v));
}

template <class T>
V<T> V<T>::meet(IvZ const& iv)const
{
   IvZ ivRes=iv_.meet(iv);
   V<T> res(ivRes); // all components are T()
   for (Z i=res.b();i<=res.e();++i){
       if (iv.ni(i)) res[i]=(*this)[i];
   }
```

```
   return res;
}

template <class T>
V<T> V<T>::join(IvZ const& iv)const
{
   IvZ ivRes=iv_.join(iv);
   V<T> res(ivRes); // all components are T()
   for (Z i=res.b();i<=res.e();++i){
       if (iv_.ni(i)) res[i]=(*this)[i];
   }
   return res;
}

template <class T >
Z V<T>::find(T const& t, bool ordered)const
{
   Z n=dim();
   Z i0=b();
   Z iEx=i0-1;
   Z res{137};
   if (n==0){
      res = iEx;
   }
   else{
      if (ordered){
         auto q=std::lower_bound(p_->begin(),p_->end(),t);
         if (q==p_->end() || *q!=t){
            res = iEx;
         }
         else{
            res = i0+q-p_->begin();
         }
      }
      else{
         auto  q=std::find(p_->begin(),p_->end(),t);
         if (q == p_->end() || *q!=t){
            res = iEx;
         }
      }
   }
   return res;
}

// modified from function locate of Press et al.

template <class T>
Z V<T>::locAsc(T const& t)const
{
   if (sz_==0){
```

```
      cpmerror("V<T>::locAsc(T): array is void");
      return -137; // never happens
   }
   if (t<fir()) return b()-1;
   if (t>=last()) return e();
      // if sz_==1 we have fir()==last and then the two previous
      // conditions are an alternative: one of them holds and we
      // are ready. Thus now sz_>=2
   Z jl=0,ju=sz_;
   while (ju-jl>1){
      Z jm=(jl+ju)/2;
      if (t >= (*p_)[jm]) jl=jm; else ju=jm; // Press et al. have > here
   }
   return jl+b();
}

template <class T>
V<T> V<T>::app(V<T> const& h)const
{
   vector<T> pf;
   vector<T> pi=*p_;
   vector<T> hp=*h.p_;
   pf.reserve(sz_+h.size());
   pf.insert(pf.end(),pi.begin(),pi.end());
   pf.insert(pf.end(),hp.begin(),hp.end());
   return V<T>(iv_.b(),pf);
}

template <class T>
V<T> V<T>::app(T const& t)const
{
   vector<T> pf(*p_);
   pf.push_back(t);
   return V<T>(iv_.b(),pf);
}

template <class T>
V<T>& V<T>::prepend_(T const& t)
{
   cow_();
   p_->push_back(t);
   std::rotate(p_.rbegin(), p_.rbegin() + 1, p_.rend());
   iv_.n_();
   sz_++;
   return *this;
}

template <class T>
V<T> V<T>::rev()const
{
```

```
   if (sz_<2){
      return *this;
      // nothing to do if we have no or one component
   }
   else{
      vector<T> p{*p_};
      for (Z i=0,j=sz_-1; i<sz_; ++i,--j) p[i]=(*p_)[j];
      return V<T>(b(),p);
   }
}

template <class T>
V<T> V<T>::resize(Z newDim)const
{
   if (newDim<=0) return V<T>();
   vector<T> p(newDim);
   for (Z i=0;i<newDim;++i){
      p[i]= i<sz_ ? (*p_)[i] : T();
   }
   return V<T>(b(),p);
}

template <class T>
V<T> V<T>::eli(IvZ const& iv)const
{
   IvZ ie=iv&dom();
   if (ie.isVoid()) return *this ;
   else{
      V<B> vb(dom());
      for (Z i=vb.b();i<=vb.e();++i){
         vb.cui(i)=!ie.hasElm(i);
      }
      return select(vb);
   }
}

template <class T>
V<T> V<T>::eli1(V<T>& h, Z i)const
{
// Here we use natural counting of components so that
// the j-th component of h is h[j-1].
   Z nEli=h.dim();
   if (i<1) i=1;
   if (sz_==0){ // nothing eliminated, nothing left
      h=V<T>(0);
      return V<T>(0);
   }
   else if (i>sz_){ // nothing eliminated
      h=V<T>(0);
      return *this;
```

```
   }
   else{ // now i>=1 and i<=sz_ and sz_>=1
      // if true, we have sz_>=1
   // the i-th component is the first to be eliminated
   // so we have i-1 components of *this which are on the left-hand
   // side of the elimination area. These have to shine up in
   // the result vector to be returned
      Z nRes1=i-1;
   // So the maximum number of
   // components of *this that run the risk to get eliminated is
   // sz_-nRes1
      Z nEliRisk=sz_-nRes1;
      if (nEli>nEliRisk) nEli=nEliRisk;
      Z nRes2=sz_-nRes1-nEli;
      Z nRes=nRes1+nRes2;
      T* pRes=new T[toN(nRes)];
      T* pEli=new T[toN(nEli)];
      T* itRes=pRes;
      T* itEli=pEli;
      T* itp=p_;
      Z j=nRes1;
      while (j--) *itRes++=*itp++;
      j=nEli;
      while (j--) *itEli++=*itp++;
      j=nRes2;
      while (j--) *itRes++=*itp++; // for j==0 nothing done
      h=V<T>(nEli,pEli);
      return V<T>(nRes,pRes);
   }
}

template <class T>
V<T> V<T>::ins(Z i, T const& t)const
{
   Word loc("V<T> V<T>::ins(Z i, T const& t)const");
   vector<T> p{*p_};
   Z iLocal=i-b();
   auto it=p.begin()+iLocal;
   p.insert(it,t);
   return V(b(),p);
}

template <class T>
V<T>& V<T>::ins_(Z i, T const& t)
{
   cow_();
   Word loc("V<T> V<T>::ins(Z i, T const& t)const");
 //  vector<T> p{p_};
   Z iLocal=i-b();
   auto it=p_.begin()+iLocal;
```

```
   p_.insert(it,t);
   sz_++;
   iv_.n_();
   return *this;
}

template <class T>
V<T> V<T>::ins(Z ia, V<T> const& h)const
{
   Z mL=3;
   Word loc("V<T> V<T>::ins(Z i, V<T> const& h)const");
   CPM_MA
   vector<T> p{*p_};
   auto it=p.begin()+ia;
   p.insert(it,h.p_->begin(),h.p_->end());
   return V(b(),p);
}

template <class T>
T V<T>::in_(T const& t, bool reversed)
// safe logic by indexing, propably not utmost performance
{
   cow_();
   if (sz_==0) return T();
   Z i;
   T res;
   if (!reversed){
      res=(*p_)[sz_-1]; // last component of array
      for (i=sz_-1;i>0;i--){
         (*p_)[i]=(*p_)[i-1]; // causal processing: on the right-hand side we
         // evaluate only expressions which were not yet changed during
         // the loop
      }
      (*p_)[0]=t;
   }
   else{
      res=(*p_)[0]; // first component of array
      for (i=0;i<sz_-1;i++){
         (*p_)[i]=(*p_)[i+1]; // causal processing: on the right-hand side we
         // evaluate only expressions which were not yet changed during
         // the loop
      }
      (*p_)[sz_-1]=t;
   }
   return res;
}

// re-indexing

template <class T>
```

```
V<T> V<T>::compose(V<Z> const& w)const
{
   V<T> res(iv_);
   for (Z i=w.b();i<=w.e();i++){
     res[i]=operator()(w[i]);
   }
   return res;
}

template <class T>
V<T> V<T>::rot(Z s, Outside mode)const
{
   V<T> res(iv_);
   for (Z i=b();i<=e();++i) res[i]=operator()(i-s,mode);
   return res;
}

template <class T>
X2< V<T>, V<Z> > V<T>::condense(
   bool (*equiv)(const T&, const T&))const
// implementation based on function eclazz of the Numerical Receipes of
// Press et al.
{
   const Z mL=3;
   static Word loc("V<T>::condense()");
   CPM_MA
   Z n=dim();
   if (n<1){ // addition 2002-02-23
      CPM_MZ
      return X2< V<T>, V<Z> >(V<T>(0),V<Z>(0));
   }
   V<Z> res2(n);
   Z k,j;
   res2[1]=1;
   for (j=2;j<=n;j++) {
      res2[j]=j;
      for (k=1;k<=(j-1);k++) {
         res2[k]=res2[res2[k]];
         if ((*equiv)((*this)[j],(*this)[k])) res2[res2[res2[k]]]=j;
      }
   }
   for (j=1;j<=n;j++) res2[j]=res2[res2[j]];
   Z m=-1;
   Z rj;
   for (j=1;j<=n;j++){
      rj=res2[j];
      if (rj>m) m=rj;
   }
   V<Z> aux(m,0); // unfortunately the NR algorithm does not guarantee
      // that there are no gaps between the valid values of rj. Therfore
```

```
      // we find out the valid ones by an additional loop
   V<Z> found(m,0);
   for (j=1;j<=n;j++){
      rj=res2[j];
      if(found[rj]==0){
         found[rj]=1;
         aux[rj]=j;
      }
   }
   Z mAct=0;
   for (j=1;j<=m;j++) mAct+=found[j];
   V<T> res1(mAct);
   Z jAct=1;
   for (j=1;j<=m;j++){ // notice that aux[j] was initialized as 0
      if (aux[j]>0) res1[jAct++]=(*this)[aux[j]];
   }
   CPM_MZ
   return X2< V<T>,V<Z> >(res1,res2);
}


template <class T>
void V<T>::set_(T const& t)
{
   cow_();
   for (Z i=0;i<sz_;++i) (*p_)[i]=t;
}

template <class T>
V<T> V<T>::set(Z i, T const& t)const
{
   Z iC=i-1;
   // now iC is a 'C-pointer index'

   if (iC<0) return *this;   // nothing changed
   else if (iC<sz_){
      // vector can already hold the new value
      V<T> res(*this);
      res[iC]=t;
      return res;
   }
   else{
      // now we have to return an enlarged vector
      Z j, sz2=iC+1;
      V<T> res(sz2);
      res[iC]=t;
      T* it=res.p_;
      T* itp=p_;
      Z i=sz_;
      while (i--) *it++ = *itp++;
```

```
      return res;
   }
}

template <class T>
T V<T>::fAcc1(T (*f)(T const&), void (*acc)(T&, T const&))const
{
   T res=T();
   for (Z i=0;i<sz_;i++) acc(res,f((*p_)[i]));
   return res;
}

template <class T>
T V<T>::fAcc2(T (*f)(T const&, T const&), void (*acc)(T&, T const&),
   V<T> const& h)const
{
   T res=T();
   for (Z i=0;i<sz_;i++){ T xi=(*p_)[i]; T hi=h.li(i); acc(res,f(xi,hi));}
   return res;
}

template <class T>
template <class Y>
Y V<T>::fAcc3(Y (*f)(T const&, T const&), void (*acc)(Y&, Y const&),
      V<T> const& h )const
{
   Y res=Y();
   for (Z i=0;i<sz_;i++) acc(res,f((*p_)[i],h[(*p_)[i]]));
   return res;
}

template <class T>
V<T> V<T>::fAcc4(F<T2<T>,T> const& f)const
{
   Z d=dim();
   V< V<T> > aux(d,V<T>(d));
   for (Z i=0;i<d;++i){
      for (Z j=0;j<i;++j){
         T fij=f(T2<T>(li(i),li(j)));
         aux.li(i).li(j)=fij;
         aux.li(j).li(i)=-fij;
      }
   }
   T* p1=new T[toN(sz_)];
   for (Z i=0;i<sz_;i++){
      T res;
      for (Z j=0;j<sz_;j++) res+=aux.li(i).li(j);
      p1[i]=res;
   }
   return V<T>(dom(),p1,"");
```

```
}

namespace{

template <class T>
T fFunc5(T2<T> const& xij, F<R,R> const& dPot)
// This is intended to represent the force which particle i feels due to
// particle j being present.
{
   T eij=xij[1]-xij[2]; // eij points from j to i. Thus a positive
   // multiple of eij corresponds to a force on i which drives it away
   // from particle j. This such is the case of a repulsive potential.
   // This has dPot(r)/dr < 0 so that with the sign built into the formula
   // for the return value we in fact have the case of the positive
   // multiple we started with.
   R r=eij.nor_();
   return eij*(-dPot(r));
}

template <class T>
T fFunc6(T const& xi, F<R,R> const& dPot)
{
   T ei=xi; // ei points from the origin to i. Same situation as in fFunc5.
   R r=ei.nor_();
   return ei*(-dPot(r));
}

}

template <class T>
V<T> V<T>::fAcc5(F<R,R> const& dPot)const
{
#ifdef CPM_Fn
   F< T2<T>, T > f = F1< T2<T>, F<R,R>, T >(dPot)(fFunc5<T>);
#else
   F<T2<T>,T> f((std::bind(fFunc5<T>,_1,dPot)));
#endif

   Z d=dim(); // one could call fAcc4 here at the cost of an additional
      // function call. Here we ask for 'utmost efficiency'.
   V< V<T> > aux(d,V<T>(d));
   for (Z i=0;i<d;++i){
      for (Z j=0;j<i;++j){
         T fij=f(T2<T>(li(i),li(j)));
         aux.li(i).li(j)=fij;
         aux.li(j).li(i)=-fij;
      }
   }
   vector<T> p1(toN(sz_));
   for (Z i=0;i<sz_;i++){
```

```
      T res;
      for (Z j=0;j<sz_;j++) res+=aux.li(i).li(j);
      p1[i]=res;
   }
   return V<T>(dom(),p1);
}

template <class T>
V<T> V<T>::fAcc6(F<R,R> const& dPot)const
{
#ifdef CPM_Fn
   F<T,T> f = F1<T,F<R,R>,T>(dPot)(fFunc6<T>);
#else
   F<T,T> f(std::bind(fFunc6<T>,_1,dPot));
#endif
   vector<T> p1(toN(sz_));
   for (Z i=0;i<sz_;++i){
      p1[i]=f(li(i));
   }
   return V<T>(dom(),p1);
}

template <class T>
V<T> V<T>::fa(T (*f)(T const&))const
{
   vector<T> p=*p_;
   for (N i=0;i<n_;++i) p[i]=f((*p_)[i]);
   return V<T>(iv_,p);
}

//template <class T>
//V<T> V<T>::faa(T (*f)(T))const
//{
//   T* p1=new T[toN(sz_)];
//   T* it=p1;
//   T* itp=p_;
//   Z i=sz_;
//   while(i--) *it++ = f(*itp++);
//   return V<T>(dom(),p1,"");
//}

template <class T>
V<T> V<T>::fsf(std::function<T(T)> f)const
{
   vector<T> p=*p_;
   for (N i=0;i<n_;++i) p[i]=f((*p_)[i]);
   return V<T>(iv_,p);
}

template <class T>
```

```
V<T> V<T>::faa(T (*f)(T))const
{
   vector<T> p=*p_;
   for (N i=0;i<n_;++i) p[i]=f((*p_)[i]);
   return V<T>(iv_,p);
}

template <class T>
V<T> V<T>::fb(T (*f)(T const&, T const&), T const& t)const
{
  vector<T> p=*p_;
   for (N i=0;i<n_;++i) p[i]=f((*p_)[i],t);
   return V<T>(dom(),p);
}

template <class T>
template <class Y>
V<T> V<T>::fb2(T (*f)(T const&, Y const&), Y const& y)const
{
   vector<T> p=*p_;
   for (N i=0;i<n_;++i) p[i]=f((*p_)[i],y);
   return V<T>(dom(),p);
}

template <class T>
template <class Y>
void V<T>::fb2_(T (*f)(T const&, Y const&), Y const& y)
{
   cow_();
   for (N i=0;i<n_;++i) (*p_)[i]=f((*p_)[i],y);
}

template <class T>
void V<T>::fc_(T (*f)(T const&, T const&), T const& t)
{
   cow_(); for (N i=0;i<n_;++i) (*p_)[i]=f((*p_)[i],t);
}

template <class T>
void V<T>::fd(T (*f)(T const&, T const&), T& t)const
{

   for (N i=0;i<n_;++i) t=f((*p_)[i],t);
}

template <class T>
V<T> V<T>::fe(T (*f)(T const&, T const&), V<T> const& h)const
{
   if (!sameDom(h)) throw Error("V<T>::fe(): domain mismatch");
   vector<T> p=*p_;
```

```
   for (N i=0;i<n_;++i) p[i]=f((*p_)[i],(*h.p_)[i]);
   return V<T>(dom(),p);
}

template <class T>
template <class Y>
V<T> V<T>::fet(T (*f)(T const&, Y const&), V<Y> const& h)const
{
   if (dom()!=h.dom()) throw Error("V<T>::fet(): domain mismatch");
   vector<T> p=p_;
   for (N i=0;i<n_;++i) p[i]=f((*p_)[i],h.pi(i));
   return V<T>(dom(),p);
}

template <class T>
V<T> V<T>::fe2(T (*f)(T const&, T const&), V<T> const& h)const
{
   IvZ iv1{iv_};
   IvZ iv2{h.iv_};
   IvZ iv = iv1|iv2;
   V<T> res(iv);
   for (Z i=iv.b();i<=iv.e();++i){
      if (iv1.hasElm(i)&&iv2.hasElm(i)) res[i]=f((*this)[i],h[i]);
   }
   return res;
}

template <class T>
void V<T>::ff_(T (*f)(T const&, T const&), V<T> const& h, Z is)
{
   cow_();
   if (is<0) is=0;
   for (Z i=0; i+is<sz_; ++i){
      if (h.valInd(i)) (*p_)[i+is]=f((*p_)[i+is],(*h.p_)[i]);
   }
}

template <class T>
void V<T>::ffl_(T (*f)(T const&, T const&), V<T> const& h)
{
   cow_();
   for (N i=0; i<n_; ++i){
      (*p_)[i]=f((*p_)[i],(*h.p_)[i]);
   }
}

template <class T>
void V<T>::fa_(F<T,T> const& f, IvZ iv)
{
   cow_();
```

```
   for (Z i=b();i<=e();++i){
      if (iv.hasElm(i)) (*this)[i]=f((*this)[i]);
   }
}

template <class T>
void V<T>::fg_(T (*f)(T const&, T const&, T const&),
      V<T> const& h, T const& t)
{
   cpmassert(sameDom(h),"dimension mismatch in V::fg_(f,V,T)");
   cow_();
   for (Z i=0;i<sz_;++i) (*p_)[i]=f((*p_)[i],h.li(i),t);
}

template <class T >
void V<T>::fh_(T const& w_1, T const& w0, T const& w1, Outside mode)
{
   cow_();
   if (sz_<2) return;
   T v_1=read(b()-1,mode);
   T v0=(*p_)[0];
   T v1=(*p_)[1];
   Z k=0;
   while (k<sz_){
      (*p_)[k]=v_1*w_1+v0*w0+v1*w1;
      k++;
      v_1=v0;
      v0=v1;
      v1=(k < sz_-1 ? (*p_)[k+1] : read(e()+1,mode));
   }
}

template <class T >
template <class Y>
void V<T>::fht_(Y const& w_1, Y const& w0, Y const& w1, Outside mode)
{
   cow_();
   if (sz_<2) return;
   T v_1=read(b()-1,mode);
   T v0=(*p_)[0];
   T v1=(*p_)[1];
   Z k=0;
   while (k<sz_){
      (*p_)[k]=v_1*w_1+v0*w0+v1*w1;
      k++;
      v_1=v0;
      v0=v1;
      v1=(k < sz_-1 ? (*p_)[k+1] : read(e()+1,mode));
   }
}
```

```
template <class T >
template <class Y>
V<T> V<T>::fh(T (*f)(T const&, T const&, T const&, Y const&, Z),
      Y const& y, Outside mode)const
{
   N n=nc();
   if (n<2) return *this;
   V<T> q=*this;
   V<T> p=*this;
   for (N i=0; i<n;++i){
      p.pi(i)=f(q.pr(i-1,mode),q.pr(i,mode),q.pr(i+1,mode),y,i);
   }
   return p;
}

template <class T >
template <class Y>
V<Y> V<T>::fi(Y (*f)(T const&, T const&), void (*acc)(Y&, Y const&))const
{
   Z i,j;
   Y* q=new Y[toN(sz_)];
   for (i=0;i<sz_;i++){
      Y yi=Y();
      for (j=0;j<sz_;j++){
         acc(yi,f((*p_)[i],(*p_)[j]));
      }
      q[i]=yi;
   }
   return V<Y>(dom(),q);
}

template <class T >
template <class Y>
V<Y> V<T>::fj(Y (*f)(T const&, T const&), void (*acc)(Y&, Y const&))const
{
   Z i,j;
   Y* q=new Y[toN(sz_)];
   for (i=0;i<sz_;i++) q[i]=Y();
   for (i=0;i<sz_;i++){
      for (j=0;j<i;j++){
         Y fij=f((*p_)[i],(*p_)[j]);
         acc(q[i],fij);
         acc(q[j],-fij);
      }
   }
   return V<Y>(dom(),q);
}

template <class T >
```

```
V<T> V<T>::select(V<B> const& s)const
{
   vector<T> pRes;
   for (Z i=b();i<=e();++i){
      bool keep{true};
      if (s.valInd(i)&&s[i]==false) keep=false;
      if (keep) pRes.push_back((*this)[i]);
   }
   return V<T>(b(),pRes);
}

template <class T >
V<IvZ> V<T>::valOn(F<T,bool> const& f)const
{
   Z mL=3;
   static Word loc("V<T>::valOn(F<T,bool>)");
   CPM_MA
   V<IvZ> res;
   bool yetFoundTrue=false;
   Z firstTrue=0, firstFalseAfterTrue=0;
   for (Z i=b();i<=e();++i){
      bool val=f((*this)[i]);
      if (val){ // we found true
         if (!yetFoundTrue){ // then start a interval of validity
            yetFoundTrue=true;
            firstTrue=i;
         }
         else{ // normally nothing to do
            // but if we are at the end of the array the
            // last pending truth interval has to be
            // considered as finished and has to be added
            if (i==e()){
               firstFalseAfterTrue=n();
               IvZ ivAct(firstTrue,firstFalseAfterTrue-1);
               res&=ivAct; // appending
               // nothing else to do since we are finished
               CPM_MZ
               return res;
            }
         }
      }
      else{ // we found false
         if (yetFoundTrue){ // then i is the terminator
            // of a truth interval
            firstFalseAfterTrue=i;
            IvZ ivAct(firstTrue,firstFalseAfterTrue-1);
            res&=ivAct;
            yetFoundTrue=false;
         }
      }
```

```
   }
   CPM_MZ
   return res;
}

template <class T>
bool V<T>::prnOn(ostream& str)const
{
   Z mL=3;
   Word loc=nameOf()&"::prnOn(...)";
   CPM_MA
   cpmwt((nameOf()&" begin").str());
   Root<IvZ>(iv_).prnOn(str);
   if (iv_.car()==0){
      Word w=loc&" no components to print ";
      cpmcerr<<w<<endl;
      cpmcerr<<" iv_.b() = "<<iv_.b()<<endl;
      goto label;
   }
   for (Z i=b();i<=e();i++){
      if (CpmRoot::wrtTit){
         Word wi("// i="); // added 2012-01-19
         // same as in S<>
         wi&=cpm(i);
         bool bi=wi.prnOn(str);
         cpmassert(bi==true,loc);
      }
      if (!Root<T>((*this)[i]).prnOn(str)){
         cpmwarning("failed to write component indexed "&cpm(i));
         cpmwarning("index range is from "&cpm(b())&" to "&cpm(e()));
         CPM_MZ
         return false;
      }
   }
label:
   cpmwt((nameOf()&" end").str());
      // for large sz_ it would be difficult to find the
      // end of the vector data if these are written to a file
      // and a human reader wants to inspect them
   CPM_MZ
   return true;
}

template <class T>
bool V<T>::scanFrom(istream& str)
{
   Z mL=3;
   Word loc=nameOf()&"::scanFrom(...)";
   CPM_MA
   cow_();
```

```
   Root<IvZ> ivIn;
   bool suc = ivIn.scanFrom(str);
   if (!suc){
      cpmwarning(loc&": can't read IvZ");
      CPM_MZ
      return false;
   }
   IvZ iv=ivIn();
   Z n=iv.car();
   cpmmessage(mL,"dimension read as "&cpm(n));
   if (n>dimMax) cpmwarning(loc&": n>dimMax");
   V<T> res(iv);
   Root<T> riIn;
   for (Z i=res.b();i<=res.e();i++){
      suc = riIn.scanFrom(str);
      if (!suc){
         Word mes="failed to read component indexed "&cpm(i)&
             " index range is from "&cpm(res.b())&" to "&cpm(res.e());
         cpmwarning(mes);
         CPM_MZ
         return false;
      }
      res.cui(i) = riIn();
   }
   *this=res;
   CPM_MZ
   return true;
}

template <class T >
Z V<T>::com(V<T> const& s)const
// short vectors < longer vectors
{
   Z d1=dim(), d2=s.dim();
   if (d1<d2) return 1;
   else if (d1>d2) return -1;
   else{
       for (Z i=0;i<d1;i++){
         Z ci=Root<T>((*p_)[i]).com(s.li(i));
         if (ci!=0) return ci;
      }
      return 0;
   }
}
/************************** V<bool> *********************************/
// since std::vector<bool> has a quite irregular special definition we get
// a lot of compilation errors if we use existing C+- code the
// instantiation V<bool> while V<B> works fine.
/*
template<>
```

```
class V<bool>{
   IvZ iv_;
   std::vector<CpmRoot::B> p_;
public:
   B const& cui(Z i)const{return (*p_)[iv_.ri(i)];}
   B& cui(Z i){return (*p_)[iv_.ri(i)];}
   B const& operator[](Z i)const{return (*p_)[iv_.ri(i)];};
   B& operator[](Z i){return (*p_)[iv_.ri(i)];};
};
*/

/*
B const& CpmArrays::V<bool>::operator[](Z i)const
{
   if (ranChcAlw){
      if (!iv_.hasElm(i)) cpmerror("index out of range");
   }
   return (*p_)[iv_.ri(i)];
}


B& CpmArrays::V<bool>::operator[](Z i)
{
   if (ranChcAlw){
      if (!iv_.hasElm(i)) cpmerror("index out of range");
   }
   return (*p_)[iv_.ri(i)];
}
*/


/************************ iterated templates *************************/
// describing multi-indexed quantities
// Notice that these all have automatically the member functions of V
// defined!

#define CPM_V1 V<T>
#define CPM_V2 V<V<T> >
#define CPM_V3 V<V<V<T> > >
#define CPM_V4 V<V<V<V<T> > > >

/************************** class VV *********************************/
template <class T>
class VV: public CPM_V2{ // matrices

   typedef CPM_V2 Base;

public:

// constructors
```

```
   VV(Z d1, Z d2):CPM_V2(d1,CPM_V1(d2)){}
   VV(Z d1, Z d2, T const& t):CPM_V2(d1, CPM_V1(d2,t)){}
      // all components are equal to t
   VV(void):CPM_V2(){}
   VV(CPM_V2 const& x):CPM_V2(x){}

   V<T> lin()const;
      // 'linear version of matrix'
      // by appending all rows

// dimension access
   Z dim1()const{ return Base::dim();}
   Z dim2()const{ return (*this)[1].dim();}

   virtual Z size()const // virtual in base V<...>
      { return dim1()==0 ? 0 : dim1()*dim2();}

// component access
   const T& operator()(Z i, Z j)const
      // returns T() for out of range indexes
   { return Base::read(i).read(j);}

   T& operator()(Z i, Z j)
   { return (*this)[i][j];}

   VV<T> trn()const
      //: transpose
      // Returns the transposed matrix
   {
      VV<T> res(dim2(),dim1());
      for (Z i=1;i<=dim1();i++){
         for (Z j=1;j<=dim2();++j){
            res[j][i] = (*this)[i][j];
         }
      }
      return res;
   }

   template <class Y>
   V<Y> each(void (*f)(T const&, Y&))const;
      //: each
      // collecting the application of f on every pixel in a linear array

   template <class Y>
   VV<Y> operator()(Y (*f)(T const&))const;
      //: operator()
      // creating a matrix with a transformed value range
};
```

```
template <class T>
template <class Y>
V<Y> VV<T>::each(void (*f)(T const&, Y&))const
{
   Z m1=dim1(),m2=dim2(),k=1,i1,i2;
   V<Y> res(m1*m2);
   for (i1=1;i1<=m1;i1++){
      for (i2=1;i2<=m2;i2++){
         f((*this)[i1][i2],res[k++]);
      }
   }
   return res;
}

template <class T>
template <class Y>
VV<Y> VV<T>::operator()(Y (*f)(T const&))const
{
   Z m=dim1(),n=dim2(),i,j;
   VV<Y> res(m,n);
   for (i=1;i<=m;i++){
      for (j=1;j<=n;j++){
         res[i][j]=f((*this)[i][j]);
      }
   }
   return res;
}

template <class T>
V<T> VV<T>::lin()const
{
   if (dim1()==0) return V<T>(0);
   else{
      V<T> res=(*this)[1];
      for (Z i=2;i<=dim1();i++) res&=(*this)[i];
      return res;
   }
}

/************************ class VVV***********************************/
// tensors of rank 3

template <class T>
class VVV: public CPM_V3{ // tensors of rank 3

      typedef CPM_V3 Base;

public:

// constructors
```

```
   VVV(Z d1, Z d2, Z d3, T  const& t):CPM_V3(d1,CPM_V2(d2,CPM_V1(d3,t))){}
   VVV(Z d1, Z d2, Z d3):CPM_V3(d1,CPM_V2(d2,CPM_V1(d3))){}
   VVV(void):CPM_V3(){}
   VVV(CPM_V3 const& x):CPM_V3(x){}

// dimension access
   Z dim1()const{ return Base::dim();}
   Z dim2()const{ return (*this)[1].dim();}
   Z dim3()const{ return (*this)[1][1].dim();}

   virtual Z size()const; // virtual in base V<...>

// component access

   const T & operator()(Z i, Z j, Z k)const
   // returns T() for out of range indexes
   { return Base::read(i).read(j).read(k);}

   T & operator()(Z i, Z j, Z k)
   { return (*this)[i][j][k];}
};

template <class T>
Z VVV<T>::size()const
{
   Z d1=dim1();
   if (d1==0) return d1;
   else{
      Z d2=dim2();
      if (d2==0) return d2;
      else{
         Z d3=dim3();
         if (d3==0) return d3;
         else return d1*d2*d3;
      }
   }
}

/********************** class VVVV ***********************************/
template <class T>
class VVVV: public CPM_V4{ // tensors of rank 4

      typedef CPM_V4 Base;

public:

// constructors

   VVVV(Z d1, Z d2, Z d3, Z d4, T const& t):
```

```
   CPM_V4(d1,CPM_V3(d2,CPM_V2(d3,CPM_V1(d4,t)))){}
   VVVV(Z d1, Z d2, Z d3, Z d4):
   CPM_V4(d1,CPM_V3(d2,CPM_V2(d3,CPM_V1(d4)))){}
   VVVV(void):CPM_V4(){}
   VVVV(CPM_V4 const& x):CPM_V4(x){}

// dimension access
   Z dim1()const{ return Base::dim();}
   Z dim2()const{ return (*this)[1].dim();}
   Z dim3()const{ return (*this)[1][1].dim();}
   Z dim4()const{ return (*this)[1][1][1].dim();}

   virtual Z size()const; // virtual in base V<...>

// component access

   T const& operator()(Z i, Z j, Z k, Z l)const
  // returns T() for out of range indexes
   { return Base::read(i).read(j).read(k).read(l);}

   T& operator()(Z i, Z j, Z k, Z l)
   { return (*this)[i][j][k][l];}
};

template <class T>
Z VVVV<T>::size()const
{
   Z d1=dim1();
   if (d1==0) return d1;
   else{
      Z d2=dim2();
      if (d2==0) return d2;
      else{
         Z d3=dim3();
         if (d3==0) return d3;
         else{
            Z d4=dim4();
            if (d4==0) return d4;
            else return d1*d2*d3*d4;
         }
      }
   }
}
#undef CPM_V4
#undef CPM_V3
#undef CPM_V2
#undef CPM_V1

}  // namespace CpmArrays
```

```
namespace CpmRoot{
// This is a pattern for partial specializations --- to which
// the specializations to class templates belong --- of the
// IO class template started in cpmnumbers.h and the
// Name class template started in cpmword.h.
// A corresponding definition is needed for all non-C+- classes which
// shall be used as template arguments of C+- containers and
// are expected to provide then the same functionality as
// corresponding C+- classes.

   template<class T>
   class IO< std::vector<T> >{ // partial specialization
    public:
      IO(){}
      bool o(std::vector<T> const& v, ostream& str)const
      { return CpmArrays::V<T>(v).prnOn(str);}
      bool i(std::vector<T>& v, istream& str)const
      {
         CpmArrays::V<T> vl;
         bool res=vl.scanFrom(str);
         v=vl.std();
         return res;
      }
   };

#if defined(CPM_NAMEOF)
   template<class T>
   class Name< std::vector<T> >{ // partial specialization
   public:
      Name(){}
      Word operator()(std::vector<T> const& vt )const
      { return Word("std::vector<"&CpmRoot::Name<T>()(T())&">");}
   };
   template<class T>
   class Name< std::set<T> >{ // partial specialization
   public:
      Name(){}
      Word operator()(std::set<T> const& vt )const
      { return Word("std::set<"&CpmRoot::Name<T>()(T())&">");}
   };
#endif
}
#endif
```

41 cpmv.cpp

```
//? cpmv.cpp
//? Status of work 2023-10-20.
//?
//?  ...


#include <cpmv.h>

using CpmRoot::Z;
using CpmRoot::Word;
using CpmArrays::V;
using CpmArrays::IvZ;

Z CpmArrays::dimMax=1000000000;
Z CpmArrays::firInd=1; // first index of arrays. Means to let C+-
   // cooperate with std containers.
bool CpmArrays::ranChc=true;
bool CpmArrays::ranChcAlw=false;
bool CpmArrays::signal=false;

   // if this is true some screen messages are enabled
   // e.g.
   // if (signal) cout<<"lazy ~V called, p_ = "<<p_<<endl;

void CpmArrays::setDimMax(Z n)
{
   if (n<0)
      cpmerror(cpmwrite(n)&" not OK as value for CpmArrays::dimMax");
   dimMax=n;
}

Z CpmArrays::makeIndValNotMember(Z i, IvZ iv, Outside mode)
{
   if (iv.hasElm(i)) return i;
   if (mode==CYCLIC) return iv.cyc(i);
   if (mode==CONSTANT) return iv.con(i);
   if (mode==DEFAULT){ cpmwarning("value DEFAULT of Outside not allowed here");return -137;}
   if (mode==ZERO){ cpmwarning("value ZERO of Outside not allowed here"); return -137;}
   else { cpmwarning("undefined value of Outside"); return -137;}
}
      //: make index valid

Z CpmArrays::safeDim(Z n)
{
   if (n<0 || n>dimMax)
      cpmerror(cpmwrite(n)&" not OK as dimension of CpmArrays::V<>");
   return n;
```

```
}

V<Z> CpmArrays::IvZtoVofZ(IvZ const& iv)
{
   Z n=iv.car();
   V<Z> res(n);
   if (n==0) return res;
   Z i,val=iv.inf();
   for (i=res.b();i<=res.e();++i) res[i]=val++;
   return res;
}

V<Z> CpmArrays::VofIvZtoVofZ(V<IvZ> const& viv)
{
   V<Z> res(0);
   for (Z i=viv.b();i<=viv.e();++i) res&=IvZtoVofZ(viv[i]);
   return res;
}

V<Word> CpmArrays::comLine(int argc, char* argv[])
// see BS4, p.254 function
//    vector<string> arguments(int argc, char* argv[])
// for the corresponding construct within namespace std.
{
   V<Word> res(argc);
   for (Z i=res.b();i<=res.e();i++) res[i]=argv[i-1];
   return res;
}
```

42 cpmva.h

```
//? cpmva.h
//? Status of work 2023-10-20.
//?
//?  ...


#ifndef CPM_VA_H_
#define CPM_VA_H_
/*
   Purpose: Defining a array class Va<T> which exploits the assumption
   that T supports arithmetics. See cpmv.h

   Recent history: 2012-01-22 function tim_(...) and mean() added
*/
#include <cpmvo.h>
#include <cpmtypes.h>
#include <type_traits>

////////////////////// class Va<> ////////////////////////////

namespace CpmArrays{

   using CpmRoot::Z;
   using CpmRoot::R;
   using CpmRoot::Word;
   using CpmRoot::Root;
   using CpmRootX::inf;

template <typename T>
   // We assume that T provides (explicitely or implicitely)
   // copy constructor, and assignement
   // or that T is a built-in type.
   // T > T, T < T, T += T, -T, T *= T, T /=T, ostream << T, istream >> T

class Va: public Vo<T>{ // version of Vo with arithmetic operations

   typedef Va<T> Type;
   typedef T ScalarType;
   typedef Vo<T> Base;
   typedef V<T> Source;

public:

//Va():Vo<T>(){}
//Va()=default;

explicit   Va(Z n=0):Vo<T>(n){}
```

```
explicit Va(IvZ ivz):Vo<T>(ivz){}

   Va(IvZ ivz, T const& t):Vo<T>(ivz,t){}

// constructors from explicit lists
explicit  Va(std::initializer_list<T> il ):Vo<T>(il){}
   // requires C++11
      // constructors from explicit lists such as
      // Va<Z> v{1,2,4,8};

   Va(Z n,T const& t):Vo<T>(n,t){}
      // initializes all n components with t

   Va(V<T> const& h):Vo<T>(h){}
   // 'down-cast'-constructor

   Va(Vo<T> const& h):Vo<T>(h){}
   // 'down-cast'-constructor
 //  Va(Va<T> const& h):Vo<T>(h){}
   // copy-constructor

/*
#if defined(CPM_USE_MOVE)
   Va(Va<T> && h)
   {
      Source::iv_=h.iv_;
      Source::sz_=h.sz_;
      Source::p_=h.p_;
      cout<<"move constructor Va"<<endl;
      h.iv_=IvZ();
      h.sz_=0;
      h.p_=nullptr;
   }
#endif
*/

   T sum(void)const;
      // returns the sum over all components

   T sumPrd(V<T> const& v)const;
      // returns the sum over (*this)[i]*v[i]. I.e.
      // the scalar product without a conjugation operation for the first
      // factor

   R frac(T const& t)const;
      // returns nt/dim() where nt=card{ j | (*this)[j]==t}
      // For dim()==0 we return 0.

   R frac(const V<T>& v)const;
```

```
      // returns nt/dim() where nt=card{ j | (*this)[j]==v[k] for some k}
      // For dim()==0 or v.dim()==0 we return 0.

   virtual Word nameOf()const{
      Word wi="Va<";
      return wi&CpmRoot::Name<T>()(T())&">";
   }

   virtual V<T>* clone(void)const{ return new Va(*this);}

   // discrete Laplacian
   Va<T> laplace(Z os)const; // os: outside but slightly different from
      // enum Outside

   template <class S>
   Va<T>& tim_(S const& s);
      //: times
      // using '*=' instead of 'tim_' causes confusion with existing calls
      // to *=.
      // Needed e.g. to multiply instances of Va<R3> by R as in
      // Va<R3> v=...;
      // v.tim_(3.14); // v*=3.14 would look more natural
  // template <class S>
  // requires !std::is_same<S,T>
  // Va<T>& operator*=(S const& s){ return tim_(s);}


   template <class S>
   Va<T> tim(S const& s)const;
      //: times
      // Non-mutating form of tim_

//   template <class S>
//   Va<T>& operator *=(S const& s);

//   template <class S>
//   Va<T> operator *(S const& s)const;

  // template <class S>
  // requires(typeid(S)!=typeid(T))
   //Va<T> operator *(S const& s)const{ return tim(s);}

   T mean(void)const{ return sum()*cpminv(R(Base::dim()));}
      // returns the sum over all components, divided by
      // their number. This makes sense only if a multiplication
      // T*R is defined

   CPM_SUM_M
   CPM_PRODUCT_M
   CPM_DIFFERENCE
```

```
   CPM_DIVISION
   CPM_SCALAR_M
   CPM_DOT_PRODUCT_LEAN
      // does no longer contain a definition of dis. This comes now from
      // V<T>
   CPM_CONJ
   CPM_NORMALIZE
   CPM_IO

private:
   static T sumFunc(T const& t1, T const& t2){ return t1+t2;}
   static T prodFunc(T const& t1, T const& t2){ return t1*t2;}
   //static T prodRFunc(T const& t1, R const& r){ return t1*r;}
   static T negFunc(T const& t) {  return -t;}
   static T idFunc(T const& t) {  return t;}
   static T invFunc(T const& t) {  return Root<T>(t).inv();}
   static T zeroFunc(T const& t)
      {  return Root<T>(t).net(0);}
   static T unitFunc(T const& t)
      {  return Root<T>(t).net(1);}
   static T conjFunc(T const& t){ return Root<T>(t).con();}
   static T dotFunc(T const& t1, T const& t2)
     { return Root<T>(t1).con()*t2;}
   static void accFunc(T& z, T const& zAdd){ z+=zAdd;}
};

// class functions:

template <class T>
T Va<T>::sum(void)const
{
   return Base::fAcc1(idFunc,accFunc);
}

template <class T>
T Va<T>::operator | (Va<T> const& h)const
{
   return Base::fAcc2(dotFunc,accFunc,h);
}

template <class T>
T Va<T>::sumPrd(V<T> const& h)const
{
   return Base::fAcc2(prodFunc,accFunc,h);
}

template <class T>
CpmRoot::R Va<T>::frac(T const& t)const
{
   Z n=Base::dim();
```

```
   if (n==0) return 0;
   Z sum=0;
   for (Z i=1;i<=n;i++) if ((*this)[i]==t) sum++;
   return ((CpmRoot::R)(sum))/n;
}

template <class T>
CpmRoot::R Va<T>::frac(const V<T>& v)const
{
   Z n=Base::dim();
   if (n==0) return 0;
   Z nv=v.dim();
   if (nv==0) return 0;
   Z i,j,sum=0;
   for (i=1;i<=n;i++){
      T ti=(*this)[i];
      for (j=1;j<=nv;j++){
         if (ti==v[j]) sum++;
      }
   }
   return ((CpmRoot::R)(sum))/n;
}

// main task

template <class T>
Va<T> Va<T>::neg(void)const
{
   //cout<<"Va::neg called"<<endl;
   V<T> v_Res=Base::fa(negFunc);
   return Va<T>(v_Res);
}

template <class T>
Va<T> Va<T>::inv(void)const
{
   V<T> v_Res=Base::fa(invFunc);
   return Va<T>(v_Res);
}

template <class T>
Va<T> Va<T>::con(void)const
{
   V<T> v_Res=Base::fa(conjFunc);
   return Va<T>(v_Res);
}

template <class T>
Va<T> Va<T>::net(Z i)const
{
```

```
   V<T> v_Res= i==1 ? Base::fa(unitFunc) : Base::fa(zeroFunc);
   return Va<T>(v_Res);
}

template <class T>
Va<T> Va<T>::laplace(Z os)const
{
   N d=Base::p_->size();
   N dm=d-1;
   using namespace CpmRoot;
  // using Base::p_;
   Va<T> res(*this);
   for (N i=1;i<dm;++i) res.pi(i)= (*Base::p_)[i]*2_R-(*Base::p_)[i-1]-
      (*Base::p_)[i+1];
   if (os==0){ // toroidal biotope
      res.pi(0)=(*Base::p_)[0]*2_R-(*Base::p_)[dm]-(*Base::p_)[1];
      res.pi(dm)=(*Base::p_)[dm]*2_R-(*Base::p_)[dm-1]-(*Base::p_)[0];
   }
   else if (os==1){ // assuming (*Base::p_)[-1] == (*Base::p_)[0] and
      // (*Base::p_)[d]=(*Base::p_)[dm] for the values of the wavefunction at
      // the sites adjacent to the bioptope's ends.
      res.pi(0)=(*Base::p_)[0]-(*Base::p_)[1];
      res.pi(dm)=(*Base::p_)[dm]-(*Base::p_)[dm-1];

   }
   else if (os==2){ // assuming enforced value 0 of the wavefunction
      // outside the biotope (i.e. (*Base::p_)[-1]=0, (*Base::p_)[d]=0
      res.pi(0)=(*Base::p_)[0]*2_R-(*Base::p_)[1];
      res.pi(dm)=(*Base::p_)[dm]*2_R-(*Base::p_)[dm-1];
   }
   else if (os==3){ // this is the most natural discrete model for
   // potential barriers of infinite height at the ends of the
   // biotope
      res.pi(0)=T();
      res.pi(dm)=T();
   }
   else{
      cpmwarning("never come to here");
   }
   return res;
}

template <class T>
Va<T>& Va<T>::operator +=(const Va<T>& s)
{
   Base::ffl_(sumFunc,s);
   return *this;
}

template <class T>
```

```
Va<T>& Va<T>::operator +=(T const& t)
{
   Base::fc_(sumFunc,t);
   return *this;
}

template <class T>
Va<T>& Va<T>::operator *=(const Va<T>& s)
{
   Base::ffl_(prodFunc,s);
   return *this;
}

template <class T>
Va<T>& Va<T>::operator *=(T const& t)
{
   Base::fc_(prodFunc,t);
   return *this;
}

template <class T>
template <class S>
Va<T>& Va<T>::tim_(S const& s)
{
   for (Z i=Base::b();i<=Base::e();++i) Base::cui(i)*=s;
   return *this;
}

//template <class T>
//template <class S>
//Va<T>& Va<T>::operator *=(S const& s)
//{
//   return tim_(s);
//}

template <class T>
template <class S>
Va<T> Va<T>::tim(S const& s)const
{
   Va<T> res=*this;
   return res.tim_(s);
}

//template <class T>
//template <class S>
//Va<T> Va<T>::operator *(S const& s)const
//{
//   return tim(s);
//}
```

```
template <class T>
bool Va<T>::prnOn(ostream& str)const
{
   return Base::prnOn(str);
}

template <class T>
bool Va<T>::scanFrom(istream& str)
{
   return Base::scanFrom(str);
}

////////////////////////// iterated templates ////////////////////////////
// describing multi-indexed quantities
////////////////////////////// class VVa ////////////////////////////////
// matrices without matrix product
// tensors of rank 2
// We derive VVa<T> from VVo<T> and not from Va<Va<T> >. In the
// latter case Va<T> would have to be the scalar type for VVa<T>
// which is not the case. I tried this 2005-08-19 but came deep
// 'in the woods'. Since thus the arithmetics interface is not
// yet fixed by the definition, we are free to define it anew in
// a more lean way as that of Va. Especially we avoid to
// define friend functions. Also simply taking over the interface
// of Va would not be adequate from the point of view of
// implementation efficiency.
// The implementation of VVa is such that it can be carried over to
// higher numbers of indexes iteratively

template <class T>
class VVa: public VVo<T>{ // version of VVo with arithmetic operations

   typedef VVo<T> Base;
   typedef VVa<T> Type;
   typedef T ScalarType;

public:
   CPM_LIN
      // notice that function con() declared here will be
      // implemented as conjugation of the matrix elements
      // only and not as accompanied by a matrix
      // transposition. The present implementation is natural
      // for an interpretation of VVa's as T-valued areal
      // distributions (as in images or wave functions)
      // Unlike the case for Va<T>, we do not include CPM_IO
      // in the interface of VVa<T>
// constructors
   VVa(Z d1, Z d2):Base(d1,d2){}
   VVa(Z d1, Z d2, T const& t):Base(d1,d2,t){}
      // all components are equal to t
```

```
   VVa():Base(){};
   VVa(V< V<T> >  const& x):Base(x){}
   VVa(VV<T> const& x):Base(x){}
   VVa(VVo<T> const& x):Base(x){}
};

template <class T>
VVa<T> VVa<T>::operator+(VVa<T> const& s)const
{
   Z d1=Base::dim(),d2=s.dim();
   Z d=inf<Z>(d1,d2);
   VVa<T> res(d,0);
   for (Z i=1;i<=d;i++){
      Va<T> ti=(*this)[i];
      Va<T> si=s[i];
      res[i]=ti+si;
   }
   return res;
}

template <class T>
VVa<T> VVa<T>::operator-(VVa<T> const& s)const
{
   Z d1=Base::dim(),d2=s.dim();
   Z d=inf<Z>(d1,d2);
   VVa<T> res(d,0);
   for (Z i=1;i<=d;i++){
      Va<T> ti=(*this)[i];
      Va<T> si=s[i];
      res[i]=ti-si;
   }
   return res;
}

template <class T>
VVa<T> VVa<T>::operator*(T const& r)const
{
   Z d=Base::dim();
   VVa<T> res(d,0);
   for (Z i=1;i<=d;i++){
      Va<T> ti=(*this)[i];
      res[i]=ti*r;
   }
   return res;
}

template <class T>
VVa<T> VVa<T>::operator-(void)const
{
   Z d=Base::dim();
```

```
   VVa<T> res(d,0);
   for (Z i=1;i<=d;i++){
      Va<T> ti=(*this)[i];
      res[i]=-ti;
   }
   return res;
}

template <class T>
VVa<T> VVa<T>::con(void)const
{
   Z d=Base::dim();
   VVa<T> res(d,0);
   for (Z i=1;i<=d;i++){
      Va<T> ti=(*this)[i];
      res[i]=ti.con();
   }
   return res;
}

template <class T>
T VVa<T>::operator|(VVa<T> const& s)const
{
   Z d1=Base::dim(),d2=s.dim();
   Z d=inf<Z>(d1,d2);
   T res=T();
   for (Z i=1;i<=d;i++){
      Va<T> ti=(*this)[i];
      Va<T> si=s[i];
      res+=(ti|si);
   }
   return res;
}

////////////////////////// class VVVa ////////////////////////////////////
// tensors of rank 3

template <class T>
class VVVa: public VVVo<T>{// version of VVVo with arithmetic operations

   typedef VVVo<T> Base;
   typedef VVVa<T> Type;
   typedef T ScalarType;

public:
   CPM_LIN
      // notice that function con() declared here will be
      // implemented as conjugation of the components.
      // The present implementation is natural
      // for an interpretation of VVVa's as T-valued spatial
```

```
      // distributions (as in wave functions).
      // Unlike the case for Va<T>, we do not include CPM_IO
      // in the interface of VVVa<T>
// constructors
   VVVa(Z d1,Z d2,Z d3):Base(d1,d2,d3){}
   VVVa(Z d1,Z d2,Z d3,T const& t):Base(d1,d2,d3,t){}
      // all components are equal to t
   VVVa(void):Base(){}
   VVVa( V< V< V<T> > >  const& x):Base(x){}
   VVVa(VVV<T> const& x):Base(x){}
   VVVa(VVVo<T> const& x):Base(x){}
};

template <class T>
VVVa<T> VVVa<T>::operator+(VVVa<T> const& s)const
{
   Z d1=Base::dim(),d2=s.dim();
   Z d=inf<Z>(d1,d2);
   VVVa<T> res(d,0,0);
   for (Z i=1;i<=d;i++){
      VVa<T> ti=(*this)[i];
      VVa<T> si=s[i];
      res[i]=ti+si;
   }
   return res;
}

template <class T>
VVVa<T> VVVa<T>::operator-(VVVa<T> const& s)const
{
   Z d1=Base::dim(),d2=s.dim();
   Z d=inf<Z>(d1,d2);
   VVVa<T> res(d,0,0);
   for (Z i=1;i<=d;i++){
      VVa<T> ti=(*this)[i];
      VVa<T> si=s[i];
      res[i]=ti-si;
   }
   return res;
}

template <class T>
VVVa<T> VVVa<T>::operator*(T const& r)const
{
   Z d=Base::dim();
   VVVa<T> res(d,0,0);
   for (Z i=1;i<=d;i++){
      VVa<T> ti=(*this)[i];
      res[i]=ti*r;
   }
```

```
   return res;
}

template <class T>
VVVa<T> VVVa<T>::operator-(void)const
{
   Z d=Base::dim();
   VVVa<T> res(d,0,0);
   for (Z i=1;i<=d;i++){
      VVa<T> ti=(*this)[i];
      res[i]=-ti;
   }
   return res;
}

template <class T>
VVVa<T> VVVa<T>::con(void)const
{
   Z d=Base::dim();
   VVVa<T> res(d,0,0);
   for (Z i=1;i<=d;i++){
      VVa<T> ti=(*this)[i];
      res[i]=ti.con();
   }
   return res;
}

template <class T>
T VVVa<T>::operator|(VVVa<T> const& s)const
{
   Z d1=Base::dim(),d2=s.dim();
   Z d=inf<Z>(d1,d2);
   T res=T();
   for (Z i=1;i<=d;i++){
      VVa<T> ti=(*this)[i];
      VVa<T> si=s[i];
      res+=(ti|si);
   }
   return res;
}

////////////////////////// class VVVVa ////////////////////////////////////
// tensors of rank 4

template <class T>
class VVVVa: public VVVVo<T>{//version of VVVVo with arithmetic operations

   typedef VVVVo<T> Base;
   typedef VVVVa<T> Type;
   typedef T ScalarType;
```

```
public:
   CPM_LIN
      // notice that function con() declared here will be
      // implemented as conjugation of the components.
      // The present implementation is natural
      // for an interpretation of VVVVa's as T-valued spatial
      // distributions (as in wave functions).
      // Unlike the case for Va<T>, we do not include CPM_IO
      // in the interface of VVVVa<T>
// constructors
   VVVVa(Z d1,Z d2,Z d3,Z d4):Base(d1,d2,d3,d4){}
   VVVVa(Z d1,Z d2,Z d3,Z d4,T const& t):Base(d1,d2,d3,d4,t){}
      // all components are equal to t
   VVVVa(void):Base(){}
   VVVVa( V< V< V< V<T> > > > const& x):Base(x){}
   VVVVa(VVVV<T> const& x):Base(x){}
   VVVVa(VVVVo<T> const& x):Base(x){}
};

template <class T>
VVVVa<T> VVVVa<T>::operator+(VVVVa<T> const& s)const
{
   Z d1=Base::dim(),d2=s.dim();
   Z d=inf<Z>(d1,d2);
   VVVVa<T> res(d,0,0,0);
   for (Z i=1;i<=d;i++){
      VVVa<T> ti=(*this)[i];
      VVVa<T> si=s[i];
      res[i]=ti+si;
   }
   return res;
}

template <class T>
VVVVa<T> VVVVa<T>::operator-(VVVVa<T> const& s)const
{
   Z d1=Base::dim(),d2=s.dim();
   Z d=inf<Z>(d1,d2);
   VVVVa<T> res(d,0,0,0);
   for (Z i=1;i<=d;i++){
      VVVa<T> ti=(*this)[i];
      VVVa<T> si=s[i];
      res[i]=ti-si;
   }
   return res;
}

template <class T>
VVVVa<T> VVVVa<T>::operator*(T const& r)const
```

```
{
   Z d=Base::dim();
   VVVVa<T> res(d,0,0,0);
   for (Z i=1;i<=d;i++){
      VVVa<T> ti=(*this)[i];
      res[i]=ti*r;
   }
   return res;
}

template <class T>
VVVVa<T> VVVVa<T>::operator-(void)const
{
   Z d=Base::dim();
   VVVVa<T> res(d,0,0,0);
   for (Z i=1;i<=d;i++){
      VVVa<T> ti=(*this)[i];
      res[i]=-ti;
   }
   return res;
}

template <class T>
VVVVa<T> VVVVa<T>::con(void)const
{
   Z d=Base::dim();
   VVVVa<T> res(d,0,0,0);
   for (Z i=1;i<=d;i++){
      VVVa<T> ti=(*this)[i];
      res[i]=ti.con();
   }
   return res;
}

template <class T>
T VVVVa<T>::operator|(VVVVa<T> const& s)const
{
   Z d1=Base::dim(),d2=s.dim();
   Z d=inf<Z>(d1,d2);
   T res=T();
   for (Z i=1;i<=d;i++){
      VVVa<T> ti=(*this)[i];
      VVVa<T> si=s[i];
      res+=(ti|si);
   }
   return res;
}

} // namespace
#endif
```

43 cpmvcow.h

```
//? cpmvcow.h
//? Status of work 2023-10-20.
//?
//?  ...


#ifndef CPM_V_H_
#define CPM_V_H_
/*

Purpose: Basic array class with valid indexing ranging in a contiguous
   subset of Z. Two cases are of particular interest: That the lowest
   valid index is 0 (as for std::vector<>) or 1 (as e.g. in the functions
   in Press et al.). When dealing with FFT index ranges
   {-n,...,-1,0,1,...,n} are adequate (not yet implemented).
   Most constructors of V create instances of V<T> with valid indexes
   starting at 1. The two functions b_(Z) and e_(Z) provide means
   to arbitrarily shift the range of valid indexes. Access to components
   by means of indexing is range checked.

History: Till October 2010 there was a class template Vl<T> with
indexing starting at 0 and - based on that (not derived from that)
a class template V<T> with indexing starting at 1. This caused an
uncomfortable situation: Whenever dealing with a topic where I felt
that efficiency would be of utmost importance (e.g. font representation
and quantum dynamics) I was tempted to use Vl<> not only as an
internal device but also in the interface of public member functions.
This made the implementation code and even the classes dependent
on a decision that with a slight twist of emphasis could also have
been made differently. The new state of affairs is that Vl has been
eliminated (actually replaced by V) in all C+- code. If it should still
shine up in some comment, ignore it.

Recent history:
    2012-01-11 function cow() renamed to cow_()
    2012-01-18 function X2<Z,bool> findAsc(T const& t)const added
    2012-01-19 function prnOn modified so that indexes are printed
      (in commentarized form) together with the components.
    2017-02-18 Short report on a project which looked promising
      but was finally abandoned:
      The rather mature state that C++ has reached with C++11 triggered the
      desire to make more systematic use of the standard library
      facilities. Especially that now all standard containers are said to
      implement moving as a replacement of copying where appropriate
      promised to make it feasible to replace T* p_ by std::vector<T> p_
      and thus free the implementation of V<T> from defining copy, move,
      assignment operators. The first disappointing problem was that then
```

```
      V<bool>, since based on std::vector<bool>, did no longer work in my
      code which used operator[](Z) for V<bool> as in all other V<T>'s.
      Of course replacing V<bool> by V<B> would help. But re-writing all
      the many functions of V<T> in the new style looks disappointingly
      tedious to me. Before making a second attempt I need to understand
      whether the new move functionality is in fact a full replacement for
      the reference counting and copy on write functionality which is
      implemented in V<T> as of today. There is a project
      'codingexperiments' in ~/e/cpm/codingexperiments which illustrates
      the malfunction of vector<bool>, which also is well known to the
      WWW. Further there is ~/e/cpm0ExperimentBasedOnvector which
      contains the experimental version of cpmv.h. In this version by far
      not all uses of T* are replaced by vector<T>.


Credit: Modified and extended from Andrew Koenig: Ruminations on C++,
AT&T Ch. 7. Authoritative and enlightening treatment. However,
compared to the present version, some essentials are missing in
Koenig's book. Also two misprints caused trouble.

This defines a template V<T> of T-valued lists or 'vectors with
T-valued components'. It is similar to std::vector<T> of the STL
but it differs from this in some respect. In describing
V<T> and these differences, I'll introduce some notions and notations
that will be used over and over in defining and stating properties of
other CPM classes
( CPM = 'C+-' or 'Classes for Physics and Mathematics').

1. V<> instances (called simply 'Vectors' here) never request more
   memory than necessary. Thus appending elements always needs new
   allocation of memory.
   V<> is more adapted to the role of vectors as compound
   (structured) quantities than to the role as a sequential container for
   its components. Therefore, appending components at the end is
   not basic to the concept and is not worth special optimizing effords.
   Notice that V<> is not a 'polymorphic container'
   i.e. the components can hold only T typed objects and not the
   additional information content which instances of classes derived
   from T may carry. We may, however, form V<T*> for any type or class
   to support polymorphism in the old-fashioned pointer-based way.
   A safer way is to use the polymorphic vector template Vp mentioned
   in item 5.

2. V<> implements 'reference counting' and 'copy on write'. Thus
   there is no need to work with references to vectors to avoid
   superfluous copying. Assume that we nevertheless observe the
   rule to declare function arguments always as references
   or constant references. Then, the main effect of reference
   counting is that large objects built within the body of a
   function (such as matrix multiplication) will not be copied
```

```
   to the outside but simply referenced. If they are no longer
   needed, they nevertheless get deleted automatically.
   If such a referenced large object gets changed by program
   action and some old client still needs the unchanged version
   for his reference, a copy of the large object is made automatically
   and made available to its user.

3. Requirements on T in order to allow the formation of V<T>:
   To illustrate the point up front: std::vector<T> will only be built
   (at least with the MS Visual C++ 5.0 compiler) if T<T and T==T are
   defined. For V<T> there is no such restriction. To be sure: T<T is an
   obvious abbreviation of the statement:
   'for all objects t1 and t2 of type T, the expression t1<t2 is defined'
   (compare David R. Musser, Atul Saini: STL Tutorial and Reference
   Guide, Addison-Wesley 1996, p. 246). Now the pertinent statement:
   V<T>  is well defined if (not iff!) T is a built-in type or a class
   which 'implements the value interface'. Then, V<T> also implements the
   value interface. If, moreover, T 'implements the strict value
   interface', then V<T> also implements the strict value interface.
   Here, a class T is said to implement the value interface if it
   publicly defines T(), T(const T&), and T& operator=(const T&).
   T is said to implement the strict value interface if, in addition, it
   has value semantics (as opposed to pointer semantics, see
   Andrew Koenig: Ruminations on C++, AT&T 1997, p. 62. and
   Bjarne Stroustrup: The C++ Programming Language, 3. edition,
   Addison-Wesley 1997, p. 294). Classes that implement the strict value
   are most convenient to use. Code only using such quantities is
   normally easy to understand. To have an even shorter denotation for
   this favorable species we call such a class a value class or a bit
   more general:

      T is a value class : <==> T is a built-in type or a class
      that implements the strict value interface.

   As was stated already, we have:

      value class T ==> value class V<T>

   The predicates 'T satisfies the value interface' and
   'T satisfies the strict value interface' can be evaluated for
   classes providing random generators by means of the test classes
   Test_v<T> and Test_sv<T> defined in file cpmtests.h. Although V<>
   provides no random generator (Vr<>, to be mentioned soon, does)
   the code of Test_sv<T> can be read as an operational definition
   of the concept 'strict value interface'. An interesting (or common-
   place ?) observation: syntactic aspects of a program are characterized
   by the program's interaction with the compiler; semantic aspects
   ('pointer sematics',...) are characterized by the data created during
   execution.
```

```
   Requiring order operators or even arithmetic operators in T, allows to
   define T-'valued' Vectors which themselves carry natural
   order/arithmetic operators.
   Therefore V is only the starting point in a series of vector
   templates with increasing functionality, accompanied by increasing
   assumptions on the structure of T. Presently this hierarchy looks as
   follows:

      V<T> , Vo<T> , Va<T> , Vr<T>

   Every such class is derived from its predecessor in this series
   by derivation without adding new data members. Thus there are
   unambiguous casts between all these classes.

      Vo : More order related methods added to V, basic order functions
         which are declared in CPM_ORDER now already here.
      Va : arithmetic operations added to Vo
      Vr : rich infrastructure supporting automated testing of
         internal consistency.

   This approach of integrating the functions (algorithms) into an
   inheritance tree of template classes is radically different from the
   STL approach. STL keeps algorithms as separate entities outside the
   classes and uses iterators and adaptors for making them work together.
   Both methods have their advantages and disadvantages. It is probably
   fair to say that the C+- approach is less universal but more
   convenient to use within the framework it fits.

4. On polymorphism: V<T*> may be formed, but doesn't implement the value
   interface for T and derived classes. However, with the smart pointer
   templates P<T>, Pp<T>, and Po<T> defined in file cpmp.h we can form
   e.g. V<Pp<T> > and get the functionality (together with some 'extras')
   which one would expect from  V<T*>. See file cpmp.h for details and
   the polymorphic version Vp of the vector templates mentioned so far.
   See test class PolymorphicMulti<*,*,*> in cpmtests.h for a operational
   definition of polymorphism of container classes.
*/

#include <cpmfl.h> // Includes <cpmuc.h> for std::size_t
   // (and std::ptrdiff_t, which is not being used so far).
   // Small and efficient function template with minimum
   // infra-structure requirements.
#include <cpmzinterval.h>
   // includes cpmsystem.h and cpmx.h
   // With using arrays something may go wrong and so messaging
   // capability is indispensible.

#include <cpmmacros.h>
   // Provides help to write debugging-friendly function blocks.
#include <vector>
```

```
#include <set>
////////////////////////////// class V<> ///////////////////////////////
// Array with index check, reference counting, and copy on write, and
// 'value semantics' (as opposed to 'pointer semantics')
// Generalized from Koenig's class Handle p. 72-73.
// Index range is now defined as an instance of class IvZ. So each
// 'vector' has its individual index range which may start with 1
// (following the convention of the Numerical Receipes) or with 0 as
// for into the 'vector' of the STL.
//  V< V<ColRef> > is the type of bitmap data in my graphical workhorse
// class Img24. This can be considered a proof for good performance of
// the class. Efficient conversion functions from and to std::vector<>
// are now provided.
// There is a nice way to iterate over the whole index range without
// mentioning the bounds as 0 and dim-1, or as 1 and dim:
//    V<T> v=....;
//    for (Z i=v.b();i<=v.e();++i) v[i]=...;

namespace CpmArrays{

   using namespace CpmStd;

   using CpmRoot::Word;
   using CpmRoot::Z;
   using CpmRoot::R;
   using CpmRoot::Root;
   using CpmSystem::Error;
   using CpmFunctions::F;

#ifdef CPM_Fn
   using CpmFunctions::F1;
#endif

// some infrastructure

   enum Begin { LEAN };
      //: begin
      // Never form V<Begin>

   enum Outside { DEFAULT, CYCLIC, CONSTANT };
      //: outside
      // Controls the meaning of 'out of range indexes'.
      // DEFAULT: definition as the default value associated with
      // the type under consideration
      // CYCLIC: setting the meaning of out of range indexes by
      // cyclic repetition of value
      // CONSTANT: continuation as constant from the nearest value

   extern Z dimMax;
      // If the dimension of an array was either the result of a
```

```
      // calculation or of reading from a file, then the result may be
      // off the programmer's intent by orders of magnitude if something
      // went wrong. So it is helpful to exclude unnaturally large arrays
      // from becoming allocated.

   extern bool ranChc;
      //: range check
      // Initialized as true.

   extern bool ranChcAlw;
      //: range check always
      // If this is true, all access operators even those the name
      // of which suggests the opposite get checked --- with poor
      // diagnostics, though.
      // Initialized as true, since access to non-allocated memory,
      // as a rule, causes disaster. I had to discover in 2008-03-03
      // that such a case happened in the workhorse function
      // CpmGraphics::Graph::mark(V< V<C> >,...)
      // on a regular basis, due to an seemingly safe but actually
      // un-safe usage of V<>::cui().

   extern bool signal;


   void setDimMax(Z n);
      // sets dimMax=n unless n<0. In this case error with message

   Z safeDim(Z n);
      // returns n for 0<=n<=dimMax, otherwise error with message

template <class T>
   // We assume that T provides (explicitly or implicitly)
   // copy constructor, and assignment or that T is a built-in type.
   // If the index operator [] is to be used and the variable ranChc
   // (which is initialized as 'true') was not set to 'false' an out
   // of range error will result in a runtime error which will be
   // documented on cpmcerr.txt. See ranChcAlw for an additional control.
   // The error message is particularly explicit (indicating the type of
   // T) when also the macro CPM_NAMEOF is defined. If this is the case,
   // the type T needs to define the member function
   // CpmRoot::Word nameOf()const. For all Cpm-classes this is the case
   // and for user classes, there should be no difficulty in adding such
   // a function. If one wants to make use of the function declared
   // by the declaration macros CPM_ORDER type T needs to define the
   // member function  CpmRoot::Z com(T const&)const;
   // If one wants to make use of the functions declared by the
   // declaration macro CPM_IO, type T needs to define the member
   // functions bool prnOn(ostream&)const and bool scanFrom(istream&).
   // These requirements do not apply to basic types:
   // For T = N, Z, R, Rh, L, bool, string one  may use all functions of
```

```
   // V<T> without further requirements.

class V{ // vector template, indexing starts with 1 by default.
   // The index range can however be shifted or even initially set
   // arbitrarily.
   // Although implementation details are inspired from handle classes,
   // the V class is a value array and not a handle class.
   // All allocations made with new T[] all de-allocations are  delete[]
   // All data are private, so the only access to data in
   // derived classes is over the public functions of the class. All
   // these incorporate reference counting internally where needed (only
   // if the preprocessing directive CPM_USECOUNT is active). The user of
   // the class can't see this and has not to be aware of this.

   typedef V<T> Type;

public:
   CPM_IO
   CPM_ORDER
   R dis(V<T> const& h)const;

explicit V(Z n=0) ;
      // Has n components. Gives an error for n<0 and n>dimMax.
      // The components are initialized by the default constructor
      // of T if T is a class for which such a constructor is defined
      // (explicitly or implicitly).
      // If T is a built-in type, initialization is done as 0.
      // See BS3, p. 131 for initialization of built-in types via
      // formal constructor calls.
      // If n>0, the first valid index is 1, which reflects the
      // normal behavior of class V.

   V(Z n, Begin bg);
      // Defined as the previous function. But the first valid index
      // is 0 (if n>0 so that there is at least one valid index).
      // This is a somewhat contrived construction: One has to make sure
      // that this does not interfere with the definition
      // V(Z n, T const& t, Z first=1) for some choice of T. Since we
      // agree on using V<T> only for C+- types T, we will never be
      // tempted to consider V<Begin>.
      // Typical usage:
      //    V<R> v(4,LEAN); // recall: enum Begin { LEAN }
      // lets v have valid indexes 0,1,2,3. v[0]=...=v[3]=0.
      //    V<R> w(4);
      // lets w have valid indexes 1,2,3,4. w[1]=...=w[4]=0.

explicit V(IvZ const& iv);
      // Has a component for each element of iv.
      // The components are initialized by the default constructor
      // of T if T is a class for which such a constructor is defined
```

```
      // (explicitely or implicitely).
      // If T is a built-in type, initialization is done as 0.
      // See BS3, p. 131 for initialization of built-in types via
      // formal constructor calls.

   V(std::vector<T> const& v, Z first);
      // Construction from a standard library vector. This is useful
      // for interaction with the standard containers (which don't
      // implement reference counting). The second argument gives the
      // begin of the valid index range. A value 0 lets the result of
      // the construction behave like v with respect to indexing.

   V(Z n, T const& t, Z first=1) ;
      // Has n components all initialized as t.
      // The third argument gives the first valid index.

   V(Z n, T const& t, Begin bg);
      // Has n components all initialized as t.
      // The third argument (that can have only one value, namely LEAN)
      // says that the first valid index is 0.

   V(IvZ const& iv, T const& t);
      // Has iv.car() components all initialized as t

   V(Z n, F<Z,T> const& f);
      // construction from a function. Of course,
      // V<T> v(n,f);
      // implies v[i]==f(i) for all valid indexes i of v.

   V(IvZ const& iv, F<Z,T> const& f);
      // construction from a function. Of course,
      // V<T> v(iv,f);
      // implies v[i]==f(i) for all valid indexes i of v.

   V(V<T> const& h);
      // copy constructor

// constructors from explicit lists
   explicit V(std::initializer_list<T> il );
      // requires C++11
      // constructors from explicit lists such as
      // V<Z> v{1,2,4,8};

   virtual ~V();
      // destructor

   V<T>& operator=(V<T> const& h);
      // assignment

   virtual V<T>* clone(void)const{ return new V(*this);}
```

```
V<T> toClnBase()const{ return *this;}
   //: to clone base

Z dim()const { return sz_;}
   //: dimension
   // Returns the number of components of the vector *this

Z size()const { return sz_;}
   //: size
   // for uniformity with STL

IvZ dom()const { return iv_;}
   //: domain
   // Notice that with the array *this there is the
   // function f: {iv_b(),...,iv_.e()}-->T, i|-->(*this)[i]
   // associated in a natural manner.
   // For this function, dom() is just the domain.
   // The understanding of arrays as functions with domains of type
   // IvZ seems to be a good guide for defining some of the member
   // functions in a more natural manner by using arguments of type
   // IvZ. Present examples are the functions fa_ and valOn.

bool isVoid()const { return iv_.isVoid();}
   //: is void
   //  short answer on whether the dimension is zero

bool valInd(Z i)const{ return iv_.hasElm(i);}
   //: valid index
   // returns the validity of i as an index of *this

Z makeIndVal(Z i, Outside mode=CYCLIC )const
{
   if (iv_.hasElm(i)) return i;
   if (mode==CYCLIC) return iv_.cyc(i);
   else return iv_.con(i);
}
   //: make index valid
   // If i is a valid index we return i. If not, the result is
   // iv_.cyc(i) for mode=CYCLIC and iv_.con(i) else. Notice that the
   // normal treatment of mode==DEFAULT works not by replacing one
   // value of the index by another. It works on the value of vector
   // components and makes use of the default constructor T()

bool sameDom(V<T> const& v)const{ return iv_==v.iv_;}
   //: same domain

std::vector<T> std()const;
   //: standard
   // Returns a STL-vector which holds all components of *this.
```

```
   virtual Word nameOf()const;
      //: name of
      // returns a name of the type

// constant access functions

   const T& operator[](Z i)const;
      // Getting to the value of a component of a const instant of
      // V<T>. For instance
      // const V<T> v = ... ;
      // T t = v[3];
      // If CPM_USECOUNT is enabled (the normal case) the actual
      // process of going from v to v[3] depends on whether v is of type
      // V<T> or const V<T>. In the non-constant case the evaluation of []
      // may involve a copy action on v (and returning the component of
      // the copy). If we intend only to read the component, such an
      // copy action is not neededd and should be avoided by casting v
      // to type const V<T>.
      // Casting v from V<T> to const V<T> works this way:
      //    v = static_cast<const V<T>>(v);
      // Casting back to mutable seems to work only by using a new name:
      //    auto vMutable = const_cast<V<T>&>(v);
      //    vMutable[1] = T(); //( for instance)
      // or by applying mutating operations to an anonymous object as in:
      //    const_cast<V<T>&>(v)[1]=T();
      // Notice the '&' here which the compiler requires. By the way, the
      // effect of the 'anonymus action' actually is to change the state
      // of v: v[1]==T().
      // My analysis of the situation is in ~/codingexperiments/main.cpp
      // In the following there are many pairs of functions
      // T const& f(...)const;
      // T& f(...);
      // for which the above considerations apply mutandis mutatis.

   T const& cui(Z i)const;
      //. component (with) unchecked index
      // getting to the value of a component for 'read', e.g.
      // T t=cui(i);
      // It helps to write efficient code in classes which use V<>-typed
      // data members.
      // Notice that writing loops by using b() and e() to define
      // the range is much safer than using limits like 1 and dim().
      // Index range check is missing only if variable ranChcAlw was
      // changed to 'false'.

   T const& cyc(Z i)const;
      //: cyclic
      // getting to the value of a component for 'read', e.g.
      // T t=cyc(i);
```

```
      // Here i is understood as cyclic (i.e. i modulo sz_). So no value
      // of the index has to be considered 'out of range' and for i
      // 'in range' cyc(i)==(*this)[i]

   T const& li(Z i)const{ return *(p_+i);}
      //. lean index
      // The valid range is for (Z i=0;i<sz_;++i) li(i)

   T& li(Z i)
      //. lean index
   {
#if defined(CPM_USECOUNT)
      cow_();
#endif
      return *(p_+i);
   }

   T const& con(Z i)const;
      //: constant
      // Same logic as cyc() but with constant continuation
      // instead of cyclic.

   T operator()(Z i, Outside mode=DEFAULT)const;
      //: ()
      // Read access defined for all i.
      // By this definition, a vector becomes a mapping from Z to T.
      // For i's outside the proper range,
      // the default T is returned for mode==DEFAULT or if
      // sz_==0. If mode==CYCLIC cyc(i) gets returned.
      // For mode==CONSTANT we return p_[0] for i<=0 and p_[sz_-1]
      // for i>=sz_.
      // Notice that the return value is not a reference in
      // accordance with function behavior.

   T const& read(Z i, Outside mode=DEFAULT)const;
      //: read
      // Same as previous function but returning a reference instead of a
      // T.
      // Reading components in an efficient (as &'s), safe and flexible
      // manner.
      // See T operator()(Z i, Outside mode=DEFAULT)const; for the
      // meaning of the second argument.

   T const& r(Z i)const{ return p_[iv_.ri(i)];}
      //. read
      // Unchecked and fast version of T const& oparator[](Z i)const

   T& w(Z i)const{ return p_[iv_.ri(i)];}
      //. write
      // Unchecked and fast version of T& oparator[](Z i)
```

```
   F<Z,T> fnc(Outside meth=DEFAULT)const;
      // function
      // returns the function Z --> T, i |--> (*this)(i,meth)

 // non-constant access functions
   T& operator[](Z i);
      //: []
      // See T const& operator[](Z i)const for discussion of details
      // that are relevant in the case that CPM_USECOUNT is defined.

   T& cui(Z i);
      //. component (with) unchecked index
      // setting to the value of a component
      // T t=...;
      // V<T> x=...;
      // x.cui(i)=t;

   T& cyc(Z i);
      //: cyclic
      // setting to the value of a component
      // T t=...;
      // V<T> x=...;
      // x.cyc(i)=t; meaning of i is modulo sz_. So i is never out of
      // range

   T& con(Z i);
      //: constant
      // Same logic as cyc() but with constant continuation
      // instead of cyclic.

   V<T>& b_(Z i){ iv_.b_(i); return *this;}
      //. set b(), i.e. the first valid index.
      // For instance
      // V<R> v("",sqrt(2),sqrt(3),sqrt(4),sqrt(5));
      // for (Z i=1;i<=v.dim();++i) cout<<v[i]<<endl;
      // v.b_(0);
      // for (Z i=0;i<v.dim();++i) cout<<v[i]<<endl;
      // shows all components of the vector in both cases.

   V<T>& e_(Z i){ iv_.e_(i); return *this;}
      //. set e(),  i.e. the last valid index.

// accessing the first and the last element for reading
   T const& fir()const;
      //: first
   T const& last()const;
      //: last

// accessing the first and the last element for writing
```

```
   T& fir();
      //: first
   T& last();
      //: last

// getting the first and the last valid index
   Z b()const { return iv_.b();}
      //. begin
      // Returns the first valid index.

   Z e()const { return iv_.e();}
      //. end
      // Returns the last valid index.
      // Allows to write loops over all components as
      // for (Z i=v.b();i<=v.e();i++) ... v[i] ...;
      // Note that this is safe also for v.dim()==0, since then
      // v[e()] will never be called. Here one could replace
      // v[i] by v.cui(i) without danger.

   Z n()const { return iv_.n();}
      //. next
      // Returns the index next to the last valid one.
      // This allows to write loops over all components as
      // for (Z i=v.b();i<v.n();i++) ... v[i] ...;
      // Note that this is safe also for v.dim()==0, since then
      // v[e()] will never be called. Here one could replace
      // v[i] by v.cui(i) without danger.

   V<T> meet(IvZ const& iv)const;
      //: meet
      // Returns a vector which, when considered as a function is the
      // restriction of function *this to the domain iv_ & iv.

   V<T> join(IvZ const& iv)const;
      //: join
      // Returns a vector which, when considered as a function is the
      // extension of function *this to the domain iv_ | iv, where
      // all function values on iv\iv_ are T().

   V<T> operator +(Z i)const{ return V<T>(iv_+i,p_,"");}
      //: operator +
      // Returns a vector res such that res.dom() is the shifted
      // domain dom()+i of *this and has the same components as
      // *this.

   V<T> operator -(Z i)const{ return V<T>(iv_-i,p_,"");}
      //: operator -
      // Returns a vector res such that res.dom() is the shifted
      // domain dom()-i of *this and has the same components as
      // *this.
```

```
void set_(T const& t);
   //: set
   // non-constant function which sets all components of (*this)
   // equal to t

V<T> set(Z i, T const& t)const;
   //: set
   // Returns a vector that originates from *this by setting the
   // value of component i.
   // Regular behaviour:
   // if we say
   // Z i=...;
   // T t= ...;
   // V<T> v=...;
   // v=v.set(i,t);
   // then the i-th component (see eli() ) of v will get the value t
   // unless i<1.
   // If i is a value for which (*this)[i-1] is not yet defined,
   // the vector becomes enlarged to the necessary size and all
   // not specified components initialized with the default
   // constructor of T

T in_(T const& t, bool reversed=false);
   //. insert
   // This operations treats *this as a shift register:
   // All components get shifted by one position 'to the right' and
   // the total length (dim) of the vector remains the same. So what
   // was the last component prior to the operation has to be removed
   // from the vector, in order to not wasting information, this
   // removed component will shine up as the return value of the
   // function. After the operation, the first component of the vector
   // will be t.
   // If reversed==true, t gets inputted at the end, and all shift
   // operations go 'to the left'.


Z locAsc(T const& t)const;
   //: locate ascending
   // Here it is assumed that *this is ascendingly strictly ordered:
   // If sz_>=2 we have p_[i]<p_[i+1] for all i \in {0,sz_-2}.
   // For sz_<2 there is no condition.
   // For sz_==0 we stop with error.
   // For t<fir() we return b()-1
   // For t>=last() we return e()
   // If none of the previous conditions was met we necessarily have
   // sz_>=2 and we return the uniquely determined j such that
   //    p_[j]<=t<p_[j+1].
   // The possible results from this regular part of the functionality
   // thus are b(),...,e()-1.
```

```
      // Notice that the very similar function locate of Press et al.
      // only guarantees p_[j]<=t<=p_[j+1] for its return value j.

   X2<Z,bool> findAsc(T const& t)const
      //: find ascending
      // The second component of the return value says if t is
      // among the components of *this. Then the first component
      // of the return value gives the i for which (*this)[i]==t.
   {
      Z i=locAsc(t);
      return X2<Z,bool>(i,valInd(i) && t==cui(i));
   }

   X2<Z,bool> find(T const& t)const;
      //: find
      // The second component of the return value says if t is
      // among the components of *this. Then the first component
      // of the return value gives the lowest i for which (*this)[i]==t.
      // No ordering of *this is assumed.

// building new objects from given ones (generative methods)

// concatenating vectors and appending components. These operations have
// always O(sz_) complexity. The corresponding combined assignements are
// less direct to implement are not considered useful in the present
// context (they would not be more efficient than the friend versions,
// since new memory has to be allocated anyway).

   V<T> app(T const& t)const;
      // returns a vector which is obtained from *this by appending t
      // as the last component
      // notice also the prepend functions to be introduced
      // after the insert-functions (in order to have the inline
      // definition available).

   V<T> app(V<T> const& h)const;
      // returns a vector which is obtained from *this by appending h
      // at the back end

   void push_back(T const& t) { *this=app(t);}
      // for uniformity with STL

   V<T>& operator<<(T const& t) { return *this=app(t);}
      // appending an element in Ruby-style
      // Allows successive application as in
      // V<Word> w;
      // w<<"many"<<"words"<<"get"<<"easily"<<"stored";

   V<T>& operator<<(V<T> const& h) { return *this=app(h);}
      // appending an array in Ruby-style
```

```
   V<T>& operator&=(T const& t) { return *this=app(t);}
   V<T>& operator&=(V<T> const& h) { return *this=app(h);}
   V<T> operator&(T const& t)const { return app(t);}
   V<T> operator&(V<T> const& h)const { return app(h);}
      // appending in my favorite style

   V<IvZ> valOn(F<T,bool> const& f)const;
      //: valid on
      // retuns the array of those sub-intervals of the
      // whole indexing interval dom() on which the function
      // dom()-->bool, i|-->f((*this)[i]) yields true.
      // Notice that the result res \in V<IvZ> is a convenient
      // representation of a subset of dom(). Considering
      // *this as a function g: dom()-->T, then the function
      // h:=g&f is of type dom()-->bool and res, as a subset
      // of dom(), is h^-1({true}).

// resizing

   V<T> resize(Z newDim)const;
      // Returns a vector res such that res.dim()=newDim. If newDim is
      // smaller than dim(), the end of *this will be cut away. If newDim
      // is larger, T()'s will be added.
      // For newDim<0 the action is as if newDim==0.

   V<T> cut(Z i)const{ return resize(sz_-i);}
      // returned is a V<T> which results from *this by removing i
      // components from the end. For exotic values of i, see code
      // and explanation of resize.

// elimination and insertion

   V<T> eli(IvZ const& iv)const;
      // eli stands for eliminate
      // Eliminating all components which belong to iv. No exceptions
      // can happen! Universal and convenient of elimination. All other
      // forms are superfluous. Moreover, they are to be considered as
      // obsolete.

   V<T> eli(Z i, Z nEli=1)const{ return eli(IvZ(nEli,i,0));}
      // eli stands for eliminate
      // returned is a vector which is obtained from *this by
      // eliminating the i-th component and the nEli-1 following ones
      // (thus nEli components are removed) and shifting all later
      // components (if there are such components left) 'to the left' to
      // close the gap.

  // V<T> eliFirst(Z nEli=1)const { return eli(1,nEli);}
   V<T> eliFirst(Z nEli=1)const { return eli(IvZ(nEli,b(),0));}
```

```
    // returned is a vector which is obtained from *this by
    // eliminating the nEli first components. If no
    // argument is provided, actually the first component
    // gets eliminated. Notice that in the three-argument constructor
    // IvZ(i,j,k) the first argument is the cardinality, the second
    // argument is the first element, and the third argument is dummy.

// V<T> eliLast()const { return eli(sz_,1);}
 V<T> eliLast()const { return eli(IvZ(e(),e()));}
    // returned is a vector which is obtained from *this by
    // eliminating the last component.

 V<T> eli1(V<T>& h, Z i)const;
    // We return a vector which results from *this by eliminating
    // h.dim() components starting at the i'th component (for i<1,
    // i==1 is understood). After the call h will hold the
    // eliminated components in due order (and no more - even if h was
    // longer before).
    // The function name ends in '1' to indicate that the first
    // argument is a non-constant reference, used for communication
    // of a part of the result.

 // condensation (contracting equivalent components into one)

 X2< V<T>, V<Z> > condense( bool (*equi)(const T&, const T&))const;
    // given an equivalence relation equi on T (t1~t2 <==>
    // equi(t1,t2)==true )
    // we divide the components of *this into equivalence classes.
    // Let the resturned pair be written as (res1,res2) . Then
    // (i) (*this)[i]~res1[res2[i]]
    // (ii) there is a j such that  (*this)[j]==res1[res2[i]]
    // Actually, the construction is made such that this j is the
    // smallest j for which (*this)[j]~res1[res2[i]].

 V<T> select(V<bool> const& s)const;
    // returned is a list which is obtained from *this by eliminating
    // all components (*this)[i] for which s[i] is defined and
    // statisfies s[i]==false. Beside of this removal, the order of the
    // components in *this is retained. So if s is defined
    // by a condition s[i]=Condition((*this)[i]), for the vector
    // component, this condition has to express a property we like to
    // have fulfilled for the result-vector of the select-operation.
    // 's expresses the favorable condition' and  n o t  the one to be
    // eliminated. Notice, that the select operation makes sense for
    // all values of s.dim().

 V<T> ins(Z i, T const& t)const;
    //: insert
    // returned is a vector which is obtained from *this by
    // inserting t as the i-th component and shifting all later
```

```
   // components 'to the right' to give room for t.
   // The phrase 'i-th component' refers to natural counting (thus
   // p_[i-1] is the i-th component of *this).

V<T> ins(Z i, V<T> const& h)const ;
   // returned is a vector which is obtained from *this by
   // inserting h as the i-th and following component,
   // and shifting all later components of *this
   // 'to the right' to give room for h.
   // The phrase 'i-th component' refers to natural counting (thus
   // p_[i-1] is the i-th component of *this).

V<T> prepend(T const& t)const
   // returns a vector which is obtained from *this by appending t
   // as the first component
   { return ins(1,t);}

V<T> prepend(V<T> const& h)const
   // returns a vector which is obtained from *this by appending h
   // at the front end
   { return ins(1,h);}

V<T> rev()const;
   //: reversed
   // returned is the reversed vector (indexing in the opposite
   // direction)

V<T>& rev_(){ return *this = rev();}
   //: reversed
   // changes *this into a reversed version of itndexing in the
   // opposite
   // direction)

// re-indexing

V<T> rot(Z s, Outside mode=CYCLIC)const;
   //: rotate
   // Name as the list transformation function Rotate of Mathematica.
   // v.rot(s)[i] == v(i-s,mode)
   // This means that we shift the component i of the original vector
   // into the new position i+s

void rot_(Z s, Outside mode=CYCLIC){ *this=rot(s,mode);}
   //: rotate
   // Same as rot, but as a mutating operation.

V<T> compose(V<Z> const& w)const;
   //:: compose
   // v.compose(w)[i] = v[w[i]]
```

```
   V<T> permute(V<Z> const& w)const{ return compose(w);}
      //:: permute

   // transforming components

   template <class Y>
   V<Y> operator()(F<T,Y> const& f)const { return V<Y>(iv_,fnc()&f);}
      // generating arrays of different type by a type changing function

   V<T> operator()(T (*f)(T const&))const{ return fa(f);}
      // generating arrays of the same type with components f(c) instead of
      // c.

   V<T> operator()(T (*f)(T))const{ return faa(f);}
      // generating arrays of the same type with components f(c) instead of
      // c.

// defining generative laws and mutating laws for various expressions by
// function pointers. The idea behind is, that for T which allows
// particular operations, we can define those without using iteration via
// a operator[] since these all are defined directly in terms of
// pointers.
// See implementation of +=, -=, ... in Va<T> how this works

   T fAcc1(T (*f)(T const&), void (*acc)(T&, T const&))const;
      // returns an accumulated value of all values f(c), where c
      // are the components of *this. Accumulation is defined by
      // the argument acc:
      // T res=T(); 'for all components c' acc(res,c)

   T fAcc2(T (*f)(T const&, T const&), void (*acc)(T&, T const&),
      V<T> const& h )const ;
      // returns an accumulated value of all values f(c,ch), where c and
      // ch are the components of *this and h respectively. Accumulation
      // is defined by the argument acc:
      // T res; 'for all components c' acc(res,c)
      // useful in defining scalar products

   template <class Y>
   Y fAcc3(Y (*f)(T const&, T const&), void (*acc)(Y&, Y const&),
      V<T> const& h )const ;
      // returns an accumulated value of all values f(c,ch), where c and
      // ch are the components of *this and h respectively. Accumulation
      // is defined by the argument acc

   V<T> fAcc4(F<T2<T>,T> const& f)const;
      // The kind of accumulation needed for computing the forces in
      // particle systems with pair interaction. Here we assume that
      // forces and positions are of the same type, e.g. R2 or R3.
      // Let x[1],...x[n] be the components of *this. We first compute
```

```
   // the incomplete matrix f(x[i],x[j]) i=1,...n, j<i and complete it
   // assuming f(x[i],x[j]) = - f(x[j],x[i]). In application mentioned
   // earlier this is the collection of mutually forces. The total force
   // on particle i is sum over j of f(x[i],x[j]). It is the i-th
   // component of the V<T>-typed return value of the function.
   // The force type T thus needs to provide operations unary - and +=.

V<T> fAcc5(F<R,R> const& dPot)const;
   // The kind of accumulation needed for computing the forces in
   // particle systems with pair interaction derived from a distance-
   // dependent (radial symmetric) potential. The argument dPot is
   // the derivative with respect to r of the r-dependent radially
   // symmetric pair potential.

V<T> fAcc6(F<R,R> const& dPot)const;
   // forces from a space-dependent potential

V<T> fa(T (*f)(T const&))const ;
   // returns a vector defined by replacing the components c of *this
   // by f(c)

V<T> faa(T (*f)(T))const ;
   // returns a vector defined by replacing the components c of *this
   // by f(c)

V<T> fb(T (*f)(T const&, T const&), T const& t)const;
   // returns a vector defined by replacing the components c of *this
   // by f(c,t)

template <class Y>
V<T> fb2(T (*f)(T const&, Y const&), Y const& t)const;
   // returns a vector defined by replacing the components c of *this
   // by f(c,t)

template <class Y>
void fb2_(T (*f)(T const&, Y const&), Y const& t);
   // replaces *this by a vector defined by replacing the components
   // c of *this by f(c,t)

void fc_(T (*f)(T const&, T const&), T const& t) ;
   // replaces *this by a vector defined by replacing the components c
   // of *this by f(c,t)

void fd(T (*f)(T const&, T const&), T& t)const;
   // replaces t by f(c,t) for each component c
   // If, for instance, f(c,t)=t+c*c we replace t by t+sum of c*c

V<T> fe(T (*f)(T const&, T const&), V<T> const& h)const;
   // returns a vector defined by replacing the components c of *this
   // by f(c,c') where c' are the components of h.
```

```
   // Error if dim()!=h.dim()

template <class Y>
V<T> fet(T (*f)(T const&, Y const&), V<Y> const& h)const;
   // returns a vector defined by replacing the components c of *this
   // by f(c,c') where c' are the Y-typed components of h.
   // Error if dim()!=h.dim(). Template version of fe. Unfortunately
   // h.p_ is not accessible in the implementation of this function.
   // This enforced introducing the non-canonical access function rep()
   // in 2014-01-23.

V<T> fe2(T (*f)(T const&, T const&), V<T> const& h)const;
   // Very similar to fe. However, the dimension of the result
   // is the maximum of dim() and h.dim(). Non-existing components
   // of any of the operands are replaced by T().

void ff_(T (*f)(T const&, T const&), V<T> const& h, Z i=0);
   // Replaces the components of *this starting from (*this)[i]
   // by f(c,c') where c' are the components of h. Thus for i=0
   // and h.dim()>=sz_ the whole array *this is affected.

void fg_(T (*f)(T const&, T const&, T const&),
   V<T> const& h, T const& t);
   // replaces the components c of *this
   // by f(c,c',t) where c' are the components of h

void fa_(F<T,T> const& f, IvZ iv);
   // replaces the components c of *this with index in iv
   // by f(c); modern form of fa.

void fh_(T const& w1, T const& w2, T const& w3, Outside mode);
   // replaces the component c[i] of *this by
   // w1*read(i-1,mode)+w2*read(i,mode)+w3*read(i+1,mode)
   // Thus makes use of multiplication in T.

template <class Y>
void fht_(Y const& w1, Y const& w2, Y const& w3, Outside mode);
   // replaces the component c[i] of *this by
   // read(i-1,mode)*w1+read(i,mode)*w2+read(i+1,mode)*w3
   // Thus makes use of multiplication in T.

template <class Y>
V<T> fh(T (*f)(T const&, T const&, T const&, Y const&, Z),
   Y const&, Outside mode)const;
   // Returns a vector in which the component c[i] is replaced
   // by f(c[i-1],c[i],c[i+1],y,i),where the i-/+1 are understood
   // according to mode. The quantity y helps to provide parameters
   // required by a concrete situation. Worked for implementing
   // the Hamiltonian corresponding to Dirac's relativistic wave
   // equation.
```

```
   template <class Y>
   V<Y> fi(Y (*f)(T const&, T const&), void (*acc)(Y&, Y const&))const;
      // Returns a vector with Y-valued components y[i] which are obtained
      // by accumulating the terms f((*this)[i],(*this)[j]).
      // Accumulation is defined by the argument acc.

   template <class Y>
   V<Y> fj(Y (*f)(T const&, T const&), void (*acc)(Y&, Y const&))const;
      // Returns a vector with Y-valued components y[i] which are obtained
      // by accumulating the terms f((*this)[i],(*this)[j]).
      // Accumulation is defined by the argument acc.
      // We assume f((*this)[i],(*this)[j]) == - f((*this)[j],(*this)[i])
      // and use this for doing only one evaluation of f
      // for the two pairs (i,j) and (j,i) and no evaluation
      // of f for any pair (i,i).
      // We assume that for any Y-object y, -y is defined.

// diagnostics
   void show(ostream& out)const ;
      // diagnostic messages;

     T *const rep()const{ return p_;} // 20014-01-24
      // This is needed in the implementation of V<T>::fet<Y>. Here we need
      // direct access to the data member p_ of a function argument of type
      // V<Y>. Unfortunately (and unexpectedly) what is OK for V<T> does
      // not work for V<Y>. This function conflicts with my ambition to
      // ban pointers from the public interface of C+- classes.

      Z getCount()const
      {
         Z res=0;
#if defined(CPM_USECOUNT)
         res=u_.getCount();
#endif
         return res;
      }

protected:
   void cow_(void);
      //. copy on write
      // The first '*this-changing' statement in the
      // body of a non-constant member function has to be
      //   cow_();

private:
// member functions
   V(IvZ iv, T* p1, Word dummy):iv_(iv){ini_(p1);dummy;}
      // p1 has to be created with new; since this is prone to misuse
      // this function has to be private.
```

```
   V(Z n, Z first, T* p1, Word dummy):iv_(n,first,137){ini_(p1);dummy;}
      // similar to previous function

   T* copy()const;
      //: copy
      // returns a pointer to a storage area that contains
      // a copy of the components of *this

// data members

#if defined(CPM_USECOUNT)
   UseCount u_;
#endif
   // u_ becomes initialized by the default constructor and thus gets
   // a count 1, saying that the piece of free store pointed to by p_
   // is in use by *this

// static data
   static const T def_;
      // allows read function to return references
// static functions
   static T fCyc(Z const& i, V<T> const& v) { return v(i,CYCLIC);}
   static T fCon(Z const& i, V<T> const& v) { return v(i,CONSTANT);}
   static T fDef(Z const& i, V<T> const& v) { return v(i,DEFAULT);}
   static size_t sit(Z i) { return static_cast<size_t>(i);}
      // size type
      // Instead of new T[i], I now write new T[sit(i)] so that
      // allocation is always fed with a parameter which fits the system
      // (in 64 bit systems, size_t may be 64 bit wide). size_t is known
      // due to inclusion of <cpmfl.h> and it is assumed to take the
      // bit-width of the machine properly into account.
   void ini_(){sz_=iv_.car();p_=new T[sit(sz_)];}
   void ini_(T *p){sz_=iv_.car();p_=p;}

protected:
// data members
   // These should be accessible to derived classes for enabling fast
   // component operations.
   IvZ iv_;
      // Set of valid indexes. Since (*this) can be considered a
      // function iv_ --> T, this object is also called the domain
      // of *this.

   Z sz_;
      // number of valid indexes (depends on iv_, equals iv_.car())

   T* p_;
      // pointer to beginning of the array.
};
```

```
//////// using V for some special template arguments//////////////////

V<Z> IvZtoVofZ(IvZ const& iv);
   //: Iv to V of Z
   // returns the Z's that make up the interval iv
   // as the components of an ordered array (increasing
   // order, of course)

V<Z> VofIvZtoVofZ(V<IvZ> const& viv);
   //: V of IvZ to V of Z
   // appends the results from applying the previous functions
   // to the viv[i] according to the obvious code
   //  { V<Z> res(0); Z n=viv.dim();
   //  for (Z i=0;i<n;++i) res&=IvZtoVofZ(viv[i]); return res;}

V<Word> comLine(int argc, char* argv[]);
   //: command line
   // converts the C-traditional command-line argument into a C+- array.

////////////////////// Implementation ////////////////////////////////
template <class T>
const T V<T>::def_=T();

template <class T>
inline   V<T>::~V(){
#if defined(CPM_USECOUNT)
   if (u_.only()){ /*cout<<"~V called"<<endl;*/delete[] p_;}
#else
   /*cout<<"~V called"<<endl;*/ delete[] p_;
#endif
}

template <class T>
Word V<T>::nameOf()const{
   Word nt=Root<T>(T()).nameOf();
   Word wi="V<";
   return wi&nt&">";
}

template <class T>
R V<T>::dis(V<T> const& h)const
{
   R res=0.;
   if (iv_!=h.iv_) return R(1);
   for (Z i=b();i<=e();++i){
      res+=Root<T>(cui(i)).dis(h.cui(i));
   }
   return res;
}
```

```
template <class T>
inline T const& V<T>::operator[](Z i)const
{
   if (ranChc){
      if (!iv_.hasElm(i)){
         cpmerror(nameOf()&
         "::operator[]const: read-index out of  range: i= "&
         cpm(i)&" iMin= "&cpm(iv_.b())&" iMax= "&cpm(iv_.e()));
      }
   }
   if (signal) cout<<" const [] called"<<endl;
   return p_[iv_.ri(i)];
}


template <class T>
inline T& V<T>::operator[](Z i)
{
   if (ranChc){
      if (!iv_.hasElm(i)){
         cpmerror(nameOf()&"::operator[]: write-index out of  range: i= "&
         cpm(i)&" iMin= "&cpm(iv_.b())&" iMax= "&cpm(iv_.e()));
      }
   }
   if (signal) cout<<" mutating [] called"<<endl;
#if defined(CPM_USECOUNT)
   cow_();  // copy on write since operator is not const
      // may change u_ and p_!
#endif
   return p_[iv_.ri(i)];
  //return p_ + iv_.ri(i); // Trying to return T& directly, i. e. without
  // a type conversion from T. Does not work!
}

// using intentionally previously defined []-indexing
// for common messaging and safeness.
template <class T>
T const& V<T>::fir()const{ return (*this)[iv_.b()];}

template <class T>
T& V<T>::fir(){ return (*this)[iv_.b()];}

template <class T>
T const& V<T>::last()const{ return (*this)[iv_.e()];}

template <class T>
T& V<T>::last(){ return (*this)[iv_.e()];}

template <class T>
```

```
inline T const& V<T>::cui(Z i)const
{
   if (ranChcAlw){
      if (!iv_.hasElm(i)) cpmerror("index out of range");
   }
   return p_[iv_.ri(i)];
}

template <class T>
inline T& V<T>::cui(Z i)
{
   if (ranChcAlw){
      if (!iv_.hasElm(i)) cpmerror("index out of range");
   }
#if defined(CPM_USECOUNT) // don't call a function if not needed
   // even if its implementation is trivial
   cow_();
#endif
   return p_[iv_.ri(i)];
}

template <class T>
T const& V<T>::cyc(Z i)const
{
   return (*this)[iv_.cyc(i)];
}

template <class T>
T& V<T>::cyc(Z i)
{
#if defined(CPM_USECOUNT)
   cow_();
#endif
   return (*this)[iv_.cyc(i)];
}

template <class T>
T const& V<T>::con(Z i)const
{
   if (sz_==0) return def_;
   return (*this)[iv_.con(i)];
}

template <class T>
T& V<T>::con(Z i)
{
   if (sz_==0){
      cpmerror("V<T>::con(Z i): i="&cpm(i)&
         " is no valid index in void array");
      return p_[0]; // never happens
```

```
   }
   cow_();
   return (*this)[iv_.con(i)];
}

template <class T>
T const& V<T>::read(Z i, Outside mode)const
{
   if (iv_.hasElm(i)) return p_[iv_.ri(i)];
   else{
      if (mode==DEFAULT) return def_; // reference to it can be returned
      else if (mode==CYCLIC) return cyc(i);
      else return con(i);
   }
}

template <class T>
T* V<T>::copy()const
{
 //  cout<<" copy() called"<<endl;
   T* itp=p_;
   T* q=new T[sit(sz_)];
   T* it=q;
   Z i=sz_;
   while (i--) *it++=*itp++;
   return q;
}

template <class T>
T V<T>::operator()(Z i, Outside mode)const
{
   if (sz_==0) return T();
   if (iv_.hasElm(i)) return p_[iv_.ri(i)];
   if (mode==DEFAULT) return T();
   else if (mode==CYCLIC) return cyc(i);
   else return con(i);
}

template <class T>
F<Z,T> V<T>::fnc(Outside mode)const
{
#ifdef CPM_Fn
   if (mode==DEFAULT) return F1<Z,V<T>,T>(*this)(fDef);
   else if (mode==CYCLIC) return F1<Z,V<T>,T>(*this)(fCyc);
   else return F1<Z,V<T>,T>(*this)(fCon);
#else
   if (mode==DEFAULT) return F<Z,T>(bind(fDef,_1,*this));
   else if (mode==CYCLIC) return F<Z,T>(bind(fCyc,_1,*this));
   else return F<Z,T>(bind(fCon,_1,*this));
#endif
```

```
}

template <class T>
void V<T>::cow_(void)
   // body is void if CPM_USECOUNT is not defined
{
#if defined(CPM_USECOUNT)
   if (u_.makeOnly()){
      p_= (sz_==0 ? 0 : copy());
      u_.startNew_();
   }
#endif
}

// constructors
// 137 is a dummy argument

template <class T>
V<T>::V(Z n):iv_(safeDim(n),1,137)
{
   ini_();
   T* q=p_;
   for (Z i=0;i<sz_;++i) *q++ = T();
}

template <class T>
V<T>::V(Z n, Begin bg):iv_(safeDim(n),0,137)
{
   ini_();
   T* q=p_;
   for (Z i=0;i<sz_;++i) *q++ = T();
}

template <class T>
V<T>::V(Z n, T const& t, Begin bg):iv_(safeDim(n),0,137)
{
   ini_();
   T* q=p_;
   for (Z i=0;i<sz_;++i) *q++ = t;
}

template <class T>
V<T>::V(Z n, T const& t, Z first):iv_(safeDim(n),first,137)
{
   ini_();
   T* q=p_;
   for (Z i=0;i<sz_;++i) *q++ = t;
}

template <class T>
```

```
V<T>::V(Z n, F<Z,T> const& f):iv_(safeDim(n),1,137)
{
   ini_();
   T* q=p_;
   Z j=iv_.b();
   for (Z i=0;i<sz_;++i) *q++ = f(j++);
}

template <class T>
V<T>::V(IvZ const& iv):iv_(iv)
{
   ini_();
   T* q=p_;
   for (Z i=0;i<sz_;++i) *q++ = T();
}

template <class T>
V<T>::V(IvZ const& iv, T const& t):iv_(iv)
{
   ini_();
   T* q=p_;
   for (Z i=0;i<sz_;++i) *q++ = t;
}

template <class T>
V<T>::V(IvZ const& iv, F<Z,T> const& f):iv_(iv)
{
   ini_();
   T* q=p_;
   for (Z i=0;i<sz_;++i) *q++ = f(i+iv_.b());
}

template <class T>
V<T>::V(std::initializer_list<T> il ):
iv_((Z)il.size(),1,137) // 137 is dummy argument
{
   ini_();
   std::uninitialized_copy(il.begin(),il.end(),p_);
      // see BS4, p. 498
}

template <class T>
V<T>::V(std::vector<T> const& v, Z first):
iv_((Z)v.size(),first,137)
{
   ini_();
   typename std::vector<T>::const_iterator i;
   T* q=p_;
   for (i=v.begin();i!=v.end();++i) *q++=*i;
}
```

```
template <class T>
std::vector<T> V<T>::std()const
{
   std::vector<T> res(sz_);
   typename std::vector<T>::iterator i;
   T* q=p_;
   for (i=res.begin();i!=res.end();++i) *i=*q++;
   return res;
}

template <class T>
#if defined(CPM_USECOUNT)
   V<T>::V(V<T> const& h) :iv_(h.iv_),sz_(h.sz_),u_(h.u_),p_(h.p_){}
#else
   V<T>::V(V<T> const& h) :iv_(h.iv_),sz_(h.sz_),p_(h.copy()){}
#endif

template <class T>
V<T>& V<T>::operator=(V<T> const& h)
{
   if (sz_==0 && h.sz_==0) return *this;
      // added 2002-03-07, should be OK
   if (this==&h) return *this;
#if !defined(CPM_USECOUNT)
   Z szMem=sz_;  // added 2000-11-17
   // definition restricted to the case !defined(CPM_USECOUNT)
   // 2005-06-22 to avoid waring on non use initialized
   // variable
#endif
   iv_=h.iv_;
   sz_=h.sz_;
#if defined(CPM_USECOUNT)
   if (u_.reattach_(h.u_)) delete[] p_;
   p_=h.p_;
#else
   if (sz_!=szMem){ // added 2000-11-17 in order to avoid
      // superfluous delete, new action
      delete[] p_;
      p_=new T[sit(sz_)];
   }
   T* q=h.p_;
   T* it=p_;
   Z i=sz_;
   while (i--) *it++ = *q++;
#endif
   return *this;
}

template <class T>
```

```
V<T> V<T>::meet(IvZ const& iv)const
{
   IvZ ivRes=iv_.meet(iv);
   V<T> res(ivRes); // all components are T()
   for (Z i=res.b();i<=res.e();++i){
       if (iv.ni(i)) res[i]=(*this)[i];
   }
   return res;
}

template <class T>
V<T> V<T>::join(IvZ const& iv)const
{
   IvZ ivRes=iv_.join(iv);
   V<T> res(ivRes); // all components are T()
   for (Z i=res.b();i<=res.e();++i){
       if (iv_.ni(i)) res[i]=(*this)[i];
   }
   return res;
}

template <class T>
X2<Z,bool> V<T>::find(T const& t)const
{
   Z i0=b();
   Z j=i0-1;
   for (Z i=i0; i<=e(); ++i){
      if (t==cui(i)){ // if this never happens we still have j == i0-1
         // hence j>=i0 signals success in finding t
         j=i;
         break;
      }
   }
   return X2<Z,bool>(j,j>=i0);
}

// modified from function locate of Press et al.

template <class T>
Z V<T>::locAsc(T const& t)const
{
   if (sz_==0){
      cpmerror("V<T>::locAsc(T): array is void");
      return -137; // never happens
   }
   if (t<fir()) return b()-1;
   if (t>=last()) return e();
      // if sz_==1 we have fir()==last and then the two previous
      // conditions are an alternative: one of them holds and we
      // are ready. Thus now sz_>=2
```

```
   Z jl=0,ju=sz_;
   while (ju-jl>1){
      Z jm=(jl+ju)/2;
      if (t >= p_[jm]) jl=jm; else ju=jm; // Press et al. have > here
   }
   return jl+b();
}

template <class T>
V<T> V<T>::app(V<T> const& h)const
{
   IvZ iv2=iv_.app(h.iv_);
   Z i,sz2=iv2.car();
   T* p2=new T[sit(sz2)];
   T* it=p2;
   T* q1=p_;
   i=sz_;
   while(i--) *it++ = *q1++;
   T* q2=h.p_;
   i=h.sz_;
   while (i--) *it++ = *q2++;
   return V<T>(iv2,p2,Word());
}

template <class T>
V<T> V<T>::app(T const& t)const
{
   IvZ iv2=iv_.app(IvZ(Z(1),Z(1)));
   Z i,sz2=iv2.car();
   T* p2=new T[sit(sz2)];
   T* it=p2;
   T* q=p_;
   i=sz_;
   while(i--) *it++ = *q++;
   *it++ = t;
   return V<T>(iv2,p2,Word());
}

template <class T>
V<T> V<T>::rev()const
{
   if (sz_<2){
      return *this;
      // nothing to do if we have no or one component
   }
   else{
      T* p2=new T[sit(sz_)];
      T* it=p2;
      T* itp=p_;
      itp+=(sz_-1);
```

```
      Z i=sz_;
      while (i--)  *it++ = *itp--;
      return V<T>(iv_,p2,Word());
   }
}

template <class T>
V<T> V<T>::resize(Z newDim)const
{
   Z n2= newDim<0 ? 0 : newDim;
   Z nCopy=(n2<=sz_ ? n2 : sz_);
   V<T> res(IvZ(n2,b(),137)); // correctly initialized even for n2>sz_
   T* it=res.p_;
   T* itp=p_;
   Z i=nCopy;
   while (i--) *it++ = *itp++;
   return res;
}

template <class T>
V<T> V<T>::eli(IvZ const& iv)const
{
   IvZ ie=iv&dom();
   if (ie.isVoid()) return *this ;
   else{
      V<bool> vb(dom());
      for (Z i=vb.b();i<=vb.e();++i){
         vb.cui(i)=!ie.hasElm(i);
      }
      return select(vb);
   }
    //return eli(i[1],i.car());
}

template <class T>
V<T> V<T>::eli1(V<T>& h, Z i)const
{
// Here we use natural counting of components so that
// the j-th component of h is h[j-1].
   Z nEli=h.dim();
   if (i<1) i=1;
   if (sz_==0){ // nothing eliminated, nothing left
      h=V<T>(0);
      return V<T>(0);
   }
   else if (i>sz_){ // nothing eliminated
      h=V<T>(0);
      return *this;
   }
   else{ // now i>=1 and i<=sz_ and sz_>=1
```

```
      // if true, we have sz_>=1
   // the i-th component is the first to be eliminated
   // so we have i-1 components of *this which are on the left-hand
   // side of the elimination area. These have to shine up in
   // the result vector to be returned
      Z nRes1=i-1;
   // So the maximum number of
   // components of *this that run the risk to get eliminated is
   // sz_-nRes1
      Z nEliRisk=sz_-nRes1;
      if (nEli>nEliRisk) nEli=nEliRisk;
      Z nRes2=sz_-nRes1-nEli;
      Z nRes=nRes1+nRes2;
      T* pRes=new T[sit(nRes)];
      T* pEli=new T[sit(nEli)];
      T* itRes=pRes;
      T* itEli=pEli;
      T* itp=p_;
      Z j=nRes1;
      while (j--) *itRes++=*itp++;
      j=nEli;
      while (j--) *itEli++=*itp++;
      j=nRes2;
      while (j--) *itRes++=*itp++; // for j==0 nothing done
      h=V<T>(nEli,pEli);
      return V<T>(nRes,pRes);
   }
}

/*************************
template <class T>
V<T> V<T>::eli(Z i, Z nEli)const
{
   Word loc("V<T>::eli(Z,Z))");
   if (sz_<1 || i>sz_ || i<1 || nEli<1){
      cpmwarning(loc&": component to be eliminated does not exist");
      return *this;
   } // now sz_>=1
   Z remaining=1+sz_-i;   // number of component i and  followers
      // is >=1 due to assert()
   Z nEliActual=( nEli<=remaining ? nEli : remaining);
      // this is the number of components which we actually will be
      // removing
   Z j, sz1=sz_-nEliActual;
   if (sz1==0) return V<T>();
   else {
      T* p1=new T[sit(sz1)];
      T* it=p1;
      T* q1=p_;
      for (j=1;j<=i-1;j++) *it++ = *q1++; // the j is a counting device
```

```
         // only, no components are taken, so starting from 1 is no
         // mistake; it is a convenience.
      for (j=1;j<=nEliActual;j++) q1++;   // advancing over the elements
      // to be removed
      for (j=i;j<=sz1;j++) *it++ = *q1++;
      return V<T>(sz1,b(),p1,Word());
   }
}
*************************/

template <class T>
V<T> V<T>::ins(Z ia, V<T> const& h)const
{
   Z mL=3;
   Word loc("V<T> V<T>::ins(Z i, V<T> const& h)const");
   CPM_MA
   Z first=b();
   Z i=ia-first+1;
   if (i<1 || i>(sz_+1)) cpmerror(
      "V<T>::ins(Z i,V<T>): unvalid argument i="&cpmwrite(i));
   Z j, sz1=sz_+h.sz_;
   T* p1=new T[sit(sz1)];
   T* it=p1;
   T* q1=p_;
   T* q2=h.p_;
   for (j=1;j<=i-1;j++)   *it++ = *q1++;
   for (j=1;j<=h.sz_;j++)  *it++ = *q2++; // h.sz_ terms
   for (j=i;j<=sz_;j++)    *it++ = *q1++;
   CPM_MZ
   return V<T>(sz1,first,p1,Word());
}

template <class T>
V<T> V<T>::ins(Z ia, T const& t)const
{
   Z mL=3;
   Word loc("V<T> V<T>::ins(Z i, T const& t)const");
   CPM_MA
   Z first=b();
   Z i=ia-first+1;
   if (i<1 || i>(sz_+1))
      cpmerror("V<T>::ins(Z i,T): unvalid argument i="&cpmwrite(i));
   Z j, sz1=sz_+1;
   T* p1=new T[sit(sz1)];
   T* itp=p_;
   T* it=p1;
   for (j=1;j<=i-1;j++) *it++ = *itp++;
   *it++ =t;
   for (j=i;j<=sz_;j++)  *it++ = *itp++;
   CPM_MZ
```

```
   return V<T>(sz1,first,p1,Word());
}

template <class T>
T V<T>::in_(T const& t, bool reversed)
// safe logic by indexing, propably not utmost performance
{
   if (sz_==0) return T();
   cow_();
   Z i;
   T res;
   if (!reversed){
      res=p_[sz_-1]; // last component of array
      for (i=sz_-1;i>0;i--){
         p_[i]=p_[i-1]; // causal processing: on the right-hand side we
         // evaluate only expressions which were not yet changed during
         // the loop
      }
      p_[0]=t;
   }
   else{
      res=p_[0]; // first component of array
      for (i=0;i<sz_-1;i++){
         p_[i]=p_[i+1]; // causal processing: on the right-hand side we
         // evaluate only expressions which were not yet changed during
         // the loop
      }
      p_[sz_-1]=t;
   }
   return res;
}

// re-indexing

template <class T>
V<T> V<T>::compose(V<Z> const& w)const
{
   V<T> res(iv_);
   for (Z i=w.b();i<=w.e();i++){
     res[i]=operator()(w[i]);
   }
   return res;
}

template <class T>
V<T> V<T>::rot(Z s, Outside mode)const
{
   V<T> res(iv_);
   for (Z i=b();i<=e();++i) res[i]=operator()(i-s,mode);
   return res;
```

```
}

template <class T>
X2< V<T>, V<Z> > V<T>::condense(
   bool (*equiv)(const T&, const T&))const
// implementation based on function eclazz of the Numerical Receipes of
// Press et al.
{
   const Z mL=3;
   static Word loc("V<T>::condense()");
   CPM_MA
   Z n=dim();
   if (n<1){ // addition 2002-02-23
      CPM_MZ
      return X2< V<T>, V<Z> >(V<T>(0),V<Z>(0));
   }
   V<Z> res2(n);
   Z k,j;
   res2[1]=1;
   for (j=2;j<=n;j++) {
      res2[j]=j;
      for (k=1;k<=(j-1);k++) {
         res2[k]=res2[res2[k]];
         if ((*equiv)((*this)[j],(*this)[k])) res2[res2[res2[k]]]=j;
      }
   }
   for (j=1;j<=n;j++) res2[j]=res2[res2[j]];
   Z m=-1;
   Z rj;
   for (j=1;j<=n;j++){
      rj=res2[j];
      if (rj>m) m=rj;
   }
   V<Z> aux(m,0); // unfortunately the NR algorithm does not guarantee
      // that there are no gaps between the valid values of rj. Therfore
      // we find out the valid ones by an additional loop
   V<Z> found(m,0);
   for (j=1;j<=n;j++){
      rj=res2[j];
      if(found[rj]==0){
         found[rj]=1;
         aux[rj]=j;
      }
   }
   Z mAct=0;
   for (j=1;j<=m;j++) mAct+=found[j];
   V<T> res1(mAct);
   Z jAct=1;
   for (j=1;j<=m;j++){ // notice that aux[j] was initialized as 0
      if (aux[j]>0) res1[jAct++]=(*this)[aux[j]];
```

```
   }
   CPM_MZ
   return X2< V<T>,V<Z> >(res1,res2);
}


template <class T>
void V<T>::set_(T const& t)
{
   cow_();
   Z i=sz_;
   T* it=p_;
   while(i--) *it++ = t;
}

template <class T>
void V<T>::show(ostream& out)const
{
   out<<endl<<"V<T>::show()";
#if defined(CPM_USECOUNT)
   out<<endl<<" usecount = "<<u_.getCount();
#endif
   out<<endl<<" start address = "<<p_;
   out<<endl<<" end address = "<<(p_+(sz_-1));
   long b=1+(long)((p_+(sz_-1))-p_);
   out<<endl<<" bytes="<<b;
}

template <class T>
V<T> V<T>::set(Z i, T const& t)const
{
   Z iC=i-1;
   // now iC is a 'C-pointer index'

   if (iC<0) return *this;   // nothing changed
   else if (iC<sz_){
      // vector can already hold the new value
      V<T> res(*this);
      res[iC]=t;
      return res;
   }
   else{
      // now we have to return an enlarged vector
      Z j, sz2=iC+1;
      V<T> res(sz2);
      res[iC]=t;
      T* it=res.p_;
      T* itp=p_;
      Z i=sz_;
      while (i--) *it++ = *itp++;
```

```
      return res;
   }
}

template <class T>
T V<T>::fAcc1(T (*f)(T const&), void (*acc)(T&, T const&))const
{
   T* p1=p_;
   T res=T();
   for (Z i=0;i<sz_;i++) acc(res,f(*p1++));
   return res;
}

template <class T>
T V<T>::fAcc2(T (*f)(T const&, T const&), void (*acc)(T&, T const&),
   V<T> const& h)const
{
   T* p1=p_;
   T* p2=h.p_;
   T res=T();
   for (Z i=0;i<sz_;i++) acc(res,f(*p1++,*p2++));
   return res;
}

template <class T>
template <class Y>
Y V<T>::fAcc3(Y (*f)(T const&, T const&), void (*acc)(Y&, Y const&),
      V<T> const& h )const
{
   T* p1=p_;
   T* p2=h.p_;
   Y res=Y();
   for (Z i=0;i<sz_;i++) acc(res,f(*p1++,*p2++));
   return res;
}

template <class T>
V<T> V<T>::fAcc4(F<T2<T>,T> const& f)const
{
   Z d=dim();
   V< V<T> > aux(d,V<T>(d));
   for (Z i=0;i<d;++i){
      for (Z j=0;j<i;++j){
         T fij=f(T2<T>(li(i),li(j)));
         aux.li(i).li(j)=fij;
         aux.li(j).li(i)=-fij;
      }
   }
   T* p1=new T[sit(sz_)];
   for (Z i=0;i<sz_;i++){
```

```
      T res;
      for (Z j=0;j<sz_;j++) res+=aux.li(i).li(j);
      p1[i]=res;
   }
   return V<T>(dom(),p1,"");
}

namespace{

template <class T>
T fFunc5(T2<T> const& xij, F<R,R> const& dPot)
// This is intended to represent the force which particle i feels due to
// particle j being present.
{
   T eij=xij[1]-xij[2]; // eij points from j to i. Thus a positive
   // multiple of eij corresponds to a force on i which drives it away
   // from particle j. This such is the case of a repulsive potential.
   // This has dPot(r)/dr < 0 so that with the sign built into the formula
   // for the return value we in fact have the case of the positive
   // multiple we started with.
   R r=eij.nor_();
   return eij*(-dPot(r));
}

template <class T>
T fFunc6(T const& xi, F<R,R> const& dPot)
{
   T ei=xi; // ei points from the origin to i. Same situation as in fFunc5.
   R r=ei.nor_();
   return ei*(-dPot(r));
}

}

template <class T>
V<T> V<T>::fAcc5(F<R,R> const& dPot)const
{
#ifdef CPM_Fn
   F< T2<T>, T > f = F1< T2<T>, F<R,R>, T >(dPot)(fFunc5<T>);
#else
   F<T2<T>,T> f((std::bind(fFunc5<T>,_1,dPot)));
#endif

   Z d=dim(); // one could call fAcc4 here at the cost of an additional
      // function call. Here we ask for 'utmost efficiency'.
   V< V<T> > aux(d,V<T>(d));
   for (Z i=0;i<d;++i){
      for (Z j=0;j<i;++j){
         T fij=f(T2<T>(li(i),li(j)));
         aux.li(i).li(j)=fij;
```

```
         aux.li(j).li(i)=-fij;
      }
   }
   T* p1=new T[sit(sz_)];
   for (Z i=0;i<sz_;i++){
      T res;
      for (Z j=0;j<sz_;j++) res+=aux.li(i).li(j);
      p1[i]=res;
   }
   return V<T>(dom(),p1,"");
}

template <class T>
V<T> V<T>::fAcc6(F<R,R> const& dPot)const
{
#ifdef CPM_Fn
   F<T,T> f = F1<T,F<R,R>,T>(dPot)(fFunc6<T>);
#else
   F<T,T> f(std::bind(fFunc6<T>,_1,dPot));
#endif
   T* p1=new T[sit(sz_)];
   for (Z i=0;i<sz_;++i){
      p1[i]=f(li(i));
   }
   return V<T>(dom(),p1,"");
}


template <class T>
V<T> V<T>::fa(T (*f)(T const&))const
{
   T* p1=new T[sit(sz_)];
   T* it=p1;
   T* itp=p_;
   Z i=sz_;
   while(i--) *it++ = f(*itp++);
   return V<T>(dom(),p1,"");
}

template <class T>
V<T> V<T>::faa(T (*f)(T))const
{
   T* p1=new T[sit(sz_)];
   T* it=p1;
   T* itp=p_;
   Z i=sz_;
   while(i--) *it++ = f(*itp++);
   return V<T>(dom(),p1,"");
}
```

```
template <class T>
V<T> V<T>::fb(T (*f)(T const&, T const&), T const& t)const
{
   T* p1=new T[sit(sz_)];
   T* it=p1;
   T* itp=p_;
   Z i=sz_;
   while(i--) *it++ = f(*itp++,t);
   return V<T>(dom(),p1,"");
}

template <class T>
template <class Y>
V<T> V<T>::fb2(T (*f)(T const&, Y const&), Y const& t)const
{
   T* p1=new T[sit(sz_)];
   T* it=p1;
   T* itp=p_;
   Z i=sz_;
   while(i--) *it++ = f(*itp++,t);
   return V<T>(dom(),p1,"");
}

template <class T>
template <class Y>
void V<T>::fb2_(T (*f)(T const&, Y const&), Y const& t)
{
   cow_();
   T val;
   T* itp=p_;
   Z i=sz_;
   while(i--){
      val=*itp;
      *itp++=f(val,t);
   }
}

template <class T>
void V<T>::fc_(T (*f)(T const&, T const&), T const& t)
{
   cow_();
   T val;
   T* itp=p_;
   Z i=sz_;
   while(i--){
      val=*itp;
      *itp++=f(val,t);
   }
}
```

```
template <class T>
void V<T>::fd(T (*f)(T const&, T const&), T& t)const
{
   T* itp=p_;
   Z i=sz_;
   while (i--) t=f(*itp++,t);
}

template <class T>
V<T> V<T>::fe(T (*f)(T const&, T const&), V<T> const& h)const
{
   if (!sameDom(h)) throw Error("V<T>::fe(): domain mismatch");
   T* p1=new T[sit(sz_)];
   T* it=p1;
   T* p2=p_;
   T* p3=h.p_;
   Z i=sz_;
   while (i--) *it++ = f(*p2++,*p3++);
   return V<T>(dom(),p1,"");
}

template <class T>
template <class Y>
V<T> V<T>::fet(T (*f)(T const&, Y const&), V<Y> const& h)const
{
   if (dom()!=h.dom()) throw Error("V<T>::fet(): domain mismatch");
   T* p1=new T[sit(sz_)];
   T* it=p1;
   T* p2=p_;
  // Y* p3=h.p_;
   Y* p3=h.rep();
   Z i=sz_;
   while (i--) *it++ = f(*p2++,*p3++);
   return V<T>(dom(),p1,"");
}

template <class T>
V<T> V<T>::fe2(T (*f)(T const&, T const&), V<T> const& h)const
{
   if (sz_==h.sz_) return fe(f,h);
   else if (sz_>h.sz_){
      T t0=T();
      T* p1=new T[sit(sz_)];
      T* it=p1;
      T* p2=p_;
      T* p3=h.p_;
      Z i=sz_;
      Z ih=h.sz_;
      while (ih--){
         i--;
```

```
         *it++ = f(*p2++,*p3++);
      }
      while(i--){
         *it++ = f(*p2++,t0);
      }
      return V<T>(dom(),p1,"");
   }
   else{
      T t0=T();
      T* p1=new T[sit(h.sz_)];
      T* it=p1;
      T* p2=p_;
      T* p3=h.p_;
      Z i=sz_;
      Z ih=h.sz_;
      while (i--){
         ih--;
         *it++ = f(*p2++,*p3++);
      }
      while(ih--){
         *it++ = f(t0,*p3++);
      }
      return V<T>(h.dom(),p1,"");
   }
}

template <class T>
void V<T>::ff_(T (*f)(T const&, T const&), V<T> const& h, Z is)
{
   if (is<0) is=0;
   Z sz1=sz_-is;
   Z sz2=h.sz_;
   Z szMin=( sz1<sz2 ? sz1 : sz2);
   if (szMin<1) return; // nothing to do
   cow_();
   T val;
   T* p2=h.p_;
   T* itp=p_+is;
   Z i=szMin;
   while (i--){
      val=*itp;
      *itp++=f(val,*p2++);
   }
}

template <class T>
void V<T>::fa_(F<T,T> const& f, IvZ iv)
{
   IvZ iva=dom()&iv;
   if (iva.isVoid()) return; // nothing to be done
```

```
   cow_();
   T val;
   T* itp=p_;
   itp+=iva.inf();
      // setting the iterator pointer to the right place
   Z i=iva.car(); // this is the number of requested loop
      // actions, logic OK as the simple case i=1 tells
   while(i--){
      val=*itp;
      *itp++=f(val);
   }
}

template <class T>
void V<T>::fg_(T (*f)(T const&, T const&, T const&),
      V<T> const& h, T const& t)
{
   Z szMin=( sz_<=h.sz_ ? sz_ : h.sz_);
   cow_();
   T val;
   T* p2=h.p_;
   T* itp=p_;
   Z i=szMin;
   while (i--){
      val=*itp;
      *itp++=f(val,*p2++,t);
   }
}

template <class T >
void V<T>::fh_(T const& w_1, T const& w0, T const& w1, Outside mode)
{
   if (sz_<2) return;
   cow_();
   T v_1=read(b()-1,mode);
   T v0=p_[0];
   T v1=p_[1];
   Z k=0;
   while (k<sz_){
      p_[k]=v_1*w_1+v0*w0+v1*w1;
      k++;
      v_1=v0;
      v0=v1;
      v1=(k < sz_-1 ? p_[k+1] : read(e()+1,mode));
   }
}

template <class T >
template <class Y>
void V<T>::fht_(Y const& w_1, Y const& w0, Y const& w1, Outside mode)
```

```
{
   if (sz_<2) return;
   cow_();
   T v_1=read(b()-1,mode);
   T v0=p_[0];
   T v1=p_[1];
   Z k=0;
   while (k<sz_){
      p_[k]=v_1*w_1+v0*w0+v1*w1;
      k++;
      v_1=v0;
      v0=v1;
      v1=(k < sz_-1 ? p_[k+1] : read(e()+1,mode));
   }
}

template <class T >
template <class Y>
V<T> V<T>::fh(T (*f)(T const&, T const&, T const&, Y const&, Z),
      Y const& y, Outside mode)const
{
   if (sz_<2) return *this;
   T* p1=new T[sit(sz_)];
   Z iL=sz_-1;
   p1[0]=f(read(b()-1,mode),p_[0],p_[1],y,b());
   p1[iL]=f(p_[iL-1],p_[iL],read(e()+1,mode),y,e());
      // don't use read-functions in the loop
   T* v_1=p_;
   T* v0=v_1+1;
   T* v1=v0+1;
   T* p2=p1+1; // used to fill in values of p1 without changing p1
   Z i=b()+1;
   while (i<e()) *p2++=f(*v_1++,*v0++,*v1++,y,i++);
   return V<T>(dom(),p1,"");
}

template <class T >
template <class Y>
V<Y> V<T>::fi(Y (*f)(T const&, T const&), void (*acc)(Y&, Y const&))const
{
   Z i,j;
   Y* q=new Y[sit(sz_)];
   for (i=0;i<sz_;i++){
      Y yi=Y();
      for (j=0;j<sz_;j++){
         acc(yi,f(p_[i],p_[j]));
      }
      q[i]=yi;
   }
   return V<Y>(dom(),q);
```

```
}

template <class T >
template <class Y>
V<Y> V<T>::fj(Y (*f)(T const&, T const&), void (*acc)(Y&, Y const&))const
{
   Z i,j;
   Y* q=new Y[sit(sz_)];
   for (i=0;i<sz_;i++) q[i]=Y();
   for (i=0;i<sz_;i++){
      for (j=0;j<i;j++){
         Y fij=f(p_[i],p_[j]);
         acc(q[i],fij);
         acc(q[j],-fij);
      }
   }
   return V<Y>(dom(),q);
}

template <class T >
V<T> V<T>::select(V<bool> const& s)const
{
   Z n=dim(),ns=s.dim();
   V<T> resPrel=*this; // the actual result may have less components
   T* itPrel=resPrel.p_;
   T* itp=p_;
   Z iAct=0;
   Z i=n;
   Z is=s.b();
   while (i--){
      bool keepIt= (is<=s.e() ? s[is++] : true);
      T tAct=*itp++;
      if (keepIt){
         *itPrel++=tAct;
         iAct++;
      }
   }
   if (iAct==n) return resPrel; // no element was purged
   else return resPrel.resize(iAct);
}

template <class T >
V<IvZ> V<T>::valOn(F<T,bool> const& f)const
{
   Z mL=3;
   static Word loc("V<T>::valOn(F<T,bool>)");
   CPM_MA
   V<IvZ> res;
   bool yetFoundTrue=false;
   Z firstTrue=0, firstFalseAfterTrue=0;
```

```
   for (Z i=b();i<=e();++i){
      bool val=f((*this)[i]);
      if (val){ // we found true
         if (!yetFoundTrue){ // then start a interval of validity
            yetFoundTrue=true;
            firstTrue=i;
         }
         else{ // normally nothing to do
            // but if we are at the end of the array the
            // last pending truth interval has to be
            // considered as finished and has to be added
            if (i==e()){
               firstFalseAfterTrue=n();
               IvZ ivAct(firstTrue,firstFalseAfterTrue-1);
               res&=ivAct; // appending
               // nothing else to do since we are finished
               CPM_MZ
               return res;
            }
         }
      }
      else{ // we found false
         if (yetFoundTrue){ // then i is the terminator
            // of a truth interval
            firstFalseAfterTrue=i;
            IvZ ivAct(firstTrue,firstFalseAfterTrue-1);
            res&=ivAct;
            yetFoundTrue=false;
         }
      }
   }
   CPM_MZ
   return res;
}

template <class T>
bool V<T>::prnOn(ostream& str)const
{
   Z mL=3;
   Word loc=nameOf()&"::prnOn(...)";
   CPM_MA
   cpmwt((nameOf()&" begin").str());
   Root<IvZ>(iv_).prnOn(str);
   T* q=p_;
   for (Z i=b();i<=e();i++){
      if (CpmRoot::wrtTit){
         Word wi("// i="); // added 2012-01-19
         // same as in S<>
         wi&=cpm(i);
         bool bi=wi.prnOn(str);
```

```
         cpmassert(bi==true,loc);
      }
      if (!Root<T>(*q++).prnOn(str)){
         cpmwarning("failed to write component indexed "&cpm(i));
         cpmwarning("index range is from "&cpm(b())&" to "&cpm(e()));
         CPM_MZ
         return false;
      }
   }
   cpmwt((nameOf()&" end").str());
      // for large sz_ it would be difficult to find the
      // end of the vector data if these are written to a file
      // and a human reader wants to inspect them
   CPM_MZ
   return true;
}

template <class T>
bool V<T>::scanFrom(istream& str)
{
   Z mL=3;
   Word loc=nameOf()&"::scanFrom(...)";
   CPM_MA
   Root<IvZ> ivIn;
   bool suc = ivIn.scanFrom(str);
   if (!suc){
      cpmwarning(loc&": can't read IvZ");
      CPM_MZ
      return false;
   }
   IvZ iv=ivIn();
   Z n=iv.car();
   cpmmessage(mL,"dimension read as "&cpm(n));
   if (n>dimMax) cpmwarning(loc&": n>dimMax");
   V<T> res(iv);
   Root<T> riIn;
   for (Z i=res.b();i<=res.e();i++){
      suc = riIn.scanFrom(str);
      if (!suc){
         Word mes="failed to read component indexed "&cpm(i)&
             " index range is from "&cpm(res.b())&" to "&cpm(res.e());
         cpmwarning(mes);
         CPM_MZ
         return false;
      }
      res.cui(i) = riIn();
   }
   *this=res;
   CPM_MZ
   return true;
```

```
}

template <class T >
Z V<T>::com(V<T> const& s)const
// short vectors < longer vectors
{
   Z d1=dim(), d2=s.dim();
   if (d1<d2) return 1;
   else if (d1>d2) return -1;
   else{
      T* q=p_;
      T* qs=s.p_;
      for (Z i=0;i<d1;i++){
         Z ci=Root<T>(*q++).com(*qs++);
         if (ci!=0) return ci;
      }
      return 0;
   }
}

/********************** iterated templates *************************/
// describing multi-indexed quantities
// Notice that these all have automatically the member functions of V
// defined!

#define CPM_V1 V<T>
#define CPM_V2 V<V<T> >
#define CPM_V3 V<V<V<T> > >
#define CPM_V4 V<V<V<V<T> > > >

/************************* class VV ***********************************/
template <class T>
class VV: public CPM_V2{ // matrices

   typedef CPM_V2 Base;

public:

// constructors

   VV(Z d1, Z d2):CPM_V2(d1,CPM_V1(d2)){}
   VV(Z d1, Z d2, T const& t):CPM_V2(d1, CPM_V1(d2,t)){}
      // all components are equal to t
   VV(void):CPM_V2(){}
   VV(CPM_V2 const& x):CPM_V2(x){}

   V<T> lin()const;
      // 'linear version of matrix'
      // by appending all rows
```

```
// dimension access
   Z dim1()const{ return Base::dim();}
   Z dim2()const{ return (*this)[1].dim();}

   virtual Z size()const // virtual in base V<...>
      { return dim1()==0 ? 0 : dim1()*dim2();}

// component access
   const T& operator()(Z i, Z j)const
      // returns T() for out of range indexes
   { return Base::read(i).read(j);}

   T& operator()(Z i, Z j)
   { return (*this)[i][j];}

   VV<T> trn()const
      //: transpose
      // Returns the transposed matrix
   {
      VV<T> res(dim2(),dim1());
      for (Z i=1;i<=dim1();i++){
         for (Z j=1;j<=dim2();++j){
            res[j][i] = (*this)[i][j];
         }
      }
      return res;
   }

   template <class Y>
   V<Y> each(void (*f)(T const&, Y&))const;
      //: each
      // collecting the application of f on every pixel in a linear array

   template <class Y>
   VV<Y> operator()(Y (*f)(T const&))const;
      //: operator()
      // creating a matrix with a transformed value range
};

template <class T>
template <class Y>
V<Y> VV<T>::each(void (*f)(T const&, Y&))const
{
   Z m1=dim1(),m2=dim2(),k=1,i1,i2;
   V<Y> res(m1*m2);
   for (i1=1;i1<=m1;i1++){
      for (i2=1;i2<=m2;i2++){
         f((*this)[i1][i2],res[k++]);
      }
   }
```

```
   return res;
}

template <class T>
template <class Y>
VV<Y> VV<T>::operator()(Y (*f)(T const&))const
{
   Z m=dim1(),n=dim2(),i,j;
   VV<Y> res(m,n);
   for (i=1;i<=m;i++){
      for (j=1;j<=n;j++){
         res[i][j]=f((*this)[i][j]);
      }
   }
   return res;
}

template <class T>
V<T> VV<T>::lin()const
{
   if (dim1()==0) return V<T>(0);
   else{
      V<T> res=(*this)[1];
      for (Z i=2;i<=dim1();i++) res&=(*this)[i];
      return res;
   }
}

/************************ class VVV***********************************/
// tensors of rank 3

template <class T>
class VVV: public CPM_V3{ // tensors of rank 3

      typedef CPM_V3 Base;

public:

// constructors

   VVV(Z d1, Z d2, Z d3, T  const& t):CPM_V3(d1,CPM_V2(d2,CPM_V1(d3,t))){}
   VVV(Z d1, Z d2, Z d3):CPM_V3(d1,CPM_V2(d2,CPM_V1(d3))){}
   VVV(void):CPM_V3(){}
   VVV(CPM_V3 const& x):CPM_V3(x){}

// dimension access
   Z dim1()const{ return Base::dim();}
   Z dim2()const{ return (*this)[1].dim();}
   Z dim3()const{ return (*this)[1][1].dim();}
```

```
   virtual Z size()const; // virtual in base V<...>

// component access

   const T & operator()(Z i, Z j, Z k)const
   // returns T() for out of range indexes
   { return Base::read(i).read(j).read(k);}

   T & operator()(Z i, Z j, Z k)
   { return (*this)[i][j][k];}
};

template <class T>
Z VVV<T>::size()const
{
   Z d1=dim1();
   if (d1==0) return d1;
   else{
      Z d2=dim2();
      if (d2==0) return d2;
      else{
         Z d3=dim3();
         if (d3==0) return d3;
         else return d1*d2*d3;
      }
   }
}

/********************** class VVVV ***********************************/
template <class T>
class VVVV: public CPM_V4{ // tensors of rank 4

      typedef CPM_V4 Base;

public:

// constructors

   VVVV(Z d1, Z d2, Z d3, Z d4, T const& t):
   CPM_V4(d1,CPM_V3(d2,CPM_V2(d3,CPM_V1(d4,t)))){}
   VVVV(Z d1, Z d2, Z d3, Z d4):
   CPM_V4(d1,CPM_V3(d2,CPM_V2(d3,CPM_V1(d4)))){}
   VVVV(void):CPM_V4(){}
   VVVV(CPM_V4 const& x):CPM_V4(x){}

// dimension access
   Z dim1()const{ return Base::dim();}
   Z dim2()const{ return (*this)[1].dim();}
   Z dim3()const{ return (*this)[1][1].dim();}
   Z dim4()const{ return (*this)[1][1][1].dim();}
```

```
   virtual Z size()const; // virtual in base V<...>

// component access

   T const& operator()(Z i, Z j, Z k, Z l)const
  // returns T() for out of range indexes
   { return Base::read(i).read(j).read(k).read(l);}

   T& operator()(Z i, Z j, Z k, Z l)
   { return (*this)[i][j][k][l];}
};

template <class T>
Z VVVV<T>::size()const
{
   Z d1=dim1();
   if (d1==0) return d1;
   else{
      Z d2=dim2();
      if (d2==0) return d2;
      else{
         Z d3=dim3();
         if (d3==0) return d3;
         else{
            Z d4=dim4();
            if (d4==0) return d4;
            else return d1*d2*d3*d4;
         }
      }
   }
}
#undef CPM_V4
#undef CPM_V3
#undef CPM_V2
#undef CPM_V1

}  // namespace CpmArrays

namespace CpmRoot{
// This is a pattern for partial specializations --- to which
// the specializations to class templates belong --- of the
// IO class template started in cpmnumbers.h and the
// Name class template started in cpmword.h.
// A corresponding definition is needed for all non-C+- classes which
// shall be used as template arguments of C+- containers and
// are expected to provide then the same functionality as
// corresponding C+- classes.

   template<class T>
```

```
   class IO< std::vector<T> >{ // partial specialization
    public:
      IO(){}
      bool o(std::vector<T> const& v, ostream& str)const
      { return CpmArrays::V<T>(v).prnOn(str);}
      bool i(std::vector<T>& v, istream& str)const
      {
         CpmArrays::V<T> vl;
         bool res=vl.scanFrom(str);
         v=vl.std();
         return res;
      }
   };

#if defined(CPM_NAMEOF)
   template<class T>
   class Name< std::vector<T> >{ // partial specialization
   public:
      Name(){}
      Word operator()(std::vector<T> const& vt )const
      { return Word("std::vector<"&CpmRoot::Name<T>()(T())&">");}
   };
   template<class T>
   class Name< std::set<T> >{ // partial specialization
   public:
      Name(){}
      Word operator()(std::set<T> const& vt )const
      { return Word("std::set<"&CpmRoot::Name<T>()(T())&">");}
   };
#endif
}
#endif
```

44 cpmviewport.h

```
//? cpmviewport.h
//? Status of work 2023-10-20.
//?
//?  ...


#ifndef CPM_VIEWPORT_H_
#define CPM_VIEWPORT_H_
/*

   class Viewport is the Cpm interface to the graphical
   capabilities of the hardware.

*/
#include <cpmv.h>

namespace CpmGraphics{

   using CpmRoot::Z;
   using CpmRoot::R;
   using CpmRoot::B;
   using CpmRoot::L;
   using CpmRoot::Word;
   using CpmArrays::V;
   using CpmArrays::IvZ;
   using namespace CpmStd;
      // see cpmnumbers.h, only I/O related stuff from std.
      // I had using namespace std before. This had the bad effect that
      // one of my function names ('ignore' in this case), even if defined
      // in an anonymous namespace, once clashed with std::ignore.
      // So, to avoid clashes with the ever expanding namespace std I will
      // never rely on std as a whole.

#if defined(WIN32)
   const Word Platform("Windows");
#else
   const Word Platform("Cygwin or Linux");
#endif

////////////////////////////////// struct xy ////////////////////////////
// Simple data structure for graphical points. As ever in graphics:
// x-axis horizontal from left to right (thus perfectly normal)
// y-axis downwards. Meaningful coordinates belong to {0,1,2,...}.
   struct xy{ // graphical points
      Z x{0},y{0};
      xy(){}
      xy(Z a, Z b):x(a),y(b){}
```

```
      xy operator+(xy p)const{ return xy(x+p.x,y+p.y);}
      xy operator-(xy p)const{ return xy(x-p.x,y-p.y);}
      xy operator-()const{ return xy(-x,-y);}
      xy& operator+=(xy p){ x+=p.x;y+=p.y;return *this;}
      xy& operator-=(xy p){ x-=p.x;y-=p.y;return *this;}
      bool operator==(xy p)const{ return x==p.x && y==p.y;}
   };

// simple data structure for graphical rectangles

   struct xyxy{ // graphical rectangles
      Z x1{0},y1{0},x2{0},y2{0}; // (x1,y1) left upper corner,
         // (x2,y2) right lower corner

      xyxy(){}

      xyxy(Z a,Z b,Z c,Z d):x1(a),y1(b),x2(c),y2(d){}
         // construction from corner coordinates

      xyxy(xy p1, xy p2):x1(p1.x),y1(p1.y),x2(p2.x),y2(p2.y){}
         // construction from corners

      xyxy(xy p, Z w, Z h):x1(p.x),y1(p.y),x2(x1+w-1),y2(y1+h-1){}
         // construction from left upper corner, width, and height

      Z w()const{ return x2-x1+1;}

      Z h()const{ return y2-y1+1;}

      R aspRat()
      //: aspect ratio
      { Z h_=h(); cpmassert(h_>=1,"xyxy::aspRat()"); return R(w())/h_;}

      bool isVoid()const{ return x1<0;}
         // since the meaningfull graphical rectangles have no
         // negative components, this is an easy way to encode
         // exceeptional behavior

      xyxy operator+(xy p)const
         { return xyxy(x1+p.x,y1+p.y,x2+p.x,y2+p.y);}
      xyxy operator-(xy p)const
         { return xyxy(x1-p.x,y1-p.y,x2-p.x,y2-p.y);}
      xyxy& operator+=(xy p){ return *this=*this+p;}
      xyxy& operator-=(xy p){ return *this=*this-p;}

      xyxy operator|(xyxy r)const;
         // smallest xyxy that contains both *this and r
         // kind of union

      xyxy& operator|=(xyxy r){ return *this=(*this)|r;}
```

```
      // | as a mutating operation

   xyxy operator&(xyxy r)const;
      // returns the set section of *this and r.
      // The result res satisfies res.isVoid()==true
      // if this section is the void set.

   xyxy& operator&=(xyxy r){ return *this=(*this)&r;}
      // & as a mutating operation

   xy operator[](Z i)const{ return i<=1 ? xy(x1,y1) : xy(x2,y2);}
      // For an instance r of xyxy r[1] is the upper left corner
      // and r[2] is the lower right corner.

   xy cor(Z i)const
   //: corner
   // Returns the i-th corner. Here corners are numbered counter
   // clock-wise starting with the left upper corner as 1. All
   // non-matching arguments give corner 1. Thus no exceptions!
   {
      if (i==2) return xy(x1,y2);
      if (i==3) return xy(x2,y2);
      if (i==4) return xy(x2,y1);
      return xy(x1,y1);
   }

   bool hasElm(xy const& p)const
   //: has element
   // Makes a xyxy a set of xy's
   {
      return IvZ(x1,x2).hasElm(p.x)&& IvZ(y1,y2).hasElm(p.y);
   }

   xyxy scaleToFit(xyxy r)const;
   //: scale to fit
   // Returns the largest rectangle that
   // 1. has the aspect ratio of r
   // 2. has the same left-upper corner as *this
   // 3. is contained in *this.

   friend ostream& operator<<(ostream& os, xyxy const& r)
   {
      os<<" "<<r.x1<<", "<<r.y1<<", "<<r.x2<<", "<<r.y2<<endl;
      return os;
   }
};

struct rgb;
class Img24;
```

```
////////////////////////// class ColRef ////////////////////////////////

const float i255=1./255;

   class ColRef{ //lean 24-bit color-values
      // light 24-bit representation of color-values
      // that easily converts (back and forth) to the low-level color
      // data

      friend struct rgb;
      friend class Img24;
      static Z x(Z z);
      typedef ColRef Type;
   public:
      static Z lowerLimit;
      static Z upperLimit;
      L r,g,b; // all values valid, so public OK
      CPM_IO
      CPM_ORDER
      ColRef():r((L)lowerLimit),g((L)lowerLimit),b((L)lowerLimit){}
      ColRef(Z r_, Z g_, Z b_):r((L)x(r_)),g((L)x(g_)),b((L)x(b_)){}
      ColRef(L r_, L g_, L b_):r(r_),g(g_),b(b_){}
      L getR()const{ return r;}
      L getG()const{ return g;}
      L getB()const{ return b;}
      L operator[](Z i)const
         { if (i==3) return b; else if (i==2) return g; else return r;}
      float glR()const{ return i255*r;}
      float glG()const{ return i255*g;}
      float glB()const{ return i255*b;}
      bool isGray()const{ return r==g && g==b;}
      ColRef add(ColRef const& cr)const
      { return ColRef((Z)r+(Z)cr.r,(Z)g+(Z)cr.g,(Z)b+(Z)cr.b);}
      Word nameOf()const{ return "ColRef";}
   };

////////////////////////// struct rgb ////////////////////////////////

struct rgb { // Z-valued red green blue
   // a simple struct to provide a convenient interface for
   // functions draw. This class will be a base class for deriving
   // a more functional class Color in namespace CpmImaging.
   Z r,g,b;

   rgb():r(ColRef::lowerLimit),g(ColRef::lowerLimit),
      b(ColRef::lowerLimit){}

   rgb(Z r_, Z g_, Z b_, bool norm=true):r(r_),g(g_),b(b_)
      {if (norm) normalize();}
```

```
rgb(R r_, R g_, R b_, bool norm=true):
r(cpmrnd(r_)),g(cpmrnd(g_)),b(cpmrnd(b_))
{if (norm) normalize();}

explicit rgb(ColRef cr):r(cr.r),g(cr.g),b(cr.b){}

virtual Word nameOf()const{ return Word("rgb");}

operator ColRef()const{ return ColRef(r,g,b);}

rgb operator~()const
   // returning the complementary color
{
   Z up=ColRef::upperLimit; return rgb(up-r,up-g,up-b);
}

void normalize(){ r=ColRef::x(r); g=ColRef::x(g); b=ColRef::x(b);}
   // ensuring the right value range by simply cutting

static rgb mix(rgb c1, rgb c2, R w1, R w2, R gamma);
   // mixing colors according to function mixingRule

R gray()const{return (r+g+b)/3.;}

bool isGray()const{ return r==g && g==b;}
   //: is gray

bool isBlack()const{ return r==0 && g==0 && b==0;}
   //:: is black

bool isWhite()const{ return r==255 && g==255 && b==255;}
   //:: is white

bool isDark()const{ return r+g+b<128;}
   //: is dark

rgb operator*(R fac)const
{ return rgb(cpmrnd(fac*r),cpmrnd(fac*g),cpmrnd(fac*g));}

static Z mixingRule(Z z1, Z z2, R w1, R w2, R gamma);
   // defines mixing of colors according to given weights
   // w1+w2=1 is not assumed
   // ( w1=w2=1 corresponds to addition of light)

void write(ostream& str)const
{ str<<endl<<"rgb: r="<<r<<" g="<<g<<" b="<<b<<endl;}

friend void operator<<(ostream& str, rgb const& c)
{ str<<c.r; str<<c.g; str<<c.b;}
```

```
   friend void operator>>(istream& str, rgb & c)
   { str>>c.r; str>>c.g; str>>c.b;}
};

////////////////////////////// class Font ////////////////////////////////
// Presently only Helvetica 12 needed from this font system

class Font{
// only Helvetica 12 needed from this font system of GLUT
   void* name;
   Z h;
      // height in pixels
   Z mw;
      // maximum width in pixels (estimation based on height)
   Z d;
      // suitable line separation taking decent ('Unterlaenge') into
      // account
public:
   explicit Font(Z type=4);
   explicit Font(R height);
      // creates an instance of a font height that comes closest
      // to the value given by the argument. See the simple
      // implementation
      // code for the details

   Z getHeight()const{ return h;}
   Z getWidth()const{ return mw;}
      // maximum width in pixels (typically width of W in proportional
      // fonts
   Z getLineSep()const{ return d;}
   void* getName()const{ return name;}
};

////////////////////////// class Rec ////////////////////////////////////
// Pixel rectangle rec_ which always fits Viewport::win()
// and linear image buffer mem_ of arbitrary size.
// The display function vis of this class shows mem_
// always whithin rec_. This function is Cpm's means
// to bring pixel-based graphical information to screen.
// Present implementation relies on OpenGL (only if
// CPM_NOGRAPHICS is not set). If CPM_NOGRAPHICS is
// defined we need no graphics libraries from the system
// and can generate all graphics as PPM-images (on an
// pixel rectangle that can be set arbitrarily
// by editing cpmconfig.ini.

class Rec{ // pixel rectangle which always fits Viewport::win()
   xyxy rec_;
   Z n_;
      // dimension of mem_
```

```
   std::vector<L> mem_;
      // should be unsigned char = L
      // Data element of this type needed in function vis()
      // by OpenGL. This is linearized data in two-fold manner:
      // The rectangular image area is un-fold to a linear chain,
      // and the r g b pixel data are represented as a list of
      // characters.

public:
   static B db;
       // double buffering (this quantity can be reset by an entry to
       // cpmconfig.ini). For non-static data members input from ini-files
       // does not work.
   explicit Rec(xyxy r=xyxy(0,0,0,0));
      // Constructor which fills the image data from the one and only
      // initialized Viewport, more precisely from the data
      // within the set section of r and the Viewport rectangle
      // Viewport::win()
      // Present implementation relies on OpenGL
   explicit Rec(V< V<ColRef> > const& bh);
      // constructs a Rec from the matrix data bh. The corresponding
      // rectangle rec_ is created such that it fits Viewport::win().
   Rec(xyxy r, L c);
      // The pixels within the rectangle r are set equal to c
      // On display this is always a gray (black to white) frame.
      // Notice that colorful colors would be more difficult to
      // handle here.
      // Rectangle is created such that it fits Viewport::win().
  // Rec(Rec const& r);
   ~Rec()=default;
  // Rec& operator=(Rec const& r);
   Z dim()const{ return n_;}
      // number of bytes = 3*number of pixels
   Z h()const{ return rec_.h();}
   Z w()const{ return rec_.w();}
   Z x1()const{ return rec_.x1;}
   Z y1()const{ return rec_.y1;}
   Z x2()const{ return rec_.x2;}
   Z y2()const{ return rec_.y2;}
   xyxy rec()const{ return rec_;}
   xy operator[](Z i)const{ return rec_[i];}
   xy cor()const{ return rec_[1];}
      //: corner
      // actually left upper corner
   Rec& operator+=(xy shift);
      // result is always within Viewport::win()
   Rec operator+(xy shift)const{ Rec res=*this; res+=shift; return res;}
   Rec& operator-=(xy shift);
      // result is always within Viewport::win()
   Rec operator-(xy shift)const{ Rec res=*this; res-=shift; return res;}
```

```
   void vis()const;
      //: visualize
      // Displays the image data in mem_ on the area rec_ (which always
      // is a subset of the viewpoint window).
      // On this function all bitmap imaging in C+- is based.
      // Present implementation relies on OpenGL. Function glDrawPixels
      // needs a L* as the last argument. So it would not be convenient to
      // replace the mem_ data element of the present class by one of type
      // V<L>.
   V< V<ColRef> > fold()const;
      // returns a matrix-like memory version of mem_
};

////////////////////////// class Viewport ///////////////////////////////
/*
class Viewport is a lean interface to the
systems graphical capabilities. It also defines and activates a
statusbar supporting a continuous message stream from a running
program going to a dedicated area of the screen.

In order for the OpenGL version to work (which is the only one that
survived, I had working implementations based on MSFC and on FLTK)
one has to do some initialisation in function main() as is done in
cpmapplication.cpp.

The class as it stands is a rather 'blind viewport'. Apart from filling
rectangles with chosen color and setting text in GLUT's native fonts
it allows  n o t to perform actions which can be seen on screen.
The viewport becomes more active only with class Img24 (see cpmimg24.h)
which provides a data element of type Rec (its name is displayBuffer) and
tools to fill and display it. Each call of an constructor of class
CpmGraphics::Frame activates Img24.
See in particular CpmGraphics::FramesetStaticSize(Z width, Z height)
which is called in int main(int argc, char *argv[]) when this function
is defined in cpmapplication.cpp
*/

class Viewport{ // Lean interface to the system's graphical capabilities.
   // The Viewport class has no non-static data member, so
   // every member function could be declared static without changing
   // their behavior. However, it is more convenient to say
   //
   // Viewport vp;
   // vp.addText(10,10,"hello world");
   //
   // than
   //
   // Viewport::addText(10,10,"hello world");
   //
   // Notice that there is no way to have more than one different
```

```
   // instances of Viewport. So we can control a single graphical area,
   // which in my normal applications is one 'application window'.
   // Classes Frame, Graph, and Frames organize as many rectangular
   // subframes of this basic window as we want.

   static bool initialized;
   static Z applWindowWidth;
   static Z applWindowHeight;
   static Z fullWindowHeight;

   static xyxy fullWindow; // whole area accessible through the Viewport
      // (i.e. the single and only instance of Viewport that may exist).
   static xyxy window;
      // fullWindow minus status bar, also called 'application window'

   static R gamma;
      // a gamma-value for the systems video screen
      // presently set to 2.2 (usual in Windows systems)
      // in code. Would not be illogical to read this in from
      // cpmsystemdependencies.h.
      // This value of gamma will be used as default for the
      // image class Img24.

   static xyxy seg1;
   static xyxy seg2;
   static xyxy seg3;
   static xyxy seg4;
      // segments (panes) of the status bar
   static ColRef colText;
   static ColRef colWindow;
   static ColRef cs1;
   static ColRef cs2;
   static ColRef cs3;
   static ColRef cs4;
      // colors of the status bar segments
   static Z heightStatusBar;
   static Z size;
   static Z rank;

public:
   static void setColText(Z r, Z g, Z b){ colText=ColRef(r,g,b);}
   static void setColText(ColRef cr){ colText=cr;}
   static void setColText(rgb crgb){ colText=(ColRef)crgb;}

   static ColRef getColText(){ return colText;}
   static ColRef getColWindow(){ return colWindow;}

   static xyxy win(){ return window;}
      // Window available for regular display activities (that do
      // avoid writing to the status bar).
```

```
static xyxy fullWin(){ return fullWindow;}
   // Graphics window including the status bar.

static Z getHeightStatusBar(){ return heightStatusBar;}
static void onStatusBar(const Word& w, Z i);
static void clrPan();
   //: clear panes
static void fill(xyxy rec, ColRef cr);
   // Fills rectangle rec with solid color cr
static void placeWord(Z tx, Z ty, Word const& w,
   const Font& font, Z wMax=500);
   // wMax sets the maximum number of pixels for the length of the
   // printed picture of w. If w is longer it gets cut ( e.g. for
   // disciplined writing on status bars.
static xyxy textLine(Z tx, Z ty, Word const& w,
   ColRef crText, R h);
   // writing w in color crText.
   // Only the pixels making up the characters are written so that
   // previous screen content is only minimally shadowed by the text.
   // The text may not be easy to read, however.
   // h is a proposal for the font hight (the actually chosen
   // hight depends on the available fonts). Introduced
   // 2005-04-14
   // The return value is a enclosing box for the text - a bit larger
   // than the minimum enclosing box.

Viewport(Z w, Z h, Word const& title);
Viewport(){}

static R getGamma(){ return gamma;}
static void setGamma(R g){ gamma=g;}
static bool isInitialized(){return initialized!=0;}
   // does not allow to change status !!
static Z pelX(){ return applWindowWidth;}
static Z pelY(){ return applWindowHeight;}

Z getCol()const{return applWindowWidth;}
   // number of columns
   // horizontal extension of the graphically accessible
   // application window in pixels. The range of the first pixel
   // access index (mostely named x or i) is {0,1,...,getCol()-1}
Z getWidth()const{return applWindowWidth;}
   // legalizing another conventional name

Z getLin()const{ return applWindowHeight;}
   // number of lines (rows)
   // vertical extension of the graphically accessible
   // application window in pixels. The range of the second pixel
   // access index (mostely named y or j) is {0,1,...,dim2()-1}
Z getHeight()const{ return applWindowHeight;}
```

```
      // legalizing another conventional name

   Z getFullHeight()const{return fullWindowHeight;}

   Z dim1()const{return applWindowWidth;}
   Z dim2()const{return fullWindowHeight;}
      // The screen area under control of Viewport consists of
      // dim1()*dim2() writable and readable pixels.
      // This

      // In the case
      // of OpenGL-implementation a part of the lines are used as
      // forming a statusbar. So these should not be used for
      // displaying images and graphics (but they perfectly
      // can from a syntactic point of view).

   Z getRank()const{ return rank;}
   Z getSize()const{ return size;}

   xyxy addText(Z x1, Z y1, Word const& text, R fontHeight=12, Z font=1);
      // (x1,y1) is the upper left corner of the writing box.
      // The return value is a box which encloses the text and is a bit
      // larger than minimal.
      // Thus the last argument remains passive since we select the font
      //  the height of which comes closest to fontHeight.

      // Here we are the names of the available fonts together with a
      // number which does not matter here.
      // font = 1 : GLUT_BITMAP_TIMES_ROMAN_10
      // font = 2 : GLUT_BITMAP_HELVETICA_10
      // font = 3 : GLUT_BITMAP_HELVETICA_12
      // font = 4 : GLUT_BITMAP_8_BY_13
      // font = 5 : GLUT_BITMAP_9_BY_15
      // font = 6 : GLUT_BITMAP_HELVETICA_18
      // font = 7 : GLUT_BITMAP_TIMES_ROMAN_24
};

}// namespace

#endif
```

45 cpmviewport.cpp

```
//? cpmviewport.cpp
//? Status of work 2023-10-20.
//?
//?  ...


#include <cpmsystemdependencies.h>
   // This file is not included in any header file in order
   // to make only those translation units dependent on these
   // macros which really depend on them.
   // Important that this precedes all other includes.
#include <cpmviewport.h>
//#include <cpmtypes.h> // experiment, needed for cpmdebug

using CpmRoot::Z;
using CpmRoot::L;
using CpmRoot::R;
using CpmRoot::B;
using CpmRoot::Word;
using CpmRoot::getByte;

using CpmSystem::Message;

using CpmArrays::V;
using CpmArrays::LEAN;

using CpmGraphics::Rec;
using CpmGraphics::xy;
using CpmGraphics::xyxy;
using CpmGraphics::Viewport;
using CpmGraphics::rgb;
using CpmGraphics::ColRef;

using namespace std;

Z CpmGraphics::Viewport::heightStatusBar=16;
////////////////////////// struct xyxy ////////////////////////////////////
namespace {
Z infZ(Z i, Z j){ return i<=j ? i : j;}
Z supZ(Z i, Z j){ return i<=j ? j : i;}
R infR(R i, R j){ return i<=j ? i : j;}
R supR(R i, R j){ return i<=j ? j : i;}
   // to avoid need of namespace CpmRootX
}

xyxy xyxy::operator|(xyxy r)const
{
```

```
   Z rx1=infZ(x1,r.x1);
   Z ry1=infZ(y1,r.y1);
   Z rx2=supZ(x2,r.x2);
   Z ry2=supZ(y2,r.y2);
   return xyxy(rx1,ry1,rx2,ry2);
}

xyxy xyxy::operator&(xyxy r)const
{
   if (x2 < r.x1) return xyxy(-1,-1,-1,-1);
   if (r.x2 < x1) return xyxy(-1,-1,-1,-1);
      // if upper end of one is smaller than the lower
      // end of the other, then the projections of
      // *this and r to the x-axis are disjoint
   if (y2 < r.y1) return xyxy(-1,-1,-1,-1);
   if (r.y2 < y1) return xyxy(-1,-1,-1,-1);
      // same for x replaced by y
   Z rx1=supZ(x1,r.x1);
   Z ry1=supZ(y1,r.y1);
   Z rx2=infZ(x2,r.x2);
   Z ry2=infZ(y2,r.y2);
   return xyxy(rx1,ry1,rx2,ry2);
}

xyxy xyxy::scaleToFit(xyxy r)const
{
   Word loc("xyxy::scaleToFit(xyxy)");
   cpmassert(!r.isVoid()&&!isVoid(),loc);
   R w1=w(), h1=h(), w2=r.w(), h2=r.h();
   R sw=R(w1)/w2; // scale factor to adjust width
   R sh=R(h1)/h2; // scale factor to adjust height
   Z wN,hN; // width and hight of the result
   R s=infR(sw,sh);
   wN=cpmtoz(s*w2);
   hN=cpmtoz(s*h2);
   return (*this)&xyxy(cor(1),wN,hN);
 }

////////////////////////// class Rec ////////////////////////////////////

B Rec::db{true}; // now works also for the panes

Rec& Rec::operator+=(xy shift)
{
   rec_+=shift;
   rec_&=Viewport::win();
   return *this;
}
Rec& Rec::operator-=(xy shift)
{
```

```
   (rec_-=shift)&=Viewport::win(); return *this;
}

Rec::Rec(xyxy r): rec_(r&Viewport::win()), n_(rec_.w()*rec_.h()*3),
   mem_(vector<L>(n_))
{
   Z mL=3;
   Word loc("Rec::Rec(xyxy r)");
   CPM_MA
#if !defined(CPM_NOGRAPHICS)
   Z x1=rec_.x1, y1=rec_.y1;
   Z w=rec_.w(), h=rec_.h();
   Z wBytes=w*3; // bit size of a line from rec
   Z gly=Viewport::fullWin().h()-y1-h;
   Z k=0;
   for (Z j=h; j>=0; j--){ // reading line by line
     L* lineBuff=new L[wBytes];
     glReadPixels(x1,gly++,w,1,GL_RGB,GL_UNSIGNED_BYTE,lineBuff);
     for (Z i=0;i<wBytes;++i) mem_[k++]=lineBuff[i];
     delete[] lineBuff;
   }
#endif
   CPM_MZ
}

Rec::Rec(xyxy r, L c): rec_(r&Viewport::win()),
   n_(rec_.w()*rec_.h()*3),mem_{vector<L>(n_,c)}{}
   // The pixels within the rectangle r are set equal to c
   // On display this is always a gray (black to white) frame.
   // Notice that multiple colors would be more difficult to
   // handle here.

Rec::Rec(V< V<ColRef> > const& bh)
{
   Z h1=bh.dim();
   Z w1=bh.fir().dim();
   xyxy r1(xy(0,0),w1,h1);
   rec_=r1&Viewport::win();
   Z height=rec_.h();
   Z width=rec_.w();
   n_=height*width*3;
   mem_=vector<L>(n_);
   Z k=0;
   for (Z i=height-1;i>=0;i--){
      for (Z j=0;j<width;j++){
         ColRef val=bh[i][j];
         mem_[k++]=val[1];
         mem_[k++]=val[2];
         mem_[k++]=val[3];
      }
```

```
   }
}

V< V<ColRef> >  Rec::fold()const
{
   Z w1=w(),h1=h();
   V<V<ColRef> > res(h1,V<ColRef>(w1,LEAN),0);
   Z k=0;
   for (Z i=h1-1;i>=0;--i){
      for (Z j=0;j<w1;++j){
         L r=mem_[k++];
         L g=mem_[k++];
         L b=mem_[k++];
         res[i][j]=ColRef(r,g,b);
      }
   }
   return res;
}

void Rec::vis()const
// screen representation of a linear block of pixel values.
// Basicly this is all we need from OpenGL
{
   Word loc("Rec::vis()const");
   Z mL=3;
   CPM_MA;

#if !defined(CPM_NOGRAPHICS)
   Z h1=rec_.h();
   Z w1=rec_.w();
   glRasterPos2i(rec_.x1,rec_.y1+h1);
   L* buff=new L[n_];
   for (Z k=0;k<n_;++k) buff[k]=mem_[k];
   glDrawPixels(w1,h1,GL_RGB,GL_UNSIGNED_BYTE,buff);
   glFlush();
   if (Rec::db) glutSwapBuffers();
      // since 2022-10-02 double buffering works alo for the panes.
   delete[] buff;
#endif
   CPM_MZ
}

////////////////////// class ColRef ////////////////////////

Z ColRef::lowerLimit=0;
Z ColRef::upperLimit=255;
   // no surprises

Z ColRef::x(Z z)
{
```

```
   if (z>ColRef::upperLimit) return ColRef::upperLimit;
   else if (z<ColRef::lowerLimit) return ColRef::lowerLimit;
   else return z;
}

bool ColRef::prnOn(ostream& str)const
{
   cpmwt("ColRef");
   cpmp(r);
   cpmp(g);
   cpmp(b);
   return true;
}

bool ColRef::scanFrom(istream& str)
{
   cpms(r);
   cpms(g);
   cpms(b);
   return true;
}

Z ColRef::com(ColRef const& obj)const
{
   cpmc(r);
   cpmc(g);
   cpmc(b);
   return 0;
}

//////////////////// class rgb ////////////////////////////

Z CpmGraphics::rgb::mixingRule(Z z1, Z z2, R w1, R w2, R gamma)
{
   if (gamma==0)
      return ColRef::upperLimit;
   else if (gamma==1)
      return cpmtoz(w1*z1+w2*z2);
   else if (gamma==2)
      return cpmtoz(sqrt(w1*z1*z1+w2*z2*z2));
   else if (gamma>0 && gamma<10){
      R p1=w1*cpmpow(R(z1),gamma);
      R p2=w2*cpmpow(R(z2),gamma);
      R gammaInv=1./gamma;
      return cpmtoz(cpmpow(p1+p2,gammaInv));
   }
   else
      return (z1>=z2 ? z1 : z2);
}
```

```
rgb CpmGraphics::rgb::mix(rgb c1, rgb c2, R w1, R w2, R gamma)
{
   Z a=mixingRule(c1.r,c2.r,w1,w2,gamma);
   Z b=mixingRule(c1.g,c2.g,w1,w2,gamma);
   Z c=mixingRule(c1.b,c2.b,w1,w2,gamma);
   rgb res(a,b,c);
   res.normalize();
   return res;
}

////////////////////////// class Viewport ////////////////////////////

// common initializations of static variables


R CpmGraphics::Viewport::gamma=2.3;
// best guess from a Kodak tool GAMMA.TIF for my office monitor
// is 2.3, also for my TFT-Monitor this seems to be the best
xyxy CpmGraphics::Viewport::window;
xyxy CpmGraphics::Viewport::fullWindow;
Z CpmGraphics::Viewport::size=1;
Z CpmGraphics::Viewport::rank=1;

#if !defined(CPM_NOGRAPHICS)
using CpmGraphics::Font;
bool CpmGraphics::Viewport::initialized=false;
Z CpmGraphics::Viewport::applWindowWidth=0;
Z CpmGraphics::Viewport::applWindowHeight=0;
Z CpmGraphics::Viewport::fullWindowHeight=0;

namespace{

Z getX()
{
   GLint x4[4];
   glGetIntegerv(GL_CURRENT_RASTER_POSITION,x4);
   return x4[0];
}

} // namespace

CpmGraphics::Font::Font(Z type)
{
   if (type==1){
      name=GLUT_BITMAP_TIMES_ROMAN_10;
      h=10;
   }
   else if (type==2){
      name=GLUT_BITMAP_HELVETICA_10;
      h=10;
```

```
   }
   else if (type==3){
      name=GLUT_BITMAP_HELVETICA_12;
      h=12;
   }
   else if (type==4){
      name=GLUT_BITMAP_8_BY_13;
      h=13;
   }
   else if (type==5){
      name=GLUT_BITMAP_9_BY_15;
      h=15;
   }
   else if (type==6){ // GLUT_BITMAP_HELVETICA_18 is very poorly designed
      // or implemented and should not be used
      // name=GLUT_BITMAP_HELVETICA_18; Using it nevertheless in the hope that it
      // became repaired meanwhile is an experiment!
      // h=18;
      name=GLUT_BITMAP_HELVETICA_18;
      h=18;
   }
   else {
      name=GLUT_BITMAP_TIMES_ROMAN_24;
      h=24;
   }
   d=cpmrnd(h*0.2);
   mw=cpmrnd(h*0.6); // as in 9 by 15 font
      // only an approximation - needed since glutBitmapWidth(...)
      // couldn't be made working so far
}

CpmGraphics::Font::Font(R fh)
{
   if (fh<=10.0){
      name=GLUT_BITMAP_HELVETICA_10;
      h=10;
   }
   else if (fh<=11.0){
      name=GLUT_BITMAP_TIMES_ROMAN_10;
      h=10;
   }
   else if (fh<=12.5){
      name=GLUT_BITMAP_HELVETICA_12;
      h=12;
   }
   else if (fh<=14.0){
      name=GLUT_BITMAP_8_BY_13;
      h=13;
   }
   else if (fh<=16.5){
```

```
      name=GLUT_BITMAP_9_BY_15;
      h=15;
   }
   else if (fh<=21.0){ // see comment to Font(6)
   //   name=GLUT_BITMAP_HELVETICA_18;
    //  h=18;
      name=GLUT_BITMAP_9_BY_15; // experiment due to L'
      h=15;
   }
   else if (fh<=24.5) {
      name=GLUT_BITMAP_TIMES_ROMAN_24;
      h=24;
   }
   else{
      name=GLUT_BITMAP_TIMES_ROMAN_24;
      h=24;
   }
   d=cpmrnd(h*0.2);
   mw=cpmrnd(h*0.6); // as in 9 by 15 font
      // only an approximation - needed since glutBitmapWidth(...)
      // couldn't be made working so far
}

// basic Cpm graphics library should be independent from CpmSystem and
// thus not to have to include cpmsystem.h

xyxy CpmGraphics::Viewport::seg1;
xyxy CpmGraphics::Viewport::seg2;
xyxy CpmGraphics::Viewport::seg3;
xyxy CpmGraphics::Viewport::seg4;

// background colors of the panes on the status bar
ColRef CpmGraphics::Viewport::cs1(L(255),L(255),L(50));
ColRef CpmGraphics::Viewport::cs2(L(210),L(255),L(110));
ColRef CpmGraphics::Viewport::cs3(L(180),L(180),L(190));
ColRef CpmGraphics::Viewport::cs4(L(165),L(165),L(255));
ColRef CpmGraphics::Viewport::colText=ColRef(L(0),L(0),L(0));
ColRef CpmGraphics::Viewport::colWindow(L(180),L(180),L(180));

void Viewport::fill(xyxy r, ColRef cr)
{
   glColor3f(cr.glR(),cr.glG(),cr.glB());
   glRecti(r.x1,r.y1,r.x2+1,r.y2+1);
   glFlush();
   if (Rec::db) glutSwapBuffers();
}

Viewport::Viewport(Z w, Z h, const Word& title)
{
      if (initialized) return;
```

```
      Word loc("Viewport::Viewport(Y,Y,Word)");
#if defined(CPM_USE_MPI)
      size=CpmMPI::Cpm_com.getSize();
      rank=CpmMPI::Cpm_com.getRank();
#else
      size=1;
      rank=1;
#endif
  //    ColRef crBac(L(180),L(180),L(180));
      applWindowWidth=w;
      fullWindowHeight=h;
      fullWindow=xyxy(xy(0,0),applWindowWidth,fullWindowHeight);
      applWindowHeight=fullWindowHeight-heightStatusBar;
      window=xyxy(xy(0,0),applWindowWidth,applWindowHeight);
      if (rank==1){
         Word mes="OpenGL window got for applWindowWidth = ";
         mes&=cpm(applWindowWidth);
         mes&=" , applWindowHeight= ";
         mes&=cpm(applWindowHeight);
         Word mes2="Platform is "&Platform;
         cout<<mes<<endl;
         cout<<mes2<<endl;
      }
      R cp1=Message::pane1(); // from cpmsystemdependencies
      R cp2=Message::pane2();
      R cp3=Message::pane3();
      R cp4=Message::pane4();
      R sumPanes=cp1+cp2+cp3+cp4;
      if (sumPanes==0){
         cp1=2;
         cp2=2;
         cp3=1;
         cp4=1;
         sumPanes=cp1+cp2+cp3+cp4;
      }
      R awsp=applWindowWidth/sumPanes;
      Z w1=cpmrnd(awsp*cp1);
      Z w2=cpmrnd(awsp*cp2);
      Z w3=cpmrnd(awsp*cp3);
      Z w4=applWindowWidth-w1-w2-w3;
      Z yL=applWindowHeight;
      Z xL=0;
      seg1=xyxy(xy(xL,yL),w1,heightStatusBar);
      xL+=w1;
      seg2=xyxy(xy(xL,yL),w2,heightStatusBar);
      xL+=w2;
      seg3=xyxy(xy(xL,yL),w3,heightStatusBar);
      xL+=w3;
      seg4=xyxy(xy(xL,yL),w4,heightStatusBar);
      initialized=true;
```

```
      R fontHeight=14;
      Font font(fontHeight);
      Z tx=font.getWidth();
      Z ty=font.getHeight();
      ColRef crText(L(0),L(0),L(255));
      textLine(tx,ty,title,crText,fontHeight);
      clrPan();
      fill(window,colWindow);
      glFlush();
      if (Rec::db) glutSwapBuffers();

   }

void Viewport::onStatusBar(const Word& w, Z i)
{
   static Z th=cpmrnd(0.75*heightStatusBar);
   R fH=12.0;
   static Font font(fH); // Helvetica 12
   if (!isInitialized() ) return;
   ColRef cr;
   xyxy seg;
   if (i==1){
      cr=cs1;
      seg=seg1;
   }
   else if (i==2){
      cr=cs2;
      seg=seg2;
   }
   else if (i==3){
      cr=cs3;
      seg=seg3;
   }
    else if (i==4){
      cr=cs4;
      seg=seg4;
   }
   fill(seg,cr);
   Z shift=2;
   placeWord(seg.x1+shift, seg.y1+th, w, font, seg.w()-shift);
 }

void Viewport::clrPan()
{
   fill(seg1,cs1);
   fill(seg2,cs2);
   fill(seg3,cs3);
   fill(seg4,cs4);
}
```

```
void Viewport::placeWord(Z tx, Z ty, const Word& w, const Font& f,
   Z wMax)
{
   glColor3f(colText.glR(),colText.glG(),colText.glB());
   Z x=tx,y=ty;
   Z xLim=tx+wMax;
   Z xM=f.getWidth();
   xLim-=xM;
   Z n=w.dim();
   glRasterPos2i(x,y);
   for (Z i=1;i<=n;++i){
      glutBitmapCharacter(f.getName(),w[i]);
      x=getX();
      if (x>xLim) break;
   }
   glFlush();
   if (Rec::db) glutSwapBuffers();
}

xyxy Viewport::textLine(Z tx, Z ty, const Word& w,
      ColRef crText, R h)
      // writing w on a line previously filled with crBac
{
   glColor3f(colText.glR(),colText.glG(),colText.glB());
   Font f(h);
   Z x=tx,y=ty;
   Z yM=f.getHeight();
   Z decender=(Z)(0.25*yM+0.5);
   Z i,n=w.dim();
   Z more=2; // makes the background rectangle a bit larger than the text
   glColor3f(crText.glR(),crText.glG(),crText.glB());
   glRasterPos2i(x,y);
   for (i=1;i<=n;++i){
       glutBitmapCharacter(f.getName(),w[i]);
   }
   Z txF=getX();
   glFlush();
   if (Rec::db) glutSwapBuffers();
   return xyxy(tx-more,ty-yM+decender-more,txF+more,ty+decender+more);
 }

xyxy Viewport::addText(Z x, Z y, Word const& line,
   R fontHeight, Z fontNumber)
{
   fontNumber; // not used
   return textLine(x,y,line,colText,fontHeight);
}

#else
// e.g. if CPM_NOGRAPHICS is defined
```

```
// trivial implementation; since then graphics
// works through a RAM-based data structure Img24

bool CpmGraphics::Viewport::initialized=true;

Viewport::Viewport(Z w, Z h, const Word& title){}
void Viewport::onStatusBar(const Word& w, Z i){}

Z CpmGraphics::Viewport::applWindowWidth=10;
Z CpmGraphics::Viewport::applWindowHeight=10;
Z CpmGraphics::Viewport::fullWindowHeight=10;

void CpmGraphics::Viewport::placeWord(Z tx, Z ty, const Word& w,
   const Font& f, Z wMax){tx;ty;w;f;wMax;}

void CpmGraphics::Viewport::fill(xyxy r, ColRef cr){r;cr;}

xyxy Viewport::textLine(Z tx, Z ty, const Word& w,
      ColRef crText, R h)
{
   tx;ty;w;crText;h;
   return xyxy();
}

xyxy Viewport::addText(Z x, Z y, Word const& text, R size, Z fontNumber)
{
   x;y;text;size;fontNumber;
   return xyxy();
}

void Viewport::clrPan()
{}

#endif
```

46 cpmvlin.h

```
//? cpmvlin.h
//? Status of work 2023-10-20.
//?
//?  ...

#ifndef CPM_VLIN_H_
#define CPM_VLIN_H_
/*
   Purpose: Defining an array class Vlin<T,S> which exploits the assumption
   that T supports just the arithmetics which occurrs in the
   definition of the leapfrog integrator ALF. For more detail see
   template class Alf<X,Y,Sc> in file cpmalf.h.
*/

#include <cpmv.h>

////////////////////// class Vlin<> ////////////////////////////

namespace CpmArrays{

   using CpmRoot::Z;
   using CpmRoot::R;
   using CpmRoot::operator""_R;
   using CpmRoot::Word;
   using CpmRoot::Root;

   using namespace std;

template <typename T, typename S>
   // In my leapfrog application I had alwys S=R, but S=C or S=Z
   // could also be useful.
   // In this application T satisfies the user defined concepts
   // affin, rLinear, measurable and the standard concept semiregular.
   // The present definition is made such that Vlin<T,S>
   // also satifies these concepts.

class Vlin: public V<T>{ // version of V with arithmetic operations
   // which, for suitable types T and S make Vlin<T,S> a linear space
   // over the field S.

   typedef V<T> Base;
   using Base::p_;
   using Base::cow_;

public:

// In this class, for simplicity, we define only constuctors for vectors
```

```
// for which the first valid index is 1 (index convention)

explicit   Vlin(Z n=0):V<T>(n){}

   Vlin(Z n, T const& t):V<T>(n,t){}

// constructors from explicit lists
explicit  Vlin(std::initializer_list<T> il ):V<T>(il){}
      // constructors from explicit lists such as
      // Va<Z> v{-2,2,4,8}; Notice v[1]== -2, v[4]==8

   Vlin(vector<T> const& v):V<T>(1,v){}

   Vlin(V<T> const& h):V<T>(h)
      {if (h.dim()>0) cpmassert(h.b()==1,"index convention");}
   // 'down-cast'(?)-constructor

   V<T> base()const{ return Base(*this);}

   Word nameOf()const{
      Word wi="Vlin<";
      return wi&CpmRoot::Name<T>()(T())&","&CpmRoot::Name<S>()(S())&">";
   }
// for concept measurable
   R abs2()const;
   R abs()const{ return cpmsqrt(abs2());}

   friend R supAbs(Vlin<T,S> const& vl ){
      R res=0_R;
      for (Z i=vl.b();i<=vl.e();++i){
         R ri=vl[i].abs();
         if (ri>res) res=ri;
      }
      return res;
   }

// for concept affine
   Vlin<T,S>& operator +=(Vlin<T,S> const& v);
   Vlin<T,S>& operator -=(Vlin<T,S> const& v);
// for concept rLinear
   Vlin<T,S> operator + (Vlin<T,S> const& v)const;
   Vlin<T,S> operator - (Vlin<T,S> const& v)const;
   Vlin<T,S>& operator *=(S const& s);
   Vlin<T,S> operator * (S const& s)const;
};

template <class T, class S>
R Vlin<T,S>::abs2(void)const
{
   auto res=1_R;
```

```
   for (auto val : *p_){
      res*=val.abs2();
   }
   return res;
}


template <class T, class S>
Vlin<T,S>& Vlin<T,S>::operator *=(S const& s)
{
   for (auto val : *p_) val*=s;
   return *this;
}

template <class T, class S>
Vlin<T,S> Vlin<T,S>::operator *(S const& s)const
{
   vector<T> res;
   for (auto val : *p_) res.push_back(val*s);
   return Vlin<T,S>(res);
}

template <class T, class S>
Vlin<T,S>& Vlin<T,S>::operator +=(Vlin<T,S> const& v)
{
   cow_();
   for (auto it1 = begin(*p_), it2=begin(*v.p_);
      it1 != end(*p_); ++it1, ++it2) *it1 += *it2;
   return *this;
}

template <class T, class S>
Vlin<T,S>& Vlin<T,S>::operator -=(Vlin<T,S> const& v)
{
   cow_();
   for (auto it1 = begin(*p_), it2=begin(*v.p_);
      it1 != end(*p_); ++it1, ++it2) *it1 -= *it2;
   return *this;
}

template <class T, class S>
Vlin<T,S> Vlin<T,S>::operator + (Vlin<T,S> const& v)const
{
   vector<T> res;
   for (auto it1 = begin(*p_), it2=begin(*v.p_);
      it1 != end(*p_); ++it1, ++it2) res.push_back(*it1 + *it2);
   return Vlin<T,S>(res);
}

template <class T, class S>
```

```
Vlin<T,S> Vlin<T,S>::operator - (Vlin<T,S> const& v)const
{
   vector<T> res;
   for (auto it1 = begin(*p_), it2=begin(*v.p_);
      it1 != end(*p_); ++it1, ++it2) res.push_back(*it1 - *it2);
   return Vlin<T,S>(res);
}

} // namespace
#endif
```

47 cpmvm.h

```
//? cpmvm.h
//? Status of work 2023-10-20.
//?
//?  ...


#ifndef CPM_VSP_H_
#define CPM_VSP_H_
/*

Purpose:
With C++11 'move semantics' was added to C++ and seduced me to base
the C+- array template V<> on std::vector<> which provided the
move facility 'under the hood'and no longer relies on
'copy on write' (there are still C+- classes that do). The present class template is
a near to minimal vector class which works with copy on write (no move semantics)
and does indexing by the same syntax as std::vector<> for staightforward speed
comparisons. In my project tut1 there is done such a comparison with a
program that does several calls to copy constructors and assignments in chain.
Here copy on write works around 700 times faster than 'move semantics'.
This test case may well be far from practice but it looks wise not to
eliminate copy on write completely from C+-.
*/

#include <memory>
//#include <vector>

////////////////////////////// class Vsp<> ////////////////////////////////
// Array without index check, reference counting, and copy on write, and
// 'value semantics'
// Generalized from Koenig's class Handle p. 72-73.

namespace CpmArrays{

//   using namespace CpmStd;

   using CpmRoot::N;

template <typename T>
class DynamicArray
{
private:
T* m_array;
N m_length;

public:
DynamicArray(N length=0)
```

```
: m_array(new T[length]), m_length(length)
{
}


~DynamicArray()
{
delete[] m_array;
}

// Copy constructor
DynamicArray(const DynamicArray &arr) = default;

// Copy assignment
DynamicArray& operator=(const DynamicArray &arr) = default;

// Move constructor
DynamicArray(DynamicArray &&arr) noexcept
:  m_array(arr.m_array), m_length(arr.m_length)
{
arr.m_length = 0;
arr.m_array = nullptr;
}

// Move assignment
DynamicArray& operator=(DynamicArray &&arr) noexcept
{
if (&arr == this)
return *this;

delete[] m_array;

m_length = arr.m_length;
m_array = arr.m_array;
arr.m_length = 0;
arr.m_array = nullptr;

return *this;
}

N getLength() const { return m_length; }
T& operator[](N index) { return m_array[index]; }
const T& operator[](N index) const { return m_array[index]; }

};


}  // namespace CpmArrays

#endif
```

48 cpmvo.h

```
//? cpmvo.h
//? Status of work 2023-10-20.
//?
//?  ...


#ifndef CPM_VO_H_
#define CPM_VO_H_
/*
   Purpose: Defining a array class Vo<T> which exploits the assumption\
    that
   T supports ordering. See cpmv.h

*/
#include <cpmv.h>
#include <utility>
#include <algorithm>
#include <unordered_set>

namespace CpmArrays{

////////////////////////// class Vo ////////////////////////////////////

template <class T>
   // We assume that T provides (explicitly or implicitly)
   // copy constructor, and assignment or that T is a built-in type.
   // T < T and T > T assumed.

class Vo : public V<T>{ // version of V with order-related operations
   // derivation which implements the order interface

   typedef Vo<T> Type;
   typedef V<T> Base;
public:

//Vo()=default;

explicit Vo(Z n=0):V<T>(n){}

explicit Vo(IvZ ivz):V<T>(ivz){}

   Vo(IvZ ivz, T const& t):V<T>(ivz,t){}

// constructors from explicit lists
explicit  Vo(std::initializer_list<T> il ):V<T>(il){}
   // requires C++11
      // constructors from explicit lists such as
```

```
   // Vo<Z> v{1,2,4,8};

Vo(Z n, T const& t):V<T>(n,t){}

Vo(const V<T>& h):V<T>(h){}
      // 'down-cast'-constructor

Vo(const Vo<T>& h):V<T>(h){}
      // 'down-cast'-constructor

virtual V<T>* clone(void)const{ return new Vo(*this);}

virtual Word nameOf()const
   //: name of
{
   Word wi="Vo<";
   return wi&CpmRoot::Name<T>()(T())&">";
}

V<Z> permutationForIncreasingOrder(void)const;
   // Returned is an array per such that i<j enforces
   //  (*this)[per[i]]<(*this)[per[j]]
   //       or
   // (*this)[per[i]]==(*this)[per[j]]
   // Assumes that valid indexing starts with 1 (i.e. b()==1)

V<Z> permutationForDecreasingOrder(void)const
   // analogous to previous function
{ return permutationForIncreasingOrder().rev();}

void order_(void);
   // Permutes the indexes such that
   // (*this)[i]<(*this)[j] || (*this)[i]==(*this)[j]
   // whenever i<j. The components (instances of T) are allowed to
   // be large objects since excessive copying is avoided.

Vo<T>& sort_(void){ order_(); return *this;}
   //: sort

void sortInc_();

void unique_(bool ord=true);
   // Calling this function changes *this (need not to be ordered)
   // into an array in which no two components have the same vale.
   // Thus 'elimination of doublets'. Nothing about potential
   // reordering is known. If the argument is 'true' the array, after
   // the doublets have been eliminated gets ascendingly ordered.

V<Z> perForDecOrd(void)const
   //: permutation for decreasing order
```

```
{ return permutationForDecreasingOrder();}

V<Z> perForIncOrd(void)const
   //: permutation for increasing order
{ return permutationForIncreasingOrder();}

Vo<T> increasingCopy(void)const;
   // returns a increasingly ordered version of (*this) without
   // changing *this

Vo<T> incCopy(void)const{ return increasingCopy();}
   //: increasing copy

Vo<T> strictlyIncreasingVersion(void)const;
   // returns a stricly increasingly ordered version of (*this)
   // which leaves out multiply appearing components
   // (again, without changing *this)

Vo<T> strIncVer(void)const{ return strictlyIncreasingVersion();}
   //: strictly increasing version

Vo<T> decreasingCopy(void)const;
   // returns a decreasingly ordered version of (*this) without
   // changing *this

Vo<T> decCopy(void)const{ return decreasingCopy();}
   //: decreasing copy

Vo<T> purge(const V<T>& s)const;
   // returned is a list which is obtained from *this by eliminating
   // all components which equal to one of the components of s. Beside
   // of this removal, the order of the components in *this is
   // retained.

Vo<T> pur(const V<T>& s)const{ return purge(s);}
   //: purge

Z locate(T const& t)const{ return loc3(t).c1();}
   // locate
   // returned is a value j such that t satisfies
   //    (*this)[j] <= x < (*this)[j+1].
   // (*this) must be monotonic increasing
   // j=b()-1 or j=e()+1 is returned to indicate that t is out of
   // range.

 X3<Z,Z,bool> loc3(T const& t)const;
   //: locate with a 'three-fold return value' res of type X3<Z,Z,bool>
   // which describes how the object t is placed relativ to the
   // components of the vector *this. Let us explain:
   // We write
```

```
    //    i for res.c1() ( an integer)
    //    j for res.c2() ( an integer)
    //    found for res.c3() ( an boolean).
    // The meaning of j depends on the quantities j and found.
    // Let us begin with discussing the variable 'found':
    // found==true implies j==0 and (*this)[i]==t.
    // found==false implies j in {-1, 0, 1}.
    //    j==0 implies (*this)[i] < t < (*this)[j+1]
    //    j==-1 implies i == iL-1 where iL is the first valid index
    //       of the vector *this
    //    j==1 implies i == iU+1 where iU is the last valid index
    //       of the vector *this.

// Vo<T> enq(T const&)const;
    //: enqueue
    // assumes that (*this) is monotonic. Then t will be inserted in
    // the orderly correct position into of a copy of (*this). This copy
    // remains monotonic and gets returned.
    // The second argument comes only in action for dim()==1, since
    // then no direction is provided by *this and a direction is needed
    // for the result (for dim()==0, the result has dim()==1 and no
    // direction).

 Z  enq_(T const&);
    // assumes that (*this) is monotonic. Then t will be inserted in
    // the orderly correct position into of a copy of (*this). This copy
    // remains monotonic and gets returned.
    // The second argument comes only in action for dim()==1, since
    // then no direction is provided by *this and a direction is needed
    // for the result (for dim()==0, the result has dim()==1 and no
    // direction).

 Z enqX_(T const& t){ enq_(t); return find(t);}
    // enplaces t into *this and returns the index i such that
    // (*this)[i]=t

 T sup()const;
    //: supremum
    // returns the largest component of *this; faster than getting the
    // largest element by ordering

 T inf()const;
    //: infimum
    // returns the smallest component of *this; faster than getting the
    // largest element by ordering

 Z indSup()const;
    //: index (of) supremum
    // returns the smallest i for which (*this)[i]=(*this).sup()
```

```
   Z indInf()const;
      //: index (of) infimum
      // returns the smallest i for which (*this)[i]=(*this).inf()

   T2<Z> indInfSup()const;
      //: returns indInf(), indSup() as a pair in a single pass

   T2<T> infSup()const;
      //: infimum supremum
      // Let res be the return value, then res.first==inf(),
      // res.second==sup(). Fast single-pass operation
};


// Function indexx from Press et al. changed into a function template
// p. 339  (role of M: see p. 333; algorithm is based on Quicksort, see
// sort(), anyway the choice of M may influence performance but never
// puts a limitation on the size of the vector to be ordered.

namespace{
   inline void swapZ(Z& i, Z& j){ Z ii=i; i=j; j=ii;}
}

template <class T>
V<Z> Vo<T>::permutationForIncreasingOrder(void)const
// Indexes an array v[1..n], i.e. outputs the array indx[1..n] such that
// v[indx[j]] is in ascending order for j=1,2,...,n. The input
// quantities n and v are not changed. Especially, v-elements are
// not copied back and forth which is essential if T is a large class
{
   const Z mL=4;
   cpmmessage(mL,"Vo<T>::permutation...(): started");
   const Z M=7;
   const Z NSTACK=50;

   Z nVec=Base::dim();
   Vo<Z> indx(nVec);
   Z i,indxt,ir=nVec,j,k,l=1,jstack=0;
   T a;

   Vo<Z> istack(NSTACK);

   for (j=1;j<=nVec;j++) indx[j]=j;
   for (;;) {
      if (ir-l < M) {
         for (j=l+1;j<=ir;j++) {
            indxt=indx[j];
            a=(*this)[indxt];
            for (i=j-1;i>=1;i--) {
               if ((*this)[indx[i]] < a) break;
```

```
            if ((*this)[indx[i]] == a) break;
            indx[i+1]=indx[i];
         }
         indx[i+1]=indxt;
      }
      if (jstack == 0) break;
      ir=istack[jstack--];
      l=istack[jstack--];
   } else {
      k=(l+ir)/2; // was >> 1 instead of /2 before
      swapZ(indx[k],indx[l+1]);
      if ((*this)[indx[l+1]] > (*this)[indx[ir]]) {
         swapZ(indx[l+1],indx[ir]);
      }
      if ((*this)[indx[l]] > (*this)[indx[ir]]) {
         swapZ(indx[l],indx[ir]);
      }
      if ((*this)[indx[l+1]] > (*this)[indx[l]]) {
         swapZ(indx[l+1],indx[l]);
      }
      i=l+1;
      j=ir;
      indxt=indx[l];
      a=(*this)[indxt];
      for (;;) {  // the i<nVec and j>1 test added by UM, otherwise in
            // patological cases of < and > relations we get a range
            // error
         do i++; while (i<nVec && (*this)[indx[i]] < a);
         do j--; while (j>1 && (*this)[indx[j]] > a);
         if (j < i) break;
         swapZ(indx[i],indx[j]);
      }
      indx[l]=indx[j];
      indx[j]=indxt;
      jstack += 2;
      if (jstack > NSTACK) cpmerror("NSTACK too small in ordering()");
      if (ir-i+1 >= j-l) {
         istack[jstack]=ir;
         istack[jstack-1]=i;
         ir=j-1;
      } else {
         istack[jstack]=j-1;
         istack[jstack-1]=l;
         l=i;
      }
   }
}
cpmmessage(mL,"Vo<T>::permutation...(): done");
return indx;
}
```

```
template <class T>
void Vo<T>::unique_(bool ord){
   std::unordered_set<T> s;
   for (Z i: Base::p_) s.insert(i);
   Base::p_.assign(s.begin(),s.end());
   if (ord) std::sort( Base::p_.begin(), Base::p_.end() );
   Base::sz_=Base::p_.size();
   Base::iv_=IvZ(Base::sz_,Base::b());
}

/*******************************************
template <class T>
Z Vo<T>::locate(T const& x)const
{
   Word loc("Z Vo<T>::locate(T&)const");

   Z sz=Base::sz_;
   cpmassert(sz>0,loc);

   Z jl=Base::b()-1;
   Z ju=Base::e()+1;

   T xl=Base::fir();
   T xu=Base::last();
   cpmassert(xl<xu,loc); // ascending order needed. Of course this is not
      // sufficient condition for this.
   Z res{jl-1};
   if (x<xl){
      res=jl;
      return res;
   }
   if (x>xu){
      res=ju;
      return res;
   }
   auto q=std::upper_bound(Base::p_.begin(),Base::p_.end(),x);
   if (q==Base::p_.end()){
      res = ju;
      return res;
   }
   else{
      Z locDis=q-Base::p_.begin();
      res=jl+locDis;
      cpmassert(Base::valInd(res),loc);
      cpmassert(Base::valInd(res+1),loc);
      cpmassert((*this)[res]<=x && x<(*this)[res+1],loc);
      return res;
   }
}
```

```
*****************************/

template <class T>
X3<Z,Z,bool> Vo<T>::loc3(T const& x)const
{
   Word loc("X3<Z,Z,bool> Vo<T>::loc3(T&)const");

   Z sz=Base::sz_;
   cpmassert(sz>0,loc);

   Z jl=Base::b()-1;
   Z ju=Base::e()+1;

   T xl=Base::fir();
   T xu=Base::last();
   cpmassert(xl<xu,loc); // ascending order needed. Of course this is not
      // sufficient condition for this.
   X3<Z,Z,bool> res(jl-1,-2,false); // never a valid return value
   if (x<xl){
      res=X3<Z,Z,bool>(jl,-1,false);
      return res;
   }
   if (x>xu){
      res=X3<Z,Z,bool>(ju,1,false);
      return res;
   }
   auto q=std::upper_bound((*Base::p_).begin(),(*Base::p_).end(),x);
   if (q==(*Base::p_).end()){
      res = X3<Z,Z,bool>(ju,1,false);
      return res;
   }
   else{
      Z locDis=q-(*Base::p_).begin();
      Z j0=jl+locDis;
      Z j1=j0+1;
      cpmassert(Base::valInd(j0),loc);
      cpmassert(Base::valInd(j1),loc);
      cpmassert((*this)[j0]<=x && x<(*this)[j1],loc);
      bool found = (*this)[j0]==x;
      return X3<Z,Z,bool>(j0,0,found);
   }
}

// functions based on order relations in T

template <class T>
Vo<T> Vo<T>::increasingCopy(void)const
{
   Vo<Z> per=permutationForIncreasingOrder();
   Z i, n=Base::dim();
```

```
   Vo<T> y(n);
   for (i=1;i<=n;i++) y[i]=(*this)[per[i]];
   return y;
}

namespace{
   template <class T>
   bool fEqual(T const& t1, T const& t2){ return t1==t2;}
}

template <class T>
Vo<T> Vo<T>::strictlyIncreasingVersion(void)const
{
   Vo<T> res=increasingCopy();
   return res.condense(fEqual).c1();
}

template <class T>
Vo<T> Vo<T>::decreasingCopy(void)const
{
   Vo<Z> per=permutationForIncreasingOrder();
   Z i, n=Base::dim();
   Vo<T> y(n);
   for (i=1;i<=n;i++) y[1+n-i]=(*this)[per[i]];
   return y;
}

template <class T>
void Vo<T>::sortInc_()
{
   std::sort(Base::p_->begin(), Base::p_->end());
}

template <class T>
void Vo<T>::order_(void){ sortInc_();}

template <class T >
Vo<T> Vo<T>::purge(const V<T>& s)const
{
   Z i,j,n=Base::dim(),ns=s.dim();
   V<B> vs(n,B(true));
   for (i=1;i<=n;i++){
      T t=(*this)[i];
      for (j=1;j<=ns;j++){
         if( t==s[j] ){ vs[i]=false; break; }
      }
   }
   return Base::select(vs);
}
```

```
template <class T >
T Vo<T>::sup()const
{
   Z n=Base::dim();
   cpmassert(n>0,"T Vo<T>::sup()const");
   T t,res=Base::fir();
   for (Z i=Base::b()+1;i<=Base::e();i++){
      t=Base::cui(i);
      if (t>res) res=t;
   }
   return res;
}

template <class T >
Z Vo<T>::indSup()const
{
   Z n=Base::dim();
   cpmassert(n>0,"Vo<T>::indSup()const");
   T t,valMax=(*this)[1];
   Z res=1;
   for (Z i=2;i<=n;i++){
      t=(*this)[i];
      if (t>valMax){ valMax=t; res=i;}
   }
   return res;
}

template <class T >
T Vo<T>::inf()const
{
   Z n=Base::dim();
   cpmassert(n>0,"T Vo<T>::inf()const");
   T t,res=Base::fir();
   for (Z i=Base::b()+1;i<=Base::e();i++){
      t=Base::cui(i);
      if (t<res) res=t;
   }
   return res;
}

template <class T >
Z Vo<T>::indInf()const
// needed as fast as possible
{
   Z n=Base::dim();
   cpmassert(n>0,"Z Vo<T>::indInf()const");
   Z res=0;
   T t,valMin=Base::cui(res);

   for (Z i=1;i<n;i++){
```

```
      t=Base::cui(i);
      if (t<valMin){ valMin=t; res=i;}
   }
   return ++res;
}

template <class T >
T2<Z> Vo<T>::indInfSup()const
{
   Z n=Base::dim();
   if (n<=0) cpmerror("Vo<T>::indInfSup(): dim()<=0");
   T t,valMin=(*this)[1],valMax=valMin;
   Z resMin=1;
   Z resMax=1;
   for (Z i=2;i<=n;i++){
      t=(*this)[i];
      if (t<valMin){ valMin=t; resMin=i;}
      if (t>valMax){ valMax=t; resMax=i;}
   }
   return T2<Z>(resMin,resMax);
}

template <class T >
T2<T> Vo<T>::infSup()const
{
   Z n=Base::dim();
   if (n<=0) cpmerror("Vo<T>::infSup(): dim()<=0");
   T t,valMin=Base::fir(),valMax=valMin;
   for (Z i=Base::b()+1;i<=Base::e();i++){
      t=Base::cui(i);
      if (t<valMin){ valMin=t; }
      if (t>valMax){ valMax=t; }
   }
   return T2<T>(valMin,valMax);
}

/**********
template <class T >
Vo<T> Vo<T>::enq(T const& t)const
{
   Z mL=3;
   Word loc("Vo<T> Vo<T>::enqueue(T const& t, bool asc)const");
   CPM_MA
   Z d=Base::dim();
   if (d==0) return Vo<T>(1,t);
   if (d==1){
      Vo<T> res(2);
      T t1=(*this)[1];
      if (asc){
         if (t1<t){
```

```
            res[1]=t1;
            res[2]=t;
         }
         else{
            res[1]=t;
            res[2]=t1;
         }
      }
      else{
         if (t1<t){
            res[1]=t;
            res[2]=t1;
         }
         else{
            res[1]=t1;
            res[2]=t;
         }
      }
      return res;
   }
   Z i=locate(t);
   CPM_MZ
   return Base::ins(i+1,t);
}
***************/

template <class T >
Z Vo<T>::enq_(T const& t)
{
   Z mL=3;
   Word loc("Vo<T> Vo<T>::enqueue(T const& t, bool asc)const");
   CPM_MA
   Z d=Base::size();
   Z dn=0;
   if (d==0){
      Base::p_.push_back(t);
      Base::iv_.n_();
      Base::sz_++;
      return 1;
   }
   if (d==1){
      T c1=Base::fir();
      if (t==c1) return 0;
      else if (t>c1){
         Base::p_.push_back(t);
         Base::iv_.n_();
         Base::sz_++;
         return 1;
      }
      else{
```

```
         Base::p_.push_back(t);
         std::rotate(Base::p_.rbegin(), Base::p_.rbegin() + 1,
         Base::p_.rend());
         Base::iv_.n_();
         Base::sz_++;
         return 1;
      }
   }
   // d >= 2
   auto y=loc3(t); // the more informative version of locate
   Z i=y.c1();
   Z j=y.c2();
   bool found=y.c3();
   if (j==1){
      Base::p_.push_back(t);
      Base::sz_++;
      Base::iv_.n_();
      dn=1;
   }
   else if(j==-1){
      Base::p_.push_back(t);
      std::rotate(Base::p_.rbegin(), Base::p_.rbegin() + 1,
         Base::p_.rend());
         Base::sz_++;
         Base::iv_.n_();
         dn=1;
   }
   else{
      if (found){
         ; // since t is already in the array, we have not to insert it.
      }
      else{
         Base::p_.insert(Base::p_.begin()+(i-Base::b()+1),t);
         Base::sz_++;
         Base::iv_.n_();
         dn = 1;
      }
   }
   CPM_MZ
   return dn;
}

////////////////////////// iterated templates ////////////////////////////
// describing multi-indexed quantities
////////////////////////////// class VVo ////////////////////////////////
// matrices

template <class T>
class VVo: public VV<T>{// version of VV with order-related operations
```

```
   typedef VV<T> Base;
   typedef VVo<T> Type;

public:
   CPM_ORDER
// constructors
   VVo():Base(){};
   VVo(Z d1, Z d2):Base(d1,d2){}
   VVo( Z d1, Z d2, T const& t):Base(d1,d2,t){}
      // all components are equal to t
 //  VVo(void):Base(){}
   VVo(VV<T> const& x):Base(x){}
   VVo( V< V<T> > const& x):Base(x){}
};

template <class T >
Z VVo<T>::com(VVo<T> const& s)const
{
   Z d1=Base::dim(), d2=s.dim();
   if (d1<d2) return 1;
   else if (d1>d2) return -1;
   else{
      for (Z i=1;i<=d1;i++){
         Vo<T> ti=(*this)[i];
         Vo<T> si=s[i];
         Z ci=CpmRoot::comT<T>(ti,si);
         if (ci!=0) return ci;
      }
      return 0;
   }
}

////////////////////////// class VVVo ///////////////////////////////////
// tensors of rank 3

template <class T>
class VVVo: public VVV<T>{// version of VVV with order-related operations

   typedef VVV<T> Base;
   typedef VVVo<T> Type;

public:
   CPM_ORDER
// constructors
   VVVo(Z d1,Z d2,Z d3):Base(d1,d2,d3){}
   VVVo(Z d1,Z d2,Z d3,T const& t):Base(d1,d2,d3,t){}
      // all components are equal to t
   VVVo(void):Base(){}
   VVVo(VVV<T> const& x):Base(x){}
   VVVo( V< V< V<T> > > const& x):Base(x){}
```

```
};

template <class T >
Z VVVo<T>::com(VVVo<T> const& s)const
{
   Z d1=Base::dim(), d2=s.dim();
   if (d1<d2) return 1;
   else if (d1>d2) return -1;
   else{
      for (Z i=1;i<=d1;i++){
         VVo<T> ti=(*this)[i];
         VVo<T> si=s[i];
         Z ci=CpmRoot::comT<T>(ti,si);
         if (ci!=0) return ci;
      }
      return 0;
   }
}

////////////////////////// class VVVVo ////////////////////////////////////
// tensors of rank 4

template <class T>
class VVVVo:public VVVV<T>{//version of VVVV with order-related operations

   typedef VVVV<T> Base;
   typedef VVVVo<T> Type;

public:
   CPM_ORDER
// constructors
   VVVVo(Z d1,Z d2,Z d3,Z d4):Base(d1,d2,d3,d4){}
   VVVVo(Z d1,Z d2,Z d3,Z d4,T const& t):Base(d1,d2,d3,d4,t){}
      // all components are equal to t
   VVVVo(void):Base(){}
   VVVVo(VVVV<T> const& x):Base(x){}
   VVVVo( V< V< V< V<T> > > > const& x):Base(x){}
};

template <class T >
Z VVVVo<T>::com(VVVVo<T> const& s)const
{
   Z d1=Base::dim(), d2=s.dim();
   if (d1<d2) return 1;
   else if (d1>d2) return -1;
   else{
      for (Z i=1;i<=d1;i++){
         VVVo<T> ti=(*this)[i];
         VVVo<T> si=s[i];
         Z ci=CpmRoot::comT<T>(ti,si);
```

```
         if (ci!=0) return ci;
      }
      return 0;
   }
}

} // namespace

#endif
```

49 cpmvr.h

```
//? cpmvr.h
//? Status of work 2023-10-20.
//?
//?  ...


#ifndef CPM_VR_H_
#define CPM_VR_H_
/*

Purpose: Defining a vector template providing the full
functionality of the rich interface

*/
#include <cpmva.h>

////////////////////////////// class Vr<> ///////////////////////////////////

namespace CpmArrays{

   using CpmRoot::R;
   using CpmRoot::Root;

template <class T>
   // We assume that T provides (explicitely or implicitely)
   // copy constructor, and assignement
   // or that T is a built-in type.

class Vr: public Va<T>{ // version of Va with a rich interface

public:
   typedef Vr<T> Type;
   typedef Va<T> Base;
   typedef V<T> Source;
   typedef T ScalarType; // was missing till 2001-10-12
      // detected by the gnu compiler

 //  Vr():Va<T>(){}
 //  Vr()=default;

   explicit Vr(Z n=0):Va<T>(n){}

   explicit Vr(IvZ ivz):Va<T>(ivz){}
      // note that Vr(0) is defined and is different from Vr()

   Vr(IvZ ivz, T const& t):Va<T>(ivz,t){}
```

```
   // constructors from explicit lists
   explicit  Vr(std::initializer_list<T> il ):Va<T>(il){}
   // requires C++11
      // constructors from explicit lists such as
      // Vr<Z> v{1,2,4,8};

   Vr(Z n, T const& t):Va<T>(n,t){}
      // initializes all n components with t

   Vr( V<T> const& h):Va<T>(h){}
      // 'down-cast'-constructor

   Vr( Vo<T> const& h):Va<T>(h){}
      // 'down-cast'-constructor

   Vr(Va<T> const& h):Va<T>(h){}
      // 'down-cast'-constructor
   Vr(Vr<T> const& h):Va<T>(h){}
      // 'down-cast'-constructor
/*
#if defined(CPM_USE_MOVE)
   Vr(Vr<T> && h)
   {
      Source::iv_=h.iv_;
      Source::sz_=h.sz_;
      Source::p_=h.p_;
      cout<<"move constructor Vr"<<endl;
      h.iv_=IvZ();
      h.sz_=0;
      h.p_=nullptr;
   }
#endif
*/

   virtual V<T>* clone(void)const{ return new Vr<T>(*this);}

   CPM_TEST
   CPM_DESCRIPTORS
      // dis from Va
};

// member functions

template <class T>
Word Vr<T>::nameOf(void)const
{
   Word w1="Vr<";
   Word w2=Root<T>(T()).nameOf();
   return w1&w2&">";
}
```

```
template <class T>
Word Vr<T>::toWord(void)const
{
   if (Base::isVoid()) return "void list";
   Word res="( ";
   Z i;
   for (i=Base::b();i<Base::e();i++){
      Word wi="("&Root<Z>(i).toWord()&","&Root<T>((*this)[i]).toWord()&"),\
       ";
      res&=wi;
   }
   i=Base::e();
   Word we="("&Root<Z>(i).toWord()&","&Root<T>((*this)[i]).toWord()&")";
   res&=we;
   res&=" )";
   return res;
}

template <class T>
Z Vr<T>::hash(void)const
{
   Z res=0;
   for (Z i=Base::b(); i<=Base::e(); i++){
      res+=Root<T>((*this)[i]).hash();
   }
   return res;
}

template <class T>
Vr<T> Vr<T>::test(Z cpl)const
{
   Z it=0;
   it=Root<Z>(it).test(cpl);
   T xt;
   xt=Root<T>(xt).test(cpl);
   return Vr<T>(it,xt);
}

template <class T>
Vr<T> Vr<T>::ran(Z j)const
   // all randomized versions of *this thus have the same dimension as
   // *this
{
   Vr<T> res(Base::dom());
   T tempX;
   if (j==0){
      for (Z i=Base::b(); i<=Base::e(); i++){
         tempX=(*this)[i];
         res[i]=Root<T>(tempX).ran(j);
```

```
      }
   }
   else{
      Z j0=1000; // this is done to avoid an obvious correlation
         // among the coordinates of x.ran(i)
         // and x.ran(i+1)
      Z jIncr=Root<Z>(j0).ran(j);
      Z jc=j;
      for (Z i=Base::b(); i<=Base::e(); i++){
         tempX=(*this)[i];
         jc+=jIncr;
         res[i]=Root<T>(tempX).ran(jc);
      }
   }
   return res;
}

template <typename T, typename S>
class Vrs: public Vr<T>{  // r: rich, s: scalar
      public:
         Vrs(){}
         Vrs(Vr<T> const& vr):Vr<T>(vr){}
         Vrs(Z n):Vr<T>(n){}
         Vrs(Z n, T const& t):Vr<T>(n,t){}
         Vrs& operator*=(S const& s);
         Vrs operator*(S const& s)const override;
     //    Vrs& operator*(S const& s);
   };

template <typename T, typename S>
Vrs<T,S> Vrs<T,S>::operator*(S const& s )const
{
   Vrs<T,S> res(*this);
   Z n=res.dim();
   for (Z i=1;i<=n;++i){
      res.cui(i)*=s;
   }
   return Vrs(res);
}

//template <typename T, typename S>
//Vrs<T,S>& Vrs<T,S>::operator*(S const& s )
//{
//   Vrs<T,S> res(*this);
//   Z n=res.dim();
//   for (Z i=1;i<=n;++i) res.cui(i)*=s;
//   return Vrs(res);
//}

template <typename T, typename S>
```

```
Vrs<T,S>& Vrs<T,S>::operator*=(S const& s )
{
   Z n=Vr<T>::dim();
   for (Z i=1;i<=n;++i) Vr<T>::cui(i)*=s;
   return *this;
}

// We do not implement the iterated templates since they are now
// endowed with sufficient functionality already as Va, VVa, VVVa,
// and VVVVa

} // namespace

#endif
```

50 cpmvsp.h

```
//? cpmvsp.h
//? Status of work 2023-10-20.
//?
//?  ...


#ifndef CPM_VSP_H_
#define CPM_VSP_H_
/*

Purpose:
With C++11 'move semantics' was added to C++ and seduced me to base
the C+- array template V<> on std::vector<> which provided the
move facility 'under the hood'and no longer relies on
'copy on write' (there are still C+- classes that do). The present class template is
a near to minimal vector class which works with copy on write (no move semantics)
and does indexing by the same syntax as std::vector<> for staightforward speed
comparisons. In my project tut1 there is done such a comparison with a
program that does several calls to copy constructors and assignments in chain.
Here copy on write works around 700 times faster than 'move semantics'.
This test case may well be far from practice but it looks wise not to
eliminate copy on write completely from C+-.
*/

#include <memory>
#include <vector>

////////////////////////////// class Vsp<> ////////////////////////////////
// Array according to modern C++: rule of zero

namespace CpmArrays{

   using namespace CpmStd;

   using CpmRoot::Word;
   using CpmRoot::N;

template <class T>
   // We assume that T provides (explicitly or implicitly)
   // copy constructor, and assignment or that T is a built-in type.

class Vsp{ // vector template, indexing starts with 0 .

public:

 //  Vsp()=default;
```

```
   explicit Vsp(N n);
      // Has n components. Gives an error for n<0 and n>dimMax.
      // The components are initialized by the default constructor
      // of T if T is a class for which such a constructor is defined
      // (explicitly or implicitly).
      // If T is a built-in type, initialization is done as 0.
      // See BS3, p. 131 for initialization of built-in types via
      // formal constructor calls.
      // If n>0, the first valid index is 1, which reflects the
      // normal behavior of class V.

    Vsp(std::vector<T> const& v);
      // Construction from a standard library vector. This is useful
      // for interaction with the standard containers (which don't
      // implement reference counting).

   Vsp(N n, T const& t) ;
      // Has n components all initialized as t.

 //  Vsp(Vsp<T> const& h)=default;
      // copy constructor

// constructors from explicit lists
   explicit Vsp(std::initializer_list<T> il );
      // requires C++11
      // constructors from explicit lists such as
      // Vsp<Z> v{1,2,4,8};

  // ~Vsp()= default;
      // destructor, we intend not to derive from this, so not virtual

 //  Vsp<T>& operator=(Vsp<T> const& h)=default;
      // assignment

   N dim()const { return sz_;}
      //: dimension
      // Returns the number of components of the vector *this

   N size()const { return sz_;}
      //: size
      // for uniformity with STL

   std::vector<T> std()const;
      //: standard
      // Returns a std::vector which holds all components of *this.

   Word nameOf()const;
      //: name of
      // returns a name of the type
```

```
// constant access functions

   const T& operator[](N i)const{ return p_.get()[i];}


   T& operator[](N i){ return p_.get()[i];}

private:

   N sz_;
      // number of valid indexes (depends on iv_, equals iv_.car())

   std::shared_ptr<T> p_;
      // pointer to beginning of the array.
};

////////////////////// Implementation ////////////////////////////////


template <class T>
Word Vsp<T>::nameOf()const{
   Word nt=Root<T>(T()).nameOf();
   Word wi="Vsp<";
   return wi&nt&">";
}


template <typename T>
Vsp<T>::Vsp(N n):sz_(n)
{
    p_=std::shared_ptr<T>(new T[n]);
}

template <class T>
Vsp<T>::Vsp(N n, T const& t):sz_(n)
{
   p_= n==0 ? nullptr : new T[n];
   for (N i=0;i<sz_;++i) p_[i] = t;
}

template <class T>
Vsp<T>::Vsp(std::initializer_list<T> il ):
sz_(il.size()) // 137 is dummy argument
{
   p_= sz_==0 ? nullptr : new T[sz_];
   std::uninitialized_copy(il.begin(),il.end(),p_);
      // see BS4, p. 498
}

template <class T>
```

```
Vsp<T>::Vsp(std::vector<T> const& v):sz_(v.size())
{
   p_= sz_==0 ? nullptr : new T[sz_];
   typename std::vector<T>::const_iterator i;
   T* q=p_;
   for (i=v.begin();i!=v.end();++i) *q++=*i;
}

template <class T>
std::vector<T> Vsp<T>::std()const
{
   std::vector<T> res(sz_);
   typename std::vector<T>::iterator i;
   T* q=p_;
   for (i=res.begin();i!=res.end();++i) *i=*q++;
   return res;
}

}  // namespace CpmArrays

#endif
```

51 cpmvuc.h

```
//? cpmvuc.h
//? Status of work 2023-10-20.
//?
//?  ...

#ifndef CPM_VucUC_H_
#define CPM_VucUC_H_
/*

Purpose: Building on std::vector

   Basic array class with valid indexing ranging in a contiguous
   subset of Z. Two cases are of particular interest: That the lowest
   valid index is 0 (as for std::vector<>) or 1 (as e.g. in the functions
   in Press et al.). When dealing with FFT index ranges
   {-n,...,-1,0,1,...,n} are adequate (not yet implemented).
   Most constructors of Vuc create instances of Vuc<T> with valid indexes
   starting at 1. The two functions b_(Z) and e_(Z) provide means
   to arbitrarily shift the range of valid indexes. Access to components
   by means of indexing is range checked.

History: Till October 2010 there was a class template Vucl<T> with
indexing starting at 0 and - based on that (not derived from that)
a class template Vuc<T> with indexing starting at 1. This caused an
uncomfortable situation: Whenever dealing with a topic where I felt
that efficiency would be of utmost importance (e.g. font representation
and quantum dynamics) I was tempted to use Vucl<> not only as an
internal device but also in the interface of public member functions.
This made the implementation code and even the classes dependent
on a decision that with a slight twist of emphasis could also have
been made differently. The new state of affairs is that Vucl has been
eliminated (actually replaced by Vuc) in all C+- code. If it should still
shine up in some comment, ignore it.

Recent history:
    2012-01-11 function cow() renamed to cow_()
    2012-01-18 function X2<Z,bool> findAsc(T const& t)const added
    2012-01-19 function prnOn modified so that indexes are printed
      (in commentarized form) together with the components.
    2017-02-18 Short report on a project which looked promising
      but was finally abandoned:
      The rather mature state that C++ has reached with C++11 triggered the
      desire to make more systematic use of the standard library
      facilities. Especially that now all standard containers are said to
      implement moving as a replacement of copying where appropriate
      promised to make it feasible to replace T* p_ by std::vector<T> p_
      and thus free the implementation of Vuc<T> from defining copy, move,
```

```
      assignment operators. The first disappointing problem was that then
      Vuc<bool>, since based on std::vector<bool>, did no longer work in my
      code which used operator[](Z) for Vuc<bool> as in all other Vuc<T>'s.
      Of course replacing Vuc<bool> by Vuc<B> would help. But re-writing all
      the many functions of Vuc<T> in the new style looks disappointingly
      tedious to me. Before making a second attempt I need to understand
      whether the new move functionality is in fact a full replacement for
      the reference counting and copy on write functionality which is
      implemented in Vuc<T> as of today. There is a project
      'codingexperiments' in ~/e/cpm/codingexperiments which illustrates
      the malfunction of vector<bool>, which also is well known to the
      WWW. Further there is ~/e/cpm0ExperimentBasedOnvector which
      contains the experimental version of cpmv.h. In this version by far
      not all uses of T* are replaced by vector<T>.

   2022-10-15 When revising the first tutorial on C+- felt the need
      base my abandoning of 'reference counting' and 'copy on write' in
      favour of the new panacea of 'move semantics' on hard facts.
      It turned out that my original test program gave for both
      std::vector and CpmArrays::Vuc nearly the same speed. Where was the
      more than hunderedfold speed advantage of CpmArray::Vuc that I saw with
      previous versions of Vuc? Since Vuc now relied on move semantics (imported
      by storing data not as T* but as std::vector<T> ) the similarity of
      speeds was no surprise, the surprise was that the speed was as low
      as was observed with a completely elementary beginners implementation.
      Only after having reintroduced reference counting the old advantage
      returned. I had to trivialize my test program to see std::vector
      switching to move semantics and getting to the speed Vuc. Probably
      one could, by inserting move commands by hand, enforce acceleration.
      For C+-, however, only fully automatic are considered acceptable.
      As a consequence, reference counting and copy on write are back again.
      Although the new code does not mention 'move' and '&&' the obtained classes
      are diagnosed as 'movable', 'semiregular', and even 'regular'.
      For technical reasons, the new version of Vuc stores data not (as its predecessor did)
      as std::vector<T> but as std::vector<T>*.

Credit: Modified and extended from Andrew Koenig: Ruminations on C++,
AT&T 1997, Ch. 7. Authoritative and enlightening treatment. In the present version
of the class Handle (p. 62) some additions are made which concern data access
in the style of an array. The function Vuc<T>::cow(), although inspired from
Koenig's book seems to be an innovation.

This defines a template Vuc<T> of T-valued lists or 'vectors with
T-valued components'. It is similar to std::vector<T>
but it differs from this in some respect. In describing
Vuc<T> and these differences, I'll introduce some notions and notations
that will be used over and over in defining and stating properties of
other CPM classes
( CPM = 'C+-' or 'Classes for Physics and Mathematics').
```

```
1. Notice that Vuc<> is not a 'polymorphic container'
   i.e. the components can hold only T typed objects and not the
   additional information content which instances of classes derived
   from T may carry. We may, however, form Vuc<T*> for any type or class
   to support polymorphism in the old-fashioned pointer-based way.
   A safer way is to use the polymorphic vector template Vucp mentioned
   in item 5.

2. Vuc<> implements 'reference counting' and 'copy on write' a wellknown
   mechanism for avoiding making copies of temporary (potentially large)
   objects. The C+- tutorial project tut1 has examples of where the
   new strategy of 'move semantics' fails to bring about the speed
   advantage which 'copy on write' provides reliably.

3. Requirements on T in order to allow the formation of Vuc<T>:
   These are the same requirements valid for std::vector<T>.
   'for all objects t1 and t2 of type T, the expression t1<t2 is defined'
   (compare David R. Musser, Atul Saini: STL Tutorial and Reference
   Guide, Addison-Wesley 1996, p. 246). Now the pertinent statement:
   Vuc<T>  is well defined if (not iff!) T is a built-in type or a class
   which 'implements the value interface'. Then, Vuc<T> also implements the
   value interface. If, moreover, T 'implements the strict value
   interface', then Vuc<T> also implements the strict value interface.
   Here, a class T is said to implement the value interface if it
   publicly defines T(), T(const T&), and T& operator=(const T&).
   T is said to implement the strict value interface if, in addition, it
   has value semantics (as opposed to pointer semantics, see
   Andrew Koenig: Ruminations on C++, AT&T 1997, p. 62. and
   Bjarne Stroustrup: The C++ Programming Language, 3. edition,
   Addison-Wesley 1997, p. 294). Classes that implement the strict value
   are most convenient to use. Code only using such quantities is
   normally easy to understand. To have an even shorter denotation for
   this favorable species we call such a class a value class or a bit
   more general:

      T is a value class : <==> T is a built-in type or a class
      that implements the strict value interface.

   As was stated already, we have:

      value class T ==> value class Vuc<T>

   The predicates 'T satisfies the value interface' and
   'T satisfies the strict value interface' can be evaluated for
   classes providing random generators by means of the test classes
   Test_v<T> and Test_sv<T> defined in file cpmtests.h. Although Vuc<>
   provides no random generator (Vucr<>, to be mentioned soon, does)
   the code of Test_sv<T> can be read as an operational definition
   of the concept 'strict value interface'. An interesting (or common-
   place ?) observation: syntactic aspects of a program are characterized
```

```
   by the program's interaction with the compiler; semantic aspects
   ('pointer sematics',...) are characterized by the data created during
   execution.

   Requiring order operators or even arithmetic operators in T, allows to
   define T-'valued' Vucectors which themselves carry natural
   order/arithmetic operators.
   Therefore Vuc is only the starting point in a series of vector
   templates with increasing functionality, accompanied by increasing
   assumptions on the structure of T. Presently this hierarchy looks as
   follows:

      Vuc<T> , Vuco<T> , Vuca<T> , Vucr<T>

   Every such class is derived from its predecessor in this series
   by derivation without adding new data members. Thus there are
   unambiguous casts between all these classes.

      Vuco : More order related methods added to Vuc, basic order functions
         which are declared in CPM_ORDER now already here.
      Vuca : arithmetic operations added to Vuco
      Vucr : rich infrastructure supporting automated testing of
         internal consistency.

   This approach of integrating the functions (algorithms) into an
   inheritance tree of template classes is radically different from the
   STL approach. STL keeps algorithms as separate entities outside the
   classes and uses iterators and adaptors for making them work together.
   Both methods have their advantages and disadvantages. It is probably
   fair to say that the C+- approach is less universal but more
   convenient to use within the framework it fits.

4. On polymorphism: Vuc<T*> may be formed, but doesn't implement the value
   interface for T and derived classes. However, with the smart pointer
   templates P<T>, Pp<T>, and Po<T> defined in file cpmp.h we can form
   e.g. Vuc<Pp<T> > and get the functionality (together with some 'extras')
   which one would expect from  Vuc<T*>. See file cpmp.h for details and
   the polymorphic version Vucp of the vector templates mentioned so far.
   See test class PolymorphicMulti<*,*,*> in cpmtests.h for a operational
   definition of polymorphism of container classes.
*/

#include <cpmfl.h> // Includes <cpmuc.h> for std::size_t
   // (and std::ptrdiff_t, which is not being used so far).
   // Small and efficient function template with minimum
   // infra-structure requirements.
#include <cpmzinterval.h>
   // includes cpmsystem.h and cpmx.h
   // With using arrays something may go wrong and so messaging
   // capability is indispensible.
```

```
#include <cpmtypes.h>

#include <cpmmacros.h>
   // Provides help to write debugging-friendly function blocks.
#include <vector>
#include <set>
////////////////////////////// class Vuc<> ////////////////////////////////
// Array with index check, reference counting, and copy on write, and
// 'value semantics' (as opposed to 'pointer semantics')
// Generalized from Koenig's class Handle p. 72-73.
// Index range is now defined as an instance of class IvZ. So each
// 'vector' has its individual index range which may start with 1
// (following the convention of the Numerical Receipes) or with 0 as
// for into the 'vector' of the STL.
//  Vuc< Vuc<ColRef> > is the type of bitmap data in my graphical workhorse
// class Img24. This can be considered a proof for good performance of
// the class. Efficient conversion functions from and to std::vector<>
// are now provided.
// There is a nice way to iterate over the whole index range without
// mentioning the bounds as 0 and dim-1, or as 1 and dim:
//    Vuc<T> v=....;
//    for (Z i=v.b();i<=v.e();++i) v[i]=...;

#define CPM_USECOUNT

namespace CpmArrays{

   using namespace CpmStd;

   using CpmRoot::Word;
   using CpmRoot::Z;
   using CpmRoot::N;
   using CpmRoot::toN;
   using CpmRoot::R;
   using CpmRoot::B;
   using CpmRoot::Root;
   using CpmSystem::Error;
   using CpmFunctions::F;
   using std::vector;

#ifdef CPM_Fn
   using CpmFunctions::F1;
#endif

// some infrastructure

//   enum Begin { LEAN };
      //: begin
      // Never form Vuc<Begin>
```

```
//   enum Outside { DEFAULT, CYCLIC, CONSTANT/*, ZERO */};
      //: outside
      // Controls the meaning of 'out of range indexes'.
      // DEFAULT: definition as the default value associated with
      // the type under consideration
      // CYCLIC: setting the meaning of out of range indexes by
      // cyclic repetition of value
      // CONSTANT: continuation as constant from the nearest value
      // ZERO: ????? not used ?

   extern Z dimMax;
      // If the dimension of an array was either the result of a
      // calculation or of reading from a file, then the result may be
      // off the programmer's intent by orders of magnitude if something
      // went wrong. So it is helpful to exclude unnaturally large arrays
      // from becoming allocated.

   extern Z firInd;
      // first index of arrays. Means to let C+-
      // cooperate with std containers.

   extern bool ranChc;
      //: range check
      // Initialized as true.

   extern bool ranChcAlw;
      //: range check always
      // If this is true, all access operators even those the name
      // of which suggests the opposite get checked --- with poor
      // diagnostics, though.
      // Initialized as true, since access to non-allocated memory,
      // as a rule, causes disaster. I had to discover in 2008-03-03
      // that such a case happened in the workhorse function
      // CpmGraphics::Graph::mark(Vuc< Vuc<C> >,...)
      // on a regular basis, due to an seemingly safe but actually
      // un-safe usage of Vuc<>::cui().

   extern bool signal;


   void setDimMax(Z n);
      // sets dimMax=n unless n<0. In this case error with message

   Z safeDim(Z n);
      // returns n for 0<=n<=dimMax, otherwise error with message

   Z makeIndVucalNotMember(Z i, IvZ iv, Outside mode);
       //: make index valid

template <class T>
```

```
   // We assume that T provides (explicitly or implicitly)
   // copy constructor, and assignment or that T is a built-in type.
   // If the index operator [] is to be used and the variable ranChc
   // (which is initialized as 'true') was not set to 'false' an out
   // of range error will result in a runtime error which will be
   // documented on cpmcerr.txt. See ranChcAlw for an additional control.
   // The error message is particularly explicit (indicating the type of
   // T) when also the macro CPM_NAMEOF is defined. If this is the case,
   // the type T needs to define the member function
   // CpmRoot::Word nameOf()const. For all Cpm-classes this is the case
   // and for user classes, there should be no difficulty in adding such
   // a function. If one wants to make use of the function declared
   // by the declaration macros CPM_ORDER type T needs to define the
   // member function  CpmRoot::Z com(T const&)const;
   // If one wants to make use of the functions declared by the
   // declaration macro CPM_IO, type T needs to define the member
   // functions bool prnOn(ostream&)const and bool scanFrom(istream&).
   // These requirements do not apply to basic types:
   // For T = N, Z, R, Rh, L, bool, string one  may use all functions of
   // Vuc<T> without further requirements.

class Vuc{ // vector template, indexing starts with 1 by default.
   // The index range can however be shifted or even initially set
   // arbitrarily.
   // Although implementation details are inspired from handle classes,
   // the Vuc class is a value array and not a handle class.
   // All allocations made with new T[] all de-allocations are  delete[]
   // All data are private, so the only access to data in
   // derived classes is over the public functions of the class. All
   // these incorporate reference counting internally where needed (only
   // if the preprocessing directive CPM_USECOUNT is active). The user of
   // the class can't see this and has not to be aware of this.

   typedef Vuc<T> Type;

public:
   CPM_IO
   CPM_ORDER
   R dis(Vuc<T> const& h)const;

   Vuc():iv_(),sz_(0),n_(0){ p_=new vector<T>(0); }
      // default constructor

explicit Vuc(Z n):iv_(n), sz_(n), n_(sz_)
   {iv_.b_(firInd); p_=new vector<T>(sz_);}
      // Has n components. Gives an error for n<0 .
      // The components are initialized by the default constructor
      // of T if T is a class for which such a constructor is defined
      // (explicitly or implicitly).
      // If T is a built-in type, initialization is done as 0.
```

```
      // See BS3, p. 131 for initialization of built-in types via
      // formal constructor calls.
      // If n>0, the first valid index is 1, which reflects the
      // normal behavior of class Vuc.

   Vuc(Z n, Begin bg);
      // Defined as the previous function. But the first valid index
      // is 0 (if n>0 so that there is at least one valid index).
      // This is a somewhat contrived construction: One has to make sure
      // that this does not interfere with the definition
      // Vuc(Z n, T const& t, Z first=1) for some choice of T. Since we
      // agree on using Vuc<T> only for C+- types T, we will never be
      // tempted to consider Vuc<Begin>.
      // Typical usage:
      //    Vuc<R> v(4,LEAN); // recall: enum Begin { LEAN }
      // lets v have valid indexes 0,1,2,3. v[0]=...=v[3]=0.
      //    Vuc<R> w(4);
      // lets w have valid indexes 1,2,3,4. w[1]=...=w[4]=0.

explicit Vuc(IvZ const& iv);
      // Has a component for each element of iv.
      // The comFponents are initialized by the default constructor
      // of T if T is a class for which such a constructor is defined
      // (explicitely or implicitely).
      // If T is a built-in type, initialization is done as 0.
      // See BS3, p. 131 for initialization of built-in types via
      // formal constructor calls.

   Vuc(Z first, vector<T> const& v);
      // Construction from a standard library vector.

   Vuc(Z n, T const& t, Z first=1) ;
      // Has n components all initialized as t.
      // The third argument gives the first valid index.

   Vuc(Z n, T const& t, Begin bg);
      // Has n components all initialized as t.
      // The third argument (that can have only one value, namely LEAN)
      // says that the first valid index is 0.

   Vuc(IvZ const& iv, T const& t);
      // Has iv.car() components all initialized as t.

   Vuc(IvZ const& iv, vector<T> const& p):iv_{iv}, sz_{iv_.car()},
      n_{toN(sz_)}
      { cpmassert(n_ == p.size(), "size mismatch in Vuc(IvZ,vector)");
         p_=new vector<T>(p);
      }
      // Has iv.car() components all initialized with the components of p.
```

```
   Vuc(Z n, F<Z,T> const& f);
      // construction from a function. Of course,
      // Vuc<T> v(n,f);
      // implies v[i]==f(i) for all valid indexes i of v.

   Vuc(IvZ const& iv, F<Z,T> const& f);
      // construction from a function. Of course,
      // Vuc<T> v(iv,f);
      // implies v[i]==f(i) for all valid indexes i of v.

 // constructors from explicit lists
   explicit Vuc(std::initializer_list<T> il );
      // requires C++11
      // constructors from explicit lists such as
      // Vuc<Z> v{0,1,2,4,8};
      // The first valid index always is 1. Therefore
      // v[1]=0,...v[5]=8

#if defined(CPM_USECOUNT)
   Vuc(Vuc<T> const& h):iv_(h.iv_),sz_(h.sz_),n_(h.n_),u_{h.u_},p_{h.p_}{}
#else
   Vuc(Vuc<T> const& h):iv_(h.iv_),sz_(h.sz_),n_(h.n_){
      p_=new vector<T>(*h.p_);
   }
#endif

   Vuc<T>& operator=(Vuc<T> const& h){   // assignment
      if (sz_==0 && h.sz_==0) return *this;
         // added 2002-03-07, should be OK
      if (this==&h) return *this;
#if !defined(CPM_USECOUNT)
      Z szMem=sz_;
#endif
      iv_=h.iv_;
      sz_=h.sz_;
      n_=h.n_;
#if defined(CPM_USECOUNT)
      if (u_.reattach_(h.u_)) p_->~vector<T>();
      p_=h.p_;
#else
      if (sz_!=szMem){
         p_->~vector<T>();
         p_=new vector<T>(sz_);
      }
      for (Z i=0;i<sz_;++i) (*p_)[i]=h.li(i);
 #endif
      return *this;
   }

   virtual ~Vuc(){
```

```
#if defined(CPM_USECOUNT)
      if (u_.only()){ p_->~vector<T>();}
#else
      p_->~vector<T>();
#endif
   }

   virtual Vuc<T>* clone(void)const{ return new Vuc(*this);}

   Vuc<T> toClnBase()const{ return *this;}
      //: to clone base

   Z dim()const { return sz_;}
      //: dimension
      // Returns the number of components of the vector *this

// Returns the number of components of the vector represented by
// the built-in integer type std::siz_t (alias N). Thus the term
// vector<X>(nc()) is perfectly right without any type conversion.
   N nc()const{ return p_.size();}
      //. number (of) components

   Z size()const { return sz_;}
      //: size
      // for uniformity with STL

   IvZ dom()const { return iv_;}
      //: domain
      // Notice that with the array *this there is the
      // function f: {iv_b(),...,iv_.e()}-->T, i|-->(*this)[i]
      // associated in a natural manner.
      // For this function, dom() is just the domain.
      // The understanding of arrays as functions with domains of type
      // IvZ seems to be a good guide for defining some of the member
      // functions in a more natural manner by using arguments of type
      // IvZ. Present examples are the functions fa_ and valOn.

   bool isVoid()const { return iv_.isVoid();}
      //: is void
      //  short answer on whether the dimension is zero

   bool valInd(Z i)const{ return iv_.hasElm(i);}
      //: valid index
      // returns the validity of i as an index of *this

   Z makeIndVucal(Z i, Outside mode=CYCLIC )const
   {
      if (iv_.hasElm(i)) return i;
      if (mode==CYCLIC) return iv_.cyc(i);
      else return iv_.con(i);
```

```
   }
      //: make index valid
      // If i is a valid index we return i. If not, the result is
      // iv_.cyc(i) for mode=CYCLIC and iv_.con(i) else. Notice that the
      // normal treatment of mode==DEFAULT works not by replacing one
      // value of the index by another. It works on the value of vector
      // components and makes use of the default constructor T()

   bool sameDom(Vuc<T> const& v)const{ return iv_==v.iv_;}
      //: same domain

   vector<T> std()const{ return *p_;}
      //: standard
      // Returns a STL-vector which holds all components of *this.

   vector<T> toVuc()const{ return *p_;}

   virtual Word nameOf()const;
      //: name of
      // returns a name of the type

// constant access functions

   const T& operator[](Z i)const;
      // Getting to the value of a component of a const instant of
      // Vuc<T>. For instance
      // const Vuc<T> v = ... ;
      // T t = v[3];
      // If CPM_USECOUNT is enabled (the normal case) the actual
      // process of going from v to v[3] depends on whether v is of type
      // Vuc<T> or const Vuc<T>. In the non-constant case the evaluation of []
      // may involve a copy action on v (and returning the component of
      // the copy). If we intend only to read the component, such an
      // copy action is not neededd and should be avoided by casting v
      // to type const Vuc<T>.
      // Casting v from Vuc<T> to const Vuc<T> works this way:
      //    v = static_cast<const Vuc<T>>(v);
      // Casting back to mutable seems to work only by using a new name:
      //    auto vMutable = const_cast<Vuc<T>&>(v);
      //    vMutable[1] = T(); //( for instance)
      // or by applying mutating operations to an anonymous object as in:
      //    const_cast<Vuc<T>&>(v)[1]=T();
      // Notice the '&' here which the compiler requires. By the way, the
      // effect of the 'anonymus action' actually is to change the state
      // of v: v[1]==T().
      // My analysis of the situation is in ~/codingexperiments/main.cpp
      // In the following there are many pairs of functions
      // T const& f(...)const;
      // T& f(...);
      // for which the above considerations apply mutandis mutatis.
```

```
T const& cui(Z i)const;
   //. component (with) unchecked index
   // getting to the value of a component for 'read', e.g.
   // T t=cui(i);
   // It helps to write efficient code in classes which use Vuc<>-typed
   // data members.
   // Notice that writing loops by using b() and e() to define
   // the range is much safer than using limits like 1 and dim().
   // Index range check is missing only if variable ranChcAlw was
   // changed to 'false'.

T const& pi(N i)const{ return (*p_)[i];}
   //. plain index
   // index range not checked.
   // It helps to write efficient code in classes which use Vuc<>-typed
   // data members.

T const& pr(Z i, Outside mode)const{ //. plain read, see below for more
   if (0 <= i && i < sz_) return (*p_)[i];
      // Notice that for i of type CpmRoot::Z the expression (*p_)[i]
      // is perfectly right. Of course, the same is true for i of
      // type std::size_t (for which CpmRoot::N is an alias).
   if (i == -1){
      if (mode==DEFAULT) return def_;
      if (mode==CYCLIC) return (*p_)[n_-1];
      if (mode==CONSTANT) return (*p_)[0];
   }
   if (i == sz_){
      if (mode==DEFAULT) return def_;
      if (mode==CYCLIC) return (*p_)[0];
      if (mode==CONSTANT) return (*p_)[n_-1];
   }
   return def_; // should never happen
}
   //. plain read
   // from an index which may be out of range by one, in which case
   // the return value is determined in an self-evident way by the
   // Outside-typed second argument.

T& pi(N i){ return (*p_)[i];}
   //. plain index
   // index range not checked.
   // It helps to write efficient code in classes which use Vuc<>-typed
   // data members.

T const& cyc(Z i)const;
   //: cyclic
   // getting to the value of a component for 'read', e.g.
   // T t=cyc(i);
```

```
    // Here i is understood as cyclic (i.e. i modulo sz_). So no value
    // of the index has to be considered 'out of range' and for i
    // 'in range' cyc(i)==(*this)[i]

  T const& li(Z i)const{ return (*p_)[i];}
    //. lean index
    // The valid range is for (Z i=0;i<sz_;++i) li(i)

  T& li(Z i){ return (*p_)[i];}
    //. lean index

  T const& con(Z i)const;
    //: constant
    // Same logic as cyc() but with constant continuation
    // instead of cyclic.

  T operator()(Z i, Outside mode=DEFAULT)const;
    //: ()
    // Read access defined for all i.
    // By this definition, a vector becomes a mapping from Z to T.
    // For i's outside the proper range,
    // the default T is returned for mode==DEFAULT or if
    // sz_==0. If mode==CYCLIC cyc(i) gets returned.
    // For mode==CONSTANT we return (*p_)[0] for i<=0 and (*p_)[sz_-1]
    // for i>=sz_.
    // Notice that the return value is not a reference in
    // accordance with function behavior.

  T const& read(Z i, Outside mode=DEFAULT)const;
    //: read
    // Same as previous function but returning a reference instead of a
    // T.
    // Reading components in an efficient (as &'s), safe and flexible
    // manner.
    // See T operator()(Z i, Outside mode=DEFAULT)const; for the
    // meaning of the second argument.

  T const& r(Z i)const{ return (*p_)[iv_.ri(i)];}
    //. read
    // Unchecked and fast version of T const& oparator[](Z i)const

  T& w(Z i)const{ return (*p_)[iv_.ri(i)];}
    //. write
    // Unchecked and fast version of T& oparator[](Z i)

  F<Z,T> fnc(Outside meth=DEFAULT)const;
    // function
    // returns the function Z --> T, i |--> (*this)(i,meth)

// non-constant access functions
```

```
   T& operator[](Z i);
      //: []
      // See T const& operator[](Z i)const for discussion of details
      // that are relevant in the case that CPM_USECOUNT is defined.

   T& cui(Z i);
      //. component (with) unchecked index
      // setting to the value of a component
      // T t=...;
      // Vuc<T> x=...;
      // x.cui(i)=t;

    T& cyc(Z i);
      //: cyclic
      // setting to the value of a component
      // T t=...;
      // Vuc<T> x=...;
      // x.cyc(i)=t; meaning of i is modulo sz_. So i is never out of
      // range

   T& con(Z i);
      //: constant
      // Same logic as cyc() but with constant continuation
      // instead of cyclic.

   Vuc<T>& b_(Z i){ cow_();iv_.b_(i); return *this;}
      //. set b(), i.e. the first valid index.
      // For instance
      // Vuc<R> v("",sqrt(2),sqrt(3),sqrt(4),sqrt(5));
      // for (Z i=1;i<=v.dim();++i) cout<<v[i]<<endl;
      // v.b_(0);
      // for (Z i=0;i<v.dim();++i) cout<<v[i]<<endl;
      // shows all components of the vector in both cases.

   Vuc<T>& e_(Z i){ cow_();iv_.e_(i); return *this;}
      //. set e(),  i.e. the last valid index.

// accessing the first and the last element for reading
   T const& fir()const;
      //: first
   T const& last()const;
      //: last

// accessing the first and the last element for writing
   T& fir();
      //: first
   T& last();
      //: last

// getting the first and the last valid index
```

```
Z b()const { return iv_.b();}
   //. begin
   // Returns the first valid index.

Z e()const { return iv_.e();}
   //. end
   // Returns the last valid index.
   // Allows to write loops over all components as
   // for (Z i=v.b();i<=v.e();i++) ... v[i] ...;
   // Note that this is safe also for v.dim()==0, since then
   // v[e()] will never be called. Here one could replace
   // v[i] by v.cui(i) without danger.

Z n()const { return iv_.n();}
   //. next
   // Returns the index next to the last valid one.
   // This allows to write loops over all components as
   // for (Z i=v.b();i<v.n();i++) ... v[i] ...;
   // Note that this is safe also for v.dim()==0, since then
   // v[e()] will never be called. Here one could replace
   // v[i] by v.cui(i) without danger.

Vuc<T> meet(IvZ const& iv)const;
   //: meet
   // Returns a vector which, when considered as a function is the
   // restriction of function *this to the domain iv_ & iv.

Vuc<T> join(IvZ const& iv)const;
   //: join
   // Returns a vector which, when considered as a function is the
   // extension of function *this to the domain iv_ | iv, where
   // all function values on iv\iv_ are T().

Vuc<T> operator +(Z i)const{ return Vuc<T>(iv_+i,p_,"");}
   //: operator +
   // Returns a vector res such that res.dom() is the shifted
   // domain dom()+i of *this and has the same components as
   // *this.

Vuc<T> operator -(Z i)const{ return Vuc<T>(iv_-i,p_,"");}
   //: operator -
   // Returns a vector res such that res.dom() is the shifted
   // domain dom()-i of *this and has the same components as
   // *this.

void set_(T const& t);
   //: set
   // non-constant function which sets all components of (*this)
   // equal to t
```

```
Vuc<T> set(Z i, T const& t)const;
   //: set
   // Returns a vector that originates from *this by setting the
   // value of component i.
   // Regular behaviour:
   // if we say
   // Z i=...;
   // T t= ...;
   // Vuc<T> v=...;
   // v=v.set(i,t);
   // then the i-th component (see eli() ) of v will get the value t
   // unless i<1.
   // If i is a value for which (*this)[i-1] is not yet defined,
   // the vector becomes enlarged to the necessary size and all
   // not specified components initialized with the default
   // constructor of T

T in_(T const& t, bool reversed=false);
   //. insert
   // This operations treats *this as a shift register:
   // All components get shifted by one position 'to the right' and
   // the total length (dim) of the vector remains the same. So what
   // was the last component prior to the operation has to be removed
   // from the vector, in order to not wasting information, this
   // removed component will shine up as the return value of the
   // function. After the operation, the first component of the vector
   // will be t.
   // If reversed==true, t gets inputted at the end, and all shift
   // operations go 'to the left'.


Z locAsc(T const& t)const;
   //: locate ascending
   // Here it is assumed that *this is ascendingly strictly ordered:
   // If sz_>=2 we have (*p_)[i]<(*p_)[i+1] for all i \in {0,sz_-2}.
   // For sz_<2 there is no condition.
   // For sz_==0 we stop with error.
   // For t<fir() we return b()-1
   // For t>=last() we return e()
   // If none of the previous conditions was met we necessarily have
   // sz_>=2 and we return the uniquely determined j such that
   //    (*p_)[j]<=t<(*p_)[j+1].
   // The possible results from this regular part of the functionality
   // thus are b(),...,e()-1.
   // Notice that the very similar function locate of Press et al.
   // only guarantees (*p_)[j]<=t<=(*p_)[j+1] for its return value j.

Z find(T const& t, bool ordered=true)const;
   //: find
   // If the return value is b()-1, there is no valid index i
```

```
      // such that (*this)[i]==t. Otherwise, the return a value i
      // such that (*this)[i]==t.
      // If ordered==true, this assumes that (*this) is in increasing
      // order. The method is by bisection and logarithically fast.
      // If no order is assumed the components are compared against t
      // starting from index b(). The first occurrence of t gets reported.

// building new objects from given ones (generative methods)

// concatenating vectors and appending components. These operations have
// always O(sz_) complexity. The corresponding combined assignements are
// less direct to implement are not considered useful in the present
// context (they would not be more efficient than the friend versions,
// since new memory has to be allocated anyway).

   Vuc<T> app(T const& t)const;
      // returns a vector which is obtained from *this by appending t
      // as the last component
      // notice also the prepend functions to be introduced
      // after the insert-functions (in order to have the inline
      // definition available).

   Vuc<T> app(Vuc<T> const& h)const;
      // returns a vector which is obtained from *this by appending h
      // at the back end

   void push_back(T const& t) { *this=app(t);}
      // for uniformity with STL

   Vuc<T>& operator<<(T const& t) { return *this&=t;}
      // appending an element in Ruby-style
      // Allows successive application as in
      // Vuc<Word> w;
      // w<<"many"<<"words"<<"get"<<"easily"<<"stored";

   Vuc<T>& operator<<(Vuc<T> const& h) { return *this=app(h);}
      // appending an array in Ruby-style

   Vuc<T>& operator&=(T const& t){
      iv_.n_();
      sz_++;
      p_->push_back(t);
      return *this;
   }

   Vuc<T>& operator&=(Vuc<T> const& h) { return *this=app(h);}
   Vuc<T> operator&(T const& t)const { return app(t);}
   Vuc<T> operator&(Vuc<T> const& h)const { return app(h);}
      // appending in my favorite style
```

```
   Vuc<IvZ> valOn(F<T,bool> const& f)const;
      //: valid on
      // retuns the array of those sub-intervals of the
      // whole indexing interval dom() on which the function
      // dom()-->bool, i|-->f((*this)[i]) yields true.
      // Notice that the result res \in Vuc<IvZ> is a convenient
      // representation of a subset of dom(). Considering
      // *this as a function g: dom()-->T, then the function
      // h:=g&f is of type dom()-->bool and res, as a subset
      // of dom(), is h^-1({true}).

// resizing

   Vuc<T> resize(Z newDim)const;
      // Returns a vector res such that res.dim()=newDim. If newDim is
      // smaller than dim(), the end of *this will be cut away. If newDim
      // is larger, T()'s will be added.
      // For newDim<0 the action is as if newDim==0.

   Vuc<T> cut(Z i)const{ return resize(sz_-i);}
      // returned is a Vuc<T> which results from *this by removing i
      // components from the end. For exotic values of i, see code
      // and explanation of resize.

// elimination and insertion

   Vuc<T> eli(IvZ const& iv)const;
      // eli stands for eliminate
      // Eliminating all components which belong to iv. No exceptions
      // can happen! Universal and convenient of elimination. All other
      // forms are superfluous. Moreover, they are to be considered as
      // obsolete.

   Vuc<T> eli(Z i, Z nEli=1)const{ return eli(IvZ(nEli,i,0));}
      // eli stands for eliminate
      // returned is a vector which is obtained from *this by
      // eliminating the i-th component and the nEli-1 following ones
      // (thus nEli components are removed) and shifting all later
      // components (if there are such components left) 'to the left' to
      // close the gap.

  // Vuc<T> eliFirst(Z nEli=1)const { return eli(1,nEli);}
   Vuc<T> eliFirst(Z nEli=1)const { return eli(IvZ(nEli,b(),0));}
      // returned is a vector which is obtained from *this by
      // eliminating the nEli first components. If no
      // argument is provided, actually the first component
      // gets eliminated. Notice that in the three-argument constructor
      // IvZ(i,j,k) the first argument is the cardinality, the second
      // argument is the first element, and the third argument is dummy.
```

```
// Vuc<T> eliLast()const { return eli(sz_,1);}
 Vuc<T> eliLast()const { return eli(IvZ(e(),e()));}
    // returned is a vector which is obtained from *this by
    // eliminating the last component.

 Vuc<T> eli1(Vuc<T>& h, Z i)const;
    // We return a vector which results from *this by eliminating
    // h.dim() components starting at the i'th component (for i<1,
    // i==1 is understood). After the call h will hold the
    // eliminated components in due order (and no more - even if h was
    // longer before).
    // The function name ends in '1' to indicate that the first
    // argument is a non-constant reference, used for communication
    // of a part of the result.

 // condensation (contracting equivalent components into one)

 X2< Vuc<T>, Vuc<Z> > condense( bool (*equi)(const T&, const T&))const;
    // given an equivalence relation equi on T (t1~t2 <==>
    // equi(t1,t2)==true )
    // we divide the components of *this into equivalence classes.
    // Let the resturned pair be written as (res1,res2) . Then
    // (i) (*this)[i]~res1[res2[i]]
    // (ii) there is a j such that  (*this)[j]==res1[res2[i]]
    // Actually, the construction is made such that this j is the
    // smallest j for which (*this)[j]~res1[res2[i]].

 Vuc<T> select(Vuc<B> const& s)const;
    // returned is a list which is obtained from *this by eliminating
    // all components (*this)[i] for which s[i] is defined and
    // statisfies s[i]==false. Beside of this removal, the order of the
    // components in *this is retained. So if s is defined
    // by a condition s[i]=Condition((*this)[i]), for the vector
    // component, this condition has to express a property we like to
    // have fulfilled for the result-vector of the select-operation.
    // 's expresses the favorable condition' and  n o t  the one to be
    // eliminated. Notice, that the select operation makes sense for
    // all values of s.dim().

 Vuc<T> ins(Z i, T const& t)const;
    //: insert
    // returned is a vector which is obtained from *this by
    // inserting t as the i-th component and shifting all later
    // components 'to the right' to give room for t.
    // The phrase 'i-th component' refers to natural counting (thus
    // (*p_)[i-1] is the i-th component of *this).

 Vuc<T>& ins_(Z i, T const& t);
    //: insert
    // returned is a vector which is obtained from *this by
```

```
   // inserting t as the i-th component and shifting all later
   // components 'to the right' to give room for t.
   // The phrase 'i-th component' refers to natural counting (thus
   // (*p_)[i-1] is the i-th component of *this).


Vuc<T> ins(Z i, Vuc<T> const& h)const ;
   // returned is a vector which is obtained from *this by
   // inserting h as the i-th and following component,
   // and shifting all later components of *this
   // 'to the right' to give room for h.
   // The phrase 'i-th component' refers to natural counting (thus
   // (*p_)[i-1] is the i-th component of *this).

Vuc<T> prepend(T const& t)const
   // returns a vector which is obtained from *this by appending t
   // as the first component
   { return ins(1,t);}

Vuc<T>& prepend_(T const& t);
   // changes *this by appending t  as the first component

Vuc<T> prepend(Vuc<T> const& h)const
   // returns a vector which is obtained from *this by appending h
   // at the front end
   { return h.app(*this);}

Vuc<T> rev()const;
   //: reversed
   // returned is the reversed vector (indexing in the opposite
   // direction)

Vuc<T>& rev_(){ return *this = rev();}
   //: reversed
   // changes *this into a reversed version of itndexing in the
   // opposite direction

// re-indexing

Vuc<T> rot(Z s, Outside mode=CYCLIC)const;
   //: rotate
   // Name as the list transformation function Rotate of Mathematica.
   // v.rot(s)[i] == v(i-s,mode)
   // This means that we shift the component i of the original vector
   // into the new position i+s

void rot_(Z s, Outside mode=CYCLIC){ *this=rot(s,mode);}
   //: rotate
   // Same as rot, but as a mutating operation.
```

```
   Vuc<T> compose(Vuc<Z> const& w)const;
      //:: compose
      // v.compose(w)[i] = v[w[i]]

   Vuc<T> permute(Vuc<Z> const& w)const{ return compose(w);}
      //:: permute

   // transforming components

   Vuc<T> operator+(Vuc<T> const& h)const{
      cout<<"operator + called"<<endl;
      Vuc<T> res(sz_);
      for (Z i=0; i<sz_; ++i){ (res.p_)[i]=(*p_)[i]+(h.p_)[i];}
      return res;
   }

   template <class Y>
   Vuc<Y> operator()(F<T,Y> const& f)const { return Vuc<Y>(iv_,fnc()&f);}
      // generating arrays of different type by a type changing function

   Vuc<T> operator()(T (*f)(T const&))const{ return fa(f);}
      // generating arrays of the same type with components f(c) instead of
      // c.

   Vuc<T> operator()(T (*f)(T))const{ return faa(f);}
      // generating arrays of the same type with components f(c) instead of
      // c.

   Vuc<T> operator()(std::function<T(T)>(f))const{ return fsf(f);}
      // generating arrays of the same type with components f(c) instead of
      // c.

// defining generative laws and mutating laws for various expressions by
// function pointers. The idea behind is, that for T which allows
// particular operations, we can define those without using iteration via
// a operator[] since these all are defined directly in terms of
// pointers.
// See implementation of +=, -=, ... in Vuca<T> how this works

   T fAcc1(T (*f)(T const&), void (*acc)(T&, T const&))const;
      // returns an accumulated value of all values f(c), where c
      // are the components of *this. Accumulation is defined by
      // the argument acc:
      // T res=T(); 'for all components c' acc(res,c)

   T fAcc2(T (*f)(T const&, T const&), void (*acc)(T&, T const&),
      Vuc<T> const& h )const ;
      // returns an accumulated value of all values f(c,ch), where c and
      // ch are the components of *this and h respectively. Accumulation
```

```
   // is defined by the argument acc:
   // T res; 'for all components c' acc(res,c)
   // useful in defining scalar products

template <class Y>
Y fAcc3(Y (*f)(T const&, T const&), void (*acc)(Y&, Y const&),
   Vuc<T> const& h )const ;
   // returns an accumulated value of all values f(c,ch), where c and
   // ch are the components of *this and h respectively. Accumulation
   // is defined by the argument acc

Vuc<T> fAcc4(F<T2<T>,T> const& f)const;
   // The kind of accumulation needed for computing the forces in
   // particle systems with pair interaction. Here we assume that
   // forces and positions are of the same type, e.g. R2 or R3.
   // Let x[1],...x[n] be the components of *this. We first compute
   // the incomplete matrix f(x[i],x[j]) i=1,...n, j<i and complete it
   // assuming f(x[i],x[j]) = - f(x[j],x[i]). In application mentioned
   // earlier this is the collection of mutually forces. The total force
   // on particle i is sum over j of f(x[i],x[j]). It is the i-th
   // component of the Vuc<T>-typed return value of the function.
   // The force type T thus needs to provide operations unary - and +=.

Vuc<T> fAcc5(F<R,R> const& dPot)const;
   // The kind of accumulation needed for computing the forces in
   // particle systems with pair interaction derived from a distance-
   // dependent (radial symmetric) potential. The argument dPot is
   // the derivative with respect to r of the r-dependent radially
   // symmetric pair potential.

Vuc<T> fAcc6(F<R,R> const& dPot)const;
   // forces from a space-dependent potential

Vuc<T> fsf(std::function<T(T)> f)const;
   // transforming the components with std::function

Vuc<T> fa(T (*f)(T const&))const ;
   // returns a vector defined by replacing the components c of *this
   // by f(c)

Vuc<T> faa(T (*f)(T))const ;
   // returns a vector defined by replacing the components c of *this
   // by f(c)

Vuc<T> fb(T (*f)(T const&, T const&), T const& t)const;
   // returns a vector defined by replacing the components c of *this
   // by f(c,t)

template <class Y>
Vuc<T> fb2(T (*f)(T const&, Y const&), Y const& t)const;
```

```
   // returns a vector defined by replacing the components c of *this
   // by f(c,t)

template <class Y>
void fb2_(T (*f)(T const&, Y const&), Y const& y);
   // replaces *this by a vector defined by replacing the components
   // c of *this by f(c,y)

void fc_(T (*f)(T const&, T const&), T const& t) ;
   // replaces *this by a vector defined by replacing the components c
   // of *this by f(c,t)

void fd(T (*f)(T const&, T const&), T& t)const;
   // replaces t by f(c,t) for each component c
   // If, for instance, f(c,t)=t+c*c we replace t by t+sum of c*c

Vuc<T> fe(T (*f)(T const&, T const&), Vuc<T> const& h)const;
   // returns a vector defined by replacing the components c of *this
   // by f(c,c') where c' are the components of h.
   // Error if dim()!=h.dim()

template <class Y>
Vuc<T> fet(T (*f)(T const&, Y const&), Vuc<Y> const& h)const;
   // returns a vector defined by replacing the components c of *this
   // by f(c,c') where c' are the Y-typed components of h.
   // Error if dim()!=h.dim(). Template version of fe. Unfortunately
   // h.p_ is not accessible in the implementation of this function.
   // This enforced introducing the non-canonical access function rep()
   // in 2014-01-23.

Vuc<T> fe2(T (*f)(T const&, T const&), Vuc<T> const& h)const;
   // Vucery similar to fe. However, the dimension of the result
   // is the maximum of dim() and h.dim(). Non-existing components
   // of any of the operands are replaced by T().

void ff_(T (*f)(T const&, T const&), Vuc<T> const& h, Z i=0);
   // Replaces the components of p_ starting from (*p_)[i]
   // by f(c,c') where c' are the components of h.p_. Thus for i=0
   // and h.dim()>=sz_ the whole array p_ is affected.

void ffl_(T (*f)(T const&, T const&), Vuc<T> const& h);
   // lean form of ff_
   // Replaces the components of p_
   // by f(c,c') where c' are the components of h.p_
   // p_ and h.p_ are assumed to be of the same size


void fg_(T (*f)(T const&, T const&, T const&),
   Vuc<T> const& h, T const& t);
   // replaces the components c of *this
```

```
   // by f(c,c',t) where c' are the components of h

void fa_(F<T,T> const& f, IvZ iv);
   // replaces the components c of *this with index in iv
   // by f(c); modern form of fa.

void fh_(T const& w1, T const& w2, T const& w3, Outside mode);
   // replaces the component c[i] of *this by
   // w1*read(i-1,mode)+w2*read(i,mode)+w3*read(i+1,mode)
   // Thus makes use of multiplication in T.

template <class Y>
void fht_(Y const& w1, Y const& w2, Y const& w3, Outside mode);
   // replaces the component c[i] of *this by
   // read(i-1,mode)*w1+read(i,mode)*w2+read(i+1,mode)*w3
   // Thus makes use of multiplication in T.

template <class Y>
Vuc<T> fh(T (*f)(T const&, T const&, T const&, Y const&, Z),
   Y const&, Outside mode)const;
   // Returns a vector in which the component c[i] is replaced
   // by f(c[i-1],c[i],c[i+1],y,i),where the i-/+1 are understood
   // according to mode. The quantity y helps to provide parameters
   // required by a concrete situation. Worked for implementing
   // the Hamiltonian corresponding to Dirac's relativistic wave
   // equation.

template <class Y>
Vuc<Y> fi(Y (*f)(T const&, T const&), void (*acc)(Y&, Y const&))const;
   // Returns a vector with Y-valued components y[i] which are obtained
   // by accumulating the terms f((*this)[i],(*this)[j]).
   // Accumulation is defined by the argument acc.

template <class Y>
Vuc<Y> fj(Y (*f)(T const&, T const&), void (*acc)(Y&, Y const&))const;
   // Returns a vector with Y-valued components y[i] which are obtained
   // by accumulating the terms f((*this)[i],(*this)[j]).
   // Accumulation is defined by the argument acc.
   // We assume f((*this)[i],(*this)[j]) == - f((*this)[j],(*this)[i])
   // and use this for doing only one evaluation of f
   // for the two pairs (i,j) and (j,i) and no evaluation
   // of f for any pair (i,i).
   // We assume that for any Y-object y, -y is defined.

  vector<T> rep()const{ return p_;} // 20014-01-24
   // This is needed in the implementation of Vuc<T>::fet<Y>. Here we need
   // direct access to the data member p_ of a function argument of type
   // Vuc<Y>. Unfortunately (and unexpectedly) what is OK for Vuc<T> does
   // not work for Vuc<Y>. This function conflicts with my ambition to
   // ban pointers from the public interface of C+- classes.
```

```
private:
// static data
   static const T def_;
      // allows read function to return references
// static functions
   static T fCyc(Z const& i, Vuc<T> const& v) { return v(i,CYCLIC);}
   static T fCon(Z const& i, Vuc<T> const& v) { return v(i,CONSTANT);}
   static T fDef(Z const& i, Vuc<T> const& v) { return v(i,DEFAULT);}
 //  static N toN(Z i) { return static_cast<N>(i);}
      // size type
      // Instead of new T[i], I now write new T[toN(i)] so that
      // allocation is always fed with a parameter which fits the system
      // (in 64 bit systems, size_t may be 64 bit wide). size_t is known
      // due to inclusion of <cpmfl.h> and it is assumed to take the
      // bit-width of the machine properly into account.
   void ini_(){sz_=iv_.car();n_=toN(sz_);p_=new vector<T>(n_);}
 //  void ini_(T *p){sz_=iv_.car();p_=p;}

 protected:

   void cow_(){
#if defined(CPM_USECOUNT)
      if (u_.makeOnly()){
         p_=new vector<T>(*p_);
         u_.startNew_();
      }
#endif
   }


// data members
   // These should be accessible to derived classes for enabling fast
   // component operations.
   IvZ iv_;
      // Set of valid indexes. Since (*this) can be considered a
      // function iv_ --> T, this object is also called the domain
      // of *this.

   Z sz_{0};
      // number of valid indexes (depends on iv_, equals iv_.car())

   N n_{0};

#if defined(CPM_USECOUNT)
   UseCount u_;
#endif

   std::vector<T> *p_;
      // data member that holds the components as a container.
```

```
};

//////// using Vuc for some special template arguments//////////////////

Vuc<Z> IvZtoVucofZ(IvZ const& iv);
   //: IvZ to Vuc of Z
   // returns the Z's that make up the interval iv
   // as the components of an ordered array (increasing
   // order, of course)

Vuc<Z> VucofIvZtoVucofZ(Vuc<IvZ> const& viv);
   //: Vuc of IvZ to Vuc of Z
   // appends the results from applying the previous functions
   // to the viv[i] according to the obvious code
   //  { Vuc<Z> res(0); Z n=viv.dim();
   //  for (Z i=0;i<n;++i) res&=IvZtoVucofZ(viv[i]); return res;}


////////////////////// Implementation ////////////////////////////////

template <class T>
const T Vuc<T>::def_=T();

template <class T>
Word Vuc<T>::nameOf()const{
   Word nt=Root<T>(T()).nameOf();
   Word wi="Vuc<";
   return wi&nt&">";
}

template <class T>
R Vuc<T>::dis(Vuc<T> const& h)const
{
   R res=0.;
   if (iv_!=h.iv_) return R(1);
   for (Z i=b();i<=e();++i){
      res+=Root<T>(cui(i)).dis(h.cui(i));
   }
   return res;
}

template <class T>
T const& Vuc<T>::operator[](Z i)const
{
   if (ranChc){
      if (!iv_.hasElm(i)){
         cout<<"index access error at i="<<i<<endl;
         cpmerror(nameOf()&
         "::operator[]const: read-index out of  range: i= "&
         cpm(i)&" iMin= "&cpm(iv_.b())&" iMax= "&cpm(iv_.e()));
```

```
      }
   }
   if (signal) cout<<" const [] called"<<endl;
   return (*p_)[iv_.ri(i)];
}


template <class T>
T& Vuc<T>::operator[](Z i)
{
   cow_();
   if (ranChc){
      if (!iv_.hasElm(i)){
         cout<<"index access error at i="<<i<<endl;
         cpmerror(nameOf()&"::operator[]: write-index out of  range: i= "&
         cpm(i)&" iMin= "&cpm(iv_.b())&" iMax= "&cpm(iv_.e()));
      }
   }
   if (signal) cout<<" mutating [] called"<<endl;
   return (*p_)[iv_.ri(i)];
}

// using intentionally previously defined []-indexing
// for common messaging and safeness.
template <class T>
T const& Vuc<T>::fir()const{ return (*this)[iv_.b()];}

template <class T>
T& Vuc<T>::fir(){ return (*this)[iv_.b()];}

template <class T>
T const& Vuc<T>::last()const{ return (*this)[iv_.e()];}

template <class T>
T& Vuc<T>::last(){ return (*this)[iv_.e()];}

template <class T>
inline T const& Vuc<T>::cui(Z i)const
{
   if (ranChcAlw){
      if (!iv_.hasElm(i)) cpmerror("index out of range");
   }
   return (*p_)[iv_.ri(i)];
}

template <class T>
inline T& Vuc<T>::cui(Z i)
{
   cow_();
   if (ranChcAlw){
```

```
      if (!iv_.hasElm(i)) cpmerror("index out of range");
   }
   return (*p_)[iv_.ri(i)];
}

template <class T>
T const& Vuc<T>::cyc(Z i)const
{
   return (*this)[iv_.cyc(i)];
}

template <class T>
T& Vuc<T>::cyc(Z i)
{
   cow_();
   return (*this)[iv_.cyc(i)];
}

template <class T>
T const& Vuc<T>::con(Z i)const
{
   if (sz_==0) return def_;
   return (*this)[iv_.con(i)];
}

template <class T>
T& Vuc<T>::con(Z i)
{
   cow_();
   if (sz_==0){
      cpmerror("Vuc<T>::con(Z i): i="&cpm(i)&
         " is no valid index in void array");
      return (*p_)[0]; // never happens
   }
   return (*this)[iv_.con(i)];
}

template <class T>
T const& Vuc<T>::read(Z i, Outside mode)const
{
   if (iv_.hasElm(i)) return (*p_)[iv_.ri(i)];
   else{
      if (mode==DEFAULT) return def_; // reference to it can be returned
      else if (mode==CYCLIC) return cyc(i);
      else return con(i);
   }
}

template <class T>
T Vuc<T>::operator()(Z i, Outside mode)const
```

```
{
   if (sz_== 0){
       return T();
   }
   else if (iv_.hasElm(i)){
      return (*p_)[iv_.ri(i)];
   }
   else if (mode==DEFAULT){
      return T();
   }
   else if (mode==CYCLIC){
      return cyc(i);
   }
   else  return con(i);
}

template <class T>
F<Z,T> Vuc<T>::fnc(Outside mode)const
{
#ifdef CPM_Fn
   if (mode==DEFAULT) return F1<Z,Vuc<T>,T>(*this)(fDef);
   else if (mode==CYCLIC) return F1<Z,Vuc<T>,T>(*this)(fCyc);
   else return F1<Z,Vuc<T>,T>(*this)(fCon);
#else
   if (mode==DEFAULT) return F<Z,T>(bind(fDef,_1,*this));
   else if (mode==CYCLIC) return F<Z,T>(bind(fCyc,_1,*this));
   else return F<Z,T>(bind(fCon,_1,*this));
#endif
}

// constructors
// 137 is a dummy argument

/*
template <class T>
Vuc<T>::Vuc(Z n):iv_(safeDim(n),1,137)
{
   ini_();
}
*/

template <class T>
Vuc<T>::Vuc(Z n, Begin bg):iv_(safeDim(n),0,137)
{
   ini_();
}

template <class T>
Vuc<T>::Vuc(Z n, T const& t, Begin bg):iv_(safeDim(n),0,137),
   sz_(iv_.car()),n_(toN(sz_)){p_= new vector<T>(n_,t);}
```

```
template <class T>
Vuc<T>::Vuc(Z n, T const& t, Z first):iv_(safeDim(n),first,137),
   sz_(iv_.car()),n_(toN(sz_)){p_= new vector<T>(n_,t);}

template <class T>
Vuc<T>::Vuc(Z n, F<Z,T> const& f):iv_(safeDim(n),1,137),sz_(iv_.car()),
n_(toN(sz_)),p_(n_)
{
   N i=0;
   for (Z j=iv_.b();i<n_;++i,++j) (*p_)[i] = f(j);
}

template <class T>
Vuc<T>::Vuc(IvZ const& iv):iv_(iv), sz_(iv_.car()), n_(toN(sz_)){ p_=new vector<T>(n_); }


template <class T>
Vuc<T>::Vuc(IvZ const& iv, T const& t):iv_(iv), sz_(iv_.car()),
   n_(toN(sz_)){ p_=new vector<T>(sz_,t);}

template <class T>
Vuc<T>::Vuc(IvZ const& iv, F<Z,T> const& f):iv_(iv),sz_(iv_.car()),
   n_(toN(sz_))
{
   p_=new vector<T>(sz_);
   N i=0;
   for (Z j=iv_.b();i<n_;++i,++j) (*p_)[i] = f(j);
}

template <class T>
Vuc<T>::Vuc(std::initializer_list<T> il ):
iv_(il.size()), sz_(iv_.car()), n_(toN(sz_)){p_=new vector<T>({il});}
   // notice that, in view of the construction of iv_, the first
   // valid index of the constructed vector is 1

template <class T>
Vuc<T>::Vuc(Z first, std::vector<T> const& v):
iv_(v.size(),first,137), sz_(iv_.car()), n_(toN(sz_)){p_=new vector<T>(v);}

template <class T>
Vuc<T> Vuc<T>::meet(IvZ const& iv)const
{
   IvZ ivRes=iv_.meet(iv);
   Vuc<T> res(ivRes); // all components are T()
   for (Z i=res.b();i<=res.e();++i){
       if (iv.ni(i)) res[i]=(*this)[i];
   }
   return res;
```

```
}

template <class T>
Vuc<T> Vuc<T>::join(IvZ const& iv)const
{
   IvZ ivRes=iv_.join(iv);
   Vuc<T> res(ivRes); // all components are T()
   for (Z i=res.b();i<=res.e();++i){
       if (iv_.ni(i)) res[i]=(*this)[i];
   }
   return res;
}

template <class T >
Z Vuc<T>::find(T const& t, bool ordered)const
{
   Z n=dim();
   Z i0=b();
   Z iEx=i0-1;
   Z res{137};
   if (n==0){
      res = iEx;
   }
   else{
      if (ordered){
         auto q=std::lower_bound(p_->begin(),p_->end(),t);
         if (q==p_->end() || *q!=t){
            res = iEx;
         }
         else{
            res = i0+q-p_->begin();
         }
      }
      else{
         auto  q=std::find(p_->begin(),p_->end(),t);
         if (q == p_->end() || *q!=t){
            res = iEx;
         }
      }
   }
   return res;
}

// modified from function locate of Press et al.

template <class T>
Z Vuc<T>::locAsc(T const& t)const
{
   if (sz_==0){
      cpmerror("Vuc<T>::locAsc(T): array is void");
```

```
      return -137; // never happens
   }
   if (t<fir()) return b()-1;
   if (t>=last()) return e();
      // if sz_==1 we have fir()==last and then the two previous
      // conditions are an alternative: one of them holds and we
      // are ready. Thus now sz_>=2
   Z jl=0,ju=sz_;
   while (ju-jl>1){
      Z jm=(jl+ju)/2;
      if (t >= (*p_)[jm]) jl=jm; else ju=jm; // Press et al. have > here
   }
   return jl+b();
}

template <class T>
Vuc<T> Vuc<T>::app(Vuc<T> const& h)const
{
   vector<T> pf;
   vector<T> pi=*p_;
   vector<T> hp=*h.p_;
   pf.reserve(sz_+h.size());
   pf.insert(pf.end(),pi.begin(),pi.end());
   pf.insert(pf.end(),hp.begin(),hp.end());
   return Vuc<T>(iv_.b(),pf);
}

template <class T>
Vuc<T> Vuc<T>::app(T const& t)const
{
   vector<T> pf(*p_);
   pf.push_back(t);
   return Vuc<T>(iv_.b(),pf);
}

template <class T>
Vuc<T>& Vuc<T>::prepend_(T const& t)
{
   cow_();
   p_->push_back(t);
   std::rotate(p_.rbegin(), p_.rbegin() + 1, p_.rend());
   iv_.n_();
   sz_++;
   return *this;
}

template <class T>
Vuc<T> Vuc<T>::rev()const
{
   if (sz_<2){
```

```
      return *this;
      // nothing to do if we have no or one component
   }
   else{
      vector<T> p{*p_};
      for (Z i=0,j=sz_-1; i<sz_; ++i,--j) p[i]=(*p_)[j];
      return Vuc<T>(b(),p);
   }
}

template <class T>
Vuc<T> Vuc<T>::resize(Z newDim)const
{
   if (newDim<=0) return Vuc<T>();
   vector<T> p(newDim);
   for (Z i=0;i<newDim;++i){
      p[i]= i<sz_ ? (*p_)[i] : T();
   }
   return Vuc<T>(b(),p);
}

template <class T>
Vuc<T> Vuc<T>::eli(IvZ const& iv)const
{
   IvZ ie=iv&dom();
   if (ie.isVoid()) return *this ;
   else{
      Vuc<B> vb(dom());
      for (Z i=vb.b();i<=vb.e();++i){
         vb.cui(i)=!ie.hasElm(i);
      }
      return select(vb);
   }
}

template <class T>
Vuc<T> Vuc<T>::eli1(Vuc<T>& h, Z i)const
{
// Here we use natural counting of components so that
// the j-th component of h is h[j-1].
   Z nEli=h.dim();
   if (i<1) i=1;
   if (sz_==0){ // nothing eliminated, nothing left
      h=Vuc<T>(0);
      return Vuc<T>(0);
   }
   else if (i>sz_){ // nothing eliminated
      h=Vuc<T>(0);
      return *this;
   }
```

```
   else{ // now i>=1 and i<=sz_ and sz_>=1
      // if true, we have sz_>=1
   // the i-th component is the first to be eliminated
   // so we have i-1 components of *this which are on the left-hand
   // side of the elimination area. These have to shine up in
   // the result vector to be returned
      Z nRes1=i-1;
   // So the maximum number of
   // components of *this that run the risk to get eliminated is
   // sz_-nRes1
      Z nEliRisk=sz_-nRes1;
      if (nEli>nEliRisk) nEli=nEliRisk;
      Z nRes2=sz_-nRes1-nEli;
      Z nRes=nRes1+nRes2;
      T* pRes=new T[toN(nRes)];
      T* pEli=new T[toN(nEli)];
      T* itRes=pRes;
      T* itEli=pEli;
      T* itp=p_;
      Z j=nRes1;
      while (j--) *itRes++=*itp++;
      j=nEli;
      while (j--) *itEli++=*itp++;
      j=nRes2;
      while (j--) *itRes++=*itp++; // for j==0 nothing done
      h=Vuc<T>(nEli,pEli);
      return Vuc<T>(nRes,pRes);
   }
}

template <class T>
Vuc<T> Vuc<T>::ins(Z i, T const& t)const
{
   Word loc("Vuc<T> Vuc<T>::ins(Z i, T const& t)const");
   vector<T> p{*p_};
   Z iLocal=i-b();
   auto it=p.begin()+iLocal;
   p.insert(it,t);
   return Vuc(b(),p);
}

template <class T>
Vuc<T>& Vuc<T>::ins_(Z i, T const& t)
{
   cow_();
   Word loc("Vuc<T> Vuc<T>::ins(Z i, T const& t)const");
 //  vector<T> p{p_};
   Z iLocal=i-b();
   auto it=p_.begin()+iLocal;
   p_.insert(it,t);
```

```
   sz_++;
   iv_.n_();
   return *this;
}

template <class T>
Vuc<T> Vuc<T>::ins(Z ia, Vuc<T> const& h)const
{
   Z mL=3;
   Word loc("Vuc<T> Vuc<T>::ins(Z i, Vuc<T> const& h)const");
   CPM_MA
   vector<T> p{*p_};
   auto it=p.begin()+ia;
   p.insert(it,h.p_->begin(),h.p_->end());
   return Vuc(b(),p);
}

template <class T>
T Vuc<T>::in_(T const& t, bool reversed)
// safe logic by indexing, propably not utmost performance
{
   cow_();
   if (sz_==0) return T();
   Z i;
   T res;
   if (!reversed){
      res=(*p_)[sz_-1]; // last component of array
      for (i=sz_-1;i>0;i--){
         (*p_)[i]=(*p_)[i-1]; // causal processing: on the right-hand side we
         // evaluate only expressions which were not yet changed during
         // the loop
      }
      (*p_)[0]=t;
   }
   else{
      res=(*p_)[0]; // first component of array
      for (i=0;i<sz_-1;i++){
         (*p_)[i]=(*p_)[i+1]; // causal processing: on the right-hand side we
         // evaluate only expressions which were not yet changed during
         // the loop
      }
      (*p_)[sz_-1]=t;
   }
   return res;
}

// re-indexing

template <class T>
Vuc<T> Vuc<T>::compose(Vuc<Z> const& w)const
```

```
{
   Vuc<T> res(iv_);
   for (Z i=w.b();i<=w.e();i++){
     res[i]=operator()(w[i]);
   }
   return res;
}

template <class T>
Vuc<T> Vuc<T>::rot(Z s, Outside mode)const
{
   Vuc<T> res(iv_);
   for (Z i=b();i<=e();++i) res[i]=operator()(i-s,mode);
   return res;
}

template <class T>
X2< Vuc<T>, Vuc<Z> > Vuc<T>::condense(
   bool (*equiv)(const T&, const T&))const
// implementation based on function eclazz of the Numerical Receipes of
// Press et al.
{
   const Z mL=3;
   static Word loc("Vuc<T>::condense()");
   CPM_MA
   Z n=dim();
   if (n<1){ // addition 2002-02-23
      CPM_MZ
      return X2< Vuc<T>, Vuc<Z> >(Vuc<T>(0),Vuc<Z>(0));
   }
   Vuc<Z> res2(n);
   Z k,j;
   res2[1]=1;
   for (j=2;j<=n;j++) {
      res2[j]=j;
      for (k=1;k<=(j-1);k++) {
         res2[k]=res2[res2[k]];
         if ((*equiv)((*this)[j],(*this)[k])) res2[res2[res2[k]]]=j;
      }
   }
   for (j=1;j<=n;j++) res2[j]=res2[res2[j]];
   Z m=-1;
   Z rj;
   for (j=1;j<=n;j++){
      rj=res2[j];
      if (rj>m) m=rj;
   }
   Vuc<Z> aux(m,0); // unfortunately the NR algorithm does not guarantee
      // that there are no gaps between the valid values of rj. Therfore
      // we find out the valid ones by an additional loop
```

```
   Vuc<Z> found(m,0);
   for (j=1;j<=n;j++){
      rj=res2[j];
      if(found[rj]==0){
         found[rj]=1;
         aux[rj]=j;
      }
   }
   Z mAct=0;
   for (j=1;j<=m;j++) mAct+=found[j];
   Vuc<T> res1(mAct);
   Z jAct=1;
   for (j=1;j<=m;j++){ // notice that aux[j] was initialized as 0
      if (aux[j]>0) res1[jAct++]=(*this)[aux[j]];
   }
   CPM_MZ
   return X2< Vuc<T>,Vuc<Z> >(res1,res2);
}


template <class T>
void Vuc<T>::set_(T const& t)
{
   cow_();
   for (Z i=0;i<sz_;++i) (*p_)[i]=t;
}

template <class T>
Vuc<T> Vuc<T>::set(Z i, T const& t)const
{
   Z iC=i-1;
   // now iC is a 'C-pointer index'

   if (iC<0) return *this;   // nothing changed
   else if (iC<sz_){
      // vector can already hold the new value
      Vuc<T> res(*this);
      res[iC]=t;
      return res;
   }
   else{
      // now we have to return an enlarged vector
      Z j, sz2=iC+1;
      Vuc<T> res(sz2);
      res[iC]=t;
      T* it=res.p_;
      T* itp=p_;
      Z i=sz_;
      while (i--) *it++ = *itp++;
      return res;
```

```
   }
}

template <class T>
T Vuc<T>::fAcc1(T (*f)(T const&), void (*acc)(T&, T const&))const
{
   T res=T();
   for (Z i=0;i<sz_;i++) acc(res,f((*p_)[i]));
   return res;
}

template <class T>
T Vuc<T>::fAcc2(T (*f)(T const&, T const&), void (*acc)(T&, T const&),
   Vuc<T> const& h)const
{
   T res=T();
   for (Z i=0;i<sz_;i++){ T xi=(*p_)[i]; T hi=h.li(i); acc(res,f(xi,hi));}
   return res;
}

template <class T>
template <class Y>
Y Vuc<T>::fAcc3(Y (*f)(T const&, T const&), void (*acc)(Y&, Y const&),
      Vuc<T> const& h )const
{
   Y res=Y();
   for (Z i=0;i<sz_;i++) acc(res,f((*p_)[i],h[(*p_)[i]]));
   return res;
}

template <class T>
Vuc<T> Vuc<T>::fAcc4(F<T2<T>,T> const& f)const
{
   Z d=dim();
   Vuc< Vuc<T> > aux(d,Vuc<T>(d));
   for (Z i=0;i<d;++i){
      for (Z j=0;j<i;++j){
         T fij=f(T2<T>(li(i),li(j)));
         aux.li(i).li(j)=fij;
         aux.li(j).li(i)=-fij;
      }
   }
   T* p1=new T[toN(sz_)];
   for (Z i=0;i<sz_;i++){
      T res;
      for (Z j=0;j<sz_;j++) res+=aux.li(i).li(j);
      p1[i]=res;
   }
   return Vuc<T>(dom(),p1,"");
}
```

```
namespace{

//template <class T>
//T fFunc5(T2<T> const& xij, F<R,R> const& dPot)
//// This is intended to represent the force which particle i feels due to
//// particle j being present.
//{
//   T eij=xij[1]-xij[2]; // eij points from j to i. Thus a positive
//   // multiple of eij corresponds to a force on i which drives it away
//   // from particle j. This such is the case of a repulsive potential.
//   // This has dPot(r)/dr < 0 so that with the sign built into the formula
//   // for the return value we in fact have the case of the positive
//   // multiple we started with.
//   R r=eij.nor_();
//   return eij*(-dPot(r));
//}

//template <class T>
//T fFunc6(T const& xi, F<R,R> const& dPot)
//{
//   T ei=xi; // ei points from the origin to i. Same situation as in fFunc5.
//   R r=ei.nor_();
//   return ei*(-dPot(r));
//}

}

template <class T>
Vuc<T> Vuc<T>::fAcc5(F<R,R> const& dPot)const
{
#ifdef CPM_Fn
   F< T2<T>, T > f = F1< T2<T>, F<R,R>, T >(dPot)(fFunc5<T>);
#else
   F<T2<T>,T> f((std::bind(fFunc5<T>,_1,dPot)));
#endif

   Z d=dim(); // one could call fAcc4 here at the cost of an additional
      // function call. Here we ask for 'utmost efficiency'.
   Vuc< Vuc<T> > aux(d,Vuc<T>(d));
   for (Z i=0;i<d;++i){
      for (Z j=0;j<i;++j){
         T fij=f(T2<T>(li(i),li(j)));
         aux.li(i).li(j)=fij;
         aux.li(j).li(i)=-fij;
      }
   }
   vector<T> p1(toN(sz_));
   for (Z i=0;i<sz_;i++){
      T res;
```

```
      for (Z j=0;j<sz_;j++) res+=aux.li(i).li(j);
      p1[i]=res;
   }
   return Vuc<T>(dom(),p1);
}

template <class T>
Vuc<T> Vuc<T>::fAcc6(F<R,R> const& dPot)const
{
#ifdef CPM_Fn
   F<T,T> f = F1<T,F<R,R>,T>(dPot)(fFunc6<T>);
#else
   F<T,T> f(std::bind(fFunc6<T>,_1,dPot));
#endif
   vector<T> p1(toN(sz_));
   for (Z i=0;i<sz_;++i){
      p1[i]=f(li(i));
   }
   return Vuc<T>(dom(),p1);
}

template <class T>
Vuc<T> Vuc<T>::fa(T (*f)(T const&))const
{
   vector<T> p=*p_;
   for (N i=0;i<n_;++i) p[i]=f((*p_)[i]);
   return Vuc<T>(iv_,p);
}

//template <class T>
//Vuc<T> Vuc<T>::faa(T (*f)(T))const
//{
//   T* p1=new T[toN(sz_)];
//   T* it=p1;
//   T* itp=p_;
//   Z i=sz_;
//   while(i--) *it++ = f(*itp++);
//   return Vuc<T>(dom(),p1,"");
//}

template <class T>
Vuc<T> Vuc<T>::fsf(std::function<T(T)> f)const
{
   vector<T> p=*p_;
   for (N i=0;i<n_;++i) p[i]=f((*p_)[i]);
   return Vuc<T>(iv_,p);
}

template <class T>
Vuc<T> Vuc<T>::faa(T (*f)(T))const
```

```
{
   vector<T> p=*p_;
   for (N i=0;i<n_;++i) p[i]=f((*p_)[i]);
   return Vuc<T>(iv_,p);
}

template <class T>
Vuc<T> Vuc<T>::fb(T (*f)(T const&, T const&), T const& t)const
{
  vector<T> p=*p_;
   for (N i=0;i<n_;++i) p[i]=f((*p_)[i],t);
   return Vuc<T>(dom(),p);
}

template <class T>
template <class Y>
Vuc<T> Vuc<T>::fb2(T (*f)(T const&, Y const&), Y const& y)const
{
   vector<T> p=*p_;
   for (N i=0;i<n_;++i) p[i]=f((*p_)[i],y);
   return Vuc<T>(dom(),p);
}

template <class T>
template <class Y>
void Vuc<T>::fb2_(T (*f)(T const&, Y const&), Y const& y)
{
   cow_();
   for (N i=0;i<n_;++i) (*p_)[i]=f((*p_)[i],y);
}

template <class T>
void Vuc<T>::fc_(T (*f)(T const&, T const&), T const& t)
{
   cow_(); for (N i=0;i<n_;++i) (*p_)[i]=f((*p_)[i],t);
}

template <class T>
void Vuc<T>::fd(T (*f)(T const&, T const&), T& t)const
{

   for (N i=0;i<n_;++i) t=f((*p_)[i],t);
}

template <class T>
Vuc<T> Vuc<T>::fe(T (*f)(T const&, T const&), Vuc<T> const& h)const
{
   if (!sameDom(h)) throw Error("Vuc<T>::fe(): domain mismatch");
   vector<T> p=*p_;
   for (N i=0;i<n_;++i) p[i]=f((*p_)[i],(*h.p_)[i]);
```

```
   return Vuc<T>(dom(),p);
}

template <class T>
template <class Y>
Vuc<T> Vuc<T>::fet(T (*f)(T const&, Y const&), Vuc<Y> const& h)const
{
   if (dom()!=h.dom()) throw Error("Vuc<T>::fet(): domain mismatch");
   vector<T> p=p_;
   for (N i=0;i<n_;++i) p[i]=f((*p_)[i],h.pi(i));
   return Vuc<T>(dom(),p);
}

template <class T>
Vuc<T> Vuc<T>::fe2(T (*f)(T const&, T const&), Vuc<T> const& h)const
{
   IvZ iv1{iv_};
   IvZ iv2{h.iv_};
   IvZ iv = iv1|iv2;
   Vuc<T> res(iv);
   for (Z i=iv.b();i<=iv.e();++i){
      if (iv1.hasElm(i)&&iv2.hasElm(i)) res[i]=f((*this)[i],h[i]);
   }
   return res;
}

template <class T>
void Vuc<T>::ff_(T (*f)(T const&, T const&), Vuc<T> const& h, Z is)
{
   cow_();
   if (is<0) is=0;
   for (Z i=0; i+is<sz_; ++i){
      if (h.valInd(i)) (*p_)[i+is]=f((*p_)[i+is],(*h.p_)[i]);
   }
}

template <class T>
void Vuc<T>::ffl_(T (*f)(T const&, T const&), Vuc<T> const& h)
{
   cow_();
   for (N i=0; i<n_; ++i){
      (*p_)[i]=f((*p_)[i],(*h.p_)[i]);
   }
}

template <class T>
void Vuc<T>::fa_(F<T,T> const& f, IvZ iv)
{
   cow_();
   for (Z i=b();i<=e();++i){
```

```
      if (iv.hasElm(i)) (*this)[i]=f((*this)[i]);
   }
}

template <class T>
void Vuc<T>::fg_(T (*f)(T const&, T const&, T const&),
      Vuc<T> const& h, T const& t)
{
   cpmassert(sameDom(h),"dimension mismatch in Vuc::fg_(f,Vuc,T)");
   cow_();
   for (Z i=0;i<sz_;++i) (*p_)[i]=f((*p_)[i],h.li(i),t);
}

template <class T >
void Vuc<T>::fh_(T const& w_1, T const& w0, T const& w1, Outside mode)
{
   cow_();
   if (sz_<2) return;
   T v_1=read(b()-1,mode);
   T v0=(*p_)[0];
   T v1=(*p_)[1];
   Z k=0;
   while (k<sz_){
      (*p_)[k]=v_1*w_1+v0*w0+v1*w1;
      k++;
      v_1=v0;
      v0=v1;
      v1=(k < sz_-1 ? (*p_)[k+1] : read(e()+1,mode));
   }
}

template <class T >
template <class Y>
void Vuc<T>::fht_(Y const& w_1, Y const& w0, Y const& w1, Outside mode)
{
   cow_();
   if (sz_<2) return;
   T v_1=read(b()-1,mode);
   T v0=(*p_)[0];
   T v1=(*p_)[1];
   Z k=0;
   while (k<sz_){
      (*p_)[k]=v_1*w_1+v0*w0+v1*w1;
      k++;
      v_1=v0;
      v0=v1;
      v1=(k < sz_-1 ? (*p_)[k+1] : read(e()+1,mode));
   }
}
```

```
template <class T >
template <class Y>
Vuc<T> Vuc<T>::fh(T (*f)(T const&, T const&, T const&, Y const&, Z),
      Y const& y, Outside mode)const
{
   N n=nc();
   if (n<2) return *this;
   Vuc<T> q=*this;
   Vuc<T> p=*this;
   for (N i=0; i<n;++i){
      p.pi(i)=f(q.pr(i-1,mode),q.pr(i,mode),q.pr(i+1,mode),y,i);
   }
   return p;
}

template <class T >
template <class Y>
Vuc<Y> Vuc<T>::fi(Y (*f)(T const&, T const&), void (*acc)(Y&, Y const&))const
{
   Z i,j;
   Y* q=new Y[toN(sz_)];
   for (i=0;i<sz_;i++){
      Y yi=Y();
      for (j=0;j<sz_;j++){
         acc(yi,f((*p_)[i],(*p_)[j]));
      }
      q[i]=yi;
   }
   return Vuc<Y>(dom(),q);
}

template <class T >
template <class Y>
Vuc<Y> Vuc<T>::fj(Y (*f)(T const&, T const&), void (*acc)(Y&, Y const&))const
{
   Z i,j;
   Y* q=new Y[toN(sz_)];
   for (i=0;i<sz_;i++) q[i]=Y();
   for (i=0;i<sz_;i++){
      for (j=0;j<i;j++){
         Y fij=f((*p_)[i],(*p_)[j]);
         acc(q[i],fij);
         acc(q[j],-fij);
      }
   }
   return Vuc<Y>(dom(),q);
}

template <class T >
Vuc<T> Vuc<T>::select(Vuc<B> const& s)const
```

```
{
   vector<T> pRes;
   for (Z i=b();i<=e();++i){
      bool keep{true};
      if (s.valInd(i)&&s[i]==false) keep=false;
      if (keep) pRes.push_back((*this)[i]);
   }
   return Vuc<T>(b(),pRes);
}

template <class T >
Vuc<IvZ> Vuc<T>::valOn(F<T,bool> const& f)const
{
   Z mL=3;
   static Word loc("Vuc<T>::valOn(F<T,bool>)");
   CPM_MA
   Vuc<IvZ> res;
   bool yetFoundTrue=false;
   Z firstTrue=0, firstFalseAfterTrue=0;
   for (Z i=b();i<=e();++i){
      bool val=f((*this)[i]);
      if (val){ // we found true
         if (!yetFoundTrue){ // then start a interval of validity
            yetFoundTrue=true;
            firstTrue=i;
         }
         else{ // normally nothing to do
            // but if we are at the end of the array the
            // last pending truth interval has to be
            // considered as finished and has to be added
            if (i==e()){
               firstFalseAfterTrue=n();
               IvZ ivAct(firstTrue,firstFalseAfterTrue-1);
               res&=ivAct; // appending
               // nothing else to do since we are finished
               CPM_MZ
               return res;
            }
         }
      }
      else{ // we found false
         if (yetFoundTrue){ // then i is the terminator
            // of a truth interval
            firstFalseAfterTrue=i;
            IvZ ivAct(firstTrue,firstFalseAfterTrue-1);
            res&=ivAct;
            yetFoundTrue=false;
         }
      }
   }
```

```
   CPM_MZ
   return res;
}

template <class T>
bool Vuc<T>::prnOn(ostream& str)const
{
   Z mL=3;
   Word loc=nameOf()&"::prnOn(...)";
   CPM_MA
   cpmwt((nameOf()&" begin").str());
   Root<IvZ>(iv_).prnOn(str);
   if (iv_.car()==0){
      Word w(" no components exist");
      cpmcerr<<w<<endl;
      cpmcerr<<" iv_.b() = "<<iv_.b()<<endl;
      goto label;
   }
   for (Z i=b();i<=e();i++){
      if (CpmRoot::wrtTit){
         Word wi("// i="); // added 2012-01-19
         // same as in S<>
         wi&=cpm(i);
         bool bi=wi.prnOn(str);
         cpmassert(bi==true,loc);
      }
      if (!Root<T>((*this)[i]).prnOn(str)){
         cpmwarning("failed to write component indexed "&cpm(i));
         cpmwarning("index range is from "&cpm(b())&" to "&cpm(e()));
         CPM_MZ
         return false;
      }
   }
label:
   cpmwt((nameOf()&" end").str());
      // for large sz_ it would be difficult to find the
      // end of the vector data if these are written to a file
      // and a human reader wants to inspect them
   CPM_MZ
   return true;
}

template <class T>
bool Vuc<T>::scanFrom(istream& str)
{
   Z mL=3;
   Word loc=nameOf()&"::scanFrom(...)";
   CPM_MA
   cow_();
   Root<IvZ> ivIn;
```

```
   bool suc = ivIn.scanFrom(str);
   if (!suc){
      cpmwarning(loc&": can't read IvZ");
      CPM_MZ
      return false;
   }
   IvZ iv=ivIn();
   Z n=iv.car();
   cpmmessage(mL,"dimension read as "&cpm(n));
   if (n>dimMax) cpmwarning(loc&": n>dimMax");
   Vuc<T> res(iv);
   Root<T> riIn;
   for (Z i=res.b();i<=res.e();i++){
      suc = riIn.scanFrom(str);
      if (!suc){
         Word mes="failed to read component indexed "&cpm(i)&
             " index range is from "&cpm(res.b())&" to "&cpm(res.e());
         cpmwarning(mes);
         CPM_MZ
         return false;
      }
      res.cui(i) = riIn();
   }
   *this=res;
   CPM_MZ
   return true;
}

template <class T >
Z Vuc<T>::com(Vuc<T> const& s)const
// short vectors < longer vectors
{
   Z d1=dim(), d2=s.dim();
   if (d1<d2) return 1;
   else if (d1>d2) return -1;
   else{
       for (Z i=0;i<d1;i++){
         Z ci=Root<T>((*p_)[i]).com(s.li(i));
         if (ci!=0) return ci;
      }
      return 0;
   }
}
/************************* Vuc<bool> *********************************/
// since std::vector<bool> has a quite irregular special definition we get
// a lot of compilation errors if we use existing C+- code the
// instantiation Vuc<bool> while Vuc<B> works fine.
/*
template<>
class Vuc<bool>{
```

```
   IvZ iv_;
   std::vector<CpmRoot::B> p_;
public:
   B const& cui(Z i)const{return (*p_)[iv_.ri(i)];}
   B& cui(Z i){return (*p_)[iv_.ri(i)];}
   B const& operator[](Z i)const{return (*p_)[iv_.ri(i)];};
   B& operator[](Z i){return (*p_)[iv_.ri(i)];};
};
*/

/*
B const& CpmArrays::Vuc<bool>::operator[](Z i)const
{
   if (ranChcAlw){
      if (!iv_.hasElm(i)) cpmerror("index out of range");
   }
   return (*p_)[iv_.ri(i)];
}


B& CpmArrays::Vuc<bool>::operator[](Z i)
{
   if (ranChcAlw){
      if (!iv_.hasElm(i)) cpmerror("index out of range");
   }
   return (*p_)[iv_.ri(i)];
}
*/


/************************ iterated templates *************************/
// describing multi-indexed quantities
// Notice that these all have automatically the member functions of Vuc
// defined!

#define CPM_Vuc1 Vuc<T>
#define CPM_Vuc2 Vuc<Vuc<T> >
#define CPM_Vuc3 Vuc<Vuc<Vuc<T> > >
#define CPM_Vuc4 Vuc<Vuc<Vuc<Vuc<T> > > >

/************************** class VucVuc *********************************/
template <class T>
class VucVuc: public CPM_Vuc2{ // matrices

   typedef CPM_Vuc2 Base;

public:

// constructors
```

```
   VucVuc(Z d1, Z d2):CPM_Vuc2(d1,CPM_Vuc1(d2)){}
   VucVuc(Z d1, Z d2, T const& t):CPM_Vuc2(d1, CPM_Vuc1(d2,t)){}
      // all components are equal to t
   VucVuc(void):CPM_Vuc2(){}
   VucVuc(CPM_Vuc2 const& x):CPM_Vuc2(x){}

   Vuc<T> lin()const;
      // 'linear version of matrix'
      // by appending all rows

// dimension access
   Z dim1()const{ return Base::dim();}
   Z dim2()const{ return (*this)[1].dim();}

   virtual Z size()const // virtual in base Vuc<...>
      { return dim1()==0 ? 0 : dim1()*dim2();}

// component access
   const T& operator()(Z i, Z j)const
      // returns T() for out of range indexes
   { return Base::read(i).read(j);}

   T& operator()(Z i, Z j)
   { return (*this)[i][j];}

   VucVuc<T> trn()const
      //: transpose
      // Returns the transposed matrix
   {
      VucVuc<T> res(dim2(),dim1());
      for (Z i=1;i<=dim1();i++){
         for (Z j=1;j<=dim2();++j){
            res[j][i] = (*this)[i][j];
         }
      }
      return res;
   }

   template <class Y>
   Vuc<Y> each(void (*f)(T const&, Y&))const;
      //: each
      // collecting the application of f on every pixel in a linear array

   template <class Y>
   VucVuc<Y> operator()(Y (*f)(T const&))const;
      //: operator()
      // creating a matrix with a transformed value range
};

template <class T>
```

```
template <class Y>
Vuc<Y> VucVuc<T>::each(void (*f)(T const&, Y&))const
{
   Z m1=dim1(),m2=dim2(),k=1,i1,i2;
   Vuc<Y> res(m1*m2);
   for (i1=1;i1<=m1;i1++){
      for (i2=1;i2<=m2;i2++){
         f((*this)[i1][i2],res[k++]);
      }
   }
   return res;
}

template <class T>
template <class Y>
VucVuc<Y> VucVuc<T>::operator()(Y (*f)(T const&))const
{
   Z m=dim1(),n=dim2(),i,j;
   VucVuc<Y> res(m,n);
   for (i=1;i<=m;i++){
      for (j=1;j<=n;j++){
         res[i][j]=f((*this)[i][j]);
      }
   }
   return res;
}

template <class T>
Vuc<T> VucVuc<T>::lin()const
{
   if (dim1()==0) return Vuc<T>(0);
   else{
      Vuc<T> res=(*this)[1];
      for (Z i=2;i<=dim1();i++) res&=(*this)[i];
      return res;
   }
}

/************************ class VucVucVuc***********************************/
// tensors of rank 3

template <class T>
class VucVucVuc: public CPM_Vuc3{ // tensors of rank 3

      typedef CPM_Vuc3 Base;

public:

// constructors
```

```
   VucVucVuc(Z d1, Z d2, Z d3, T  const& t):CPM_Vuc3(d1,CPM_Vuc2(d2,CPM_Vuc1(d3,t))){}
   VucVucVuc(Z d1, Z d2, Z d3):CPM_Vuc3(d1,CPM_Vuc2(d2,CPM_Vuc1(d3))){}
   VucVucVuc(void):CPM_Vuc3(){}
   VucVucVuc(CPM_Vuc3 const& x):CPM_Vuc3(x){}

// dimension access
   Z dim1()const{ return Base::dim();}
   Z dim2()const{ return (*this)[1].dim();}
   Z dim3()const{ return (*this)[1][1].dim();}

   virtual Z size()const; // virtual in base Vuc<...>

// component access

   const T & operator()(Z i, Z j, Z k)const
   // returns T() for out of range indexes
   { return Base::read(i).read(j).read(k);}

   T & operator()(Z i, Z j, Z k)
   { return (*this)[i][j][k];}
};

template <class T>
Z VucVucVuc<T>::size()const
{
   Z d1=dim1();
   if (d1==0) return d1;
   else{
      Z d2=dim2();
      if (d2==0) return d2;
      else{
         Z d3=dim3();
         if (d3==0) return d3;
         else return d1*d2*d3;
      }
   }
}

/********************** class VucVucVucVuc ***********************************/
template <class T>
class VucVucVucVuc: public CPM_Vuc4{ // tensors of rank 4

      typedef CPM_Vuc4 Base;

public:

// constructors

   VucVucVucVuc(Z d1, Z d2, Z d3, Z d4, T const& t):
   CPM_Vuc4(d1,CPM_Vuc3(d2,CPM_Vuc2(d3,CPM_Vuc1(d4,t)))){}
```

```
   VucVucVucVuc(Z d1, Z d2, Z d3, Z d4):
   CPM_Vuc4(d1,CPM_Vuc3(d2,CPM_Vuc2(d3,CPM_Vuc1(d4)))){}
   VucVucVucVuc(void):CPM_Vuc4(){}
   VucVucVucVuc(CPM_Vuc4 const& x):CPM_Vuc4(x){}

// dimension access
   Z dim1()const{ return Base::dim();}
   Z dim2()const{ return (*this)[1].dim();}
   Z dim3()const{ return (*this)[1][1].dim();}
   Z dim4()const{ return (*this)[1][1][1].dim();}

   virtual Z size()const; // virtual in base Vuc<...>

// component access

   T const& operator()(Z i, Z j, Z k, Z l)const
  // returns T() for out of range indexes
   { return Base::read(i).read(j).read(k).read(l);}

   T& operator()(Z i, Z j, Z k, Z l)
   { return (*this)[i][j][k][l];}
};

template <class T>
Z VucVucVucVuc<T>::size()const
{
   Z d1=dim1();
   if (d1==0) return d1;
   else{
      Z d2=dim2();
      if (d2==0) return d2;
      else{
         Z d3=dim3();
         if (d3==0) return d3;
         else{
            Z d4=dim4();
            if (d4==0) return d4;
            else return d1*d2*d3*d4;
         }
      }
   }
}
#undef CPM_Vuc4
#undef CPM_Vuc3
#undef CPM_Vuc2
#undef CPM_Vuc1

}  // namespace CpmArrays

//namespace CpmRoot{
```

```
//// This is a pattern for partial specializations --- to which
//// the specializations to class templates belong --- of the
//// IO class template started in cpmnumbers.h and the
//// Name class template started in cpmword.h.
//// A corresponding definition is needed for all non-C+- classes which
//// shall be used as template arguments of C+- containers and
//// are expected to provide then the same functionality as
//// corresponding C+- classes.
//
//   template<class T>
//   class IO< std::vector<T> >{ // partial specialization
//    public:
//      IO(){}
//      bool o(std::vector<T> const& v, ostream& str)const
//      { return CpmArrays::Vuc<T>(v).prnOn(str);}
//      bool i(std::vector<T>& v, istream& str)const
//      {
//         CpmArrays::Vuc<T> vl;
//         bool res=vl.scanFrom(str);
//         v=vl.std();
//         return res;
//      }
//   };
//
//#if defined(CPM_NAMEOF)
//   template<class T>
//   class Name< std::vector<T> >{ // partial specialization
//   public:
//      Name(){}
//      Word operator()(std::vector<T> const& vt )const
//      { return Word("std::vector<"&CpmRoot::Name<T>()(T())&">");}
//   };
//   template<class T>
//   class Name< std::set<T> >{ // partial specialization
//   public:
//      Name(){}
//      Word operator()(std::set<T> const& vt )const
//      { return Word("std::set<"&CpmRoot::Name<T>()(T())&">");}
//   };
//#endif
//}
#endif
```

52 cpmword.h

```
//? cpmword.h
//? Status of work 2023-10-20.
//?
//?  ...


#ifndef CPM_WORD_H_
#define CPM_WORD_H_
/*
   Description: Declares class Word which is now essentially identical
      to string of the standard library. It is formed from it according
      to the advice in BS3, end of Chapter 20.3.

      Word is a minimal string class to which Cpm classes refer for
         type identification and for messaging
*/
#include <cpminterfaces.h>
#include <stdlib.h>
#if !defined(CPM_NAMEOF)
   #include <typeinfo>
#endif

namespace CpmRoot{

   using namespace CpmStd;

   class Word;

   Word toWord(R x, Z prc);

   Word toWord(unsigned int i, const char* ff);
      // ff standard format string
   Word toWord(unsigned long int i, const char* ff);
   Word toWord(int i, const char* ff);
   Word toWord(long int i, const char* ff);
   Word toWord(R r, const char* ff);

   ofstream& to_ofstream(Word const& w);
   ifstream& to_ifstream(Word const& w);
   string toString(Word const& w);

// concatenation of words described by operator &
   Word operator &(Word const&, Word const&);
   Word operator &(const char* cp, Word const& w);
   Word operator &(Word const& w, const char* cp);
   Word operator &(char cp[], Word const& w);
   Word operator &(Word const& w, char cp[]);
```

```
// for convenience same is allowed to be denoted +
   Word operator +(Word const&, Word const&);
   Word operator +(const char* cp, Word const& w);
   Word operator +(Word const& w, const char* cp);
   Word operator +(char cp[], Word const& w);
   Word operator +(Word const& w, char cp[]);

////////////////////// class Word ////////////////////////////////////

   class Word{ //  wrapping charcter strings, rich functionaliy
      // Instances of Word are dynamically allocated strings of
      // characters like those of string (on which class the present
      // implementation is based). Input and Output works as expected only
      // if we have a word in the narrower sense that the character
      // string does not contain 'whitespace'.
      // Due to functions toBool(), toZ(), toR() a Word is a universal
      // carrier of numerical information. For instance the content
      // of a command line may be represented as a list of Words (an
      // instance of V<Word> in C+- )

   string rep_;

   static bool scnLin;
      // modifies the function scanFrom() in a way that also character
      // strings containing blanks can be read in to yield a single Word.
      // Such a facility is needed if Word is to be used as the data type
      // of names of more complex entities like organizations, theories,
      // etc.: Certainly 'set theory' and 'Eastman Kodak Company' should
      // be allowed names.

   static Z endOfWord; // end of Word
      // 0 nothing, 1 blank, else newline after w.prnOn()

   typedef Word Type;
   typedef Word CloneBase;

public:

   static void setScanLine(bool sl){ scnLin=sl;}
      //: set scan line
      // see private static data member scnLin
   static bool getScanLine(){ return scnLin;}
      //: get scan line
      // see private static data member scnLin

   static void setEndOfWord(Z i){ endOfWord=i;}
      //: set end of word
      // see private static data member endOfWord
   static Z getEndOfWord(){ return endOfWord;}
```

```
      //: get end of word
      // see private static data member endOfWord

   Word()=default;

   explicit Word(char c):rep_(1,c){}
   // a Word consisting of just one character c

   explicit Word(Z i, char c='a'):rep_(i,c){}
   // a Word consisting of i copies of character c
   // useful for getting new words by changing letters at given positions

   Word(const char* charPtr):rep_(charPtr){}
      // construction of a Word from char*, and thus from literals
      // such as Word w("convenient programming")
      // Although this defines a conversion from const char* to
      // Word, in a statement  cout<<"This is ..."<<endl;  the
      // interpretation of "This is ..." as const char* applies,
      // since exact matches are preferred over matches which employ
      // user-defined conversions. This is important, since
      // cout<<Word("This is ...")<<endl; may append some separator
      // to the input string depending on the value of the
      // static data member endOfWord.

   explicit Word(string const& s):rep_(s){}
      // construction of a Word from string

   explicit Word(ostringstream const& ost):rep_(ost.str()){}
      // construction of a Word from a string stream
      // This is the most convenient way to create detailed texts held as
      // a Word, as in the following code fragment:
      // ostringstream ost;
      // ost<<...<<...endl<<...;
      // Word text(ost);

   Word(Word const&)=default;

//   Word():rep_(){}
      // default constructor, creates the void word which consists of
      // 0 characters

   const char* operator-(void)const{ return rep_.c_str();}
      // explicit conversion from Word to const char*
      // The result is owned by the Word and the user should not try to
      // delete it.

   const char* c_str(void)const{ return rep_.c_str();}
      // to make Word and string as much alike as possible

   string str()const{ return rep_;}
```

```
      //: string

   string toStr()const{ return rep_;}
      //: to string

   string std()const{ return rep_;}
      //. standard

   Word skipChr(char c)const;
      //: skip character
      // Returns a Word which results from *this by eliminating c
      // everywhere.

   void skipSpc_();
      //: skip space, mutating function
      // eliminates all characters which by the function isspace(char)
      // of <cctype> is classified as space

   void skipSurSpc_();
      //: skip surrounding space
      // eliminates all space characters (whitespace) as above which
      // surround a chain of space separated traditional words.
      // So the result is a robust representation for a textual content
      // of a line of text, since unvisible parts at the beginning and
      // the end get eliminated
      // added 2004-10-12

   Word firstWord()const;
      // returns the first whitespace separated part of the string rep_
      // (which may contain whitespace). It would be nice to
      // return a complete list of Words here, but we prefer not to
      // build on any array class in the present basic class.
      // The result may be the void Word.

   Word firstWord_();
      // returns the first whitespace separated part of the string rep_
      // (which may contain whitespace) and changes *this into
      // the part of the word which remains.
      // The ending '_' is reminescent of Ruby's '!' as a indicator
      // for mutating functions. Calling this function in succession
      // allows to create a complete list of words in a line without
      // having a list- or array-class available at this point.

// we have to define later a general template function
// template <class T1, class T2>
// operator & (const T1&, const T2&)
// This implies that in (non-template !) functions operator & only exact
// matches will be accepted (see Stroustrup p. 589), certainly not the
// ones that can be achieved by userdefined conversions. So, we have to
// take care of char* and char[] in this place
```

```
// concatenation of words described by operator &
   Word& operator &=(Word const& w);
   Word& operator &=(const char* cp);
   friend Word operator &(Word const&, Word const&);
   friend Word operator &(const char* cp, Word const& w);
   friend Word operator &(Word const& w, const char* cp);
   friend Word operator &(char cp[], Word const& w);
   friend Word operator &(Word const& w, char cp[]);

// for convenience same is allowed to be denoted +. This is convenient
// since + binds stronger than & and especially stronger than <<
   Word& operator +=(Word const& w);
   Word& operator +=(const char* cp);
   friend Word operator +(Word const&, Word const&);
   friend Word operator +(const char* cp, Word const& w);
   friend Word operator +(Word const& w, const char* cp);
   friend Word operator +(char cp[], Word const& w);
   friend Word operator +(Word const& w, char cp[]);

// access functions
   Z dim(void)const{ return rep_.empty() ? 0_Z : Z(rep_.length());}
      //: dimension
      // is 0 for the void Word
   Z b(void)const{ return 1;}
      // b for begin, first index of a letter
   Z e(void)const{ return dim();}
      // e for end, last index of a letter
      // A loop over the letters of Word w takes the form
      // for (Z i=w.b();i<=w.e();++i){ ... = w[i];}
      // This is correct also for the void word.
      // This looping style is also valid for all C+- array classes.
   Z size(void)const{ return dim();}
      //: size
   Z length(void)const{ return dim();}
      // for uniformity with std library types

   bool isVoid()const { return dim()==0;}
      //: is void
      //  short answer on whether the dimension is zero

   friend Z length(Word const& w){ return w.dim();}
      // number of characters (no addition for a terminator)

   char operator[](Z i)const;
      // save reading of characters. Valid indexes are 1...dim()

   char& operator[](Z i);
      // save overwriting of characters in a word.
```

```
   bool valInd(Z i)const{ return i>0 && i<=dim();}
      //: valid index
      // returns the validity of i as an index of *this

   Word cut(Z i)const;
      //: cut
      // returned is a Word which results from *this by removing i
      // components from the end. If i<0 we remove -i components from
      // the beginning.

// functions dealing Words as filenames
   Word resize(Z newDim)const;
      // returns a Word res such that res.dim()=newDim. If newDim is
      // smaller than dim(), the end of *this will be cut away.
      // If newDim is larger, blanks will be added.

   Word remExt(bool extended=false)const;
      //: remove extension
      // Intention of the function: returns what is left if any
      // file name extension (if there is one) is removed from
      // *this. So for *this=="myprog.ini" we return "myprog".
      // Actually we read and copy *this from the beginning until we
      // find a character which signifies the beginning of the extension.
      // In nomal mode this is a dot, in extended mode this is any capital
      // letter. The extended mode arrose from the need to distinguish
      // executables made by different compilers by a suffix such as
      // palaMS.exe, palaMinGW.exe, pala_ms.exe. Since the application
      // framework in cpmapplication.cpp deduces the program name,
      // and thus the name of the ini-file to search, from
      // the name of the executable, the above-mentioned executables would
      // try to read ini-files palaMS.ini, palaMinGW.ini, and
      // pala_ms.ini. They should, however, read simply pala.ini.
      // This is achieved by removing the extension from the executables
      // name for a true value of the argument.
      // The presently implemented notion of extension in 'extended mode'
      // is, that it is everything what is not a lower-case letter or a
      // digit. This reflects .... obsolete

   Word remApp()const;
      //: remove appendix
      // Here, we remove the maximum contiguous string formed out of
      // capital letters and underscores at the end of *this. So name
      // appendices reflecting compilers or compiler options should be
      // chosen to consist of capital letters and/or underscores. So
      // not all of the functions in the previous function comment work:
      // Instead of palaMinGW, pala_ms one should use palaMINGW and
      // pala_MS

   Word appBefExt(Word const& app)const;
      //: append before extension
```

```
      // Word("myprog.ini").appBefExt("2") yields "myprog2.ini"

   Word baseName()const;
      //: base name
      // Only the characters right to the right-most "\" or "/"
      // are returned as a Word. If *this is a file name including
      // a directory path, then the actual file name gets extracted and
      // returned. (corresponds do File:basename of Ruby)

   bool readable()const;
      // returns true if *this is the name of a file that can be
      // opened and gives a valid istream to read from.
      // The stream will be closed after the test and not returned.

   Word makeFileName()const;
      //: make file name
      // returns a version of *this in which all characters which may not
      // be acceptable in file names are replaced by "_"
// end of functions dealing Words as filenames

   Word rev()const;
      //: reverse
      // returns the reversed version of *this, i.e. the one read from
      // the end

   Word& rev_(){ return *this=rev();}
      //: reverse
      // mutating form of reversal

   Word slc_(char c, bool firstSep=true);
      //: slice
      // If c does not occur in *this, we return the original *this and
      // change *this to Word("").
      // One could do it the other way round, then the implementation
      // of remExt in terms of the present function would need to handle
      // the case that no dot is present (as for executables in Linux)
      // would need special treatment.
      // Now the interesting case:
      // If c appears in *this, we fix the position where it
      // appears in the leftmost position for firstSep==true
      // and in the rightmost position else.
      // we return the part of the word which is left of c
      // and change *this to the part right of c. Then we have for
      // Word w=...;
      // Word wMem=w;
      // char c=...; // such that c occurs in w !
      // Word wl=w.slc_(c);
      // the property
      // wl&Word(c)&w==wMem;
```

```
   Word tail(Z i)const;
      //: tail
      // if i>=dim() we return *this. Else we remove as many components
      // from the beginning that only i remain (' the last i components of
      // the original Word') and return the result.

   Z find(char c)const;
      //: find
      // returns the first position (1.....dim()) of a c in the
      // Word *this. If the character c isn't there we get 0

    Z find(Word const& w)const;
      //: find
      // returns the first position (1.....dim()) of a
      // subword w in the Word *this. If the Word w isn't there
      // we get 0

   static Word write(R n, Z fieldLength=0_Z);
   static Word write(Z n, Z fieldLength=0_Z);
   static Word write(N n, Z fieldLength=0_Z);
        // If fieldLength=0 the natural length of the number is fully
      // outputted. Else the length is casted or extended to fieldLength.
   static Word write(string h);
   static Word write(bool b);

   static Word ascii();
      // All ASCII characters implemented in in the fonts
      // used by CpmGraphics::Frame. These characters are concatenated
      // in a Word. Helpful for font inspection and development.

   R toR()const;
      // outputs a R obtained by converting *this
   Z toZ()const;
      // outputs a Z obtained by converting *this
   N toN()const;
      // outputs a N obtained by converting *this
   bool toBool()const;
      // outputs a bool obtained by converting *this

// these declarations are repeated outside the class
// and have only influence on the implementation

   friend Word toWord(unsigned int i, const char* ff);
   friend Word toWord(unsigned long int i, const char* ff);
   friend Word toWord(int i, const char* ff);
   friend Word toWord(long int i, const char* ff);
   friend Word toWord(R r, const char* ff);

   friend ofstream& to_ofstream(Word const& w);
   friend ifstream& to_ifstream(Word const& w);
```

```
   friend string toString(Word const& w);

   bool readLine(istream& in){return CpmRoot::readLine(rep_,in);}
      // making *this equal to the content of the next non-comment
      // line in stream in

   CPM_IO
      // scanFrom(in);  makes *this equal to the next noncomment
      // whitespace separated sequence of characters in stream in.
      // Thus the code
      //
      //    Word w("This is a word");
      //    w.prnOn(aStream);
      //    Word w2;
      //    w2.scanFrom(aStream)
      //
      // will yield w2=="This"
      // and not w2=="This is a word"! This is sometimes what one wants,
      // sometimes not.
   CPM_ORDER
   CPM_TEST_ALL // implementation borrowed from string
// Declarations of CPM_DESCRIPTORS together with implementation
   Word toWord()const{ return *this;}
   Word nameOf()const{ return "Word";}
};

inline string toString(Word const& w){ return w.rep_;}

#define cpmwrite CpmRoot::Word::write
#define cpm CpmRoot::Word::write

// conversion of selected basic types to string
// as a basis for doing the conversion to Word in a consistent
// manner.
// The parameter prc (precision) makes no sense for integer types
// Most conversions from built-in types to Word, which have not
// a char*-typed formatting string as one argument, are based on these
// toStr-functions. An exception are the functions write which have
// precisely defined writing field length which is needed for writing
// e.g. to tables or quietly on the status bar.
// prc<=0 means that no precision gets set in the code, so that we then
// rely on the default precision set by the compiler.
   string toStr(Z x);
   string toStr(N x);
   string toStr(L x);
   string toStr(bool x);
   string toStr(R const& x, Z prc=0_Z);
/*
In cpmnumbers.h the following list of interface service class
templates was defined
```

```
    IO<T>, Comp<T>, Inv<T>, AbsSqr<T>, Abs<T>, Dis<T>,
    Test<T>, Ran<T>, Hash<T>, Conj<T>, Neutrals<T>
and the following two completing members were annonced
    Name<T>, ToWord<T>.
These will be defined now:
*/
////////////////// class Name<> ////////////////////////////////////////
// We need a template class istead of a template function since we need
// specializations also for templates such as std::vector<> (see cpmv.h).
// Whether one has CPM_NAMEOF defined or not --- the C+- basic
// built-in types always have the C+- name (i.e 'R', not 'double')
// For the more complex types, the names created by typeinfo
// (which look acceptable) are used for CPM_NAMEOF not defined.

   template<class T>
   class Name{ // tool class template for defining the nameOf function
   public:
      Name(){}
      Word operator()(T const& t)const
      {
#if defined(CPM_NAMEOF)
         return t.nameOf();
#else
         return Word(typeid(t).name());
#endif
      }
   };

   template<>
   class Name<Z>{ // Z: integers
   public:
      Name(){}
      Word operator()(Z const& t)const
         { t; return Word("Z");}
   };

   template<>
   class Name<N>{ // N: natural numbers
   public:
      Name(){}
      Word operator()(N const& t)const
         { t; return Word("N");}
   };

   template<>
   class Name<R>{ // multiple precision real numbers
   public:
      Name(){}
      Word operator()(R const& t)const
         { t; return Word("R");}
```

```
   };

   template<>
   class Name<L>{ // L: letter
   public:
      Name(){}
      Word operator()(L const& t)const
         { t; return Word("L");}
   };

   template<>
   class Name<bool>{ // specialization
   public:
      Name(){}
      Word operator()(bool const& t)const
         { t; return Word("bool");}
   };

   template<>
   class Name<char>{ // specialization
   public:
      Name(){}
      Word operator()(char const& t)const
         { t; return Word("char");}
   };

   template<>
   class Name<string>{ // specialization
   public:
      Name(){}
      Word operator()(string const& t)const
         { t; return Word("string");}
   };

   template<class T>
   Word nameOf(T const& t){ return Name<T>()(t); }
      //: name of
   // convenient syntax for getting a name of the type of
   // a quantity. Definition is such that one may also give
   // Cpm-names to class templates. See cpmv.h
   // for treating template <class T> std::vector<T> in this way.

////////////////////// class ToWord<> ////////////////////////////////

   template<class T>
   class ToWord{
   public:
      ToWord(){}
      Word operator()(T const& t)const{ return t.toWord();}
   };
```

```
template<>
class ToWord<R>{
public:
   ToWord(){}
   Word operator()(R const& t)const{ return Word(toStr(t));}
};

template<>
class ToWord<Z>{
public:
   ToWord(){}
   Word operator()(Z const& t)const{ return Word(toStr(t));}
};

template<>
class ToWord<N>{
public:
   ToWord(){}
   Word operator()(N const& t)const{ return Word(toStr(t));}
};

template<>
class ToWord<L>{
public:
   ToWord(){}
   Word operator()(L const& t)const{ return Word(toStr(t));}
};

template<>
class ToWord<bool>{
public:
   ToWord(){}
   Word operator()(bool const& t)const{ return Word(toStr(t));}
};

template<>
class ToWord<string>{
public:
   ToWord(){}
   Word operator()(string const& t)const{ return Word(t);}
};

template<class T>
Word toWord(T const& t){ return ToWord<T>()(t); }
   //: to word
   // Presently not important, MPI-functionalty relies on
   // prnOn and scanFrom

////////////////////////////// class Root<> ////////////////////////////
```

```
   // A final step towards unification of infrastructure functions:
   // The Root class template provides member functions which the
   // argument type T may not define. Even if type T is only accessible
   // through its declaration, we may specialize the 13 basic
   // Root - related interface service class templates
   //
   //    IO<T>, Comp<T>, Inv<T>, AbsSqr<T>, Abs<T>, Dis<T>, Test<T>,
   //    Ran<T>, Hash<T>, Conj<T>, Neutrals<T>, ToWord<T>, Name<T>
   //
   // for this type T and thus make Root<T> equipped with these member
   // functions.
   // How this class simplifies the implementation of template classes
   // can be seen from the implementation of the interfaces
   // CPM_IO and CPM_ORDER in cpmv.h
   // Notice that in addition to the template argument type T, there are
   // the following types involved in the definition of Root<>:
   //
   // bool
   // std::istream
   // std::ostream
   // CpmRoot::Z
   // CpmRoot::R
   // CpmRoot::Word
   //
   // Notice also, that the true nature of CpmRoot::Z and CpmRoot::R
   // depends on the macros CPM_LONG and CPM_MP in cpmdefinitions.h.
   // This is an overview of all 26 member functions of Root<T>:
/*
      Root(T const& t = T());
      T operator()(void)const;
      bool prnOn(ostream& str)const;
      bool scanFrom(istream& str);
      Z com(T const& t)const;
      bool operator == ( T const& t)const;
      bool operator != (T const& t)const;
      bool operator < (T const& t)const;
      bool operator > (T const& t)const;
      bool operator <= (T const& t)const;
      bool operator >= (T const& t)const;
      T inf(T const& t)const;
      T sup(T const& t)const;
      Z hash()const;
      R absSqr()const;
      R abs()const;
      R dis(T const& t)const;
      T inv()const;
      T operator!()const;
      T net(Z i=0)const;
      T con()const;
```

```
      T operator~()const;
      T test(Z tvs)const;
      T ran(Z j=0)const;
      Word nameOf()const
      Word toWord()const
*/

   template<class T>
   class Root{ // Provides the basic functions for dealing with CpmRoot's
     // basic types as member functions (or operators) of a class
     // template.
      T a_;
   public:
      explicit Root(T const& t = T()):a_(t){}
      //Root(T& t , Word const& ):a_(t){}
         // should allow to change t e.g. in
         // Root<T>(t,"").scanFrom(in);
         // does not work however!
      T operator()(void)const{ return a_;}
      bool prnOn(ostream& str)const{ return IO<T>().o(a_,str);}
         //: print on
         // we don't redefine ostream& << T
         // Root<T>(t).prnOn(out);
         // is the general idiom for writing a T-typed object to stream.
         // Notice that the I/O-macros cpmp(X) and cpms(X) are implemented
         // based on functions prnOnT and scanFromT and that class Root
         // can hardly be used for this purpose (macros are always
         // special).
      bool scanFrom(istream& str){ return IO<T>().i(a_,str);}
         //: scan from
         // we don't redefine istream& >> T
         // The general idiom for reading a T-typed object t from
         // stream is:
         // Root<T> tIn;
         // bool suc=tIn.scanFrom(str);
         // t=tIn();
         // A more compact way (avoiding Root) is
         // bool suc=scanT(t,str);
         // or
         // bool CpmRoot::scanFrom(t,str);

      Z com(T const& t)const{ return Comp<T>()(a_,t);}
         //: compare
      bool operator == ( T const& t)const{ return 0==com(t);}
      bool operator != (T const& t)const{ return 0!=com(t);}
      bool operator < (T const& t)const{ return 0 < com(t);}
      bool operator > (T const& t)const{ return 0 > com(t);}
      bool operator <= (T const& t)const{ return 0 <= com(t);}
      bool operator >= (T const& t)const{ return 0 >= com(t);}
      T inf(T const& t)const{ return 0 < com(t) ? a_ : t;}
```

```
      //: infimum
   T sup(T const& t)const{ return 0 > com(t) ? a_ : t;}
      //: supremum
   Z hash()const{ return Hash<T>()(a_);}
      //: hash
   R absSqr()const{ return AbsSqr<T>()(a_);}
      //: absolute (value) squared
   R abs()const{ return Abs<T>()(a_);}
      //: absolute (value)
   R dis(T const& t)const{ return Dis<T>()(a_,t);}
      //: distance
   T inv()const{return Inv<T>()(a_);}
      //: inverse
   T operator!()const{return Inv<T>()(a_);}
   T net(Z i=0)const{ return Neutrals<T>()(a_,i);}
      //: neutrals
   T con()const{ return Conj<T>()(a_);}
      //: conjugate
   T operator~()const{ return Conj<T>()(a_);}
   T test(Z tvs)const{ return Test<T>()(a_,tvs);}
      //: test, //. tvs: test value size ~ complexity
   T ran(Z j=0)const{ return Ran<T>()(a_,j);}
      //: random
   Word nameOf()const{ return Name<T>()(a_);}
      //: name of
   Word toWord()const{ return ToWord<T>()(a_);}
      //: to word
};

template<typename T> // not clear. Seems not to work with T = R
bool scanFrom(T& t, istream& str)
{
   //Root<T> rt(t);
   Root<T> tIn;
   bool b=tIn.scanFrom(str); // reading the 'value' of tIn from str
   if (b) t = tIn();
   return b;
}

} // namespace

#define cpmnam CpmRoot::nameOf
#define cpmtow CpmRoot::toWord

#endif
```

53 cpmword.cpp

```
//? cpmword.cpp
//? Status of work 2023-10-20.
//?
//?  ...


#include <cpmtypes.h>
#include <stdio.h>
#include <cctype>
#include <iomanip>

using namespace CpmStd;
using CpmRoot::Word;
using CpmRoot::Z;
using CpmRoot::N;
using CpmRoot::R;
using CpmRoot::L;
using CpmRoot::toDouble;
using CpmRoot::skipLeadingWhitespace;
using CpmRoot::skipTrailingWhitespace;

bool Word::scnLin=true;
   // true is dangerous for the presenly implemented movie file
   // production. Don't change without careful analysis of the
   // consequences

Z Word::endOfWord=2; // was 1
   // 0 nothing, 1 blank, else endl
   // choice 2 is the most stable since a endl, in addition to '\n'
   // also puts flush

bool Word::prnOn(ostream& str)const
{
   if (endOfWord==0) str<<rep_;
   else if (endOfWord==1) str<<rep_<<" ";
   else              str<<rep_<<endl; // now the default case
   //return (str!=0);
   return !str ? false : true;
}

bool Word::scanFrom(istream& str)
{
   if (scnLin) CpmRoot::readLine(rep_,str);
   else        CpmRoot::read(rep_,str);
   //return (str!=0);
   return !str ? false : true;
}
```

```
Z Word::find(char c)const
{
   Z res=0;
   Z n=dim();
   for (Z i=0;i<n;++i){
      if (rep_[i]==c){
         res=i+1;
         break;
      }
   }
   return res;
}

Z Word::find(const Word& w)const
{
   string::size_type res=rep_.find(w.rep_);
   if (res==string::npos) return 0;
   else return 1+(Z)res;
}

Word Word::resize(Z newDim)const
{
   Z n1=dim();
   Z n2= newDim<0 ? 0 : newDim;
   char blank=' ';
   Word res(n2,blank);
   Z nMin = n1<=n2 ? n1 : n2;
   for (Z i=1;i<=nMin;++i) res[i]=(*this)[i];
   return res;
}

Word Word::slc_(char c, bool firstSep)
{
   Z d=dim();
   if (d==0) return *this;
   Word temp= firstSep ? *this : rev();
   Z ic=temp.find(c);
   if (ic==0){ Word res=*this; *this=""; return res;}
   if (d==1 && ic==1){ // Then *this consists of a single c
      *this="";
      return "";
   }
   Word wl(ic-1);
   Word wr(d-ic);
   Z i,j;
   for (i=1;i<ic;++i) wl[i]=temp[i];
   for (i=ic+1,j=1;i<=d;++i,++j) wr[j]=temp[i];
   if (firstSep){
      *this=wr;
```

```
      return wl;
   }
   else{
      *this=wl.rev();
      return wr.rev();
   }
}

Word Word::appBefExt(Word const& app)const
{
   Word dot(".");
   Z posDot=find(dot);
   if (posDot==0) return *this&app;
   else{
      Z nameL=posDot-1;
      Z extL=dim()-posDot+1;
      Word name=resize(nameL);
      Word ext=tail(extL);
      name&=app;
      name&=ext;
      return name;
   }
}

Word Word::baseName()const
{
   Word res;
   string::size_type i1=rep_.find_last_of("/");
   string::size_type i2=rep_.find_last_of("\\");
   string repx=rep_;
   if (i1==string::npos && i2==string::npos){
      ; // nothing to do
   }
   else if (i1==string::npos){ // then i2 is a valid string position that
      // holds a backslash
      repx.erase(0,i2+1);
   }
   else if (i2==string::npos){// then i2 is a valid string position that
      // holds a slash
      repx.erase(0,i1+1);
   }
   else{
      string::size_type imax=i1;
      if (i2>i1) imax=i2;
         // then right of imax there is np slash or backslash
      repx.erase(0,imax+1);
   }
   return Word(repx);
}
```

```
char Word::operator[](Z i)const
{
   return rep_.at(i-1); // checked access accorcing to the error handling
      // facilities of the standard library
}

char& Word::operator[](Z i)
{
   return rep_.at(i-1);// checked access accorcing to the error handling
      // facilities of the standard library
}

Word Word::cut(Z i)const
{
   string res=rep_;
   if (i>=0){
      res.erase(res.length()-i,i);
   }
   else{
      res.erase(0,-i);
   }
   return Word(res);
}

Word Word::tail(Z n)const
{
   Z d=dim();
   if (d<=n) return *this;
   Z diff=d-n; // notice diff>=1
   Word res(n);
   for (Z i=0;i<n;i++) res.rep_[i]=rep_[i+diff];
      // notice that for i==n-1, we have i+diff=n-1+d-n=d-1
      // so that we just stop with the last letter of rep_.
      // No essential efficiency compromises. Should be fast.
      // We do not use the substring functionality of the standard
      // library since my compiler does not understand Stroustrup's name
      // Basic_substring<char> even after using namespace std;
      // Also forming a new template class for this minor purpose
      // seems not be justified; also the floppy description of the
      // substring constructors on p.596 of BSC3 would require some
      // testing for being sure what the function do in case of
      // non-matching arguments.
   return res;
}

Word CpmRoot::operator &(const Word& w1, const Word& w2)
{
   return Word(w1.rep_+w2.rep_);
}
```

```
Word CpmRoot::operator +(const Word& w1, const Word& w2)
{
     return Word(w1.rep_+w2.rep_);
}

Word Word::write(bool x)
{
   return CpmRoot::toWord(x);
}

Word Word::write(string x)
{
   return Word(x);
}

#define CPM_SC\
   ostringstream ost;\
   if (length>0) ost<<std::setfill('0')<<std::setw(length)<<n;\
   else ost<<n;\
   return Word(ost.str())

Word Word::write(R n, Z length){CPM_SC;}
Word Word::write(Z n, Z length){CPM_SC;}
Word Word::write(N n, Z length){CPM_SC;}

#undef CPM_SC

R Word::toR()const
{
   stringstream sstr;
   sstr<<rep_;
   R res;
   sstr>>res;
   return res;
}

Z Word::toZ()const
{
   stringstream sstr;
   sstr<<rep_;
   Z res;
   sstr>>res;
   return res;
}

N Word::toN()const
{
   stringstream sstr;
   sstr<<rep_;
   N res;
```

```
   sstr>>res;
   return res;
}

bool Word::toBool()const
{
   if (rep_=="true" || rep_=="TRUE" ||
      rep_=="wahr" || rep_=="1" || rep_=="W")
      return true;
   else
      return false;
}

namespace{
   const int fieldLength=32;
   // privat name to file
}

#if defined(_WINDOWS)
   #define CPM_SPF sprintf_s
#else
   #define CPM_SPF sprintf
#endif

#define CPM_SC\
   char p[fieldLength];\
   CPM_SPF(p,ff,x);\
   return Word(p)

Word CpmRoot::toWord(N x, const char* ff){CPM_SC;}
Word CpmRoot::toWord(Z x, const char* ff){CPM_SC;}
Word CpmRoot::toWord(R y, const char* ff){ double x=cpmtod(y); CPM_SC;}

#undef CPM_SPF
#undef CPM_SC

Word CpmRoot::operator &(const char* cp, const Word& w)
{ return Word(cp)&w;}
Word CpmRoot::operator &(const Word& w, const char* cp)
{ return w&Word(cp);}
Word CpmRoot::operator &(char cp[], const Word& w)
{ return Word(cp)&w;}
Word CpmRoot::operator &(const Word& w, char cp[])
{ return w&Word(cp);}
Word& Word::operator &=(const Word& w)
{ return *this=(*this)&w;}
Word& Word::operator &=(const char* cp)
{ return *this=(*this)&Word(cp);}

// for convenience same is allowed to be denoted +
```

```
Word CpmRoot::operator +(const char* cp, const Word& w)
{ return Word(cp)&w;}
Word CpmRoot::operator +(const Word& w, const char* cp)
{ return w&Word(cp);}
Word CpmRoot::operator +(char cp[], const Word& w)
{ return Word(cp)&w;}
Word CpmRoot::operator +(const Word& w, char cp[])
{ return w&Word(cp);}
Word& Word::operator +=(const Word& w)
{ return *this=(*this)&w;}
Word& Word::operator +=(const char* cp)
{ return *this=(*this)&Word(cp);}

ofstream& CpmRoot::to_ofstream(const Word& w)
{
    ofstream*  outptr=new ofstream(w.rep_.c_str());
    return *outptr;
}

ifstream& CpmRoot::to_ifstream(const Word& w)
{
    ifstream*  inptr=new ifstream(w.rep_.c_str());
    return *inptr;
}

bool Word::readable()const
{
   ifstream ifs(rep_.c_str());
   if (ifs) return true;
   else return false;
}

Word Word::firstWord()const
{
   istringstream ist(rep_);
   string st;
   ist>>st;
   return Word(st);
}

Word Word::firstWord_()
{
   istringstream ist(rep_);
   string st;
   ist>>st;
   rep_=string();
   char c;
   while (ist>>c) rep_+=c;
   return Word(st);
```

```
}

Z Word::com(const Word& w)const
{
   if (rep_<w.rep_) return 1;
   else if (rep_>w.rep_) return -1;
   else return 0;
}

Word Word::makeFileName()const
{
   string::const_iterator i;
   ostringstream res;
   for (i=rep_.begin(); i!=rep_.end(); ++i){
      if (isalnum(*i)) res<<*i;  else res<<'_';
   }
   return Word(res);
}

void Word::skipSpc_()
{
   string::const_iterator i;
   ostringstream res;
   for (i=rep_.begin(); i!=rep_.end(); ++i){
      if (!isspace(*i)) res<<*i;
   }
   rep_=res.str();
}

Word Word::skipChr(char c)const
{
   string::const_iterator i;
   ostringstream res;
   for (i=rep_.begin(); i!=rep_.end(); ++i){
      if (*i!=c) res<<*i;
   }
   return Word(res);
}

namespace{
   bool isOK(char c, bool extended)
   {
      if (!extended) return c!='.';
      else return c!='_'&& (islower(c)||isdigit(c));
   }
}

// we should remove any contiguous array of upper case characters
// (here '_' is considered an upper case character).
```

```
Word Word::remExt(bool extended)const
{
   string::const_iterator i;
   ostringstream res;
   for (i=rep_.begin(); i!=rep_.end(); ++i){
      if (isOK(*i,extended)) res<<*i;
      else break;
   }
   return Word(res);
}

Word Word::remApp()const
{
   Z d=dim();
   char l=(*this)[d];
   while(isupper(l)||l=='_'){
      d--;
      l=(*this)[d];
   }
   return resize(d);
}

void Word::skipSurSpc_()
{
   string line=skipLeadingWhitespace(rep_);
   rep_=skipTrailingWhitespace(line);
}

R Word::abs(void)const{ return CpmRoot::absT<string>(rep_);}
R Word::absSqr(void)const{ return CpmRoot::absSqrT<string>(rep_);}
Word Word::con(void)const{ return Word(CpmRoot::conT<string>(rep_));}
Word Word::inv(void)const{ return Word(CpmRoot::invT<string>(rep_));}
Word Word::rev(void)const{ return Word(CpmRoot::invT<string>(rep_));}

Word Word::test(Z complexity)const
{
   string s;
   string res=CpmRoot::testT<string>(s,complexity);
   return Word(res);
}

Word Word::ran(Z j)const
{
   string res=CpmRoot::ranT<string>(rep_,j);
         // rep_ was res till 2016-03-10
   return Word(res);
}

Z Word::hash()const
{
```

```
   return CpmRoot::hashT<string>(rep_);
}

R Word::dis(Word const& x)const
{
   return CpmRoot::disT<string>(rep_,x.rep_);
}

Word Word::net(Z i)const
{
   return Word(CpmRoot::netT<string>(rep_,i));
}

// functions CpmRoot::toStr

#define CPM_SC\
   ostringstream ost;\
   if (prc>0) ost.precision(prc);\
   ost<<x;\
   return ost.str()

string CpmRoot::toStr(R const& x, Z prc){CPM_SC;}
#if defined(CPM_ULM)
   string CpmRoot::toStr(rwrap const& x, Z prc){CPM_SC;}
#endif
#undef CPM_SC

#define CPM_SC ostringstream ost; ost<<x; return ost.str()
string CpmRoot::toStr(Z x){ CPM_SC; }
string CpmRoot::toStr(N x){ CPM_SC; }
string CpmRoot::toStr(L x){ CPM_SC; }
string CpmRoot::toStr(bool x){ CPM_SC; }
#undef CPM_SC

const char asciiList[]={
   '0','1','2','3','4','5','6','7','8','9',
   'a','b','c','d','e','f','g','h','i','j','k',
   'l','m','n','o','p','q','r','s','t','u','v',
   'w','x','y','z',
   'A','B','C','D','E','F','G','H','I','J','K',
   'L','M','N','O','P','Q','R','S','T','U','V',
   'W','X','Y','Z',
   '\x20','\x21','\x22','\x23','\x24','\x25','\x26','\x27',
   '\x28','\x29','\x2a','\x2b','\x2c','\x2d','\x2e','\x2f',
   '\x3a','\x3b','\x3c','\x3d','\x3e','\x3f','\x40',
   '\x5b','\x5c','\x5d','\x5e','\x5f','\x60',
   '\x7b','\x7c','\x7d','\x7e','\x7f'
};

Word Word::ascii()
```

```
{
   ostringstream ost;
   ost<<'0'<<'1'<<'2'<<'3'<<'4'<<'5'<<'6'<<'7'<<'8'<<'9'<<
   'a'<<'b'<<'c'<<'d'<<'e'<<'f'<<'g'<<'h'<<'i'<<'j'<<'k'<<
   'l'<<'m'<<'n'<<'o'<<'p'<<'q'<<'r'<<'s'<<'t'<<'u'<<'v'<<
   'w'<<'x'<<'y'<<'z'<<
   'A'<<'B'<<'C'<<'D'<<'E'<<'F'<<'G'<<'H'<<'I'<<'J'<<'K'<<
   'L'<<'M'<<'N'<<'O'<<'P'<<'Q'<<'R'<<'S'<<'T'<<'U'<<'V'<<
   'W'<<'X'<<'Y'<<'Z'<<
   '\x20'<<'\x21'<<'\x23'<<'\x24'<<'\x25'<<'\x26'<<
   '\x28'<<'\x29'<<'\x2a'<<'\x2b'<<'\x2c'<<'\x2d'<<'\x2e'<<'\x2f'<<
   '\x3a'<<'\x3b'<<'\x3c'<<'\x3d'<<'\x3e'<<'\x3f'<<'\x40'<<
   '\x5b'<<'\x5c'<<'\x5d'<<
   '\x7b'<<'\x7c'<<'\x7d'<<'\x7e';
   return Word(ost.str());
}
```

54 cpmx.h

```
//? cpmx.h
//? Status of work 2023-10-20.
//?
//?  ...


#ifndef CPM_X_H_
#define CPM_X_H_
/*

Purpose: Defining lean cartesian products of 2-8 factors
   to be used similar to pair<> of the standard library.
   Needed for enabling functions to return longer value
   lists than just pairs.
   Non-constant reference arguments as a work-around are considered
   bad style here.
   If one needs cartesian products very often as function arguments or as
   function return values, this might be an indication that it would
   natural to define one or more classes which aggregate these components
   into a meaningful object.

   All these cartesian producs define IO and order operations in the
   same manner as Vl and V do this. This is a considerable simplification
   compared to my old approach to have templates Xno, Xnio for those
   template arguments which implement order operations and IO operations.

   Notice that access to components can be done via references and
   values. Notice also that, other than for pair, first,second,...
   are functions and not data. So the following is correct:
   X2<R,Z> a(4,6);
   Z i=a.second(); // not i=a.second;
      // i.e. i=6
   However,
   Z i=a.c2();
   looks nicer to me

   Addition February 2009: For pairs, triplets, and quartets there are now
   also homotypic versions. Component access is given here through public
   data members x1, x2, x3, x4.
   Also here, IO and order relations are implemented.
*/
#include <cpmword.h>
#include <vector>
#include <tuple>

namespace CpmArrays{
```

```
   using CpmRoot::Word;
   using CpmRoot::Z;
   using namespace CpmStd;
   using std::tuple; // 2015-06-11 conversion to std::tupel added.
   using std::get;

// Classes introduced by this file with fixed names:

//       X2, X3, X4, X5, X6, X7, X8
//       T2, T3, T4

// All template arguments are assumed to have (not necessarily to define
// explicitely) default constructor, copy constructor, and operator =

// Lists of 2,3,4,5,6,7 or 8 components (hetero-typic lists)
//   and
// homo-typic lists with 2,3, or 4 components.
// Components of Xi accessible by member functions c1(),c2(),...
// Components of Ti accessible by member functions c1(),c2(),... and by
// conventional indexing (first valid index is 1)
// (indexing added 2011-03-22)
////////////////////////////// class X2<> ////////////////////////////

template <class Y1, class Y2>
class X2{ // heterotypic pairs
   // Lean Cartesian product of Y1 with Y2
protected:
   Y1 x1;
   Y2 x2;
   typedef X2<Y1,Y2> Type;

public:
   CPM_IO
   CPM_ORDER
   Z dim()const{ return 2;}
 // constructors
   X2(Y1 const& x1_, Y2 const& x2_):x1(x1_),x2(x2_){}
   X2(void):x1(),x2(){}
   X2(X2<Y1,Y2> const& z): x1(z.x1), x2(z.x2){}
   X2(tuple<Y1,Y2> const& t): x1(get<0>(t)),x2(get<1>(t)){}

// 'modern' access functions. Similar to operator[]
// getting the second component of object a is done like
//
// ...=a.c2();
//
// setting is done like
//
// a.c2()=...;
//
```

```
   Y1& c1(){ return x1;}
   Y2& c2(){ return x2;}
   Y1 const& c1()const{ return x1;}
   Y2 const& c2()const{ return x2;}
   Y1 first()const{ return x1;}
   Y2 second()const{ return x2;}

   Word nameOf()const
   {
       return Word("X2< "&
          CpmRoot::Name<Y1>()(Y1())&
          ","&
          CpmRoot::Name<Y2>()(Y2())&
          " >");
   }

   tuple<Y1,Y2> toTup()const{ return tuple<Y1,Y2>{x1,x2};}

};

template <class Y1, class Y2>
Z X2<Y1,Y2>::com(X2<Y1,Y2> const& s)const
{
   if (c1()<s.c1()) return 1;
   if (c1()>s.c1()) return -1;
   if (c2()<s.c2()) return 1;
   if (c2()>s.c2()) return -1;
   return 0;
}

template <class Y1, class Y2>
bool X2<Y1,Y2>::prnOn(ostream& str)const
{
   cpmp(x1);
   cpmp(x2);
   return true;
}

template <class Y1, class Y2>
bool X2<Y1,Y2>::scanFrom(istream& str)
{
   cpms(x1);
   cpms(x2);
   return true;
}

////////////////////////////// class T2<> ////////////////////////////

template <class X>
class T2{ // homotypic pairs
```

```
   // Lean Cartesian product of X with itself.
public:
   X x1, x2;
   typedef T2<X> Type;
   CPM_IO
   CPM_ORDER
   Z dim()const{ return 2;}
 // constructors
   T2(X const& x1_, X const& x2_):x1(x1_),x2(x2_){}
   T2(void):x1(),x2(){}
   T2(T2<X> const& z): x1(z.x1), x2(z.x2){}
   X& operator[](Z i){ if (i>=2) return x2; else return x1;}
   X const& operator[](Z i)const{ if (i>=2) return x2; else return x1;}
   X& fir(){ return x1;}
      //: first
   X& last(){ return x2;}
      //: last
   Z b()const { return 1;}
      //. begin
      // Returns the first valid index.
   Z e()const { return 2;}
      //. end
      // Returns the last valid index.
   X& c1(void){ return x1;}
   X& c2(void){ return x2;}
   X const& c1(void)const{ return x1;}
   X const& c2(void)const{ return x2;}
   Word nameOf()const
   {
      return Word("T2< "&CpmRoot::Name<X>()(x1)&" >");
   }
};

template <class Y>
Z T2<Y>::com(T2<Y> const& s)const
{
   if (x1<s.x1) return 1;
   if (x1>s.x1) return -1;
   if (x2<s.x2) return 1;
   if (x2>s.x2) return -1;
   return 0;
}

template <class Y>
bool T2<Y>::prnOn(ostream& str)const
{
   cpmp(x1);
   cpmp(x2);
   return true;
}
```

```
template <class Y>
bool T2<Y>::scanFrom(istream& str)
{
   cpms(x1);
   cpms(x2);
   return true;
}

////////////////////////////// class X3<> //////////////////////////

// for more factors:

template <class Y1, class Y2, class Y3>
class  X3{ // heterotypic triplets
protected:
   Y1 x1;
   Y2 x2;
   Y3 x3;
   typedef X3<Y1,Y2,Y3> Type;
public:
   CPM_IO
   CPM_ORDER
   Z dim()const{ return 3;}
   X3(Y1 const& x1_, Y2 const& x2_, Y3 const& x3_):x1(x1_),x2(x2_),
      x3(x3_){}
   X3(X2<Y1,Y2> const& y_, Y3 const& x3_):x1(y_.c1()),x2(y_.c2()),
      x3(x3_){}
   X3(void):x1(),x2(),x3(){}
   X3(X3<Y1,Y2,Y3> const& z): x1(z.x1), x2(z.x2), x3(z.x3){}
   X3(tuple<Y1,Y2,Y3> const& t): x1(get<0>(t)),x2(get<1>(t)),
      x3(get<2>(t)){}

   Y1& c1(void){ return x1;}
   Y2& c2(void){ return x2;}
   Y3& c3(void){ return x3;}

   Y1 const& c1(void)const{ return x1;}
   Y2 const& c2(void)const{ return x2;}
   Y3 const& c3(void)const{ return x3;}

   Y1 first()const{ return x1;}
   Y2 second()const{ return x2;}
   Y3 third()const{ return x3;}

   Word nameOf()const
   {
       return Word("X3< "&
          CpmRoot::Name<Y1>()(Y1())&
          ","&
```

```
          CpmRoot::Name<Y2>()(Y2())&
          ","&
          CpmRoot::Name<Y3>()(Y3())&
          " >");
   }

   tuple<Y1,Y2,Y3> toTup()const{ return tuple<Y1,Y2,Y3>{x1,x2,x3};}

};

template <class Y1, class Y2, class Y3>
Z X3<Y1,Y2,Y3>::com(X3<Y1,Y2,Y3> const& s)const
{
   if (c1()<s.c1()) return 1;
   if (c1()>s.c1()) return -1;
   if (c2()<s.c2()) return 1;
   if (c2()>s.c2()) return -1;
   if (c3()<s.c3()) return 1;
   if (c3()>s.c3()) return -1;
   return 0;
}

template <class Y1, class Y2, class Y3>
bool X3<Y1,Y2,Y3>::prnOn(ostream& str)const
{
   cpmp(x1);
   cpmp(x2);
   cpmp(x3);
   return true;
}

template <class Y1, class Y2, class Y3>
bool X3<Y1,Y2,Y3>::scanFrom(istream& str)
{
   cpms(x1);
   cpms(x2);
   cpms(x3);
   return true;
}

////////////////////////////// class T3<> ////////////////////////////

template <class X>
class T3{ // homotypic triplets
   // Lean Cartesian product of X with itself.
public:
   X x1, x2, x3;
   typedef T3<X> Type;
   CPM_IO
   CPM_ORDER
```

```
   Z dim()const{ return 3;}
 // constructors
   T3(X const& x1_, X const& x2_, X const& x3_):x1(x1_),x2(x2_),x3(x3_){}
   T3(void):x1(),x2(),x3(){}
   T3(T3<X> const& z): x1(z.x1), x2(z.x2),x3(z.x3){}

   X& operator[](Z i){ if (i>=3) return x3; else if (i==2) return x2;
    else return x1;}
   X const& operator[](Z i)const{ if (i>=3) return x3; else if (i==2)
    return x2; else return x1;}
   X& fir(){ return x1;}
      //: first
   X& last(){ return x3;}
      //: last
   Z b()const { return 1;}
      //. begin
      // Returns the first valid index.
   Z e()const { return 3;}
      //. end
      // Returns the last valid index.

   X& c1(void){ return x1;}
   X& c2(void){ return x2;}
   X& c3(void){ return x3;}

   X const& c1(void)const{ return x1;}
   X const& c2(void)const{ return x2;}
   X const& c3(void)const{ return x3;}

   Word nameOf()const
   {
      return Word("T3< "&CpmRoot::Name<X>()(x1)&" >");
   }
};

template <class Y>
Z T3<Y>::com(T3<Y> const& s)const
{
   if (x1<s.x1) return 1;
   if (x1>s.x1) return -1;
   if (x2<s.x2) return 1;
   if (x2>s.x2) return -1;
   if (x3<s.x3) return 1;
   if (x3>s.x3) return -1;
   return 0;
}

template <class Y>
bool T3<Y>::prnOn(ostream& str)const
{
```

```
   cpmp(x1);
   cpmp(x2);
   cpmp(x3);
   return true;
}

template <class Y>
bool T3<Y>::scanFrom(istream& str)
{
   cpms(x1);
   cpms(x2);
   cpms(x3);
   return true;
}

////////////////////////////// class X4<> //////////////////////////////

template <class Y1, class Y2, class Y3, class Y4>
class  X4{ // heterotypic 4-tuples
protected:
   Y1 x1;
   Y2 x2;
   Y3 x3;
   Y4 x4;
   typedef X4<Y1,Y2,Y3,Y4> Type;

public:
   CPM_IO
   CPM_ORDER
   Z dim()const{ return 4;}
   X4(Y1 const& x1_, Y2 const& x2_, Y3 const& x3_, Y4 const& x4_):
      x1(x1_),x2(x2_),x3(x3_),x4(x4_){}
   X4(X3<Y1,Y2,Y3> const& y_, Y4 const& x4_):
      x1(y_.c1()),x2(y_.c2()),x3(y_.c3()),x4(x4_){}
   X4(void):x1(),x2(),x3(),x4(){}
   X4(X4<Y1,Y2,Y3,Y4> const& z): x1(z.x1), x2(z.x2), x3(z.x3),x4(z.x4){}
   X4(tuple<Y1,Y2,Y3,Y4> const& t): x1(get<0>(t)),x2(get<1>(t)),
      x3(get<2>(t)),x4(get<3>(t)){}

   Y1& c1(void){ return x1;}
   Y2& c2(void){ return x2;}
   Y3& c3(void){ return x3;}
   Y4& c4(void){ return x4;}

   Y1 const& c1(void)const{ return x1;}
   Y2 const& c2(void)const{ return x2;}
   Y3 const& c3(void)const{ return x3;}
   Y4 const& c4(void)const{ return x4;}

   Y1 first()const{ return x1;}
```

```
   Y2 second()const{ return x2;}
   Y3 third()const{ return x3;}
   Y4 fourth()const{ return x4;}

   Word nameOf()const
   {
       return Word("X4< "&
          CpmRoot::Name<Y1>()(Y1())&
          ","&
          CpmRoot::Name<Y2>()(Y2())&
          ","&
          CpmRoot::Name<Y3>()(Y3())&
          ","&
          CpmRoot::Name<Y4>()(Y4())&
          " >");
   }

   tuple<Y1,Y2,Y3,Y4> toTup()const
   {
      return tuple<Y1,Y2,Y3,Y4>{x1,x2,x3,x4};
   }

};

template <class Y1, class Y2, class Y3, class Y4>
Z X4<Y1,Y2,Y3,Y4>::com(X4<Y1,Y2,Y3,Y4> const& s)const
{
   if (c1()<s.c1()) return 1;
   if (c1()>s.c1()) return -1;
   if (c2()<s.c2()) return 1;
   if (c2()>s.c2()) return -1;
   if (c3()<s.c3()) return 1;
   if (c3()>s.c3()) return -1;
   if (c4()<s.c4()) return 1;
   if (c4()>s.c4()) return -1;
   return 0;
}

template <class Y1, class Y2, class Y3, class Y4>
bool X4<Y1,Y2,Y3,Y4>::prnOn(ostream& str)const
{
   cpmp(x1);
   cpmp(x2);
   cpmp(x3);
   cpmp(x4);
   return true;
}

template <class Y1, class Y2, class Y3, class Y4>
bool X4<Y1,Y2,Y3,Y4>::scanFrom(istream& str)
```

```
{
   cpms(x1);
   cpms(x2);
   cpms(x3);
   cpms(x4);
   return true;
}

////////////////////////////// class T4<> ////////////////////////////////

template <class X>
class T4{ // homotypic quartets
   // Lean Cartesian product of X with itself.
public:
   X x1, x2, x3, x4;
   typedef T4<X> Type;
   CPM_IO
   CPM_ORDER
   Z dim()const{ return 4;}
 // constructors
   T4(X const& x1_, X const& x2_, X const& x3_, X const& x4_):x1(x1_),
      x2(x2_),x3(x3_),x4(x4_){}
   T4(void):x1(),x2(),x3(),x4(){}
   T4(T4<X> const& z): x1(z.x1), x2(z.x2),x3(z.x3),x4(z.x4){}

   X& operator[](Z i)
   { if (i>=4) return x4; else if (i==3) return x3;
      else if (i==2) return x2; else return x1;}

   X const& operator[](Z i)const
   { if (i>=4) return x4; else if (i==3) return x3;
      else if (i==2) return x2; else return x1;}

   X& fir(){ return x1;}
      //: first
   X& last(){ return x4;}
      //: last
   Z b()const { return 1;}
      //. begin
      // Returns the first valid index.
   Z e()const { return 4;}
      //. end
      // Returns the last valid index.

   X& c1(void){ return x1;}
   X& c2(void){ return x2;}
   X& c3(void){ return x3;}
   X& c4(void){ return x4;}

   X const& c1(void)const{ return x1;}
```

```
   X const& c2(void)const{ return x2;}
   X const& c3(void)const{ return x3;}
   X const& c4(void)const{ return x4;}

   Word nameOf()const
   {
      return Word("T4< "&CpmRoot::Name<X>()(x1)&" >");
   }
};

template <class Y>
Z T4<Y>::com(T4<Y> const& s)const
{
   if (x1<s.x1) return 1;
   if (x1>s.x1) return -1;
   if (x2<s.x2) return 1;
   if (x2>s.x2) return -1;
   if (x3<s.x3) return 1;
   if (x3>s.x3) return -1;
   if (x4<s.x4) return 1;
   if (x4>s.x4) return -1;
   return 0;
}

template <class Y>
bool T4<Y>::prnOn(ostream& str)const
{
   cpmp(x1);
   cpmp(x2);
   cpmp(x3);
   cpmp(x4);
   return true;
}

template <class Y>
bool T4<Y>::scanFrom(istream& str)
{
   cpms(x1);
   cpms(x2);
   cpms(x3);
   cpms(x4);
   return true;
}

////////////////////////////// class X5<> //////////////////////////////

template <class Y1, class Y2, class Y3, class Y4, class Y5>
class  X5{ // heterotypic 5-tuples
protected:
   Y1 x1;
```

```
   Y2 x2;
   Y3 x3;
   Y4 x4;
   Y5 x5;
   typedef X5<Y1,Y2,Y3,Y4,Y5> Type;
public:
   CPM_IO
   CPM_ORDER
   Z dim()const{ return 5;}
   X5(Y1 const& x1_, Y2 const& x2_, Y3 const& x3_,
      Y4 const& x4_, Y5 const& x5_):x1(x1_),x2(x2_),x3(x3_),
      x4(x4_),x5(x5_){}
   X5(X4<Y1,Y2,Y3,Y4> const& y_, Y5 const& x5_):
      x1(y_.c1()),x2(y_.c2()),x3(y_.c3()),
      x4(y_.c4()),x5(x5_){}
   X5(void):x1(),x2(),x3(),x4(),x5(){}
   X5(X5<Y1,Y2,Y3,Y4,Y5> const& z): x1(z.x1), x2(z.x2),
      x3(z.x3),x4(z.x4),x5(z.x5){}

   X5(tuple<Y1,Y2,Y3,Y4,Y5> const& t): x1(get<0>(t)),x2(get<1>(t)),
      x3(get<2>(t)),x4(get<3>(t)),x5(get<4>(t)){}

   Y1& c1(void){ return x1;}
   Y2& c2(void){ return x2;}
   Y3& c3(void){ return x3;}
   Y4& c4(void){ return x4;}
   Y5& c5(void){ return x5;}

   Y1 const& c1(void)const{ return x1;}
   Y2 const& c2(void)const{ return x2;}
   Y3 const& c3(void)const{ return x3;}
   Y4 const& c4(void)const{ return x4;}
   Y5 const& c5(void)const{ return x5;}

   Y1 first()const{ return x1;}
   Y2 second()const{ return x2;}
   Y3 third()const{ return x3;}
   Y4 fourth()const{ return x4;}
   Y5 fifth()const{ return x5;}

   Word nameOf()const
   {
       return Word("X5< "&
          CpmRoot::Name<Y1>()(Y1())&
          ","&
          CpmRoot::Name<Y2>()(Y2())&
          ","&
          CpmRoot::Name<Y3>()(Y3())&
          ","&
          CpmRoot::Name<Y4>()(Y4())&
```

```
          ","&
          CpmRoot::Name<Y5>()(Y5())&
          " >");
   }

   tuple<Y1,Y2,Y3,Y4,Y5> toTup()const
   {
      return tuple<Y1,Y2,Y3,Y4,Y5>{x1,x2,x3,x4,x5};
   }
};

template <class Y1, class Y2, class Y3, class Y4, class Y5>
Z X5<Y1,Y2,Y3,Y4,Y5>::com(X5<Y1,Y2,Y3,Y4,Y5> const& s)const
{
   if (c1()<s.c1()) return 1;
   if (c1()>s.c1()) return -1;
   if (c2()<s.c2()) return 1;
   if (c2()>s.c2()) return -1;
   if (c3()<s.c3()) return 1;
   if (c3()>s.c3()) return -1;
   if (c4()<s.c4()) return 1;
   if (c4()>s.c4()) return -1;
   if (c5()<s.c5()) return 1;
   if (c5()>s.c5()) return -1;
   return 0;
}

template <class Y1, class Y2, class Y3, class Y4, class Y5>
bool X5<Y1,Y2,Y3,Y4,Y5>::prnOn(ostream& str)const
{
   cpmp(x1);
   cpmp(x2);
   cpmp(x3);
   cpmp(x4);
   cpmp(x5);
   return true;
}

template <class Y1, class Y2, class Y3, class Y4, class Y5>
bool X5<Y1,Y2,Y3,Y4,Y5>::scanFrom(istream& str)
{
   cpms(x1);
   cpms(x2);
   cpms(x3);
   cpms(x4);
   cpms(x5);
   return true;
}

////////////////////////////// class X6<> ////////////////////////////
```

```
template <class Y1, class Y2, class Y3, class Y4, class Y5, class Y6>
class  X6{ // heterotypic 6-tuples
protected:
   Y1 x1;
   Y2 x2;
   Y3 x3;
   Y4 x4;
   Y5 x5;
   Y6 x6;
   typedef X6<Y1,Y2,Y3,Y4,Y5,Y6> Type;
public:
   CPM_IO
   CPM_ORDER
   Z dim()const{ return 6;}

   X6(Y1 const& x1_, Y2 const& x2_, Y3 const& x3_,
      Y4 const& x4_, Y5 const& x5_, Y6 const& x6_):x1(x1_),x2(x2_),
      x3(x3_), x4(x4_),x5(x5_),x6(x6_){}

   X6(X5<Y1,Y2,Y3,Y4,Y5> const& y_, Y6 const& x6_):
      x1(y_.c1()),x2(y_.c2()),x3(y_.c3()),
      x4(y_.c4()),x5(y_.c5()),x6(x6_){}

   X6(void):x1(),x2(),x3(),x4(),x5(),x6(){}

   X6(X6<Y1,Y2,Y3,Y4,Y5,Y6> const& z): x1(z.x1), x2(z.x2),
      x3(z.x3),x4(z.x4),x5(z.x5),x6(z.x6){}

   X6(tuple<Y1,Y2,Y3,Y4,Y5,Y6> const& t): x1(get<0>(t)),x2(get<1>(t)),
      x3(get<2>(t)),x4(get<3>(t)),x5(get<4>(t)),x6(get<5>(t)){}

   Y1& c1(void){ return x1;}
   Y2& c2(void){ return x2;}
   Y3& c3(void){ return x3;}
   Y4& c4(void){ return x4;}
   Y5& c5(void){ return x5;}
   Y6& c6(void){ return x6;}

   Y1 const& c1(void)const{ return x1;}
   Y2 const& c2(void)const{ return x2;}
   Y3 const& c3(void)const{ return x3;}
   Y4 const& c4(void)const{ return x4;}
   Y5 const& c5(void)const{ return x5;}
   Y6 const& c6(void)const{ return x6;}

   Y1 first()const{ return x1;}
   Y2 second()const{ return x2;}
   Y3 third()const{ return x3;}
   Y4 fourth()const{ return x4;}
```

```
   Y5 fifth()const{ return x5;}
   Y6 sixth()const{ return x6;}

   Word nameOf()const
   {
       return Word("X6< "&
          CpmRoot::Name<Y1>()(Y1())&
          ","&
          CpmRoot::Name<Y2>()(Y2())&
          ","&
          CpmRoot::Name<Y3>()(Y3())&
          ","&
          CpmRoot::Name<Y4>()(Y4())&
          ","&
          CpmRoot::Name<Y5>()(Y5())&
          ","&
          CpmRoot::Name<Y6>()(Y6())&
          " >");
   }

   tuple<Y1,Y2,Y3,Y4,Y5,Y6> toTup()const
   {
      return tuple<Y1,Y2,Y3,Y4,Y5,Y6>{x1,x2,x3,x4,x5,x6};
   }
};

template <class Y1, class Y2, class Y3, class Y4, class Y5, class Y6>
Z X6<Y1,Y2,Y3,Y4,Y5,Y6>::com(X6<Y1,Y2,Y3,Y4,Y5,Y6> const& s)const
{
   if (c1()<s.c1()) return 1;
   if (c1()>s.c1()) return -1;
   if (c2()<s.c2()) return 1;
   if (c2()>s.c2()) return -1;
   if (c3()<s.c3()) return 1;
   if (c3()>s.c3()) return -1;
   if (c4()<s.c4()) return 1;
   if (c4()>s.c4()) return -1;
   if (c5()<s.c5()) return 1;
   if (c5()>s.c5()) return -1;
   if (c6()<s.c6()) return 1;
   if (c6()>s.c6()) return -1;
   return 0;
}

template <class Y1, class Y2, class Y3, class Y4, class Y5, class Y6>
bool X6<Y1,Y2,Y3,Y4,Y5,Y6>::prnOn(ostream& str)const
{
   cpmp(x1);
   cpmp(x2);
   cpmp(x3);
```

```
   cpmp(x4);
   cpmp(x5);
   cpmp(x6);
   return true;
}

template <class Y1, class Y2, class Y3, class Y4, class Y5, class Y6>
bool X6<Y1,Y2,Y3,Y4,Y5,Y6>::scanFrom(istream& str)
{
   cpms(x1);
   cpms(x2);
   cpms(x3);
   cpms(x4);
   cpms(x5);
   cpms(x6);
   return true;
}

////////////////////////////// class X7<> //////////////////////////

template <class Y1, class Y2, class Y3, class Y4, class Y5,
   class Y6, class Y7>
class  X7{ // heterotypic 7-tuples
protected:
   Y1 x1;
   Y2 x2;
   Y3 x3;
   Y4 x4;
   Y5 x5;
   Y6 x6;
   Y7 x7;
   typedef X7<Y1,Y2,Y3,Y4,Y5,Y6,Y7> Type;
public:
   CPM_IO
   CPM_ORDER
   Z dim()const{ return 7;}
   X7(Y1 const& x1_, Y2 const& x2_, Y3 const& x3_,
      Y4 const& x4_, Y5 const& x5_, Y6 const& x6_, Y7 const& x7_):
      x1(x1_),x2(x2_),x3(x3_),x4(x4_),x5(x5_),x6(x6_),x7(x7_){}

   X7(X6<Y1,Y2,Y3,Y4,Y5,Y6> const& y_, Y7 const& x7_):
      x1(y_.c1()),x2(y_.c2()),x3(y_.c3()),
      x4(y_.c4()),x5(y_.c5()),x6(y_.c6()),x7(x7_){}

   X7(void):x1(),x2(),x3(),x4(),x5(),x6(),x7(){}
   X7(X7<Y1,Y2,Y3,Y4,Y5,Y6,Y7> const& z): x1(z.x1), x2(z.x2),
      x3(z.x3),x4(z.x4),x5(z.x5),x6(z.x6),x7(z.x7){}

   X7(tuple<Y1,Y2,Y3,Y4,Y5,Y6,Y7> const& t): x1(get<0>(t)),x2(get<1>(t)),
      x3(get<2>(t)),x4(get<3>(t)),x5(get<4>(t)),x6(get<5>(t)),
```

```
   x7(get<6>(t)){}


Y1& c1(void){ return x1;}
Y2& c2(void){ return x2;}
Y3& c3(void){ return x3;}
Y4& c4(void){ return x4;}
Y5& c5(void){ return x5;}
Y6& c6(void){ return x6;}
Y7& c7(void){ return x7;}

Y1 const& c1(void)const{ return x1;}
Y2 const& c2(void)const{ return x2;}
Y3 const& c3(void)const{ return x3;}
Y4 const& c4(void)const{ return x4;}
Y5 const& c5(void)const{ return x5;}
Y6 const& c6(void)const{ return x6;}
Y7 const& c7(void)const{ return x7;}

Y1 first()const{ return x1;}
Y2 second()const{ return x2;}
Y3 third()const{ return x3;}
Y4 fourth()const{ return x4;}
Y5 fifth()const{ return x5;}
Y6 sixth()const{ return x6;}
Y7 seventh()const{ return x7;}

Word nameOf()const
{
    return Word("X7< "&
       CpmRoot::Name<Y1>()(Y1())&
       ","&
       CpmRoot::Name<Y2>()(Y2())&
       ","&
       CpmRoot::Name<Y3>()(Y3())&
       ","&
       CpmRoot::Name<Y4>()(Y4())&
       ","&
       CpmRoot::Name<Y5>()(Y5())&
       ","&
       CpmRoot::Name<Y6>()(Y6())&
       ","&
       CpmRoot::Name<Y7>()(Y7())&
       " >");
}

tuple<Y1,Y2,Y3,Y4,Y5,Y6,Y7> toTup()const
{
   return tuple<Y1,Y2,Y3,Y4,Y5,Y6,Y7>{x1,x2,x3,x4,x5,x6,x7};
}
```

```
};

template <class Y1, class Y2, class Y3, class Y4, class Y5,
   class Y6, class Y7>
Z X7<Y1,Y2,Y3,Y4,Y5,Y6,Y7>::com(X7<Y1,Y2,Y3,Y4,Y5,Y6,Y7> const& s)const
{
   if (c1()<s.c1()) return 1;
   if (c1()>s.c1()) return -1;
   if (c2()<s.c2()) return 1;
   if (c2()>s.c2()) return -1;
   if (c3()<s.c3()) return 1;
   if (c3()>s.c3()) return -1;
   if (c4()<s.c4()) return 1;
   if (c4()>s.c4()) return -1;
   if (c5()<s.c5()) return 1;
   if (c5()>s.c5()) return -1;
   if (c6()<s.c6()) return 1;
   if (c6()>s.c6()) return -1;
   if (c7()<s.c7()) return 1;
   if (c7()>s.c7()) return -1;
   return 0;
}

template <class Y1, class Y2, class Y3, class Y4, class Y5,
   class Y6, class Y7>
bool X7<Y1,Y2,Y3,Y4,Y5,Y6,Y7>::prnOn(ostream& str)const
{
   cpmp(x1);
   cpmp(x2);
   cpmp(x3);
   cpmp(x4);
   cpmp(x5);
   cpmp(x6);
   cpmp(x7);
   return true;
}

template <class Y1, class Y2, class Y3, class Y4, class Y5,
   class Y6, class Y7>
bool X7<Y1,Y2,Y3,Y4,Y5,Y6,Y7>::scanFrom(istream& str)
{
   cpms(x1);
   cpms(x2);
   cpms(x3);
   cpms(x4);
   cpms(x5);
   cpms(x6);
   cpms(x7);
   return true;
```

```
}

///////////////////////////// class X8<> ////////////////////////////

template <class Y1, class Y2, class Y3, class Y4, class Y5,
   class Y6, class Y7, class Y8>
class  X8{ // heterotypic 8-tuples
protected:
   Y1 x1;
   Y2 x2;
   Y3 x3;
   Y4 x4;
   Y5 x5;
   Y6 x6;
   Y7 x7;
   Y8 x8;
   typedef X8<Y1,Y2,Y3,Y4,Y5,Y6,Y7,Y8> Type;
public:
   CPM_IO
   CPM_ORDER
   Z dim()const{ return 8;}
   X8(Y1 const& x1_, Y2 const& x2_, Y3 const& x3_,
      Y4 const& x4_, Y5 const& x5_, Y6 const& x6_, Y7 const& x7_,
      Y8 const& x8_):
      x1(x1_),x2(x2_),x3(x3_),x4(x4_),x5(x5_),x6(x6_),x7(x7_),x8(x8_){}

   X8(X7<Y1,Y2,Y3,Y4,Y5,Y6,Y7> const& y_, Y8 const& x8_):
      x1(y_.c1()),x2(y_.c2()),x3(y_.c3()),
      x4(y_.c4()),x5(y_.c5()),x6(y_.c6()),x7(y_.c7()),x8(x8_){}

   X8(void):x1(),x2(),x3(),x4(),x5(),x6(),x7(),x8(){}
   X8(X8<Y1,Y2,Y3,Y4,Y5,Y6,Y7,Y8> const& z): x1(z.x1), x2(z.x2),
      x3(z.x3),x4(z.x4),x5(z.x5),x6(z.x6),x7(z.x7),x8(z.x8){}

   X8(tuple<Y1,Y2,Y3,Y4,Y5,Y6,Y7,Y8> const& t): x1(get<0>(t)),
      x2(get<1>(t)),x3(get<2>(t)),x4(get<3>(t)),x5(get<4>(t)),
      x6(get<5>(t)),x7(get<6>(t)),x8(get<7>(t)){}

   Y1& c1(void){ return x1;}
   Y2& c2(void){ return x2;}
   Y3& c3(void){ return x3;}
   Y4& c4(void){ return x4;}
   Y5& c5(void){ return x5;}
   Y6& c6(void){ return x6;}
   Y7& c7(void){ return x7;}
   Y8& c8(void){ return x8;}

   Y1 const& c1(void)const{ return x1;}
   Y2 const& c2(void)const{ return x2;}
   Y3 const& c3(void)const{ return x3;}
```

```
   Y4 const& c4(void)const{ return x4;}
   Y5 const& c5(void)const{ return x5;}
   Y6 const& c6(void)const{ return x6;}
   Y7 const& c7(void)const{ return x7;}
   Y8 const& c8(void)const{ return x8;}

   Y1 first()const{ return x1;}
   Y2 second()const{ return x2;}
   Y3 third()const{ return x3;}
   Y4 fourth()const{ return x4;}
   Y5 fifth()const{ return x5;}
   Y6 sixth()const{ return x6;}
   Y7 seventh()const{ return x7;}
   Y8 eighth()const{ return x8;}

   Word nameOf()const
   {
       return Word("X8< "&
          CpmRoot::Name<Y1>()(Y1())&
          ","&
          CpmRoot::Name<Y2>()(Y2())&
          ","&
          CpmRoot::Name<Y3>()(Y3())&
          ","&
          CpmRoot::Name<Y4>()(Y4())&
          ","&
          CpmRoot::Name<Y5>()(Y5())&
          ","&
          CpmRoot::Name<Y6>()(Y6())&
          ","&
          CpmRoot::Name<Y7>()(Y7())&
          ","&
          CpmRoot::Name<Y8>()(Y8())&
          " >");
   }

   tuple<Y1,Y2,Y3,Y4,Y5,Y6,Y7,Y8> toTup()const
   {
      return tuple<Y1,Y2,Y3,Y4,Y5,Y6,Y7,Y8>{x1,x2,x3,x4,x5,x6,x7,x8};
   }

};

template <class Y1, class Y2, class Y3, class Y4, class Y5,
   class Y6, class Y7, class Y8>
Z X8<Y1,Y2,Y3,Y4,Y5,Y6,Y7,Y8>::com(
   X8<Y1,Y2,Y3,Y4,Y5,Y6,Y7,Y8> const& s)const
{
   if (c1()<s.c1()) return 1;
   if (c1()>s.c1()) return -1;
```

```
   if (c2()<s.c2()) return 1;
   if (c2()>s.c2()) return -1;
   if (c3()<s.c3()) return 1;
   if (c3()>s.c3()) return -1;
   if (c4()<s.c4()) return 1;
   if (c4()>s.c4()) return -1;
   if (c5()<s.c5()) return 1;
   if (c5()>s.c5()) return -1;
   if (c6()<s.c6()) return 1;
   if (c6()>s.c6()) return -1;
   if (c7()<s.c7()) return 1;
   if (c7()>s.c7()) return -1;
   if (c8()<s.c8()) return 1;
   if (c8()>s.c8()) return -1;
   return 0;
}

template <class Y1, class Y2, class Y3, class Y4, class Y5,
   class Y6, class Y7, class Y8>
bool X8<Y1,Y2,Y3,Y4,Y5,Y6,Y7,Y8>::prnOn(ostream& str)const
{
   cpmp(x1);
   cpmp(x2);
   cpmp(x3);
   cpmp(x4);
   cpmp(x5);
   cpmp(x6);
   cpmp(x7);
   cpmp(x8);
   return true;
}

template <class Y1, class Y2, class Y3, class Y4, class Y5,
   class Y6, class Y7, class Y8>
bool X8<Y1,Y2,Y3,Y4,Y5,Y6,Y7,Y8>::scanFrom(istream& str)
{
   cpms(x1);
   cpms(x2);
   cpms(x3);
   cpms(x4);
   cpms(x5);
   cpms(x6);
   cpms(x7);
   cpms(x8);
   return true;
}

// In a context where X1,X2,X3,X4,X5,X6,X7,X8 are used as
// names for template arguments one could get in trouble
// by a statement
```

```
//
// using CpmArrays::X2;
//
// thus one should avoid this. Then the full name CpmArrays::X2 would
// be necessary  which is too long a name for this innocent construct.
// By the following definition we may use cpmX2, which follows the same
// notational scheme as cpmerror, cpmwrite, ...

#define cpmX2 CpmArrays::X2
#define cpmX3 CpmArrays::X3
#define cpmX4 CpmArrays::X4
#define cpmX5 CpmArrays::X5
#define cpmX6 CpmArrays::X6
#define cpmX7 CpmArrays::X7
#define cpmX8 CpmArrays::X8

#define cpmT2 CpmArrays::T2
#define cpmT3 CpmArrays::T3
#define cpmT4 CpmArrays::T4

} // CpmArrays

#endif
```

55 cpmzinterval.h

```
//? cpmzinterval.h
//? Status of work 2023-10-20.
//?
//?  ...


#ifndef CPM_ZINTERVAL_H_
#define CPM_ZINTERVAL_H_
/*
    Description:
      class for 'intervals' in Z

    Recent history:
    2012-01-03 functions setUnn, setSec, setDif, leftOf, rigOf added,
      #include <cpmx.h> added

*/

#include <cpmsystem.h>
#include <cpmx.h>
   // needed for functions that return a pair of IvZ objects

namespace CpmArrays{

   using CpmRoot::Z;
  // using CpmRoot::div;
   using CpmRoot::modWL;
   using CpmRoot::R;
   using CpmRoot::rnd;
   using CpmRoot::Word;

////////////////////// class IvZ ////////////////////////////////////
// class IvZ will be used in the definition of the basic array template
// V<> which, in turn, will be used to define the set template S<>.
// So it is not possible at this point to relate IvZ with S<Z>
// although clearly each instance of IvZ determines uniquely
// an instance of S<Z>. The C++ classes IvZ and S<Z> are very
// different in implementation and efficiency. For instance
//    IvZ iv(1,10000);
// defines a 'set' of 10000 integers by memorizing just two
// integers (1 and 10000 in this case). Whereas
//    S<Z> s(10000);
// (which calls a private constructor and thus may appear only in
// member function definition code) allocates memory for
// 10000 integers. When V<> will have been defined, we can give
// as V<IvZ> a storage-economic representation of arbitrary
// finite subsets of Z.
```

```
// That IvZ 'represents' subsets of Z expresses itself in
// the member function
//    bool hasElm(Z const& i)const
// which S<Z> defines with just the same signature.

class IvZ{
   // 'intervals of integers', i.e. contiguous finite subsets of Z
   // The void set is included; this is important since T-valued arrays
   // indexed by i's ranging over some instance of IvZ will be used
   // for defining our basic array template class V<T>. V<T> has to have
   // void arrays, which correspond to a void index set.
   // No virtual functions for efficency.
// independent data
   Z c_;
      // c for 'cardinality' arbitrary non-negative integer,
      // 0 for the void set.
   Z l_;
      // l for 'left', arbitrary integer
      // if c_==0 per def l_==1

// dependent quantity
   Z r_;
      // r for 'right'
      // r_= l_ + c_ - 1
      // the right-most element of *this if *this is not void.
      // it is the left neigbor of l_ if the set is void.

// initializing the dependent data
   void ini_(){
      if (c_<=0){ c_=0; l_=1; r_=l_-1;}
      else r_=l_+c_-1;
   }

// enabling declaration macros
   typedef IvZ Type;
public:
// infra-structure by declaration macros

   CPM_IO
      // stream interaction
   CPM_ORDER
      // order-related functions
      // The void interval preceeds every non-void interval.
      // All void intervals are equal.
      // For non-void intervals we use lexicographic order of the pairs
      // (l_,r_).
   //CPM_RFD
      // rule of five via default
   CPM_TEST_X
      // functions ran, test, hash, dis, abs, absSqr
```

```
   Word nameOf()const{ return "IvZ";}
      //: name of
      // class name of *this

   Word toWord()const;
      //: to word
      // text form of *this



// constructors

   IvZ():c_(0),l_(1){ini_();}
      // default constructor. The l_(1) expresses our preference of
      // vectors for which 1 is the lowest index value. Since this
      // constructor has no element, l_ has no significance.

//   IvZ()=default;
//   IvZ(const IvZ &)=default;
//constexpr IvZ(IvZ &&)=default;
//   IvZ& operator=(const IvZ &)=default;
//constexpr IvZ& operator=(IvZ &&)=default;

   explicit IvZ(Z n):c_(n),l_(1){ini_();}
      // The 'standard set' with n elements.
      // For n=0 it is the default constructor, which makes the void set.
      // Here, by definition, the smallest element of any non-void
      // standard set is 1 (not 0!).

   IvZ(Z i, Z j)
   {
      if (i<=j){ c_=j-i+1; l_=i; }
      else { c_=i-j+1; l_=j; }
      r_=l_+c_-1;
   }
      // the minimum contiguous subset of Z which contains both
      // i and j; no order restrictions to i,j. Notice that the
      // void set can't be obtained that way.

   IvZ(Z c, Z l, Z dummy):c_(c),l_(l){dummy; ini_();}
      // constructor which gives the cardinality and the first
      // element. The void set is obtained for c==0

   IvZ(R a, R b);
      // construction of the smallest IvZ which contains a and b.

// basic properties
   bool isVoid()const{ return c_==0;}
      //: is void
```

```
      //':' means that the name is formed according to the CPM standard
      // scheme.

   Z car()const{ return c_;}
      //: cardinality
      // cardinality of the set given by *this

   Z size()const{ return c_;}
      //: size
      // equals the cardinality of the set given by *this

   Z width()const{ return c_==0 ? 0 : c_-1;}
      //:: width
      //'::' means that the name is longer than the one formed according
      // to the CPM standard abbreviation scheme. In most cases
      // no abbreviation at all.
      // distance between the last point and the first set.
      // Thus diameter of the set. Obviously the diameter
      // of a singlet set is zero, as is the diameter of a void set.

// The intention behind inf() and sup() is that
// iv==IvZ(iv.inf(),iv.sup()) for every non-void iv \in IvZ
// Thus inf() and sup() have to warn if called for a void set.
   Z inf()const;
      //: infimum

   Z sup()const;
      //: supremum

   Z mean()const{ return l_ + c_/2; }
      //: mean
      // IvZ(1,5).mean() = 3 = median of (1,2,3,4,5)
      // Iv(1,4).mean() = 3 lowerst integer which is >= median of
      // (1,2,3,4)

   Z b()const{ return l_;}
      //. begin
      //'.' means that the name is shorter than the one formed according
      // to the CPM standard abbreviation scheme. In most cases
      // abbreviation to the first letter of the full name.

   Z n()const{ return r_+1;}
      //. next
      // relative to the numbers belonging to *this
      // this is the next number with respect to standard order

   IvZ& n_(){ c_++; ini_(); return *this; }
      //. next
      // iv.n_() results from iv if the number iv.n() is appended
```

```
IvZ& p_(){ c_++; l_--; ini_(); return *this; }
   //. previous
   // iv.n_() results from iv if the number iv.n() is appended

Z e()const{ return r_;}
   //. end
   // actually the last number belonging to *this
   // if *this is non-void as a set.
   // Looping over an IvZ ivz can allways be done as
   // for (Z i=ivz.b();i<=ivz.e();++i) ... ivz[i]....
   // or, equally well
   // for (Z i=ivz.b();i<ivz.n();++i) ... ivz[i]....

Z fir()const{ return l_;}
   //: first

Z last()const{ return r_;}
   //: last

Z operator[](Z i)const{ if (i<=1) return l_; else return r_;}
   //: []

Z gap(IvZ const& iv);
   //: gap
   // Distance between the sets *this and iv.
   // Here the understanding is that the distance from a void
   // set to any set is 0.

Z relPos(Z i)const
   //: relative position
   // gives a code for the location of i relative to *this.
{
   if (i<l_) return -1;
   else if (i>r_) return 1;
   else return 0;
}

Z relInd(Z i)const{ return i-l_;}
   //: relative index
   // gives the difference of i with respect to the left end
   // ('the origin') of *this.
   // If *this is void the result is meaningless. Nevertheless
   // for efficiency reasons we do not test voidness.

Z fromRelInd(Z i)const{ return l_+i;}
   //: from relative index
   // iv.fromRelInd(i) returns an index j such that i = iv.relInd(j)

Z ri(Z i)const{ return i-l_;}
   //. relative index
```

```
      // short form of relInd

   Z cyc(Z const& i)const
      //: cyclic
      // Z-->Z, i|-->iv.cyc(i) is a periodic function with period
      // iv.car(), which coincides with the idendity function id on
      // the set iv.
      // Test: MathematicaMisc/cyc.nb. This did not work:
      // Mathematica has a different reminder definition from % of C++
      // Now I have made a function which follows the Mathematica
      // convention.
      {
         return l_+modWL(i-l_,c_);
      }

   Z con(Z i)const
      //: constant
      // Z-->Z, i|-->iv.con(i) is the identity function id on
      // iv and defined as the constant continuation of id
      // outside of iv.
   {
      if (i<=l_) return l_;
      else if (i>=r_) return r_;
      else return i;
   }

   IvZ leftOf(Z i)const;
      //: left of
      // Returns the part of *this the elements j of which
      // satisfy j < i (and the void interval else).
//   {
//      if (l_ >= i) return IvZ();
//      if (r_ < i) return *this;
//      return IvZ(l_,i-1);
//   }

   IvZ rigOf(Z i)const;
      //: right of
      // Returns the part of *this the elements j of which
      // satisfy  i < j (and the void interval else).
//   {
//      if (r_ <= i) return IvZ();
//      if (l_ > i) return *this;
//      return IvZ(i+1,r_);
//   }

   IvZ operator|(IvZ const& iv)const;
      // minimal interval that contains the union
      // join or l.u.b. (lowest upper bound) in lattice terminology
```

```
   IvZ join(IvZ const& iv)const{ return (*this)|iv;}
      //:join

   IvZ operator&(IvZ const& iv)const;
      // section, intersection
      // meet or g.l.b (greates lower bound) in lattice terminology.

   IvZ meet(IvZ const& iv)const{ return (*this)&iv;}
      //: meet

// set operations may yield a pair of IvZs as result
   T2<IvZ> setUnn(IvZ const& iv)const;
      //: set union
      // Consider T2<IvZ> res = iv1.setUnn(iv2);
      // If the set union of iv1 and iv2 is contiguous, the corresponding
      // IvZ-object will be res.e(), and res.b() is set to be void.
      // This order is chosen to have res.b() < res.e() in all cases.
      // Notice that the void IvZ preceeds all IvZ's.
      // If the set union is not contiguous it determines two IvZ-objects
      // which will be stored in res such that res.b() < res.e().

   T2<IvZ> setSec(IvZ const& iv)const;//{return T2<IvZ>(IvZ(),meet(iv));}
      //: set section
      // Coincides with lattice meet, which it returns as the second
      // component of the result. The first component is set to be void.
      // (Then the components are ordered).

   T2<IvZ> setDif(IvZ const& iv)const;
      //: set difference
      // Consider T2<IvZ> res = iv1.setDif(iv2);
      // If the set difference iv1\iv2 is contiguous, the corresponding
      // IvZ-object will be res.e(), and res.b() is set to be void.
      // See setUnn for reasons.
      // If the set union is not contiguous it determines two IvZ-objects
      // which will be stored in res such that res.b() < res.e().

// further functions
   IvZ app(IvZ const& iv)const
   {
      if (c_==0) return iv;
      else return IvZ(c_+iv.c_,l_);
   }
      //: append
      // We return a IvZ which is obtained from *this (if it is not void)
      // by appending iv.car() elements at the right end. If this is void
      // we simply return iv.

   bool hasElm(Z i)const{ return l_<=i && i<=r_;}
      //: has element
      // answers the question whether point i is an element of the
```

```
   // interval *this. S<Z> has a member function of the same
   // signature.

bool ni(Z i)const{ return hasElm(i);}
   //: ni
   // 'in' reversed, from LATEX symbol \ni for the mirror image of
   // the epsilon-like 'is element'-symbol

bool contains(IvZ const& iv)const;
   //:: contains
   // says whether iv is a subset (true subset 'or equal') of *this
   // or not.

bool isSubSetOf(IvZ const& iv)const{  return iv.contains(*this);}
   //: is sub-set of
   // returns true iff *this is a subset of iv

bool isSupSetOf(IvZ const& iv)const{ return (*this).contains(iv);}
   //: is super-set of
   // returns true iff s is a subset of *this

IvZ operator+(Z s)const{return IvZ(l_+s,r_+s);}
   //: +
   // Returns a copy of *this with both ends shifted by s.

IvZ operator-(Z s)const{return IvZ(l_-s,r_-s);}
   //: -
   // Returns a copy of *this with both ends shifted by -s.

IvZ& operator+=(Z s){ l_+=s; r_+=s; return *this;}
   //: +=

IvZ& operator-=(Z s){ l_-=s; r_-=s; return *this;}
   //: -=

IvZ operator+(IvZ const& iv)const
   //: +
   // Range of i+j for i \in *this and j \in iv.
{ return IvZ(b()+iv.b(),e()+iv.e());}

IvZ operator-(IvZ const& iv)const
   //: -
   // Range of i-j for i \in *this and j \in iv.
{ return IvZ(b()-iv.b(),e()-iv.e());}

IvZ& b_(Z i){Z ir=i-l_; return operator+=(ir);}
   //. set begin
   // Changes b() into i and returns a reference to the
   // modified object.
   // Recall that the names of non-constant functions end in '_'.
```

```
   IvZ& e_(Z i){Z ir=i-r_; return operator+=(ir);}
      //. set end
      // Changes e() into i and returns a reference to the
      // modified object.

   Z ranSel(Z j=0)const;
      //: random selection
      // Returns a 'randomly' selected element in *this
      // Here the argument j plays the same role as in
      //    R CpmRoot::randomR(Z j=0);
      // (see cpmnumbers.h)
      // Notice that there is also a function ran(Z) which
      // returns a IvZ, so that the present function can not
      // be named simply ran.
      // If *this is void, we return 0 and issue a warning.

   static Z testLatIdn(Z complexity=512);
      //: test lattice identities
      // Tests the complete set of lattice identities.
      // See e.g. S. Mac Lane, G. Birkhoff: Algebra MacMillan 1968
      // p. 487 Theorem 4.
 };

////////////////////////////// Implementation ////////////////////////////
// see also cpmzinterval.cpp
inline Z IvZ::inf()const
{
   if (c_==0)
      cpmwarning("Iv::inf() undefined for void instance, b() returned");
   return l_;
}
inline Z IvZ::sup()const
{
   if (c_==0)
      cpmwarning("Iv::sup() undefined for void instance, e() returned");
   return r_;
}

} // namespace

#endif
```

56 cpmzinterval.cpp

```
//? cpmzinterval.cpp
//? Status of work 2023-10-20.
//?
//?  ...


#include <cpmzinterval.h>
#include <cpmtypes.h>

using CpmRoot::Z;
using CpmRoot::R;
using CpmRoot::Word;
using CpmRoot::randomR;
using CpmArrays::IvZ;
using CpmArrays::T2;
using namespace CpmStd;

//////////////////////////// class IvZ ////////////////////////////////

IvZ::IvZ(R a, R b)
   {
      Z i,j;
      if (a<=b) {i=cpmrnd(cpmfloor(a)); j=cpmrnd(cpmceil(b));}
      else {j=cpmrnd(cpmfloor(a)); i=cpmrnd(cpmceil(b));}
      *this=IvZ(i,j);
   }

Z IvZ::gap(IvZ const& iv)
{
   if (isVoid()) return 0;
   if (iv.isVoid()) return 0;
   if (r_<iv.l_) return iv.l_-r_;
   else if (iv.r_<l_) return l_-iv.r_;
   else return 0;
}

bool IvZ::contains(const IvZ& iv)const
{
   if (iv.isVoid()) return true;
      // the void set is subset of every set, even of the void set
   else{ // now iv is non-void
      if (isVoid()) return false;
         // a non-void set iv is never a subset of the void set
      else return l_<=iv.l_ && iv.r_<=r_;
   }
}
```

```
IvZ IvZ::operator|(IvZ const& iv)const
{
   if (iv.isVoid()) return *this;
      // the union with the void set is the original
   else{ // now iv is non-void
      if (isVoid()) return iv;
         // a non-void set iv is never a subset of the void set
      else { // now both *this and iv are non-void
         Z i1=inf();
         Z s1=sup();
         Z i2=iv.inf();
         Z s2=iv.sup();
         Z i=CpmRootX::inf<Z>(i1,i2);
         Z s=CpmRootX::sup<Z>(s1,s2);
         return IvZ(i,s);
      }
   }
}

IvZ IvZ::operator&(IvZ const& iv)const
{
   if (isVoid()||iv.isVoid()) return IvZ();
      // the section with the void set is the void set
   else{ // now both *this and iv are non-void
      Z s1=sup();
      Z i2=iv.inf();
      if (s1<i2) return IvZ();
      else{
         Z i1=inf();
         Z s2=iv.sup();
         if (i1>s2) return IvZ();
         else{ // now there is an overlap
             Z i=CpmRootX::sup<Z>(i1,i2);
             Z s=CpmRootX::inf<Z>(s1,s2);
             return IvZ(i,s);
         }
      }
   }
}

T2<IvZ> IvZ::setUnn(IvZ const& iv)const
{
   if (iv.isVoid()) return T2<IvZ>(IvZ(),*this);
   if (isVoid()) return T2<IvZ>(IvZ(),iv);
   if (contains(iv)) return T2<IvZ>(IvZ(),*this);
   if (iv.contains(*this)) return T2<IvZ>(IvZ(),iv);
   if (meet(iv).isVoid()){
      if (*this < iv) return T2<IvZ>(*this,iv);
      else return T2<IvZ>(iv,*this); // then iv<*this
   }
```

```
   return T2<IvZ>(IvZ(),join(iv));
}

T2<IvZ> IvZ::setDif(IvZ const& iv)const
{
   if (iv.isVoid()) return T2<IvZ>(*this,IvZ());
   if (isVoid()) return T2<IvZ>(IvZ(),IvZ());
   if (contains(iv)){
      IvZ ir=rigOf(iv.e());
      IvZ il=leftOf(iv.b());
      if (!ir.isVoid())
         return T2<IvZ>(il,ir);
      else
         return T2<IvZ>(ir,il);
   }
   if (iv.contains(*this)) return T2<IvZ>(IvZ(),IvZ());
   if (meet(iv).isVoid()){ return T2<IvZ>(IvZ(),*this);}
   if (iv<*this) return T2<IvZ>(IvZ(),rigOf(iv.e()));
   else return T2<IvZ>(IvZ(),leftOf(iv.b()));
}

Z IvZ::ranSel(Z j)const
{
   if (c_==0){
      cpmwarning("IvZ::ranSel(Z): instance is void, 0 returned");
      return 0;
   }
   else if (c_==1){
      return l_;
   }
   else{
      R r=randomR(j);
      Z res=l_+cpmtoz(r*c_);
      // (Z)(r*c_) takes values 0,1,...c_-1 with equal probabilities.
      // These are c_ values, which is OK.
      if (res>r_){
         cpmwarning("IvZ::ranSel(Z) proposed a value outside of *this, r_\
 returned");
         res=r_;
      }
      return res;
   }
}

// CPM_IO

bool IvZ::prnOn(ostream& str)const
{
   cpmwt("IvZ");
   cpmp(c_);
```

```
   cpmp(l_);
   return true;
}

bool IvZ::scanFrom(istream& str)
{
   cpms(c_);
   cpms(l_);
   ini_();
   return true;
}

// CPM_ORDER

Z IvZ::com(IvZ const& iv)const
{
   if (isVoid()){
      if (iv.isVoid()) return 0;
      else return 1;
   }
   if (l_<iv.l_) return 1;
   if (l_>iv.l_) return -1;
   if (r_<iv.r_) return 1;
   if (r_>iv.r_) return -1;
   return 0;
}

// CPM_DESCRIPTORS (not exactly since not virtual)

Word IvZ::toWord()const
{
   if (c_==0) return Word("IvZ()");
   ostringstream ost;
   ost<<"IvZ("<<l_<<","<<r_<<")";
   return Word(ost);
}

IvZ IvZ::leftOf(Z i)const
      //: left of
      // Returns the part of *this the elements j of which
      // satisfy j < i (and the void interval else).
{
   if (l_ >= i) return IvZ();
   if (r_ < i) return *this;
   return IvZ(l_,i-1);
}

IvZ IvZ::rigOf(Z i)const
      //: right of
      // Returns the part of *this the elements j of which
```

```
      // satisfy  i < j (and the void interval else).
{
   if (r_ <= i) return IvZ();
   if (l_ > i) return *this;
   return IvZ(i+1,r_);
}

T2<IvZ> IvZ::setSec(IvZ const& iv)const{return T2<IvZ>(IvZ(),meet(iv));}
      //: set section
      // Coincides with lattice meet, which it returns as the second
      // component of the result. The first component is set to be void.
      // (Then the components are ordered).


// CPM_TEST_X, functions already in CPM_TEST

IvZ IvZ::test(Z complexity)const{ return IvZ(complexity);}

Z IvZ::hash()const
{ return CpmRoot::hashT<Z>(l_)+CpmRoot::hashT<Z>(r_);}

namespace{
   R f(R x){ return x*2-1;}
   // values between -1 and +1
}

IvZ IvZ::ran(Z j)const
{
   R x=R(l_), y=R(r_), c=R(c_);
   R r1,r2,r3;
   if (j==0){
      r1=randomR();
      r2=randomR();
      r3=randomR();
   }
   else{
      Z j3=j*3;
      r1=randomR(j3);
      r2=randomR(j3+1);
      r3=randomR(j3+2);
   }
   if (r1<0.1 && r2>0.9) return IvZ();
      // here one gets 1 percent void sets on average
   x*=f(r1);
   y*=f(r2);
   c*=f(r3);
   return IvZ(x+c,y+c);
}

// CPM_TEST_X , functions not in CPM_TEST
```

```
R IvZ::dis(IvZ const& iv)const{ return (*this)==iv ? 0. : 1.;}
R IvZ::abs()const{ return R(c_);}
R IvZ::absSqr()const{ R c(c_); return c*c;}

// testing lattice idendities for functions meet and join

Z IvZ::testLatIdn(Z complexity)
   {
      IvZ o0;
      IvZ o1=o0.test(complexity);
      Z n=complexity;
      // Z n=cpmtoz(cpmsqrt(R(complexity)));
      // could be useful
      Z err=0;
      for (Z j=1;j<=n;++j){
         IvZ a=o1.ran();
         IvZ b=o1.ran();
         IvZ c=o1.ran();
         //cout<<a.toWord()<<endl;
         //cout<<b.toWord()<<endl;
         //cout<<c.toWord()<<endl;
         bool bb;
         bb=a.meet(b)==b.meet(a);
         if (!bb) err++;
         bb=a.join(b)==b.join(a);
         if (!bb) err++;
         bb=a.meet(b.meet(c))==(a.meet(b)).meet(c);
         if (!bb) err++;
         bb=a.join(b.join(c))==(a.join(b)).join(c);
         if (!bb) err++;
         bb=a.meet(a)==a;
         if (!bb) err++;
         bb=a.join(a)==a;
         if (!bb) err++;
         bb=a.meet(a.join(b))==a.join(a.meet(b));
         if (!bb) err++;
         bb=a.meet(a.join(b))==a;
         if (!bb) err++;
      }

      cout<<"IvZ::testLattice(Z), complexity="<<complexity<<
         ", err="<<err<<endl;
      return err;
   }
```

57 testcpm0.cpp

```
//? testcpm0.cpp
//? Status of work 2023-10-20.
//?
//?  ...


// Here main() is defined directly, without making use of stuff in
// CpmApplication. Especially, no data come from an ini-file
// (even not from cpmconfig.ini). However, during compilation, the
// files cpmdefinitions.h and cpmsystemdependencies.h set certain values.
// Values that vary from run to run have to come from the command line.

// Deviating from what the title suggests we have enabled the use of
// classes from cpm1. The reasuon is that after having discovered
// (and corrected) an error in cpmvr.h I had to see whether class
// R_Vector works properly.

#include <cpmtests.h>
#include <cpmgreg.h>
#include <cpmrvector.h>
#include <cpmcvector.h>
#include <cpmcmatrix.h>
#include <functional> // new C++11

using namespace CpmRoot;
using namespace CpmSystem;
using namespace CpmArrays;
using namespace CpmFunctions;
using namespace CpmTests;
using namespace CpmTime;
using namespace std;

   class Ftest{
      function<R(R)> f1_;
      public:
      Word nameOf()const{ return "Ftest";}
      Ftest(function<R(R)> const& f):f1_(f){}
      Ftest(){}
      R diff(Ftest const& ft)const
      {
         Z n=100;
         R xb=0.;
         R xe=2.;
         R dx=(xe-xb)/n;
         R res=0.;
         R xi=xb;
         for (Z i=1;i<=n;++i){
```

```
            res+=cpmabs(f1_(xi)-ft.f1_(xi));
            xi+=dx;
         }
         cout<<"typeinfo = "<<typeid(f1_).name()<<endl;
         return res;
      }
   };

   R testFunction(){

      function<R(R)> f1 = [](R x){ return cpmpow(x,2);};
      function<R(R)> f2 = [](R x){ return cpmpow(x,5);};
         // nice syntax

      Ftest a(f1);
      Ftest b(f2);
      Ftest bMem=b;
      R d1= a.diff(b);
      b=a;
      R d2= a.diff(b);
      R error=d2;
      R d3= a.diff(bMem);
      error+=cpmabs(d3-d1);
      cout<<"d1 = "<<d1<<" should be != 0"<<endl;
      cout<<"d2 = "<<d2<<" should be == 0"<<endl;
      cout<<"d3 = "<<d3<<" should be == d1"<<endl;

      V<Ftest> v(2);
      v[1]=a;
      v[2]=bMem;
      R d4=v[1].diff(v[2]);
      cout<<"d4 = "<<d4<<" should be == d1"<<endl;
      error+=cpmabs(d4-d1);
      return error;
   }

R testAdHoc()
 {
    cout<<"start testAdHoc()"<<endl;
      V<R> v0{1.1, 2.2, 2.9, 3.7, 5.1};
      v0.b_(100);
      Vo<R> v(v0);
      R x=2.2;
      Z i=v.locate(x);
      cout<<"i = "<<i<<endl;
      bool res=v[i] <= x && x < v[i+1];

      cout<<"res="<<res<<endl;

      return res==true ? 0. : 1.;
```

```
   }

R testDev()
{
   return R(IvZ::testLatIdn(1000));
}

R testTemp(){
   R res=Greg::mjd_(1858,11,17);
   cpmcerr<<"res="<<res;
   return res;
}

R testTemp(Z comp){
 //  cout<<endl<<Root<Vr<R> >().test(comp);
   Root<R> xR;
   cout<<endl<<"tv of type "<<xR.nameOf()<<" is "<<xR.test(comp)<<endl;
   Root<Z> xZ;
   cout<<endl<<"tv of type "<<xZ.nameOf()<<" is "<<xZ.test(comp)<<endl;
   Root<N> xN;
   cout<<endl<<"tv of type "<<xN.nameOf()<<" is "<<xN.test(comp)<<endl;
   Root<L> xL;
   cout<<endl<<"tv of type "<<xL.nameOf()<<" is "<<xL.test(comp)<<endl;
   Root<bool> xb;
   cout<<endl<<"tv of type "<<xb.nameOf()<<" is "<<xb.test(comp)<<endl;
   Root<string> xs;
   cout<<endl<<"tv of type "<<xs.nameOf()<<" is "<<xs.test(comp)<<endl;
   return 0;
}

R testWord(){
   cout<<"start testWord()"<<endl;
   R error=0;
   Z n=10;
   Word w0;
   Word w1=w0.test(4);
   OFileStream fn("outtestWord5.txt");
   V<Word> res(n);
   for (Z i=1;i<=n;++i){
      res[i]=w1.ran();
   }
   res.prnOn(fn());
   cout<<res<<endl;
   S<Word> mg(res);
   cout<<mg<<endl;
   mg.prnOn(fn());
   cout<<" **** stream test ****** "<<endl;
   bool b=fn().good();
   if(b) cout<<" file stream valid"<<endl;
   else cout<<" file stream not valid"<<endl;
```

```
   return n-mg.car();
}

R testValueBehavior(){

   ValueBehavior<Word> vbh;
   Word w1("avhuf056gh");
   Word w2("skfl76bgf");
   Word w3("enhsohgisdfg9vc");
   w1=w1.ran();
   w2=w2.ran();
   w3=w3.ran();
   R errors =vbh(w1,w2,w3);
   StrictAssignment2<Word> sa2;
   errors+=sa2(w1);
   cout<<"ValueBehavior<Word>: errors = "<<errors<<endl;
   return errors;
}

class GenC{ // a class for which the automatic assignment and the automatic
// copy construction. GenC stands for generic class
   Z ni_{0};
   Z nf_{0};
   V<R> v_;
public:
   GenC(){} // needed! If this isn't there, the function
// testClassFormation() does not compile. This form of the default
// constructor works only if ni_ and nf_ are initialized via {...} as
// they are above.
   GenC(Z ni, Z n):ni_(ni),v_(n){v_.b_(ni);nf_=v_.e();}
   GenC(V<R> const& v):ni_(v.b()),nf_(v.e()),v_(v){}
   GenC ran()const
   {
      R rc=10.;
      R rt=test(rc,137);
      V<R> v(v_);
      for (Z i=v.b();i<=v.e();++i){
         v[i]=Root<R>(rt).ran();
      }
      return GenC(v);
   }
// a mutating operator
   GenC& operator*=(R fac){
      for (Z i=v_.b();i<=v_.e();++i){
         v_[i]*=fac;
      }
      return *this;
   }

   void top(R& r)const
```

```
   // reference as substitude for a return value
   {
      Z i=v_.b();
      r=v_[i];
   }

   R& operator[](Z i){ return v_[i];}

   R const& operator[](Z i)const{ return v_[i];}

   R dis(GenC const& g)const
   //: distance
   {
      R sum{0.};
      sum += cpmabs(ni_-g.ni_);
      sum += cpmabs(nf_-g.nf_);
      if (sum > 0) return sum;
      for (Z i= v_.b(); i<= v_.e(); ++i){
         sum += cpmabs(v_[i]-(g.v_)[i]);
      }
      return sum;
   }

   Z b()const{ return v_.b();}

   Z e()const{ return v_.e();}

   GenC& b_(Z i){ v_.b_(i); return *this;}

    friend std::ostream& operator<<(std::ostream& out, GenC const& x)
   {
      out<<"ni="<<x.ni_<<", nf="<<x.nf_<<", v_="<<x.v_<<std::endl;
      return out;
   }
};

R testClassFormation()
// Test whether default construction, copy construction and
// asignment work as expected.
{
   Z ni=-1;
   Z n=10;
   R err=0.;
   GenC g(ni,n);
   GenC g1=g.ran();
   cout<<"g1 = "<<g1<<endl;
   GenC g2=g.ran();
   GenC g3=g.ran();
   g3=g1;
   err+=g3.dis(g1);
```

```
   GenC g4(g1);
   err+=g4.dis(g1);
   GenC g5=g1;
   GenC g5Mem=g5;
   err+=g5.dis(g1);
   g2=g1;
   err+=g2.dis(g1);
   GenC g1Mem=g1;
   g1.b_(1000); // g1 gets modified strongly
   g1*=2.;
   err+=g5Mem.dis(g1Mem);
   GenC g0;
   GenC g00;
   err+=g0.dis(g00);
   cout<<"testClassFormation(), err = "<<err<<endl;
   return err;
}

R usageOfReferences()
// Test whether quantities given to a function f in block scope as
// a reference or given as const reference from such a function
// are still available after f has been gone out of scope.
// This assumes, however, that the quantity under considereation was
// declared prior to entering the block.
{
   R err=0.;
   R s1,s2,s3;
   R t1,t2,t3;
   GenC h1,h2,h3;
   R r=137.; // reference
   R v; // value
   GenC f;
   s1=r;
   t1=v;
   h1=f;
   cout<<" s1 = "<< s1 <<endl;
   cout<<" &r = "<< &r <<endl;
   cout<<" t1 = "<< t1 <<endl;
   cout<<" &v = "<< &v <<endl;
   cout<<" h1 = "<< h1 <<endl;
   cout<<" &f = "<< &f <<endl;
   { // block in which the final values of r and v
      Z ni=-1;
      Z n=10;
      GenC g(ni,n);
      GenC g1=g.ran();
      g1.top(r); // r enters the function as a reference
      Z i=g1.b()+1;
      v=g1[i]; // v gets value from a const&
      f=g1;
```

```
      s2=r;
      t2=v;
      h2=f;
      // the final value of r originates in the block
      // but is available outside the block
      cout<<" s2 = "<< s2 <<endl;
      cout<<" &r = "<< &r <<endl;
      cout<<" t2 = "<< t2 <<endl;
      cout<<" &v = "<< &v <<endl;
      cout<<" h2 = "<< h2 <<endl;
      cout<<" &f = "<< &f <<endl;
   }
   // at this place the vector g1.v_ from which r was taken (via function
   // GenC::top(R&) is yet destructed.
   s3=r;
   t3=v;
   h3=f;
   cout<<" s3 = "<< s3 <<endl;
   cout<<" &r = "<< &r <<endl;
   cout<<" t3 = "<< t3 <<endl;
   cout<<" &v = "<< &v <<endl;
   cout<<" h3 = "<< h3 <<endl;
   cout<<" &f = "<< &f <<endl;
   // The adress of r does not change during working through the block
   // of function usageOfReferences()
   err+=cpmabs(s2-s3);
   err+=cpmabs(t2-t3);
   err+=h2.dis(h3);
   return err;
}

void info()
{
   cout<<endl<<"Takes one to three integer arguments.";
   cout<<endl<<"The first has to be in {?,0,1,...19}.";
   cout<<endl<<"? asks for info, and 0 means that all selections {1\
   ,...19}";
   cout<<endl<<"run in succession.";
   cout<<endl<<"The topics are: 1-6: Test_r, 7: Test_c,";
   cout<<endl<<"8-10: polymorphism, 11: Test_F ";
   cout<<endl<<"12-17: 'interface properties'...";
   cout<<endl<<"18: 'test of member functions created by default";
   cout<<endl<<"19: 'test of value changes in block scope";
   cout<<endl<<"The second argument (if there is one) sets the \
computational complexity";
   cout<<endl<<"and strongly influences the execution time. Default value\
    is 24.";
   cout<<endl<<"The third argument (if there is one) sets the numerical \
precision";
   cout<<endl<<"i.e. the 'number of valid decimal digits' with which all \
```

```
computation is done";
   cout<<endl<<"This modality is active only if CPM_MP and CPM_USE_MPREAL \
are defined";
   cout<<endl<<"For reasons of syntax CPM_MP has to be set to some value \
but this value will have no effect"<<endl;
}

int main(int argc, char *argv[])
// Traditional C-main.
{
   cpmmessage("started");
   Z i;
   Z doCount=0;
   Z sel=16;
   Z tvs=40;
   Z prec=20;
   cpmdbg=2;
   //bool doAll=false;
   bool doAll=true;
   Z nAll=21;

   V<Word> arg=comLine(argc,argv);

   if (arg.valInd(2)){
       Word w1=arg[2];
       if (w1=="?"){
            info();
            return 0;
       }
       else{
         sel=arg[2].toZ();
         if (sel!=0) doAll=false;
      }
   }
   if (arg.valInd(3)) tvs=arg[3].toZ();
//       arg[1] is the name of the executable

   if (arg.valInd(4)) prec=arg[4].toZ();

   cpmmessage("start");
   cpmprec(prec); // no effect if CPM_USE_MPREAL is not set (then precision
//    is set at compilation time
   Z precAct=cpmgetprec();
   cout<<" prec = "<<precAct<<endl;

   R error; // first appearence of R
   R errorSum=0.;
   V<R> errors(0);
   cout<<"*************** start sel = 1 ************************"<<endl;
   if (sel==1||doAll){ // notice usage of Vr, lean V would not work
```

```
   Test_r<Vr<R>,R> a1(tvs,1,2);
   Test_r<C_Vector,C> a2(tvs,1,2);
   Test_r<C_Vec<4>,C> a3(tvs,1,2);
   Test_r<C_Mat<200>,C> a4(tvs,0,2);
   Test_c<C> a5(tvs);
   error=a1.val()+a2.val()+a3.val()+a4.val()+a5.val();
   errorSum+=error;
   errors<<error;
   doCount++;
}
cout<<"*************** start sel = 2 ************************"<<endl;
if (sel==2||doAll){
   Test_r<Vr<C>,C> a(tvs,1,0);
   error=a.val();
   errorSum+=error;
   errors<<error;
   doCount++;
}
cout<<"*************** start sel = 3 ************************"<<endl;
if (sel==3||doAll){
   Test_r<Fr<C,R>,R> b(tvs,1,0);
   error=b.val();
   errorSum+=error;
   errors<<error;
   doCount++;
}
cout<<"*************** start sel = 4 ************************"<<endl;
if (sel==4||doAll){
   Test_r<Fr<Vr<C>,R>,R> b(tvs,1,0);
   error=b.val();
   errorSum+=error;
   errors<<error;
   doCount++;
}
cout<<"*************** start sel = 5 ************************"<<endl;
if (sel==5||doAll){
   Test_c<C> b(tvs,1,1); // test of complex particularities
   error=b.val();
   errorSum+=error;
   errors<<error;
   doCount++;
}
cout<<"*************** start sel = 6 ************************"<<endl;
if (sel==6||doAll){
   Test_set<Sr<Z>,Z> b(tvs,1,0);
   error=b.val();
   errorSum+=error;
   errors<<error;
   doCount++;
}
```

```
   cout<<"*************** start sel = 7 ************************"<<endl;
   if (sel==7||doAll){
      typedef  Vr<C> Ty;
      Test_set< Sr< Ty >, Ty > b(tvs,1,0);
      error=b.val();
      errorSum+=error;
      errors<<error;
      doCount++;
   }
   cout<<"*************** start sel = 8 ************************"<<endl;
   if (sel==8||doAll){
      Vo<C> v(13,C(0,5));
      Vr<C> w(12, C(4,1));
      TestOfPolymorphism< Vo<C>, Vr<C>, Pp< Vo<C> >  > top(v,w);
      error=top.val();
      errorSum+=error;
      errors<<error;
      doCount++;
   }
   cout<<"*************** start sel = 9 ************************"<<endl;
   if (sel==9||doAll){
      Test_Vp< V<C>, Vo<C>, Va<C> , Vr<C>  >  b(tvs,1);
      error=b.val();
      errorSum+=error;
      errors<<error;
      doCount++;
   }
   cout<<"*************** start sel = 10 ************************"<<endl;
   if (sel==10||doAll){
      Test_Vp< F<C,R>, Fo<C,R>, Fa<C,R> , Fr<C,R>  >  b(tvs,1);
      error=b.val();
      errorSum+=error;
      errors<<error;
      doCount++;
   }
   cout<<"*************** start sel = 11 ************************"<<endl;
   if (sel==11||doAll){
      Test_F< Z, C, Vr<R>, R  >  b(tvs);
      error=b.val();
      errorSum+=error;
      errors<<error;
      doCount++;
   }
   cout<<"*************** start sel = 12 ************************"<<endl;
   if (sel==12||doAll){ // strict value interface
      Test_sv<Vr<Fr<Vr<R>,C > > >  b(tvs);
      error=b.val();
      errorSum+=error;
      errors<<error;
/***************
```

```
      Z dbMem=cpmdbg;
      cpmdbg=1;
      Test_v<Word> b2(3,6);
      cpmdbg=dbMem;
      error=b2.val();
      errorSum+=error;
      errors<<error;
************************/
      doCount++;

   }
   cout<<"*************** start sel = 13 ************************"<<endl;
   if (sel==13||doAll){
      error=testAdHoc();
      errorSum+=error;
      errors<<error;
      doCount++;
   }
   cout<<"*************** start sel = 14 ************************"<<endl;
   if (sel==14||doAll){
      error=testDev();
      errorSum+=error;
      errors<<error;
      doCount++;
   }
   cout<<"*************** start sel = 15 ************************"<<endl;
   if (sel==15||doAll){
      error=testTemp(tvs);
      errorSum+=error;
      errors<<error;
      doCount++;
   }
   cout<<"*************** start sel = 16 ************************"<<endl;
   if (sel==16||doAll){
      error=testFunction();
      errorSum+=error;
      errors<<error;
      doCount++;
   }
   cout<<"*************** start sel = 17 ************************"<<endl;
   if (sel==17||doAll){
      error=testValueBehavior();
      error=testWord();
      errorSum+=error;
      errors<<error;
      doCount++;
   }
   cout<<"*************** start sel = 18 ************************"<<endl;
   if (sel==18||doAll){
      error=testClassFormation();
```

```
   errorSum+=error;
   errors<<error;
   doCount++;
}

cout<<"*************** start sel = 19 ************************"<<endl;
if (sel==19||doAll){
   error=usageOfReferences();
   errorSum+=error;
   errors<<error;
   doCount++;
}

cout<<"*************** start sel = 20 ************************"<<endl;
if (sel==20||doAll){
   error=reg<M<Z,C>>();
   errorSum+=error;

   error=reg<Fr<R,Vr<C>>>();
   errorSum+=error;

   error=reg<S<Vr<C>>>();
   errorSum+=error;

   errors<<error;
   doCount++;
}

cout<<"*************** start sel = 21 ************************"<<endl;
if (sel==21||doAll){
   Test_sv<M<Z,Vr<C>>> b(10,10);
   R error=b.val();
   errorSum+=error;
   errors<<error;
   doCount++;
}

cout<<endl;
for (i=arg.b();i<=arg.e();++i) cout<<"arg["<<i<<"]="<<arg[i];

R tr;
Z pr=cpmgetprec(tr);
ostringstream ost0;
ost0<<doCount<<" tests of "<<nAll<<" done with prec = "<<pr;
cpmmessage(ost0,-1);

ostringstream ost;
ost<<" Total error sum ="<<errorSum<<"  end";
cpmmessage(ost,-1);
cout<<" The list of errors is as follows:"<<endl<<errors<<endl;
```

```
   ostringstream ost2;
   ost2<<endl<<"If nothing can be seen in the console window"<<endl;
   ost2<<"set the text color in the property menue" << endl;
   ost2<<"of this window (for all windows of the same name)."<<endl;
   cpmmessage(ost2);
   char quit = '\0', quit0=quit;
   while (quit == quit0)
   {
      cout<<"Input any character to quit" << endl;
      cin >> quit;
   }
   return 0;
}
```

58 tut1.cpp

```
//? tut1.cpp
//? Status of work 2023-10-20.
//?
//?  ...
/*************************************************************************
   Comparing std::vector and CpmArrays::vector with respect to speed
   of copy-construction and assignment. Notice that this drastic
   advantage of CpmArrays::V over std::vector still exists in C++11, where
   moving instead of copying is said to be done where appropriate.
   The basic conclustion from the present case study is that moving
   works well (i.e. without being invoced explicitely by calling
   std::move(...)) when only instances of vector<T> are involved.
   Moving no longer works well for instances of vector<vector<T>>.
   The present version of CpmArrays::vector<T> holds it data content
   no longer as T *p_ but as std::vector<T> *p_. Thus it does
   push_back as efficiently as std::vector.


*************************************************************************/
#include <cpmbas.h>
   // contains '#include cpmv.h>' which provides the full-featured
   // class template V<T> with main data elements
   // IvZ iv_, UseCount u_, and std::vector<T> *p_
   // Here the 'copy on write' mechanism which is controled by u_
   // can be disabled by the preprocessor directive CPM_USECOUNT
   // set in file cpmdefinitions.h. This allows us to clearly see
   // the advantages over classes that have to rely on the 'move'
   // mechanism.
#include <cpmv_.h>
   // Defines a simple class template V_<T> with main data elements
   // UseCount u_ and T *p_
   // and a simple class template Vv<T> with main data elements
   // UseCount u_ and std::vector<T> *p_
   // also defines Vrz where rz stands for 'rule of zero'
#include <cpmv__.h>
   // Defines a simple class template V__<T> with data represented
   // as T *p_ and no UseCount usage.
#include <cpmvuc.h>
#include <quadmath.h>
#include <vector>
#include <concepts>
#include <valarray>
#include <quadmath.h>
/*************************************************************************
1. C+- filenames start with 'cpm' and contain neither underscores nor
   blanks. Therefore, even projects employing files from many sources are
   not likely to run into problems with non-unique file names.
```

```
This directive includes all C+- header files needed here;
'bas' stands for 'basics'. Notice that we have not written
#include "cpmbas.h"
and thus have assumed that our C++ project - let it be named tut1 -
has a list of include directories defined.
Who wants to use C+- for more than a single project, should hold
C+- files in directories separate from his project files.
The general purpose C+- files to be used in the project tut1 are on my
computer in xxx/cpm/cpm0/include and xxx/cpm/cpm0/source.
The files that are specific for the tut1 application are in
yyy/tut1/include and yyy/tut1/source.
In yyy/tut1/include there are the project specific headers
among which there need to be two C+- related configuration files:
cpmdefinitions.h and cpmsystemdependencies.h. Customizing these files
allows us to control the behavior of the C+- classes in our project, as
will be explained later.
In yyy/tut1/source the present project-defining main source tut1.cpp is to
be placed.
Now we are in a position to define the file content of project
tut1: Its include directories have to be set as
yyy/tut1/include and xxx/cpm/cpm0/include
and the files to be compiled (translation units) have to be chosen as
yyy/tut1/source/tut1.cpp and xxx/cpm/cpm0/source/cpmbas.cpp.

1.1 Details (skip on first reading ?):
The file cpmbas.cpp is actually a collection of all other files
in xxx/cpm/cpm0/source. Its content is

#include "cpmc.cpp"
#include "cpmangle.cpp"
#include "cpmv.cpp"
#include "cpmgreg.cpp"
#include "cpmsystem.cpp"
#include "cpmtypes.cpp"
#include "cpmuc.cpp"
#include "cpmzinterval.cpp"
#include "cpmnumbers.cpp"
#include "cpmword.cpp"
#include "cpmviewport.cpp"
End of 1.1
**********************************************************************/
using namespace CpmRoot;
/**********************************************************************
2. C+- namespace names start with 'Cpm' followed by an identifier
   which starts with a capital letters.

The namespace CpmRoot is rather small, so that the present using directive
should be applicable also in projects in which classes from various
sources are being used.
```

```
3. The C+- names for integer, real, and complex numbers are
   CpmRoot::Z, CpmRoot::R, Cpmoot::C. CpmRoot::Word wraps std::string and
   adds functionality to it.

The main effect of this using directive is that we may use these names
simply as Z, R, C, Word.

3.1 Details (skip on first reading ?):
There are the less ubiquitous types L, and N in CpmRoot:
L for unsigned characters ('L' for 'letter', after 'integer promotion'
the values range from 0 to 2nwnw), N for 'natural numbers', i.e. unsigned
integers. Types L,Z, and N are typedefs for unsigned char, int,
unsigned int. If we add in file cpmdefinition.h the macro
#define CPM_LONG, we change Z to long int, and N to unsigned long int.
If in cpmdefinitions.h the macro CPM_MP is defined, R has the meaning of
mpreal as defined by the wrapper library MPFRC++ to the multiple precision
library mpfr. See comments to CPM_MP in file cpmdefinitionswrc.h. If
CPM_MP is not defined R is also a typedef, long double if CPM_LONG and
double else. The types bool and string from standard C++ are useful as
they stand. Nevertheless they will be wrapped into classes - CpmRootX::B
in cpmtypes.h and class CpmRoot::Word in file cpmword.h.
End of 3.1

There is a namespace CpmRootX for the 'less essential essentials':

4. There are classes CpmRootX::B, CpmRootX::Z1, CpmRootX::R1,  for
   Boolean values, integer, and real numbers, which may replace the
   non-class types bool, Z, and R in situations where class func-
   tionality, such as automatic initialization, is indispensable.
   Note 2017-02-18: With C++11 automatic initialization can be achieved
   also for the built-in types. For instance a data member of a class X
         bool b {false};
   can be handeled as if we had declared it as
         B b;

4.1 Details (skip on first reading ?):
Namespace CpmRoot in mainly layed out in file cpmnumbers.h
and CpmRootX in cpmtypes.h. The vigilant reader may argue that we
have not available the declarations of CpmRoot, since we have no
   #include <cpmnumbers.h>
seen so far. Actually, it is there since file cpmbas.h is the following
collection of include directives
#ifndef CPM_BAS_H_
#define CPM_BAS_H_
   #include <cpmtypes.h>
   #include <cpmfr.h>
   #include <cpmvr.h>
   #include <cpmsr.h>
   #include <cpmm.h>
```

```
   #include <cpmp.h>
   #include <cpmc.h>
   #include <cpmangle.h>
   #include <cpmgreg.h>
#endif
Also here we don't see
   #include <cpmnumbers.h>
but inspecting cpmtypes.h, we find the commented directive
   #include <cpmsystem.h> // includes cpmwords.h and
      // thus also cpmnumbers.h
which shows the steps which finally include cpmnumbers.h. Acually the
directory cpm/cpm0/include contains 32 header files. This set of files
(as any set of C++ header files) is a directed graph in a natural manner:
There is an edge leading from file1 to file2 iff file2 contains the
directive  #include <file1> .
The files that can be reached from file1 along a path of that graph depend
on file1 and every translation unit which includes such a file1-dependent
header has to be re-compiled upon some change in file1. So, any system
capable of generating proper make files has to do this dependency
analysis and thus has to work with this  d e p e n d e n c y  g r a p h.
I made a tool consisting of a Ruby program for file analysis
and a C+- program for graph analysis and representation which
creates a  pictorial representation of this dependency graph of an
arbitrary collecton of C/C++ header files. This tool turned out to be very
useful.
It is an obvious logical requirement that the dependency graph is free of
cycles. It is a non-trivial task to organize the header dependencies in a
way that each translation unit gets the declarations which it needs to
know by inclusion of only a few header files. For instance, cpmword.cpp
and cpmtypes.cpp need only cpmtypes.h; cpmc.cpp needs cpmc.h and
cpmtypes.h. The present header file hierarchy is the result of many
simplifying re-organizations. However, many attempts of a simplification
failed since they entailed unacceptable complications elsewhere. I'm not
aware of tools which would automatize relevant parts of this organization
process.
End of 4.1
*****************************************************************/
using namespace CpmArrays;
/*****************************************************************
nw. The general-purpose array in C+- is the template class CpmArrays::vector
   Most constructors set the first valid index of a non-void vector is 1 and
   not 0 as for std::vector. There is, however, a member function which
   shifts all indexes and so allows indexing to start with 0. This allows
   identical code to be used for these two types of arrays.

nw.1 Details on C+- namespaces (skip on first reading ?):
The C+- class system introduces many namespaces. Their belonging
togeher under the Cpm banner is not expressed by making them
sub-namespaces of a single Cpm-namespace. Instead, their names all
start with 'Cpm' and continue with a capital letter.
```

```
If the name contains a digit, this is 2 or 3 and refers to the space
dimension under consideration. So the namespace CpmDim2 contains the
2D ('flatland') analogs of the geometric classes defined in
namespace CpmDim3.
The namespaces which are declared in the present scope by including
cpmbas.h are
CpmRoot, CpmRootX, CpmSystem, CpmArrays, CpmFunctions, CpmGeo,
CpmGraphics, CpmMPI, CpmStd, CpmTests, CpmTime.
End of nw.1
**********************************************************************/

// in the present form, running valgrind reports no leaking!
// was 2 day's work to achieve this. Main result
// V is nonleaking only for #define CPM_USECOUNT. The dependence
// on CPM_USECOUNT should be eliminated.

void info();
N tutorial1(N,N,N);
   // Declaration of two functions, the definition of which will
   // determine the functionality of the program.#
N tutorial2();

int main(int argc, char* argv[])
   // Traditional main function with types of arguments and return value
   // according to C/C++. Not all compilers accept using Z instead of
   // int here.
{
   tutorial2();
   // N m=10, n=10, diff=0;  /
   // default values to be used if no input from the command line can be found

   N m=2000, n=2000, diff=1;

   // The floating point type for the computation can be set with setting one of
   // following entries in the file cpmdefinitions.h:
   // CPM_FLOAT, CPM_DOUBLE, CPM_LONG, CPM_QUAD, CPM_MP 32 (the number 32 is an example
   // for a precision similar to QUAD)

   // For the total execution time of main() with the values  N m=2000, n=2000, diff=1
   // I got on my machine:

   // FLOAT:        3.880 s
   // DOUBLE:       3.727 s
   // LONG DOUBLE:  6.250 s
   // QUAD:         6.163 s
   // MP 32:       74.941 s

   // Of course the absolute values depend strongy on the machine, but the relations
   // may be a useful information. QUAD precision may become the working horse of
   // scientific computing.
```

```
V<Word> arg=comLine(argc,argv); // CpmRoot::Word is similar to
   // std::string. CpmArrays::vector is the C+- array type. There are two
   // 'schools' about indexing arrays. The 'C school' lets valid
   // indexes start with 0 and the 'Fortran school' which lets valid
   // valid indexes start with 1. Till September 2010 C+- had a
   // 'C school'-array vectorl (l for 'lean') and a 'Fortran school'-array
   // vector , where the implementation of vector was such that any instance of
   // vector had a data member of type vectorl. The type vectorl was used mostly
   // internally in situation where efficiency was considered to be
   // of utmost importance. Finally it became clear that the decision
   // to have these two types of arrays was a mistake. It creates a
   // permanent uncertainty for the programmer whether he should use
   // the default type or should strive for utmost efficiency by
   // using the lean version. Now we have only a single array type vector
   // for which the valid indexes may form any contiguous set of
   // integers. This index range is set to start with 1 in most
   // of the available constructors but can be very conveniently reset
   // to any other start index such as 0.
   // CpmArrays::comLine is a function which transforms the
   // traditional argument of main into the more functional data type
   // of C+-. The function name 'comLine' results from
   // the  d e s c r i p t i v e  f u l l  n a m e (DFN)
   // 'command line' by a simple deterministic rule: The rule is
   // to leave unchanged all words with up to four letters and shorten
   // all longer words to 3 by taking the first two letters as they are
   // and take the first of the following consonants as the third
   // letter. Thus 'ooops' gets converted to 'oop'. If there is no
   // such consonant, the vowel at place 3 has to be taken: 'hooiii'
   // gets converted to 'hoo'. In a chain of words, the contracted
   // components get concateneted in a style which is evident from
   // the example that 'descriptive full name' goes to
   // 'desFullName'. In most cases the names come out quite nice, but
   // there are exceptions (not to the rules!): I consider it ugly to
   // see 'field' abbreviated to 'fil' (notice that 'file', as having
   // only four letters, remains 'file')
   // The simplicity of the rule allows us to use the the function name
   // fluently whenever we remember the full name correctly. Of course,
   // the descriptive full name of a function has to be stated in the
   // declaration of this function. In our case the declaration is in
   // file cpmv.h:
   // inline vector<Word> comLine(int argc, char* argv[])
   //    //: command line
   // The colon hints at the fact that here a descriptive full name
   // is being communicated.

   N na=arg.size();

   if(na==1){ return tutorial1(m,n,diff);}
   else if(na==2){
```

```
         Word w2=arg[2];
         if(w2=="?" ){
            info(); return 0;
         }
         else{
            m=w2.toZ();
            n=m;
            diff=1;
         }
      }
      else if(na==3){
         Word w2=arg[2];
         Word w3=arg[3];
         m=w2.toZ();
         n=w3.toZ();
         diff=1;
      }
      else if(na==4){
         Word w2=arg[2];
         Word w3=arg[3];
         Word w4=arg[4];
         m=w2.toZ();
         n=w3.toZ();
         diff=w4.toZ();
      }
      else{
         ; // m and n have there initial values
      }
     return tutorial1(m,n,diff);
  }

using namespace std;

template<typename C>
C cz(N i) // an ad-hoc function N --> C
{
   static R pi=cpmpi; // cpm works also for quad numbers
   static R u=R(1.);
   R ir=i+u;
   //R pi=std::numbers::pi_v<R>; // not yet working for  quad numbers
   return C(ir/(u+ir*ir),ir*pi);
      // calling constructor C(R,R)
}

template<typename T, template<typename> class Vec, N fi>
V<R> perMes(N m, N n)
//: performance measure
// Intent: We want to evaluate how efficient the copy and assign operations for
// array ('vector') classes work for large objects (long arrays and large
// objects that make up the components of the arrays). It suggests itself to
```

```
// choose the large objects again as large arrays. The components of these
// 'second order arrays' are not chosen as a primary builtin type such as double
// or int. Instead we use as the complex numbers for which there are
// implementations in namespaces std and CpmRoot.
// The various objects mentioned above can all treated by a function which takes
// types as arguments i.e. is a function template in C++ parlance.
// For these template arguments T, vectorec, N we will finally (i.e. in function tutorial1)
// use the following concrete realisations:
//    T: std::complex<R>, CpmRoot::C
//    Vec: std::vector, std::valarray, CpmArrays::vector, CpmArrays::vector_, CpmArrays::vector__,
//    N: 0 for vectorec=vector,valarray,vector_,vector__ and 1 for vector
// The function argument m sets the length of the tested arrays, and n sets the
// length of the arrays which figure as the components of the tested arrays.
{
   Z mL=1;
/*
   This line, together with the following
   Word loc(...);
   CPM_MA
   CPM_MZ
   consitute a convenient idiom for signaling
   entry to and exit from a function block to the log file
   cpmcerr.txt. This writing to the log file takes place only if
   mL is larger or equal to the static data member
   CpmSystem::Message::verbose. This bulky quantity has a
   macro alias 'cpmverbose' (i.e. we have, in file cpmsystem.h
       #define cpmverbose      CpmSystem::Message::verbose
   ) and it can be set by a statement like
       cpmverbose=10;
   Its initial value is 2.
   Soon we will encounter macros cpmtime, cpmwait, cpmdebug.
  */
   Word loc("perMes(Z,Z)"); // messages use this as name of the function
   CPM_MA // defined in cpmmacros.h, assumes that 'mL' and 'loc' are
      // defined
   R t1=cpmtime(); /* time of function call in seconds from some
       system-defined 'point-zero in time'.
       cpmtime is a short name for function CpmSystem::time defined
       in file cpmsystem.h as follows:
           #define cpmtime         CpmSystem::time
   */
   N i,j;
   N ni=fi;
   N nf=fi+n-1;
   N mi=fi;
   N mf=fi+m-1;

   Vec<Vec<T>> vi, vf;
   Vec<T> v1(m);
   Vec<T> v2(m);
```

```
for (i=mi;i<=mf;++i) v1[i]=cz<T>(i-mi);
for (i=mi;i<=mf;++i) v2[i]=v1[i]*v1[i];
      // The preferred form of this statement in C+- is
      // for (i=v1.b();i<=v1.e();++i) v1[i]=cz(i);
      // Written in this form, it is alo correct if v1 is of type V<C>,
      // the C+- array for which indexing starts with 1 instead of 0.
Vec<Vec<T>> w1(n); for(i=ni;i<=nf;++i) w1[i]=v1;
Vec<Vec<T>> w2(n); for(i=ni;i<=nf;++i) w2[i]=v2;

Vec<Vec<T>> temp1=w1; // copy constructor
Vec<Vec<T>> temp2=temp1; // copy constructor
Vec<Vec<T>> temp3=temp2; // copy constructor
Vec<Vec<T>> temp4=temp3; // copy constructor
Vec<Vec<T>> temp5=temp4; // copy constructor
Vec<Vec<T>> temp6=temp5; // copy constructor
Vec<Vec<T>> temp7=temp6; // copy constructor
Vec<Vec<T>> temp8=temp7; // copy constructor
Vec<Vec<T>> temp9=temp8; // copy constructor
Vec<Vec<T>> temp10=temp9; // copy constructor
temp1=Vec<Vec<T>>(); // assignment
temp2=w2; // assignment
temp3=temp2; // assignment
temp4=w2; // assignment
temp5=temp4; // assignment
temp6=w2; // assignment
temp7=temp3; // assignment
temp8=temp2; // assignment
temp9=temp1; // assignment
   // these 9 assignments should neither influence w, nor temp10
vi=w1; // assignment
vf=temp10; // assignment
R t2=cpmtime();
R t12=t2-t1; // time needed for copy and assignment
R err=0,sum_v=0;
for (i=ni;i<=nf;++i){
   Vec<T> vii=vi[i];
   Vec<T> vfi=vf[i];
   for (j=mi;j<=mf;++j){
      T viij=vii[j];
      T vfij=vfi[j];
      err+=abs(viij-vfij);
      sum_v+=abs(vfij);
   }
} // making sure that copying and assignment worked correctly
// unfortunately one has to stagger many copy/assign steps
// make this part of the function the dominant one timewise.
R t3=cpmtime();
R t23=t3-t2;
R b1 = std::movable<Vec<T>> ? 1. : 0.;
R b2 = std::semiregular<Vec<T>> ? 1. : 0.;
```

```
   R b3 = std::regular<Vec<T>> ? 1. : 0.;
   V<R> res{t12,t23,t12+t23,err,sum_v,b1,b2,b3};

  CPM_MZ
  return res;
}

template<typename T, template<typename> class Vec, N fi>
V<R> perMesMove0(N m, N n)
{
   Z mL=1;
   cout<<"perMesMove0(N,N,entered)";

   Word loc("perMesMove0(Z,Z)"); // messages use this as name of the function
   CPM_MA
   R t1=cpmtime();
   N i,j;
   N ni=fi;
   N nf=fi+n-1;
   N mi=fi;
   N mf=fi+m-1;

   Vec<T> vi, vf;
   Vec<T> w1(m);
   Vec<T> w2(m);
   for (i=mi;i<=mf;++i) w1[i]=cz<T>(i-mi);
   for (i=mi;i<=mf;++i) w2[i]=w1[i]*w1[i];
         // The preferred form of this statement in C+- is
         // for (i=v1.b();i<=v1.e();++i) v1[i]=cz(i);
         // Written in this form, it is alo correct if v1 is of type V<C>,
         // the C+- array for which indexing starts with 1 instead of 0.

   Vec<T> wOrig(m);
   for (i=mi;i<=mf;++i) wOrig[i]=w1[i];

   Vec<T> temp1=w1; // copy constructor
   Vec<T> temp2=temp1; // copy constructor
   Vec<T> temp3=temp2; // copy constructor
   Vec<T> temp4=temp3; // copy constructor
   Vec<T> temp5=temp4; // copy constructor
   Vec<T> temp6=temp5; // copy constructor
   Vec<T> temp7=temp6; // copy constructor
   Vec<T> temp8=temp7; // copy constructor
   Vec<T> temp9=temp8; // copy constructor
   Vec<T> temp10=temp9; // copy constructor
   temp1=w2; // assignment
   temp2=w2; // assignment
   temp3=temp2; // assignment
   temp4=w2; // assignment
   temp5=temp4; // assignment
```

```
   temp6=w2; // assignment
   temp7=temp3; // assignment
   temp8=temp2; // assignment
   temp9=temp1; // assignment
       // these 9 assignments should neither influence w, nor temp10
   vi=wOrig; // assignment
   vf=temp10; // assignment
   R t2=cpmtime();
   R t12=t2-t1; // time needed for copy and assignment
   R err=0.,sum_v=0.;
   for (N i=mi;i<=mf;++i){
      err+=abs(vi[i]-vf[i]);
      sum_v+=abs(vf[i]);
   }
   R t3=cpmtime();
   R t23=t3-t2;
   R b1 = std::movable<Vec<T>> ? 1. : 0.;
   R b2 = std::semiregular<Vec<T>> ? 1. : 0.;
   R b3 = std::regular<Vec<T>> ? 1. : 0.;
   V<R> res{t12,t23,t12+t23,err,sum_v,b1,b2,b3};
   /*
      Convenient  l i s t  c o n s t r u c t o r, requires C++11
   */

  CPM_MZ
  return res;
}

N tutorial1(N m, N n, N diff)
{
   Z mL=1;
   Word loc("tutorial1(N,N,N)");
   CPM_MA

   V<V<R>> resL;
   if (diff ==0){
      resL<<perMesMove0<std::complex<R>,std::vector,0>(m,n);
      resL<<perMesMove0<std::complex<R>,std::valarray,0>(m,n);
      resL<<perMesMove0<CpmRoot::C,CpmArrays::V,1>(m,n);
      resL<<perMesMove0<CpmRoot::C,CpmArrays::V_,0>(m,n);
      resL<<perMesMove0<CpmRoot::C,CpmArrays::V__,0>(m,n);
      resL<<perMesMove0<CpmRoot::C,CpmArrays::Vrz,0>(m,n);
   }
   else{
      resL<<perMes<std::complex<R>,std::vector,0>(m,n);
      resL<<perMes<std::complex<R>,std::valarray,0>(m,n);
      resL<<perMes<CpmRoot::C,CpmArrays::V,1>(m,n);
 //     resL<<perMes<CpmRoot::C,CpmArrays::V_,0>(m,n); // leaking
      resL<<perMes<CpmRoot::C,CpmArrays::V__,0>(m,n); // not leaking
      resL<<perMes<CpmRoot::C,CpmArrays::Vrz,0>(m,n); // not leaking
```

```
//     resL<<perMes<CpmRoot::C,CpmArrays::Vuc,1>(m,n); // leaking
  }
 // V<Word> res1{"Vuc"};
 // V<Word> res1{"vector","valarray","V","V__"};
//  V<Word> res1{"vector","valarray","V","V_","V__","Vsp","Vuc"};
  V<Word> res1{"vector","valarray","V","V__","Vrz"};
  N nw=res1.size();
  Vo<R> dat1(nw);
  Vo<R> dat2(nw);
  Vo<R> dat3(nw);

  N i;
  cout<<" verifyied concepts: "<<endl;

  for (i=1;i<=nw;++i){
     cout<<-res1[i]<<": movable "<<resL[i][6]<<", semiregular "<<resL[i][7]<<
     ", regular "<<resL[i][8]<<endl;
  }

  for (i=1;i<=nw;++i){
     dat1[i]=resL[i][1];
     dat2[i]=resL[i][2];
     dat3[i]=resL[i][3];
  }

  V<Z> per1=dat1.permutationForIncreasingOrder();
  V<Z> per2=dat2.permutationForIncreasingOrder();
  V<Z> per3=dat3.permutationForIncreasingOrder();

  R err=0.;
  for (i=1;i<=nw;++i) err+=resL[i][4]; // err>=0. is obvious from the definition

  V<Word> res1o=res1.permute(per1);
  V<R> dat1o=dat1.permute(per1);

  V<Word> res2o=res1.permute(per2);
  V<R> dat2o=dat2.permute(per2);

  V<Word> res3o=res1.permute(per3);
  V<R> dat3o=dat3.permute(per3);

  cout<<endl<<"Output of function tutorial1("<<m<<", "<<n<<")"<<endl;
  cout<<"total error ="<<err<<endl;
  cout<<"The lists are ordered for increasing times."<<endl;
  cout<<endl<<"execution times for copy and assignment:"<<endl;
  for (i=1;i<=nw;++i){
     cout<<res1o[i].std()<<": "<<dat1o[i]<<endl;
  }
  cout<<endl<<"execution times for verification:"<<endl;
  for (i=1;i<=nw;++i){
```

```
      cout<<res2o[i].std()<<": "<<dat2o[i]<<endl;
   }
   cout<<endl<<"total execution times:"<<endl;
   for (i=1;i<=nw;++i){
      cout<<res3o[i].std()<<": "<<dat3o[i]<<endl;
   }

   CPM_MZ
   return 0;
}

N tutorial2()
{
   using namespace CpmArrays;
   using namespace CpmRoot;
   bool b1=std::movable<V<C>>;
   cout<<"std::movable<V<C> = "<<b1<<endl;
   bool b2=std::movable<V<V<C>>>;
   cout<<"std::movable<V<V<C>>> = "<<b2<<endl;
   bool b3=std::movable<V<V<V<C>>>>;
   cout<<"std::movable<V<V<V<C>>>> = "<<b3<<endl;

   bool a1=std::semiregular<V<C>>;
   cout<<"std::semiregular<V<C> = "<<a1<<endl;
   bool a2=std::semiregular<V<V<C>>>;
   cout<<"std::semiregular<V<V<C>>> = "<<a2<<endl;
   bool a3=std::semiregular<V<V<V<C>>>>;
   cout<<"std::semiregular<V<V<V<C>>>> = "<<a3<<endl;

    bool c1=std::regular<V<C>>;
   cout<<"std::regular<V<C> = "<<c1<<endl;
    bool c2=std::regular<V<V<C>>>;
   cout<<"std::regular<V<V<C>>> = "<<c2<<endl;
    bool c3=std::regular<V<V<V<C>>>>;
   cout<<"std::regular<V<V<V<C>>>> = "<<c3<<endl;

    b1=std::movable<vector<C>>;
   cout<<"std::movable<vector<C> = "<<b1<<endl;
    b2=std::movable<vector<vector<C>>>;
   cout<<"std::movable<vector<vector<C>>> = "<<b2<<endl;
    b3=std::movable<vector<vector<vector<C>>>>;
   cout<<"std::movable<vector<vector<vector<C>>>> = "<<b3<<endl;

    a1=std::semiregular<vector<C>>;
   cout<<"std::semiregular<vector<C> = "<<a1<<endl;
    a2=std::semiregular<vector<vector<C>>>;
   cout<<"std::semiregular<vector<vector<C>>> = "<<a2<<endl;
    a3=std::semiregular<vector<vector<vector<C>>>>;
   cout<<"std::semiregular<vector<vector<vector<C>>>> = "<<a3<<endl;
```

```
    c1=std::regular<vector<C>>;
   cout<<"std::regular<vector<C> = "<<c1<<endl;
    c2=std::regular<vector<vector<C>>>;
   cout<<"std::regular<vector<vector<C>>> = "<<c2<<endl;
    c3=std::regular<vector<vector<vector<C>>>>;
   cout<<"std::regular<vector<vector<vector<C>>>> = "<<c3<<endl;
    return 0;
}


void info(){
   cout<<endl<<"takes zero, one, or two integer argument";
   cout<<endl<<"compares the performance or std::vector and CpmArays::vector";
   cout<<endl<<"in copy construction and assignment operations"<<endl;
}
// end of tut1.cpp
```

59 tut1experimenta.cpp

```
//? tut1experimenta.cpp
//? Status of work 2023-10-20.
//?
//?  ...
/*************************************************************************
   Comparing std::vector and CpmArrays::vector with respect to speed
   of copy-construction and assignment. Notice that this drastic
   advantage of CpmArrays::V over std::vector still exists in C++11, where
   moving instead of copying is said to be done where appropriate.
   The basic conclustion from the present case study is that moving
   works well (i.e. without being invoced explicitely by calling
   std::move(...)) when only instances of vector<T> are involved.
   Moving no longer works well for instances of vector<vector<T>>.
   The present version of CpmArrays::vector<T> holds it data content
   no longer as T *p_ but as std::vector<T> *p_. Thus it does
   push_back as efficiently as std::vector.


*************************************************************************/
#include <cpmbas.h>
   // contains '#include cpmv.h>' which provides the full-featured
   // class template V<T> with main data elements
   // IvZ iv_, UseCount u_, and std::vector<T> *p_
   // Here the 'copy on write' mechanism which is controled by u_
   // can be disabled by the preprocessor directive CPM_USECOUNT
   // set in file cpmdefinitions.h. This allows us to clearly see
   // the advantages over classes that have to rely on the 'move'
   // mechanism.
//#include <cpmv_.h>
   // Defines a simple class template V_<T> with main data elements
   // UseCount u_ and T *p_
   // and a simple class template Vv<T> with main data elements
   // UseCount u_ and std::vector<T> *p_
//#include <cpmv__.h>
   // Defines a simple class template V__<T> with data represented
   // as T *p_ and no UseCount usage.

//#include <vector>




using namespace CpmRoot;
using namespace CpmArrays;

void test6()
```

```
{
   Z n=100;
 //  auto r0=1234567.89_R;
   for (Z i=1;i<=n;++i){
      //std::vector<R> a(n);
      V<R> a(n);
   }
   return;
}

int main()
   // Traditional main function with types of arguments and return value
   // according to C/C++. Not all compilers accept using Z instead of
   // int here.
{
   test6();
   return 0_Z;
}
```

60 tut1experimenta.cpp

```
//? tut1experimenta.cpp
//? Status of work 2023-10-20.
//?
//?  ...
/*************************************************************************
   Comparing std::vector and CpmArrays::vector with respect to speed
   of copy-construction and assignment. Notice that this drastic
   advantage of CpmArrays::V over std::vector still exists in C++11, where
   moving instead of copying is said to be done where appropriate.
   The basic conclustion from the present case study is that moving
   works well (i.e. without being invoced explicitely by calling
   std::move(...)) when only instances of vector<T> are involved.
   Moving no longer works well for instances of vector<vector<T>>.
   The present version of CpmArrays::vector<T> holds it data content
   no longer as T *p_ but as std::vector<T> *p_. Thus it does
   push_back as efficiently as std::vector.


*************************************************************************/
#include <cpmbas.h>
   // contains '#include cpmv.h>' which provides the full-featured
   // class template V<T> with main data elements
   // IvZ iv_, UseCount u_, and std::vector<T> *p_
   // Here the 'copy on write' mechanism which is controled by u_
   // can be disabled by the preprocessor directive CPM_USECOUNT
   // set in file cpmdefinitions.h. This allows us to clearly see
   // the advantages over classes that have to rely on the 'move'
   // mechanism.
//#include <cpmv_.h>
   // Defines a simple class template V_<T> with main data elements
   // UseCount u_ and T *p_
   // and a simple class template Vv<T> with main data elements
   // UseCount u_ and std::vector<T> *p_
//#include <cpmv__.h>
   // Defines a simple class template V__<T> with data represented
   // as T *p_ and no UseCount usage.

//#include <vector>




using namespace CpmRoot;
using namespace CpmArrays;

void test6()
```

```
{
   Z n=100;
 //  auto r0=1234567.89_R;
   for (Z i=1;i<=n;++i){
      //std::vector<R> a(n);
      V<R> a(n);
   }
   return;
}

int main()
   // Traditional main function with types of arguments and return value
   // according to C/C++. Not all compilers accept using Z instead of
   // int here.
{
   test6();
   return 0_Z;
}
```

61 tut1old1.cpp

```
//? tut1old1.cpp
//? Status of work 2023-10-20.
//?
//?  ...


/*************************************************************************
   Comparing std::vector and CpmArays::V with respect to speed
   of copy-construction and assignment. Notice that this drastic
   advantage of CpmArays::V over std::vector still exists in C++11, where
   moving instead of copying is said to be done where appropriate.
*************************************************************************/
#include <cpmbas.h>
#include <cpmv_.h>
#include <quadmath.h>
#include <valarray>
/*************************************************************************
1. C+- filenames start with 'cpm' and contain neither underscores nor
   blanks. Therefore, even projects employing files from many sources are
   not likely to run into problems with non-unique file names.

This directive includes all C+- header files needed here;
'bas' stands for 'basics'. Notice that we have not written
#include "cpmbas.h"
and thus have assumed that our C++ project - let it be named tut1 -
has a list of include directories defined.
Who wants to use C+- for more than a single project, should hold
C+- files in directories separate from his project files.
The general purpose C+- files to be used in the project tut1 are on my
computer in xxx/cpm/cpm0/include and xxx/cpm/cpm0/source.
The files that are specific for the tut1 application are in
yyy/tut1/include and yyy/tut1/source.
In yyy/tut1/include there are the project specific headers
among which there need to be two C+- related configuration files:
cpmdefinitions.h and cpmsystemdependencies.h. Customizing these files
allows us to control the behavior of the C+- classes in our project, as
will be explained later.
In yyy/tut1/source the present project-defining main source tut1.cpp is to
be placed.
Now we are in a position to define the file content of project
tut1: Its include directories have to be set as
yyy/tut1/include and xxx/cpm/cpm0/include
and the files to be compiled (translation units) have to be chosen as
yyy/tut1/source/tut1.cpp and xxx/cpm/cpm0/source/cpmbas.cpp.

1.1 Details (skip on first reading ?):
The file cpmbas.cpp is actually a collection of all other files
```

```
in xxx/cpm/cpm0/source. Its content is

#include "cpmc.cpp"
#include "cpmangle.cpp"
#include "cpmv.cpp"
#include "cpmgreg.cpp"
#include "cpmsystem.cpp"
#include "cpmtypes.cpp"
#include "cpmuc.cpp"
#include "cpmzinterval.cpp"
#include "cpmnumbers.cpp"
#include "cpmword.cpp"
#include "cpmviewport.cpp"
End of 1.1
*****************************************************************/
using namespace CpmRoot;
/*************************************************************************
2. C+- namespace names start with 'Cpm' followed by an identifier
   which starts with a capital letters.

The namespace CpmRoot is rather small, so that the present using directive
should be applicable also in projects in which classes from various
sources are being used.

3. The C+- names for integer, real, and complex numbers are
   CpmRoot::Z, CpmRoot::R, Cpmoot::C. CpmRoot::Word wraps std::string and
   adds functionality to it.

The main effect of this using directive is that we may use these names
simply as Z, R, C, Word.

3.1 Details (skip on first reading ?):
There are the less ubiquitous types L, and N in CpmRoot:
L for unsigned characters ('L' for 'letter', after 'integer promotion'
the values range from 0 to 255), N for 'natural numbers', i.e. unsigned
integers. Types L,Z, and N are typedefs for unsigned char, int,
unsigned int. If we add in file cpmdefinition.h the macro
#define CPM_LONG, we change Z to long int, and N to unsigned long int.
If in cpmdefinitions.h the macro CPM_MP is defined, R has the meaning of
mpreal as defined by the wrapper library MPFRC++ to the multiple precision
library mpfr. See comments to CPM_MP in file cpmdefinitionswrc.h. If
CPM_MP is not defined R is also a typedef, long double if CPM_LONG and
double else. The types bool and string from standard C++ are useful as
they stand. Nevertheless they will be wrapped into classes - CpmRootX::B
in cpmtypes.h and class CpmRoot::Word in file cpmword.h.
End of 3.1

There is a namespace CpmRootX for the 'less essential essentials':

4. There are classes CpmRootX::B, CpmRootX::Z1, CpmRootX::R1,  for
```

```
   Boolean values, integer, and real numbers, which may replace the
   non-class types bool, Z, and R in situations where class func-
   tionality, such as automatic initialization, is indispensable.
   Note 2017-02-18: With C++11 automatic initialization can be achieved
   also for the built-in types. For instance a data member of a class X
         bool b {false};
   can be handeled as if we had declared it as
         B b;

4.1 Details (skip on first reading ?):
Namespace CpmRoot in mainly layed out in file cpmnumbers.h
and CpmRootX in cpmtypes.h. The vigilant reader may argue that we
have not available the declarations of CpmRoot, since we have no
   #include <cpmnumbers.h>
seen so far. Actually, it is there since file cpmbas.h is the following
collection of include directives
#ifndef CPM_BAS_H_
#define CPM_BAS_H_
   #include <cpmtypes.h>
   #include <cpmfr.h>
   #include <cpmvr.h>
   #include <cpmsr.h>
   #include <cpmm.h>
   #include <cpmp.h>
   #include <cpmc.h>
   #include <cpmangle.h>
   #include <cpmgreg.h>
#endif
Also here we don't see
   #include <cpmnumbers.h>
but inspecting cpmtypes.h, we find the commented directive
   #include <cpmsystem.h> // includes cpmwords.h and
      // thus also cpmnumbers.h
which shows the steps which finally include cpmnumbers.h. Acually the
directory cpm/cpm0/include contains 32 header files. This set of files
(as any set of C++ header files) is a directed graph in a natural manner:
There is an edge leading from file1 to file2 iff file2 contains the
directive  #include <file1> .
The files that can be reached from file1 along a path of that graph depend
on file1 and every translation unit which includes such a file1-dependent
header has to be re-compiled upon some change in file1. So, any system
capable of generating proper make files has to do this dependency
analysis and thus has to work with this  d e p e n d e n c y  g r a p h.
I made a tool consisting of a Ruby program for file analysis
and a C+- program for graph analysis and representation which
creates a  pictorial representation of this dependency graph of an
arbitrary collecton of C/C++ header files. This tool turned out to be very
useful.
It is an obvious logical requirement that the dependency graph is free of
cycles. It is a non-trivial task to organize the header dependencies in a
```

```
way that each translation unit gets the declarations which it needs to
know by inclusion of only a few header files. For instance, cpmword.cpp
and cpmtypes.cpp need only cpmtypes.h; cpmc.cpp needs cpmc.h and
cpmtypes.h. The present header file hierarchy is the result of many
simplifying re-organizations. However, many attempts of a simplification
failed since they entailed unacceptable complications elsewhere. I'm not
aware of tools which would automatize relevant parts of this organization
process.
End of 4.1
*************************************************************************/
using namespace CpmArrays;
/*************************************************************************
5. The general-purpose array in C+- is the template class CpmArrays::V
   Most constructors set the first valid index of a non-void V is 1 and
   not 0 as for std::vector. There is, however, a member function which
   shifts all indexes and so allows indexing to start with 0. This allows
   identical code to be used for these two types of arrays.

5.1 Details on C+- namespaces (skip on first reading ?):
The C+- class system introduces many namespaces. Their belonging
togeher under the Cpm banner is not expressed by making them
sub-namespaces of a single Cpm-namespace. Instead, their names all
start with 'Cpm' and continue with a capital letter.
If the name contains a digit, this is 2 or 3 and refers to the space
dimension under consideration. So the namespace CpmDim2 contains the
2D ('flatland') analogs of the geometric classes defined in
namespace CpmDim3.
The namespaces which are declared in the present scope by including
cpmbas.h are
CpmRoot, CpmRootX, CpmSystem, CpmArrays, CpmFunctions, CpmGeo,
CpmGraphics, CpmMPI, CpmStd, CpmTests, CpmTime.
End of 5.1
*************************************************************************/

void info();
Z tutorial1(Z,Z);
   // Declaration of two functions, the definition of which will
   // determine the functionality of the program.

int main(int argc, char* argv[])
   // Traditional main function with types of arguments and return value
   // according to C/C++. Not all compilers accept using Z instead of
   // int here.
{
 //  Z m=200, n=20000; // default values to be used if no input from the
      //  command line can be found
   Z m=1000, n=1000;
      // 5000, 5000 works fast for CPM_R but not for CPM_RLONG
      // 2022-03-27 timing series for m=1000, n=1000
      // with the new fixed precisions FLOAT and QUAD
```

```
   // CPM_FLOAT  t12_v = 0.060168
   // CPM_DOUBLE t12_v = 0.116369
   // CPM_LONG   t12_v = 0.207234
   // CPM_QUAD   t12_v = 0.192921
   // CPM_MP 33  t12_v = 1.130933
   // This simple picture emerges:
   //    speedFLOAT : speedDOUBLE : speedQUAD = 2 : 2 : 2
   // CPM_LONG essentially equal to CPM_QUAD and CPM_MP 33 is
   // ~ 6 times slower than eqally precise CPM_QUAD.

V<Word> arg=comLine(argc,argv); // CpmRoot::Word is similar to
   // std::string. CpmArrays::V is the C+- array type. There are two
   // 'schools' about indexing arrays. The 'C school' lets valid
   // indexes start with 0 and the 'Fortran school' which lets valid
   // valid indexes start with 1. Till September 2010 C+- had a
   // 'C school'-array Vl (l for 'lean') and a 'Fortran school'-array
   // V , where the implementation of V was such that any instance of
   // V had a data member of type Vl. The type Vl was used mostly
   // internally in situation where efficiency was considered to be
   // of utmost importance. Finally it became clear that the decision
   // to have these two types of arrays was a mistake. It creates a
   // permanent uncertainty for the programmer whether he should use
   // the default type or should strive for utmost efficiency by
   // using the lean version. Now we have only a single array type V
   // for which the valid indexes may form any contiguous set of
   // integers. This index range is set to start with 1 in most
   // of the available constructors but can be very conveniently reset
   // to any other start index such as 0.
   // CpmArrays::comLine is a function which transforms the
   // traditional argument of main into the more functional data type
   // of C+-. The function name 'comLine' results from
   // the  d e s c r i p t i v e  f u l l  n a m e (DFN)
   // 'command line' by a simple deterministic rule: The rule is
   // to leave unchanged all words with up to four letters and shorten
   // all longer words to 3 by taking the first two letters as they are
   // and take the first of the following consonants as the third
   // letter. Thus 'ooops' gets converted to 'oop'. If there is no
   // such consonant, the vowel at place 3 has to be taken: 'hooiii'
   // gets converted to 'hoo'. In a chain of words, the contracted
   // components get concateneted in a style which is evident from
   // the example that 'descriptive full name' goes to
   // 'desFullName'. In most cases the names come out quite nice, but
   // there are exceptions (not to the rules!): I cosider it ugly to
   // see 'field' abbreviated to 'fil' (notice that 'file', as having
   // only four letters, remains 'file')
   // The simplicity of the rule allows us to use the the function name
   // fluently whenever we remember the full name correctly. Of course,
   // the descriptive full name of a function has to be stated in the
   // declaration of this function. In our case the declaration is in
   // file cpmv.h:
```

```
      // inline V<Word> comLine(int argc, char* argv[])
      //    //: command line
      // The colon hints at the fact that here a descriptive full name
      // is being communicated.
   if (arg.valInd(2)){ // valInd's DFN is 'valid index'
      // C+- arrays judge the validity of an index directly without
      // a need to ask for 'size' and making a comparison.
      Word w2=arg[2];
      if (w2=="?"){ // Thus a question mark as the second entry of the
            // command line triggers a call to function info.
         info();
         return 0;
      }
      else{
         m=w2.toZ(); // Converting Word to Z, for getting a control
            // parameter. There are also conversions toR(), toBool(),
            // and toStr()
         if (arg.valInd(3)) n=arg[3].toZ();
      }
   }
   return tutorial1(m,n);
  }

using namespace std;

C cz(Z i) // an ad-hoc function Z --> C
// CpmRoot::C is the class of complex numbers
{
   R ir=i;
   return C(ir/(1_R+ir*ir),ir*cpmpi); // there is a CpmRoot::Pi = 3.14159...
      // calling constructor C(R,R)
}

// Now we define the performance measuring functions perf_v and perf_V.
// These functions have to execute identical code for two different
// vector templates: std::vector in perf_v and CpmArrays::V in perf_V.
// Since we want to iterate these templates, it would not be easy to
// achieve this code doublication by defining a template function.
// Such a template would need at least two template arguments and would
// enforce unnatural expressions.

// This function is designed as to maximize the benefit from reference
// counting and to report the time spent on copy-construction and
// assignment (not of the tedious verification part of the function).

#define CopyOnWrite
//#undef CopyOnWrite
#define UseValarray
//#undef UseValarray
```

```
#if defined(UseValarray)
   #define Vec std::valarray
#else
   #define Vec std::vector
#endif
V<R> perf_v(Z m, Z n)
// performance measuring function for std::vector or std::valarray
{
   Z mL=1;
/*
   This line, together with the following
   Word loc(...);
   CPM_MA
   CPM_MZ
   consitute a convenient idiom for signaling
   entry to and exit from a function block to the log file
   cpmcerr.txt. This writing to the log file takes place only if
   mL is larger or equal to the static data member
   CpmSystem::Message::verbose. This bulky quantity has a
   macro alias 'cpmverbose' (i.e. we have, in file cpmsystem.h
       #define cpmverbose      CpmSystem::Message::verbose
   ) and it can be set by a statement like
       cpmverbose=10;
   Its initial value is 2.
   Soon we will encounter macros cpmtime, cpmwait, cpmdebug.
   See the remarks on the macro namespace in function perf_V.
*/
   Word loc("perf_v(Z,Z)"); // messages use this as name of the function
   CPM_MA // defined in cpmmacros.h, assumes that 'mL' and 'loc' are
      // defined
   R t1=cpmtime(); /* time of function call in seconds from some
       system-defined 'point-zero in time'.
       cpmtime is a short name for function CpmSystem::time defined
       in file cpmsystem.h as follows:
           #define cpmtime         CpmSystem::time
   */
   Z i,j;

   Vec<Vec<C>> vi, vf;
   Vec<C> v1(m);
   for (i=0;i<m;++i) v1[i]=cz(i);
         // The preferred form of this statement in C+- is
         // for (i=v1.b();i<=v1.e();++i) v1[i]=cz(i);
         // Written in this form, it is alo correct if v1 is of type V<C>,
         // the C+- array for which indexing starts with 1 instead of 0.
#if defined(UseValarray)
   Vec< Vec<C> > w(v1,n);
#else
   Vec< Vec<C> > w(n,v1);
```

```
#endif
     // If we use valarray for Vec, one has to write w(v1,n) instead.
     // This is a inconvenience of the C++ standard library which is
     // due to their multi-source origin.
   Vec< Vec<C> > temp1=w; // copy constructor
   Vec< Vec<C> > temp2=temp1; // copy constructor
   Vec< Vec<C> > temp3=temp2; // copy constructor
   temp2=Vec< Vec<C> >(0); // assignment
   vi=w; // assignment
   vf=temp3; // assignment
   R t2=cpmtime();
   R t12=t2-t1; // time needed for copy and assignment
   R err=0,sum_v=0;
   for (i=0;i<n;++i){
      for (j=0;j<m;++j){ // sic!
         err+=(vi[i][j]-vf[i][j]).abs();
         sum_v+=(vf[i][j]).abs();
      }
   } // making sure that copying and assignment worked correctly
   R t3=cpmtime();
   R t23=t3-t2;
   V<R> res{t12,t23,err,sum_v};
/*
   Convenient  l i s t  c o n s t r u c t o r, requires C++11
*/
  CPM_MZ
  return res;
}
#undef Vec
#if defined(CopyOnWrite)
   #define Vec V_
   // The copy on write array V_ has the same index range as std::vector.
   // So in this case the code is the same as the one vor std::vector
V<R> perf_V(Z m, Z n)
// performance measuring function for CpmArray::V_
   // with the adaption to C++20 my V template builds on std::vector
   // and is essentially as efficient as this. The particular advantage
   // that the old copy on write strategy showed for chained calls to
   // copy constructors (which is not really needed in practice) is gone
   // with this adaption. Here we have preserved this former general advantage
   // by defining a lean class template V_<> which has the same access functions
   // as std::vector but implements copy construction and assignment via
   // copy on write instead of 'move semantics'.
{
   Z mL=1;
   Word loc("perf_V(Z,Z)");

   CPM_MA  // macro not followed by a semi-colon
   R t1=cpmtime(); // macro followed by a semi-colon
   N i,j;
```

```
   Vec< Vec<C> > vi, vf;
   {
      Vec<C> v1(m);

      for (i=0;i<m;++i) v1[i]=cz(i);

      Vec< Vec<C> > w(n,v1);
      Vec< Vec<C> > temp1=w;
      Vec< Vec<C> > temp2=temp1;
      Vec< Vec<C> > temp3=temp2;
      temp2=Vec< Vec<C> >();
      vi=w;
      vf=temp3;
   }
   R t2=cpmtime();
   R t12=t2-t1;
   R err=0,sum_V=0;
   for (i=0;i<n;++i){
      for (j=0;j<m;++j){
         err+=(vi[i][j]-vf[i][j]).abs();
         sum_V+=vf[i][j].abs();
      }
   }
   R t3=cpmtime();
   R t23=t3-t2;
   CPM_MZ
   return V<R>{t12,t23,err,sum_V};
}
#else // this is now present C+-
#define Vec V
V<R> perf_V(Z m, Z n)
// performance measuring function for CpmArrays::V
   // with the adaption to C++20 my V template builds on std::vector
   // and is essentially as efficient as this. The particular advantage
   // that the old copy on write strategy showed for chained calls to
   // copy constructors (which is not really needed in practice) is gone
   // with this adaption.

{
   Z mL=1;
   Word loc("perf_V(Z,Z)");

   CPM_MA  // macro not followed by a semi-colon
   R t1=cpmtime(); // macro followed by a semi-colon
   N i,j;
   Vec< Vec<C> > vi, vf;
   {
      Vec<C> v1(m);
      for (i=1;i<=m;++i) v1[i]=cz(i-1); // there is one function cz for different
         // ranges of valid indexes. Therefore i-1 instead of i in this case.
```

```
      Vec< Vec<C> > w(n,v1);
      Vec< Vec<C> > temp1=w;
      Vec< Vec<C> > temp2=temp1;
      Vec< Vec<C> > temp3=temp2;
      temp2=Vec< Vec<C> >();
      vi=w;
      vf=temp3;
   }
   R t2=cpmtime();
   R t12=t2-t1;
   R err=0,sum_V=0;
   for (i=1;i<=n;++i){
      for (j=1;j<=m;++j){ // cui() is same as [] but without rane check
         err+=(vi[i][j]-vf[i][j]).abs();
         sum_V+=vf[i][j].abs();
      }
   }
   R t3=cpmtime();
   R t23=t3-t2;
   CPM_MZ
   return V<R>{t12,t23,err,sum_V};
}
#endif
#undef Vec

Z tutorial1(Z m, Z n)
{
   Z mL=1;
   Word loc("tutorial1(Z,Z)");

   CPM_MA
   V<R> res_V=perf_V(m,n), res_v=perf_v(m,n);
   R t12_V=res_V[1], t12_v=res_v[1];
   R t23_V=res_V[2], t23_v=res_v[2];
   R t_V=t12_V+t23_V, t_v=t12_v+t23_v;
   R err_V=res_V[3], err_v=res_v[3];
   R sum_V=res_V[4], sum_v=res_v[4];
   R fac12=t12_v*cpminv(t12_V);
   R fac23=t23_v*cpminv(t23_V);
   R fac=t_v*cpminv(t_V);
   R diff=sum_v-sum_V;
   cout<<setprecision(20)<<endl;
      // diff should be 0 up to numerical noise
   cout<<"m="<<m<<", n="<<n<<endl;
   cout<<"t12_v="<<t12_v<<", t12_V="<<t12_V<<endl;
   cout<<"t12 is the execution time for the copy and assignment part"
       <<endl;
   cout<<"Since V<> now builds on std::vector<> we see that the"<<endl;
   cout<<"two are very similar in speed."<<endl;
```

```
   cout<<"t23_v="<<t23_v<<", t23_V="<<t23_V<<endl;
   cout<<"fac12="<<fac12<<endl;
   cout<<"fac23="<<fac23<<endl;
   cout<<" fac="<<fac<<endl;
   cout<<" fac is t_v/t_V so that values < 1 indicate an overall advantage for std::vector"<<endl;
   cout<<" sum_v="<<sum_v<<", sum_V="<<sum_V<<",  diff= "<<diff<<endl;
      // output of the result to the console
   cpmdebug(m);
   cpmdebug(n);
   cpmdebug(t12_v);
   cpmdebug(t12_V);
   cpmdebug(t23_v);
   cpmdebug(t23_V);
   cpmdebug(fac12);
   cpmdebug(fac23);
   cpmdebug(fac);
   cpmdebug(err_v);
   cpmdebug(err_V);
   cpmdebug(sum_v);
   cpmdebug(sum_V);
   cpmdebug(diff);
      // convenient documentation of the main result on
      // the auto-generated log file cpmcerr.txt.
      // Defined in file cpmtypes.h as
   // #define cpmdebug(X) cpmcerr<<endl<<"C+- debug: "<<#X "="<< X <<endl
      // Here are the corresponding lines of this file:
   CPM_MZ
   return 0;
}

void info(){
   cout<<endl<<"takes zero, one, or two integer argument";
   cout<<endl<<"compares the performance or std::vector and CpmArays::V";
   cout<<endl<<"in copy construction and assignment operations"<<endl;
}
// end of tut1.cpp
```

62 tut1old1.cpp

```
//? tut1old1.cpp
//? Status of work 2023-10-20.
//?
//?  ...


/*************************************************************************
   Comparing std::vector and CpmArays::V with respect to speed
   of copy-construction and assignment. Notice that this drastic
   advantage of CpmArays::V over std::vector still exists in C++11, where
   moving instead of copying is said to be done where appropriate.
*************************************************************************/
#include <cpmbas.h>
#include <cpmv_.h>
#include <quadmath.h>
#include <valarray>
/*************************************************************************
1. C+- filenames start with 'cpm' and contain neither underscores nor
   blanks. Therefore, even projects employing files from many sources are
   not likely to run into problems with non-unique file names.

This directive includes all C+- header files needed here;
'bas' stands for 'basics'. Notice that we have not written
#include "cpmbas.h"
and thus have assumed that our C++ project - let it be named tut1 -
has a list of include directories defined.
Who wants to use C+- for more than a single project, should hold
C+- files in directories separate from his project files.
The general purpose C+- files to be used in the project tut1 are on my
computer in xxx/cpm/cpm0/include and xxx/cpm/cpm0/source.
The files that are specific for the tut1 application are in
yyy/tut1/include and yyy/tut1/source.
In yyy/tut1/include there are the project specific headers
among which there need to be two C+- related configuration files:
cpmdefinitions.h and cpmsystemdependencies.h. Customizing these files
allows us to control the behavior of the C+- classes in our project, as
will be explained later.
In yyy/tut1/source the present project-defining main source tut1.cpp is to
be placed.
Now we are in a position to define the file content of project
tut1: Its include directories have to be set as
yyy/tut1/include and xxx/cpm/cpm0/include
and the files to be compiled (translation units) have to be chosen as
yyy/tut1/source/tut1.cpp and xxx/cpm/cpm0/source/cpmbas.cpp.

1.1 Details (skip on first reading ?):
The file cpmbas.cpp is actually a collection of all other files
```

```
in xxx/cpm/cpm0/source. Its content is

#include "cpmc.cpp"
#include "cpmangle.cpp"
#include "cpmv.cpp"
#include "cpmgreg.cpp"
#include "cpmsystem.cpp"
#include "cpmtypes.cpp"
#include "cpmuc.cpp"
#include "cpmzinterval.cpp"
#include "cpmnumbers.cpp"
#include "cpmword.cpp"
#include "cpmviewport.cpp"
End of 1.1
*****************************************************************/
using namespace CpmRoot;
/*************************************************************************
2. C+- namespace names start with 'Cpm' followed by an identifier
   which starts with a capital letters.

The namespace CpmRoot is rather small, so that the present using directive
should be applicable also in projects in which classes from various
sources are being used.

3. The C+- names for integer, real, and complex numbers are
   CpmRoot::Z, CpmRoot::R, Cpmoot::C. CpmRoot::Word wraps std::string and
   adds functionality to it.

The main effect of this using directive is that we may use these names
simply as Z, R, C, Word.

3.1 Details (skip on first reading ?):
There are the less ubiquitous types L, and N in CpmRoot:
L for unsigned characters ('L' for 'letter', after 'integer promotion'
the values range from 0 to 255), N for 'natural numbers', i.e. unsigned
integers. Types L,Z, and N are typedefs for unsigned char, int,
unsigned int. If we add in file cpmdefinition.h the macro
#define CPM_LONG, we change Z to long int, and N to unsigned long int.
If in cpmdefinitions.h the macro CPM_MP is defined, R has the meaning of
mpreal as defined by the wrapper library MPFRC++ to the multiple precision
library mpfr. See comments to CPM_MP in file cpmdefinitionswrc.h. If
CPM_MP is not defined R is also a typedef, long double if CPM_LONG and
double else. The types bool and string from standard C++ are useful as
they stand. Nevertheless they will be wrapped into classes - CpmRootX::B
in cpmtypes.h and class CpmRoot::Word in file cpmword.h.
End of 3.1

There is a namespace CpmRootX for the 'less essential essentials':

4. There are classes CpmRootX::B, CpmRootX::Z1, CpmRootX::R1,  for
```

```
   Boolean values, integer, and real numbers, which may replace the
   non-class types bool, Z, and R in situations where class func-
   tionality, such as automatic initialization, is indispensable.
   Note 2017-02-18: With C++11 automatic initialization can be achieved
   also for the built-in types. For instance a data member of a class X
         bool b {false};
   can be handeled as if we had declared it as
         B b;

4.1 Details (skip on first reading ?):
Namespace CpmRoot in mainly layed out in file cpmnumbers.h
and CpmRootX in cpmtypes.h. The vigilant reader may argue that we
have not available the declarations of CpmRoot, since we have no
   #include <cpmnumbers.h>
seen so far. Actually, it is there since file cpmbas.h is the following
collection of include directives
#ifndef CPM_BAS_H_
#define CPM_BAS_H_
   #include <cpmtypes.h>
   #include <cpmfr.h>
   #include <cpmvr.h>
   #include <cpmsr.h>
   #include <cpmm.h>
   #include <cpmp.h>
   #include <cpmc.h>
   #include <cpmangle.h>
   #include <cpmgreg.h>
#endif
Also here we don't see
   #include <cpmnumbers.h>
but inspecting cpmtypes.h, we find the commented directive
   #include <cpmsystem.h> // includes cpmwords.h and
      // thus also cpmnumbers.h
which shows the steps which finally include cpmnumbers.h. Acually the
directory cpm/cpm0/include contains 32 header files. This set of files
(as any set of C++ header files) is a directed graph in a natural manner:
There is an edge leading from file1 to file2 iff file2 contains the
directive  #include <file1> .
The files that can be reached from file1 along a path of that graph depend
on file1 and every translation unit which includes such a file1-dependent
header has to be re-compiled upon some change in file1. So, any system
capable of generating proper make files has to do this dependency
analysis and thus has to work with this  d e p e n d e n c y  g r a p h.
I made a tool consisting of a Ruby program for file analysis
and a C+- program for graph analysis and representation which
creates a  pictorial representation of this dependency graph of an
arbitrary collecton of C/C++ header files. This tool turned out to be very
useful.
It is an obvious logical requirement that the dependency graph is free of
cycles. It is a non-trivial task to organize the header dependencies in a
```

```
way that each translation unit gets the declarations which it needs to
know by inclusion of only a few header files. For instance, cpmword.cpp
and cpmtypes.cpp need only cpmtypes.h; cpmc.cpp needs cpmc.h and
cpmtypes.h. The present header file hierarchy is the result of many
simplifying re-organizations. However, many attempts of a simplification
failed since they entailed unacceptable complications elsewhere. I'm not
aware of tools which would automatize relevant parts of this organization
process.
End of 4.1
**************************************************************************/
using namespace CpmArrays;
/**************************************************************************
5. The general-purpose array in C+- is the template class CpmArrays::V
   Most constructors set the first valid index of a non-void V is 1 and
   not 0 as for std::vector. There is, however, a member function which
   shifts all indexes and so allows indexing to start with 0. This allows
   identical code to be used for these two types of arrays.

5.1 Details on C+- namespaces (skip on first reading ?):
The C+- class system introduces many namespaces. Their belonging
togeher under the Cpm banner is not expressed by making them
sub-namespaces of a single Cpm-namespace. Instead, their names all
start with 'Cpm' and continue with a capital letter.
If the name contains a digit, this is 2 or 3 and refers to the space
dimension under consideration. So the namespace CpmDim2 contains the
2D ('flatland') analogs of the geometric classes defined in
namespace CpmDim3.
The namespaces which are declared in the present scope by including
cpmbas.h are
CpmRoot, CpmRootX, CpmSystem, CpmArrays, CpmFunctions, CpmGeo,
CpmGraphics, CpmMPI, CpmStd, CpmTests, CpmTime.
End of 5.1
**************************************************************************/

void info();
Z tutorial1(Z,Z);
   // Declaration of two functions, the definition of which will
   // determine the functionality of the program.

int main(int argc, char* argv[])
   // Traditional main function with types of arguments and return value
   // according to C/C++. Not all compilers accept using Z instead of
   // int here.
{
 //  Z m=200, n=20000; // default values to be used if no input from the
      //  command line can be found
   Z m=1000, n=1000;
      // 5000, 5000 works fast for CPM_R but not for CPM_RLONG
      // 2022-03-27 timing series for m=1000, n=1000
      // with the new fixed precisions FLOAT and QUAD
```

```
   // CPM_FLOAT  t12_v = 0.060168
   // CPM_DOUBLE t12_v = 0.116369
   // CPM_LONG   t12_v = 0.207234
   // CPM_QUAD   t12_v = 0.192921
   // CPM_MP 33  t12_v = 1.130933
   // This simple picture emerges:
   //    speedFLOAT : speedDOUBLE : speedQUAD = 2 : 2 : 2
   // CPM_LONG essentially equal to CPM_QUAD and CPM_MP 33 is
   // ~ 6 times slower than eqally precise CPM_QUAD.

V<Word> arg=comLine(argc,argv); // CpmRoot::Word is similar to
   // std::string. CpmArrays::V is the C+- array type. There are two
   // 'schools' about indexing arrays. The 'C school' lets valid
   // indexes start with 0 and the 'Fortran school' which lets valid
   // valid indexes start with 1. Till September 2010 C+- had a
   // 'C school'-array Vl (l for 'lean') and a 'Fortran school'-array
   // V , where the implementation of V was such that any instance of
   // V had a data member of type Vl. The type Vl was used mostly
   // internally in situation where efficiency was considered to be
   // of utmost importance. Finally it became clear that the decision
   // to have these two types of arrays was a mistake. It creates a
   // permanent uncertainty for the programmer whether he should use
   // the default type or should strive for utmost efficiency by
   // using the lean version. Now we have only a single array type V
   // for which the valid indexes may form any contiguous set of
   // integers. This index range is set to start with 1 in most
   // of the available constructors but can be very conveniently reset
   // to any other start index such as 0.
   // CpmArrays::comLine is a function which transforms the
   // traditional argument of main into the more functional data type
   // of C+-. The function name 'comLine' results from
   // the  d e s c r i p t i v e  f u l l  n a m e (DFN)
   // 'command line' by a simple deterministic rule: The rule is
   // to leave unchanged all words with up to four letters and shorten
   // all longer words to 3 by taking the first two letters as they are
   // and take the first of the following consonants as the third
   // letter. Thus 'ooops' gets converted to 'oop'. If there is no
   // such consonant, the vowel at place 3 has to be taken: 'hooiii'
   // gets converted to 'hoo'. In a chain of words, the contracted
   // components get concateneted in a style which is evident from
   // the example that 'descriptive full name' goes to
   // 'desFullName'. In most cases the names come out quite nice, but
   // there are exceptions (not to the rules!): I cosider it ugly to
   // see 'field' abbreviated to 'fil' (notice that 'file', as having
   // only four letters, remains 'file')
   // The simplicity of the rule allows us to use the the function name
   // fluently whenever we remember the full name correctly. Of course,
   // the descriptive full name of a function has to be stated in the
   // declaration of this function. In our case the declaration is in
   // file cpmv.h:
```

```
      // inline V<Word> comLine(int argc, char* argv[])
      //    //: command line
      // The colon hints at the fact that here a descriptive full name
      // is being communicated.
   if (arg.valInd(2)){ // valInd's DFN is 'valid index'
      // C+- arrays judge the validity of an index directly without
      // a need to ask for 'size' and making a comparison.
      Word w2=arg[2];
      if (w2=="?"){ // Thus a question mark as the second entry of the
            // command line triggers a call to function info.
         info();
         return 0;
      }
      else{
         m=w2.toZ(); // Converting Word to Z, for getting a control
            // parameter. There are also conversions toR(), toBool(),
            // and toStr()
         if (arg.valInd(3)) n=arg[3].toZ();
      }
   }
   return tutorial1(m,n);
  }

using namespace std;

C cz(Z i) // an ad-hoc function Z --> C
// CpmRoot::C is the class of complex numbers
{
   R ir=i;
   return C(ir/(1_R+ir*ir),ir*cpmpi); // there is a CpmRoot::Pi = 3.14159...
      // calling constructor C(R,R)
}

// Now we define the performance measuring functions perf_v and perf_V.
// These functions have to execute identical code for two different
// vector templates: std::vector in perf_v and CpmArrays::V in perf_V.
// Since we want to iterate these templates, it would not be easy to
// achieve this code doublication by defining a template function.
// Such a template would need at least two template arguments and would
// enforce unnatural expressions.

// This function is designed as to maximize the benefit from reference
// counting and to report the time spent on copy-construction and
// assignment (not of the tedious verification part of the function).

#define CopyOnWrite
//#undef CopyOnWrite
#define UseValarray
//#undef UseValarray
```

```
#if defined(UseValarray)
   #define Vec std::valarray
#else
   #define Vec std::vector
#endif
V<R> perf_v(Z m, Z n)
// performance measuring function for std::vector or std::valarray
{
   Z mL=1;
/*
   This line, together with the following
   Word loc(...);
   CPM_MA
   CPM_MZ
   consitute a convenient idiom for signaling
   entry to and exit from a function block to the log file
   cpmcerr.txt. This writing to the log file takes place only if
   mL is larger or equal to the static data member
   CpmSystem::Message::verbose. This bulky quantity has a
   macro alias 'cpmverbose' (i.e. we have, in file cpmsystem.h
       #define cpmverbose      CpmSystem::Message::verbose
   ) and it can be set by a statement like
       cpmverbose=10;
   Its initial value is 2.
   Soon we will encounter macros cpmtime, cpmwait, cpmdebug.
   See the remarks on the macro namespace in function perf_V.
*/
   Word loc("perf_v(Z,Z)"); // messages use this as name of the function
   CPM_MA // defined in cpmmacros.h, assumes that 'mL' and 'loc' are
      // defined
   R t1=cpmtime(); /* time of function call in seconds from some
       system-defined 'point-zero in time'.
       cpmtime is a short name for function CpmSystem::time defined
       in file cpmsystem.h as follows:
           #define cpmtime         CpmSystem::time
   */
   Z i,j;

   Vec<Vec<C>> vi, vf;
   Vec<C> v1(m);
   for (i=0;i<m;++i) v1[i]=cz(i);
         // The preferred form of this statement in C+- is
         // for (i=v1.b();i<=v1.e();++i) v1[i]=cz(i);
         // Written in this form, it is alo correct if v1 is of type V<C>,
         // the C+- array for which indexing starts with 1 instead of 0.
#if defined(UseValarray)
   Vec< Vec<C> > w(v1,n);
#else
   Vec< Vec<C> > w(n,v1);
```

```
#endif
     // If we use valarray for Vec, one has to write w(v1,n) instead.
     // This is a inconvenience of the C++ standard library which is
     // due to their multi-source origin.
   Vec< Vec<C> > temp1=w; // copy constructor
   Vec< Vec<C> > temp2=temp1; // copy constructor
   Vec< Vec<C> > temp3=temp2; // copy constructor
   temp2=Vec< Vec<C> >(0); // assignment
   vi=w; // assignment
   vf=temp3; // assignment
   R t2=cpmtime();
   R t12=t2-t1; // time needed for copy and assignment
   R err=0,sum_v=0;
   for (i=0;i<n;++i){
      for (j=0;j<m;++j){ // sic!
         err+=(vi[i][j]-vf[i][j]).abs();
         sum_v+=(vf[i][j]).abs();
      }
   } // making sure that copying and assignment worked correctly
   R t3=cpmtime();
   R t23=t3-t2;
   V<R> res{t12,t23,err,sum_v};
/*
   Convenient  l i s t  c o n s t r u c t o r, requires C++11
*/
  CPM_MZ
  return res;
}
#undef Vec
#if defined(CopyOnWrite)
   #define Vec V_
   // The copy on write array V_ has the same index range as std::vector.
   // So in this case the code is the same as the one vor std::vector
V<R> perf_V(Z m, Z n)
// performance measuring function for CpmArray::V_
   // with the adaption to C++20 my V template builds on std::vector
   // and is essentially as efficient as this. The particular advantage
   // that the old copy on write strategy showed for chained calls to
   // copy constructors (which is not really needed in practice) is gone
   // with this adaption. Here we have preserved this former general advantage
   // by defining a lean class template V_<> which has the same access functions
   // as std::vector but implements copy construction and assignment via
   // copy on write instead of 'move semantics'.
{
   Z mL=1;
   Word loc("perf_V(Z,Z)");

   CPM_MA  // macro not followed by a semi-colon
   R t1=cpmtime(); // macro followed by a semi-colon
   N i,j;
```

```
   Vec< Vec<C> > vi, vf;
   {
      Vec<C> v1(m);

      for (i=0;i<m;++i) v1[i]=cz(i);

      Vec< Vec<C> > w(n,v1);
      Vec< Vec<C> > temp1=w;
      Vec< Vec<C> > temp2=temp1;
      Vec< Vec<C> > temp3=temp2;
      temp2=Vec< Vec<C> >();
      vi=w;
      vf=temp3;
   }
   R t2=cpmtime();
   R t12=t2-t1;
   R err=0,sum_V=0;
   for (i=0;i<n;++i){
      for (j=0;j<m;++j){
         err+=(vi[i][j]-vf[i][j]).abs();
         sum_V+=vf[i][j].abs();
      }
   }
   R t3=cpmtime();
   R t23=t3-t2;
   CPM_MZ
   return V<R>{t12,t23,err,sum_V};
}
#else // this is now present C+-
#define Vec V
V<R> perf_V(Z m, Z n)
// performance measuring function for CpmArrays::V
   // with the adaption to C++20 my V template builds on std::vector
   // and is essentially as efficient as this. The particular advantage
   // that the old copy on write strategy showed for chained calls to
   // copy constructors (which is not really needed in practice) is gone
   // with this adaption.

{
   Z mL=1;
   Word loc("perf_V(Z,Z)");

   CPM_MA  // macro not followed by a semi-colon
   R t1=cpmtime(); // macro followed by a semi-colon
   N i,j;
   Vec< Vec<C> > vi, vf;
   {
      Vec<C> v1(m);
      for (i=1;i<=m;++i) v1[i]=cz(i-1); // there is one function cz for different
         // ranges of valid indexes. Therefore i-1 instead of i in this case.
```

```
      Vec< Vec<C> > w(n,v1);
      Vec< Vec<C> > temp1=w;
      Vec< Vec<C> > temp2=temp1;
      Vec< Vec<C> > temp3=temp2;
      temp2=Vec< Vec<C> >();
      vi=w;
      vf=temp3;
   }
   R t2=cpmtime();
   R t12=t2-t1;
   R err=0,sum_V=0;
   for (i=1;i<=n;++i){
      for (j=1;j<=m;++j){ // cui() is same as [] but without rane check
         err+=(vi[i][j]-vf[i][j]).abs();
         sum_V+=vf[i][j].abs();
      }
   }
   R t3=cpmtime();
   R t23=t3-t2;
   CPM_MZ
   return V<R>{t12,t23,err,sum_V};
}
#endif
#undef Vec

Z tutorial1(Z m, Z n)
{
   Z mL=1;
   Word loc("tutorial1(Z,Z)");

   CPM_MA
   V<R> res_V=perf_V(m,n), res_v=perf_v(m,n);
   R t12_V=res_V[1], t12_v=res_v[1];
   R t23_V=res_V[2], t23_v=res_v[2];
   R t_V=t12_V+t23_V, t_v=t12_v+t23_v;
   R err_V=res_V[3], err_v=res_v[3];
   R sum_V=res_V[4], sum_v=res_v[4];
   R fac12=t12_v*cpminv(t12_V);
   R fac23=t23_v*cpminv(t23_V);
   R fac=t_v*cpminv(t_V);
   R diff=sum_v-sum_V;
   cout<<setprecision(20)<<endl;
      // diff should be 0 up to numerical noise
   cout<<"m="<<m<<", n="<<n<<endl;
   cout<<"t12_v="<<t12_v<<", t12_V="<<t12_V<<endl;
   cout<<"t12 is the execution time for the copy and assignment part"
       <<endl;
   cout<<"Since V<> now builds on std::vector<> we see that the"<<endl;
   cout<<"two are very similar in speed."<<endl;
```

```
   cout<<"t23_v="<<t23_v<<", t23_V="<<t23_V<<endl;
   cout<<"fac12="<<fac12<<endl;
   cout<<"fac23="<<fac23<<endl;
   cout<<" fac="<<fac<<endl;
   cout<<" fac is t_v/t_V so that values < 1 indicate an overall advantage for std::vector"<<endl;
   cout<<" sum_v="<<sum_v<<", sum_V="<<sum_V<<",  diff= "<<diff<<endl;
      // output of the result to the console
   cpmdebug(m);
   cpmdebug(n);
   cpmdebug(t12_v);
   cpmdebug(t12_V);
   cpmdebug(t23_v);
   cpmdebug(t23_V);
   cpmdebug(fac12);
   cpmdebug(fac23);
   cpmdebug(fac);
   cpmdebug(err_v);
   cpmdebug(err_V);
   cpmdebug(sum_v);
   cpmdebug(sum_V);
   cpmdebug(diff);
      // convenient documentation of the main result on
      // the auto-generated log file cpmcerr.txt.
      // Defined in file cpmtypes.h as
   // #define cpmdebug(X) cpmcerr<<endl<<"C+- debug: "<<#X "="<< X <<endl
      // Here are the corresponding lines of this file:
   CPM_MZ
   return 0;
}

void info(){
   cout<<endl<<"takes zero, one, or two integer argument";
   cout<<endl<<"compares the performance or std::vector and CpmArays::V";
   cout<<endl<<"in copy construction and assignment operations"<<endl;
}
// end of tut1.cpp
```

63 tut1old2.cpp

```
//? tut1old2.cpp
//? Status of work 2023-10-20.
//?
//?  ...


/*************************************************************************
   Comparing std::vector and CpmArays::V with respect to speed
   of copy-construction and assignment. Notice that this drastic
   advantage of CpmArays::V over std::vector still exists in C++11, where
   moving instead of copying is said to be done where appropriate.
*************************************************************************/
#include <cpmbas.h>
#include <cpmv_.h>
#include <cpmv__.h>
#include <vector>
#include <valarray>
#include <quadmath.h>
/*************************************************************************
1. C+- filenames start with 'cpm' and contain neither underscores nor
   blanks. Therefore, even projects employing files from many sources are
   not likely to run into problems with non-unique file names.

This directive includes all C+- header files needed here;
'bas' stands for 'basics'. Notice that we have not written
#include "cpmbas.h"
and thus have assumed that our C++ project - let it be named tut1 -
has a list of include directories defined.
Who wants to use C+- for more than a single project, should hold
C+- files in directories separate from his project files.
The general purpose C+- files to be used in the project tut1 are on my
computer in xxx/cpm/cpm0/include and xxx/cpm/cpm0/source.
The files that are specific for the tut1 application are in
yyy/tut1/include and yyy/tut1/source.
In yyy/tut1/include there are the project specific headers
among which there need to be two C+- related configuration files:
cpmdefinitions.h and cpmsystemdependencies.h. Customizing these files
allows us to control the behavior of the C+- classes in our project, as
will be explained later.
In yyy/tut1/source the present project-defining main source tut1.cpp is to
be placed.
Now we are in a position to define the file content of project
tut1: Its include directories have to be set as
yyy/tut1/include and xxx/cpm/cpm0/include
and the files to be compiled (translation units) have to be chosen as
yyy/tut1/source/tut1.cpp and xxx/cpm/cpm0/source/cpmbas.cpp.
```

```
1.1 Details (skip on first reading ?):
The file cpmbas.cpp is actually a collection of all other files
in xxx/cpm/cpm0/source. Its content is

#include "cpmc.cpp"
#include "cpmangle.cpp"
#include "cpmv.cpp"
#include "cpmgreg.cpp"
#include "cpmsystem.cpp"
#include "cpmtypes.cpp"
#include "cpmuc.cpp"
#include "cpmzinterval.cpp"
#include "cpmnumbers.cpp"
#include "cpmword.cpp"
#include "cpmviewport.cpp"
End of 1.1
**************************************************************************/
using namespace CpmRoot;
/*************************************************************************
2. C+- namespace names start with 'Cpm' followed by an identifier
   which starts with a capital letters.

The namespace CpmRoot is rather small, so that the present using directive
should be applicable also in projects in which classes from various
sources are being used.

3. The C+- names for integer, real, and complex numbers are
   CpmRoot::Z, CpmRoot::R, Cpmoot::C. CpmRoot::Word wraps std::string and
   adds functionality to it.

The main effect of this using directive is that we may use these names
simply as Z, R, C, Word.

3.1 Details (skip on first reading ?):
There are the less ubiquitous types L, and N in CpmRoot:
L for unsigned characters ('L' for 'letter', after 'integer promotion'
the values range from 0 to 2nwnw), N for 'natural numbers', i.e. unsigned
integers. Types L,Z, and N are typedefs for unsigned char, int,
unsigned int. If we add in file cpmdefinition.h the macro
#define CPM_LONG, we change Z to long int, and N to unsigned long int.
If in cpmdefinitions.h the macro CPM_MP is defined, R has the meaning of
mpreal as defined by the wrapper library MPFRC++ to the multiple precision
library mpfr. See comments to CPM_MP in file cpmdefinitionswrc.h. If
CPM_MP is not defined R is also a typedef, long double if CPM_LONG and
double else. The types bool and string from standard C++ are useful as
they stand. Nevertheless they will be wrapped into classes - CpmRootX::B
in cpmtypes.h and class CpmRoot::Word in file cpmword.h.
End of 3.1

There is a namespace CpmRootX for the 'less essential essentials':
```

```
4. There are classes CpmRootX::B, CpmRootX::Z1, CpmRootX::R1,  for
   Boolean values, integer, and real numbers, which may replace the
   non-class types bool, Z, and R in situations where class func-
   tionality, such as automatic initialization, is indispensable.
   Note 2017-02-18: With C++11 automatic initialization can be achieved
   also for the built-in types. For instance a data member of a class X
         bool b {false};
   can be handeled as if we had declared it as
         B b;

4.1 Details (skip on first reading ?):
Namespace CpmRoot in mainly layed out in file cpmnumbers.h
and CpmRootX in cpmtypes.h. The vigilant reader may argue that we
have not available the declarations of CpmRoot, since we have no
   #include <cpmnumbers.h>
seen so far. Actually, it is there since file cpmbas.h is the following
collection of include directives
#ifndef CPM_BAS_H_
#define CPM_BAS_H_
   #include <cpmtypes.h>
   #include <cpmfr.h>
   #include <cpmvr.h>
   #include <cpmsr.h>
   #include <cpmm.h>
   #include <cpmp.h>
   #include <cpmc.h>
   #include <cpmangle.h>
   #include <cpmgreg.h>
#endif
Also here we don't see
   #include <cpmnumbers.h>
but inspecting cpmtypes.h, we find the commented directive
   #include <cpmsystem.h> // includes cpmwords.h and
      // thus also cpmnumbers.h
which shows the steps which finally include cpmnumbers.h. Acually the
directory cpm/cpm0/include contains 32 header files. This set of files
(as any set of C++ header files) is a directed graph in a natural manner:
There is an edge leading from file1 to file2 iff file2 contains the
directive  #include <file1> .
The files that can be reached from file1 along a path of that graph depend
on file1 and every translation unit which includes such a file1-dependent
header has to be re-compiled upon some change in file1. So, any system
capable of generating proper make files has to do this dependency
analysis and thus has to work with this  d e p e n d e n c y  g r a p h.
I made a tool consisting of a Ruby program for file analysis
and a C+- program for graph analysis and representation which
creates a  pictorial representation of this dependency graph of an
arbitrary collecton of C/C++ header files. This tool turned out to be very
useful.
```

```
It is an obvious logical requirement that the dependency graph is free of
cycles. It is a non-trivial task to organize the header dependencies in a
way that each translation unit gets the declarations which it needs to
know by inclusion of only a few header files. For instance, cpmword.cpp
and cpmtypes.cpp need only cpmtypes.h; cpmc.cpp needs cpmc.h and
cpmtypes.h. The present header file hierarchy is the result of many
simplifying re-organizations. However, many attempts of a simplification
failed since they entailed unacceptable complications elsewhere. I'm not
aware of tools which would automatize relevant parts of this organization
process.
End of 4.1
*****************************************************************/
using namespace CpmArrays;
/*****************************************************************
nw. The general-purpose array in C+- is the template class CpmArrays::V
   Most constructors set the first valid index of a non-void V is 1 and
   not 0 as for std::vector. There is, however, a member function which
   shifts all indexes and so allows indexing to start with 0. This allows
   identical code to be used for these two types of arrays.

nw.1 Details on C+- namespaces (skip on first reading ?):
The C+- class system introduces many namespaces. Their belonging
togeher under the Cpm banner is not expressed by making them
sub-namespaces of a single Cpm-namespace. Instead, their names all
start with 'Cpm' and continue with a capital letter.
If the name contains a digit, this is 2 or 3 and refers to the space
dimension under consideration. So the namespace CpmDim2 contains the
2D ('flatland') analogs of the geometric classes defined in
namespace CpmDim3.
The namespaces which are declared in the present scope by including
cpmbas.h are
CpmRoot, CpmRootX, CpmSystem, CpmArrays, CpmFunctions, CpmGeo,
CpmGraphics, CpmMPI, CpmStd, CpmTests, CpmTime.
End of nw.1
*****************************************************************/

void info();
N tutorial1(N,N,N);
   // Declaration of two functions, the definition of which will
   // determine the functionality of the program.#
//N tutorial2(N,N);

int main(int argc, char* argv[])
   // Traditional main function with types of arguments and return value
   // according to C/C++. Not all compilers accept using Z instead of
   // int here.
{
  //  N m=100, n=100;  //default values to be used if no input from the
   N m=2000, n=2000, diff=1; // default values to be used if no input from the
      //  command line can be found
```

```
   // nw000, nw000 works fast for CPM_R but not for CPM_RLONG
   // 2022-10-10 timing series for m=2000, n=2000 with O3 optimizing
   // with the new fixed precisions FLOAT and QUAD
   // CPM_FLOAT  t12+t23 V = 0.439032
   // CPM_DOUBLE t12+t23 V= 0.438204
   // CPM_LONG   t12+t23 V= 0.nw67039
   // CPM_QUAD   t12+t23 V= nw.336nw3
   // CPM_MP 32  t12+t23 V= 8.80929


V<Word> arg=comLine(argc,argv); // CpmRoot::Word is similar to
   // std::string. CpmArrays::V is the C+- array type. There are two
   // 'schools' about indexing arrays. The 'C school' lets valid
   // indexes start with 0 and the 'Fortran school' which lets valid
   // valid indexes start with 1. Till September 2010 C+- had a
   // 'C school'-array Vl (l for 'lean') and a 'Fortran school'-array
   // V , where the implementation of V was such that any instance of
   // V had a data member of type Vl. The type Vl was used mostly
   // internally in situation where efficiency was considered to be
   // of utmost importance. Finally it became clear that the decision
   // to have these two types of arrays was a mistake. It creates a
   // permanent uncertainty for the programmer whether he should use
   // the default type or should strive for utmost efficiency by
   // using the lean version. Now we have only a single array type V
   // for which the valid indexes may form any contiguous set of
   // integers. This index range is set to start with 1 in most
   // of the available constructors but can be very conveniently reset
   // to any other start index such as 0.
   // CpmArrays::comLine is a function which transforms the
   // traditional argument of main into the more functional data type
   // of C+-. The function name 'comLine' results from
   // the  d e s c r i p t i v e  f u l l  n a m e (DFN)
   // 'command line' by a simple deterministic rule: The rule is
   // to leave unchanged all words with up to four letters and shorten
   // all longer words to 3 by taking the first two letters as they are
   // and take the first of the following consonants as the third
   // letter. Thus 'ooops' gets converted to 'oop'. If there is no
   // such consonant, the vowel at place 3 has to be taken: 'hooiii'
   // gets converted to 'hoo'. In a chain of words, the contracted
   // components get concateneted in a style which is evident from
   // the example that 'descriptive full name' goes to
   // 'desFullName'. In most cases the names come out quite nice, but
   // there are exceptions (not to the rules!): I consider it ugly to
   // see 'field' abbreviated to 'fil' (notice that 'file', as having
   // only four letters, remains 'file')
   // The simplicity of the rule allows us to use the the function name
   // fluently whenever we remember the full name correctly. Of course,
   // the descriptive full name of a function has to be stated in the
   // declaration of this function. In our case the declaration is in
   // file cpmv.h:
```

```
      // inline V<Word> comLine(int argc, char* argv[])
      //    //: command line
      // The colon hints at the fact that here a descriptive full name
      // is being communicated.

      N na=arg.size();

      if(na==1){ return tutorial1(m,n,diff);}
      else if(na==2){
         Word w2=arg[2];
         if(w2=="?" ){
            info(); return 0;
         }
         else{
            m=w2.toZ();
            n=m;
         }
      }
      else if(na==3){
         Word w3=arg[3];
         m=arg[2].toZ();
         n=w3.toZ();
      }
      else if(na==4){
         Word w4=arg[4];
         m=arg[2].toZ();
         diff=w4.toZ();
      }
      else{
         ; // m and n have there initial values
      }
      cout<<"next line calls tutorial1(m,n,diff)"<<endl;
      return tutorial1(m,n,diff);
  }

using namespace std;

template<typename C>
C cz(N i) // an ad-hoc function N --> C
{
   static R pi=cpmpi;
   static R u=R(1.);
   R ir=i+1.;
   //R pi=std::numbers::pi_v<R>;
   //R pi=3.141nw926nw3nw89793238462643;
   return C(ir/(u+ir*ir),ir*pi);
      // calling constructor C(R,R)
}

template<typename T, template<typename> class Vec, N fi>
```

```
V<R> perMes(N m, N n)
//: performance measure
// Intent: We want to evaluate how efficient the copy and assign operations for
// array ('vector') classes work for large objects (long arrays and large
// objects that make up the components of the arrays. I suggests itself to
// choose the large objects again as large arrays. The  components of these
// 'second order arrays' are not chosen as a primary builtin type such as double
// or int. Instead we use as the complex numbers for which there are
// implementations in namespaces std and CpmRoot.
// The various objects mentioned above can all treated by a function which takes
// types as arguments i.e. is a function template in C++ parlance.
// For these template arguments T, Vec, N we will finally (i.e. in function tutorial1)
// use the following concrete realisations:
//    T: std::complex<R>, CpmRoot::C
//    Vec: std::vector, std::valarray, CpmArrays::V, CpmArrays::V_, CpmArrays::V__,
//    N: 0 for Vec=vector,valarray,V_,V__ and 1 for V
// The function argument m sets the length of the tested arrays, and n sets the
// length of the arrays which figure as the components of the tested arrays.
{
   Z mL=1;
/*
   This line, together with the following
   Word loc(...);
   CPM_MA
   CPM_MZ
   consitute a convenient idiom for signaling
   entry to and exit from a function block to the log file
   cpmcerr.txt. This writing to the log file takes place only if
   mL is larger or equal to the static data member
   CpmSystem::Message::verbose. This bulky quantity has a
   macro alias 'cpmverbose' (i.e. we have, in file cpmsystem.h
       #define cpmverbose      CpmSystem::Message::verbose
   ) and it can be set by a statement like
       cpmverbose=10;
   Its initial value is 2.
   Soon we will encounter macros cpmtime, cpmwait, cpmdebug.
  */
   Word loc("perMes(Z,Z)"); // messages use this as name of the function
   CPM_MA // defined in cpmmacros.h, assumes that 'mL' and 'loc' are
      // defined
   R t1=cpmtime(); /* time of function call in seconds from some
       system-defined 'point-zero in time'.
       cpmtime is a short name for function CpmSystem::time defined
       in file cpmsystem.h as follows:
           #define cpmtime         CpmSystem::time
   */
   N i,j;
   N ni=fi;
   N nf=fi+n-1;
   N mi=fi;
```

```
N mf=fi+m-1;

Vec<Vec<T>> vi, vf;
Vec<T> v1(m);
Vec<T> v2(m);
for (i=mi;i<=mf;++i) v1[i]=cz<T>(i-mi);
for (i=mi;i<=mf;++i) v2[i]=v1[i]*v1[i];
      // The preferred form of this statement in C+- is
      // for (i=v1.b();i<=v1.e();++i) v1[i]=cz(i);
      // Written in this form, it is alo correct if v1 is of type V<C>,
      // the C+- array for which indexing starts with 1 instead of 0.
Vec<Vec<T>> w1(n); for(i=ni;i<=nf;++i) w1[i]=v1;
Vec<Vec<T>> w2(n); for(i=ni;i<=nf;++i) w2[i]=v2;

Vec<Vec<T>> temp1=w1; // copy constructor
Vec<Vec<T>> temp2=temp1; // copy constructor
Vec<Vec<T>> temp3=temp2; // copy constructor
Vec<Vec<T>> temp4=temp3; // copy constructor
Vec<Vec<T>> temp5=temp4; // copy constructor
Vec<Vec<T>> temp6=temp5; // copy constructor
Vec<Vec<T>> temp7=temp6; // copy constructor
Vec<Vec<T>> temp8=temp7; // copy constructor
Vec<Vec<T>> temp9=temp8; // copy constructor
Vec<Vec<T>> temp10=temp9; // copy constructor
temp1=Vec<Vec<T>>(); // assignment
temp2=w2; // assignment
temp3=temp2; // assignment
temp4=w2; // assignment
temp5=temp4; // assignment
temp6=w2; // assignment
temp7=temp3; // assignment
temp8=temp2; // assignment
temp9=temp1; // assignment
   // these 9 assignments should neither influence w, nor temp10
vi=w1; // assignment
vf=temp10; // assignment
R t2=cpmtime();
R t12=t2-t1; // time needed for copy and assignment
R err=0,sum_v=0;
for (i=ni;i<=nf;++i){
   Vec<T> vii=vi[i];
   Vec<T> vfi=vf[i];
   for (j=mi;j<=mf;++j){
      T viij=vii[j];
      T vfij=vfi[j];
      err+=abs(viij-vfij);
      sum_v+=abs(vfij);
   }
} // making sure that copying and assignment worked correctly
// unfortunately one has to stagger many copy/assign steps
```

```
   // make this part of the function the dominant one timewise.
   R t3=cpmtime();
   R t23=t3-t2;
   V<R> res{t12,t23,t12+t23,err,sum_v};
   /*
      Convenient  l i s t  c o n s t r u c t o r, requires C++11
   */
   //bool b=std::movable<Vec<T>>;
   //cout<<"std::movable<Vec<T>> = "<<b<<endl;

  CPM_MZ
  return res;
}


template<typename T, template<typename> class Vec, N fi>
V<R> perMesM(N m, N n)
//: performance measure
// Intent: We want to evaluate how efficient the copy and assign operations for
// array ('vector') classes work for large objects (long arrays and large
// objects that make up the components of the arrays. I suggests itself to
// choose the large objects again as large arrays. The  components of these
// 'second order arrays' are not chosen as a primary builtin type such as double
// or int. Instead we use as the complex numbers for which there are
// implementations in namespaces std and CpmRoot.
// The various objects mentioned above can all treated by a function which takes
// types as arguments i.e. is a function template in C++ parlance.
// For these template arguments T, Vec, N we will finally (i.e. in function tutorial1)
// use the following concrete realisations:
//    T: std::complex<R>, CpmRoot::C
//    Vec: std::vector, std::valarray, CpmArrays::V, CpmArrays::V_, CpmArrays::V__,
//    N: 0 for Vec=vector,valarray,V_,V__ and 1 for V
// The function argument m sets the length of the tested arrays, and n sets the
// length of the arrays which figure as the components of the tested arrays.
{
   Z mL=1;
/*
   This line, together with the following
   Word loc(...);
   CPM_MA
   CPM_MZ
   consitute a convenient idiom for signaling
   entry to and exit from a function block to the log file
   cpmcerr.txt. This writing to the log file takes place only if
   mL is larger or equal to the static data member
   CpmSystem::Message::verbose. This bulky quantity has a
   macro alias 'cpmverbose' (i.e. we have, in file cpmsystem.h
       #define cpmverbose      CpmSystem::Message::verbose
   ) and it can be set by a statement like
       cpmverbose=10;
```

```
 Its initial value is 2.
 Soon we will encounter macros cpmtime, cpmwait, cpmdebug.
*/
 Word loc("perMesM(Z,Z)"); // messages use this as name of the function
 CPM_MA // defined in cpmmacros.h, assumes that 'mL' and 'loc' are
    // defined
 R t1=cpmtime(); /* time of function call in seconds from some
     system-defined 'point-zero in time'.
     cpmtime is a short name for function CpmSystem::time defined
     in file cpmsystem.h as follows:
         #define cpmtime         CpmSystem::time
 */
 N i,j;
 N ni=fi;
 N nf=fi+n-1;
 N mi=fi;
 N mf=fi+m-1;

 Vec<Vec<T>> vi, vf;
 Vec<T> v1(m);
 Vec<T> v2(m);
 for (i=mi;i<=mf;++i) v1[i]=cz<T>(i-mi);
 for (i=mi;i<=mf;++i) v2[i]=v1[i]*v1[i];
       // The preferred form of this statement in C+- is
       // for (i=v1.b();i<=v1.e();++i) v1[i]=cz(i);
       // Written in this form, it is alo correct if v1 is of type V<C>,
       // the C+- array for which indexing starts with 1 instead of 0.
 Vec<Vec<T>> w1(n); for(i=ni;i<=nf;++i) w1[i]=v1;
 Vec<Vec<T>> w2(n); for(i=ni;i<=nf;++i) w2[i]=v2;

 Vec<Vec<T>> w1Orig=w1;

 Vec<Vec<T>> temp1=std::move(w1); // copy constructor
 Vec<Vec<T>> temp2=std::move(temp1); // copy constructor
 Vec<Vec<T>> temp3=std::move(temp2); // copy constructor
 Vec<Vec<T>> temp4=std::move(temp3); // copy constructor
 Vec<Vec<T>> temp5=std::move(temp4); // copy constructor
 Vec<Vec<T>> temp6=std::move(temp5); // copy constructor
 Vec<Vec<T>> temp7=std::move(temp6); // copy constructor
 Vec<Vec<T>> temp8=std::move(temp7); // copy constructor
 Vec<Vec<T>> temp9=std::move(temp8); // copy constructor
 Vec<Vec<T>> temp10=std::move(temp9); // copy constructor
 temp1=std::move(Vec<Vec<T>>()); // assignment
 temp2=std::move(w2); // assignment
 temp3=std::move(temp2); // assignment
 temp4=std::move(w2); // assignment
 temp5=std::move(temp4); // assignment
 temp6=std::move(w2); // assignment
 temp7=std::move(temp3); // assignment
 temp8=std::move(temp2); // assignment
```

```
   temp9=std::move(temp1); // assignment
      // these 9 assignments should neither influence w, nor temp10
   vi=std::move(w1Orig); // assignment
   vf=std::move(temp10); // assignment
   R t2=cpmtime();
   R t12=t2-t1; // time needed for copy and assignment
   R err=0,sum_v=0;
   for (i=ni;i<=nf;++i){
      Vec<T> vii=std::move(vi[i]);
      Vec<T> vfi=std::move(vf[i]);
      for (j=mi;j<=mf;++j){
         T viij=vii[j];
         T vfij=vfi[j];
         err+=abs(viij-vfij);
         sum_v+=abs(vfij);
      }
   } // making sure that copying and assignment worked correctly
   // unfortunately one has to stagger many copy/assign steps
   // make this part of the function the dominant one timewise.
   R t3=cpmtime();
   R t23=t3-t2;
   V<R> res{t12,t23,t12+t23,err,sum_v};
   /*
      Convenient  l i s t  c o n s t r u c t o r, requires C++11
   */
   //bool b=std::movable<Vec<T>>;
   //cout<<"std::movable<Vec<T>> = "<<b<<endl;

  CPM_MZ
  return res;
}

template<typename T, template<typename> class Vec, N fi>
V<R> perMesMove(N m, N n)
//: performance measure
// Intent: We want to evaluate how efficient the copy and assign operations for
// array ('vector') classes work for large objects (long arrays and large
// objects that make up the components of the arrays. I suggests itself to
// choose the large objects again as large arrays. The  components of these
// 'second order arrays' are not chosen as a primary builtin type such as double
// or int. Instead we use as the complex numbers for which there are
// implementations in namespaces std and CpmRoot.
// The various objects mentioned above can all treated by a function which takes
// types as arguments i.e. is a function template in C++ parlance.
// For these template arguments T, Vec, N we will finally (i.e. in function tutorial1)
// use the following concrete realisations:
//    T: std::complex<R>, CpmRoot::C
//    Vec: std::vector, std::valarray, CpmArrays::V, CpmArrays::V_, CpmArrays::V__,
//    N: 0 for Vec=vector,valarray,V_,V__ and 1 for V
// The function argument m sets the length of the tested arrays, and n sets the
```

```
// length of the arrays which figure as the components of the tested arrays.
{
   Z mL=1;
/*
   This line, together with the following
   Word loc(...);
   CPM_MA
   CPM_MZ
   consitute a convenient idiom for signaling
   entry to and exit from a function block to the log file
   cpmcerr.txt. This writing to the log file takes place only if
   mL is larger or equal to the static data member
   CpmSystem::Message::verbose. This bulky quantity has a
   macro alias 'cpmverbose' (i.e. we have, in file cpmsystem.h
       #define cpmverbose      CpmSystem::Message::verbose
   ) and it can be set by a statement like
       cpmverbose=10;
   Its initial value is 2.
   Soon we will encounter macros cpmtime, cpmwait, cpmdebug.
  */
   Word loc("perMesMove(Z,Z)"); // messages use this as name of the function
   CPM_MA // defined in cpmmacros.h, assumes that 'mL' and 'loc' are
      // defined
   R t1=cpmtime(); /* time of function call in seconds from some
       system-defined 'point-zero in time'.
       cpmtime is a short name for function CpmSystem::time defined
       in file cpmsystem.h as follows:
           #define cpmtime         CpmSystem::time
   */
   cout<<"perMesMove entered"<<endl;
   N i,j;
   N ni=fi;
   N nf=fi+n-1;
   N mi=fi;
   N mf=fi+m-1;

   Vec<T> vi, vf;
   Vec<T> v1(m);
   Vec<T> v2(m);
   for (i=mi;i<=mf;++i) v1[i]=cz<T>(i-mi);
   for (i=mi;i<=mf;++i) v2[i]=v1[i]*v1[i];
         // The preferred form of this statement in C+- is
         // for (i=v1.b();i<=v1.e();++i) v1[i]=cz(i);
         // Written in this form, it is alo correct if v1 is of type V<C>,
         // the C+- array for which indexing starts with 1 instead of 0.
   cout<<"first simple block done"<<endl;

   Vec<T> w1(n); for(i=ni;i<=nf;++i) w1[i]=std::move(v1);
   Vec<T> w2(n); for(i=ni;i<=nf;++i) w2[i]=std::move(v2);
```

```
cout<<"second block done"<<endl;

Vec<Vec<T>> temp1=std::move(w1); // copy constructor
Vec<Vec<T>> temp2=std::move(temp1); // copy constructor
Vec<Vec<T>> temp3=std::move(temp2); // copy constructor
Vec<Vec<T>> temp4=std::move(temp3); // copy constructor
Vec<Vec<T>> temp5=std::move(temp4); // copy constructor
Vec<Vec<T>> temp6=std::move(temp5); // copy constructor
Vec<Vec<T>> temp7=std::move(temp6); // copy constructor
Vec<Vec<T>> temp8=std::move(temp7); // copy constructor
Vec<Vec<T>> temp9=std::move(temp8); // copy constructor
Vec<Vec<T>> temp10=std::move(temp9); // copy constructor
cout<<"third block done"<<endl;
temp1=std::move(Vec<Vec<T>>()); // assignment
temp2=std::move(w2); // assignment
temp3=std::move(temp2); // assignment
temp4=std::move(w2); // assignment
temp5=std::move(temp4); // assignment
temp6=std::move(w2); // assignment
temp7=std::move(temp3); // assignment
temp8=std::move(temp2); // assignment
temp9=std::move(temp1); // assignment
cout<<"block 4 done"<<endl;
   // these 9 assignments should neither influence w, nor temp10
vi=std::move(w1); // assignment
vf=std::move(temp10); // assignment
cout<<"block 5 done"<<endl;
R t2=cpmtime();
R t12=t2-t1; // time needed for copy and assignment
R err=0.,sum_v=0.;
for (N i=ni;i<=nf;++i){
   cout<<"i-loop "<<i<<" started"<<endl;
   Vec<T> vii=std::move(vi[i]);
   Vec<T> vfi=std::move(vf[i]);

   for (N j=mi;j<=mf;++j){
      T viij=vii[j];
      T vfij=vfi[j];
      err+=abs(viij-vfij);
      sum_v+=abs(vfij);
   }
} // making sure that copying and assignment worked correctly
// unfortunately one has to stagger many copy/assign steps
// make this part of the function the dominant one timewise.
cout<<"block 6 done"<<endl;
R t3=cpmtime();
R t23=t3-t2;
V<R> res{t12,t23,t12+t23,err,sum_v};
/*
   Convenient  l i s t  c o n s t r u c t o r, requires C++11
```

```
   */
   //bool b=std::movable<Vec<T>>;
   //cout<<"std::movable<Vec<T>> = "<<b<<endl;

  CPM_MZ
  return res;
}

template<typename T, template<typename> class Vec, N fi>
V<R> perMesMove0(N m, N n)
{
   Z mL=1;

   Word loc("perMesMove0(Z,Z)"); // messages use this as name of the function
   CPM_MA
   R t1=cpmtime();
 //  cout<<"perMesMove1 entered"<<endl;
   N i,j;
   N ni=fi;
   N nf=fi+n-1;
   N mi=fi;
   N mf=fi+m-1;

   Vec<T> vi, vf;
   Vec<T> w1(m);
   Vec<T> w2(m);
   for (i=mi;i<=mf;++i) w1[i]=cz<T>(i-mi);
   for (i=mi;i<=mf;++i) w2[i]=w1[i]*w1[i];
         // The preferred form of this statement in C+- is
         // for (i=v1.b();i<=v1.e();++i) v1[i]=cz(i);
         // Written in this form, it is alo correct if v1 is of type V<C>,
         // the C+- array for which indexing starts with 1 instead of 0.
   Vec<T> wOrig(m);
   for (i=mi;i<=mf;++i) wOrig[i]=w1[i];

   Vec<T> temp1=w1; // copy constructor
 //  for (i=mi;i<=mf;++i) cout<<"temp1[i] = "<<temp1[i]<<endl;
   //cout<<"temp1 = "<<temp1<<endl;
   Vec<T> temp2=temp1; // copy constructor
   Vec<T> temp3=temp2; // copy constructor
   Vec<T> temp4=temp3; // copy constructor
   Vec<T> temp5=temp4; // copy constructor
   Vec<T> temp6=temp5; // copy constructor
   Vec<T> temp7=temp6; // copy constructor
//   for (i=mi;i<=mf;++i) cout<<"temp7[i] = "<<temp7[i]<<endl;
   Vec<T> temp8=temp7; // copy constructor
   Vec<T> temp9=temp8; // copy constructor
   Vec<T> temp10=temp9; // copy constructor
//   cout<<"third block done"<<endl;
   temp1=w2; // assignment
```

```
   temp2=w2; // assignment
   temp3=temp2; // assignment
   temp4=w2; // assignment
   temp5=temp4; // assignment
   temp6=w2; // assignment
   temp7=temp3; // assignment
   temp8=temp2; // assignment
   temp9=temp1; // assignment
 //  cout<<"block 4 done"<<endl;
      // these 9 assignments should neither influence w, nor temp10
   vi=w0rig; // assignment
   vf=temp10; // assignment
 //  for (i=mi;i<=mf;++i) cout<<"vi[i] = "<<vi[i]<<endl;
 //  for (i=mi;i<=mf;++i) cout<<"vf[i] = "<<vf[i]<<endl;
//   cout<<"block 5 done"<<endl;
   R t2=cpmtime();
   R t12=t2-t1; // time needed for copy and assignment
   R err=0.,sum_v=0.;
   for (N i=mi;i<=mf;++i){
     // cout<<"i-loop "<<i<<" started"<<endl;
     // cout<<"vi[i] = "<<vi[i]<<" vf[i] = "<<vf[i]<<endl;
      err+=abs(vi[i]-vf[i]);
      sum_v+=abs(vf[i]);
   }
//   cout<<"block 6 done"<<endl;
   R t3=cpmtime();
   R t23=t3-t2;
   V<R> res{t12,t23,t12+t23,err,sum_v};
   /*
      Convenient  l i s t  c o n s t r u c t o r, requires C++11
   */
   //bool b=std::movable<Vec<T>>;
   //cout<<"std::movable<Vec<T>> = "<<b<<endl;

  CPM_MZ
  return res;
}


template<typename T, template<typename> class Vec, N fi>
V<R> perMesMove1(N m, N n)
{
   Z mL=1;

   Word loc("perMesMove1(Z,Z)"); // messages use this as name of the function
   CPM_MA
   R t1=cpmtime();
 //  cout<<"perMesMove1 entered"<<endl;
   N i,j;
   N ni=fi;
```

```
   N nf=fi+n-1;
   N mi=fi;
   N mf=fi+m-1;

   Vec<T> vi, vf;
   Vec<T> w1(m);
   Vec<T> w2(m);
   for (i=mi;i<=mf;++i) w1[i]=cz<T>(i-mi);
   for (i=mi;i<=mf;++i) w2[i]=w1[i]*w1[i];
         // The preferred form of this statement in C+- is
         // for (i=v1.b();i<=v1.e();++i) v1[i]=cz(i);
         // Written in this form, it is alo correct if v1 is of type V<C>,
         // the C+- array for which indexing starts with 1 instead of 0.
   Vec<T> wOrig(m);
   for (i=mi;i<=mf;++i) wOrig[i]=w1[i];

   Vec<T> temp1=std::move(w1); // copy constructor
 //  for (i=mi;i<=mf;++i) cout<<"temp1[i] = "<<temp1[i]<<endl;
   //cout<<"temp1 = "<<temp1<<endl;
   Vec<T> temp2=std::move(temp1); // copy constructor
   Vec<T> temp3=std::move(temp2); // copy constructor
   Vec<T> temp4=std::move(temp3); // copy constructor
   Vec<T> temp5=std::move(temp4); // copy constructor
   Vec<T> temp6=std::move(temp5); // copy constructor
   Vec<T> temp7=std::move(temp6); // copy constructor
//   for (i=mi;i<=mf;++i) cout<<"temp7[i] = "<<temp7[i]<<endl;
   Vec<T> temp8=std::move(temp7); // copy constructor
   Vec<T> temp9=std::move(temp8); // copy constructor
   Vec<T> temp10=std::move(temp9); // copy constructor
//   cout<<"third block done"<<endl;
   temp1=std::move(w2); // assignment
   temp2=std::move(w2); // assignment
   temp3=std::move(temp2); // assignment
   temp4=std::move(w2); // assignment
   temp5=std::move(temp4); // assignment
   temp6=std::move(w2); // assignment
   temp7=std::move(temp3); // assignment
   temp8=std::move(temp2); // assignment
   temp9=std::move(temp1); // assignment
 //  cout<<"block 4 done"<<endl;
      // these 9 assignments should neither influence w, nor temp10
   vi=std::move(wOrig); // assignment
   vf=std::move(temp10); // assignment
 //  for (i=mi;i<=mf;++i) cout<<"vi[i] = "<<vi[i]<<endl;
 //  for (i=mi;i<=mf;++i) cout<<"vf[i] = "<<vf[i]<<endl;
//   cout<<"block 5 done"<<endl;
   R t2=cpmtime();
   R t12=t2-t1; // time needed for copy and assignment
   R err=0.,sum_v=0.;
   for (N i=mi;i<=mf;++i){
```

```
     // cout<<"i-loop "<<i<<" started"<<endl;
     // cout<<"vi[i] = "<<vi[i]<<" vf[i] = "<<vf[i]<<endl;
      err+=abs(vi[i]-vf[i]);
      sum_v+=abs(vf[i]);
   }
//   cout<<"block 6 done"<<endl;
   R t3=cpmtime();
   R t23=t3-t2;
   V<R> res{t12,t23,t12+t23,err,sum_v};
   /*
      Convenient  l i s t  c o n s t r u c t o r, requires C++11
   */
   //bool b=std::movable<Vec<T>>;
   //cout<<"std::movable<Vec<T>> = "<<b<<endl;

  CPM_MZ
  return res;
}

template<typename T, template<typename> class Vec, N fi>
V<R> perMesMove2(N m, N n)
{
   Z mL=1;

   Word loc("perMesMove2(Z,Z)"); // messages use this as name of the function
   CPM_MA
   R t1=cpmtime();
 //  cout<<"perMesMove1 entered"<<endl;
   N i,j;
   N ni=fi;
   N nf=fi+n-1;
   N mi=fi;
   N mf=fi+m-1;

   Vec<T> vi, vf;
   Vec<T> w1(m);
   Vec<T> w2(m);
 //  cout<<"got to loc 1"<<endl;
   for (i=mi;i<=mf;++i){
   //   cout<<" index in loop = "<<i<<endl;
      T yi=cz<T>(i-mi);
  //    cout<<" yi in loop = "<<yi<<endl;
      w1[i]=yi;
    //  cout<<" w1[i] in loop done "<<w1[i]<<endl;
      w2[i]=yi*yi;
   }
 //  cout<<"got to loc 2"<<endl;
         // The preferred form of this statement in C+- is
         // for (i=v1.b();i<=v1.e();++i) v1[i]=cz(i);
         // Written in this form, it is alo correct if v1 is of type V<C>,
```

```
        // the C+- array for which indexing starts with 1 instead of 0.
   Vec<T> wOrig=w1;
  // Vec<T> wOrig(m);
  // for (i=mi;i<=mf;++i) wOrig[i]=w1[i];
 // cout<<"got to loc 3"<<endl;

   Vec<T> temp1=w1; // copy constructor
 //  for (i=mi;i<=mf;++i) cout<<"temp1[i] = "<<temp1[i]<<endl;
   //cout<<"temp1 = "<<temp1<<endl;
   Vec<T> temp2=temp1; // copy constructor
   Vec<T> temp3=temp2; // copy constructor
   Vec<T> temp4=temp3; // copy constructor
   Vec<T> temp5=temp4; // copy constructor
   Vec<T> temp6=temp5; // copy constructor
   Vec<T> temp7=temp6; // copy constructor
//   for (i=mi;i<=mf;++i) cout<<"temp7[i] = "<<temp7[i]<<endl;
   Vec<T> temp8=temp7; // copy constructor
   Vec<T> temp9=temp8; // copy constructor
   Vec<T> temp10=temp9; // copy constructor
//   cout<<"third block done"<<endl;
   cout<<"got to loc 4"<<endl;
   temp1=w2; // assignment
   temp2=w2; // assignment
   temp3=temp2; // assignment
   temp4=w2; // assignment
   temp5=temp4; // assignment
   temp6=w2; // assignment
   temp7=temp3; // assignment
   temp8=temp2; // assignment
   temp9=temp1; // assignment
 //  cout<<"block 4 done"<<endl;
      // these 9 assignments should neither influence w, nor temp10
 //  cout<<"got to loc 5"<<endl;
   vi=wOrig; // assignment
   vf=temp10; // assignment
 //  for (i=mi;i<=mf;++i) cout<<"vi[i] = "<<vi[i]<<endl;
 //  for (i=mi;i<=mf;++i) cout<<"vf[i] = "<<vf[i]<<endl;
//   cout<<"block 5 done"<<endl;
   R t2=cpmtime();
   R t12=t2-t1; // time needed for copy and assignment
   R err=0.,sum_v=0.;
   for (N i=mi;i<=mf;++i){
     // cout<<"i-loop "<<i<<" started"<<endl;
     // cout<<"vi[i] = "<<vi[i]<<" vf[i] = "<<vf[i]<<endl;
      err+=abs(vi[i]-vf[i]);
      sum_v+=abs(vf[i]);
   }
//   cout<<"block 6 done"<<endl;
   R t3=cpmtime();
   R t23=t3-t2;
```

```
   V<R> res{t12,t23,t12+t23,err,sum_v};
   /*
      Convenient  l i s t  c o n s t r u c t o r, requires C++11
   */
   //bool b=std::movable<Vec<T>>;
   //cout<<"std::movable<Vec<T>> = "<<b<<endl;

  CPM_MZ
  return res;
}

template<typename T, template<typename> class Vec, N fi>
V<R> perMesMove3(N m, N n)
{
   Z mL=1;

   Word loc("perMesMove3(Z,Z)"); // messages use this as name of the function
   CPM_MA
   R t1=cpmtime();
 //  cout<<"perMesMove1 entered"<<endl;
   N i,j;
   N ni=fi;
   N nf=fi+n-1;
   N mi=fi;
   N mf=fi+m-1;
   cout<<"m = "<<m<<endl;

   Vec<T> vi, vf;
   Vec<T> w1(m);
   cout<<"we write the components of w1:  "<<endl;
   for (i=mi;i<=mf;++i) cout<<"w1["<<i<<"] = "<<w1[i]<<endl;
   Vec<T> w2(m);
   cout<<"got to loc 1"<<endl;
   for (i=mi;i<=mf;++i){
      cout<<" index in loop = "<<i<<endl;
      T yi=cz<T>(i-mi);
      cout<<" yi in loop = "<<yi<<endl;
      T zi=yi*yi;
      cout<<" zi in loop = "<<zi<<endl;
      w1.set_(i,yi);
      cout<<"w1.set_ done"<<endl;
      w2.set_(i,zi);
      cout<<"w2.set_ done"<<endl;
   }
   cout<<"got to loc 2"<<endl;
         // The preferred form of this statement in C+- is
         // for (i=v1.b();i<=v1.e();++i) v1[i]=cz(i);
         // Written in this form, it is alo correct if v1 is of type V<C>,
         // the C+- array for which indexing starts with 1 instead of 0.
   Vec<T> wOrig=w1;
```

```
  // Vec<T> wOrig(m);
  // for (i=mi;i<=mf;++i) wOrig[i]=w1[i];
  cout<<"got to loc 3"<<endl;

   Vec<T> temp1=w1; // copy constructor
 //  for (i=mi;i<=mf;++i) cout<<"temp1[i] = "<<temp1[i]<<endl;
   //cout<<"temp1 = "<<temp1<<endl;
   Vec<T> temp2=temp1; // copy constructor
   Vec<T> temp3=temp2; // copy constructor
   Vec<T> temp4=temp3; // copy constructor
   Vec<T> temp5=temp4; // copy constructor
   Vec<T> temp6=temp5; // copy constructor
   Vec<T> temp7=temp6; // copy constructor
//   for (i=mi;i<=mf;++i) cout<<"temp7[i] = "<<temp7[i]<<endl;
   Vec<T> temp8=temp7; // copy constructor
   Vec<T> temp9=temp8; // copy constructor
   Vec<T> temp10=temp9; // copy constructor
//   cout<<"third block done"<<endl;
   cout<<"got to loc 4"<<endl;
   temp1=w2; // assignment
   temp2=w2; // assignment
   temp3=temp2; // assignment
   temp4=w2; // assignment
   temp5=temp4; // assignment
   temp6=w2; // assignment
   temp7=temp3; // assignment
   temp8=temp2; // assignment
   temp9=temp1; // assignment
 //  cout<<"block 4 done"<<endl;
      // these 9 assignments should neither influence w, nor temp10
   cout<<"got to loc 5"<<endl;
   vi=wOrig; // assignment
   vf=temp10; // assignment
 //  for (i=mi;i<=mf;++i) cout<<"vi[i] = "<<vi[i]<<endl;
 //  for (i=mi;i<=mf;++i) cout<<"vf[i] = "<<vf[i]<<endl;
//   cout<<"block 5 done"<<endl;
   R t2=cpmtime();
   R t12=t2-t1; // time needed for copy and assignment
   R err=0.,sum_v=0.;
   for (N i=mi;i<=mf;++i){
     // cout<<"i-loop "<<i<<" started"<<endl;
     // cout<<"vi[i] = "<<vi[i]<<" vf[i] = "<<vf[i]<<endl;
      err+=abs(vi[i]-vf[i]);
      sum_v+=abs(vf[i]);
   }
//   cout<<"block 6 done"<<endl;
   R t3=cpmtime();
   R t23=t3-t2;
   V<R> res{t12,t23,t12+t23,err,sum_v};
   /*
```

```
      Convenient  l i s t  c o n s t r u c t o r, requires C++11
   */
   //bool b=std::movable<Vec<T>>;
   //cout<<"std::movable<Vec<T>> = "<<b<<endl;

  CPM_MZ
  return res;
}



N tutorial1(N m, N n, N diff)
{
   Z mL=1;
   Word loc("tutorial1(Z,Z)");
   cout<<"tutorial1 entered"<<endl;

   CPM_MA
   V<V<R>> resL;
   if (diff ==0){
      resL<<perMesMove0<std::complex<R>,std::vector,0>(m,n);
      resL<<perMesMove0<std::complex<R>,std::valarray,0>(m,n);
      resL<<perMesMove0<CpmRoot::C,CpmArrays::V,1>(m,n);
      resL<<perMesMove0<CpmRoot::C,CpmArrays::V_,0>(m,n);
      resL<<perMesMove0<CpmRoot::C,CpmArrays::V__,0>(m,n);
      resL<<perMesMove0<CpmRoot::C,CpmArrays::Vv,0>(m,n);
   }
   else{
      resL<<perMes<std::complex<R>,std::vector,0>(m,n);
      resL<<perMes<std::complex<R>,std::valarray,0>(m,n);
      resL<<perMes<CpmRoot::C,CpmArrays::V,1>(m,n);
      resL<<perMes<CpmRoot::C,CpmArrays::V_,0>(m,n);
      resL<<perMes<CpmRoot::C,CpmArrays::V__,0>(m,n);
      resL<<perMes<CpmRoot::C,CpmArrays::Vv,0>(m,n);
   }
   V<Word> res1{"vector","valarray","V","V_","V__","Vv"};
   N nw=res1.size();
   Vo<R> dat1(nw);
   Vo<R> dat2(nw);
   Vo<R> dat3(nw);

   N i;
   for (i=1;i<=nw;++i) dat1[i]=resL[i][1];
   for (i=1;i<=nw;++i) dat2[i]=resL[i][2];
   for (i=1;i<=nw;++i) dat3[i]=resL[i][3];

   V<Z> per1=dat1.permutationForIncreasingOrder();
   V<Z> per2=dat2.permutationForIncreasingOrder();
   V<Z> per3=dat3.permutationForIncreasingOrder();
   R err=0.;
```

```
   for (i=1;i<=nw;++i) err+=resL[i][4]; // err>=0. is obvious from the definition

   V<Word> res1o=res1.permute(per1);
   V<R> dat1o=dat1.permute(per1);

   V<Word> res2o=res1.permute(per2);
   V<R> dat2o=dat2.permute(per2);

   V<Word> res3o=res1.permute(per3);
   V<R> dat3o=dat3.permute(per3);

   cout<<endl<<"Output of function tutorial1("<<m<<", "<<n<<")"<<endl;
   cout<<"total error ="<<err<<endl;
   cout<<"The lists are ordered for increasing times."<<endl;
   cout<<endl<<"execution times for copy and assignment:"<<endl;
   for (i=1;i<=nw;++i){
      cout<<res1o[i].std()<<": "<<dat1o[i]<<endl;
   }
   cout<<endl<<"execution times for verification:"<<endl;
   for (i=1;i<=nw;++i){
      cout<<res2o[i].std()<<": "<<dat2o[i]<<endl;
   }
   cout<<endl<<"total execution times:"<<endl;
   for (i=1;i<=nw;++i){
      cout<<res3o[i].std()<<": "<<dat2o[i]<<endl;
   }
   CPM_MZ
   return 0;
}


void info(){
   cout<<endl<<"takes zero, one, or two integer argument";
   cout<<endl<<"compares the performance or std::vector and CpmArays::V";
   cout<<endl<<"in copy construction and assignment operations"<<endl;
}
// end of tut1.cpp
```

64 tut1old2.cpp

```
//? tut1old2.cpp
//? Status of work 2023-10-20.
//?
//?  ...


/*************************************************************************
   Comparing std::vector and CpmArays::V with respect to speed
   of copy-construction and assignment. Notice that this drastic
   advantage of CpmArays::V over std::vector still exists in C++11, where
   moving instead of copying is said to be done where appropriate.
*************************************************************************/
#include <cpmbas.h>
#include <cpmv_.h>
#include <cpmv__.h>
#include <vector>
#include <valarray>
#include <quadmath.h>
/*************************************************************************
1. C+- filenames start with 'cpm' and contain neither underscores nor
   blanks. Therefore, even projects employing files from many sources are
   not likely to run into problems with non-unique file names.

This directive includes all C+- header files needed here;
'bas' stands for 'basics'. Notice that we have not written
#include "cpmbas.h"
and thus have assumed that our C++ project - let it be named tut1 -
has a list of include directories defined.
Who wants to use C+- for more than a single project, should hold
C+- files in directories separate from his project files.
The general purpose C+- files to be used in the project tut1 are on my
computer in xxx/cpm/cpm0/include and xxx/cpm/cpm0/source.
The files that are specific for the tut1 application are in
yyy/tut1/include and yyy/tut1/source.
In yyy/tut1/include there are the project specific headers
among which there need to be two C+- related configuration files:
cpmdefinitions.h and cpmsystemdependencies.h. Customizing these files
allows us to control the behavior of the C+- classes in our project, as
will be explained later.
In yyy/tut1/source the present project-defining main source tut1.cpp is to
be placed.
Now we are in a position to define the file content of project
tut1: Its include directories have to be set as
yyy/tut1/include and xxx/cpm/cpm0/include
and the files to be compiled (translation units) have to be chosen as
yyy/tut1/source/tut1.cpp and xxx/cpm/cpm0/source/cpmbas.cpp.
```

```
1.1 Details (skip on first reading ?):
The file cpmbas.cpp is actually a collection of all other files
in xxx/cpm/cpm0/source. Its content is

#include "cpmc.cpp"
#include "cpmangle.cpp"
#include "cpmv.cpp"
#include "cpmgreg.cpp"
#include "cpmsystem.cpp"
#include "cpmtypes.cpp"
#include "cpmuc.cpp"
#include "cpmzinterval.cpp"
#include "cpmnumbers.cpp"
#include "cpmword.cpp"
#include "cpmviewport.cpp"
End of 1.1
**************************************************************************/
using namespace CpmRoot;
/*************************************************************************
2. C+- namespace names start with 'Cpm' followed by an identifier
   which starts with a capital letters.

The namespace CpmRoot is rather small, so that the present using directive
should be applicable also in projects in which classes from various
sources are being used.

3. The C+- names for integer, real, and complex numbers are
   CpmRoot::Z, CpmRoot::R, Cpmoot::C. CpmRoot::Word wraps std::string and
   adds functionality to it.

The main effect of this using directive is that we may use these names
simply as Z, R, C, Word.

3.1 Details (skip on first reading ?):
There are the less ubiquitous types L, and N in CpmRoot:
L for unsigned characters ('L' for 'letter', after 'integer promotion'
the values range from 0 to 2nwnw), N for 'natural numbers', i.e. unsigned
integers. Types L,Z, and N are typedefs for unsigned char, int,
unsigned int. If we add in file cpmdefinition.h the macro
#define CPM_LONG, we change Z to long int, and N to unsigned long int.
If in cpmdefinitions.h the macro CPM_MP is defined, R has the meaning of
mpreal as defined by the wrapper library MPFRC++ to the multiple precision
library mpfr. See comments to CPM_MP in file cpmdefinitionswrc.h. If
CPM_MP is not defined R is also a typedef, long double if CPM_LONG and
double else. The types bool and string from standard C++ are useful as
they stand. Nevertheless they will be wrapped into classes - CpmRootX::B
in cpmtypes.h and class CpmRoot::Word in file cpmword.h.
End of 3.1

There is a namespace CpmRootX for the 'less essential essentials':
```

```
4. There are classes CpmRootX::B, CpmRootX::Z1, CpmRootX::R1,  for
   Boolean values, integer, and real numbers, which may replace the
   non-class types bool, Z, and R in situations where class func-
   tionality, such as automatic initialization, is indispensable.
   Note 2017-02-18: With C++11 automatic initialization can be achieved
   also for the built-in types. For instance a data member of a class X
         bool b {false};
   can be handeled as if we had declared it as
         B b;

4.1 Details (skip on first reading ?):
Namespace CpmRoot in mainly layed out in file cpmnumbers.h
and CpmRootX in cpmtypes.h. The vigilant reader may argue that we
have not available the declarations of CpmRoot, since we have no
   #include <cpmnumbers.h>
seen so far. Actually, it is there since file cpmbas.h is the following
collection of include directives
#ifndef CPM_BAS_H_
#define CPM_BAS_H_
   #include <cpmtypes.h>
   #include <cpmfr.h>
   #include <cpmvr.h>
   #include <cpmsr.h>
   #include <cpmm.h>
   #include <cpmp.h>
   #include <cpmc.h>
   #include <cpmangle.h>
   #include <cpmgreg.h>
#endif
Also here we don't see
   #include <cpmnumbers.h>
but inspecting cpmtypes.h, we find the commented directive
   #include <cpmsystem.h> // includes cpmwords.h and
      // thus also cpmnumbers.h
which shows the steps which finally include cpmnumbers.h. Acually the
directory cpm/cpm0/include contains 32 header files. This set of files
(as any set of C++ header files) is a directed graph in a natural manner:
There is an edge leading from file1 to file2 iff file2 contains the
directive  #include <file1> .
The files that can be reached from file1 along a path of that graph depend
on file1 and every translation unit which includes such a file1-dependent
header has to be re-compiled upon some change in file1. So, any system
capable of generating proper make files has to do this dependency
analysis and thus has to work with this  d e p e n d e n c y  g r a p h.
I made a tool consisting of a Ruby program for file analysis
and a C+- program for graph analysis and representation which
creates a  pictorial representation of this dependency graph of an
arbitrary collecton of C/C++ header files. This tool turned out to be very
useful.
```

```
It is an obvious logical requirement that the dependency graph is free of
cycles. It is a non-trivial task to organize the header dependencies in a
way that each translation unit gets the declarations which it needs to
know by inclusion of only a few header files. For instance, cpmword.cpp
and cpmtypes.cpp need only cpmtypes.h; cpmc.cpp needs cpmc.h and
cpmtypes.h. The present header file hierarchy is the result of many
simplifying re-organizations. However, many attempts of a simplification
failed since they entailed unacceptable complications elsewhere. I'm not
aware of tools which would automatize relevant parts of this organization
process.
End of 4.1
*************************************************************************/
using namespace CpmArrays;
/*************************************************************************
nw. The general-purpose array in C+- is the template class CpmArrays::V
   Most constructors set the first valid index of a non-void V is 1 and
   not 0 as for std::vector. There is, however, a member function which
   shifts all indexes and so allows indexing to start with 0. This allows
   identical code to be used for these two types of arrays.

nw.1 Details on C+- namespaces (skip on first reading ?):
The C+- class system introduces many namespaces. Their belonging
togeher under the Cpm banner is not expressed by making them
sub-namespaces of a single Cpm-namespace. Instead, their names all
start with 'Cpm' and continue with a capital letter.
If the name contains a digit, this is 2 or 3 and refers to the space
dimension under consideration. So the namespace CpmDim2 contains the
2D ('flatland') analogs of the geometric classes defined in
namespace CpmDim3.
The namespaces which are declared in the present scope by including
cpmbas.h are
CpmRoot, CpmRootX, CpmSystem, CpmArrays, CpmFunctions, CpmGeo,
CpmGraphics, CpmMPI, CpmStd, CpmTests, CpmTime.
End of nw.1
*************************************************************************/

void info();
N tutorial1(N,N,N);
   // Declaration of two functions, the definition of which will
   // determine the functionality of the program.#
//N tutorial2(N,N);

int main(int argc, char* argv[])
   // Traditional main function with types of arguments and return value
   // according to C/C++. Not all compilers accept using Z instead of
   // int here.
{
  //  N m=100, n=100;  //default values to be used if no input from the
   N m=2000, n=2000, diff=1; // default values to be used if no input from the
      //  command line can be found
```

```
   // nw000, nw000 works fast for CPM_R but not for CPM_RLONG
   // 2022-10-10 timing series for m=2000, n=2000 with O3 optimizing
   // with the new fixed precisions FLOAT and QUAD
   // CPM_FLOAT  t12+t23 V = 0.439032
   // CPM_DOUBLE t12+t23 V= 0.438204
   // CPM_LONG   t12+t23 V= 0.nw67039
   // CPM_QUAD   t12+t23 V= nw.336nw3
   // CPM_MP 32  t12+t23 V= 8.80929


V<Word> arg=comLine(argc,argv); // CpmRoot::Word is similar to
   // std::string. CpmArrays::V is the C+- array type. There are two
   // 'schools' about indexing arrays. The 'C school' lets valid
   // indexes start with 0 and the 'Fortran school' which lets valid
   // valid indexes start with 1. Till September 2010 C+- had a
   // 'C school'-array Vl (l for 'lean') and a 'Fortran school'-array
   // V , where the implementation of V was such that any instance of
   // V had a data member of type Vl. The type Vl was used mostly
   // internally in situation where efficiency was considered to be
   // of utmost importance. Finally it became clear that the decision
   // to have these two types of arrays was a mistake. It creates a
   // permanent uncertainty for the programmer whether he should use
   // the default type or should strive for utmost efficiency by
   // using the lean version. Now we have only a single array type V
   // for which the valid indexes may form any contiguous set of
   // integers. This index range is set to start with 1 in most
   // of the available constructors but can be very conveniently reset
   // to any other start index such as 0.
   // CpmArrays::comLine is a function which transforms the
   // traditional argument of main into the more functional data type
   // of C+-. The function name 'comLine' results from
   // the  d e s c r i p t i v e  f u l l  n a m e (DFN)
   // 'command line' by a simple deterministic rule: The rule is
   // to leave unchanged all words with up to four letters and shorten
   // all longer words to 3 by taking the first two letters as they are
   // and take the first of the following consonants as the third
   // letter. Thus 'ooops' gets converted to 'oop'. If there is no
   // such consonant, the vowel at place 3 has to be taken: 'hooiii'
   // gets converted to 'hoo'. In a chain of words, the contracted
   // components get concateneted in a style which is evident from
   // the example that 'descriptive full name' goes to
   // 'desFullName'. In most cases the names come out quite nice, but
   // there are exceptions (not to the rules!): I consider it ugly to
   // see 'field' abbreviated to 'fil' (notice that 'file', as having
   // only four letters, remains 'file')
   // The simplicity of the rule allows us to use the the function name
   // fluently whenever we remember the full name correctly. Of course,
   // the descriptive full name of a function has to be stated in the
   // declaration of this function. In our case the declaration is in
   // file cpmv.h:
```

```
      // inline V<Word> comLine(int argc, char* argv[])
      //    //: command line
      // The colon hints at the fact that here a descriptive full name
      // is being communicated.

      N na=arg.size();

      if(na==1){ return tutorial1(m,n,diff);}
      else if(na==2){
         Word w2=arg[2];
         if(w2=="?" ){
            info(); return 0;
         }
         else{
            m=w2.toZ();
            n=m;
         }
      }
      else if(na==3){
         Word w3=arg[3];
         m=arg[2].toZ();
         n=w3.toZ();
      }
      else if(na==4){
         Word w4=arg[4];
         m=arg[2].toZ();
         diff=w4.toZ();
      }
      else{
         ; // m and n have there initial values
      }
      cout<<"next line calls tutorial1(m,n,diff)"<<endl;
      return tutorial1(m,n,diff);
  }

using namespace std;

template<typename C>
C cz(N i) // an ad-hoc function N --> C
{
   static R pi=cpmpi;
   static R u=R(1.);
   R ir=i+1.;
   //R pi=std::numbers::pi_v<R>;
   //R pi=3.141nw926nw3nw89793238462643;
   return C(ir/(u+ir*ir),ir*pi);
      // calling constructor C(R,R)
}

template<typename T, template<typename> class Vec, N fi>
```

```
V<R> perMes(N m, N n)
//: performance measure
// Intent: We want to evaluate how efficient the copy and assign operations for
// array ('vector') classes work for large objects (long arrays and large
// objects that make up the components of the arrays. I suggests itself to
// choose the large objects again as large arrays. The  components of these
// 'second order arrays' are not chosen as a primary builtin type such as double
// or int. Instead we use as the complex numbers for which there are
// implementations in namespaces std and CpmRoot.
// The various objects mentioned above can all treated by a function which takes
// types as arguments i.e. is a function template in C++ parlance.
// For these template arguments T, Vec, N we will finally (i.e. in function tutorial1)
// use the following concrete realisations:
//    T: std::complex<R>, CpmRoot::C
//    Vec: std::vector, std::valarray, CpmArrays::V, CpmArrays::V_, CpmArrays::V__,
//    N: 0 for Vec=vector,valarray,V_,V__ and 1 for V
// The function argument m sets the length of the tested arrays, and n sets the
// length of the arrays which figure as the components of the tested arrays.
{
   Z mL=1;
/*
   This line, together with the following
   Word loc(...);
   CPM_MA
   CPM_MZ
   consitute a convenient idiom for signaling
   entry to and exit from a function block to the log file
   cpmcerr.txt. This writing to the log file takes place only if
   mL is larger or equal to the static data member
   CpmSystem::Message::verbose. This bulky quantity has a
   macro alias 'cpmverbose' (i.e. we have, in file cpmsystem.h
       #define cpmverbose      CpmSystem::Message::verbose
   ) and it can be set by a statement like
       cpmverbose=10;
   Its initial value is 2.
   Soon we will encounter macros cpmtime, cpmwait, cpmdebug.
  */
   Word loc("perMes(Z,Z)"); // messages use this as name of the function
   CPM_MA // defined in cpmmacros.h, assumes that 'mL' and 'loc' are
      // defined
   R t1=cpmtime(); /* time of function call in seconds from some
       system-defined 'point-zero in time'.
       cpmtime is a short name for function CpmSystem::time defined
       in file cpmsystem.h as follows:
           #define cpmtime         CpmSystem::time
   */
   N i,j;
   N ni=fi;
   N nf=fi+n-1;
   N mi=fi;
```

```
N mf=fi+m-1;

Vec<Vec<T>> vi, vf;
Vec<T> v1(m);
Vec<T> v2(m);
for (i=mi;i<=mf;++i) v1[i]=cz<T>(i-mi);
for (i=mi;i<=mf;++i) v2[i]=v1[i]*v1[i];
      // The preferred form of this statement in C+- is
      // for (i=v1.b();i<=v1.e();++i) v1[i]=cz(i);
      // Written in this form, it is alo correct if v1 is of type V<C>,
      // the C+- array for which indexing starts with 1 instead of 0.
Vec<Vec<T>> w1(n); for(i=ni;i<=nf;++i) w1[i]=v1;
Vec<Vec<T>> w2(n); for(i=ni;i<=nf;++i) w2[i]=v2;

Vec<Vec<T>> temp1=w1; // copy constructor
Vec<Vec<T>> temp2=temp1; // copy constructor
Vec<Vec<T>> temp3=temp2; // copy constructor
Vec<Vec<T>> temp4=temp3; // copy constructor
Vec<Vec<T>> temp5=temp4; // copy constructor
Vec<Vec<T>> temp6=temp5; // copy constructor
Vec<Vec<T>> temp7=temp6; // copy constructor
Vec<Vec<T>> temp8=temp7; // copy constructor
Vec<Vec<T>> temp9=temp8; // copy constructor
Vec<Vec<T>> temp10=temp9; // copy constructor
temp1=Vec<Vec<T>>(); // assignment
temp2=w2; // assignment
temp3=temp2; // assignment
temp4=w2; // assignment
temp5=temp4; // assignment
temp6=w2; // assignment
temp7=temp3; // assignment
temp8=temp2; // assignment
temp9=temp1; // assignment
   // these 9 assignments should neither influence w, nor temp10
vi=w1; // assignment
vf=temp10; // assignment
R t2=cpmtime();
R t12=t2-t1; // time needed for copy and assignment
R err=0,sum_v=0;
for (i=ni;i<=nf;++i){
   Vec<T> vii=vi[i];
   Vec<T> vfi=vf[i];
   for (j=mi;j<=mf;++j){
      T viij=vii[j];
      T vfij=vfi[j];
      err+=abs(viij-vfij);
      sum_v+=abs(vfij);
   }
} // making sure that copying and assignment worked correctly
// unfortunately one has to stagger many copy/assign steps
```

```
   // make this part of the function the dominant one timewise.
   R t3=cpmtime();
   R t23=t3-t2;
   V<R> res{t12,t23,t12+t23,err,sum_v};
   /*
      Convenient  l i s t  c o n s t r u c t o r, requires C++11
   */
   //bool b=std::movable<Vec<T>>;
   //cout<<"std::movable<Vec<T>> = "<<b<<endl;

  CPM_MZ
  return res;
}


template<typename T, template<typename> class Vec, N fi>
V<R> perMesM(N m, N n)
//: performance measure
// Intent: We want to evaluate how efficient the copy and assign operations for
// array ('vector') classes work for large objects (long arrays and large
// objects that make up the components of the arrays. I suggests itself to
// choose the large objects again as large arrays. The  components of these
// 'second order arrays' are not chosen as a primary builtin type such as double
// or int. Instead we use as the complex numbers for which there are
// implementations in namespaces std and CpmRoot.
// The various objects mentioned above can all treated by a function which takes
// types as arguments i.e. is a function template in C++ parlance.
// For these template arguments T, Vec, N we will finally (i.e. in function tutorial1)
// use the following concrete realisations:
//    T: std::complex<R>, CpmRoot::C
//    Vec: std::vector, std::valarray, CpmArrays::V, CpmArrays::V_, CpmArrays::V__,
//    N: 0 for Vec=vector,valarray,V_,V__ and 1 for V
// The function argument m sets the length of the tested arrays, and n sets the
// length of the arrays which figure as the components of the tested arrays.
{
   Z mL=1;
/*
   This line, together with the following
   Word loc(...);
   CPM_MA
   CPM_MZ
   consitute a convenient idiom for signaling
   entry to and exit from a function block to the log file
   cpmcerr.txt. This writing to the log file takes place only if
   mL is larger or equal to the static data member
   CpmSystem::Message::verbose. This bulky quantity has a
   macro alias 'cpmverbose' (i.e. we have, in file cpmsystem.h
       #define cpmverbose      CpmSystem::Message::verbose
   ) and it can be set by a statement like
       cpmverbose=10;
```

```
 Its initial value is 2.
 Soon we will encounter macros cpmtime, cpmwait, cpmdebug.
*/
 Word loc("perMesM(Z,Z)"); // messages use this as name of the function
 CPM_MA // defined in cpmmacros.h, assumes that 'mL' and 'loc' are
    // defined
 R t1=cpmtime(); /* time of function call in seconds from some
     system-defined 'point-zero in time'.
     cpmtime is a short name for function CpmSystem::time defined
     in file cpmsystem.h as follows:
         #define cpmtime         CpmSystem::time
 */
 N i,j;
 N ni=fi;
 N nf=fi+n-1;
 N mi=fi;
 N mf=fi+m-1;

 Vec<Vec<T>> vi, vf;
 Vec<T> v1(m);
 Vec<T> v2(m);
 for (i=mi;i<=mf;++i) v1[i]=cz<T>(i-mi);
 for (i=mi;i<=mf;++i) v2[i]=v1[i]*v1[i];
       // The preferred form of this statement in C+- is
       // for (i=v1.b();i<=v1.e();++i) v1[i]=cz(i);
       // Written in this form, it is alo correct if v1 is of type V<C>,
       // the C+- array for which indexing starts with 1 instead of 0.
 Vec<Vec<T>> w1(n); for(i=ni;i<=nf;++i) w1[i]=v1;
 Vec<Vec<T>> w2(n); for(i=ni;i<=nf;++i) w2[i]=v2;

 Vec<Vec<T>> w1Orig=w1;

 Vec<Vec<T>> temp1=std::move(w1); // copy constructor
 Vec<Vec<T>> temp2=std::move(temp1); // copy constructor
 Vec<Vec<T>> temp3=std::move(temp2); // copy constructor
 Vec<Vec<T>> temp4=std::move(temp3); // copy constructor
 Vec<Vec<T>> temp5=std::move(temp4); // copy constructor
 Vec<Vec<T>> temp6=std::move(temp5); // copy constructor
 Vec<Vec<T>> temp7=std::move(temp6); // copy constructor
 Vec<Vec<T>> temp8=std::move(temp7); // copy constructor
 Vec<Vec<T>> temp9=std::move(temp8); // copy constructor
 Vec<Vec<T>> temp10=std::move(temp9); // copy constructor
 temp1=std::move(Vec<Vec<T>>()); // assignment
 temp2=std::move(w2); // assignment
 temp3=std::move(temp2); // assignment
 temp4=std::move(w2); // assignment
 temp5=std::move(temp4); // assignment
 temp6=std::move(w2); // assignment
 temp7=std::move(temp3); // assignment
 temp8=std::move(temp2); // assignment
```

```
   temp9=std::move(temp1); // assignment
      // these 9 assignments should neither influence w, nor temp10
   vi=std::move(w1Orig); // assignment
   vf=std::move(temp10); // assignment
   R t2=cpmtime();
   R t12=t2-t1; // time needed for copy and assignment
   R err=0,sum_v=0;
   for (i=ni;i<=nf;++i){
      Vec<T> vii=std::move(vi[i]);
      Vec<T> vfi=std::move(vf[i]);
      for (j=mi;j<=mf;++j){
         T viij=vii[j];
         T vfij=vfi[j];
         err+=abs(viij-vfij);
         sum_v+=abs(vfij);
      }
   } // making sure that copying and assignment worked correctly
   // unfortunately one has to stagger many copy/assign steps
   // make this part of the function the dominant one timewise.
   R t3=cpmtime();
   R t23=t3-t2;
   V<R> res{t12,t23,t12+t23,err,sum_v};
   /*
      Convenient  l i s t  c o n s t r u c t o r, requires C++11
   */
   //bool b=std::movable<Vec<T>>;
   //cout<<"std::movable<Vec<T>> = "<<b<<endl;

  CPM_MZ
  return res;
}

template<typename T, template<typename> class Vec, N fi>
V<R> perMesMove(N m, N n)
//: performance measure
// Intent: We want to evaluate how efficient the copy and assign operations for
// array ('vector') classes work for large objects (long arrays and large
// objects that make up the components of the arrays. I suggests itself to
// choose the large objects again as large arrays. The  components of these
// 'second order arrays' are not chosen as a primary builtin type such as double
// or int. Instead we use as the complex numbers for which there are
// implementations in namespaces std and CpmRoot.
// The various objects mentioned above can all treated by a function which takes
// types as arguments i.e. is a function template in C++ parlance.
// For these template arguments T, Vec, N we will finally (i.e. in function tutorial1)
// use the following concrete realisations:
//    T: std::complex<R>, CpmRoot::C
//    Vec: std::vector, std::valarray, CpmArrays::V, CpmArrays::V_, CpmArrays::V__,
//    N: 0 for Vec=vector,valarray,V_,V__ and 1 for V
// The function argument m sets the length of the tested arrays, and n sets the
```

```
// length of the arrays which figure as the components of the tested arrays.
{
   Z mL=1;
/*
   This line, together with the following
   Word loc(...);
   CPM_MA
   CPM_MZ
   consitute a convenient idiom for signaling
   entry to and exit from a function block to the log file
   cpmcerr.txt. This writing to the log file takes place only if
   mL is larger or equal to the static data member
   CpmSystem::Message::verbose. This bulky quantity has a
   macro alias 'cpmverbose' (i.e. we have, in file cpmsystem.h
       #define cpmverbose      CpmSystem::Message::verbose
   ) and it can be set by a statement like
       cpmverbose=10;
   Its initial value is 2.
   Soon we will encounter macros cpmtime, cpmwait, cpmdebug.
  */
   Word loc("perMesMove(Z,Z)"); // messages use this as name of the function
   CPM_MA // defined in cpmmacros.h, assumes that 'mL' and 'loc' are
      // defined
   R t1=cpmtime(); /* time of function call in seconds from some
       system-defined 'point-zero in time'.
       cpmtime is a short name for function CpmSystem::time defined
       in file cpmsystem.h as follows:
           #define cpmtime         CpmSystem::time
   */
   cout<<"perMesMove entered"<<endl;
   N i,j;
   N ni=fi;
   N nf=fi+n-1;
   N mi=fi;
   N mf=fi+m-1;

   Vec<T> vi, vf;
   Vec<T> v1(m);
   Vec<T> v2(m);
   for (i=mi;i<=mf;++i) v1[i]=cz<T>(i-mi);
   for (i=mi;i<=mf;++i) v2[i]=v1[i]*v1[i];
         // The preferred form of this statement in C+- is
         // for (i=v1.b();i<=v1.e();++i) v1[i]=cz(i);
         // Written in this form, it is alo correct if v1 is of type V<C>,
         // the C+- array for which indexing starts with 1 instead of 0.
   cout<<"first simple block done"<<endl;

   Vec<T> w1(n); for(i=ni;i<=nf;++i) w1[i]=std::move(v1);
   Vec<T> w2(n); for(i=ni;i<=nf;++i) w2[i]=std::move(v2);
```

```
cout<<"second block done"<<endl;

Vec<Vec<T>> temp1=std::move(w1); // copy constructor
Vec<Vec<T>> temp2=std::move(temp1); // copy constructor
Vec<Vec<T>> temp3=std::move(temp2); // copy constructor
Vec<Vec<T>> temp4=std::move(temp3); // copy constructor
Vec<Vec<T>> temp5=std::move(temp4); // copy constructor
Vec<Vec<T>> temp6=std::move(temp5); // copy constructor
Vec<Vec<T>> temp7=std::move(temp6); // copy constructor
Vec<Vec<T>> temp8=std::move(temp7); // copy constructor
Vec<Vec<T>> temp9=std::move(temp8); // copy constructor
Vec<Vec<T>> temp10=std::move(temp9); // copy constructor
cout<<"third block done"<<endl;
temp1=std::move(Vec<Vec<T>>()); // assignment
temp2=std::move(w2); // assignment
temp3=std::move(temp2); // assignment
temp4=std::move(w2); // assignment
temp5=std::move(temp4); // assignment
temp6=std::move(w2); // assignment
temp7=std::move(temp3); // assignment
temp8=std::move(temp2); // assignment
temp9=std::move(temp1); // assignment
cout<<"block 4 done"<<endl;
   // these 9 assignments should neither influence w, nor temp10
vi=std::move(w1); // assignment
vf=std::move(temp10); // assignment
cout<<"block 5 done"<<endl;
R t2=cpmtime();
R t12=t2-t1; // time needed for copy and assignment
R err=0.,sum_v=0.;
for (N i=ni;i<=nf;++i){
   cout<<"i-loop "<<i<<" started"<<endl;
   Vec<T> vii=std::move(vi[i]);
   Vec<T> vfi=std::move(vf[i]);

   for (N j=mi;j<=mf;++j){
      T viij=vii[j];
      T vfij=vfi[j];
      err+=abs(viij-vfij);
      sum_v+=abs(vfij);
   }
} // making sure that copying and assignment worked correctly
// unfortunately one has to stagger many copy/assign steps
// make this part of the function the dominant one timewise.
cout<<"block 6 done"<<endl;
R t3=cpmtime();
R t23=t3-t2;
V<R> res{t12,t23,t12+t23,err,sum_v};
/*
   Convenient  l i s t  c o n s t r u c t o r, requires C++11
```

```
   */
   //bool b=std::movable<Vec<T>>;
   //cout<<"std::movable<Vec<T>> = "<<b<<endl;

  CPM_MZ
  return res;
}

template<typename T, template<typename> class Vec, N fi>
V<R> perMesMove0(N m, N n)
{
   Z mL=1;

   Word loc("perMesMove0(Z,Z)"); // messages use this as name of the function
   CPM_MA
   R t1=cpmtime();
 //  cout<<"perMesMove1 entered"<<endl;
   N i,j;
   N ni=fi;
   N nf=fi+n-1;
   N mi=fi;
   N mf=fi+m-1;

   Vec<T> vi, vf;
   Vec<T> w1(m);
   Vec<T> w2(m);
   for (i=mi;i<=mf;++i) w1[i]=cz<T>(i-mi);
   for (i=mi;i<=mf;++i) w2[i]=w1[i]*w1[i];
         // The preferred form of this statement in C+- is
         // for (i=v1.b();i<=v1.e();++i) v1[i]=cz(i);
         // Written in this form, it is alo correct if v1 is of type V<C>,
         // the C+- array for which indexing starts with 1 instead of 0.
   Vec<T> wOrig(m);
   for (i=mi;i<=mf;++i) wOrig[i]=w1[i];

   Vec<T> temp1=w1; // copy constructor
 //  for (i=mi;i<=mf;++i) cout<<"temp1[i] = "<<temp1[i]<<endl;
   //cout<<"temp1 = "<<temp1<<endl;
   Vec<T> temp2=temp1; // copy constructor
   Vec<T> temp3=temp2; // copy constructor
   Vec<T> temp4=temp3; // copy constructor
   Vec<T> temp5=temp4; // copy constructor
   Vec<T> temp6=temp5; // copy constructor
   Vec<T> temp7=temp6; // copy constructor
//   for (i=mi;i<=mf;++i) cout<<"temp7[i] = "<<temp7[i]<<endl;
   Vec<T> temp8=temp7; // copy constructor
   Vec<T> temp9=temp8; // copy constructor
   Vec<T> temp10=temp9; // copy constructor
//   cout<<"third block done"<<endl;
   temp1=w2; // assignment
```

```
   temp2=w2; // assignment
   temp3=temp2; // assignment
   temp4=w2; // assignment
   temp5=temp4; // assignment
   temp6=w2; // assignment
   temp7=temp3; // assignment
   temp8=temp2; // assignment
   temp9=temp1; // assignment
 //  cout<<"block 4 done"<<endl;
      // these 9 assignments should neither influence w, nor temp10
   vi=w0rig; // assignment
   vf=temp10; // assignment
 //  for (i=mi;i<=mf;++i) cout<<"vi[i] = "<<vi[i]<<endl;
 //  for (i=mi;i<=mf;++i) cout<<"vf[i] = "<<vf[i]<<endl;
//   cout<<"block 5 done"<<endl;
   R t2=cpmtime();
   R t12=t2-t1; // time needed for copy and assignment
   R err=0.,sum_v=0.;
   for (N i=mi;i<=mf;++i){
     // cout<<"i-loop "<<i<<" started"<<endl;
     // cout<<"vi[i] = "<<vi[i]<<" vf[i] = "<<vf[i]<<endl;
      err+=abs(vi[i]-vf[i]);
      sum_v+=abs(vf[i]);
   }
//   cout<<"block 6 done"<<endl;
   R t3=cpmtime();
   R t23=t3-t2;
   V<R> res{t12,t23,t12+t23,err,sum_v};
   /*
      Convenient  l i s t  c o n s t r u c t o r, requires C++11
   */
   //bool b=std::movable<Vec<T>>;
   //cout<<"std::movable<Vec<T>> = "<<b<<endl;

  CPM_MZ
  return res;
}


template<typename T, template<typename> class Vec, N fi>
V<R> perMesMove1(N m, N n)
{
   Z mL=1;

   Word loc("perMesMove1(Z,Z)"); // messages use this as name of the function
   CPM_MA
   R t1=cpmtime();
 //  cout<<"perMesMove1 entered"<<endl;
   N i,j;
   N ni=fi;
```

```
   N nf=fi+n-1;
   N mi=fi;
   N mf=fi+m-1;

   Vec<T> vi, vf;
   Vec<T> w1(m);
   Vec<T> w2(m);
   for (i=mi;i<=mf;++i) w1[i]=cz<T>(i-mi);
   for (i=mi;i<=mf;++i) w2[i]=w1[i]*w1[i];
         // The preferred form of this statement in C+- is
         // for (i=v1.b();i<=v1.e();++i) v1[i]=cz(i);
         // Written in this form, it is alo correct if v1 is of type V<C>,
         // the C+- array for which indexing starts with 1 instead of 0.
   Vec<T> wOrig(m);
   for (i=mi;i<=mf;++i) wOrig[i]=w1[i];

   Vec<T> temp1=std::move(w1); // copy constructor
 //  for (i=mi;i<=mf;++i) cout<<"temp1[i] = "<<temp1[i]<<endl;
   //cout<<"temp1 = "<<temp1<<endl;
   Vec<T> temp2=std::move(temp1); // copy constructor
   Vec<T> temp3=std::move(temp2); // copy constructor
   Vec<T> temp4=std::move(temp3); // copy constructor
   Vec<T> temp5=std::move(temp4); // copy constructor
   Vec<T> temp6=std::move(temp5); // copy constructor
   Vec<T> temp7=std::move(temp6); // copy constructor
//   for (i=mi;i<=mf;++i) cout<<"temp7[i] = "<<temp7[i]<<endl;
   Vec<T> temp8=std::move(temp7); // copy constructor
   Vec<T> temp9=std::move(temp8); // copy constructor
   Vec<T> temp10=std::move(temp9); // copy constructor
//   cout<<"third block done"<<endl;
   temp1=std::move(w2); // assignment
   temp2=std::move(w2); // assignment
   temp3=std::move(temp2); // assignment
   temp4=std::move(w2); // assignment
   temp5=std::move(temp4); // assignment
   temp6=std::move(w2); // assignment
   temp7=std::move(temp3); // assignment
   temp8=std::move(temp2); // assignment
   temp9=std::move(temp1); // assignment
 //  cout<<"block 4 done"<<endl;
      // these 9 assignments should neither influence w, nor temp10
   vi=std::move(wOrig); // assignment
   vf=std::move(temp10); // assignment
 //  for (i=mi;i<=mf;++i) cout<<"vi[i] = "<<vi[i]<<endl;
 //  for (i=mi;i<=mf;++i) cout<<"vf[i] = "<<vf[i]<<endl;
//   cout<<"block 5 done"<<endl;
   R t2=cpmtime();
   R t12=t2-t1; // time needed for copy and assignment
   R err=0.,sum_v=0.;
   for (N i=mi;i<=mf;++i){
```

```
     // cout<<"i-loop "<<i<<" started"<<endl;
     // cout<<"vi[i] = "<<vi[i]<<" vf[i] = "<<vf[i]<<endl;
      err+=abs(vi[i]-vf[i]);
      sum_v+=abs(vf[i]);
   }
//   cout<<"block 6 done"<<endl;
   R t3=cpmtime();
   R t23=t3-t2;
   V<R> res{t12,t23,t12+t23,err,sum_v};
   /*
      Convenient  l i s t  c o n s t r u c t o r, requires C++11
   */
   //bool b=std::movable<Vec<T>>;
   //cout<<"std::movable<Vec<T>> = "<<b<<endl;

  CPM_MZ
  return res;
}

template<typename T, template<typename> class Vec, N fi>
V<R> perMesMove2(N m, N n)
{
   Z mL=1;

   Word loc("perMesMove2(Z,Z)"); // messages use this as name of the function
   CPM_MA
   R t1=cpmtime();
 //  cout<<"perMesMove1 entered"<<endl;
   N i,j;
   N ni=fi;
   N nf=fi+n-1;
   N mi=fi;
   N mf=fi+m-1;

   Vec<T> vi, vf;
   Vec<T> w1(m);
   Vec<T> w2(m);
 //  cout<<"got to loc 1"<<endl;
   for (i=mi;i<=mf;++i){
   //   cout<<" index in loop = "<<i<<endl;
      T yi=cz<T>(i-mi);
  //    cout<<" yi in loop = "<<yi<<endl;
      w1[i]=yi;
    //  cout<<" w1[i] in loop done "<<w1[i]<<endl;
      w2[i]=yi*yi;
   }
 //  cout<<"got to loc 2"<<endl;
         // The preferred form of this statement in C+- is
         // for (i=v1.b();i<=v1.e();++i) v1[i]=cz(i);
         // Written in this form, it is alo correct if v1 is of type V<C>,
```

```
        // the C+- array for which indexing starts with 1 instead of 0.
   Vec<T> wOrig=w1;
  // Vec<T> wOrig(m);
  // for (i=mi;i<=mf;++i) wOrig[i]=w1[i];
 // cout<<"got to loc 3"<<endl;

   Vec<T> temp1=w1; // copy constructor
 //  for (i=mi;i<=mf;++i) cout<<"temp1[i] = "<<temp1[i]<<endl;
   //cout<<"temp1 = "<<temp1<<endl;
   Vec<T> temp2=temp1; // copy constructor
   Vec<T> temp3=temp2; // copy constructor
   Vec<T> temp4=temp3; // copy constructor
   Vec<T> temp5=temp4; // copy constructor
   Vec<T> temp6=temp5; // copy constructor
   Vec<T> temp7=temp6; // copy constructor
//   for (i=mi;i<=mf;++i) cout<<"temp7[i] = "<<temp7[i]<<endl;
   Vec<T> temp8=temp7; // copy constructor
   Vec<T> temp9=temp8; // copy constructor
   Vec<T> temp10=temp9; // copy constructor
//   cout<<"third block done"<<endl;
   cout<<"got to loc 4"<<endl;
   temp1=w2; // assignment
   temp2=w2; // assignment
   temp3=temp2; // assignment
   temp4=w2; // assignment
   temp5=temp4; // assignment
   temp6=w2; // assignment
   temp7=temp3; // assignment
   temp8=temp2; // assignment
   temp9=temp1; // assignment
 //  cout<<"block 4 done"<<endl;
      // these 9 assignments should neither influence w, nor temp10
 //  cout<<"got to loc 5"<<endl;
   vi=wOrig; // assignment
   vf=temp10; // assignment
 //  for (i=mi;i<=mf;++i) cout<<"vi[i] = "<<vi[i]<<endl;
 //  for (i=mi;i<=mf;++i) cout<<"vf[i] = "<<vf[i]<<endl;
//   cout<<"block 5 done"<<endl;
   R t2=cpmtime();
   R t12=t2-t1; // time needed for copy and assignment
   R err=0.,sum_v=0.;
   for (N i=mi;i<=mf;++i){
     // cout<<"i-loop "<<i<<" started"<<endl;
     // cout<<"vi[i] = "<<vi[i]<<" vf[i] = "<<vf[i]<<endl;
      err+=abs(vi[i]-vf[i]);
      sum_v+=abs(vf[i]);
   }
//   cout<<"block 6 done"<<endl;
   R t3=cpmtime();
   R t23=t3-t2;
```

```
   V<R> res{t12,t23,t12+t23,err,sum_v};
   /*
      Convenient  l i s t  c o n s t r u c t o r, requires C++11
   */
   //bool b=std::movable<Vec<T>>;
   //cout<<"std::movable<Vec<T>> = "<<b<<endl;

  CPM_MZ
  return res;
}

template<typename T, template<typename> class Vec, N fi>
V<R> perMesMove3(N m, N n)
{
   Z mL=1;

   Word loc("perMesMove3(Z,Z)"); // messages use this as name of the function
   CPM_MA
   R t1=cpmtime();
 //  cout<<"perMesMove1 entered"<<endl;
   N i,j;
   N ni=fi;
   N nf=fi+n-1;
   N mi=fi;
   N mf=fi+m-1;
   cout<<"m = "<<m<<endl;

   Vec<T> vi, vf;
   Vec<T> w1(m);
   cout<<"we write the components of w1:  "<<endl;
   for (i=mi;i<=mf;++i) cout<<"w1["<<i<<"] = "<<w1[i]<<endl;
   Vec<T> w2(m);
   cout<<"got to loc 1"<<endl;
   for (i=mi;i<=mf;++i){
      cout<<" index in loop = "<<i<<endl;
      T yi=cz<T>(i-mi);
      cout<<" yi in loop = "<<yi<<endl;
      T zi=yi*yi;
      cout<<" zi in loop = "<<zi<<endl;
      w1.set_(i,yi);
      cout<<"w1.set_ done"<<endl;
      w2.set_(i,zi);
      cout<<"w2.set_ done"<<endl;
   }
   cout<<"got to loc 2"<<endl;
         // The preferred form of this statement in C+- is
         // for (i=v1.b();i<=v1.e();++i) v1[i]=cz(i);
         // Written in this form, it is alo correct if v1 is of type V<C>,
         // the C+- array for which indexing starts with 1 instead of 0.
   Vec<T> wOrig=w1;
```

```
  // Vec<T> wOrig(m);
  // for (i=mi;i<=mf;++i) wOrig[i]=w1[i];
  cout<<"got to loc 3"<<endl;

   Vec<T> temp1=w1; // copy constructor
 //  for (i=mi;i<=mf;++i) cout<<"temp1[i] = "<<temp1[i]<<endl;
   //cout<<"temp1 = "<<temp1<<endl;
   Vec<T> temp2=temp1; // copy constructor
   Vec<T> temp3=temp2; // copy constructor
   Vec<T> temp4=temp3; // copy constructor
   Vec<T> temp5=temp4; // copy constructor
   Vec<T> temp6=temp5; // copy constructor
   Vec<T> temp7=temp6; // copy constructor
//   for (i=mi;i<=mf;++i) cout<<"temp7[i] = "<<temp7[i]<<endl;
   Vec<T> temp8=temp7; // copy constructor
   Vec<T> temp9=temp8; // copy constructor
   Vec<T> temp10=temp9; // copy constructor
//   cout<<"third block done"<<endl;
   cout<<"got to loc 4"<<endl;
   temp1=w2; // assignment
   temp2=w2; // assignment
   temp3=temp2; // assignment
   temp4=w2; // assignment
   temp5=temp4; // assignment
   temp6=w2; // assignment
   temp7=temp3; // assignment
   temp8=temp2; // assignment
   temp9=temp1; // assignment
 //  cout<<"block 4 done"<<endl;
      // these 9 assignments should neither influence w, nor temp10
   cout<<"got to loc 5"<<endl;
   vi=wOrig; // assignment
   vf=temp10; // assignment
 //  for (i=mi;i<=mf;++i) cout<<"vi[i] = "<<vi[i]<<endl;
 //  for (i=mi;i<=mf;++i) cout<<"vf[i] = "<<vf[i]<<endl;
//   cout<<"block 5 done"<<endl;
   R t2=cpmtime();
   R t12=t2-t1; // time needed for copy and assignment
   R err=0.,sum_v=0.;
   for (N i=mi;i<=mf;++i){
     // cout<<"i-loop "<<i<<" started"<<endl;
     // cout<<"vi[i] = "<<vi[i]<<" vf[i] = "<<vf[i]<<endl;
      err+=abs(vi[i]-vf[i]);
      sum_v+=abs(vf[i]);
   }
//   cout<<"block 6 done"<<endl;
   R t3=cpmtime();
   R t23=t3-t2;
   V<R> res{t12,t23,t12+t23,err,sum_v};
   /*
```

```
      Convenient  l i s t  c o n s t r u c t o r, requires C++11
   */
   //bool b=std::movable<Vec<T>>;
   //cout<<"std::movable<Vec<T>> = "<<b<<endl;

  CPM_MZ
  return res;
}



N tutorial1(N m, N n, N diff)
{
   Z mL=1;
   Word loc("tutorial1(Z,Z)");
   cout<<"tutorial1 entered"<<endl;

   CPM_MA
   V<V<R>> resL;
   if (diff ==0){
      resL<<perMesMove0<std::complex<R>,std::vector,0>(m,n);
      resL<<perMesMove0<std::complex<R>,std::valarray,0>(m,n);
      resL<<perMesMove0<CpmRoot::C,CpmArrays::V,1>(m,n);
      resL<<perMesMove0<CpmRoot::C,CpmArrays::V_,0>(m,n);
      resL<<perMesMove0<CpmRoot::C,CpmArrays::V__,0>(m,n);
      resL<<perMesMove0<CpmRoot::C,CpmArrays::Vv,0>(m,n);
   }
   else{
      resL<<perMes<std::complex<R>,std::vector,0>(m,n);
      resL<<perMes<std::complex<R>,std::valarray,0>(m,n);
      resL<<perMes<CpmRoot::C,CpmArrays::V,1>(m,n);
      resL<<perMes<CpmRoot::C,CpmArrays::V_,0>(m,n);
      resL<<perMes<CpmRoot::C,CpmArrays::V__,0>(m,n);
      resL<<perMes<CpmRoot::C,CpmArrays::Vv,0>(m,n);
   }
   V<Word> res1{"vector","valarray","V","V_","V__","Vv"};
   N nw=res1.size();
   Vo<R> dat1(nw);
   Vo<R> dat2(nw);
   Vo<R> dat3(nw);

   N i;
   for (i=1;i<=nw;++i) dat1[i]=resL[i][1];
   for (i=1;i<=nw;++i) dat2[i]=resL[i][2];
   for (i=1;i<=nw;++i) dat3[i]=resL[i][3];

   V<Z> per1=dat1.permutationForIncreasingOrder();
   V<Z> per2=dat2.permutationForIncreasingOrder();
   V<Z> per3=dat3.permutationForIncreasingOrder();
   R err=0.;
```

```
   for (i=1;i<=nw;++i) err+=resL[i][4]; // err>=0. is obvious from the definition

   V<Word> res1o=res1.permute(per1);
   V<R> dat1o=dat1.permute(per1);

   V<Word> res2o=res1.permute(per2);
   V<R> dat2o=dat2.permute(per2);

   V<Word> res3o=res1.permute(per3);
   V<R> dat3o=dat3.permute(per3);

   cout<<endl<<"Output of function tutorial1("<<m<<", "<<n<<")"<<endl;
   cout<<"total error ="<<err<<endl;
   cout<<"The lists are ordered for increasing times."<<endl;
   cout<<endl<<"execution times for copy and assignment:"<<endl;
   for (i=1;i<=nw;++i){
      cout<<res1o[i].std()<<": "<<dat1o[i]<<endl;
   }
   cout<<endl<<"execution times for verification:"<<endl;
   for (i=1;i<=nw;++i){
      cout<<res2o[i].std()<<": "<<dat2o[i]<<endl;
   }
   cout<<endl<<"total execution times:"<<endl;
   for (i=1;i<=nw;++i){
      cout<<res3o[i].std()<<": "<<dat2o[i]<<endl;
   }
   CPM_MZ
   return 0;
}


void info(){
   cout<<endl<<"takes zero, one, or two integer argument";
   cout<<endl<<"compares the performance or std::vector and CpmArays::V";
   cout<<endl<<"in copy construction and assignment operations"<<endl;
}
// end of tut1.cpp
```

65 survey.txt

```
2023-10-20   output by Ruby class UM2::CpmSurvey

         C+- higher-level names:
 Class/struct content of namespaces, together with
 the header files in which these are declared

Directories under consideration are:
 /home/ulrich/Desktop/e/cpm
        /source_publishing

 Number of files considered: 61
 Number of program lines considered: 34764
 Number of classes and structs found: 206

List of namespaces found:
   CpmArrays  introduced in file  cpmvm.h
   CpmFunctions  introduced in file  cpmfo.h
   CpmGeo  introduced in file  cpmangle.h
   CpmGraphics  introduced in file  cpmviewport.h
   CpmMPI  introduced in file  cpmmpi.h
   CpmRoot  introduced in file  cpmsystem.h
   CpmRootX  introduced in file  cpmtypes.h
   CpmStd  introduced in file  cpmnumbers.h
   CpmSystem  introduced in file  cpmsystem.h
   CpmTests  introduced in file  cpmtests.h
   CpmTime  introduced in file  cpmgreg.h
 (not always the file is given that in the
 dependency tree is next to the root)

Classes and structs by namespace:

 Namespace CpmArrays:
   IvZ   class in file   cpmzinterval.h
      'intervals of integers', i.e. contiguous finite subsets of Z

   M   class in file   cpmm.h
      map, associative array, dictionary, hash

   M   class in file   cpmm2.h
      map, associative array, dictionary, hash

   P   class in file   cpmuc.h
      'P' as in 'pointer', constant smart pointer.

   Po   class in file   cpmp.h
      Adding order operations to Pp<T>
```

```
Pp   class in file   cpmp.h
   polymorphic smart pointers

S   class in file   cpms.h
   sets of homotypic elements

Sr   class in file   cpmsr.h
   S with a rich interface

T2   class in file   cpmx.h
   homotypic pairs

T3   class in file   cpmx.h
   homotypic triplets

T4   class in file   cpmx.h
   homotypic quartets

UseCount   class in file   cpmuc.h
   support for reference counting and 'copy on write'

V   class in file   cpmv.h
   vector template, indexing starts with 1 by default.

V   class in file   cpmvcow.h
   vector template, indexing starts with 1 by default.

V<bool>   class in file   cpmv.h
VV   class in file   cpmv.h
   matrices

VV   class in file   cpmvcow.h
   matrices

VVV   class in file   cpmv.h
   tensors of rank 3

VVV   class in file   cpmvcow.h
   tensors of rank 3

VVVV   class in file   cpmv.h
   tensors of rank 4

VVVV   class in file   cpmvcow.h
   tensors of rank 4

VVVVa   class in file   cpmva.h
   version of VVVVo with arithmetic operations

VVVVo   class in file   cpmvo.h
```

```
   version of VVVV with order-related operations

VVVa   class in file   cpmva.h
   version of VVVo with arithmetic operations

VVVo   class in file   cpmvo.h
   version of VVV with order-related operations

VVa   class in file   cpmva.h
   version of VVo with arithmetic operations

VVo   class in file   cpmvo.h
   version of VV with order-related operations

Va   class in file   cpmva.h
   version of Vo with arithmetic operations

Vlin   class in file   cpmvlin.h
   version of V with arithmetic operations

Vo   class in file   cpmvo.h
   version of V with order-related operations

Vp   class in file   cpmp.h
   polymorphic V template

Vr   class in file   cpmvr.h
   version of Va with a rich interface

Vrs   class in file   cpmvr.h
   r: rich, s: scalar

Vsp   class in file   cpmvsp.h
   vector template, indexing starts with 0 .

Vuc   class in file   cpmvuc.h
   vector template, indexing starts with 1 by default.

Vuc<bool>   class in file   cpmvuc.h
VucVuc   class in file   cpmvuc.h
   matrices

VucVucVuc   class in file   cpmvuc.h
   tensors of rank 3

VucVucVucVuc   class in file   cpmvuc.h
   tensors of rank 4

X2   class in file   cpmx.h
   heterotypic pairs
```

```
  X3   class in file   cpmx.h
     heterotypic triplets

  X4   class in file   cpmx.h
     heterotypic 4-tuples

  X5   class in file   cpmx.h
     heterotypic 5-tuples

  X6   class in file   cpmx.h
     heterotypic 6-tuples

  X7   class in file   cpmx.h
     heterotypic 7-tuples

  X8   class in file   cpmx.h
     heterotypic 8-tuples


Namespace CpmFunctions:
  Bind1   class in file   cpmf.h
     binding the first parameter

  Bind2   class in file   cpmf.h
     binding the second parameter

  ConvertDomain   class in file   cpmf.h
     converting the function domain

  ConvertRange   class in file   cpmf.h
     converting the function range

  F   class in file   cpmfl.h
     functions as a class

  F1   class in file   cpmfl.h
     functions with one parameter

  F1_1   class in file   cpmfl.h
     as F1, but the first parameter is the function argument

  F2   class in file   cpmfl.h
     functions with two parameters

  F2_1   class in file   cpmfl.h
     as F2, but the first parameter is the function argument

  F2_2   class in file   cpmfl.h
     as F2, but the second parameter is the function argument
```

```
F3   class in file   cpmfl.h
   functions with three parameters

F3_1   class in file   cpmfl.h
   as F3, but the first parameter is the function argument

F3_2   class in file   cpmfl.h
   as F3, but the second parameter is the function argument

F3_3   class in file   cpmfl.h
   as F3, but the third parameter is the function argument

F4   class in file   cpmfl.h
   functions with four parameters

F4_1   class in file   cpmfl.h
   as F4, but the first parameter is the function argument

F4_2   class in file   cpmfl.h
   as F4, but the second parameter is the function argument

F4_3   class in file   cpmfl.h
   as F4, but the third parameter is the function argument

F4_4   class in file   cpmfl.h
   as F4, but the fourth parameter is the function argument

F5   class in file   cpmf.h
   functions with five parameters

F6   class in file   cpmf.h
   functions with six parameters

F_2   class in file   cpmf.h
   functions of two variables

Fa   class in file   cpmfa.h
   version of Fo with arithmetics operations

FncObj   class in file   cpmfl.h
   function objects

Fo   class in file   cpmfo.h
   version of F with order-related operations

Fr   class in file   cpmfr.h
   version of Fa with rich interface
```

```
Namespace CpmGeo:
  Angle   class in file   cpmangle.h
     angles


Namespace CpmGraphics:
  ColRef   class in file   cpmviewport.h
     lean 24-bit color-values

  Font   class in file   cpmviewport.h
     only Helvetica 12 needed from this font system of GLUT

  Rec   class in file   cpmviewport.h
     pixel rectangle which always fits Viewport::win()

  Viewport   class in file   cpmviewport.h
     Lean interface to the system's graphical capabilities.

  rgb   class in file   cpmviewport.h
     Z-valued red green blue

  xy   class in file   cpmviewport.h
     graphical points

  xyxy   class in file   cpmviewport.h
     graphical rectangles


Namespace CpmMPI:
  Com   class in file   cpmmpi.h
     class version of MPI's communicator concept for parallel computing

  Com   class in file   cpmmpi.h
     trivial implementation of Com


Namespace CpmRoot:
  Abs   class in file   cpmnumbers.h
     absolute value

  Abs<L>   class in file   cpmnumbers.h
     absolute value

  Abs<N>   class in file   cpmnumbers.h
     absolute value

  Abs<R>   class in file   cpmnumbers.h
     absolute value

  Abs<Z>   class in file   cpmnumbers.h
```

```
   absolute value

Abs<bool>   class in file   cpmnumbers.h
   absolute value

Abs<string>   class in file   cpmnumbers.h
   here string is interpreted as an array of L's

AbsSqr   class in file   cpmnumbers.h
   absolute (value) squared

AbsSqr<L>   class in file   cpmnumbers.h
   absolute (value) squared

AbsSqr<N>   class in file   cpmnumbers.h
   absolute (value) squared

AbsSqr<R>   class in file   cpmnumbers.h
   absolute (value) squared

AbsSqr<Z>   class in file   cpmnumbers.h
   absolute (value) squared

AbsSqr<bool>   class in file   cpmnumbers.h
   absolute (value) squared

AbsSqr<string>   class in file   cpmnumbers.h
   here string is interpreted as an array of L's

B   class in file   cpmsystem.h
   boolean values as a class, for which V<B> behaves regularly

C   class in file   cpmc.h
   complex numbers

Comp   class in file   cpmnumbers.h
   compare

Comp<L>   class in file   cpmnumbers.h
Comp<N>   class in file   cpmnumbers.h
Comp<R>   class in file   cpmnumbers.h
Comp<Z>   class in file   cpmnumbers.h
Comp<bool>   class in file   cpmnumbers.h
Comp<string>   class in file   cpmnumbers.h
Conj   class in file   cpmnumbers.h
   conjugation

Conj<L>   class in file   cpmnumbers.h
Conj<N>   class in file   cpmnumbers.h
Conj<R>   class in file   cpmnumbers.h
```

```
Conj<Z>   class in file   cpmnumbers.h
Conj<bool>   class in file   cpmnumbers.h
Conj<string>   class in file   cpmnumbers.h
Conv   class in file   cpmnumbers.h
   conversion

Dis   class in file   cpmnumbers.h
Dis<L>   class in file   cpmnumbers.h
Dis<N>   class in file   cpmnumbers.h
Dis<R>   class in file   cpmnumbers.h
Dis<Z>   class in file   cpmnumbers.h
Dis<bool>   class in file   cpmnumbers.h
Dis<string>   class in file   cpmnumbers.h
Hash   class in file   cpmnumbers.h
   hash value

Hash<L>   class in file   cpmnumbers.h
Hash<N>   class in file   cpmnumbers.h
Hash<R>   class in file   cpmnumbers.h
Hash<Z>   class in file   cpmnumbers.h
Hash<bool>   class in file   cpmnumbers.h
Hash<string>   class in file   cpmnumbers.h
IO   class in file   cpmnumbers.h
   input output class template

IO<L>   class in file   cpmnumbers.h
   specialization of IO

IO<N>   class in file   cpmnumbers.h
   specialization of IO

IO<R>   class in file   cpmnumbers.h
   specialization of IO

IO<Z>   class in file   cpmnumbers.h
   specialization of IO

IO<bool>   class in file   cpmnumbers.h
   specialization of IO

IO<string>   class in file   cpmnumbers.h
   specialization of IO

Inv   class in file   cpmnumbers.h
   inverse

Inv<L>   class in file   cpmnumbers.h
   inverse, defined in an arbitrary manner

Inv<N>   class in file   cpmnumbers.h
```

```
   inverse

Inv<R>   class in file   cpmnumbers.h
   inverse

Inv<Z>   class in file   cpmnumbers.h
   inverse

Inv<bool>   class in file   cpmnumbers.h
   inverse

Inv<string>   class in file   cpmnumbers.h
   reverse string

Name   class in file   cpmword.h
   tool class template for defining the nameOf function

Name<CpmRootX::fpVoidToVoid>   class in file   cpmtypes.h
   also this type needs a name

Name<CpmRootX::fpWordToVoid>   class in file   cpmtypes.h
   also this type needs a name

Name<L>   class in file   cpmword.h
   L: letter

Name<N>   class in file   cpmword.h
   N: natural numbers

Name<R>   class in file   cpmword.h
   multiple precision real numbers

Name<Z>   class in file   cpmword.h
   Z: integers

Name<bool>   class in file   cpmword.h
   specialization

Name<char>   class in file   cpmword.h
   specialization

Name<string>   class in file   cpmword.h
   specialization

Neutrals   class in file   cpmnumbers.h
   neutral elements

Neutrals<L>   class in file   cpmnumbers.h
Neutrals<N>   class in file   cpmnumbers.h
Neutrals<R>   class in file   cpmnumbers.h
```

```
  Neutrals<Z>   class in file   cpmnumbers.h
  Neutrals<bool>   class in file   cpmnumbers.h
  Neutrals<string>   class in file   cpmnumbers.h
  Ran   class in file   cpmnumbers.h
     random value

  Ran<L>   class in file   cpmnumbers.h
  Ran<N>   class in file   cpmnumbers.h
  Ran<R>   class in file   cpmnumbers.h
  Ran<Z>   class in file   cpmnumbers.h
  Ran<bool>   class in file   cpmnumbers.h
  Ran<string>   class in file   cpmnumbers.h
  Root   class in file   cpmword.h
     Provides the basic functions for dealing with CpmRoot's

  Test   class in file   cpmnumbers.h
     test value

  Test<L>   class in file   cpmnumbers.h
  Test<N>   class in file   cpmnumbers.h
  Test<R>   class in file   cpmnumbers.h
  Test<Z>   class in file   cpmnumbers.h
  Test<bool>   class in file   cpmnumbers.h
  Test<string>   class in file   cpmnumbers.h
  ToWord   class in file   cpmword.h
  ToWord<L>   class in file   cpmword.h
  ToWord<N>   class in file   cpmword.h
  ToWord<R>   class in file   cpmword.h
  ToWord<Z>   class in file   cpmword.h
  ToWord<bool>   class in file   cpmword.h
  ToWord<string>   class in file   cpmword.h
  Word   class in file   cpmword.h
     wrapping charcter strings, rich functionaliy


Namespace CpmRootX:
  R1   class in file   cpmtypes.h
     real numbers as a class

  Z1   class in file   cpmtypes.h
     integer numbers as a class


Namespace CpmSystem:
  Error   class in file   cpmsystem.h
     used if C+- classes throw errors

  Exception   class in file   cpmsystem.h
     simple debugging tool
```

```
  IFileStream   class in file   cpmsystem.h
     input file stream

  Message   class in file   cpmsystem.h
     provides basic output (unidirectional communication)

  OFileStream   class in file   cpmsystem.h
     output file stream

  Timer   class in file   cpmsystem.h
     implements e.g. getSecondsLeft()


Namespace CpmTests:
  Assignment   class in file   cpmtests.h
     testing consistency of assignment

  CopyConstructor   class in file   cpmtests.h
     testing consistency of copy constructor

  DefaultConstructor   class in file   cpmtests.h
     testing consistency of default constructor

  PolymorphicMulti   class in file   cpmtests.h
     testing polymorphic containers

  PolymorphicSingle   class in file   cpmtests.h
     testing polymorphic containers

  StrictAssignment   class in file   cpmtests.h
     testing strict consistency of assignment

  StrictAssignment2   class in file   cpmtests.h
     testing strict consistency of assignment

  Sym   class in file   cpmtests.h
     testing symmetry of equality

  TestBase   class in file   cpmtests.h
     base class for classes which test that class T

  TestOfPolymorphism   class in file   cpmtests.h
     testing consistent polymorphic behavior

  Test_F   class in file   cpmtests.h
     Testing the chaining operation of F

  Test_Vp   class in file   cpmtests.h
     testing behavior of Vp<>
```

```
  Test_c   class in file   cpmtests.h
     testing complex classes

  Test_logic   class in file   cpmtests.h
     test models of propositional logic

  Test_r   class in file   cpmtests.h
     testing the r-interface

  Test_rm   class in file   cpmtests.h
     Testing the modified r-interface.

  Test_set   class in file   cpmtests.h
     testing behavior of sets

  Test_sv   class in file   cpmtests.h
     testing the strict value interface

  Test_v   class in file   cpmtests.h
     testing the value interface

  Trans   class in file   cpmtests.h
     testing transitivity of equality

  ValueBehavior   class in file   cpmtests.h
     testing the property of a class T


Namespace CpmTime:
  Greg   class in file   cpmgreg.h
     time, date, and Gregorian Calendar

  TimeStyle   class in file   cpmgreg.h
     how to interprete time with respect to the globe
```

66 headerdependencies.txt

```
2023-10-20    output of Ruby class UM1::HeaderFileHierarchy
The directories under consideration are:

        /home/ulrich/Desktop/e/cpm/source_publishing
The 49 header files under consideration are in alphabetic order:
    cpmangle.h
    cpmbas.h
    cpmbasicinterfaces.h
    cpmbasictypes.h
    cpmc.h
    cpmcompdef.h
    cpmconfigwrc.ini
    cpmdefinitions.h
    cpmf.h
    cpmfa.h
    cpmfl.h
    cpmfo.h
    cpmfr.h
    cpmgreg.h
    cpminterfaces.h
    cpmm.h
    cpmm2.h
    cpmmacros.h
    cpmmpi.h
    cpmnumbers.h
    cpmp.h
    cpms.h
    cpmsr.h
    cpmsystem.h
    cpmsystemdependencies.h
    cpmtests.h
    cpmtypes.h
    cpmuc.h
    cpmv.h
    cpmvCOW.h
    cpmvMem1.h
    cpmvOld1.h
    cpmvOld2.h
    cpmvOld3.h
    cpmva.h
    cpmvcow.h
    cpmviewport.h
    cpmvlin.h
    cpmvm.h
    cpmvo.h
    cpmvold1.h
    cpmvold2.h
```

```
    cpmvold3.h
    cpmvr.h
    cpmvsp.h
    cpmvuc.h
    cpmword.h
    cpmx.h
    cpmzinterval.h

 Which header files are included in a header file ?


cpmangle.h includes:
  cpmc.h  cpmtypes.h  cpmv.h
cpmbas.h includes:
  cpmangle.h  cpmc.h  cpmfr.h  cpmgreg.h
  cpmm.h  cpmp.h  cpmsr.h  cpmtypes.h
  cpmvr.h
cpmbasicinterfaces.h includes:
  cpmbasictypes.h
cpmbasictypes.h includes:
  cpmdefinitions.h
cpmc.h includes:
  cpmword.h
cpmcompdef.h includes:
  none of the files under consideration
cpmconfigwrc.ini includes:
  none of the files under consideration
cpmdefinitions.h includes:
  cpmcompdef.h
cpmf.h includes:
  cpmv.h
cpmfa.h includes:
  cpmfo.h  cpmtypes.h
cpmfl.h includes:
  cpmuc.h  cpmword.h
cpmfo.h includes:
  cpmf.h
cpmfr.h includes:
  cpmfa.h  cpmvo.h
cpmgreg.h includes:
  cpmangle.h
cpminterfaces.h includes:
  cpmbasicinterfaces.h  cpmmpi.h  cpmnumbers.h
cpmm.h includes:
  cpmsr.h  cpmv.h
cpmm2.h includes:
  cpms.h
cpmmacros.h includes:
  none of the files under consideration
cpmmpi.h includes:
```

```
  cpmdefinitions.h
cpmnumbers.h includes:
  cpmbasicinterfaces.h
cpmp.h includes:
  cpmv.h
cpms.h includes:
  cpmtypes.h  cpmv.h
cpmsr.h includes:
  cpms.h  cpmvr.h
cpmsystem.h includes:
  cpmword.h
cpmsystemdependencies.h includes:
  none of the files under consideration
cpmtests.h includes:
  cpmc.h  cpmfr.h  cpmm.h  cpmp.h
  cpmsr.h  cpmtypes.h  cpmvr.h
cpmtypes.h includes:
  cpmmacros.h  cpmsystem.h  cpmx.h
cpmuc.h includes:
  cpmbasicinterfaces.h
cpmv.h includes:
  cpmfl.h  cpmmacros.h  cpmtypes.h  cpmzinterval.h
cpmvCOW.h includes:
  cpmfl.h  cpmmacros.h  cpmzinterval.h
cpmvMem1.h includes:
  cpmfl.h  cpmmacros.h  cpmtypes.h  cpmzinterval.h
cpmvOld1.h includes:
  cpmfl.h  cpmmacros.h  cpmtypes.h  cpmzinterval.h
cpmvOld2.h includes:
  cpmfl.h  cpmmacros.h  cpmtypes.h  cpmzinterval.h
cpmvOld3.h includes:
  cpmfl.h  cpmmacros.h  cpmtypes.h  cpmzinterval.h
cpmva.h includes:
  cpmtypes.h  cpmvo.h
cpmvcow.h includes:
  cpmfl.h  cpmmacros.h  cpmzinterval.h
cpmviewport.h includes:
  cpmv.h
cpmvlin.h includes:
  cpmv.h
cpmvm.h includes:
  none of the files under consideration
cpmvo.h includes:
  cpmv.h
cpmvold1.h includes:
  cpmfl.h  cpmmacros.h  cpmtypes.h  cpmzinterval.h
cpmvold2.h includes:
  cpmfl.h  cpmmacros.h  cpmtypes.h  cpmzinterval.h
cpmvold3.h includes:
  cpmfl.h  cpmmacros.h  cpmtypes.h  cpmzinterval.h
```

```
cpmvr.h includes:
  cpmva.h
cpmvsp.h includes:
  none of the files under consideration
cpmvuc.h includes:
  cpmfl.h  cpmmacros.h  cpmtypes.h  cpmzinterval.h
cpmword.h includes:
  cpminterfaces.h
cpmx.h includes:
  cpmword.h
cpmzinterval.h includes:
  cpmsystem.h  cpmx.h

 In which header files the header files are included ?


cpmangle.h is included in:
  cpmbas.h  cpmgreg.h
cpmbas.h is included in:
  no header file
cpmbasicinterfaces.h is included in:
  cpmuc.h  cpminterfaces.h  cpmnumbers.h
cpmbasictypes.h is included in:
  cpmbasicinterfaces.h
cpmc.h is included in:
  cpmbas.h  cpmangle.h  cpmtests.h
cpmcompdef.h is included in:
  cpmdefinitions.h
cpmconfigwrc.ini is included in:
  no header file
cpmdefinitions.h is included in:
  cpmmpi.h  cpmbasictypes.h
cpmf.h is included in:
  cpmfo.h
cpmfa.h is included in:
  cpmfr.h
cpmfl.h is included in:
  cpmvold3.h  cpmvold1.h  cpmvOld2.h  cpmvOld1.h
  cpmvcow.h  cpmvuc.h  cpmvold2.h  cpmvMem1.h
  cpmvOld3.h  cpmvCOW.h  cpmv.h
cpmfo.h is included in:
  cpmfa.h
cpmfr.h is included in:
  cpmbas.h  cpmtests.h
cpmgreg.h is included in:
  cpmbas.h
cpminterfaces.h is included in:
  cpmword.h
cpmm.h is included in:
  cpmbas.h  cpmtests.h
```

```
cpmm2.h is included in:
  no header file
cpmmacros.h is included in:
  cpmvold3.h  cpmvold1.h  cpmvOld2.h  cpmvOld1.h
  cpmtypes.h  cpmvcow.h  cpmvuc.h  cpmvold2.h
  cpmvMem1.h  cpmvOld3.h  cpmvCOW.h  cpmv.h
cpmmpi.h is included in:
  cpminterfaces.h
cpmnumbers.h is included in:
  cpminterfaces.h
cpmp.h is included in:
  cpmbas.h  cpmtests.h
cpms.h is included in:
  cpmsr.h  cpmm2.h
cpmsr.h is included in:
  cpmm.h  cpmbas.h  cpmtests.h
cpmsystem.h is included in:
  cpmzinterval.h  cpmtypes.h
cpmsystemdependencies.h is included in:
  no header file
cpmtests.h is included in:
  no header file
cpmtypes.h is included in:
  cpmvold3.h  cpmbas.h  cpmangle.h  cpmvold1.h
  cpmtests.h  cpmvOld2.h  cpmvOld1.h  cpmvuc.h
  cpmvold2.h  cpms.h  cpmvMem1.h  cpmfa.h
  cpmvOld3.h  cpmva.h  cpmv.h
cpmuc.h is included in:
  cpmfl.h
cpmv.h is included in:
  cpmp.h  cpmm.h  cpmangle.h  cpmvlin.h
  cpmviewport.h  cpms.h  cpmf.h  cpmvo.h
cpmvCOW.h is included in:
  no header file
cpmvMem1.h is included in:
  no header file
cpmvOld1.h is included in:
  no header file
cpmvOld2.h is included in:
  no header file
cpmvOld3.h is included in:
  no header file
cpmva.h is included in:
  cpmvr.h
cpmvcow.h is included in:
  no header file
cpmviewport.h is included in:
  no header file
cpmvlin.h is included in:
  no header file
```

```
cpmvm.h is included in:
  no header file
cpmvo.h is included in:
  cpmfr.h  cpmva.h
cpmvold1.h is included in:
  no header file
cpmvold2.h is included in:
  no header file
cpmvold3.h is included in:
  no header file
cpmvr.h is included in:
  cpmsr.h  cpmbas.h  cpmtests.h
cpmvsp.h is included in:
  no header file
cpmvuc.h is included in:
  no header file
cpmword.h is included in:
  cpmsystem.h  cpmfl.h  cpmx.h  cpmc.h
cpmx.h is included in:
  cpmzinterval.h  cpmtypes.h
cpmzinterval.h is included in:
  cpmvold3.h  cpmvold1.h  cpmvOld2.h  cpmvOld1.h
  cpmvcow.h  cpmvuc.h  cpmvold2.h  cpmvMem1.h
  cpmvOld3.h  cpmvCOW.h  cpmv.h
```