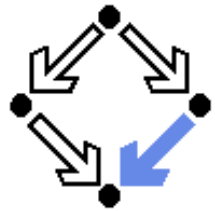
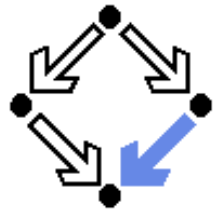
Numerics

Wolfgang Schreiner
Wolfgang.Schreiner@risc.uni-linz.ac.at

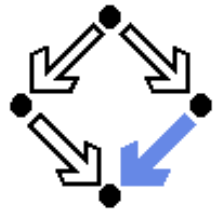
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
http://www.risc.uni-linz.ac.at





1. Integers and Floating Point Numbers
2. Complex Numbers
3. Rational Numbers
4. Numerical Vectors
5. Numerical Algorithms

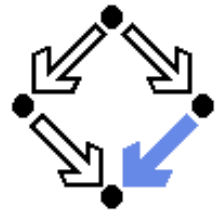
<cstdlib>: C General Utilties



```
C Standard General Utilities Library

This header defines several general purpose functions, including dynamic
memory management, random number generation, communication with the
environment, integer arthmetics, searching, sorting and converting.

Integer arithmethics:
abs Absolute value (function)
div Integral division (function)
labs Absolute value (function)
ldiv Integral division (function)

Types
div_t Structure returned by div (type)
ldiv_t Structure returned by div and ldiv (type)
size_t Unsigned integral type (type)

...
```

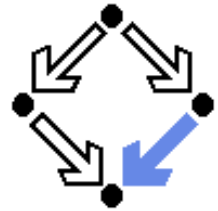
In C++, these functions are overloaded in various numerical types.

Example: abs



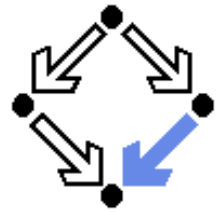
```
int abs ( int n );
long abs ( long n );

Absolute value

Returns the absolute value of parameter n.

In C++, this function is overloaded in <cmath> for floating-point types (see
cmath abs), in <complex> for complex numbers (see complex abs), and in
<valarray> for valarrays (see valarray abs).

Parameters

n
    Integral value.

Return Value
The absolute value of n.

Portability
In C, only the int version exists.
```

Example: `div`



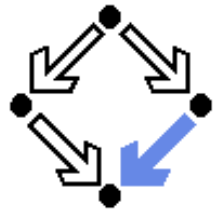
```
div_t div ( int numerator, int denominator );
ldiv_t div ( long numerator, long denominator );

Integral division

Returns the integral quotient and remainder of the division of numerator
by denominator as a structure of type div_t or ldiv_t, which has two members:
quot and rem.

Parameters

numerator
    Numerator.
denom
    Denominator.

Return Value
The result is returned by value in a structure defined in <cstdlib>,
which has two members. For div_t, these are, in either order:
  int quot; int rem;
and for ldiv_t:
  long quot; long rem;
```

<cmath>: C Numerics Library



```
C numerics library

cmath declares a set of functions to compute common mathematical operations
and transformations:

Trigonometric functions:
cos     Compute cosine (function)
sin     Compute sine (function)
tan     Compute tangent (function)
acos    Compute arc cosine (function)
asin    Compute arc sine (function)
atan    Compute arc tangent (function)
atan2   Compute arc tangent with two parameters (function)

Hyperbolic functions:
cosh    Compute hyperbolic cosine (function)
sinh    Compute hyperbolic sine (function)
tanh    Compute hyperbolic tangent (function)
```

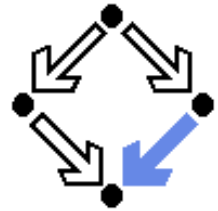
<cmath>: C Numerics Library



```
Exponential and logarithmic functions:
exp     Compute exponential function (function)
frexp   Get significand and exponent (function)
ldexp   Generate number from significand and exponent (function)
log     Compute natural logarithm (function)
log10   Compute common logarithm (function)
modf    Break into fractional and integral parts (function)

Power functions
pow     Raise to power (function)
sqrt    Compute square root (function)

Rounding, absolute value and remainder functions:
ceil    Round up value (function)
fabs    Compute absolute value (function)
floor   Round down value (function)
fmod    Compute remainder of division (function)
```

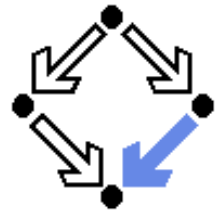
In C++, these functions are overloaded in various numerical types.

Example: cos



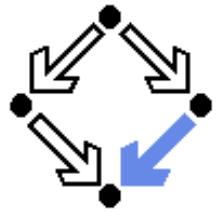
```
     double cos (      double x );
      float cos (       float x );
long double cos ( long double x );

Compute cosine

Returns the cosine of an angle of x radians.

In C++, this function is overloaded in <complex> and <valarray>
(see complex cos and valarray cos).

Parameters

x
    Floating point value representing an angle expressed in radians.

Return Value
Cosine of x.

Portability
In C, only the double version of this function exists with this name.
```

Example: Ranges of Numerical Types



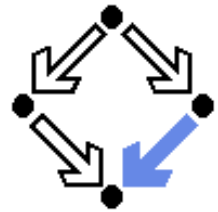
```
#include <iostream>
#include <limits>
using namespace std;

template<typename T>
void showMinMax( ) {
  cout << "min: " << numeric_limits<T>::min( ) << endl;
  cout << "max: " << numeric_limits<T>::max( ) << endl << endl;
}

int main( ) {
  cout << "short:" << endl; showMinMax<short>( );
  cout << "int:" << endl; showMinMax<int>( );
  cout << "long:" << endl; showMinMax<long>( );
  cout << "float:" << endl; showMinMax<float>( );
  cout << "double:" << endl; showMinMax<double>( );
  cout << "long double:" << endl; showMinMax<long double>( );
  cout << "unsigned short:" << endl; showMinMax<unsigned short>( );
  cout << "unsigned int:" << endl; showMinMax<unsigned int>( );
  cout << "unsigned long:" << endl; showMinMax<unsigned long>( );
}
```

D. Ryan Stepens et al: "C++ Cookbook"



1. Integers and Floating Point Numbers
2. **Complex Numbers**
3. Rational Numbers
4. Numerical Vectors
5. Numerical Algorithms

<complex>: Complex Numbers



```
template <typename T> class complex;
```

Complex number class

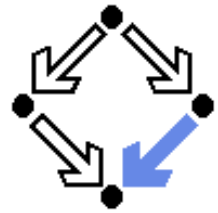
The complex class is designed to hold two components of the same type, that conform a complex number. ...

The class has been implemented to provide as similar a functionality as the one of a numerical type when this was possible, therefore, complex objects can be assigned, compared, inserted and extracted, as well as several arithmetical operators have been overloaded to be used on them directly.

complex specializations
complex is specialized for the three fundamental floating-point types: float, double and long double.

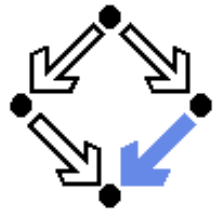
These specializations have the same members as the template, but optimize its implementation for these fundamental types, as well as they allow operations with other instantiations of complex (complex objects with different template argument).

<complex>: Complex Numbers



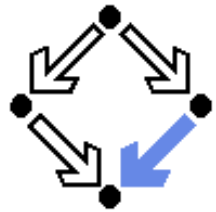
```
template <typename T> class complex {
public:
  typedef T value_type;

  complex (const T& re = T(), const T& im = T());
  complex (const complex& cmplx);
  template<class X> complex (const complex<X>& cmplx);

  T real() const; T imag() const;

  complex& operator= (const T& val); complex& operator= (const complex& rhs);

  complex& operator+= (const T& val); complex& operator-= (const T& val);
  complex& operator*= (const T& val); complex& operator/= (const T& val);

  template<class X> complex& operator= (const complex<X>& rhs);
  template<class X> complex& operator+= (const complex<X>& rhs);
  template<class X> complex& operator-= (const complex<X>& rhs);
  template<class X> complex& operator*= (const complex<X>& rhs);
  template<class X> complex& operator/= (const complex<X>& rhs);
};
```

<complex>: Complex Numbers



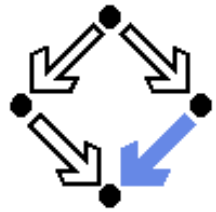
```
template<class T> complex<T> operator+(const complex<T>& x, const complex<T>& y);
template<class T> complex<T> operator+(const complex<T>& x, const T& val);
template<class T> complex<T> operator+(const T& val, const complex<T>& y);

template<class T> complex<T> operator-(const complex<T>& x, const complex<T>& y);
template<class T> complex<T> operator-(const complex<T>& x, const T& val);
template<class T> complex<T> operator-(const T& val, const complex<T>& y);

template<class T> complex<T> operator*(const complex<T>& x, const complex<T>& y);
template<class T> complex<T> operator*(const complex<T>& x, const T& val);
template<class T> complex<T> operator*(const T& val, const complex<T>& y);

template<class T> complex<T> operator/(const complex<T>& x, const complex<T>& y);
template<class T> complex<T> operator/(const complex<T>& x, const T& val);
template<class T> complex<T> operator/(const T& val, const complex<T>& y);

template<class T> complex<T> operator+(const complex<T>& y);
template<class T> complex<T> operator-(const complex<T>& y);
```

<complex>: Complex Numbers



```
template<class T> complex<T> operator==(const complex<T>& x, const complex<T>& y);
template<class T> complex<T> operator==(const complex<T>& x, const T& val);
template<class T> complex<T> operator==(const T& val, const complex<T>& y);

template<class T> complex<T> operator!=(const complex<T>& x, const complex<T>& y);
template<class T> complex<T> operator!=(const complex<T>& x, const T& val);
template<class T> complex<T> operator!=(const T& val, const complex<T>& y);

template<class T, class charT, class traits>
  basic_istream<charT,traits>&
    operator>> (basic_istream<charT,traits>& istr, const complex<T>& y);
template<class T, class charT, class traits>
  basic_ostream<charT,traits>&
    operator<< (basic_ostream<charT,traits>& ostr, const complex<T>& y);

template<class T> T real (const complex<T>& x);
template<class T> T imag (const complex<T>& x);
template<class T> complex<T> polar (const T& rho, const T& theta = 0);
template<class T> T arg (const complex<T>& x);
template<class T> T norm (const complex<T>& x);
```

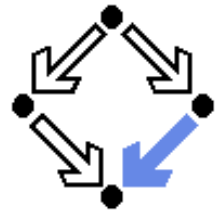
<complex>: Complex Numbers



```
template<class T> T abs (const complex<T>& x);
template<class T> complex<T> conj (const complex<T>& x);
template<class T> complex<T> cos (const complex<T>& x);
template<class T> complex<T> cosh (const complex<T>& x);
template<class T> complex<T> exp (const complex<T>& x);
template<class T> complex<T> log (const complex<T>& x);
template<class T> complex<T> log10 (const complex<T>& x);
template<class T> complex<T> pow (const complex<T>& x, int y);
template<class T> complex<T> pow (const complex<T>& x, const complex<T>& y);
template<class T> complex<T> pow (const complex<T>& x, const T& y);
template<class T> complex<T> pow (const T& x, const complex<T>& y);
template<class T> complex<T> sin (const complex<T>& x);
template<class T> complex<T> sinh (const complex<T>& x);
template<class T> complex<T> sqrt (const complex<T>& x);
template<class T> complex<T> tan (const complex<T>& x);
template<class T> complex<T> tanh (const complex<T>& x);
```

Very similar to use as the fundamental numeric types.

Example: Working with Polar Coordinates

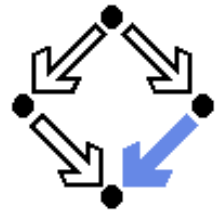


Let us manipulate polar coordinates.

```
#include <complex>
#include <iostream>

using namespace std;
...

int main( )  {
  double rho = 3.0; // magnitude
  double theta = 3.141592 / 2; // angle
  complex<double> coord = polar(rho, theta);
  cout << "rho = " << abs(coord) << ", theta = " << arg(coord) << endl;
  coord += polar(4.0, 0.0);
  cout << "rho = " << abs(coord) << ", theta = " << arg(coord) << endl;
}

rho = 3, theta = 1.5708
rho = 5, theta = 0.643501
```

D. Ryan Stepens et al: "C++ Cookbook"

Example: A Polar Coordinate Class

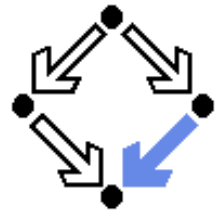


Let us have a polar representation of complex numbers.

```
#include <complex>
#include <iostream>

using namespace std;

...

int main( )  {
  double rho = 3.0; // magnitude
  double theta = 3.141592 / 2; // angle
  Polar coord(rho, theta);
  cout << "rho = " << coord.rho( ) << ", theta = " << coord.theta( ) << endl;
  coord += Polar(4.0, 0.0);
  cout << "rho = " << coord.rho( ) << ", theta = " << coord.theta( ) << endl;
  system("pause");
}
```

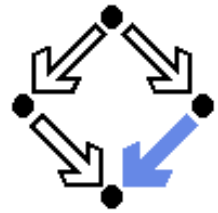
D. Ryan Stepens et al: "C++ Cookbook"

Example: A Polar Coordinate Class

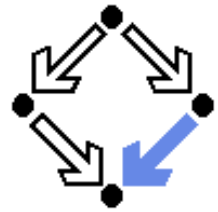


```
template<typename T> class BasicPolar
{
  complex<T> m;
  public:
    typedef BasicPolar self;

    // constructors
    BasicPolar( ) : m( ) {  }
    BasicPolar(const self& x) : m(x.m) {  }
    BasicPolar(const T& rho, const T& theta) : m(polar(rho, theta)) { }

    // public member functions
    T rho( ) const { return abs(m); }
    T theta( ) const { return arg(m); }

    // comparison operators
    friend bool operator==(const self& x, const self& y) { return x.m == y.m; }
    friend bool operator!=(const self& x, const self& y) { return x.m != y.m; }

    ...
};
```

Example: A Polar Coordinate Class

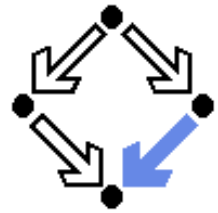


```
template<typename T> class BasicPolar
{
    ...

    // assignment operations
    self operator-( ) { return Polar(-m); }
    self& operator+=(const self& x) { m += x.m; return *this; }
    self& operator-=(const self& x) { m -= x.m; return *this; }
    self& operator*=(const self& x) { m *= x.m; return *this; }
    self& operator/=(const self& x) { m /= x.m; return *this; }
    operator complex<T>( ) const { return m; }

    // binary operations
    friend self operator+(self x, const self& y) { return x += y; }
    friend self operator-(self x, const self& y) { return x -= y; }
    friend self operator*(self x, const self& y) { return x *= y; }
    friend self operator/(self x, const self& y) { return x /= y; }

};

typedef BasicPolar<double> Polar;
```



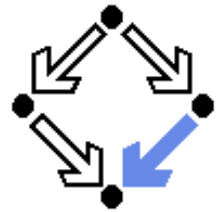
1. Integers and Floating Point Numbers
2. Complex Numbers
3. **Rational Numbers**
4. Numerical Vectors
5. Numerical Algorithms

Rational Numbers



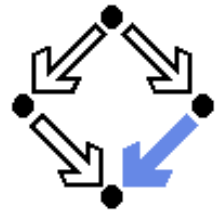
```
template<typename T> class rational {
  T num_; T den_; void reduce();
public:
  typedef T value_type;
  rational()                  : num_(0),   den_(1) {}
  rational(value_type num)    : num_(num), den_(1) {}
  rational(value_type num, value_type den): num_(num), den_(den) { reduce(); }
  rational(const rational& r): num_(r.num_), den_(r.den_) {}
  template<typename U>
  rational(const rational<U>& r): num_(r.num_), den_(r.den_) { reduce(); }

  rational& operator=(const rational& r)
    { num_ = r.num_; den_ = r.den_; return *this; }
  template<typename U> rational& operator=(const rational<U>& r)
    { assign(r.numerator(), r.denominator()); return *this; }

  void assign(value_type n, value_type d) { num_ = n; den_ = d; reduce(); }

  value_type numerator()   const { return num_; }
  value_type denominator() const { return den_; }
};
```

Rational Numbers



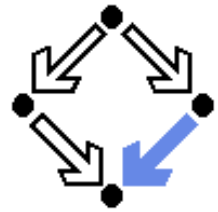
```
// Reduce the numerator and denominator by the gcd.
// Ensure the denominator is nonnegative.
template<typename T>
void rational<T>::reduce() {
  if (den_ < 0) { den_ = -den_; num_ = -num_; }
  T d = gcd(num_, den_);
  num_ /= d;
  den_ /= d;
}

// Greatest common divisor using Euclid's algorithm.
template<typename T>
T gcd(T n, T d) {
  n = abs(n);
  while (d != 0) {
    T t = n % d;
    n = d;
    d = t;
  }
  return n;
}
```

Rational Numbers



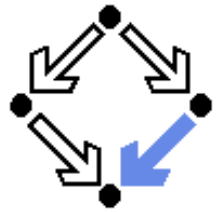
```
// Multiplication assignment operator.
template<typename T, typename U>
rational<T>& operator*=(rational<T>& dst, const rational<U>& src) {
  dst.assign(dst.numerator() * src.numerator(),
             dst.denominator() * src.denominator());
  return dst;
}

// Multiply two rational numbers.
template<typename T>
rational<T> operator*(const rational<T>& a, const rational<T>& b)
{ rational<T> result(a); result *= b; return result; }

// Multiply rational times an integral value.
template<typename T>
 rational<T> operator*(const T& a, const rational<T>& b)
{ return rational<T>(a * b.numerator(), b.denominator()); }

template<typename T>
rational<T> operator*(const rational<T>& a, const T& b)
{ return rational<T>(b * a.numerator(), a.denominator()); }
```

Rational Numbers



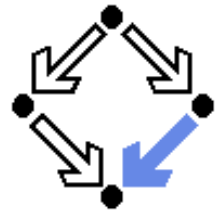
```
// Comparison. All other comparisons can be implemented
// in terms of operator== and operator<.
template<typename T>
bool operator==(const rational<T>& a, const rational<T>& b)
{
  // Precondition: both operands are reduced.
  return a.numerator() == b.numerator() &&
         a.denominator() == b.denominator();
}

template<typename T>
bool operator<(const rational<T>& a, const rational<T>& b)
{
  return a.numerator() * b.denominator() <
         b.numerator() * a.denominator();
}
```

Ray Lischner: “C++ in a Nutshell”



1. Integers and Floating Point Numbers
2. Complex Numbers
3. Rational Numbers
4. **Numerical Vectors**
5. Numerical Algorithms

<valarray>: Numerical Vectors



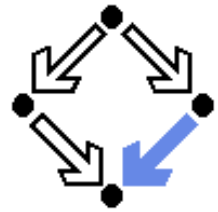
Vectors optimized for numerical operations.

```
#include <valarray>
#include <iostream>

using namespace std;

int main( ) {
  valarray<int> v(3);
  v[0] = 1; v[1] = 2; v[2] = 3;
  cout << v[0] << ", " << v[1] << ", " << v[2] << endl;
  v = v + v;
  cout << v[0] << ", " << v[1] << ", " << v[2] << endl;
  v /= 2;
  cout << v[0] << ", " << v[1] << ", " << v[2] << endl;
}
```

```
1, 2, 3
2, 4, 6
1, 2, 3
```

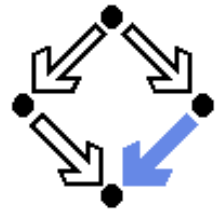
D. Ryan Stepens et al: "C++ Cookbook"

<valarray>: Constructors



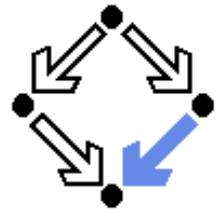
```
template <typename T> class valarray {
public:
  typedef T value_type;

  valarray();
  explicit valarray (size_t n);
  valarray (const T& val, size_t n);
  valarray (const T* p, size_t n);
  valarray (const valarray& x);
  valarray (const slice_array<T>& sub);
  valarray (const gslice_array<T>& sub);
  valarray (const mask_array<T>& sub);
  valarray (const indirect_array<T>& sub);

  ~valarray ();
  ...
}
```

<valarray>: Constructor Example



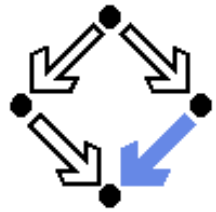
```
#include <iostream>
#include <valarray>
using namespace std;

int main ()
{
  int init[]= {10,20,30,40};
  valarray<int> first;            // (empty)
  valarray<int> second (5);       // 0 0 0 0 0
  valarray<int> third (10,3);     // 10 10 10
  valarray<int> fourth (init,4); // 10 20 30 40
  valarray<int> fifth (fourth);  // 10 20 30 40

  cout << "fifth sums " << fifth.sum() << endl;

  return 0;
}
```

<valarray>: Members



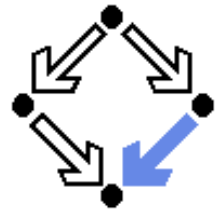
```
template <typename T> class valarray {
  ...
  size_t size() const;                    // number of elements in valarray
  void resize (size_t sz, T c = T()); // resize valarray

  T max() const;                          // maximum element in valarray
  T min() const;                          // minimum element in valarray
  T sum() const;                          // sum of all elements in valarray

  // n > 0: shift left, n < 0: shift right
  valarray<T> shift (int n) const;    // shift by n elements
  valarray<T> cshift (int n) const;   // cyclically shift by n elements

  valarray<T> apply (T func(T)) const; // apply func to every element
  valarray<T> apply (T func(const T&)) const;
  ...
}
```

<valarray>: Examples



```
int increment (int x) {return ++x;}
int init[]={10,20,30,40,50};

valarray<int> avalarray (init,5);          // 10 20 30 40 50
int s = avalarray.sum();                   // 150

valarray<int> foo (init,5);                // 10 20 30 40 50
valarray<int> bar = foo.apply(increment); // 11 21 31 41 51

valarray<int> myvalarray (init,5);    // 10 20 30 40 50
myvalarray = myvalarray.cshift(2);    // 30 40 50 10 20
myvalarray = myvalarray.cshift(-1);   // 20 30 40 50 10
```

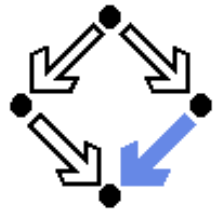
<valarray>: Access



```
template <typename T> class valarray {
  ...
              T  operator[] (size_t n) const;
              T& operator[] (size_t n);
     valarray<T> operator[] (slice sub) const;
   slice_array<T> operator[] (slice sub);
     valarray<T> operator[] (const gslice& sub) const;
  gslice_array<T> operator[] (const gslice& sub);
     valarray<T> operator[] (const valarray<bool>& mask) const;
    mask_array<T> operator[] (const valarray<bool>& mask);
     valarray<T> operator[] (const valarray<size_t>& indices) const;
indirect_array<T> operator[] (const valarray<size_t>& indices);
  ...
}
```

Access by indices, slices, generalized slices, masks, and index vectors.

<valarray>: **class** slice



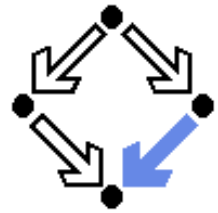
This class represents a valarray slice selector. It does not contain any element - it only describes a selection of elements in a valarray to be used as an index in valarray::operator[] .

A valarray slice is specified by a starting index, a size, and a stride.

The starting index (start) is the index of the first element in the selection.
The size (size) is the number of elements in the selection.
The stride (stride) is the separation between the elements in the valarray that are selected.

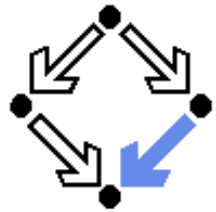
Therefore, a slice with a stride higher than 1 does not select contiguous elements in the valarray; For example, slice(3,4,5) selects the elements 3, 8, 13 and 18.

<valarray>: **class** slice



```
class slice {
public:
  slice ();
  slice (size_t start, size_t length, size_t stride);
  slice (const slice& slc);
  size_t start() const;
  size_t stride() const;
  size_t end() const;
}

valarray<int> foo (10);
for (int i=0; i<10; ++i) foo[i]=i;

slice slc (2,4,1);
valarray<int> bar = foo[slc];

cout << "slice starting at " << slc.start() << ":\n";
for (size_t n=0; n<bar.size(); n++) cout << bar[n] << ' '; cout << endl;

slice starting at 2:
2 3 4 5
```

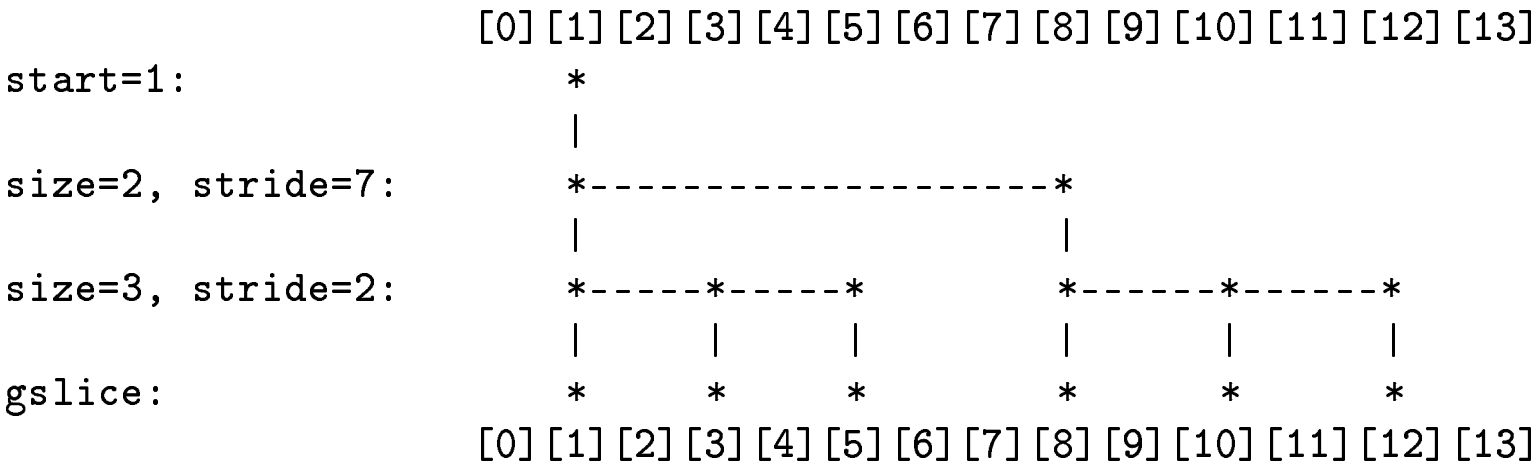
<valarray>: **class** gslice



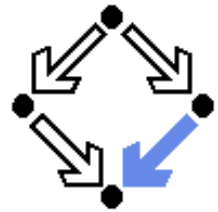
This class represents a valarray generalized slice (multidimensional slice) selector specified by a starting index, a set of sizes, and a set of strides. It produces a multidimensional combination of slice selections, where:

The starting index (start) is the index of the first element in the selection.
The size (size) is the number of elements selected in each dimension.
The stride (stride) is the separation between the elements that are selected (the size of each dimension).

For example, a gslice with start = 1, size = {2, 3}, stride = {7, 2} selects

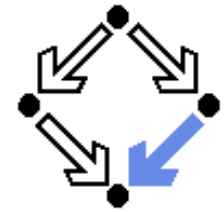


<valarray>: **class** gslice

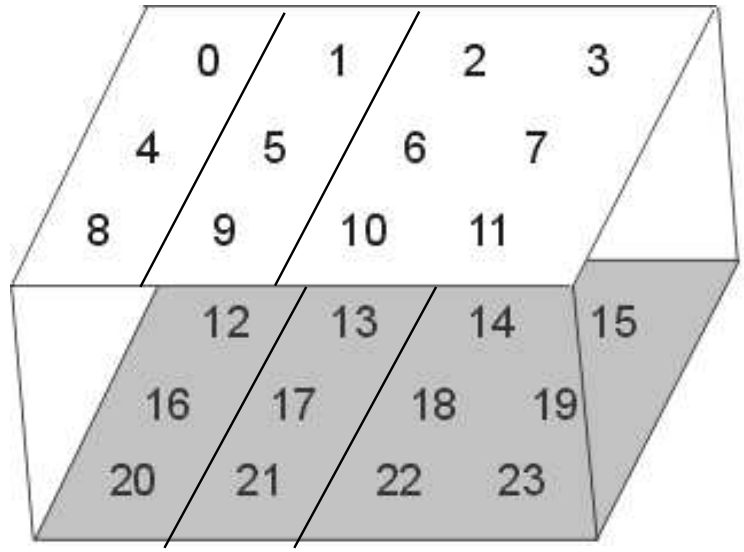


```
class gslice {
public:
  gslice ();
  gslice (size_t start, size_t length, size_t stride);
  slice (const slice& slc);

  size_t start() const;
  valarray<size_t> stride() const;
  valarray<size_t> end() const;
}

valarray<int> foo (14); for (int i=0; i<14; ++i) foo[i]=i;

size_t start=1; size_t lengths[]= {2,3}; size_t strides[]= {7,2};

gslice mygslice (start,valarray<size_t>(lengths,2),valarray<size_t>(strides,2));
valarray<int> bar = foo[mygslice];

for (size_t n=0; n<bar.size(); n++) cout << bar[n] << ' '; cout << endl;

1 3 5 8 10 12
```

<valarray>: Multi-Dimensional Arrays



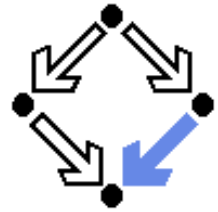
```
valarray<int> va(int a0, int a1) {
  valarray<int> result(2);
  result[0] = a0; result[1] = a1;
  return result;
}
```




```
valarray<int> a(24);
for (size_t i = 0; i < a.size(); ++i)
  a[i] = i;
cout << a[gslice(1, va(2, 3), va(12, 4))] << '\n';

1 5 9 13 17 21
```

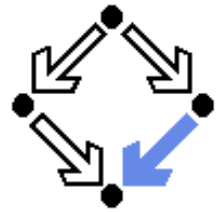
Representation of matrices by valarray and gslice.

<valarray>: Access Example



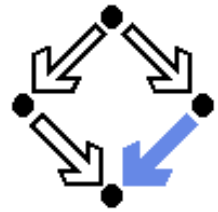
```
valarray<int> myarray (10);                 //  0  0  0  0  0  0  0  0  0  0

myarray[slice(2,3,3)]=10;                   //  0  0 10  0  0 10  0  0 10  0

size_t lengths[]={2,2};
size_t strides[]={6,2};                     //  0 20 10 20  0 10  0 20 10 20
myarray[gslice(1, valarray<size_t>(lengths,2),
  valarray<size_t>(strides,2))]=20;

valarray<bool> mymask (10);
for (int i=0; i<10; ++i) mymask[i]= ((i%2)==0);
myarray[mymask] += valarray<int>(3,5);    //  3 20 13 20  3 10  3 20 13 20

size_t sel[]= {2,5,7};
valarray<size_t> myselection (sel,3);     //  3 20 99 20  3 99  3 99 13 20
myarray[myselection]=99;

for (size_t i=0; i<myarray.size(); ++i) cout << myarray[i] << ' ';

3 20 99 20 3 99 3 99 13 20
```

<valarray>: Assignment



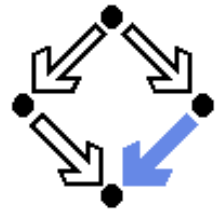
```
template <typename T> class valarray {
  ...
  valarray<T>& operator=(const valarray<T>& x);
  valarray<T>& operator=(const T& val);
  valarray<T>& operator=(const slice_array<T>& sub);
  valarray<T>& operator=(const gslice_array<T>& sub);
  valarray<T>& operator=(const mask_array<T>& sub);
  valarray<T>& operator=(const indirect_array<T>& sub);
  ...
}

valarray<int> foo (4);    //  0 0 0 0
valarray<int> bar (2,4);  //  0 0 0 0   2 2 2 2
foo = bar;                //  2 2 2 2   2 2 2 2
bar = 5;                  //  2 2 2 2   5 5 5 5
foo = bar[slice (1,2,1)]; //      5 5   5 5 5 5
```

<valarray>: Member Operations



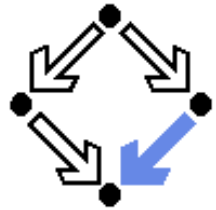
```
template <typename T> class valarray {
  ...
  valarray<T> operator+() const;
  valarray<T> operator-() const;
  valarray<T> operator~() const;
  valarray<bool> operator!() const;

  valarray<T>& operator*= (const valarray<T>& rhs);  // also: const T& rhs
  valarray<T>& operator/= (const valarray<T>& rhs);  // also: const T& rhs
  valarray<T>& operator%= (const valarray<T>& rhs);  // also: const T& rhs
  valarray<T>& operator+= (const valarray<T>& rhs);  // also: const T& rhs
  valarray<T>& operator-= (const valarray<T>& rhs);  // also: const T& rhs
  valarray<T>& operator^= (const valarray<T>& rhs);  // also: const T& rhs
  valarray<T>& operator&= (const valarray<T>& rhs);  // also: const T& rhs
  valarray<T>& operator|= (const valarray<T>& rhs);  // also: const T& rhs
  valarray<T>& operator<<= (const valarray<T>& rhs); // also: const T& rhs
  valarray<T>& operator>>= (const valarray<T>& rhs); // also: const T& rhs
  ...
}
```

<valarray>: Global Operations



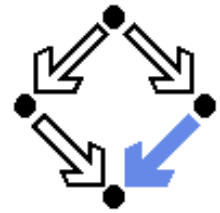
```
template <class T> valarray<T> operator+
  (const valarray<T>& lhs, const valarray<T>& rhs);
template <class T> valarray<T> operator+
  (const T& val, const valarray<T>& rhs);
template <class T> valarray<T> operator+
  (const valarray<T>& lhs, const T& val);

// analogous for other operations:

// operator-, operator*, operator/, operator%
// operator^, operator&, operator|
// operator<<, operator>>
// operator&&, operator||
// operator==, operator!=, operator<, operator<=, operator>, operator>=
```

<valarray>: Arithmetic Example



```
int init[]= {10,20,30,40};
                                  //      foo:              bar:
                                  //  --------------------------
valarray<int> foo (init, 4);      //  10 20 30 40
valarray<int> bar (25,4);         //  10 20 30 40     25 25 25 25

bar += foo;                       //  10 20 30 40     35 45 55 65

foo = bar + 10;                   //  45 55 65 75     35 45 55 65

foo -= 10;                        //  35 45 55 65     35 45 55 65

valarray<bool> comp = (foo==bar);

if (comp.min() == true)
  cout << "They are equal.\n";
else
  cout << "They are not equal.\n";

They are equal.
```

<valarray>: Helper Classes



```
template <class T> slice_array;
template <class T> gslice_array;
template <class T> mask_array;
template <class T> indirect_array;
```

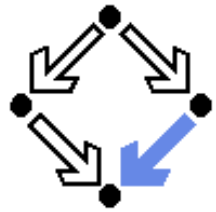
These classes are used as intermediate types returned by the non-constant versions of valarrays's subscript operator (valarray operator[]) when used with slices, generalized slices, masks, or index vectors.

An object of such a class contains references to the elements in the valarray object that were selected, and overloads the assignment and compound assignment operators, allowing direct access to the elements in the selection when used as l-value (left-value of an assignment operation).

When used as r-value (right-value of an assignment operation), it can initialize a valarray object, since valarray has a constructor taking a an object of this class as argument.

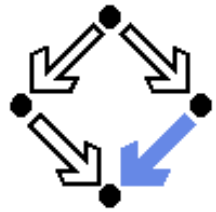
All the constructors of this class are private, preventing direct instantiations by a program - its only purpose is to be an efficient way to access the elements selected by a slice with valarray's operator[] .

<valarray>: Example



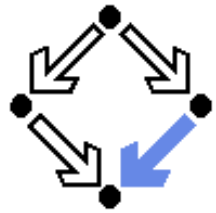
```
valarray<int> foo (9);
slice myslice;

for (int i=0; i<9; ++i) foo[i]=i;          //  0  1  2  3  4  5  6  7  8
                                           //     |     |     |
myslice=slice(1,3,2);                      //     v     v     v
foo[myslice] *= valarray<int>(10,3);       //  0 10  2 30  4 50  6  7  8
                                           //  |        |        |
myslice = slice (0,3,3);                   //  v        v        v
foo[myslice] = 99;                         // 99 10  2 99  4 50 99  7  8

cout << "foo:\n";
for (size_t n=0; n<foo.size(); n++) cout << foo[n] << ' '; cout << endl;

99 10 2 99 4 50 99 7 8
```

Typically there is need for the direct use of these classes.



1. Integers and Floating Point Numbers
2. Complex Numbers
3. Rational Numbers
4. Numerical Vectors
5. **Numerical Algorithms**

<numeric>: Numerical Algorithms



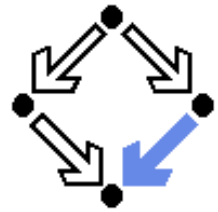
Generalized numeric operations

This header describes four algorithms especifically designed to operate
on numeric sequences that support certain operators.

Due to their flexibility, they can also be adapted for other kinds of sequences.

```
// Accumulates (by default: adds) all the values
// in the range [first,last) to init.
template <class I, class T>
T accumulate(I first, I last, T init);
template <class I, class T, class Op>
T accumulate(I first, I last, T init, Op op);

// Assigns to every element in the range starting at result the difference
// between its corresponding elements in the range [first,last) and the
// one preceding it (except for *result which is assigned *first).
template <class I, class O>
O adjacent_difference(I first, I last, O result);
template <class I, class O, class Op>
O adjacent_difference (I first, I last, O result, Op op);
```

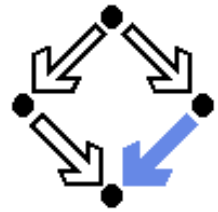
<numeric>: Numerical Algorithms



```
// Returns the result of accumulating init with the inner products of the
// pairs formed by the elements of two ranges starting at first1 and first2.
template <class I1, class I2, class T>
T inner_product(I1 first1, I1 last1, I2 first2, T init);
template <class I1, class I2, class T, class Op1, class Op2>
T inner_product( I1 first1, I1 last1, I2 first2, T init, Op1 op1, Op2 op2 );

// Assigns to every element in the range starting at result the partial sum
// of the corresponding elements in the range [first,last).
template <class I, class O>
O partial_sum(I first, I last, O result);
template <class I, class O, class Op>
O partial_sum(I first, I last, O result, Op op);
```

The full power of the template parameters will be explained later.

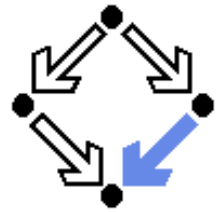
<numeric>: Examples



```
int myaccumulator (int x, int y) {return x-y;}
int myproduct (int x, int y) {return x+y;}

int init = 100;
int series1[] = {10,20,30};
int series2[] = {1,2,3};

cout << "using default inner_product: ";
cout << inner_product(series1,series1+3,series2,init);
cout << endl;

cout << "using custom functions: ";
cout << inner_product(series1,series1+3,series2,init,myaccumulator,myproduct);
cout << endl;

using default inner_product: 240
using custom functions: 34
```

The functions are directly applicable to plain arrays.

<numeric>: Examples



```
template<class T> T* valarray_begin(valarray<T>& a)
{ return &a[0]; }
template<class T> T* valarray_end(valarray<T>& a)
{ return valarray_begin(a)+a.size(); }

int init = 100;
int series1[] = {10,20,30};
int series2[] = {1,2,3};
valarray<int> s1(series1, 3);
valarray<int> s2(series2, 3);

cout << "using default inner_product: ";
cout << inner_product(valarray_begin(s1),valarray_end(s1),
                      valarray_begin(s2),init);
cout << endl;

using default inner_product: 240
```

With some provision, these functions are also applicable to valarrays.