

Nicolai M. Josuttis

The C++ Standard Library

A Tutorial and Reference

Second Edition

Supplementary Chapter

Print book ISBN-13: 978-0-321-62321-8

Print book ISBN-10: 0-321-62321-5

Copyright © 2012 Pearson Education, Inc.
All rights reserved.

Contents

This Supplementary Chapter provides additional material that could not be included in *The C++ Standard Library, Second Edition*, due to its size.

| | | |
|------------|--|-------------|
| S.1 | Bitsets | 1103 |
| S.1.1 | Examples of Using Bitsets | 1104 |
| S.1.2 | Class <code>bitset<></code> in Detail | 1107 |
| S.2 | Valarrays | 1114 |
| S.2.1 | Getting to Know Valarrays | 1114 |
| S.2.2 | Valarray Subsets | 1121 |
| S.2.3 | Class <code>valarray</code> in Detail | 1136 |
| S.2.4 | Valarray Subset Classes in Detail | 1142 |
| S.3 | Allocators and Memory Functions in Detail | 1148 |
| S.3.1 | Scoped Allocators | 1148 |
| S.3.2 | A User-Defined Allocator for C++98 | 1150 |
| S.3.3 | The Default Allocator | 1153 |
| S.3.4 | Allocators in Detail | 1155 |
| S.3.5 | Utilities for Uninitialized Memory in Detail | 1159 |

S.1 Bitsets

As introduced in Section 12.5, page 650, bitsets model fixed-sized arrays of bits or Boolean values and are useful for managing sets of flags, where variables may represent any combination of flags. C and old C++ programs usually use type `long` for arrays of bits and manipulate them with the bit operators, such as `&`, `|`, and `~`. The class `bitset` has the advantage that bitsets may contain any number of bits, and additional operations are provided. For example, you can assign single bits and can read and write bitsets as a sequence of 0s and 1s.

Note that you can't change the number of bits in a `bitset`. The number of bits is the template parameter. If you need a container for a variable number of bits or Boolean values, you can use the class `vector<bool>` (described in Section 7.3.6, page 281).

The class `bitset` is defined in the header file `<bitset>`:

```
#include <bitset>
```

In `<bitset>`, the class `bitset` is defined as a class template, with the number of bits as the template parameter:

```
namespace std {  
    template <size_t Bits>  
        class bitset;  
}
```

In this case, the template parameter is not a type but an unsigned integral value (see Section 3.2, page 33, for details about this language feature).

Templates with different template arguments are different types. You can compare and combine bitsets only with the same number of bits.

Recent Changes with C++11

C++98 specified almost all features of bitsets. Here is a list of the most important features added with C++11:

- Bitsets now can be initialized by string literals (see Section 12.5.1, page 653).
- Conversions to and from numeric values now support type `unsigned long long`. For this, `to_ulong()` was introduced (see Section 12.5.1, page 653).
- Conversions to and from strings now allow you to specify the character interpreted as set and unset bit.
- Member `all()` is now provided to check whether all bits are set.
- To use bitsets in unordered containers, a default hash function is provided (see Section 7.9.2, page 363).

S.1.1 Examples of Using Bitsets

Using Bitsets as Sets of Flags

The first example demonstrates how to use bitsets to manage a set of flags. Each flag has a value that is defined by an enumeration type. The value of the enumeration type is used as the position of the bit in the bitset. In particular, the bits represent colors. Thus, each enumeration value defines one color. By using a bitset, you can manage variables that might contain any combination of colors:

```
// contadapt/bitset1.cpp

#include <bitset>
#include <iostream>
using namespace std;

int main()
{
    // enumeration type for the bits
    // - each bit represents a color
    enum Color { red, yellow, green, blue, white, black, ...,
                numColors };

    // create bitset for all bits/colors
    bitset<numColors> usedColors;

    // set bits for two colors
    usedColors.set(red);
    usedColors.set(blue);

    // print some bitset data
    cout << "bitfield of used colors:  " << usedColors << endl;
    cout << "number    of used colors:  " << usedColors.count() << endl;
    cout << "bitfield of unused colors: " << ~usedColors << endl;

    // if any color is used
    if (usedColors.any()) {
        // loop over all colors
        for (int c = 0; c < numColors; ++c) {
            // if the actual color is used
            if (usedColors[(Color)c]) {
                ...
            }
        }
    }
}
```

Using Bitsets for I/O with Binary Representation

A useful feature of bitsets is the ability to convert integral values into a sequence of bits, and vice versa. This is done simply by creating a temporary bitset:

```
// contadapt/bitset2.cpp

#include <bitset>
#include <iostream>
#include <string>
#include <limits>
using namespace std;

int main()
{
    // print some numbers in binary representation
    cout << "267 as binary short:      "
         << bitset<numeric_limits<unsigned short>::digits>(267)
         << endl;

    cout << "267 as binary long:        "
         << bitset<numeric_limits<unsigned long>::digits>(267)
         << endl;

    cout << "10,000,000 with 24 bits:  "
         << bitset<24>(1e7) << endl;

    // write binary representation into string
    string s = bitset<42>(12345678).to_string();
    cout << "12,345,678 with 42 bits: " << s << endl;

    // transform binary representation into integral number
    cout << "\"1000101011\" as number:  "
         << bitset<100>("1000101011").to_ullong() << endl;
}
```

Depending on the number of bits for `short` and `long`, the program might produce the following output:

[illegible]

In this example, the following expression converts 267 into a bitset with the number of bits of type `unsigned short` (see Section 5.3, page 116, for a discussion of numeric limits):

```
bitset<numeric_limits<unsigned short>::digits>(267)
```

The output operator for `bitset` prints the bits as a sequence of characters 0 and 1.

You can output bitsets directly or use their value as a string:

```
string s = bitset<42>(12345678).to_string();
```

Note that before C++11, you had to write

```
string s = bitset<42>(12345678).to_string<char, char_traits<char>,
                                     allocator<char> >();
```

here because `to_string()` is a member template, and there were no default values for the template arguments defined.

Similarly, the following expression converts a sequence of binary characters into a bitset, for which `to_ullong()` yields the integral value:

```
bitset<100>("1000101011")
```

Note that the number of bits in the bitset should be smaller than `sizeof(unsigned long long)`. The reason is that you get an exception when the value of the bitset can't be represented as `unsigned long long`.¹

Note also that before C++11, you had to convert the initial value to type `string` explicitly:

```
bitset<100>(string("1000101011"))
```

¹ Before C++11, type `unsigned long` was not provided, so you could call only `to_ulong()` here. `to_ulong()` is still callable if the number of bits is smaller than `sizeof(unsigned long)`.

S.1.2 Class `bitset<>` in Detail

The `bitset` class provides the following operations.

Create, Copy, and Destroy Operations

For bitsets, some special constructors are defined. However, no special copy constructor, assignment operator, or destructor is defined. Thus, bitsets are assigned and copied with the default operations that copy bitwise.

`bitset<bits>::bitset ()`

- The default constructor.
- Creates a `bitset` with all bits initialized with zero.
- For example:

```
bitset<50> flags;           // flags: 0000...000000
                           // thus, 50 unset bits
```

`bitset<bits>::bitset (unsigned long long value)`

- Creates a `bitset` that is initialized according to the bits of the integral value *value*.
- If the number of bits of *value* is too small, the leading bit positions are initialized to zero.
- Before C++11, the type of *value* was `unsigned long`.
- For example:

```
bitset<50> flags(7);       // flags: 0000...000111
```

`explicit bitset<bits>::bitset (const string& str)`

`bitset<bits>::bitset (const string& str, string::size_type str_idx)`

`bitset<bits>::bitset (const string& str, string::size_type str_idx, string::size_type str_num)`

`bitset<bits>::bitset (const string& str, string::size_type str_idx, string::size_type str_num, string::charT zero)`

`bitset<bits>::bitset (const string& str, string::size_type str_idx, string::size_type str_num, string::charT zero, string::charT one)`

- All forms create a `bitset` that is initialized by the string *str* or a substring of *str*.
- The string or substring may contain only the characters '0' and '1' or *zero* and *one*.
- *str_idx* is the index of the first character of *str* that is used for initialization.
- If *str_num* is missing, all characters from *str_idx* to the end of *str* are used.
- If the string or substring has fewer characters than necessary, the leading bit positions are initialized to zero.

- If the string or substring has more characters than necessary, the remaining characters are ignored.
- Throw `out_of_range` if `str_idx > str.size()`.
- Throw `invalid_argument` if one of the characters is neither `'0'/zero` nor `'1'/one`.
- Parameters `zero` and `one` are provided since C++11.
- Note that this constructor is a member template (see Section 3.2, page 34). For this reason, no implicit type conversion from `const char*` to `string` for the first parameter is provided, which before C++11, ruled out passing a string literal.
- For example:

```
bitset<50> flags(string("1010101"));           //flags: 0000...0001010101
bitset<50> flags(string("1111000"),2,3);       //flags: 0000...0000000110
```

explicit **bitset**<*bits*>::**bitset** (const *charT** *str*)

bitset<*bits*>::**bitset** (const *charT** *str*, *string::size_type* *str_num*)

bitset<*bits*>::**bitset** (const *charT** *str*, *string::size_type* *str_num*,
 string::charT *zero*)

bitset<*bits*>::**bitset** (const *charT** *str*, *string::size_type* *str_num*,
 string::charT *zero*, *string::charT* *one*)

- All forms create a `bitset` that is initialized by the character sequence *str*.
- The string or substring may contain only the characters `'0'` and `'1'` or *zero* and *one*.
- If *str_num* is missing, all characters of *str* are used.
- If (*str_num* out of) *str* are fewer characters than necessary, the leading bit positions are initialized to zero.
- If *str* has more characters than necessary, the remaining characters are ignored.
- Throw `invalid_argument` if one of the characters is neither `'0'/zero` nor `'1'/one`.
- Parameters *zero* and *one* are provided since C++11.
- For example:

```
bitset<50> flags("1010101");           //flags: 0000...0001010101
```

Nonmanipulating Operations

size_t **bitset**<*bits*>::**size** () const

- Returns the number of bits (thus, *bits*).

size_t **bitset**<*bits*>::**count** () const

- Returns the number of set bits (bits with value 1).

bool **bitset**<*bits*>::**all** () const

- Returns whether all bits are set.
- Provided since C++11.

bool **bitset**<*bits*>::**any** () const

- Returns whether any bit is set.

bool **bitset**<*bits*>::**none** () const

- Returns whether no bit is set.

bool **bitset**<*bits*>::**test** (size_t *idx*) const

- Returns whether the bit at position *idx* is set.
- Throws `out_of_range` if *idx* >= `size()`.

bool **bitset**<*bits*>::**operator ==** (const **bitset**<*bits*>& *bits*) const

- Returns whether all bits of **this* and *bits* have the same value.

bool **bitset**<*bits*>::**operator !=** (const **bitset**<*bits*>& *bits*) const

- Returns whether any bits of **this* and *bits* have a different value.

Manipulating Operations

bitset<*bits*>& **bitset**<*bits*>::**set** ()

- Sets all bits to true.
- Returns the modified **bitset**.

bitset<*bits*>& **bitset**<*bits*>::**set** (size_t *idx*)

- Sets the bit at position *idx* to true.
- Returns the modified **bitset**.
- Throws `out_of_range` if *idx* >= `size()`.

bitset<*bits*>& **bitset**<*bits*>::**set** (size_t *idx*, bool *value*)

- Sets the bit at position *idx* according to *value*.
- Returns the modified **bitset**.
- Throws `out_of_range` if *idx* >= `size()`.
- Before C++11, type *value* had type `int` so that 0 set the bit to false, and any value other than 0 set it to true.

bitset<*bits*>& **bitset**<*bits*>::**reset** ()

- Resets all bits to false (assigns 0 to all bits).
- Returns the modified **bitset**.

`bitset<bits>& bitset<bits>::reset (size_t idx)`

- Resets the bit at position *idx* to false.
- Returns the modified bitset.
- Throws `out_of_range` if *idx* \geq `size()`.

`bitset<bits>& bitset<bits>::flip ()`

- Toggles all bits (sets unset bits and vice versa).
- Returns the modified bitset.

`bitset<bits>& bitset<bits>::flip (size_t idx)`

- Toggles the bit at position *idx*.
- Returns the modified bitset.
- Throws `out_of_range` if *idx* \geq `size()`.

`bitset<bits>& bitset<bits>::operator ^= (const bitset<bits>& bits)`

- The bitwise exclusive-OR operator.
- Toggles the value of all bits that are set in *bits* and leaves all other bits unchanged.
- Returns the modified bitset.

`bitset<bits>& bitset<bits>::operator |= (const bitset<bits>& bits)`

- The bitwise OR operator.
- Sets all bits that are set in *bits* and leaves all other bits unchanged.
- Returns the modified bitset.

`bitset<bits>& bitset<bits>::operator &= (const bitset<bits>& bits)`

- The bitwise AND operator.
- Resets all bits that are not set in *bits* and leaves all other bits unchanged.
- Returns the modified bitset.

`bitset<bits>& bitset<bits>::operator <<= (size_t num)`

- Shifts all bits by *num* positions to the left.
- Returns the modified bitset.
- The lowest *num* bits are set to false.

`bitset<bits>& bitset<bits>::operator >>= (size_t num)`

- Shifts all bits by *num* positions to the right.
- Returns the modified bitset.
- The highest *num* bits are set to false.

Access with Operator []

```
bitset<bits>::reference bitset<bits>::operator [ ] (size_t idx)
bool bitset<bits>::operator [ ] (size_t idx) const
```

- Both forms return the bit at position *idx*.
- The first form for nonconstant bitsets uses a proxy type to enable the use of the return value as a modifiable value (lvalue). See the next paragraphs for details.
- The caller must ensure that *idx* is a valid index; otherwise, the behavior is undefined.

When it is called for nonconstant bitsets, `operator []` returns a special temporary object of type `bitset<>::reference`. That object is used as a proxy² that allows certain modifications with the bit that is accessed by `operator []`. In particular, the following five operations are provided for references:

1. `reference& operator= (bool)`
Sets the bit according to the passed value.
2. `reference& operator= (const reference&)`
Sets the bit according to another reference.
3. `reference& flip ()`
Toggles the value of the bit.
4. `operator bool () const`
Converts the value into a Boolean value (automatically).
5. `bool operator~ () const`
Returns the complement (toggled value) of the bit.

For example, you can write the following statements:

```
bitset<50> flags;
...
flags[42] = true;           // set bit 42
flags[13] = flags[42];      // assign value of bit 42 to bit 13
flags[42].flip();          // toggle value of bit 42
if (flags[13]) {           // if bit 13 is set,
    flags[10] = ~flags[42]; // then assign complement of bit 42 to bit 10
}
```

Creating New Modified Bitsets

```
bitset<bits> bitset<bits>::operator ~ () const
```

- Returns a new bitset that has all bits toggled with respect to `*this`.

² A proxy allows you to keep control where usually no control is provided. This is often used to get more security. In this case, the proxy maintains control to allow certain operations, although the return value in principle behaves as `bool`.

```
bitset<bits> bitset<bits>::operator << (size_t num) const
```

- Returns a new bitset that has all bits shifted to the left by *num* position.

```
bitset<bits> bitset<bits>::operator >> (size_t num) const
```

- Returns a new bitset that has all bits shifted to the right by *num* position.

```
bitset<bits> operator & (const bitset<bits>& bits1,  
                        const bitset<bits>& bits2)
```

- Returns the bitwise computing of operator AND of *bits1* and *bits2*.
- Returns a new bitset that has only those bits set in *bits1* and in *bits2*.

```
bitset<bits> operator | (const bitset<bits>& bits1,  
                       const bitset<bits>& bits2)
```

- Returns the bitwise computing of operator OR of *bits1* and *bits2*.
- Returns a new bitset that has only those bits set in *bits1* or in *bits2*.

```
bitset<bits> operator ^ (const bitset<bits>& bits1,  
                       const bitset<bits>& bits2)
```

- Returns the bitwise computing of operator exclusive-OR of *bits1* and *bits2*.
- Returns a new bitset that has only those bits set in *bits1* and not set in *bits2* or vice versa.

Operations for Type Conversions

```
unsigned long bitset<bits>::to_ulong () const
```

- Returns the integral value that the bits of the bitset represent.
- Throws `overflow_error` if the integral value can't be represented by type `unsigned long`.

```
unsigned long long bitset<bits>::to_ullong () const
```

- Returns the integral value that the bits of the bitset represent.
- Throws `overflow_error` if the integral value can't be represented by type `unsigned long long`.
- Provided since C++11.

```
string bitset<bits>::to_string () const
```

```
string bitset<bits>::to_string (charT zero) const
```

```
string bitset<bits>::to_string (charT zero, charT one) const
```

- Returns a string that contains the value of the bitset as a binary representation written with characters '0' for unset bits and '1' for set bits.
- The order of the characters is equivalent to the order of the bits with descending index.
- Parameters *zero* and *one* are provided since C++11.

- For example:


```
bitset<50> b;
std::string s;
...
s = b.to_string();
```
- This is a function template that is parametrized only by the return type, which before C++11 had no default values. Thus, according to the language rules before C++11, you had to write the following:


```
s = b.to_string<char, char_traits<char>, allocator<char> >()
```

Input/Output Operations

istream& operator >> (istream& strm, bitset<bits>& bits)

- Reads into *bits* a bitset as a character sequence of characters '0' and '1'.
- Reads until any one of the following happens:
 - At most, *bits* characters are read.
 - End-of-file occurs in *strm*.
 - The next character is neither '0' nor '1'.
- Returns *strm*.
- If the number of bits read is less than the number of bits in the bitset, the bitset is filled with leading zeros.
- If it can't read any character, this operator sets `ios::failbit` in *strm*, which might throw the corresponding exception (see Section 15.4.4, page 762).

ostream& operator << (ostream& strm, const bitset<bits>& bits)

- Writes *bits* converted into a string that contains the binary representation (thus, as a sequence of '0' and '1').
- Uses `to_string()` (see Section S.1.2, page 1112) to create the output characters.
- Returns *strm*.
- See Section 12.5.1, page 652, for an example.

Hash Support

Since C++11, class `bitset<>` provides a specialization for a hash function:

```
namespace std {
    template <size_t N> struct hash<bitset<N> >;
}
```

See Section 7.9.2, page 363, for details.

S.2 Valarrays

Since C++98, the C++ standard library provides class `valarray<>` for processing arrays of numeric values. A valarray is a representation of the mathematical concept of a linear sequence of values. It has one dimension, but you can get the illusion of higher dimensionality by special techniques of computed indices and powerful subsetting capabilities. Therefore, a valarray can be used as a base both for vector and matrix operations and for processing mathematical systems of polynomial equations with good performance.

The valarray classes enable some tricky optimizations to get good performance for the processing of value arrays. However, it is not clear how important this component of the C++ standard library will be in the future, because other interesting developments perform even better. One of the most interesting examples is the Blitz system. If you are interested in numeric processing, you should look at it. For details, see <http://www.oonumerics.org/blitz/>.

The valarray classes were not designed very well. In fact, nobody tried to determine whether the final specification worked. This happened because nobody felt “responsible” for these classes. The people who introduced valarrays to the C++ standard library left the committee long before the first C++ standard was finished. For example, to use valarrays, you often need some inconvenient and time-consuming type conversions (see Section S.2.2, page 1121).

Recent Changes with C++11

C++98 specified almost all features of the classes for valarrays. Here is a list of the most important features added with C++11:

- Valarrays now support move semantics. In fact, a move constructor and a move assignment operator are provided.
- The class now provides `swap()` functions (see Section S.2.3, page 1137).
- You can now use an initializer list to initialize a valarray or assign new values to it (see Section S.2.1, page 1116).
- Class `valarray` now provides `begin()` and `end()` for iterating over the elements of a valarray (see Section S.2.1, page 1119).
- Operator `[]` now returns a constant reference instead of a copy (see Section S.2.3, page 1139).

S.2.1 Getting to Know Valarrays

Valarrays are one-dimensional arrays with elements numbered sequentially from zero. They provide the ability to do some numeric processing for all or a subset of the values in one or more value arrays. For example, you can process the statement

$$z = a*x*x + b*x + c$$

with `a`, `b`, `c`, `x`, and `z` being arrays that contain hundreds of numeric values. In doing this, you have the advantage of a simple notation. Also, the processing is done with good performance because the classes provide some optimizations that avoid the creation of temporary objects while processing the whole statement. In addition, special interfaces and auxiliary classes provide the ability to process

only a certain subset of value arrays or to do some multidimensional processing. In this way, the valarray concept also helps to implement vector and matrix operations and classes.

The standard guarantees that valarrays are alias free. That is, any value of a nonconstant valarray is accessed through a unique path. Thus, operations on these values can get optimized better because the compiler does not have to take into account that the data could be accessed through another path.

Header File

Valarrays are declared in the header file `<valarray>`:

```
#include <valarray>
```

In particular, the following classes are declared in `<valarray>`:

```
namespace std {
    template<typename T> class valarray;           // numeric array of type T

    class slice;                                   // slice out of a valarray
    template<typename T> class slice_array;

    class gslice;                                 // a generalized slice
    template<typename T> class gslice_array;

    template<typename T> class mask_array;        // a masked valarray

    template<typename T> class indirect_array;    // an indirect valarray
}
```

The classes have the following meanings:

- `valarray` is the core class that manages an array of numeric values.
- `slice` and `gslice` are provided to define a BLAS-like³ slice as a subset of a valarray.
- `slice_array`, `gslice_array`, `mask_array`, and `indirect_array` are internal auxiliary classes that are used to store temporary values or data. You can't use those classes in your programming interface directly. They are created indirectly by certain valarray operations.

All classes are templated for the type of the elements. In principle, the type could be any data type. However, according to the nature of valarrays, it should be a numeric data type.

Creating Valarrays

When you create a valarray, you usually pass the number of elements as a parameter:

```
std::valarray<int> va1(10);           // valarray of ten ints with value 0
```

³ The Basic Linear Algebra Subprograms (BLAS) library provides computational kernels for several of the fundamental linear algebra operations, such as matrix multiply, the solution of triangular systems, and simple vector operations.

```
std::valarray<float> va2(5.7,10); // valarray of ten floats with value 5.7
                                // (note the order)
```

If you pass one argument, it is used as the size. The elements are initialized by the default constructor of their type. Elements of fundamental data types are initialized by zero (see Section 3.2.1, page 37, for a description of why fundamental data types may be initialized by a default constructor). If you pass a second value, the first is used as the initial value for the elements, whereas the second specifies the number of elements. Note that the order of passing two arguments to the constructor differs from that of all other classes of the C++ standard library. All STL container classes use the first numeric argument as the number of elements and the second argument as the initial value.

Since C++11, you can also initialize a valarray with an initializer list:

```
std::valarray<int> va3 = { 3, 6, 18, 3, 22 };
```

Before C++11, you had to use an ordinary C-style array:

```
int array[] = { 3, 6, 18, 3, 22 };
```

```
// initialize valarray by elements of an ordinary array
```

```
std::valarray<int> va3(array, sizeof(array)/sizeof(array[0]));
```

```
// initialize by the second to the fourth element
```

```
std::valarray<int> va4(array+1,3);
```

The valarray creates copies of the passed values. Thus, you can pass temporary data for initialization.

Valarray Operations

For valarrays, the subscript operator is defined to access the element of a valarray. As usual, the first element has the index 0:

```
va[0] = 3 * va[1] + va[2];
```

In addition, all ordinary numeric operators are defined: addition, subtraction, multiplication, modulo, negation, bit operators, comparison operators, and logical operators, as well as all assignment operators. These operators are called for each valarray element that is processed by the operation. Thus, the result of a valarray operation is a valarray that has the same number of elements as the operands and that contains the result of the element-wise computation. For example, the statement

```
va1 = va2 * va3;
```

is equivalent to

```
va1[0] = va2[0] * va3[0];
```

```
va1[1] = va2[1] * va3[1];
```

```
va1[2] = va2[2] * va3[2];
```

```
...
```

If the number of elements of the combined valarrays differs, the result is undefined.

Of course, the operations are available only if the element's type supports them. The exact meaning of the operation depends on the meaning of the operation for the elements. Thus, all these operations simply do the same for each element or pair of elements in the valarrays they process.

For binary operations, one of the operands may be a single value of the element's type. In this case, the single value is combined with each element of the valarray that is used as the other operand. For example, the statement

```
va1 = 4 * va2;
```

is equivalent to

```
va1[0] = 4 * va2[0];
va1[1] = 4 * va2[1];
va1[2] = 4 * va2[2];
...
```

Note that the type of the single value has to match exactly the element type of the valarray. Thus, the previous example works only if the element type is `int`. The following statement would fail:

```
std::valarray<double> va(20);
...
va = 4 * va;    // ERROR: type mismatch
```

The schema of binary operations also applies to comparison operators. Thus, operator `==` does *not* return a single Boolean value that shows whether both valarrays are equal. Instead, the operator returns a new valarray with the same number of elements of type `bool`, where each value is the result of the individual comparison. For example, in the following code

```
std::valarray<double> va1(10);
std::valarray<double> va2(10);
std::valarray<bool> vab(10);
...
vab = (va1 == va2);
```

the last statement is equivalent to

```
vab[0] = (va1[0] == va2[0]);
vab[1] = (va1[1] == va2[1]);
vab[2] = (va1[2] == va2[2]);
...
vab[9] = (va1[9] == va2[9]);
```

For this reason, you can't sort valarrays by using operator `<`, and you can't use them as elements in STL containers if the test for equality is performed with operator `==` (see Section 6.11.1, page 244, for the requirements of elements of STL containers).

The following program demonstrates a simple use of valarrays:

```
// num/valarray1.cpp

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

// print valarray
```

```
template <typename T>
void printValarray (const valarray<T>& va)
{
    for (int i=0; i<va.size(); i++) {
        cout << va[i] << ' ';
    }
    cout << endl;
}

int main()
{
    // define two valarrays with ten elements
    valarray<double> va1(10), va2(10);

    // assign values 0.0, 1.1, up to 9.9 to the first valarray
    for (int i=0; i<10; i++) {
        va1[i] = i * 1.1;
    }

    // assign -1 to all elements of the second valarray
    va2 = -1;

    // print both valarrays
    printValarray(va1);
    printValarray(va2);

    // print minimum, maximum, and sum of the first valarray
    cout << "min(): " << va1.min() << endl;
    cout << "max(): " << va1.max() << endl;
    cout << "sum(): " << va1.sum() << endl;

    // assign values of the first to the second valarray
    va2 = va1;

    // remove all elements of the first valarray
    va1.resize(0);

    // print both valarrays again
    printValarray(va1);
    printValarray(va2);
}
```

The program has the following output:

```
0 1.1 2.2 3.3 4.4 5.5 6.6 7.7 8.8 9.9
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1
min(): 0
max(): 9.9
sum(): 49.5
```

```
0 1.1 2.2 3.3 4.4 5.5 6.6 7.7 8.8 9.9
```

Since C++11, you can use range-based for loops to process the elements, because valarrays provide global `begin()` and `end()` functions (see Section S.2.3, page 1139):

```
//print valarray (since C++11):
template <typename T>
void printValarray (const valarray<T>& va)
{
    for (const T& elem: va) {
        cout << elem << ' ';
    }
    cout << endl;
}
```

Transcendental Functions

The transcendental operations (trigonometric and exponential) are defined as equivalent to the numeric operators. In general, the operations are performed with all elements in the valarrays. For binary operations, one of the operands may be a single value instead, which is used as one operand, with all elements of the valarrays as the other operand.

All these operations are defined as global functions instead of member functions in order to provide automatic type conversion for subsets of valarrays for both operands (subsets of valarrays are covered in Section S.2.2, page 1121).

Here is a second example of the use of valarrays. It demonstrates the use of transcendental operations:

```
// num/valarray2.cpp

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

//print valarray
template <typename T>
void printValarray (const valarray<T>& va)
{
    for (int i=0; i<va.size(); i++) {
        cout << va[i] << ' ';
    }
}
```

```
        cout << endl;
    }

    int main()
    {
        // create and initialize valarray with nine elements
        valarray<double> va(9);
        for (int i=0; i<va.size(); i++) {
            va[i] = i * 1.1;
        }

        // print valarray
        printValarray(va);

        // double values in the valarray
        va *= 2.0;

        // print valarray again
        printValarray(va);

        // create second valarray initialized by the values of the first plus 10
        valarray<double> vb(va+10.0);

        // print second valarray
        printValarray(vb);

        // create third valarray as a result of processing both existing valarrays
        valarray<double> vc(9);
        vc = sqrt(va) + vb/2.0 - 1.0;

        // print third valarray
        printValarray(vc);
    }
```

The program has the following output:

```
0 1.1 2.2 3.3 4.4 5.5 6.6 7.7 8.8
0 2.2 4.4 6.6 8.8 11 13.2 15.4 17.6
10 12.2 14.4 16.6 18.8 21 23.2 25.4 27.6
4 6.58324 8.29762 9.86905 11.3665 12.8166 14.2332 15.6243 16.9952
```

S.2.2 Valarray Subsets

The subscript operator [] is overloaded for special auxiliary objects of valarrays. These auxiliary objects define subsets of valarrays in various ways, thus providing an elegant way to operate on certain subsets of valarrays (with both read and write access).

The subset of a valarray is defined by using a certain subset definition as the index. For example:

```
va[std::slice(2,4,3)]    // four elements with distance 3 starting from index 2
va[va>7]                 // all elements with a value greater than 7
```

If a subset definition, such as `std::slice(2,4,3)` or `va>7`, is used with a constant valarray, the expression returns a new valarray with the corresponding elements. However, if such a subset definition is used with a nonconstant valarray, the expression returns a temporary object of a special auxiliary valarray class. This temporary object does not contain the subset values but rather the definition of the subset. Thus, the evaluation of expressions is deferred until the values are needed to compute a final result.

This mechanism, called *lazy evaluation*, has the advantage that no temporary values for expressions are computed. This saves time and memory. In addition, the technique provides reference semantics. Thus, the subsets are logical sets of references to the original values. You can use these subsets as the destination (lvalue) of a statement. For example, you could assign one subset of a valarray the result of a multiplication of two other subsets of the same valarray.

However, because “temporaries” are avoided, some unexpected conditions might occur when elements in the destination subset are also used in a source subset. Therefore, any operation of valarrays is guaranteed to work only if the elements of the destination subset and the elements of all source subsets are distinct.

With smart definitions of subsets, you can give valarrays the semantics of two or more dimensions. Thus, in a way, valarrays may be used as multidimensional arrays.

There are four ways to define subsets of valarrays:

1. Slices
2. General slices
3. Masked subsets
4. Indirect subsets

Valarray Subset Problems

Before discussing the individual subsets, I have to mention a general problem. The handling of valarray subsets is not well designed. You can create subsets easily, but you can't combine them easily with other subsets. Unfortunately, you almost always need an explicit type conversion to valarray. The reason is that the C++ standard library does not specify that valarray subsets provide the same operations as valarrays.

For example, to multiply two subsets and assign the result to a third subset, you can't write the following:

```
// ERROR: conversions missing
va[std::slice(0,4,3)] = va[std::slice(1,4,3)] * va[std::slice(2,4,3)];
```

Instead, you have to code by using a cast:

```
va[std::slice(0,4,3)]
    = static_cast<std::valarray<double>>(va[std::slice(1,4,3)]) *
      static_cast<std::valarray<double>>(va[std::slice(2,4,3)]);
```

or by using an old-style cast:

```
va[std::slice(0,4,3)]
    = std::valarray<double>(va[std::slice(1,4,3)]) *
      std::valarray<double>(va[std::slice(2,4,3)]);
```

This is tedious and error prone. Even worse, without good optimization, it may cost performance because each cast creates a temporary object, which could be avoided without the cast.

To make the handling a bit more convenient, you can use the following function template:

```
// template to convert valarray subset into valarray
template <typename T>
inline
std::valarray<typename T::value_type> VA (const T& valarray_subset)
{
    return std::valarray<typename T::value_type>(valarray_subset);
}
```

By using this template, you could write

```
va[std::slice(0,4,3)] = VA(va[std::slice(1,4,3)]) *
                      VA(va[std::slice(2,4,3)]);    // OK
```

However, the performance penalty remains.

If you use a certain element type, you could also use a simple type definition:

```
typedef valarray<double> VAD;
```

By using this type definition, you could also write

```
va[std::slice(0,4,3)] = VAD(va[std::slice(1,4,3)]) *
                      VAD(va[std::slice(2,4,3)]);    // OK
```

provided that the elements of `va` have type `double`.

Slices

A slice defines a set of indices that has three properties:

1. The starting index
2. The number of elements (size)
3. The distance between elements (stride)

You can pass these three properties exactly in the same order as parameters to the constructor of class `slice`. For example, the following expression specifies four elements, starting with index 2 with distance 3:

```
std::slice(2,4,3)
```

In other words, the expression specifies the following set of indices:

```
2 5 8 11
```

The stride may be negative. For example, the expression

```
std::slice(9,5,-2)
```

specifies the following indices:

```
9 7 5 3 1
```

To define the subset of a valarray, you simply use a slice as an argument of the subscript operator. For example, the following expression specifies the subset of the valarray `va` that contains the elements with the indices 2, 5, 8, and 11:

```
va[std::slice(2,4,3)]
```

It's up to the caller to ensure that all these indices are valid.

If the subset qualified by a slice is a subset of a constant valarray, the subset is a new valarray. If the valarray is nonconstant, the subset has reference semantics to the original valarray. The auxiliary class `slice_array` is provided for this:

```
namespace std {
    class slice;

    template <typename T>
    class slice_array;

    template <typename T>
    class valarray {
    public:
        // slice of a constant valarray returns a new valarray
        valarray<T> operator[] (slice) const;

        // slice of a variable valarray returns a slice_array
        slice_array<T> operator[] (slice);
        ...
    };
}
```

For `slice_arrays`, the following operations are defined:

- Assign a single value to all elements.
- Assign another valarray (or valarray subset).
- Call any computed assignment operation, such as operators `+=` and `*=`.

For any other operation, you have to convert the subset to a valarray (see Section S.2.2, page 1121). Note that the class `slice_array<>` is intended purely as an internal helper class for slices and should be transparent to the user. Thus, all constructors and the assignment operator of class `slice_array<>` are private.

For example, the statement

```
va[std::slice(2,4,3)] = 2;
```

assigns 2 to the third, sixth, ninth, and twelfth elements of the valarray `va`. It is equivalent to the following statements:

```
va[2] = 2;
va[5] = 2;
va[8] = 2;
va[11] = 2;
```

As another example, the following statement squares the values of the elements with index 2, 5, 8, and 11:

```
va[std::slice(2,4,3)]
    *= std::valarray<double>(va[std::slice(2,4,3)]);
```

As mentioned in Section S.2.2, page 1121, you can't write

```
va[std::slice(2,4,3)] *= va[std::slice(2,4,3)];    // ERROR
```

But using the `VA()` function template mentioned in Section S.2.2, page 1122, you can write

```
va[std::slice(2,4,3)] *= VA(va[std::slice(2,4,3)]);    // OK
```

By passing different slices of the same valarray, you can combine different subsets and store the result in another subset of the valarray. For example, the statement

```
va[std::slice(0,4,3)] = VA(va[std::slice(1,4,3)]) *
                        VA(va[std::slice(2,4,3)]);
```

is equivalent to the following:

```
va[0] = va[1] * va[2];
va[3] = va[4] * va[5];
va[6] = va[7] * va[8];
va[9] = va[10] * va[11];
```

If you consider your valarray as a two-dimensional matrix, this example is nothing else but vector multiplication (Figure S.1). However, note that the order of the individual assignments is not defined. Therefore, the behavior is undefined if the destination subset contains elements that are used in the source subsets.

In the same way, more complicated statements are possible. For example:

```
va[std::slice(0,100,3)]
    = std::pow(VA(va[std::slice(1,100,3)]) * 5.0,
              VA(va[std::slice(2,100,3)]));
```

Note again that a single value, such as 5.0 in this example, has to match the element type of the valarray exactly.

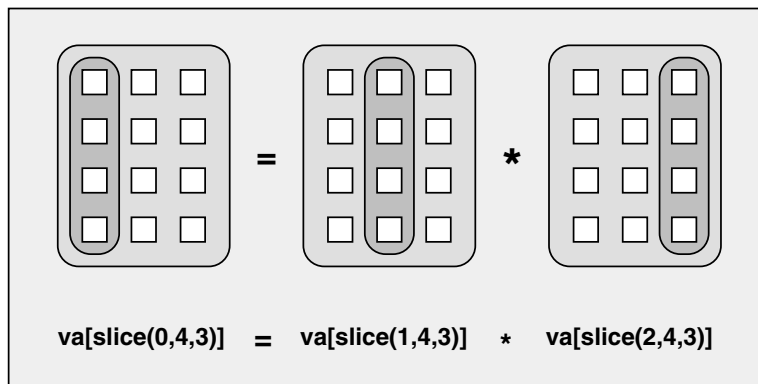


Figure S.1. Vector Multiplication by Valarray Slices

The following program demonstrates a complete example of using valarray slices:

```
// num/slice1.cpp

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

// print valarray line-by-line
template <typename T>
void printValarray (const valarray<T>& va, int num)
{
    for (int i=0; i<va.size()/num; ++i) {
        for (int j=0; j<num; ++j) {
            cout << va[i*num+j] << ' ';
        }
        cout << endl;
    }
    cout << endl;
}

int main()
{
    // valarray with 12 elements
    // - four rows
    // - three columns
    valarray<double> va(12);

    // fill valarray with values
```

```

    for (int i=0; i<12; i++) {
        va[i] = i;
    }

    printValarray (va, 3);

    // first column = second column raised to the third column
    va[slice(0,4,3)] = pow (valarray<double>(va[slice(1,4,3)]),
                           valarray<double>(va[slice(2,4,3)]));

    printValarray (va, 3);

    // create valarray with three times the third element of va
    valarray<double> vb(va[slice(2,4,0)]);

    // multiply the third column by the elements of vb
    va[slice(2,4,3)] *= vb;

    printValarray (va, 3);

    // print the square root of the elements in the second row
    printValarray (sqrt(valarray<double>(va[slice(3,3,1)])));

    // double the elements in the third row
    va[slice(2,4,3)] = valarray<double>(va[slice(2,4,3)]) * 2.0;

    printValarray (va, 3);
}

```

The program has the following output:

```

0 1 2
3 4 5
6 7 8
9 10 11

1 1 2
1024 4 5
5.7648e+006 7 8
1e+011 10 11

1 1 4
1024 4 10
5.7648e+006 7 16
1e+011 10 22

```

```

32 2 3.16228

1 1 8
1024 4 20
5.7648e+006 7 32
1e+011 10 44

```

General Slices

General slices, or *gslices*, are the general form of slices. Similar to slices, which provide the ability to handle a subset that is one out of two dimensions, *gslices* allow the handling of subsets of multidimensional arrays. In principle, *gslices* have the same properties as slices:

- Starting index
- Number of elements (size)
- Distance between elements (stride)

Unlike slices, however, the number and distance of elements in a *gslice* are arrays of values. The number of elements in such an array is equivalent to the number of dimensions used. For example, a *gslice* having the state

```

start:  2
size:   [ 4 ]
stride: [ 3 ]

```

is equivalent to a slice because the array handles one dimension. Thus, the *gslice* defines four elements with distance 3, starting with index 2:

```

2  5  8  11

```

However, a *gslice* having the state

```

start:  2
size:   [  2 4 ]
stride: [ 10 3 ]

```

handles two dimensions. The smallest index handles the highest dimension. Thus, this *gslice* specifies starting from index 2, twice with distance 10, four elements with distance 3:

```

2   5   8  11
12  15  18  21

```

Here is an example of a slice with three dimensions:

```

start:  2
size:   [  3 2 4 ]
stride: [ 30 10 3 ]

```

It specifies starting from index 2, three times with distance 30, twice with distance 10, four elements with distance 3:

```

    2   5   8  11
12  15  18  21

32  35  38  41
42  45  48  51

62  65  68  71
72  75  78  81

```

The ability to use arrays to define size and stride is the only difference between `gslices` and `slices`. Apart from this, `gslices` behave the same as `slices` in the following five ways:

1. To define a concrete subset of a valarray, you simply pass a `gslice` as the argument to the subscript operator of the valarray.
2. If the valarray is constant, the resulting expression is a new valarray.
3. If the valarray is nonconstant, the resulting expression is a `gslice_array` that represents the elements of the valarray with reference semantics:

```

namespace std {
    class gslice;

    template <typename T>
    class gslice_array;

    template <typename T>
    class valarray {
    public:
        // gslice of a constant valarray returns a new valarray
        valarray<T> operator[] (const gslice&) const;

        // gslice of a variable valarray returns a gslice_array
        gslice_array<T> operator[] (const gslice&);
        ...
    };
}

```

4. For `gslice_array`, the assignment and computed assignment operators are provided to modify the elements of the subset.
5. By using type conversions, you can combine a `gslice` array with other valarrays and subsets of valarrays (see Section S.2.2, page 1121).

The following program demonstrates the use of valarray `gslices`:

```

// num/gslice1.cpp

#include <iostream>
#include <valarray>

```

```

using namespace std;

// print three-dimensional valarray line-by-line
template <typename T>
void printValarray3D (const valarray<T>& va, int dim1, int dim2)
{
    for (int i=0; i<va.size()/(dim1*dim2); ++i) {
        for (int j=0; j<dim2; ++j) {
            for (int k=0; k<dim1; ++k) {
                cout << va[i*dim1*dim2+j*dim1+k] << ' ';
            }
            cout << '\n';
        }
        cout << '\n';
    }
    cout << endl;
}

int main()
{
    // valarray with 24 elements
    // - two groups
    // - four rows
    // - three columns
    valarray<double> va(24);

    // fill valarray with values
    for (int i=0; i<24; i++) {
        va[i] = i;
    }

    // print valarray
    printValarray3D (va, 3, 4);

    // we need two two-dimensional subsets of three times 3 values
    // in two 12-element arrays
    size_t lengthvalues[] = { 2, 3 };
    size_t stridevalues[] = { 12, 3 };
    valarray<size_t> length(lengthvalues,2);
    valarray<size_t> stride(stridevalues,2);

    // assign the second column of the first three rows
    // to the first column of the first three rows

```

```

    va[gslice(0,length,stride)]
        = valarray<double>(va[gslice(1,length,stride)]);

    // add and assign the third of the first three rows
    // to the first of the first three rows
    va[gslice(0,length,stride)]
        += valarray<double>(va[gslice(2,length,stride)]);

    // print valarray
    printValarray3D (va, 3, 4);
}

```

The program has the following output:

```

0 1 2
3 4 5
6 7 8
9 10 11

12 13 14
15 16 17
18 19 20
21 22 23

3 1 2
9 4 5
15 7 8
9 10 11

27 13 14
33 16 17
39 19 20
21 22 23

```

Masked Subsets

Mask arrays provide another way to define a subset of a valarray. You can mask the elements by using a Boolean expression. For example, in the expression

```
va[va > 7]
```

the subexpression

```
va > 7
```

returns a valarray with the size of `va`, where a Boolean value states whether each element is greater than 7. The subscript operator uses the Boolean valarray to specify all elements for which the Boolean expression yields true. Thus, in the valarray `va`,

```
va[va > 7]
```

specifies the subset of elements that is greater than 7.

Apart from this, mask arrays behave the same as all valarray subsets in the following five ways:

1. To define a concrete subset of a valarray, you simply pass a valarray of Boolean values as the argument to the subscript operator of the valarray.
2. If the valarray is constant, the resulting expression is a new valarray.
3. If the valarray is nonconstant, the resulting expression is a `mask_array` that represents the elements of the valarray with reference semantics:

```
namespace std {
    template <typename T>
    class mask_array;

    template <typename T>
    class valarray {
    public:
        // masking a constant valarray returns a new valarray
        valarray<T> operator[] (const valarray<bool>&) const;

        // masking a variable valarray returns a mask_array
        mask_array<T> operator[] (const valarray<bool>&);
        ...
    };
}
```

4. For `mask_array`, the assignment and computed assignment operators are provided to modify the elements of the subset.
5. By using type conversions, you can combine a mask array with other valarrays and subsets of valarrays (see Section S.2.2, page 1121).

The following program demonstrates the use of masked subsets of valarrays:

```
// num/maskarray1.cpp

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

// print valarray line-by-line
template <typename T>
void printValarray (const valarray<T>& va, int num)
{
    for (int i=0; i<va.size()/num; ++i) {
```

```

        for (int j=0; j<num; ++j) {
            cout << va[i*num+j] << ' ';
        }
        cout << endl;
    }
    cout << endl;
}

int main()
{
    // valarray with 12 elements
    // - four rows
    // - three columns
    valarray<double> va(12);

    // fill valarray with values
    for (int i=0; i<12; i++) {
        va[i] = i;
    }

    printValarray (va, 3);

    // assign 77 to all values that are less than 5
    va[va<5.0] = 77.0;

    // add 100 to all values that are greater than 5 and less than 9
    va[va>5.0 && va<9.0]
        = valarray<double>(va[va>5.0 && va<9.0]) + 100.0;

    printValarray (va, 3);
}

```

The program has the following output:

```

0 1 2
3 4 5
6 7 8
9 10 11

77 77 77
77 77 5
106 107 108
9 10 11

```


Note that the type of a numeric value that is compared with the valarray has to match the type of the valarray exactly. So, using an `int` value to compare it with a valarray of doubles would not compile:

```
valarray<double> va(12);
...
va[va<5] = 77;    //ERROR
```

Indirect Subsets

Indirect arrays provide the fourth and last way to define a subset of a valarray. Here, you simply define the subset of a valarray by passing an array of indices. Note that the indices that specify the subset don't have to be sorted and may occur twice.

Apart from this, indirect arrays behave the same as all valarray subsets in the following five ways:

1. To define a concrete subset of a valarray, you simply pass a valarray of elements of type `size_t` as the argument to the subscript operator of the valarray.
2. If the valarray is constant, the resulting expression is a new valarray.
3. If the valarray is nonconstant, the resulting expression is an `indirect_array` that represents the elements of the valarray with reference semantics:

```
namespace std {
    template <typename T>
    class indirect_array;

    template <typename T>
    class valarray {
    public:
        // indexing a constant valarray returns a new valarray
        valarray<T> operator[] (const valarray<size_t>&) const;

        // indexing a variable valarray returns an indirect_array
        indirect_array<T> operator[] (const valarray<size_t>&);
        ...
    };
}
```

4. For `indirect_array`, the assignment and computed assignment operators are provided to modify the elements of the subset.
5. By using type conversions, you can combine an indirect array with other valarrays and subsets of valarrays (see Section S.2.2, page 1121).

The following program demonstrates how to use indirect arrays:

```
// num/indirectarray1.cpp

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

// print valarray as two-dimensional array
template <typename T>
void printValarray (const valarray<T>& va, int num)
{
    for (int i=0; i<va.size()/num; i++) {
        for (int j=0; j<num; j++) {
            cout << va[i*num+j] << ' ';
        }
        cout << endl;
    }
    cout << endl;
}

int main()
{
    // create valarray for 12 elements
    valarray<double> va(12);

    // initialize valarray by values 1.01, 2.02, ... 12.12
    for (int i=0; i<12; i++) {
        va[i] = (i+1) * 1.01;
    }
    printValarray(va,4);

    // create array of indexes
    // - note: element type has to be size_t
    valarray<size_t> idx(4);
    idx[0] = 8;
    idx[1] = 0;
    idx[2] = 3;
    idx[3] = 7;

    // use array of indexes to print the ninth, first, fourth, and eighth elements
    printValarray(valarray<double>(va[idx]), 4);
}
```

```
// change the first and fourth elements and print them again indirectly
va[0] = 11.11;
va[3] = 44.44;
printValarray(valarray<double>(va[idx]), 4);

// now select the second, third, sixth, and ninth elements
// and assign 99 to them
idx[0] = 1;
idx[1] = 2;
idx[2] = 5;
idx[3] = 8;
va[idx] = 99;

// print the whole valarray again
printValarray (va, 4);
}
```

The valarray `idx` is used to define the subset of the elements in valarray `va`. The program has the following output:

```
1.01 2.02 3.03 4.04
5.05 6.06 7.07 8.08
9.09 10.1 11.11 12.12

9.09 1.01 4.04 8.08

9.09 11.11 44.44 8.08

11.11 99 99 44.44
5.05 99 7.07 8.08
99 10.1 11.11 12.12
```

S.2.3 Class `valarray` in Detail

The class template `valarray<>` is the core part of the `valarray` component. It is parametrized by the type of the elements:

```
namespace std {  
    template <typename T>  
        class valarray;  
}
```

The size is not part of the type. Thus, in principle, you can process `valarrays` with different sizes and can change the size. However, changing the size of a `valarray` is provided only to make a two-step initialization (creating and setting the size), which is necessary to manage arrays of `valarrays`. Note that the result of combining `valarrays` of different size is undefined.

Create, Copy, and Destroy Operations

`valarray::valarray ()`

- The default constructor.
- Creates an empty `valarray`.
- This constructor is provided only to enable the creation of arrays of `valarrays`. The next step is to give them the correct size by using the `resize()` member function.

`explicit valarray::valarray (size_t num)`

- Creates a `valarray` that contains *num* elements.
- The elements are initialized by their default constructors, which is 0 for fundamental data types.

`valarray::valarray (initializer-list)`

- Creates a `valarray` that contains the values of *initializer-list*.
- Available since C++11.

`valarray::valarray (const T& value, size_t num)`

- Creates a `valarray` that contains *num* elements.
- The elements are initialized by *value*.
- Note that the order of parameters is unusual. All other classes of the C++ standard library provide an interface in which *num* is the first parameter and *value* is the second parameter.

`valarray::valarray (const T* array, size_t num)`

- Creates a `valarray` that contains *num* elements.
- The elements are initialized by the values of the elements in *array*.
- The caller must ensure that *array* contains *num* elements; otherwise, the behavior is undefined.

valarray::valarray (const *valarray*& *va*)

valarray::valarray (*valarray*&& *va*)

- The copy and move constructor.
- Creates a valarray as a copy of *va* or with the elements of *va* moved.
- The move constructor is provided since C++11.

valarray::~~valarray ()

- The destructor.
- Destroys all elements and frees the memory.

In addition, you can create valarrays initialized by objects of the internal auxiliary classes `slice_array`, `gslice_array`, `mask_array`, and `indirect_array`. See pages 1142, 1144, 1145, and 1146, respectively, for details about these classes.

Assignment Operations

valarray& **valarray::operator =** (const *valarray*& *va*)

valarray& **valarray::operator =** (*valarray*&& *va*)

- Copy or move assigns the elements of the valarray *va*.
- If *va* has a different size, the behavior is undefined.
- The value of an element on the left side of any valarray assignment operator should not depend on the value of another element on that left side. In other words, if an assignment overwrites values that are used on the right side of the assignment, the result is undefined. This means that you should not use an element on the left side anywhere in the expression on the right side. The reason is that the order of the evaluation of valarray statements is not defined. See Section S.2.2, page 1124, and Section S.2.2, page 1121, for details.
- The move assignment operator is provided since C++11.

void **valarray::swap** (*valarray*& *va2*)

void **swap** (*valarray*& *va1*, *valarray*& *va2*)

- Both forms swap the value of two valarrays:
 - The member function swaps the contents of `*this` and *va2*.
 - The global function swaps the contents of *va1* and *va2*.
- If possible, you should prefer these functions over assignment because they are faster. In fact, they are guaranteed to have constant complexity.
- Provided since C++11.

valarray& **valarray::operator =** (*initializer-list*)

- Assigns the values of *initializer-list*.
- Returns `*this`.
- Available since C++11.

`valarray& valarray::operator = (const T& value)`

- Assigns *value* to each element of the valarray.
- The size of the valarray is not changed. Pointers and references to the elements remain valid.

In addition, you can assign values of the internal auxiliary classes `slice_array`, `gslice_array`, `mask_array`, and `indirect_array`. See pages 1142, 1144, 1145, and 1146, respectively, for details about these classes.

Member Functions

Class `valarray` provides the following member functions.

`size_t valarray::size () const`

- Returns the current number of elements.

`void valarray::resize (size_t num)`

`void valarray::resize (size_t num, T value)`

- Both forms change the size of the valarray to *num*.
- If the size grows, the new elements are initialized by their default constructor or with *value*, respectively.
- Both forms invalidate all pointers and references to elements of the valarray.
- These functions are provided only to enable the creation of arrays of valarrays. After creating them with the default constructor, you should give them the correct size by calling this function.

`T valarray::min () const`

`T valarray::max () const`

- The first form returns the minimum value of all elements.
- The second form returns the maximum value of all elements.
- The elements are compared with operator `<` or `>`, respectively. Thus, these operators must be provided for the element type.
- If the valarray contains no elements, the return value is undefined.

`T valarray::sum () const`

- Returns the sum of all elements.
- The elements are processed by operator `+=`. Thus, this operator has to be provided for the element type.
- If the valarray contains no elements, the return value is undefined.

`valarray valarray::shift (int num) const`

- Returns a new valarray in which all elements are shifted by *num* positions.
- The returned valarray has the same number of elements.
- Elements of positions that were shifted are initialized by their default constructor.

- The direction of the shifting depends on the sign of *num*:
 - If *num* is positive, it shifts to the left/front. Thus, elements get a smaller index.
 - If *num* is negative, it shifts to the right/back. Thus, elements get a higher index.

valarray **valarray::cshift** (int *num*) const

- Returns a new valarray in which all elements are shifted cyclically by *num* positions.
- The returned valarray has the same number of elements.
- The direction of the shifting depends on the sign of *num*:
 - If *num* is positive, it shifts to the left/front. Thus, elements get a smaller index or are inserted at the back.
 - If *num* is negative, it shifts to the right/back. Thus, elements get a higher index or are inserted at the front.

valarray **valarray::apply** (T *op*(T)) const

valarray **valarray::apply** (T *op*(const T&)) const

- Both forms return a new valarray with all elements processed by *op*().
- The returned valarray has the same number of elements.
- For each element of **this*, it calls
 op(elem)
 and initializes the corresponding element in the new returned valarray by its result.

Element Access

T& **valarray::operator []** (size_t *idx*)

const *T&* **valarray::operator []** (size_t *idx*) const

- Both forms return the valarray element that has index *idx* (the first element has index 0).
- The nonconstant version returns a reference. So, you can modify the element that is specified and returned by this operator. The reference is guaranteed to be valid as long as the valarray exists, and no function is called that modifies the size of the valarray.
- Before C++11, the return type of the second form was just *T*.

iterator **begin** (*valarray& va*)

const_iterator **begin** (const *valarray& va*)

- Both forms return a random-access iterator for the beginning of the valarray (the position of the first element).
- The name of the iterator type is undefined.
- If the container is empty, the calls are equivalent to *valarray::end*().
- Available since C++11.

iterator **end** (*valarray& va*)

const_iterator **end** (*const valarray& va*)

- Both forms return a random-access iterator for the end of the container (the position after the last element).
- The name of the iterator type is undefined.
- If the container is empty, the calls are equivalent to *valarray::begin()*.
- Available since C++11.

Valarray Operators

Unary valarray operators have the following format:

valarray ***valarray::unary-op*** () *const*

- A unary operator returns a new valarray that contains all values of **this* modified by ***unary-op***.
- ***unary-op*** may be any of the following:
 - operator +
 - operator -
 - operator ~
 - operator !
- The return type for operator ! is *valarray<bool>*.

Binary valarray operators (except comparison and assignment operators) have the following format:

valarray ***binary-op*** (*const valarray& va1*, *const valarray& va2*)

valarray ***binary-op*** (*const valarray& va*, *const T& value*)

valarray ***binary-op*** (*const T& value*, *const valarray& va*)

- These operators return a new valarray with the same number of elements as *va*, *va1*, or *va2*. The new valarray contains the result of computing ***binary-op*** for each value pair.
- If only one operand is passed as a single *value*, that operand is combined with each element of *va*.
- ***binary-op*** may be any of the following:
 - operator +
 - operator -
 - operator *
 - operator /
 - operator %
 - operator ^
 - operator &
 - operator |
 - operator <<
 - operator >>
- If *va1* and *va2* have different numbers of elements, the result is undefined.

The logical and comparison operators follow the same schema. However, their return values are a valarray of Boolean values:

```
valarray<bool> logical-op (const valarray& va1, const valarray& va2)
```

```
valarray<bool> logical-op (const valarray& va, const T& value)
```

```
valarray<bool> logical-op (const T& value, const valarray& va)
```

- These operators return a new valarray with the same number of elements as *va*, *va1*, or *va2*. The new valarray contains the result of computing **logical-op** for each value pair.
- If only one operand is passed as a single *value*, that operand is combined with each element of *va*.
- **logical-op** may be any of the following:
 - operator ==
 - operator !=
 - operator <
 - operator <=
 - operator >
 - operator >=
 - operator &&
 - operator ||
- If *va1* and *va2* have different numbers of elements, the result is undefined.

Similarly, computed assignment operators are defined for valarrays:

```
valarray& valarray::assign-op (const valarray& va)
```

```
valarray& valarray::assign-op (const T& value)
```

- Both forms call for each element in **this* **assign-op** with the corresponding element of *va* or *value*, respectively, as the second operand.
- They return a reference to the modified valarray.
- **assign-op** may be any of the following:
 - operator +=
 - operator -=
 - operator *=
 - operator /=
 - operator %=
 - operator &=
 - operator |=
 - operator ^=
 - operator <<=
 - operator >>=
- If **this* and *va2* have different numbers of elements, the result is undefined.
- References and pointers to modified elements stay valid as long as the valarray exists, and no function is called that modifies the size of the valarray.

Transcendental Functions

```

valarray abs (const valarray& va)
valarray pow (const valarray& va1, const valarray& va2)
valarray pow (const valarray& va, const T& value)
valarray pow (const T& value, const valarray& va)
valarray exp (const valarray& va)
valarray sqrt (const valarray& va)
valarray log (const valarray& va)
valarray log10 (const valarray& va)
valarray sin (const valarray& va)
valarray cos (const valarray& va)
valarray tan (const valarray& va)
valarray sinh (const valarray& va)
valarray cosh (const valarray& va)
valarray tanh (const valarray& va)
valarray asin (const valarray& va)
valarray acos (const valarray& va)
valarray atan (const valarray& va)
valarray atan2 (const valarray& va1, const valarray& va2)
valarray atan2 (const valarray& va, const T& value)
valarray atan2 (const T& value, const valarray& va)

```

- All these functions return a new valarray with the same number of elements as *va*, *va1*, or *va2*. The new valarray contains the result of the corresponding operation called for each element or pair of elements.
- If *va1* and *va2* have different numbers of elements, the result is undefined.

S.2.4 Valarray Subset Classes in Detail

This subsection describes the subset classes for valarray in detail. However, these classes are very simple and do not provide many operations. For this reason, I provide only their declarations and a few remarks.

Class `slice` and Class `slice_array`

Objects of class `slice_array` are created by using a `slice` as the index of a nonconstant valarray:

```

namespace std {
    template <typename T>
    class valarray {

```

```

        public:
            ...
            slice_array<T> operator[] (slice);
            ...
    };
}

```

The exact definition of the public interface of class `slice` is as follows:

```

namespace std {
    class slice {
    public:
        slice ();           // empty subset
        slice (size_t start, size_t size, size_t stride);

        size_t start() const;
        size_t size() const;
        size_t stride() const;
    };
}

```

The default constructor creates an empty subset. With the `start()`, `size()`, and `stride()` member functions, you can query the properties of a slice.

The class `slice_array` provides the following operations:

```

namespace std {
    template <typename T>
    class slice_array {
    public:
        typedef T value_type;

        void operator= (const T&);
        void operator= (const valarray<T>&) const;
        void operator*= (const valarray<T>&) const;
        void operator/= (const valarray<T>&) const;
        void operator%=(const valarray<T>&) const;
        void operator+=(const valarray<T>&) const;
        void operator-=(const valarray<T>&) const;
        void operator^=(const valarray<T>&) const;
        void operator&=(const valarray<T>&) const;
        void operator|=(const valarray<T>&) const;
        void operator<<=(const valarray<T>&) const;
        void operator>>=(const valarray<T>&) const;
        ~slice_array();
    private:
        slice_array();
    };
}

```

```

        slice_array(const slice_array&);
        slice_array& operator=(const slice_array&);
        ...
    };
}

```

Note that class `slice_array<>` is intended purely as an internal helper class for slices and should be transparent to the user. Thus, all constructors and the assignment operator of class `slice_array<>` are private.

Class `gslice` and Class `gslice_array`

Objects of class `gslice_array` are created by using a `gslice` as the index of a nonconstant valarray:

```

namespace std {
    template <typename T>
    class valarray {
    public:
        ...
        gslice_array<T> operator[] (const gslice&);
        ...
    };
}

```

The exact definition of the public interface of `gslice` is as follows:

```

namespace std {
    class gslice {
    public:
        gslice ();           // empty subset
        gslice (size_t start,
                const valarray<size_t>& size,
                const valarray<size_t>& stride);

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

```

The default constructor creates an empty subset. With the `start()`, `size()`, and `stride()` member functions, you can query the properties of a `gslice`.

The class `gslice_array` provides the following operations:

```

namespace std {
    template <typename T>
    class gslice_array {

```

```

public:
    typedef T value_type;

    void operator= (const T&);
    void operator= (const valarray<T>&) const;
    void operator*= (const valarray<T>&) const;
    void operator/= (const valarray<T>&) const;
    void operator%= (const valarray<T>&) const;
    void operator+= (const valarray<T>&) const;
    void operator-= (const valarray<T>&) const;
    void operator^= (const valarray<T>&) const;
    void operator&= (const valarray<T>&) const;
    void operator|= (const valarray<T>&) const;
    void operator<=(const valarray<T>&) const;
    void operator>=(const valarray<T>&) const;
    ~gslice_array();
private:
    gslice_array();
    gslice_array(const gslice_array<T>&);
    gslice_array& operator=(const gslice_array<T>&);
    ...
};
}

```

As with `slice_array<>`, note that class `gslice_array<>` is intended purely as an internal helper class for `gslices` and should be transparent to the user. Thus, all constructors and the assignment operator of class `gslice_array<>` are private.

Class `mask_array`

Objects of class `mask_array` are created by using a `valarray<bool>` as the index of a nonconstant `valarray`:

```

namespace std {
    template <typename T>
    class valarray {
    public:
        ...
        mask_array<T> operator[] (const valarray<bool>&);
        ...
    };
}

```

The class `mask_array` provides the following operations:

```

namespace std {

```

```

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

    void operator= (const T&);
    void operator= (const valarray<T>&) const;
    void operator*= (const valarray<T>&) const;
    void operator/= (const valarray<T>&) const;
    void operator%= (const valarray<T>&) const;
    void operator+= (const valarray<T>&) const;
    void operator-= (const valarray<T>&) const;
    void operator^= (const valarray<T>&) const;
    void operator&= (const valarray<T>&) const;
    void operator|= (const valarray<T>&) const;
    void operator<=(const valarray<T>&) const;
    void operator>=(const valarray<T>&) const;
    ~mask_array();
private:
    mask_array();
    mask_array(const mask_array<T>&);
    mask_array& operator=(const mask_array<T>&);
    ...
};
}

```

Again, note that class `mask_array<>` is intended purely as an internal helper class and should be transparent to the user. Thus, all constructors and the assignment operator of class `mask_array<>` are private.

Class `indirect_array`

Objects of class `indirect_array` are created by using a `valarray<size_t>` as the index of a nonconstant `valarray`:

```

namespace std {
    template <typename T>
    class valarray {
    public:
        ...
        indirect_array<T> operator[] (const valarray<size_t>&);
        ...
    };
}

```

The class `indirect_array` provides the following operations:

```

namespace std {
    template <typename T>
    class indirect_array {
    public:
        typedef T value_type;

        void operator= (const T&);
        void operator= (const valarray<T>&) const;
        void operator*= (const valarray<T>&) const;
        void operator/= (const valarray<T>&) const;
        void operator%= (const valarray<T>&) const;
        void operator+= (const valarray<T>&) const;
        void operator-= (const valarray<T>&) const;
        void operator^= (const valarray<T>&) const;
        void operator&= (const valarray<T>&) const;
        void operator|= (const valarray<T>&) const;
        void operator<=(const valarray<T>&) const;
        void operator>=(const valarray<T>&) const;
        ~indirect_array();
    private:
        indirect_array();
        indirect_array(const indirect_array<T>&);
        indirect_array& operator=(const indirect_array<T>&);
        ...
    };
}

```

As usual, class `indirect_array<>` is intended purely as an internal helper class and should be transparent to the user. Thus, all constructors and the assignment operator of `indirect_array<>` are private.

S.3 Allocators and Memory Functions in Detail

As introduced in Section 4.6, page 57, and Chapter 19, allocators represent a special memory model and are an abstraction used to translate the *need* to use memory into a raw *call* for memory. This section describes the details of using allocators and memory management, including:

- Scoped allocators
- The allocator interface and the default allocator
- Providing an allocator before C++11
- Utilities for uninitialized memory

S.3.1 Scoped Allocators

With C++98, the concept of allocators had a couple of constraints:

- Container objects were not required to hold their allocator as members. Allocators were only a part of the type. For this reason:
 - Allocators having the same type were assumed to be equal so that memory allocated by one allocator could be deallocated by another allocator of the same type.
 - It was not possible to change the memory resource at runtime.
 - Allocators had limitations in the ability to hold a state, such as bookkeeping information or information about where to allocate next.
 - Allocators were not swapped when the containers were swapped.
- Allocators were not handled consistently, because copy constructors copied allocators, whereas the assignment operator did not. Thus, the copy constructor did not have the same effect as calling the default constructor and the assignment operator afterward.

For example:

```
std::list<int> list1; // equivalent to:
                    // std::list<int,std::allocator<int>> list1;
std::list<int,MyAlloc<int>> list2;

list1 == list2;      // ERROR: different types
list1 = list2;      // ERROR: different types
```

This even applied to strings, being a kind of STL container:

```
typedef std::basic_string<char, std::char_traits<char>,
                        MyAlloc<char>> MyString;
std::list<std::string> list3;
std::list<MyString,MyAlloc<MyString>> list4;
...
list4.push_back(list3.front()); // ERROR: different element types
std::equal_range(list3.begin(),list3.end(),
                 list4.begin(),list4.end()); // ERROR: different element types
```


A couple of proposed changes to C++11 are intended to solve all these issues. For example:

- Library components are no longer allowed to assume that all allocators of a specific type compare equal.
- *Allocator traits* solve some of the problems created by stateful allocators.
- A *scoped allocator* adapter provides the ability used to propagate the allocator from the container to the contained elements.

These enhancements provide the *tools* to build a polymorphic allocator, which allows to compare or assign two containers with different allocator types that are derived from a common allocator type the container uses. However, due to time constraints, a standard polymorphic allocator is not part of C++11.⁴

Thus, the purpose of class `scoped_allocator_adaptor` is to allow propagating the allocator from the container to the contained elements. Note, however, that to provide backward compatibility, C++11 still provides the old allocator model by default. Internally, some allocator and allocator traits members allow switching to the new model, which class `scoped_allocator_adaptor<>` uses to provide a convenient interface for this feature.

To consistently use an allocator for both the container and its elements you can program the following:

```
#include <scoped_allocator>

// use standard allocator for container and all elements:
std::list<int, std::scoped_allocator_adaptor<std::allocator<int>>> list1;

// use allocator MyAlloc<> for container and all elements:
std::list<int, std::scoped_allocator_adaptor<MyAlloc<int>>> list2;
```

You can use an alias templates (see Section 3.1.9, page 27) instead. For example:

```
#include <scoped_allocator>

template <typename T, typename Alloc>
using MyList = std::list<T, std::scoped_allocator_adaptor<Alloc<T>>>;

MyList<int, MyAlloc<int>>, list2;
```

To use a different allocator for the elements and for the container, you have to pass the element allocator as an additional template argument. For example:

```
std::list<int, std::scoped_allocator_adaptor<MyAlloc1<int>,
                                           MyAlloc2<int>>> list;
```

In the same way you can pass even more allocator types to specify the allocator of the element of the elements and so on.

⁴ Thanks to Pablo Halpern for providing the details for this paragraph.

S.3.2 A User-Defined Allocator for C++98

As introduced in Section 19.2, page 1024, it is pretty easy to use your own allocator by providing the following:

- A type definition of the `value_type`, which is simply the passed template parameter type.
- A template constructor, which copies the internal state while changing the type.
- A member `allocate()`, which provides new memory.
- A member `deallocate()`, which releases memory that is no longer needed.
- Constructors and a destructor, if necessary, to initialize, copy, and clean up the internal state.
- Operators `==` and `!=`.

However, before C++11, a lot of default values for user-defined allocators were not provided. As a result, you had to provide additional members, which were more or less trivial. In correspondence to the allocator provided at Section 19.2, page 1024, let's look at an allocator that fulfills the requirements for a C++98 allocator:

```
// alloc/myalloc03.hpp

#include <cstddef>
#include <memory>
#include <limits>

template <typename T>
class MyAlloc {
public:
    // type definitions
    typedef std::size_t      size_type;
    typedef std::ptrdiff_t   difference_type;
    typedef T*               pointer;
    typedef const T*         const_pointer;
    typedef T&               reference;
    typedef const T&         const_reference;
    typedef T                value_type;

    // constructors and destructor
    // - nothing to do because the allocator has no state
    MyAlloc() throw() {}
    MyAlloc(const MyAlloc&) throw() {}
    template <typename U>
    MyAlloc (const MyAlloc<U>&) throw() {}
    ~MyAlloc() throw() {}
}
```

```

// allocate but don't initialize num elements of type T
T* allocate (std::size_t num, const void* hint = 0) {
    // allocate memory with global new
    return static_cast<T*> (::operator new(num*sizeof(T)));
}

// deallocate storage p of deleted elements
void deallocate (T* p, std::size_t num) {
    // deallocate memory with global delete
    ::operator delete(p);
}

// return address of values
T* address (T& value) const {
    return &value;
}
const T* address (const T& value) const {
    return &value;
}

// return maximum number of elements that can be allocated
std::size_t max_size () const throw() {
    //for numeric_limits see Section 5.3, page 115
    return std::numeric_limits<std::size_t>::max() / sizeof(T);
}

// initialize elements of allocated storage p with value value
void construct (T* p, const T& value) {
    // initialize memory with placement new
    ::new((void*)p)T(value);
}

// destroy elements of initialized storage p
void destroy (T* p) {
    // destroy objects by calling their destructor
    p->~T();
}

// rebind allocator to type U
template <typename U>
struct rebind {
    typedef MyAlloc<U> other;
};
};

```

```

// return that all specializations of this allocator are interchangeable
template <typename T1, typename T2>
bool operator== (const MyAlloc<T1>&,
                 const MyAlloc<T2>&) throw() {
    return true;
}
template <typename T1, typename T2>
bool operator!= (const MyAlloc<T1>&,
                 const MyAlloc<T2>&) throw() {
    return false;
}

```

As you can see, you also have to provide a couple of additional types, `address()` members, `max_size()`, `construct()`, `destroy()`, and the `rebind` structure. The details of these members are described in Section S.3.4, page 1155.

The `rebind<>` Member Template

Part of the allocator structure is a member template called `rebind<>`. This template structure provides the ability for any allocator to allocate storage of another type indirectly. For example, if `Allocator` is an allocator type, then

```
Allocator::rebind<T2>::other
```

is the type of the same allocator specialized for elements of type `T2`.

You can use `rebind<>` if you implement a container and have to allocate memory for a type that differs from the element's type. For example, to implement a deque, you typically need memory for arrays that manage blocks of elements (see the typical implementation of a deque in Section 7.4, page 283). Thus, you need an allocator to allocate arrays of pointers to elements:

```

namespace std {
    template <typename T,
              typename Allocator = allocator<T> >
    class deque {
        ...
    private:
        // rebind allocator for type T*
        typedef typename allocator_traits<Allocator>::rebind_alloc
            PtrAllocator;
        Allocator    alloc;           // allocator for values of type T
        PtrAllocator block_alloc;     // allocator for values of type T*
        T**          elems;          // array of blocks of elements
        ...
    };
}

```

To manage the elements of a deque, you must have one allocator to handle arrays/blocks of elements and another allocator to handle the array of element blocks. The latter has type `PtrAllocator`, which is the same allocator as for the elements. By using `rebind<>`, the allocator for the elements (`Allocator`) is bound to the type of an array of elements (`T*`).

Before C++11, `PtrAllocator` would be defined as follows:

```
typedef typename Allocator:: template rebind<T*>::other
PtrAllocator;
```

S.3.3 The Default Allocator

The default allocator, which is used if no specific allocator is passed, is declared as follows:⁵

```
namespace std {
    template <typename T>
    class allocator {
    public:
        // type definitions
        typedef size_t      size_type;
        typedef ptrdiff_t   difference_type;
        typedef T*          pointer;
        typedef const T*    const_pointer;
        typedef T&          reference;
        typedef const T&    const_reference;
        typedef T            value_type;

        // rebind allocator to type U
        template <typename U>
        struct rebind {
            typedef allocator<U> other;
        };

        // return address of values
        pointer      address(reference value) const noexcept;
        const_pointer address(const_reference value) const noexcept;

        // constructors and destructor
        allocator() noexcept;
        allocator(const allocator&) noexcept;
```

⁵ Before C++11, `construct()` and `destroy()` were not member templates, and `construct()` had a value of type `const T&` as second argument.

```

template <typename U>
    allocator(const allocator<U>&) noexcept;
~allocator();

// return maximum number of elements that can be allocated
size_type max_size() const noexcept;

// allocate but don't initialize num elements of type T
pointer allocate(size_type num,
                 allocator<void>::const_pointer hint = 0);

// initialize elements of allocated storage p with values args
template <typename U, typename... Args>
    void construct(U* p, Args&&... args);

// delete elements of initialized storage p
template <typename U>
    void destroy(U* p);

// deallocate storage p of deleted elements
void deallocate(pointer p, size_type num);
};
}

```

The default allocator uses the global operators `new` and `delete` to allocate and deallocate memory. Thus, `allocate()` may throw a `bad_alloc` exception. However, the default allocator may be optimized by reusing deallocated memory or by allocating more memory than needed to save time in additional allocations. So, the exact moments when operators `new` and `delete` are called are unspecified. See Section 19.2, page 1024, for a possible implementation of the default allocator.

The default allocator has the following specialization for type `void`:

```

namespace std {
    template <>
    class allocator<void> {
    public:
        typedef void*      pointer;
        typedef const void* const_pointer;
        typedef void       value_type;
        template <typename U>
        struct rebind {
            typedef allocator<U> other;
        };
    };
}

```

S.3.4 Allocators in Detail

According to the specified requirements, allocators have to provide the following types and operations. Allocators that are not provided for the standard containers may have fewer requirements.

Type Definitions

allocator::value_type

- The type of the elements.
- It is usually equivalent to `T` for an `allocator<T>`.

allocator::size_type

- The type for unsigned integral values that can represent the size of the largest object in the allocation model.
- Default: `std::make_unsigned<allocator::difference_type>::type`, which is usually equivalent to `std::size_t`.

allocator::difference_type

- The type for signed integral values that can represent the difference between any two pointers in the allocation model.
- Default: `std::pointer_traits<allocator::pointer>::difference_type`, which is usually equivalent to `std::ptrdiff_t`.

allocator::pointer

- The type of a pointer to the element type.
- Default: `value_type*`.

allocator::const_pointer

- The type of a constant pointer to the element type.
- Default: `const value_type*`.

allocator::void_pointer

- The type of a pointer to type `void`.
- Default: `void*`.
- Provided since C++11.

allocator::const_void_pointer

- The type of a constant pointer to type `void`.
- Default: `const void*`.
- Provided since C++11.

***allocator* : :reference**

- The type of a reference to the element type.
- It is usually equivalent to `T&` for an `allocator<T>`.
- No longer required for allocators since C++11.

***allocator* : :const_reference**

- The type of a constant reference to the element type.
- It is usually equivalent to `const T&` for an `allocator<T>`.
- No longer required for allocators since C++11.

***allocator* : :rebind**

- A template structure that provides the ability for any allocator to allocate storage of another type indirectly.
- It has to be declared as follows (which is the default since C++11):

```
template <typename T>
class allocator {
public:
    template <typename U>
    struct rebind {
        typedef allocator<U> other;
    };
    ...
}
```

- See Section S.3.2, page 1152, for an explanation of the purpose of `rebind<>`.

***allocator* : :propagate_on_container_copy_assignment**

- Type trait to signal whether an allocator should be copied when the container is copy assigned.
- Default: `false_type`.
- Provided since C++11.

***allocator* : :propagate_on_container_move_assignment**

- Type trait to signal whether an allocator should be moved when the container is move assigned.
- Default: `false_type`.
- Provided since C++11.

***allocator* : :propagate_on_container_swap**

- Type trait to signal whether an allocator should be swapped when the container is swapped.
- Default: `false_type`.
- Provided since C++11.

Operations

allocator*::*allocator ()

- The default constructor.
- Creates an allocator object.
- No longer required since C++11 (however, at least one non-copy/non-converting constructor must exist).

allocator*::*allocator (const *allocator*& a)

- The copy constructor.
- Copies an allocator object so that storage allocated from the original and from the copy can be deallocated via the other.

allocator*::*allocator (*allocator*&& a)

- The move constructor.
- If it is not provided, the copy constructor is used.
- Moves an allocator object so that storage allocated from a can be deallocated via **this*.
- Provided since C++11.

allocator*::~*allocator ()

- The destructor.
- Destroys an allocator object.

pointer ***allocator*::*address*** (reference *value*)

const_pointer ***allocator*::*address*** (const_reference *value*)

- The first form returns a nonconstant pointer to the nonconstant *value*.
- The second form returns a constant pointer to the constant *value*.
- No longer required since C++11.

size_type ***allocator*::*max_size*** ()

- Returns the largest value that can be passed meaningfully to *allocate*() to allocate storage.
- Default: `std::numeric_limits<allocator::size_type>::max()`.

pointer ***allocator*::*allocate*** (size_type *num*)

pointer ***allocator*::*allocate*** (size_type *num*,
allocator<void>::const_pointer *hint*)

- Both forms return storage for *num* elements of type T.
- The elements are not constructed/initialized (no constructors are called).
- The optional second argument has an implementation-specific meaning. For example, it may be used by an implementation to help improve performance.
- If *num* equals 0, the return value is unspecified.

void **allocator::deallocate** (pointer *p*, size_type *num*)

- Frees the storage to which *p* refers.
- The storage of *p* has to be allocated by `allocate()` of the same or an equal allocator.
- The elements must have been destroyed already.

void **allocator::construct** (U* *p*, Args&&... *args*)

- Initializes the storage of one element to which *p* refers with *args*.
- Default: `::new((void*)p) U(std::forward<Args>(args)...) .`
- Before C++11, the second argument was a `const T&`.

void **allocator::destroy** (U* *p*)

- Destroys the object to which *p* refers without deallocating the storage.
- Calls the destructor for the object.
- Default: `p->~U()`.

bool **operator ==** (const *allocator& a1*, const *allocator& a2*)

- Returns `true` if allocators *a1* and *a2* are interchangeable.
- Two allocators are interchangeable if storage allocated from each can be deallocated via the other.
- Before C++11, to be usable by the standard containers, allocators of the same type were required to be interchangeable, so that this function always had to return `true`.

bool **operator !=** (const *allocator& a1*, const *allocator& a2*)

- Returns `true` if two allocators are not interchangeable.
- It is equivalent to `!(a1 == a2)`.
- Before C++11, to be usable by the standard containers, allocators of the same type were required to be interchangeable, so that this function always had to return `false`.

allocator **select_on_container_copy_construction** ()

- Returns the allocator to be used to copy the allocator on container copy construction.
- Default: returns `*this`.
- Provided since C++11.

S.3.5 Utilities for Uninitialized Memory in Detail

This section describes the auxiliary functions for uninitialized memory in detail. The exemplary exception-safe implementation of these functions is based with permission on code by Greg Colvin.

```
void uninitialized_fill (ForwardIterator beg, ForwardIterator end,
                        const T& value)
```

- Initializes the elements in the range $[beg, end)$ with *value*.
- This function either succeeds or has no effect.
- This function usually is implemented as follows:

```
namespace std {
    template <typename ForwIter, typename T>
    void uninitialized_fill(ForwIter beg, ForwIter end,
                          const T& value)
    {
        typedef typename iterator_traits<ForwIter>::value_type VT;
        ForwIter save(beg);
        try {
            for (; beg!=end; ++beg) {
                ::new (static_cast<void*>(&*beg))VT(value);
            }
        }
        catch (...) {
            for (; save!=beg; ++save) {
                save->~VT();
            }
            throw;
        }
    }
}
```

```
ForwardIterator uninitialized_fill_n (ForwardIterator beg,
                                       Size num, const T& value)
```

- Initializes *num* elements starting from *beg* with *value*.
- Returns the position after the last initialized element.
- This function either succeeds or has no effect.
- This function usually is implemented as follows:

```
namespace std {
    template <typename ForwIter, typename Size, typename T>
    ForwIter uninitialized_fill_n (ForwIter beg, Size num,
                                  const T& value)
    {
```

```

typedef typename iterator_traits<ForwIter>::value_type VT;
ForwIter save(beg);
try {
    for ( ; num--; ++beg) {
        ::new (static_cast<void*>(&*beg))VT(value);
    }
return beg;
}
catch (...) {
    for ( ; save!=beg; ++save) {
        save->~VT();
    }
    throw;
}
}
}

```

- See Section 19.3, page 1028, for an example of the use of `uninitialized_fill_n()`.
- Before C++11, the return type was `void`.

ForwardIterator **uninitialized_copy** (InputIterator *sourceBeg*,
InputIterator *sourceEnd*,
ForwardIterator *destBeg*)

- Initializes the memory starting at *destBeg* with the elements in the range $[sourceBeg, sourceEnd)$.
- Returns the position after the last initialized element.
- The function either succeeds or has no effect.
- The function usually is implemented as follows:

```

namespace std {
    template <typename InputIter, typename ForwIter>
    ForwIter uninitialized_copy(InputIter pos, InputIter end,
                              ForwIter dest)
    {
        typedef typename iterator_traits<ForwIter>::value_type VT;
        ForwIter save(dest);
        try {
            for ( ; pos!=end; ++pos, ++dest) {
                ::new (static_cast<void*>(&*dest))VT(*pos);
            }
            return dest;
        }
        catch (...) {
            for ( ; save!=dest; ++save) {
                save->~VT();
            }
        }
    }
}

```

```

        }
        throw;
    }
}

```

- See Section 19.3, page 1028, for an example of the use of `uninitialized_copy()`.

ForwardIterator **uninitialized_copy_n** (InputIterator *sourceBeg*,
Size *num*, ForwardIterator *destBeg*)

- Initializes *num* elements of the memory starting at *destBeg* with the elements starting with *sourceBeg*.
- Returns the position after the last initialized element.
- The function either succeeds or has no effect.
- The function usually is implemented as follows:

```

namespace std {
    template <typename ForwIter, typename Size, typename ForwIter>
    ForwIter uninitialized_copy_n(InputIter pos, Size num,
                                ForwIter dest)
    {
        typedef typename iterator_traits<ForwIter>::value_type VT;
        ForwIter save(dest);
        try {
            for ( ; num>0; ++pos,++dest,--num) {
                ::new (static_cast<void*>(&*dest))VT(*pos);
            }
            return dest;
        }
        catch (...) {
            for ( ; save!=dest; ++save) {
                save->~VT();
            }
            throw;
        }
    }
}

```

- Provided since C++11.