- Table of Contents
- Previous Chapter
- Next Chapter

Chapter 18: The Standard Template Library

The Standard Template Library (STL) is a general purpose library consisting of containers, generic algorithms, iterators, function objects, allocators, adaptors and data structures. The data structures used by the algorithms are *abstract* in the sense that the algorithms can be used with (practically) any data type.

The algorithms can process these abstract data types because they are *template* based. This chapter does not cover template *construction* (see chapter 21 for that). Rather, it focuses on the *use* of the algorithms.

Several elements also used by the standard template library have already been discussed in the **C++** Annotations. In chapter 12 abstract containers were discussed, and in section 11.10 function objects were introduced. Also, *iterators* were mentioned at several places in this document.

The main components of the STL are covered in this and the next chapter. Iterators, adaptors, smart pointers, multi threading and other features of the STL are discussed in coming sections. Generic algorithms are covered in the next chapter (19).

Allocators take care of the memory allocation within the STL. The default allocator class suffices for most applications, and is not further discussed in the **C++** Annotations.

All elements of the STL are defined in the standard namespace. Therefore, a `using namespace std` or a comparable directive is required unless it is preferred to specify the required namespace explicitly. In header files the `std` namespace should explicitly be used (cf. section 7.11.1).

In this chapter the empty angle bracket notation is frequently used. In code a typename must be supplied between the angle brackets. E.g., `plus<>` is used in the **C++** Annotations, but in code `plus<string>` may be encountered.

18.1: Predefined function objects

Before using the predefined function objects presented in this section the `<functional>` header file must be included.

Function objects play important roles in generic algorithms. For example, there exists a generic algorithm `sort` expecting two iterators defining the range of objects that should be sorted, as well as a function object calling the appropriate comparison operator for two objects. Let's take a quick look at this situation. Assume strings are stored in a vector, and we want to sort the vector in descending order. In that case, sorting the vector `stringVec` is as simple as:

```
    sort(stringVec.begin(), stringVec.end(), greater<string>());
```

The last argument is recognized as a *constructor*: it is an *instantiation* of the `greater<>` class template, applied to strings. This object is called as a function object by the `sort` generic algorithm. The generic algorithm calls the function object's `operator()` member to compare two string objects. The function object's `operator()` will, in turn, call `operator>` of the string data type. Eventually, when `sort` returns, the first element of the vector will contain the string having the greatest string value of all.

The function object's `operator()` itself is *not* visible at this point. Don't confuse the parentheses in the `` `greater<string>()' `` argument with calling `operator()`. When `operator()` is actually used inside `sort`, it receives two arguments: two strings to compare for `greaterness'. Since `greater<string>::operator()` is defined inline, the call itself is not actually present in the above `sort` call. Instead `sort` calls `string::operator>` through `greater<string>::operator()`.

Now that we know that a constructor is passed as argument to (many) generic algorithms, we can design our own function objects. Assume we want to sort our vector case-insensitively. How do we proceed? First we note that the default `string::operator<` (for an incremental sort) is not appropriate, as it does case sensitive comparisons. So, we provide our own `CaseInsensitive` class, which compares two strings case insensitively. Using the `POSIX` function `strcasecmp`, the following program performs the trick. It case-insensitively sorts its command-line arguments in ascending alphabetic order:

```
  #include <iostream>
  #include <string>
  #include <cstring>
  #include <algorithm>
  using namespace std;

  class CaseInsensitive
  {
      public:
          bool operator()(string const &left, string const &right) const
          {
              return strcasecmp(left.c_str(), right.c_str()) < 0;
          }
  };
  int main(int argc, char **argv)
  {
      sort(argv, argv + argc, CaseInsensitive{});
      for (int idx = 0; idx < argc; ++idx)
          cout << argv[idx] << " ";
      cout << '\n';
  }
```

The default constructor of the class CaseInsensitive is used to provide sort with its final argument. So the only member function that must be defined is CaseInsensitive::operator(). Since we know it's called with string arguments, we define it to expect two string arguments, which are used when calling strcasecmp. Furthermore, the function call operator operator() is defined inline, so that it does not produce overhead when called by the sort function. The sort function calls the function object with various combinations of strings. If the compiler grants our inline requests, it will in fact call strcasecmp, skipping two extra function calls.

The comparison function object is often a *predefined function object*. Predefined function object classes are available for many commonly used operations. In the following sections the available predefined function objects are presented, together with some examples showing their use. Near the end of the section about function objects *function adaptors* are introduced.

Predefined function objects are used predominantly with generic algorithms. Predefined function objects exists for arithmetic, relational, and logical operations.

18.1.1: Arithmetic function objects

The arithmetic function objects support the standard arithmetic operations: addition, subtraction, multiplication, division, modulo and negation. These function objects invoke the corresponding operators of the data types for which they are instantiated. For example, for addition the function object plus<Type> is available. If we replace Type by size_t then the addition operator for size_t values is used, if we replace Type by string, the addition operator for strings is used. For example:

```
#include <iostream>
#include <string>
#include <functional>
using namespace std;

int main(int argc, char **argv)
{
    plus<size_t> uAdd;              // function object to add size_ts

    cout << "3 + 5 = " << uAdd(3, 5) << '\n';

    plus<string> sAdd;              // function object to add strings

    cout << "argv[0] + argv[1] = " << sAdd(argv[0], argv[1]) << '\n';
}
/*
    Output when called as: a.out going

    3 + 5 = 8
    argv[0] + argv[1] = a.outgoing
*/
```

Why is this useful? Note that the function object can be used with all kinds of data types (not only with the predefined datatypes) supporting the operator called by the function object.

Suppose we want to perform an operation on a left hand side operand which is always the same variable and a right hand side argument for which, in turn, all elements of an array should be used. E.g., we want to compute the sum of all elements in an array; or we want to concatenate all the strings in a text-array. In situations like these function objects come in handy.

As stated, function objects are heavily used in the context of the generic algorithms, so let's take a quick look ahead at yet another one.

The generic algorithm `accumulate` visits all elements specified by an iterator-range, and performs a requested binary operation on a common element and each of the elements in the range, returning the accumulated result after visiting all elements specified by the iterator range. It's easy to use this algorithm. The next program accumulates all command line arguments and prints the final string:

```
#include <iostream>
#include <string>
#include <functional>
#include <numeric>
using namespace std;

int main(int argc, char **argv)
{
    string result =
            accumulate(argv, argv + argc, string(), plus<string>());

    cout << "All concatenated arguments: " << result << '\n';
}
```

The first two arguments define the (iterator) range of elements to visit, the third argument is `string`. This anonymous string object provides an initial value. We could also have used

```
string("All concatenated arguments: ")
```

in which case the `cout` statement could simply have been `cout << result << '\n'`. The string-addition operation is used, called from `plus<string>`. The final concatenated string is returned.

Now we define a class `Time`, overloading `operator+`. Again, we can apply the predefined function object `plus`, now tailored to our newly defined datatype, to add times:

```
#include <iostream>
#include <string>
#include <vector>
#include <functional>
#include <numeric>
using namespace std;

class Time
{
    friend ostream &operator<<(ostream &str, Time const &time);
    size_t d_days;
    size_t d_hours;
    size_t d_minutes;
    size_t d_seconds;
    public:
        Time(size_t hours, size_t minutes, size_t seconds);
        Time &operator+=(Time const &rhs);
};
Time &&operator+(Time const &lhs, Time const &rhs)
{
    Time ret(lhs);
    return std::move(ret += rhs);
}
Time::Time(size_t hours, size_t minutes, size_t seconds)
:
    d_days(0),
    d_hours(hours),
    d_minutes(minutes),
    d_seconds(seconds)
{}
Time &Time::operator+=(Time const &rhs)
{
    d_seconds   += rhs.d_seconds;
    d_minutes   += rhs.d_minutes   + d_seconds / 60;
    d_hours     += rhs.d_hours     + d_minutes / 60;
    d_days      += rhs.d_days      + d_hours   / 24;
    d_seconds   %= 60;
    d_minutes   %= 60;
    d_hours     %= 24;
    return *this;
}
ostream &operator<<(ostream &str, Time const &time)
{
    return cout << time.d_days << " days, " << time.d_hours <<
                                                " hours, " <<
                    time.d_minutes << " minutes and " <<
                    time.d_seconds << " seconds.";
}
int main(int argc, char **argv)
{
    vector<Time> tvector;

    tvector.push_back(Time( 1, 10, 20));
    tvector.push_back(Time(10, 30, 40));
    tvector.push_back(Time(20, 50,  0));
    tvector.push_back(Time(30, 20, 30));

    cout <<
        accumulate
        (
            tvector.begin(), tvector.end(), Time(0, 0, 0), plus<Time>()
        ) <<
        '\n';
}
//  Displays: 2 days, 14 hours, 51 minutes and 30 seconds.
```
The design of the above program is fairly straightforward. Time defines a constructor, it defines an insertion operator and it defines its own operator+, adding two time objects. In main four Time objects are stored in a vector<Time> object. Then, accumulate is used to compute the accumulated time. It returns a Time object, which is inserted into cout.

While this section's first example illustrated using a *named* function object, the last two examples illustrate how *anonymous* objects can be passed to the (`accumulate`) function .

The STL supports the following set of arithmetic function objects. The function call operator (`operator()`) of these function objects calls the matching arithmetic operator for the objects that are passed to the function call operator, returning that arithmetic operator's return value. The arithmetic operator that is actually called is mentioned below:

- `plus<>`: calls the binary `operator+`;
- `minus<>`: calls the binary `operator-`;
- `multiplies<>`: calls the binary `operator*`;
- `divides<>`: calls `operator/`;
- `modulus<>`: calls `operator%`;
- `negate<>`: calls the unary `operator-`. This arithmetic function object is a unary function object as it expects one argument.

In the next example the transform generic algorithm is used to toggle the signs of all elements of an array. Transform expects two iterators, defining the range of objects to be transformed; an iterator defining the begin of the destination range (which may be the same iterator as the first argument); and a function object defining a unary operation for the indicated data type.

```
#include <iostream>
#include <string>
#include <functional>
#include <algorithm>
using namespace std;

int main(int argc, char **argv)
{
   int iArr[] = { 1, -2, 3, -4, 5, -6 };

   transform(iArr, iArr + 6, iArr, negate<int>());

   for (int idx = 0; idx < 6; ++idx)
      cout << iArr[idx] << ", ";
   cout << '\n';
}
// Displays:  -1, 2, -3, 4, -5, 6,
```

18.1.2: Relational function objects

The relational operators are called by the relational function objects. All standard relational operators are supported: ==, !=, >, >=, < and <=.

The STL supports the following set of relational function objects. The function call operator (`operator()`) of these function objects calls the matching relational operator for the objects that are passed to the function call operator, returning that relational operator's return value. The relational operator that is actually called is mentioned below:

- `equal_to<>`: calls `operator==`;
- `not_equal_to<>`: calls `operator!=`;
- `greater<>`: calls `operator>`;
- `greater_equal<>`: calls `operator>=`;
- `less<>`: this object's member `operator()` calls `operator<`;
- `less_equal<>`: calls `operator<=`.

An example using the relational function objects in combination with sort is:

```
#include <iostream>
#include <string>
#include <functional>
#include <algorithm>
using namespace std;

int main(int argc, char **argv)
{
    sort(argv, argv + argc, greater_equal<string>());

    for (int idx = 0; idx < argc; ++idx)
        cout << argv[idx] << " ";
    cout << '\n';

    sort(argv, argv + argc, less<string>());

    for (int idx = 0; idx < argc; ++idx)
        cout << argv[idx] << " ";
    cout << '\n';
}
```

The example illustrates how strings may be sorted alphabetically and reversed alphabetically. By passing greater_equal<string> the strings are sorted in *decreasing* order (the first word will be the 'greatest'), by passing less<string> the strings are sorted in *increasing* order (the first word will be the 'smallest').

Note that `argv` contains `char *` values, and that the relational function object expects a `string`. The promotion from `char const *` to `string` is silently performed.

18.1.3: Logical function objects

The logical operators are called by the logical function objects. The standard logical operators are supported: `and`, `or`, and `not`.

The STL supports the following set of logical function objects. The function call operator (`operator()`) of these function objects calls the matching logical operator for the objects that are passed to the function call operator, returning that logical operator's return value. The logical operator that is actually called is mentioned below:

- `logical_and<>`: calls `operator&&`;
- `logical_or<>`: calls `operator||`;
- `logical_not<>`: calls `operator!`.

An example using `operator!` is provided in the following trivial program, using `transform` to transform the logicalvalues stored in an array:

```
#include <iostream>
#include <string>
#include <functional>
#include <algorithm>
using namespace std;

int main(int argc, char **argv)
{
    bool bArr[] = {true, true, true, false, false, false};
    size_t const bArrSize = sizeof(bArr) / sizeof(bool);

    for (size_t idx = 0; idx < bArrSize; ++idx)
        cout << bArr[idx] << " ";
    cout << '\n';

    transform(bArr, bArr + bArrSize, bArr, logical_not<bool>());

    for (size_t idx = 0; idx < bArrSize; ++idx)
        cout << bArr[idx] << " ";
    cout << '\n';
}
/*
  Displays:

    1 1 1 0 0 0
    0 0 0 1 1 1
*/
```

18.1.4: The `std::not_fn' negator

A *negator* is a function object toggling the truth value of a function that's called from the negator: if the function returns `true` the negator returns `false` and vv.

The standard negator is `std::not_fn`, declared in the `<functional>` header file.

The function `not_fn` expects a (movable) object as its argument, returning the negated value of the return value of its argument's function call operator.

As an example consider a `main` function defining an array of `int` values:

```
int main()
{
    int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
}
```

To count the number of even values `count_if`, using a lambda function can be used:

```
    cout <<
        count_if(arr, arr + size(arr),
            [&](int value)
            {
                return (value & 1 == 0);
            }
        ) << '\n';
```

To count the number of odd values, `not_fn` can be used in the above code like so:

```
    cout <<
        count_if(arr, arr + size(arr),
            not_fn(
                [&](int value)
                {
                    return (value & 1 == 0);
                }
            )
        ) << '\n';
```

Of course, in this simple example the lambda function could also easily have been modified. But if instead of a lambda function an existing class implementing a function object had been used it would have been difficult or impossible to change the behavior of that class. If the class offers moving operations then `not_fn` can be used to negate the values returned by that class's function call operator.

18.2: Iterators

In addition to the conceptual iterator types presented in this section the STL defines several adaptors allowing objects to be passed as iterators. These adaptors are presented in the upcoming sections. Before those adaptors can be used the `<iterator>` header file must be included.

Although standard iterators are candidates for deprecation (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0174r1.html#2.1), this does not mean that they will (soon) be removed from the `std` library. It's probably a suboptimal strategy to `reinvent' your own. Instead, it is advised to continue using the `std::iterator` classes until they have officially been replaced by alternatives.

Iterators are objects acting like pointers. Iterators have the following general characteristics:

- Two iterators may be compared for (in)equality using the `==` and `!=` operators. The *ordering* operators (e.g., `>`, `<`) can usually not be used.
- Given an iterator `iter`, `*iter` represents the object the iterator points to (alternatively, `iter->` can be used to reach the members of the object the iterator points to).
- `++iter` or `iter++` advances the iterator to the next element. The notion of advancing an iterator to the next element is consequently applied: several containers support *reversed_iterator* types, in which the `++iter` operation actually reaches a previous element in a sequence.
- *Pointer arithmetic* may be used with iterators of containers storing their elements consecutively in memory like `vector` and `deque`. For such containers `iter + 2` points to the second element beyond the one to which `iter` points. See also section 18.2.1, covering `std::distance`.
- Merely defining an iterator is comparable to having a 0-pointer. Example:

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

int main()
{
    vector<int>::iterator vi;

    cout << &*vi;       // prints 0
}
```

STL containers usually define members offering iterators (i.e., they define their own type `iterator`). These members are commonly called `begin` and `end` and (for reversed iterators (type `reverse_iterator`)) `rbegin` and `rend`.

Standard practice requires iterator ranges to be *left inclusive*. The notation `[left, right)` indicates that `left` is an iterator pointing to the first element, while `right` is an iterator pointing just *beyond* the last element. The iterator range is *empty* when `left == right`.

The following example shows how all elements of a vector of strings can be inserted into `cout` using its iterator ranges `[begin(), end())`, and `[rbegin(), rend())`. Note that the `for-loops` for both ranges are identical. Furthermore it nicely illustrates how the `auto` keyword can be used to define the type of the loop control variable instead of using a much more verbose variable definition like `vector<string>::iterator` (see also section 3.3.6):

```
#include <iostream>
#include <vector>
#include <string>
```

```
using namespace std;

int main(int argc, char **argv)
{
    vector<string> args(argv, argv + argc);

    for (auto iter = args.begin(); iter != args.end(); ++iter)
        cout << *iter << " ";
    cout << '\n';

    for (auto iter = args.rbegin(); iter != args.rend(); ++iter)
        cout << *iter << " ";
    cout << '\n';
}
```

Furthermore, the STL defines *const_iterator* types that must be used when visiting a series of elements in a constant container. Whereas the elements of the vector in the previous example could have been altered, the elements of the vector in the next example are immutable, and `const_iterators` are required:

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

int main(int argc, char **argv)
{
    vector<string> const args(argv, argv + argc);

    for
    (
        vector<string>::const_iterator iter = args.begin();
            iter != args.end();
                ++iter
    )
        cout << *iter << " ";
    cout << '\n';

    for
    (
        vector<string>::const_reverse_iterator iter = args.rbegin();
            iter != args.rend();
                ++iter
    )
        cout << *iter << " ";
    cout << '\n';
}
```

The examples also illustrate that plain pointers can be used as iterators. The initialization `vector<string> args(argv, argv + argc)` provides the `args` vector with a pair of pointer-based iterators: `argv` points to the first element to initialize `args` with, `argv + argc` points just beyond the last element to be used, `++argv` reaches the next command line argument. This is a general pointer characteristic, which is why they too can be used in situations where iterators are expected.

The STL defines five types of iterators. These iterator types are expected by generic algorithms, and in order to create a particular type of iterator yourself it is important to know their characteristics. In general, iterators (see also section 22.14) must define:

- `operator==`, testing two iterators for equality,
- `operator!=`, testing two iterators for inequality,
- `operator++`, incrementing the iterator, as prefix operator,
- `operator*`, to access the element the iterator refers to,

The following types of iterators are used when describing generic algorithms in chapter 19:

- **InputIterators**:

  InputIterators are used to read from a container. The dereference operator is guaranteed to work as `rvalue` in expressions. Instead of an InputIterator it is also possible to use (see below) Forward-, Bidirectional- or RandomAccessIterators. Notations like `InputIterator1` and `InputIterator2` may be used as well. In these cases, numbers are used to indicate which iterators `belong together'. E.g., the generic algorithm `inner_product` has the following prototype:

  ```
  Type inner_product(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, Type init);
  ```

  `InputIterator1 first1` and `InputIterator1 last1` define a pair of input iterators on one range, while `InputIterator2 first2` defines the beginning of another range. Analogous notations may be used with other iterator types.

- **OutputIterators**:

  OutputIterators can be used to write to a container. The dereference operator is guaranteed to work as an `lvalue` in expressions, but not necessarily as `rvalue`. Instead of an OutputIterator it is also possible to use (see below) Forward-, Bidirectional- or RandomAccessIterators.

- **ForwardIterators**:

  ForwardIterators combine InputIterators and OutputIterators. They can be used to traverse containers in one direction, for reading and/or writing. Instead of a ForwardIterator it is also possible to use (see below) Bidirectional- or RandomAccessIterators.

- **BidirectionalIterators**:

BidirectionalIterators can be used to traverse containers in both directions, for reading and writing. Instead of a BidirectionalIterator it is also possible to use (see below) a RandomAccessIterator.

- **RandomAccessIterators**:

  RandomAccessIterators provide random access to container elements. An algorithm like `sort` requires a RandomAccessIterator, and can therefore *not* be used to sort the elements of lists or maps, which only provide BidirectionalIterators.

The example given with the RandomAccessIterator illustrates how to relate iterators and generic algorithms: look for the iterator that's required by the (generic) algorithm, and then see whether the datastructure supports the required type of iterator. If not, the algorithm cannot be used with the particular datastructure.

18.2.1: std::distance and std::size

Earlier, in section 18.2 it was stated that iterators support pointer arithmetic for containers storing their elements consecutively in memory. This is not completely true: to determine the number of elements between the elements to which two iterators refer the iterator must support the subtraction operator.

Using pointer arithmetic to compute the number of elements between two iterators in, e.g., a `std::list` or `std::unordered_map` is not possible, as these containers do not store their elements consecutively in memory.

The function `std::distance` fills in that little gap: `std::distance` expects two InputIterators and returns the number of elements between them.

Before using `distance` the `<iterator>` header file must be included.

If the iterator specified as first argument exceeds the iterator specified as its second argument then the number of elements is non-positive, otherwise it is non-negative. If the number of elements cannot be determined (e.g., the iterators do not refer to elements in the same container), then `distance's` return value is undefined.

Example:

```
#include <iostream>
#include <unordered_map>

using namespace std;

int main()
{
    unordered_map<int, int> myMap = {{1, 2}, {3, 5}, {-8, 12}};
    cout << distance(++myMap.begin(), myMap.end()) << '\n'; // shows: 2
}
```

The `iterator` header file also defines the function `std::size`, returning the number of elements in a containers (as returned by the container's `size` member) or of an array whose dimension is known to the compiler at the point of `std::size's` call. E.g., if the size of an array `data` is known to the compiler, then to call a function `handler` (expecting the address of the first element of an array and the address of the location just beyond that array) the following statement can be used:

```
handler(data, data + std::size(data));
```

As noted, the std::size function is defined in the iterator header. However, it's also guaranteed available when including the header file of a container supporting iterators (including the string header file).

18.2.2: Insert iterators

Generic algorithms often require a target container into which the results of the algorithm are deposited. For example, the copy generic algorithm has three parameters. The first two define the range of visited elements, the third defines the first position where the results of the copy operation should be stored.

With the `copy` algorithm the number of elements to copy is usually available beforehand, since that number can usually be provided by pointer arithmetic. However, situations exist where pointer arithmetic cannot be used. Analogously, the number of resulting elements sometimes differs from the number of elements in the initial range. The generic algorithm unique_copy is a case in point. Here the number of elements that are copied to the destination container is normally not known beforehand.

In situations like these an *inserter* adaptor function can often be used to create elements in the destination container. There are three types of inserter adaptors:

- `back_inserter`: calls the container's `push_back` member to add new elements at the end of the container. E.g., to copy all elements of `source` in reversed order to the back of `destination`, using the copy generic algorithm:

-
  ```
  copy(source.rbegin(), source.rend(), back_inserter(destination));
  ```

- `front_inserter` calls the container's `push_front` member, adding new elements at the beginning of the container. E.g., to copy all elements of `source` to the front of the destination container (thereby also reversing the order of the elements):

-
  ```
  copy(source.begin(), source.end(), front_inserter(destination));
  ```

- `inserter` calls the container's `insert` member adding new elements starting at a specified starting point. E.g., to copy all elements of `source` to the destination container, starting at the beginning of `destination`, shifting up existing elements to beyond the newly inserted elements:

```
•  copy(source.begin(), source.end(), inserter(destination,
•      destination.begin()));
```

The inserter adaptors require the existence of two typedefs:

- `typedef Data value_type`, where `Data` is the data type stored in the class offering `push_back, push_front` or `insert` members (Example: `typedef std::string value_type`);

- `typedef value_type const &const_reference`

Concentrating on back_inserter, this iterator expects the name of a container supporting a member push_back. The inserter's operator() member calls the container's push_back member. Objects of any class supporting a push_back member can be passed as arguments to back_inserter provided the class adds

```
    typedef DataType const &const_reference;
```

to its interface (where DataType const & is the type of the parameter of the class's member push_back). Example:

```
    #include <iostream>
    #include <algorithm>
    #include <iterator>
    using namespace std;

    class Insertable
    {
        public:
            typedef int value_type;
            typedef int const &const_reference;

            void push_back(int const &)
            {}
    };

    int main()
    {
        int arr[] = {1};
        Insertable insertable;

        copy(arr, arr + 1, back_inserter(insertable));
    }
```

18.2.3: Iterators for `istream' objects

The istream_iterator<Type> can be used to define a set of iterators for istream objects. The general form of the istream_iterator iterator is:

```
    istream_iterator<Type> identifier(istream &in)
```

Here, Type is the type of the data elements read from the istream stream. It is used as the `begin' iterator in an interator range. Type may be any type for which operator>> is defined in combination with istream objects.

The default constructor is used as the end-iterator and corresponds to the end-of-stream. For example,

```
istream_iterator<string> endOfStream;
```

The *stream* object that was specified when defining the begin-iterator is *not* mentioned with the default constructor.

Using `back_inserter` and `istream_iterator` adaptors, all strings from a stream can easily be stored in a container. Example (using anonymous `istream_iterator` adaptors):

```
#include <iostream>
#include <iterator>
#include <string>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
    vector<string> vs;

    copy(istream_iterator<string>(cin), istream_iterator<string>(),
         back_inserter(vs));

    for
    (
        vector<string>::const_iterator begin = vs.begin(), end = vs.end();
            begin != end; ++begin
    )
        cout << *begin << ' ';
    cout << '\n';
}
```

18.2.3.1: Iterators for `istreambuf' objects

Input iterators are also available for streambuf objects.

To read from `streambuf` objects supporting input operations `istreambuf_iterators` can be used, supporting the operations that are also available for `istream_iterator`. Different from the latter iterator type `istreambuf_iterators` support three constructors:

- `istreambuf_iterator<Type>`:

    The end iterator of an iterator range is created using the default `istreambuf_iterator` constructor. It represents the end-of-stream condition when extracting values of type `Type` from the `streambuf`.

- `istreambuf_iterator<Type>(streambuf *)`:

  A pointer to a `streambuf` may be used when defining an `istreambuf_iterator`. It represents the begin iterator of an iterator range.

- `istreambuf_iterator<Type>(istream)`:

  An *istream* may be also used when defining an `istreambuf_iterator`. It accesses the `istream`'s `streambuf` and it also represents the begin iterator of an iterator range.

In section 18.2.4.1 an example is given using both istreambuf_iterators and ostreambuf_iterators.

18.2.4: Iterators for `ostream' objects

An ostream_iterator<Type> adaptor can be used to pass an ostream to algorithms expecting an OutputIterator. Two constructors are available for defining ostream_iterators:

```
    ostream_iterator<Type> identifier(ostream &outStream);
    ostream_iterator<Type> identifier(ostream &outStream, char const *delim);
```

Type is the type of the data elements that should be inserted into an ostream. It may be any type for which operator<< is defined in combination with ostream objects. The latter constructor can be used to separate the individual Type data elements by delimiter strings. The former constructor does not use any delimiters.

The example shows how `istream_iterators` and an `ostream_iterator` may be used to copy information of a file to another file. A subtlety here is that you probably want to use `in.unsetf(ios::skipws)`. It is used to clear the `ios::skipws` flag. As a consequence whitespace characters are simply returned by the operator, and the file is copied character by character. Here is the program:

```
    #include <iostream>
    #include <algorithm>
    #include <iterator>
    using namespace std;

    int main()
    {
        cin.unsetf(ios::skipws);
        copy(istream_iterator<char>(cin), istream_iterator<char>(),
             ostream_iterator<char>(cout));
    }
```

18.2.4.1: Iterators for `ostreambuf' objects

Output iterators are also available for streambuf objects.

To write to `streambuf` objects supporting output operations `ostreambuf_iterators` can be used, supporting the operations that are also available for `ostream_iterator`. `Ostreambuf_iterators` support two constructors:

- `ostreambuf_iterator<Type>(streambuf *)`:

  A pointer to a `streambuf` may be used when defining an `ostreambuf_iterator`. It can be used as an OutputIterator.

- `ostreambuf_iterator<Type>(ostream)`:

  An *ostream* may be also used when defining an `ostreambuf_iterator`. It accesses the `ostream`'s `streambuf` and it can also be used as an OutputIterator.

The next example illustrates the use of both istreambuf_iterators and ostreambuf_iterators when copying a stream in yet another way. Since the stream's streambufs are directly accessed the streams and stream flags are bypassed. Consequently there is no need to clear ios::skipws as in the previous section, while the next program's efficiency probably also exceeds the efficiency of the program shown in the previous section.

```
#include <iostream>
#include <algorithm>
#include <iterator>
using namespace std;

int main()
{
   istreambuf_iterator<char> in(cin.rdbuf());
   istreambuf_iterator<char> eof;
   ostreambuf_iterator<char> out(cout.rdbuf());

   copy(in, eof, out);
}
```

18.3: The class 'unique_ptr'

Before using the unique_ptr class presented in this section the <memory> header file must be included.

When pointers are used to access dynamically allocated memory strict bookkeeping is required to prevent memory leaks. When a pointer variable referring to dynamically allocated memory goes out of scope, the dynamically allocated memory becomes inaccessible and the program suffers from a memory leak. Consequently, the programmer has to make sure that the dynamically allocated memory is returned to the common pool just before the pointer variable goes out of scope.

When a pointer variable points to a dynamically allocated single value or object, bookkeeping requirements are greatly simplified when the pointer variable is defined as a `std::unique_ptr` object.

Unique_ptrs are *objects* masquerading as pointers. Since they are objects, their destructors are called when they go out of scope. Their destructors automatically delete the dynamically allocated memory to which they point. Unique_ptrs (and their cousins shared_ptrs (cf. section 18.4) are also called *smart pointers*).

`Unique_ptrs` have several special characteristics:

- when assigning a `unique_ptr` to another *move semantics* is used. If move semantics is not available compilation fails. On the other hand, if compilation succeeds then the used containers or generic algorithms support the use of `unique_ptrs`. Here is an example:

```
std::unique_ptr<int> up1(new int);
std::unique_ptr<int> up2(up1);      // compilation error
```

  The second definition fails to compile as `unique_ptr`'s copy constructor is private (the same holds true for the assignment operator). But the `unique_ptr` class *does* offer facilities to initialize and assign from *rvalue references*:

```
class unique_ptr                        // interface partially shown
{
    public:
        unique_ptr(unique_ptr &&tmp);   // rvalues bind here
    private:
        unique_ptr(const unique_ptr &other);
};
```

  In the next example move semantics is used and so it compiles correctly:

```
unique_ptr<int> cp(unique_ptr<int>(new int));
```

- a `unique_ptr` object should only point to memory that was made available dynamically, as only dynamically allocated memory can be deleted.

- multiple `unique_ptr` objects should not be allowed to point to the same block of dynamically allocated memory. The `unique_ptr`'s interface was designed to prevent this from happening. Once a `unique_ptr` object goes out of scope, it deletes the memory it points to, immediately changing any other object also pointing to the allocated memory into a wild pointer.

- When a class `Derived` is derived from `Base`, then a newly allocated `Derived` class object can be assigned to a `unique_ptr<Base>`, without having to define a virtual destructor for `Base`. The `Base *` pointer that is returned by the `unique_ptr` object can simply be cast statically to `Derived`, and `Derived's` destructor is automatically called as well, if the unique_ptr definition is provided with a *deleter* function address. This is illustrated in the next example:

```
    class Base
    { ... };
    class Derived: public Base
    {
        ...
        public:
            // assume Derived has a member void process()

            static void deleter(Base *bp);
    };
    void Derived::deleter(Base *bp)
    {
        delete static_cast<Derived *>(bp);
    }
    int main()
    {
        unique_ptr<Base, void (*)(Base *)> bp(new Derived,
&Derived::deleter);
        static_cast<Derived *>(bp.get())->process(); // OK!

    } // here ~Derived is called: no polymorphism required.
```

The class unique_ptr offers several member functions to access the pointer itself or to have a unique_ptr point to another block of memory. These member functions (and unique_ptr constructors) are introduced in the next few sections.

Unique_ptr can also be used with containers and (generic) algorithms. They can properly destruct any type of object, as their constructors accept customizable deleters. In addition, arrays can be handled by unique_ptrs.

18.3.1: Defining `unique_ptr' objects

There are three ways to define unique_ptr objects. Each definition contains the usual <type> specifier between angle brackets:

- The default constructor simply creates a unique_ptr object that does not point to a particular block of memory. Its pointer is initialized to 0 (zero):

```
unique_ptr<type> identifier;
```

  This form is discussed in section 18.3.2.

- The *move constructor* initializes an unique_ptr object. Following the use of the move constructor its unique_ptr argument no longer points to the dynamically allocated memory and its pointer data member is turned into a zero-pointer:

- `unique_ptr<type> identifier(another unique_ptr for type);`

  This form is discussed in section 18.3.3.

- The form that is used most often initializes a `unique_ptr` object to the block of dynamically allocated memory that is passed to the object's constructor. Optionally `deleter` can be provided. A (free) function (or function object) receiving the `unique_ptr`'s pointer as its argument can be passed as deleter. It is supposed to return the dynamically allocated memory to the common pool (doing nothing if the pointer equals zero).

- `unique_ptr<type> identifier (new-expression [, deleter]);`

  This form is discussed in section 18.3.4.

18.3.2: Creating a plain `unique_ptr'

Unique_ptr's default constructor defines a unique_ptr not pointing to a particular block of memory:

```
    unique_ptr<type> identifier;
```

The pointer controlled by the unique_ptr object is initialized to 0 (zero). Although the unique_ptr object itself is not the pointer, its value *can* be compared to 0. Example:

```
    unique_ptr<int> ip;

    if (!ip)
        cout << "0-pointer with a unique_ptr object\n";
```

Alternatively, the member get can be used (cf. section 18.3.5).

18.3.3: Moving another `unique_ptr'

A unique_ptr may be initialized using an rvalue reference to a unique_ptr object for the same type:

```
    unique_ptr<type> identifier(other unique_ptr object);
```

The move constructor is used, e.g., in the following example:

```
    void mover(unique_ptr<string> &&param)
    {
        unique_ptr<string> tmp(move(param));
    }
```

Analogously, the assignment operator can be used. A unique_ptr object may be assigned to a temporary unique_ptr object of the same type (again move-semantics is used). For example:

```
    #include <iostream>
    #include <memory>
    #include <string>

    using namespace std;

    int main()
    {
        unique_ptr<string> hello1(new string("Hello world"));
        unique_ptr<string> hello2(move(hello1));
        unique_ptr<string> hello3;

        hello3 = move(hello2);
        cout << // *hello1 << '\n' <<   // would have segfaulted
                // *hello2 << '\n' <<   // same
                *hello3 << '\n';
    }
    // Displays:    Hello world
```
The example illustrates that

- `hello1` is initialized by a pointer to a dynamically allocated `string` (see the next section).

- The `unique_ptr hello2` grabs the pointer controlled by `hello1` using a move constructor. This effectively changes `hello1` into a 0-pointer.

- Then `hello3` is defined as a default `unique_ptr<string>`. But then it grabs its value using move-assignment from `hello2` (which, as a consequence, is changed into a 0-pointer).

If `hello1` or `hello2` had been inserted into `cout` a *segmentation fault* would have resulted. The reason for this should now be clear: it is caused by dereferencing 0-pointers. In the end, only `hello3` actually points to the originally allocated `string`.

18.3.4: Pointing to a newly allocated object

A `unique_ptr` is most often initialized using a pointer to dynamically allocated memory. The generic form is:
```
    unique_ptr<type [, deleter_type]> identifier(new-expression
            [, deleter = deleter_type{}]);
```
The second (template) argument (`deleter(_type)`) is optional and may refer to a free function or function object handling the destruction of the allocated memory. A deleter is used, e.g., in situations where a double pointer is allocated and the destruction must visit each nested pointer to destroy the allocated memory (see below for an illustration).

Here is an example initializing a `unique_ptr` pointing to a `string` object:

```
    unique_ptr<string> strPtr(new string("Hello world"));
```
The argument that is passed to the constructor is the pointer returned by `operator new`. Note that type does *not* mention the pointer. The type that is used in the `unique_ptr` construction is the same as the type that is used in `new` expressions.

Here is an example showing how an explicitly defined deleter may be used to delete a dynamically allocated array of pointers to strings:

```
    #include <iostream>
    #include <string>
```

```
#include <memory>
using namespace std;

struct Deleter
{
    size_t d_size;
    Deleter(size_t size = 0)
    :
        d_size(size)
    {}
    void operator()(string **ptr) const
    {
        for (size_t idx = 0; idx < d_size; ++idx)
            delete ptr[idx];
        delete[] ptr;
    }
};
int main()
{
    unique_ptr<string *, Deleter> sp2{ new string *[10], Deleter{ 10 } };

    Deleter &obj = sp2.get_deleter();
}
```

A `unique_ptr` can be used to reach the member functions that are available for objects allocated by the `new` expression. These members can be reached as if the `unique_ptr` was a plain pointer to the dynamically allocated object. For example, in the following program the text `` `C++ ``' is inserted behind the word `` `hello ``':

```
#include <iostream>
#include <memory>
#include <cstring>
using namespace std;

int main()
{
    unique_ptr<string> sp(new string("Hello world"));

    cout << *sp << '\n';
    sp->insert(strlen("Hello "), "C++ ");
    cout << *sp << '\n';
}
/*
    Displays:
        Hello world
        Hello C++ world
*/
```

18.3.5: Operators and members

The class unique_ptr offers the following operators:

- `unique_ptr<Type> &operator=(unique_ptr<Type> &&tmp)`:

  This operator transfers the memory pointed to by the rvalue `unique_ptr` object to the lvalue `unique_ptr` object using *move semantics*. So, the rvalue object *loses* the memory it pointed at and turns into a 0-pointer. An existing `unique_ptr` may be assigned to another `unique_ptr` by converting it to an rvalue reference first using `std::move`. Example:

  ```
  unique_ptr<int> ip1(new int);
  unique_ptr<int> ip2;

  ip2 = std::move(ip1);
  ```

- `operator bool() const`:

  This operator returns `false` if the `unique_ptr` does not point to memory (i.e., its `get` member, see below, returns 0). Otherwise, `true` is returned.

- `Type &operator*()`:

  This operator returns a reference to the information accessible via a `unique_ptr` object . It acts like a normal pointer dereference operator.

- `Type *operator->()`:

  This operator returns a pointer to the information accessible via a `unique_ptr` object. This operator allows you to select members of an object accessible via a `unique_ptr` object. Example:

  ```
  unique_ptr<string> sp(new string("hello"));
  cout << sp->c_str();
  ```

The class `unique_ptr` supports the following member functions:

- `Type *get()`:

  A pointer to the information controlled by the `unique_ptr` object is returned. It acts like `operator->`. The returned pointer can be inspected. If it is zero the `unique_ptr` object does not point to any memory.

- `Deleter &unique_ptr<Type>::get_deleter()`:

  A reference to the deleter object used by the `unique_ptr` is returned.

- `Type *release()`:

A pointer to the information accessible via a `unique_ptr` object is returned. At the same time the object itself becomes a 0-pointer (i.e., its pointer data member is turned into a 0-pointer). This member can be used to transfer the information accessible via a `unique_ptr` object to a plain `Type` pointer. After calling this member the proper destruction of the dynamically allocated memory is the responsibility of the programmer.

- `void reset(Type *)`:

  The dynamically allocated memory controlled by the `unique_ptr` object is returned to the common pool; the object thereupon controls the memory to which the argument that is passed to the function points. It can also be called without argument, turning the object into a 0-pointer. This member function can be used to assign a new block of dynamically allocated memory to a `unique_ptr` object.

- `void swap(unique_ptr<Type> &)`:

  Two identically typed `unique_ptrs` are swapped.

18.3.6: Using `unique_ptr' objects for arrays

When a `unique_ptr` is used to store arrays the dereferencing operator makes little sense but with arrays `unique_ptr` objects benefit from index operators. The distinction between a single object `unique_ptr` and a `unique_ptr` referring to a dynamically allocated array of objects is realized through a template specialization.

With dynamically allocated arrays the following syntax is available:

- the index (`[]`) notation is used to specify that the smart pointer controls a dynamically allocated *array*. Example:

```
unique_ptr<int[]> intArr(new int[3]);
```

- the index operator can be used to access the array's elements. Example:

```
intArr[2] = intArr[0];
```

In these cases the smart pointer's destructors call `delete[]` rather than `delete`.

18.4: The class `shared_ptr'

In addition to the class `unique_ptr` the class `std::shared_ptr<Type>` is available, which is a reference counting smart pointer.

Before using `shared_ptrs` the `<memory>` header file must be included.

The shared pointer automatically destroys its contents once its reference count has decayed to zero. As with `unique_ptr`, when defining a `shared_ptr<Base>` to store a newly allocated `Derived` class object, the returned `Base *` may be cast to a `Derived *` using a `static_cast`: polymorphism isn't required, and when resetting the `shared_ptr` or when the `shared_ptr` goes out of scope, no slicing occurs, and `Derived`'s destructor (or, if configured: deleter) is called (cf. section 18.3).

`Shared_ptrs` support copy and move constructors as well as standard and move overloaded assignment operators.

Like `unique_ptrs, shared_ptrs` may refer to dynamically allocated arrays.

18.4.1: Defining `shared_ptr' objects

There are four ways to define `shared_ptr` objects. Each definition contains the usual `<type>` specifier between angle brackets:

- The default constructor simply creates a `shared_ptr` object that does not point to a particular block of memory. Its pointer is initialized to 0 (zero):

```
shared_ptr<type> identifier;
```

  This form is discussed in section 18.4.2.

- The copy constructor initializes a `shared_ptr` so that both objects share the memory pointed at by the existing object. The copy constructor also increments the `shared_ptr`'s reference count. Example:

```
shared_ptr<string> org(new string("hi there"));
shared_ptr<string> copy(org);   // reference count now 2
```

- The move constructor initializes a `shared_ptr` with the pointer and reference count of a temporary `shared_ptr`. The temporary `shared_ptr` is changed into a 0-pointer. An existing `shared_ptr` may have its data moved to a newly defined `shared_ptr` (turning the existing `shared_ptr` into a 0-pointer as well). In the next example a temporary, anonymous `shared_ptr` object is constructed, which is then used to construct `grabber`. Since `grabber`'s constructor receives an anonymous temporary object, the compiler uses `shared_ptr`'s move constructor:

```
shared_ptr<string> grabber(shared_ptr<string>(new string("hi there")));
```

- The form that is used most often initializes a `shared_ptr` object to the block of dynamically allocated memory that is passed to the object's constructor. Optionally `deleter` can be provided. A (free) function (or function object) receiving the `shared_ptr`'s pointer as its argument can be passed as deleter. It is supposed to return the dynamically allocated memory to the common pool (doing nothing if the pointer equals zero).

- `shared_ptr<type> identifier (new-expression [, deleter]);`

  This form is discussed in section 18.4.3.

18.4.2: Creating a plain `shared_ptr'

Shared_ptr's default constructor defines a shared_ptr not pointing to a particular block of memory:

```
    shared_ptr<type> identifier;
```

The pointer controlled by the shared_ptr object is initialized to 0 (zero). Although the shared_ptr object itself is not the pointer, its value *can* be compared to 0. Example:

```
    shared_ptr<int> ip;

    if (!ip)
        cout << "0-pointer with a shared_ptr object\n";
```

Alternatively, the member get can be used (cf. section 18.4.4).

18.4.3: Pointing to a newly allocated object

Most often a shared_ptr is initialized by a dynamically allocated block of memory. The generic form is:

```
    shared_ptr<type> identifier(new-expression [, deleter]);
```

The second argument (deleter) is optional and refers to a function object or free function handling the destruction of the allocated memory. A deleter is used, e.g., in situations where a double pointer is allocated and the destruction must visit each nested pointer to destroy the allocated memory (see below for an illustration). It is used in situations comparable to those encountered with unique_ptr (cf. section 18.3.4).

Here is an example initializing a `shared_ptr` pointing to a `string` object:

```
    shared_ptr<string> strPtr(new string("Hello world"));
```

The argument that is passed to the constructor is the pointer returned by operator new. Note that type does *not* mention the pointer. The type that is used in the shared_ptr construction is the same as the type that is used in new expressions.

The next example illustrates that two `shared_ptrs` indeed share their information. After modifying the information controlled by one of the objects the information controlled by the other object is modified as well:

```
    #include <iostream>
    #include <memory>
    #include <cstring>
    using namespace std;

    int main()
```

```
{
   shared_ptr<string> sp(new string("Hello world"));
   shared_ptr<string> sp2(sp);

   sp->insert(strlen("Hello "), "C++ ");
   cout << *sp << '\n' <<
         *sp2 << '\n';
}
/*
   Displays:
      Hello C++ world
      Hello C++ world
*/
```

18.4.4: Operators and members

The class `shared_ptr` offers the following operators:

- `shared_ptr &operator=(shared_ptr<Type> const &other)`:

  Copy assignment: the reference count of the operator's left hand side operand is reduced. If the reference count decays to zero the dynamically allocated memory controlled by the left hand side operand is deleted. Then it shares the information with the operator's right hand side operand, incrementing the information's reference count.

- `shared_ptr &operator=(shared_ptr<Type> &&tmp)`:

  Move assignment: the reference count of the operator's left hand side operand is reduced. If the reference count decays to zero the dynamically allocated memory controlled by the left hand side operand is deleted. Then it grabs the information controlled by the operator's right hand side operand which is turned into a 0-pointer.

- `operator bool() const`:

  If the `shared_ptr` actually points to memory `true` is returned, otherwise, `false` is returned.

- `Type &operator*()`:

  A reference to the information stored in the `shared_ptr` object is returned. It acts like a normal pointer.

- `Type *operator->()`:

  A pointer to the information controlled by the `shared_ptr` object is returned. Example:

```
shared_ptr<string> sp(new string("hello"));
cout << sp->c_str() << '\n';
```

The following member function member functions are supported:

- `Type *get()`:

  A pointer to the information controlled by the `shared_ptr` object is returned. It acts like `operator->`. The returned pointer can be inspected. If it is zero the `shared_ptr` object does not point to any memory.

- `Deleter &get_deleter()`:

  A reference to the `shared_ptr`'s deleter (function or function object) is returned.

- `void reset(Type *)`:

  The reference count of the information controlled by the `shared_ptr` object is reduced and if it decays to zero the memory it points to is deleted. Thereafter the object's information will refer to the argument that is passed to the function, setting its shared count to 1. It can also be called without argument, turning the object into a 0-pointer. This member function can be used to assign a new block of dynamically allocated memory to a `shared_ptr` object.

- `void reset(Type *, DeleterType &&)`:

  This variant of the previous member accepts a specific `Deleter` type: if `Type` is a base-class and derived class objects are used, these derived class objects may require specific actions at destruction time. When the previous member is used, then eventually the newly assigned object's destructor is called without using an explicit deleter function. The current member ensures that by the time the shared counter has decayed to zero the provided deleter is used.

- `void shared_ptr<Type>::swap(shared_ptr<Type> &&)`:

  Two identically typed `shared_ptrs` are swapped.

- `bool unique() const`:

  If the current object is the only object referring to the memory controlled by the object `true` is returned otherwise (including the situation where the object is a 0-pointer) `false` is returned.

- `size_t use_count() const`:

  The number of objects sharing the memory controlled by the object is returned.

18.4.5: Casting shared pointers

Be cautious when using standard **C++** style casts in combination with shared_ptr objects. Consider the following two classes:

```
struct Base
{};
struct Derived: public Base
{};
```

As with unique_ptr, when defining a shared_ptr<Base> to store a newly allocated Derived class object, the returned Base * may be cast to a Derived * using a static_cast: polymorphism isn't required, and when resetting the shared_ptr or when the shared_ptr goes out of scope, no slicing occurs, and Derived's destructor is called (cf. section 18.3).

Of course, a shared_ptr<Derived> can easily be defined. Since a Derived object is also a Base object, a pointer to Derived can be considered a pointer to Base without using casts, but a static_cast could be used to force the interpretation of a Derived * to a Base *:

```
Derived d;
static_cast<Base *>(&d);
```

However, a plain static_cast cannot be used when initializing a shared pointer to a Base using the get member of a shared pointer to a Derived object. The following code snipped eventually results in an attempt to delete the dynamically allocated Base object twice:

```
shared_ptr<Derived> sd{ new Derived };
shared_ptr<Base> sb{ static_cast<Base *>(sd.get()) };
```

Since sd and sb point at the same object ~Base will be called for the same object when sb goes out of scope and when sd goes out of scope, resulting in premature termination of the program due to a *double free* error.

These errors can be prevented using casts that were specifically designed for being used with shared_ptrs. These casts use specialized constructors that create a shared_ptr pointing to memory but shares ownership (i.e., a reference count) with an existing shared_ptr. These special casts are:

- std::static_pointer_cast<Base>(std::shared_ptr<Derived> ptr):

  A shared_ptr to a Base class object is returned. The returned shared_ptr refers to the base class portion of the Derived class to which the shared_ptr<Derived> ptr refers. Example:

  ```
  shared_ptr<Derived> dp{ new Derived };
  shared_ptr<Base> bp = static_pointer_cast<Base>(dp);
  ```

- std::const_pointer_cast<Class>(std::shared_ptr<Class const> ptr):

A shared_ptr to a Class class object is returned. The returned shared_ptr refers to a non-const Class object whereas the ptr argument refers to a Class const object. Example:

```
shared_ptr<Derived const> cp{ new Derived };
shared_ptr<Derived> ncp = const_pointer_cast<Derived>(cp);
```

- std::dynamic_pointer_cast<Derived>(std::shared_ptr<Base> ptr):

  A shared_ptr to a Derived class object is returned. The Base class must have at least one virtual member function, and the class Derived, inheriting from Base may have overridden Base's virtual member(s). The returned shared_ptr refers to a Derived class object if the dynamic cast from Base * to Derived * succeeded. If the dynamic cast did not succeed the shared_ptr's get member returns 0. Example (assume Derived and Derived2 were derived from Base):

```
shared_ptr<Base> bp(new Derived());
cout << dynamic_pointer_cast<Derived>(bp).get() << ' ' <<
        dynamic_pointer_cast<Derived2>(bp).get() << '\n';
```

  The first get returns a non-0 pointer value, the second get returns 0.

18.4.6: Using `shared_ptr' objects for arrays

Different from the unique_ptr class no specialization exists for the shared_ptr class to handle dynamically allocated arrays of objects.

But like unique_ptrs, with shared_ptrs referring to arrays the dereferencing operator makes little sense while in these circumstances shared_ptr objects would benefit from index operators.

It is not difficult to create a class shared_array offering such facilities. The class template shared_array, derived from shared_ptr merely should provide an appropriate *deleter* to make sure that the array and its elements are properly destroyed. In addition it should define the index operator and, when applicable should declare the derefencing operators using delete.

Here is an example showing how shared_array can be defined and used:

```
struct X
{
    ~X()
    {
        cout << "destr\n";  // show the object's destruction
    }
};
template <typename Type>
class shared_array: public shared_ptr<Type>
{
    struct Deleter          // Deleter receives the pointer
    {                    // and calls delete[]
      void operator()(Type* ptr)
      {
        delete[] ptr;
      }
    };
    public:
        shared_array(Type *p)          // other constructors
        :                              // not shown here
           shared_ptr<Type>(p, Deleter())
        {}
        Type &operator[](size_t idx)    // index operators
        {
           return shared_ptr<Type>::get()[idx];
        }
        Type const &operator[](size_t idx) const
        {
           return shared_ptr<Type>::get()[idx];
        }
        Type &operator*() = delete;     // delete pointless members
        Type const &operator*() const = delete;
        Type *operator->() = delete;
        Type const *operator->() const = delete;
};
int main()
{
    shared_array<X> sp{ new X[3] };
    sp[0] = sp[1];
}
```

18.5: Smart smart pointer construction: `make_shared' and `make_unique'

Usually a shared_ptr is initialized at definition time with a pointer to a newly allocated object. Here is an example:

```
std::shared_ptr<string> sptr(new std::string("hello world"))
```

In such statements *two* memory allocation calls are used: one for the allocation of the std::string and one used interally by std::shared_ptr's constructor itself.

The two allocations can be combined into one single allocation (which is also slightly more efficient than explicitly calling `shared_ptr`'s constructor) using the `make_shared` template. The function template `std::make_shared` has the following prototype:

```
template<typename Type, typename ...Args>
std::shared_ptr<Type> std::make_shared(Args ...args);
```

Before using `make_shared` the `<memory>` header file must be included.

This function template allocates an object of type `Type`, passing `args` to its constructor (using *perfect forwarding*, see section 22.5.2), and returns a `shared_ptr` initialized with the address of the newly allocated `Type` object.

Here is how the above `sptr` object can be initialized using `std::make_shared`. Notice the use of `auto` which frees us from having to specify `sptr`'s type explicitly:

```
    auto sptr(std::make_shared<std::string>("hello world"));
```
After this initialization `std::shared_ptr<std::string> sptr` has been defined and initialized. It could be used as follows:
```
    std::cout << *sptr << '\n';
```

In addition to `make_shared` the function `std::make_unique` can be used. It can be used `make_shared` but returns a `std::unique_ptr` rather than a `shared_ptr`.

18.6: Classes having pointer data members

Classes having pointer data members require special attention. In particular at construction time one must be careful to prevent wild pointers and/or memory leaks. Consider the following class defining two pointer data members:
```
    class Filter
    {
        istream *d_in;
        ostream *d_out;
        public:
            Filter(char const *in, char const *out);
    };
```
Assume that `Filter` objects filter information read from `*d_in` and write the filtered information to `*d_out`. Using pointers to streams allows us to have them point at any kind of stream like `istreams, ifstreams, fstreams` or `istringstreams`. The shown constructor could be implemented like this:
```
    Filter::Filter(char const *in, char const *out)
    :
        d_in(new ifstream{ in }),
        d_out(new ofstream{ out })
    {
        if (!*d_in || !*d_out)
            throw string("Input and/or output stream not available");
    }
```
Of course, the construction could fail. `new` could throw an exception; the stream constructors could throw exceptions; or the streams could not be opened in which case an exception is thrown from the constructor's body. Using a function try block helps. Note that if `d_in`'s initialization throws, there's nothing to be worried about. The `Filter` object hasn't been constructed, its destructor is not called and processing continues at the point where the thrown exception is caught. But `Filter`'s destructor is also not called when `d_out`'s initialization or the constructor's `if` statement throws: no object, and hence no destructor is called. This may result in memory leaks, as `delete` isn't called for `d_in` and/or `d_out`. To prevent this, `d_in` and `d_out` must first be initialized to 0 and only then the initialization can be performed:

```
    Filter::Filter(char const *in, char const *out)
    try
    :
        d_in(0),
        d_out(0)
    {
        d_in = new ifstream{ in };
        d_out = new ofstream{ out };

        if (!*d_in || !*d_out)
            throw string("Input and/or output stream not available");
    }
    catch (...)
    {
        delete d_out;
        delete d_in;
    }
```
This quickly gets complicated, though. If Filter harbors yet another data member of a class whose constructor needs two streams then that data cannot be constructed or it must itself be converted into a pointer:
```
    Filter::Filter(char const *in, char const *out)
    try
    :
        d_in(0),
        d_out(0)
        d_filterImp(*d_in, *d_out)    // won't work
    { ... }

    // instead:

    Filter::Filter(char const *in, char const *out)
    try
    :
        d_in(0),
        d_out(0),
        d_filterImp(0)
    {
        d_in = new ifstream(in);
        d_out = new ofstream(out);
        d_filterImp = new FilterImp(*d_in, *d_out);
        ...
    }
    catch (...)
    {
        delete d_filterImp;
        delete d_out;
        delete d_in;
    }
```
Although the latter alternative works, it quickly gets hairy. In situations like these smart pointers should be used to prevent the hairiness. By defining the stream pointers as (smart pointer) objects they will, once constructed, properly be destroyed even if the rest of the constructor's code throws exceptions. Using a FilterImp and two unique_ptr data members Filter's setup and its constructor becomes:
```
    class Filter
```

```
{
    std::unique_ptr<std::ifstream> d_in;
    std::unique_ptr<std::ofstream> d_out;
    FilterImp d_filterImp;
    ...
};

Filter::Filter(char const *in, char const *out)
try
:
    d_in(new ifstream(in)),
    d_out(new ofstream(out)),
    d_filterImp(*d_in, *d_out)
{
    if (!*d_in || !*d_out)
        throw string("Input and/or output stream not available");
}
```

We're back at the original implementation but this time without having to worry about wild pointers and memory leaks. If one of the member initializers throws the destructors of previously constructed data members (which are now objects) are always called.

As a rule of thumb: when classes need to define pointer data members they should define those pointer data members as smart pointers if there's any chance that their constructors throw exceptions.

18.7: Regular Expressions

C++ itself provides facilities for handling regular expressions. Regular expressions were already available in **C++** via its **C** heritage (as **C** has always offered functions like `regcomp` and `regexec`), but the dedicated regular expression facilities have a richer interface than the traditional **C** facilities, and can be used in code using templates.

Before using the specific **C++** implementations of regular expressions the header file `<regex>` must be included.

Regular expressions are extensively documented elsewhere (e.g., **regex**(7), Friedl, J.E.F Mastering Regular Expressions, O'Reilly). The reader is referred to these sources for a refresher on the topic of regular expressions. In essence, regular expressions define a small meta-language recognizing textual units (like `numbers', `identifiers', etc.). They are extensively used in the context of *lexical scanners* (cf. section 24.6.1) when defining the sequence of input characters associated with *tokens*. But they are also intensively used in other situations. Programs like **sed**(1) and **grep**(1) use regular expressions to find pieces of text in files having certain characteristics, and a program like **perl**(1) adds some `sugar' to the regular expression language, simplifying the construction of regular expressions. However, though extremely useful, it is also well known that regular expressions tend to be very hard to read. Some even call the regular expression language a *write-only language*: while specifying a regular expression it's often clear why it's written in a particular way. But the opposite, understanding what a regular expression is supposed to represent if you lack the proper context, can be extremely difficult. That's why, from the onset and as a *rule of thumb*, it is stressed that an appropriate comment should be provided, with *each* regular expression, as to what it is supposed to match.

In the upcoming sections first a short overview of the regular expression language is provided, which is then followed by the facilities **C++** is currently offering for using regular expressions. These facilities mainly consist of classes helping you to specify regular expression, matching them to text, and determining which parts of the text (if any) match (parts of) the text being analyzed.

18.7.1: The regular expression mini language

Regular expressions are expressions consisting of elements resembling those of numeric expressions. Regular expressions consist of basic elements and operators, having various priorities and associations. Like numeric expressions, parentheses can be used to group elements together to form a unit on which operators operate. For an extensive discussion the reader is referred to, e.g., section 15.10 of the ecma-international.org page, which describes the characteristics of the regular expressions used by default by **C++**'s `regex` classes.

C++'s default definition of regular expressions distinguishes the following *atoms*:

- `x`: the character `x';
- `.`: any character except for the newline character;
- `[xyz]`: a character class; in this case, either an `x', a `y', or a `z' matches the regular expression. See also the paragraph about character classes below;
- `[abj-oZ]`: a character class containing a range of characters; this regular expression matches an `a', a `b', any letter from `j' through `o', or a `Z'. See also the paragraph about character classes below;
- `[^A-Z]`: a negated character class: this regular expression matches any character but those in the class beyond `^`. In this case, any character *except for* an uppercase letter. See also the paragraph about character classes below;
- `[:predef:]`: a *predefined* set of characters. See below for an overview. When used, it is interpreted as an element in a character class. It is therefore always embedded in a set of square brackets defining the character class (e.g., `[[:alnum:]]`);
- `\X`: if X is `a', `b', `f', `n', `r', `t', or `v', then the ANSI-C interpretation of `\x'. Otherwise, a literal `X' (used to escape operators such as `*`);
- `(r)`: the regular expression `r`. It is used to override precedence (see below), but also to define `r` as a *marked sub-expression* whose matching characters may directly be retrieved from, e.g., a `std::smatch` object (cf. section 18.7.3);

- `(?:r)`: the regular expression `r`. It is used to override precedence (see below), but it is *not* regarded as a *marked sub-expression*;

In addition to these basic atoms, the following special atoms are available (which can also be used in character classes):

- `\s`: a whitespace character;
- `\S`: any character but a whitespace character;
- `\d`: a decimal digit character;
- `\D`: any character but a decimal digit character;
- `\w`: an alphanumeric character or an underscore (_) character;
- `\W`: any character but an alphanumeric character or an underscore (_) character.

Atoms may be concatenated. If `r` and `s` are atoms then the regular expression `rs` matches a target text if the target text matches `r` *and* `s`, in that order (without any intermediate characters inside the target text). E.g., the regular expression `[ab][cd]` matches the target text `ac`, but not the target text `a:c`.

Atoms may be combined using operators. Operators bind to the preceding atom. If an operator should operate on multiple atoms the atoms must be surrounded by parentheses (see the last element in the previous itemization). To use an operator character as an atom it can be escaped. Eg., `*` represents an operator, `\*` the atom character star. Note that character classes do not recognize escape sequences: `[\*]` represents a character class consisting of two characters: a backslash and a star.

The following operators are supported (`r` and `s` represent regular expression atoms):

- `r*`: zero or more `rs`;
- `r+`: one or more `rs`;
- `r?`: zero or one `rs` (that is, an optional r);
- `r{m, n}`: where `1 <= m <= n`: matches `r' at least m, but at most n times;
- `r{m,}`: where `1 <= m`: matches `r' at least m times;
- `r{m}`: where `1 <= m`: matches `r' exactly m times;

- `r|s`: matches either an \`r' or an \`s'. This operator has a lower priority than any of the multiplication operators;
- `^r` : `^` is a pseudo operator. This expression matches \`r', if appearing at the beginning of the target text. If the `^`-character is not the first character of a regular expression it is interpreted as a literal `^`-character;
- `r$`: `$` is a pseudo operator. This expression matches \`r', if appearing at the end of the target text. If the `$`-character is not the last character of a regular expression it is interpreted as a literal `$`-character;

When a regular expression contains marked sub-expressions and multipliers, and the marked sub-expressions are multiply matched, then the target's final sub-string matching the marked sub-expression is reported as the text matching the marked sub-expression. E.g, when using `regex_search` (cf. section 18.7.4.3), marked sub-expression (`((a|b)+\s?)`), and target text `a a b`, then `a a b` is the fully matched text, while `b` is reported as the sub-string matching the first and second marked sub-expressions.

18.7.1.1: Character classes

Inside a character class all regular expression operators lose their special meanings, except for the special atoms `\s`, `\S`, `\d`, `\D`, `\w`, and `\W`; the character range operator -; the end of character class operator `]`; and, at the beginning of the character class, `^`. Except in combination with the special atoms the escape character is interpreted as a literal backslash character (to define a character class containing a backslash and a `d` simply use `[d\]`).

To add a closing bracket to a character class use `[]` immediately following the initial open-bracket, or start with `[^]` for a negated character class not containing the closing bracket. Minus characters are used to define character ranges (e.g., `[a-d]`, defining `[abcd]`) (be advised that the actual range may depend on the locale being used). To add a literal minus character to a character class put it at the very beginning (`[-`, or `[^-`) or at the very end (`-]`) of a character class.

Once a character class has started, all subsequent characters are added to the class's set of characters, until the final closing bracket (`]`) has been reached.

In addition to characters and ranges of characters, character classes may also contain *predefined sets of character*. They are:

```
    [:alnum:] [:alpha:] [:blank:]
    [:cntrl:] [:digit:] [:graph:]
    [:lower:] [:print:] [:punct:]
    [:space:] [:upper:] [:xdigit:]
```

These predefined sets designate sets of characters equivalent to the corresponding standard **C** `isXXX` function. For example, `[:alnum:]` defines all characters for which **isalnum**(3) returns true.

18.7.2: Defining regular expressions: std::regex

Before using the `(w)regex` class presented in this section the `<regex>` header file must be included.

The types `std::regex` and `std::wregex` define regular expression patterns. They define, respectively the types `basic_regex<char>` and `basic_regex<wchar_t>` types. Below, the class `regex` is used, but in the examples `wregex` could also have been used.

Regular expression facilities were, to a large extent, implemented through templates, using, e.g., the `basic_string<char>` type (which is equal to `std::string`). Likewise, generic types like *OutputIter* (output iterator) and *BidirConstIter* (bidirectional const iterator) are used with several functions. Such functions are function templates. Function templates determine the actual types from the arguments that are provided at *call-time*.

These are the steps that are commonly taken when using regular expressions:

- First, a regular expression is defined. This involves defining or modifying a `regex` object.
- Then the regular expression is provided with a *target text*, which may result in sections of the target text matching the regular expression.
- The sections of the target text matching (or not matching) the regular expression are retrieved to be processed elsewhere, or:
- The sections of the target text matching (or not matching) the regular expression are directly modified by existing regular expression facilities, after which the modified target text may be processed elsewhere.

The way `regex` objects handle regular expressions can be configured using a `bit_or` combined set of `std::regex_constants` values, defining a `regex::flag_type` value. These `regex_constants` are:

- `std::regex_constants::awk`:

  awk(1)'s (POSIX) regular expression grammar is used to specify regular exressions (e.g., regular expressions are delimited by `/`-characters, like `/\w+/`; for further details and for details of other regular expression grammars the reader should consult the man-pages of the respective programs);

- `std::regex_constants::basic`:

  the basic POSIX regular expression grammar is used to specify regular expressions;

- `std::regex_constants::collate`:

the character range operator (`-`) used in character classes defines a locale sensitive range (e.g., `[a-k]`);

- `std::regex_constants::ECMAScript`:

  this `flag_type` is used by default by `regex` constructors. The regular expression uses the Modified ECMAScript regular expression grammar;

- `std::regex_constants::egrep`:

  egrep(1)'s (POSIX) regular expression grammar is used to specify regular expressions. This is the same grammar as used by `regex_constants::extended`, with the addition of the newline character (`'\n'`) as an alternative for the `'|'`-operator;

- `std::regex_constants::extended`:

  the extended POSIX regular expression grammar is used to specify regular expressions;

- `std::regex_constants::grep`:

  grep(1)'s (POSIX) regular expression grammar is used to specify regular expressions. This is the same grammar as used by `regex_constants::basic`, with the addition of the newline character (`'\n'`) as an alternative for the `'|'`-operator;

- `std::regex_constants::icase`:

  letter casing in the target string is ignored. E.g., the regular expression `A` matches `a` and `A`;

- `std::regex_constants::nosubs`:

  When performing matches, all sub-expressions (`(expr)`) are treated as non-marked (`?:expr`);

- `std::regex_constants::optimize`:

  optimizes the speed of matching regular expressions, at the cost of slowing down the construction of the regular expression somewhat. If the same regular expression object is frequently used then this flag may substantially improve the speed of matching target texts;

Constructors

The default, move and copy constructors are available. Actually, the default constructor defines one parameter of type `regex::flag_type`, for which the value `regex_constants::ECMAScript` is used by default.

- `regex()`:

  the default constructor defines a `regex` object not containing a regular expression;

- `explicit regex(char const *pattern)`:

  defines a `regex` object containing the regular expression found at `pattern`;

- `regex(char const *pattern, std::size_t count)`:

  defines a `regex` object containing the regular expression found at the first `count` characters of `pattern`;

- `explicit regex(std::string const &pattern)`:

  defines a `regex` object containing the regular expression found at `pattern`. This constructor is defined as a member template, accepting a `basic_string`-type argument which may also use non-standard character traits and allocators;

- `regex(ForwardIterator first, ForwardIterator last)`:

  defines a `regex` object containing the regular expression found at the (forward) iterator range `[first, last)`. This constructor is defined as a member template, accepting any forward iterator type (e.g., plain `char` pointers) which can be used to define the regular expression's pattern;

- `regex(std::initializer_list<Char> init)`:

  defines a `regex` object containing the regular expression from the characters in the initializer list `init`.

  Here are some examples:

```
    std::regex re("\\w+");      // matches a sequence of alpha-numeric
                                // and/or underscore characters

    std::regex re{'\\', 'w', '+'} ;     // idem

    std::regex re(R"(\w+xxx")", 3);     // idem
```

Member functions

- `regex &operator=(RHS)`:

    The copy and move assignment operators are available. Otherwise, RHS may be:

    - an NTBS (of type `char const *`);
    - a `std::string const &` (or any compatible `std::basic_string`);
    - a `std::initializer_list<char>`;

- `regex &assign(RHS)`:

    This member accepts the same arguments as `regex's` constructors, including the (optional) `regex_constants` values;

- `regex::flag_type flag() const`:

    Returns the `regex_constants` flags that are active for the current `regex` object. E.g.,

```
    int main()
    {
        regex re;

        regex::flag_type flags = re.flags();

        cout <<                                // displays: 16 0
0
            (re.flags() & regex_constants::ECMAScript) << ' '  <<
            (re.flags() & regex_constants::icase) << ' '  <<
            (re.flags() & regex_constants::awk) << ' '  << '\n';
    }
```

Note that when a combination of `flag_type` values is specified at construction-time that only those flags that were specified are set. E.g., when `re(regex_constants::icase)` would have been specified the `cout` statement would have shown `0 1 0`. It's also possible to specify conflicting combinations of flag-values like `regex_constants::awk | regex_constants::grep`. The construction of such `regex` objects succeeds, but should be avoided.

- `locale_type get_loc() const`:

    Returns the locale that is associated with the current `regex` object;

- `locale_type imbue(locale_type locale)`:

Replaces the `regex` object's current locale setting with `locale`, returning the replaced locale;

- `unsigned mark_count() const:`

  The number of *marked sub-expressions* in the `regex` objext is returned. E.g.,

```
    int main()
    {
        regex re("(\\w+)([[:alpha:]]+)");
        cout << re.mark_count() << '\n';        // displays: 2
    }
```

- `void swap(regex &other) noexcept:`

  Swaps the current `regex` object with `other`. Also available as a free function: `void swap(regex &lhs, regex &rhs)`, swapping `lhs` and `rhs`.

18.7.3: Retrieving matches: std::match_results

Once a regex object is available, it can be used to match some target text against the regular expression. To match a target text against a regular expression the following functions, described in the next section (18.7.4), are available:

- `regex_match` merely matches a target text against a regular expression, informing the caller whether a match was found or not;
- `regex_search` also matches a target text against a regular expression, but allows retrieval of matches of marked sub-expressions (i.e., parenthesized regular expressions);
- `regex_replace` matches a target text against a regular expression, and replaces pieces of matched sections of the target text by another text.

These functions must be provided with a target text and a `regex` object (which is not modified by these functions). Usually another argument, a `std::match_results` object is also passed to these functions, to contain the results of the regular expression matching procedure.

Before using the `match_results` class the `<regex>` header file must be included.

Examples of using `match_results` objects are provided in section 18.7.4. This and the next section are primarily for referential purposes.

Various specializations of the class `match_results` exist. The specialization that is used should match the specializations of the used `regex` class. E.g., if the regular expression was specified as a `char const *` the `match_results` specialization should also operate on `char const *` values. The various specializations of `match_results` have been given names that can easily be remembered, so selecting the appropriate specialization is simple.

The class `match_results` has the following specializations:

- `cmatch`:

  defines `match_results<char const *>`, using a `char const *` type of iterator. It should be used with a `regex(char const *)` regular expression specification;

- `wcmatch`:

  defines `match_results<wchar_ const *>`, using a `wchar_t const *` type of iterator. It should be used with a `regex(wchar_t const *)` regular expression specification;

- `smatch`:

  defines `match_results<std::string::const_iterator>`, using a `std::string::const_iterator` type of iterator. It should be used with a `regex(std::string const &)` regular expression specification;

- `wsmatch`:

  defines `match_results<std::wstring::const_iterator>`, using a `std::wstring::const_iterator` type of iterator. It should be used with a `regex(wstring const &)` regular expression specification.

Constructors

The default, copy, and move constructors are available. The default constructor defines an `Allocator const &` parameter, which by default is initialized to the default allocator. Normally, objects of the class `match_results` receive their match-related information by passing them to the above-mentioned functions, like `regex_match`. When returning from these functions members of the class `match_results` can be used to retrieve specific results of the matching process.

Member functions

- `match_results &operator=`:

  The copy and move assignment operators are available;

- `std::string const &operator[](size_t idx) const`:

Returns a (const) reference to sub-match `idx`. With `idx` value 0 a reference to the full match is returned. If `idx >= size()` (see below) a reference to an empty sub-range of the target string is returned. The behavior of this member is undefined if the member `ready()` (see below) returns `false`;

- `Iterator begin() const`:

  Returns an iterator to the first sub-match. `Iterator` is a const-iterator for `const match_results` objects;

- `Iterator cegin() const`:

  Returns an iterator to the first sub-match. `Iterator` is a const-iterator;

- `Iterator cend() const`:

  Returns an iterator pointing beyond the last sub-match. `Iterator` is a const-iterator;

- `Iterator end() const`:

  Returns an iterator pointing beyond the last sub-match. `Iterator` is a const-iterator for `const match_results` objects;

- `ReturnType format(Parameters) const`:

  As this member requires a fairly extensive description, it would break the flow of the current overview. This member is used in combination with the `regex_replace` function, and it is therefore covered in detail in that function's section (18.7.4.5);

- `allocator_type get_allocator() const`:

  Returns the object's allocator;

- `bool empty() const`:

  Returns `true` if the `match_results` object contains no matches (which is also returned after merely using the default constructor). Otherwise it returns `false`;

- `int length(size_t idx = 0) const`:

  Returns the length of sub-match `idx`. By default the length of the full match is returned. If `idx >= size()` (see below) 0 is returned;

- `size_type max_size() const`:

  Returns the maximum number of sub-matches that can be contained in a `match_results` object. This is an implementation dependent constant value;

- `int position(size_t idx = 0) const`:

  Returns the offset in the target text of the first character of sub-match `idx`. By default the position of the first character of the full match is returned. If `idx >= size()` (see below) -1 is returned;

- `std::string const &prefix() const`:

  Returns a (const) reference to a sub-string of the target text that ends at the first character of the full match;

- `bool ready() const`:

  No match results are available from a default constructed `match_results` object. It receives its match results from one of the mentioned matching functions. Returns `true` once match results are available, and `false` otherwise.

- `size_type size() const`:

  Returns the number of sub-matches. E.g., with a regular expression `(abc)|(def)` and target `defcon` three submatches are reported: the total match (def); the empty text for `(abc)`; and `def` for the `(def)` marked sub-expression.

  Note: when multipliers are used only the last match is counted and reported. E.g., for the pattern `(a|b)+` and target `aaab` *two* sub-matches are reported: the total match `aaab`, and the last match (`b`);

- `std::string str(size_t idx = 0) const`:

  Returns the characters defining sub-match `idx`. By default this is the full match. If `idx >= size()` (see below) an empty string returned;

- `std::string const &suffix() const`:

  Returns a (const) reference to a sub-string of the target text that starts beyond the last character of the full match;

- `void swap(match_results &other) noexcept`:

Swaps the current `match_results` object with `other`. Also available as a free function: `void swap(match_results &lhs, match_results &rhs)`, swapping `lhs` and `rhs`.

18.7.4: Regular expression matching functions

Before using the functions presented in this section the `<regex>` header file must be included.

There are three major families of functions that can be used to match a target text against a regular expression. Each of these functions, as well as the `match_results::format` member, has a final `std::regex_constants::match_flag_type` parameter (see the next section), which is given the default value `regex_constants::match_default` which can be used to fine-tune the way the regular expression and the matching process is being used. This final parameter is not explicitly mentioned with the regular expression matching functions or with the `format` member. The three families of functions are:

- `bool std::regex_match(Parameters)`:

  This family of functions is used to match a regular expression against a target text. Only if the regular expression matches the full target text `true` is returned; otherwise `false` is returned. Refer to section 18.7.4.2 for an overview of the available overloaded `regex_match` functions;

- `bool std::regex_search(Parameters)`:

  This family of functions is also used to match a regular expression against a target text. This function returns true once the regular expression matches a sub-string of the target text; otherwise `false` is returned. See below for an overview of the available overloaded `regex_search` functions;

- `ReturnType std::regex_replace(Parameters)`:

  This family of functions is used to produce modified texts, using the characters of a target string, a `regex` object and a format string. This member closely resembles the functionality of the `match_results::format` member discussed in section 18.7.4.4.

The match_results::format member can be used after regex_replace and is discussed after covering regex_replace (section 18.7.4.4).

18.7.4.1: The std::regex_constants::match_flag_type flags

All overloaded format members and all regular expression matching functions accept a final regex_constants::match_flag_type argument, which is a bit-masked type, for which the bit_or operator can be used. All format members by default specify the argument match_default.

The `match_flag_type` enumeration defines the following values (below, ``[first, last)`' refers to the character sequence being matched).

- `format_default` (not a bit-mask value, but a default value which is equal to 0). With just this specification ECMAScript rules are used to construct strings in `std::regex_replace`;

- `format_first_only`: `std::regex_replace` only replaces the first match;

- `format_no_copy`: non-matching strings are not passed to the output by `std::regex_replace`;

- `format_sed`: POSIX **sed**(1) rules are used to construct strings in `std::regex_replace`;

- `match_any`: if multiple matches are possible, then any match is an acceptable result;

- `match_continuous`: sub-sequences are only matching if they start at `first`;

- `match_not_bol`: the first character in `[first, last)` is treated as an ordinary character: `^` does not match `[first, first)`;

- `match_not_bow`: `\b` does not match `[first, first)`;

- `match_default` (not a bit-mask value, but equal to 0): the default value of the final argument that's passed to the regular expression matching functions and `match_results::format` member. ECMAScript rules are used to construct strings in `std::regex_replace`;

- `match_not_eol`: the last character in `[first, last)` is treated as an ordinary character: `$` does not match `[last,last)`;

- `match_not_eow`: `\b` does not match `[last, last)`;

- `match_not_null`: empty sequences are not considered matches;

- `match_prev_avail`: `--first` refers to a valid character position. When specified `match_not_bol` and `match_not_bow` are ignored;

18.7.4.2: Matching full texts: std::regex_match

The regular expression matching function std::regex_match returns true if the regular expression defined in its provided regex argument *fully* matches the provided target text. This means that match_results::prefix and match_results::suffix must return empty strings. But defining sub-expressions is OK.

The following overloaded variants of this function are available:

- `bool regex_match(BidirConstIter first, BidirConstIter last, std::match_results &results, std::regex const &re)`:

  `BidirConstIter` is a bidirectional const iterator. The range `[first, last)` defines the target text. The match results are returned in `results`. The types of the iterators must match the type of the `match_results` that's used. E.g., a `cmatch` should be used if the iterators are of `char const *` types, and a `smatch` should be used if the iterators are of `string::const_iterator` types. Similar correspondence requirements hold true for the other overloaded versions of this function;

- `bool regex_match(BidirConstIter first, BidirConstIter last, std::regex const &re)`:

  this function behaves like the previous function, but does not return the results of the matching process in a `match_results` object;

- `bool regex_match(char const *target, std::match_results &results, std::regex const &re)`:

  this function behaves like the first overloaded variant, using the characters in `target` as its target text;

- `bool regex_match(char const *str, std::regex const &re)`:

  this function behaves like the previous function but does not return the match results;

- `bool regex_match(std::string const &target, std::match_results &results, std::regex const &re)`:

  this function behaves like the first overloaded variant, using the characters in `target` as its target text;

- `bool regex_match(std::string const &str, std::regex const &re)`:

  this function behaves like the previous function but does not return the match results;

- `bool regex_match(std::string const &&, std::match_results &, std::regex &) = delete` (the `regex_match` function does not accept temporary `string` objects as target strings, as this would result in invalid string iterators in the `match_result` argument.)

Here is a small example: the regular expression matches the matched text (provided by argv[1]) if it starts with 5 digits and then merely contains letters ([[:alpha:]]). The digits can be retrieved as sub-expression 1:

```
#include <iostream>
#include <regex>

using namespace std;

int main(int argc, char const **argv)
{
    regex re("(\\d{5})[[:alpha:]]+");

    cmatch results;

    if (not regex_match(argv[1], results, re))
        cout << "No match\n";
    else
        cout << "size: " << results.size() << ": " <<
                results.str(1) << " -- " << results.str() << '\n';
}
```

18.7.4.3: Partially matching text: std::regex_search

Different from regex_match the regular expression matching function std::regex_search returns true if the regular expression defined in its regex argument partially matches the target text.

The following overloaded variants of this function are available:

- `bool regex_search(BidirConstIter first, BidirConstIter last, std::match_results &results, std::regex const &re)`:

  `BidirConstIter` is a bidirectional const iterator. The range `[first, last)` defines the target text. The match results are returned in `results`. The types of the iterators must match the type of the `match_results` that's used. E.g., a `cmatch` should be used if the iterators are of `char const *` types, and a `smatch` should be used if the iterators are of `string::const_iterator` types. Similar correspondence requirements hold true for the other overloaded versions of this function;

- `bool regex_search(BidirConstIter first, BidirConstIter last, std::regex const &re)`:

  this function behaves like the previous function, but does not return the results of the matching process in a `match_results` object;

- `bool regex_search(char const *target, std::match_results &results, std::regex const &re)`:

this function behaves like the first overloaded variant, using the characters in `target` as its target text;

- `bool regex_search(char const *str, std::regex const &re)`:

  this function behaves like the previous function but does not return the match results;

- `bool regex_search(std::string const &target, std::match_results &results, std::regex const &re)`:

  this function behaves like the first overloaded variant, using the characters in `target` as its target text;

- `bool regex_search(std::string const &str, std::regex const &re)`:

  this function behaves like the previous function but does not return the match results;

- `bool regex_search(std::string const &&, std::match_results &, std::regex &) = delete`:

  the `regex_search` function does not accept temporary `string` objects as target strings, as this would result in invalid string iterators in the `match_result` argument.

The following example illustrates how regex_search could be used:

```
#include <iostream>
#include <string>
#include <regex>

using namespace std;

int main()
{
   while (true)
    {
       cout << "Enter a pattern or plain Enter to stop: ";

       string pattern;
       if (not getline(cin, pattern) or pattern.empty())
           break;

       regex re(pattern);
       while (true)
       {
           cout << "Enter a target text for `" << pattern << "'\n"
                   "(plain Enter for the next pattern): ";

```

```
            string text;
            if (not getline(cin, text) or text.empty())
                break;

            smatch results;
            if (not regex_search(text, results, re))
                cout << "No match\n";
            else
            {
                cout << "Prefix: "  << results.prefix() << "\n"
                        "Match:  "  << results.str()    << "\n"
                        "Suffix: "  << results.suffix() << "\n";
                for (size_t idx = 1; idx != results.size(); ++idx)
                    cout << "Match " << idx << " at offset " <<
                                results.position(idx) << ": " <<
                                results.str(idx) << '\n';
            }
        }
    }
}
```

18.7.4.4: The member std::match:_results::format

The match_results::formatformat member is a rather complex member function of the class match_results, which can be used to modify text which was previously matched against a regular expression, e.g., using the function regex_search. Because of its complexity and because the functionality of another regular expression processing function (regex_replace) offers similar functionality it is discussed at this point in the **C++** Annotations, just before discussing the regex_replace function.

The `format` member operates on (sub-)matches contained in a `match_results` object, using a *format string*, and producing text in which format specifiers (like `$&`) are replaced by matching sections of the originally provided target text. In addition, the `format` member recognizes all standard **C** escape sequences (like `\n`). The `format` member is used to create text that is modified with respect to the original target text.

As a preliminary illustration: if `results` is a `match_results` object and `match[0]` (the fully matched text) equals `` ` ``hello world`', then calling `format` with the format string `this is [$&]` produces the text `this is [hello world]`. Note the specification `$&` in this format string: this is an example of a format specifier. Here is an overview of all supported format specifiers:

- `` $` ``: corresponds to the text returned by the `prefix` member: all characters in the original target text up to the first character of the fully matched text;

- `$&`: corresponds to the fully matched text (i.e., the text returned by the `match_results::str` member);

- `$n`: (where `n` is an integral natural number): corresponds to the text returned bu `operator[](n)`;

- `$'`: corresponds to the text returned by the `suffix` member: all characters in the original target string beyond the last character of the fully matched text;

- `$$`: corresponds to the single `$` character.

Four overloaded versions of the `format` members are available. All overloaded versions define a final `regex_constants::match_flag_type` parameter, which is by default initialized to `match_default`. This final parameter is not explicitly mentioned in the following coverage of the `format` members.

To further illustrate the way the `format` members can be used it is assumed that the following code has been executed:

```
1:    regex re("([[:alpha:]]+)\\s+(\\d+)");  // letters blanks digits
2:
3:    smatch results;
4:    string target("this value 1024 is interesting");
5:
6:    if (not regex_search(target, results, re))
7:        return 1;
```

After calling `regex_search` (line 6) the results of the regular expression matching process are available in the `match_results results` object that is defined in line 3.

The first two overloaded `format` functions expect an output-iterator to where the formatted text is written. These overloaded members return the final output iterator, pointing just beyond the character that was last written.

- `OutputIter format(OutputIter out, char const *first, char const *last) const`:

  the characters in the range `[first, last)` are applied to the sub-expressions stored in the `match_results` object, and the resulting string is inserted at `out`. An illustration is provided with the next overloaded version;

- `OutputIter format(OutputIter out, std::string const &fmt) const`:

  the contents of `fmt` are applied to the sub-expressions stored in the `match_results` object, and the resulting string is inserted at `out`. The next line of code inserts the value 1024 into `cout` (note that `fmt` *must* be a `std::string`, hence the explicit use of the `string` constructor):

  ```
  results.format(ostream_iterator<char>(cout, ""), string("$2"));
  ```

The remaining two overloaded `format` members expect a `std::string` or an NTBS defining the format string. Both members return a `std::string` containing the formatted text:

- `std::string format(std::string const &fmt) const`

- `std::string format(char const *fmt) const`

The next example shows how a string can be obtained in which the order of the first and second marked sub-expressions contained in the previously obtained match_results object have been swapped:

```
string reverse(results.format("$2 and $1"));
```

18.7.4.5: Modifying target strings: std::regex_replace

The family of std::regex_replace functions uses regular expressions to perform substitution on sequences of characters. Their functionality closely resembles the functionality of the match_results::format member discussed in the previous section. The following overloaded variants are available:

- `OutputIt regex_replace(OutputIter out, BidirConstIter first, BidirConstIter last, std::regex const &re, std::string const &fmt)`:

  `OutputIter` is an output iterator; `BidirConstIter` a bidirectional const iterator.

  The function returns the possibly modified text in an iterator range `[out, retvalue)`, where `out` is the output iterator passed as the first argument to `regex_replace`, and `retvalue` is the output iterator returned by `regex_replace`.

  The function matches the text at the range `[first, last)` against the regular expression stored in `re`. If the regular expression does *not* match the target text in the range `[first, last)` then the target text is literally copied to `out`. If the regular expression *does* match the target text then

  - first, the match result's prefix is copied to `out`. The prefix equals the initial characters of the target text up to the very first character of the fully matched text.

  - next, the matched text is replaced by the contents of the `fmt` format string, in which the format specifiers can be used that were described in the previous section (section 18.7.4.4), and the replaced text is copied to `out`;

  - finally, the match result's suffix is copied to `out`. The suffix equals all characters of the target text beyond the last character of the matched text.

  The workings of `regex_replace` is illustrated in the next example:

```
    regex re("([[:alpha:]]+)\\s+(\\d+)");  // letters blanks digits

    string target("this value 1024 is interesting");

```

```
 5:     regex_replace(ostream_iterator<char>(cout, ""),
target.begin(),
 6:                     target.end(), re, string("$2"));
```

In line 5 `regex_replace` is called. Its format string merely contains `$2`, matching 1024 in the target text. The prefix ends at the word `value`, the suffix starts beyond 1024, so the statement in line 5 inserts the text

```
this 1024 is interesting
```

into the standard output stream.

- `OutputIt regex_replace( OutputIter out, BidirConstIter first, BidirConstIter last, std::regex const &re, char const *fmt)`:

  This variant behaves like the first variant. When using, in the above example, `"$2"` instead of `string("$2")`, then this variant would have been used;

- `std::string regex_replace(std::string const &str, std::regex const &re, std::string const &fmt)`:

  This variant returns a `std::string` containing the modified text, and expects a `std::string` containing the target text. Other than that, it behaves like the first variant. To use this overloaded variant in the above example the statement in line 5 could have been replaced by the following statement, initializing the `string result`:

  ```
  string result(regex_replace(target, re, string("$2")));
  ```

- `std::string regex_replace(std::string const &str, std::regex const &re, char const *fmt)`:

  After changing, in the above statement, `string("$2")` into `"$2"`, this variant is used, behaving exactly like the previous variant;

- `std::string regex_replace(char const *str, std::regex const &re, std::string const &fmt)`:

  This variant uses a `char const *` to point to the target text, and behaves exactly like the previous but one variant;

- `std::string regex_replace(char const *str, std::regex const &re, char const *fmt)`:

  This variant also uses a `char const *` to point to the target text, and also behaves exactly like the previous but one variant;

18.8: Randomization and Statistical Distributions

Before the statistical distributions and accompanying random number generators can be used the `<random>` header file must be included.

The STL offers several standard mathematical (statistical) distributions. These distributions allow programmers to obtain randomly selected values from a selected distribution.

These statistical distributions need to be provided with a random number generating object. Several of such random number generating objects are provided, extending the traditional `rand` function that is part of the **C** standard library.

These random number generating objects produce pseudo-random numbers, which are then processed by the statistical distribution to obtain values that are randomly selected from the specified distribution.

Although the STL offers various statistical distributions their functionality is fairly limited. The distributions allow us to obtain a random number from these distributions, but probability density functions or cumulative distribution functions are currently not provided by the STL. These functions (distributions as well as the density and the cumulative distribution functions) are, however, available in other libraries, like the boost math library (specifically: http://www.boost.org/doc/libs/1_44_0/libs/math/doc/sf_and_dist/html/index.html).

It is beyond the scope of the **C++** Annotations to discuss the mathematical characteristics of the various statistical distributions. The interested reader is referred to the pertinent mathematical textbooks (like Stuart and Ord's (2009) *Kendall's Advanced Theory of Statistics*, Wiley) or to web-locations like http://en.wikipedia.org/wiki/Bernoulli_distribution.

18.8.1: Random Number Generators

The following generators are available:

Class template	Integral/Floating point	Quality	Speed	Size of state
`linear_congruential_engine`	Integral	Medium	Medium	1
`subtract_with_carry_engine`	Both	Medium	Fast	25
mersenne_twister_engine	Integral	Good	Fast	624

The `linear_congruential_engine` random number generator computes

value$_{i+1}$ = OPENPAa * value$_{i}$ + c+) % m

It expects template arguments for, respectively, the data type to contain the generated random values; the multiplier `a`; the additive constant `c`; and the modulo value `m`. Example:

```
  linear_congruential_engine<int, 10, 3, 13> lincon;
```

The linear_congruential generator may be seeded by providing its constructor with a seeding-argument. E.g., lincon(time(0)).

The subtract_with_carry_engine random number generator computes

$$\text{value}_i = (\text{value}_{i-s} - \text{value}_{i-r} - \text{carry}_{i-1}) \% m$$

It expects template arguments for, respectively, the data type to contain the generated random values; the modulo value m; and the subtractive constants s and r. Example:

```
  subtract_with_carry_engine<int, 13, 3, 13> subcar;
```

The subtract_with_carry_engine generator may be seeded by providing its constructor with a seeding-argument. E.g., subcar(time(0)).

The predefined mersenne_twister_engine mt19937 (predefined using a typedef defined by the <random> header file) is used in the examples below. It can be constructed using `mt19937 mt' or it can be seeded by providing its constructor with an argument (e.g., mt19937 mt(time(0))). Other ways to initialize the mersenne_twister_engine are beyond the scope of the **C++** Annotations (but see Lewis *et al.* (Lewis, P.A.W., Goodman, A.S., and Miller, J.M. (1969), A pseudorandom number generator for the System/360, IBM Systems Journal, 8, 136-146.) (1969)).

The random number generators may also be seeded by calling their members seed accepting unsigned long values or generator functions (as in lc.seed(time(0)), lc.seed(mt)).

The random number generators offer members min and max returning, respectively, their minimum and maximum values (inclusive). If a reduced range is required the generators can be nested in a function or class adapting the range.

18.8.2: Statistical distributions

In the following sections the various statistical distributions that are supported by **C++** are covered. The notation RNG is used to indicate a *Random Number Generator* and URNG is used to indicate a *Uniform Random Number Generator*. With each distribution a struct param_type is defined containing the distribution's parameters. The organization of these param_type structs depends on (and is described at) the actual distribution.

All distributions offer the following members (*result_type* refers to the type name of the values returned by the distribution):

- result_type max() const
  returns the distribution's least upper bound;

- result_type min() const
  returns the distribution's greatest lower bound;

- param_type param() const
  returns the object's param_type struct;

- `void param(const param_type &param)` redefines the parameters of the distribution;

- `void reset()`: clears all of its cached values;

All distributions support the following operators (*distribution-name* should be replaced by the name of the intended distribution, e.g., `normal_distribution`):

- `template<typename URNG> result_type operator()(URNG &urng)`
  returns the next random value from the statistical distribution, with the function object `urng` returning the next random number selected from a uniform random distribution;

- `template<typename URNG> result_type operator()`
  `(URNG &urng, param_type &param)`
  returns the next random value from the statistical distribution initialized with the parameters provided by the `param` struct. The function object `urng` returns the next random number selected from a uniform random distribution;

- `std::istream &operator>>(std::istream &in, distribution-name &object)`:
  The parameters of the distribution are extracted from an `std::istream`;

- `std::ostream &operator<<(std::ostream &out, distribution-name const &bd)`: The parameters of the distribution are inserted into an `std::ostream`

The following example shows how the distributions can be used. Replacing the name of the distribution (`normal_distribution`) by another distribution's name is all that is required to switch distributions. All distributions have parameters, like the mean and standard deviation of the normal distribution, and all parameters have default values. The names of the parameters vary over distributions and are mentioned below at the individual distributions. Distributions offer members returning or setting their parameters.

Most distributions are defined as class templates, requiring the specification of a data type that is used for the function's return type. If so, an empty template parameter type specification (<>) will get you the default type. The default types are either `double` (for real valued return types) or `int` (for integral valued return types). The template parameter type specification must be omitted with distributions that are not defined as template classes.

Here is an example showing the use of the statistical distributions, applied to the normal distribution:

```
#include <iostream>
#include <ctime>
#include <random>
using namespace std;

int main()
{
   std::mt19937 engine(time(0));
```

```
    std::normal_distribution<> dist;

    for (size_t idx = 0; idx < 10; ++idx)
        std::cout << "a random value: " << dist(engine) << "\n";

    cout << '\n' <<
        dist.min() << " " << dist.max() << '\n';
}
```

18.8.2.1: Bernoulli distribution

The bernoulli_distribution is used to generate logical truth (boolean) values with a certain probability p. It is equal to a binomial distribution for one experiment (cf 18.8.2.2).

The bernoulli distribution is *not* defined as a class template.

Defined types:

```
    typedef bool result_type;
    struct param_type
    {
      explicit param_type(double prob = 0.5);
      double p() const;                     // returns prob
    };
```

Constructor and members:

- `bernoulli_distribution(double prob = 0.5)`
  constructs a bernoulli distribution with probability `prob` of returning `true`;

- `double p() const`
  returns `prob`;

- `result_type min() const`
  returns `false`;

- `result_type max() const`
  returns `true`;

18.8.2.2: Binomial distribution

The binomial_distribution<IntType = int> is used to determine the probability of the number of successes in a sequence of n independent success/failure experiments, each of which yields success with probability p.

The template type parameter `IntType` defines the type of the generated random value, which must be an integral type.

Defined types:

```
typedef IntType result_type;
struct param_type
{
  explicit param_type(IntType trials, double prob = 0.5);
  IntType t() const;                    // returns trials
  double p() const;                     // returns prob
};
```

Constructors and members and example:

- `binomial_distribution<>(IntType trials = 1, double prob = 0.5)` constructs a binomial distribution for `trials` experiments, each having probability `prob` of success.

- `binomial_distribution<>(param_type const &param)` constructs a binomial distribution according to the values stored in the `param` struct.

- `IntType t() const`
  returns `trials`;

- `double p() const`
  returns `prob`;

- `result_type min() const`
  returns 0;

- `result_type max() const`
  returns `trials`;

18.8.2.3: Cauchy distribution

The `cauchy_distribution<RealType = double>` looks similar to a normal distribution. But cauchy distributions have heavier tails. When studying hypothesis tests that assume normality, seeing how the tests perform on data from a Cauchy distribution is a good indicator of how sensitive the tests are to heavy-tail departures from normality.

The mean and standard deviation of the Cauchy distribution are undefined.

Defined types:

```
typedef RealType result_type;

struct param_type
{
    explicit param_type(RealType a = RealType(0),
                        RealType b = RealType(1));
    double a() const;
    double b() const;
};
```

Constructors and members:

- `cauchy_distribution<>(RealType a = RealType(0), RealType b = RealType(1))` constructs a cauchy distribution with specified `a` and `b` parameters.

- `cauchy_distribution<>(param_type const &param)` constructs a cauchy distribution according to the values stored in the `param` struct.

- `RealType a() const`
  returns the distribution's `a` parameter;

- `RealType b() const`
  returns the distribution's `b` parameter;

- `result_type min() const`
  returns the smallest positive `result_type` value;

- `result_type max() const`
  returns the maximum value of `result_type`;

18.8.2.4: Chi-squared distribution

The `chi_squared_distribution<RealType = double>` with `n` degrees of freedom is the distribution of a sum of the squares of `n` independent standard normal random variables.

Note that even though the distribution's parameter `n` usually is an integral value, it doesn't have to be integral, as the chi_squared distribution is defined in terms of functions (`exp` and `Gamma`) that take real arguments (see, e.g., the formula shown in the `<bits/random.h>` header file, provided with the Gnu `g++` compiler distribution).

The chi-squared distribution is used, e.g., when testing the goodness of fit of an observed distribution to a theoretical one.

Defined types:

```
typedef RealType result_type;

struct param_type
{
    explicit param_type(RealType n = RealType(1));

    RealType n() const;
};
```

Constructors and members:

- `chi_squared_distribution<>(RealType n = 1)` constructs a chi_squared distribution with specified number of degrees of freedom.
- `chi_squared_distribution<>(param_type const &param)` constructs a chi_squared distribution according to the value stored in the `param` struct;
- `IntType n() const`
  returns the distribution's degrees of freedom;
- `result_type min() const`
  returns 0;
- `result_type max() const`
  returns the maximum value of `result_type`;

18.8.2.5: Extreme value distribution

The extreme_value_distribution<RealType = double> is related to the Weibull distribution and is used in statistical models where the variable of interest is the minimum of many random factors, all of which can take positive or negative values.

It has two parameters: a location parameter `a` and scale parameter `b`. See also http://www.itl.nist.gov/div898/handbook/apr/section1/apr163.htm

Defined types:

```
typedef RealType result_type;

struct param_type
{
   explicit param_type(RealType a = RealType(0),
                   RealType b = RealType(1));

   RealType a() const;     // the location parameter
   RealType b() const;     // the scale parameter
};
```

Constructors and members:

- `extreme_value_distribution<>(RealType a = 0, RealType b = 1)` constructs an extreme value distribution with specified `a` and `b` parameters;
- `extreme_value_distribution<>(param_type const &param)` constructs an extreme value distribution according to the values stored in the `param` struct.

- `RealType a() const`
  returns the distribution's location parameter;

- `RealType stddev() const`
  returns the distribution's scale parameter;

- `result_type min() const`
  returns the smallest positive value of `result_type`;

- `result_type max() const`
  returns the maximum value of `result_type`;

18.8.2.6: Exponential distribution

The `exponential_distribution<RealType = double>` is used to describe the lengths between events that can be modelled with a homogeneous Poisson process. It can be interpreted as the continuous form of the geometric distribution.

Its parameter `prob` defines the distribution's *lambda* parameter, called its *rate* parameter. Its expected value and standard deviation are both `1 / lambda`.

Defined types:

```
typedef RealType result_type;

struct param_type
{
   explicit param_type(RealType lambda = RealType(1));

   RealType lambda() const;
};
```

Constructors and members:

- `exponential_distribution<>(RealType lambda = 1)` constructs an exponential distribution with specified `lambda` parameter.

- `exponential_distribution<>(param_type const &param)` constructs an exponential distribution according to the value stored in the `param` struct.

- `RealType lambda() const`
  returns the distribution's `lambda` parameter;

- `result_type min() const`
  returns 0;

- `result_type max() const`
  returns the maximum value of `result_type`;

18.8.2.7: Fisher F distribution

The `fisher_f_distribution<RealType = double>` is intensively used in statistical methods like the Analysis of Variance. It is the distribution resulting from dividing two *Chi-squared* distributions.

It is characterized by two parameters, being the degrees of freedom of the two chi-squared distributions.

Note that even though the distribution's parameter `n` usually is an integral value, it doesn't have to be integral, as the Fisher F distribution is constructed from Chi-squared distributions that accept a non-integral parameter value (see also section 18.8.2.4).

Defined types:

```
typedef RealType result_type;

struct param_type
{
    explicit param_type(RealType m = RealType(1),
                        RealType n = RealType(1));

    RealType m() const; // The degrees of freedom of the nominator
    RealType n() const; // The degrees of freedom of the denominator
};
```

Constructors and members:

- `fisher_f_distribution<>(RealType m = RealType(1), RealType n = RealType(1))` constructs a fisher_f distribution with specified degrees of freedom.

- `fisher_f_distribution<>(param_type const &param)` constructs a fisher_f distribution according to the values stored in the `param` struct.

- `RealType m() const`
  returns the degrees of freedom of the nominator;

- `RealType n() const`
  returns the degrees of freedom of the denominator;

- `result_type min() const`
  returns 0;

- `result_type max() const`
  returns the maximum value of `result_type`;

18.8.2.8: Gamma distribution

The gamma_distribution<RealType = double> is used when working with data that are not distributed according to the normal distribution. It is often used to model waiting times.

It has two parameters, `alpha` and `beta`. Its expected value is `alpha * beta` and its standard deviation is `alpha * beta`^{2}.

Defined types:

```
typedef RealType result_type;

struct param_type
{
   explicit param_type(RealType alpha = RealType(1),
                     RealType beta = RealType(1));

   RealType alpha() const;
   RealType beta() const;
};
```

Constructors and members:

- `gamma_distribution<>(RealType alpha = 1, RealType beta = 1)` constructs a gamma distribution with specified `alpha` and `beta` parameters.

- `gamma_distribution<>(param_type const &param)` constructs a gamma distribution according to the values stored in the `param` struct.

- `RealType alpha() const`
  returns the distribution's `alpha` parameter;

- `RealType beta() const`
  returns the distribution's `beta` parameter;

- `result_type min() const`
  returns 0;

- `result_type max() const`
  returns the maximum value of `result_type`;

18.8.2.9: Geometric distribution

The geometric_distribution<IntType = int> is used to model the number of bernoulli trials (cf. 18.8.2.1) needed until the first success.

It has one parameter, `prob`, representing the probability of success in an individual bernoulli trial.

Defined types:

```
typedef IntType result_type;

struct param_type
{
   explicit param_type(double prob = 0.5);
   double p() const;
};
```

Constructors, members and example:

- `geometric_distribution<>(double prob = 0.5)` constructs a geometric distribution for bernoulli trials each having probability `prob` of success.

- `geometric_distribution<>(param_type const &param)` constructs a geometric distribution according to the values stored in the `param` struct.

- `double p() const`
  returns the distribution's `prob` parameter;

- `param_type param() const`
  returns the object's `param_type` structure;

- `void param(const param_type &param)` redefines the parameters of the distribution;

- `result_type min() const`
  returns the distribution's lower bound (= `0`);

- `result_type max() const`
  returns the distribution's upper bound;

- `template<typename URNG> result_type operator()(URNG &urng)`
  returns the next random value from the geometric distribution

- `template<typename URNG> result_type operator()`
  `(URNG &urng, param_type &param)`
  returns the next random value from a geometric distribution initialized by the provided `param` struct.

- The random number generator that is passed to the generating functions must return integral values. Here is an example showing how the geometric distribution can be used:

```
• #include <iostream>
• #include <ctime>
• #include <random>
•
```

```
int main()
{
    std::linear_congruential_engine<unsigned, 7, 3, 61> engine(0);

    std::geometric_distribution<> dist;

    for (size_t idx = 0; idx < 10; ++idx)
        std::cout << "a random value: " << dist(engine) << "\n";

    std::cout << '\n' <<
        dist.min() << " " << dist.max() << '\n';

}
```

18.8.2.10: Log-normal distribution

The `lognormal_distribution<RealType = double>` is a probability distribution of a random variable whose logarithm is normally distributed. If a random variable `X` has a normal distribution, then Y = e^x has a log-normal distribution.

It has two parameters, *m* and *s* representing, respectively, the mean and standard deviation of `ln(X)`.

Defined types:

```
typedef RealType result_type;

struct param_type
{
    explicit param_type(RealType m = RealType(0),
                        RealType s = RealType(1));

    RealType m() const;
    RealType s() const;
};
```

Constructor and members:

- `lognormal_distribution<>(RealType m = 0, RealType s = 1)` constructs a log-normal distribution for a random variable whose mean and standard deviation is, respectively, `m` and `s`.

- `lognormal_distribution<>(param_type const &param)` constructs a log-normal distribution according to the values stored in the `param` struct.

- `RealType m() const`
  returns the distribution's `m` parameter;

- `RealType stddev() const`
  returns the distribution's `s` parameter;

- `result_type min() const`
  returns 0;

- `result_type max() const`
  returns the maximum value of `result_type`;

18.8.2.11: Normal distribution

The `normal_distribution<RealType = double>` is commonly used in science to describe complex phenomena. When predicting or measuring variables, errors are commonly assumed to be normally distributed.

It has two parameters, *mean* and *standard deviation*.

Defined types:

```
typedef RealType result_type;

struct param_type
{
    explicit param_type(RealType mean = RealType(0),
                        RealType stddev = RealType(1));

    RealType mean() const;
    RealType stddev() const;
};
```

Constructors and members:

- `normal_distribution<>(RealType mean = 0, RealType stddev = 1)` constructs a normal distribution with specified `mean` and `stddev` parameters. The default parameter values define the *standard normal distribution*;

- `normal_distribution<>(param_type const &param)` constructs a normal distribution according to the values stored in the `param` struct.

- `RealType mean() const`
  returns the distribution's `mean` parameter;

- `RealType stddev() const`
  returns the distribution's `stddev` parameter;

- `result_type min() const`
  returns the lowest positive value of `result_type`;

- `result_type max() const`
  returns the maximum value of `result_type`;

18.8.2.12: Negative binomial distribution

The negative_binomial_distribution<IntType = int> probability distribution describes the number of successes in a sequence of Bernoulli trials before a specified number of failures occurs. For example, if one throws a die repeatedly until the third time 1 appears, then the probability distribution of the number of other faces that have appeared is a negative binomial distribution.

It has two parameters: (`IntType`) k (> 0), being the number of failures until the experiment is stopped and (`double`) p the probability of success in each individual experiment.

Defined types:

```
typedef IntType result_type;

struct param_type
{
   explicit param_type(IntType k = IntType(1), double p = 0.5);

   IntType k() const;
   double p() const;
};
```

Constructors and members:

- `negative_binomial_distribution<>(IntType k = IntType(1), double p = 0.5)`
  constructs a negative_binomial distribution with specified `k` and `p` parameters;

- `negative_binomial_distribution<>(param_type const &param)` constructs a negative_binomial distribution according to the values stored in the `param` struct.

- `IntType k() const`
  returns the distribution's `k` parameter;

- `double p() const`
  returns the distribution's `p` parameter;

- `result_type min() const`
  returns 0;

- `result_type max() const`
  returns the maximum value of `result_type`;

18.8.2.13: Poisson distribution

The poisson_distribution<IntType = int> is used to model the probability of a number of events occurring in a fixed period of time if these events occur with a known probability and independently of the time since the last event.

It has one parameter, `mean`, specifying the expected number of events in the interval under consideration. E.g., if on average 2 events are observed in a one-minute interval and the duration of the interval under study is 10 minutes then `mean = 20`.

Defined types:

```
typedef IntType result_type;

struct param_type
{
    explicit param_type(double mean = 1.0);

    double mean() const;
};
```

Constructors and members:

- `poisson_distribution<>(double mean = 1)` constructs a poisson distribution with specified `mean` parameter.

- `poisson_distribution<>(param_type const &param)` constructs a poisson distribution according to the values stored in the `param` struct.

- `double mean() const`
  returns the distribution's `mean` parameter;

- `result_type min() const`
  returns 0;

- `result_type max() const`
  returns the maximum value of `result_type`;

18.8.2.14: Student t distribution

The student_t_distribution<RealType = double> is a probability distribution that is used when estimating the mean of a normally distributed population from small sample sizes.

It is characterized by one parameter: the degrees of freedom, which is equal to the sample size - 1.

Defined types:

```
typedef RealType result_type;
```

```
struct param_type
{
    explicit param_type(RealType n = RealType(1));

    RealType n() const;    // The degrees of freedom
};
```

Constructors and members:

- `student_t_distribution<>(RealType n = RealType(1))` constructs a student_t distribution with indicated degrees of freedom.
- `student_t_distribution<>(param_type const &param)` constructs a student_t distribution according to the values stored in the `param` struct.
- `RealType n() const`
  returns the degrees of freedom;
- `result_type min() const`
  returns 0;
- `result_type max() const`
  returns the maximum value of `result_type`;

18.8.2.15: Uniform int distribution

The `uniform_int_distribution<IntType = int>` can be used to select integral values randomly from a range of uniformly distributed integral values.

It has two parameters, `a` and `b`, specifying, respectively, the lowest value that can be returned and the highest value that can be returned.

Defined types:

```
typedef IntType result_type;

struct param_type
{
    explicit param_type(IntType a = 0, IntType b = max(IntType));

    IntType a() const;
    IntType b() const;
};
```

Constructors and members:

- `uniform_int_distribution<>(IntType a = 0, IntType b = max(IntType))`
  constructs a uniform_int distribution for the specified range of values.

- `uniform_int_distribution<>(param_type const &param)` constructs a uniform_int distribution according to the values stored in the `param` struct.

- `IntType a() const`
  returns the distribution's `a` parameter;

- `IntType b() const`
  returns the distribution's `b` parameter;

- `result_type min() const`
  returns the distribution's `a` parameter;

- `result_type max() const`
  returns the distribution's `b` parameter;

18.8.2.16: Uniform real distribution

The `uniform_real_distribution<RealType = double>` can be used to select `RealType` values randomly from a range of uniformly distributed `RealType` values.

It has two parameters, `a` and `b`, specifying, respectively, the half-open range of values (`[a, b)`) that can be returned by the distribution.

Defined types:

```
typedef RealType result_type;

struct param_type
{
   explicit param_type(RealType a = 0, RealType b = max(RealType));

   RealType a() const;
   RealType b() const;
};
```

Constructors and members:

- `uniform_real_distribution<>(RealType a = 0, RealType b = max(RealType))`
  constructs a uniform_real distribution for the specified range of values.

- `uniform_real_distribution<>(param_type const &param)` constructs a uniform_real distribution according to the values stored in the `param` struct.

- `RealType a() const`
  returns the distribution's `a` parameter;

- `RealType b() const`
  returns the distribution's `b` parameter;

- `result_type min() const`
  returns the distribution's `a` parameter;

- `result_type max() const`
  returns the distribution's `b` parameter;

18.8.2.17: Weibull distribution

The `weibull_distribution<RealType = double>` is commonly used in reliability engineering and in survival (life data) analysis.

It has two or three parameters and the two-parameter variant is offered by the STL. The three parameter variant has a shape (or slope) parameter, a scale parameter and a location parameter. The two parameter variant implicitly uses the location parameter value 0. In the two parameter variant the shape parameter (a) and the scale parameter (b) are provided. See http://www.weibull.com/hotwire/issue14/relbasics14.htm for an interesting coverage of the meaning of the Weibull distribution's parameters.

Defined types:

```
typedef RealType result_type;

struct param_type
{
   explicit param_type(RealType a = RealType{ 1 },
                  RealType b = RealType{ 1 });

   RealType a() const;     // the shape (slope) parameter
   RealType b() const;     // the scale parameter
};
```

Constructors and members:

- `weibull_distribution<>(RealType a = 1, RealType b = 1)` constructs a weibull distribution with specified `a` and `b` parameters;

- `weibull_distribution<>(param_type const &param)` constructs a weibull distribution according to the values stored in the `param` struct.

- `RealType a() const`
  returns the distribution's shape (or slope) parameter;

- `RealType stddev() const`
  returns the distribution's scale parameter;

- `result_type min() const`
  returns 0;

- `result_type max() const`
  returns the maximum value of `result_type`;

18.9: The std::experimental/filesystem namespace

Several system calls are usually available in the context of the **C** programming language. Such function (like **rename**(2), truncate(2), **opendir**(2), and **realpath**(3)) are of course also available in **C++**, but their signatures and use are often less attractive, as they expect `char const *` parameters, and may use static buffers or memory allocation based on **malloc**(3) and **free**(3).

Wrappers around these functions have been available since 2003 in the Boost library Currently, **C++** directly supports these facilities in the `std::(experimental::)filesystem` namespace. To use these facilities the header file must be included. In addition, programs using facilities from the `std::(experimental::)filesystem` namespace must be linked against the `stdc++fs` library: `-lstdc++fs`.

Although the `filesystem` namespace currently is nested under the `experimental` namespace, all facilities are available, and it's likely that in due time `experimental' in front of `filesystem` will be dropped.

The `filesystem` namespace is extensive: it offers more than 10 different classes, and over 30 different free functions.

In this and subsequent subsections the notation `fs::` is used to refer to the namespace `std::(experimental::)filesystem`.

18.9.1: File system exceptions: filesystem_error

The std::(experimental::)filesystem namespace offers its own exception type filesystem_error. Its constructors have the following signatures (the bracketed parameters are optional):

```
    filesystem_error(string const &what,
                    [path const &path1, [path const &path2,]]
                    error_code ec);
```

As filesystem facilities are closely related to standard system functions, errc error code enumeration values can be used to obtain error_codes to pass to filesystem_error, as illustrated by the following little demo-program:

```
    #include <iostream>
    #include <experimental/filesystem>

    namespace fs = std::experimental::filesystem;
```

```
using namespace std;

int main()
try
{
   try
   {
      throw fs::filesystem_error{ "exception encountered", "p1", "p2",
                           make_error_code(errc::address_in_use) };
   }
   catch (fs::filesystem_error const &fse)
   {
      cerr << fse.what() << ",\n" <<
            fse.path1() << ",\n" <<
            fse.path2() << ",\n" <<
            fse.code() << '\n';

      throw;
   }
}
catch (exception const &ec)
{
   cerr << ec.what() << '\n';
}
```

18.9.2: Names of file system entries: path

Objects of the class `fs::path` contain names of file system entries. The class `path` is a value-type class: a default constructor (empty path) as well as standard copy/move construction/assignment facilities are available. In addition, the following constructors can be used:

- `path(string &&tmp)`
- `path(Type const &source)`
- `path(InputIter begin, InputIter end)`

These constructors expect character sequences (including NTBSs) in various forms as their arguments. Conceptually, these sequences consist of the following elements (all optional)

- a root-name, e.g., a disk-name (like `E:`) or device indicator (like `//nfs`);
- a root-directory, present if it is the first character after the (optional) root-name;
- filename characters (not containing directory separators). In addition the `single dot filename' (`.`) represents the current directory and the `double dot filename' (`..`) represents the current directory's parent directory;

- directory separators (by default the forward slash). Multiple consecutive separators are treated like one separator.

The constructors also define a last `format ftmp = auto_format` parameter, which probably almost never requires a non-default specification (for this parameter, refer to cppreference.)

Its modifying member functions are

- `path &append(Type const &arg)` or `path &operator/=(Type const &arg)`: the arguments that can be passed to the constructors (including the iterators) can also be passed to these members. Before adding the argument the current and `arg's` contents are separated by a directory separator (unless the resulting path would be absolute if the source path is empty and the argument does not represent an absolute path);

- `void clear()`: the `path's` contents are erased;

- `int compare(Type const &other)`: returns the result of lexicographically comparing the current path's contents with `other`. `Other` can be a `path`, a string-type or a NTBS;

- `path &concat(Type const &arg)` or `path &operator+=(Type const &arg)`: similar to `append`, but no directory separator will be used when adding `arg's` contents to the current `path's` contents;

- `path &remove_filename()`: removes the last component of the stored path. If only a root-directory is stored, then the root directory is removed. Note that the last directory separator is kept, unless it is the only path element;

- `path &replace_extension(path const &replacement = path{} )`: replaces the extension of the last component of the stored path (including the extension's dot) with `replacement`. The extension is removed if `replacement` is empty. If the `path` calling `replace_extension` has no extension then `replacement` is added;

- `path &replace_filename(path const &replacement)`: replaces the last component of the stored path with `replacement`, which itself may contain multiple path elements. If only a root-directory is stored, then it is replaced by `replacement`. The member's behavior is undefined If before calling `replace_filename path.empty()` returns `true`;

Accessors (no arguments, const members) return the path's contents as an NTBS (`c_str`), as a string (`string, wstring, u8string, u16string, u32string`) (possibly prefixed by `gneric_`, like `generic_string`) or as components of path specifications, returned as `path`. Example:

```
fs::path path{ "/usr/local/bin" };
cout << path.string() << '\n';      // shows:   /usr/local/bin
```

Path objects may be inserted into streams using the << (stream insertion) operator, in which case double quotes surround the displayed path name. The double quotes are removed when accessing the path's contents as NTBS or string, and also when assigning (or casting) it to a string.

Path objects may also be extracted from streams using the >> (stream extraction) operator. In this case a path may optionally be surrounded by double quotes. The extracted path again contains its surrounding quotes.

Begin and end iterators can be used to iterate over all of the path's components: each component is returned as a path, root names and root directories are returned as initial components, followed by the individual directory (and final filename) components. The directory separators themselves are not returned when iterating over a path's components.

Decomposers, returning objects (empty if the requested component is not present): root_name, root_directory, root_path, relative_path, parent_path (i.e., the current contents from which the last element has been removed), filename, stem (i.e., the filename without its dot-extension), and extension. Example:

```
    fs::path path{ "/usr/local/bin" };
    cout << path.relative_path() << '\n';   // shows:  "usr/local/bin"
                                            // (note the double quotes)
```

When prefixed by has_ the member returns a bool which is true if the component is present. Also available: is_absolute, is_relative

In addition to the member functions various free operators are available: ==, !=, <, <=, >, and >= comparing two path objects; / returning the contatenated lhs and rhs. Comparisons use lexicographical comparisons (as if by comparing the return values of their string members).

To convert a path (which must refer to an existing directory entry) to its canonical form (i.e., a path not containing . or .. components) the free function canonical (cf. section 18.9.7) can be used:

```
    fs::path path{ "/usr/local/bin/../../share/man" };
    cout << canonical(path) << '\n';    // shows:   "/usr/share/man"
```

18.9.3: Handling directories: directory_entry, (recursive_)directory_iterator

Objects of the class directory_entry contain names and statuses of directory entries. In addition to all standard constructors and assignment operators it defines a constructor expecting a path:

```
    directory_entry(path const &entry);
```

Entry does not have to exist.

Its member functions are:

- void assign(path const &dest):

the current path is replaced by `dest`;

- `void replace_filename(path const &dest)`:

  the last entry of the current path name (which may be empty if the current name ends in a directory separator) is replaced by `dest`;

- `path const &path() const`, `operator path const &() const`:

  returns the current path name;

- `file_status status([error_code &ec])`:

  returns type and attributes of the current path name. If the current path name refers to a symlink, and the symlink's type and status is required, then use `symlink_status` (see also section 18.9.5)

.

Also, `directory_entry` objects may be compared using the `==, !=, <, <=, >,` and `>=` operators, returning the result of applying the operator to their `path` objects.

18.9.4: Visiting directory entries: (recursive_)directory_iterator

The filesystem namespace offers two classes that simplify directory processing: objects of the class directory_iterator are (input) iterators iterating over the entries of directories; and objects of the class recursive_directory_iterator are (input) iterators recursively visiting all entries of directories.

The classes `(recursive_)directory_iterator` support default, copy, and move constructors. Objects of both classes are constructed from a `path` and an optional `error_code`. E.g.,

```
    directory_iterator(path const &dest [, error_code &ec]);
```
All members of standard input iterators (cf. section 18.2) are supported by these classes. Their currently stored path is returned by their dereference operators:
```
    cout << *fs::directory_iterator{ "/home" } << '\n'; // shows the first
                                                        // entry under /home
```

End-iterators matching these objects are the classes' default constructed objects. In addition, `filesystem::begin` and `filesystem::end` are available and are automatically used by range-based for loops. E.g., all entries of the `/var/log` directory are displayed (surrounded by double quotes) by the following statement:

```
    for (auto &entry: fs:directory_iterator("/var/log"))
        cout << entry << '\n';
```
Alternatively, for-statements explicitly defining iterators can also be used. E.g.,
```
    for (
    auto iter = fs:directory_iterator("/var/log"),
         end = fs::directory_iterator{};
           iter != end;
               ++iter
)
    cout << entry << '\n';
```

An `fs::(recursive_)directory_iterator base{"/var/log"}` object represents the first element of its directory. Explicitly defined iterators can be used, like `auto &iter = begin(base), auto iter = begin(base), auto &iter = base` or `auto iter = base`: they all refer to `base's` data, and incrementing them data also advances `base` to its next element:

```
fs::recursive_directory_iterator base{ "/var/log/" };
auto iter = base;
                            // final two elements show identical paths,
                            // different from the first element.
cout << *iter << ' ' << *++iter << ' ' << *base << '\n';
```

The `recursive_directory_iterator` also accepts a `directory_options` argument (see below), by default specified as `directory_options::none`:

```
recursive_directory_iterator(path const &dest,
                        directory_options options [, error_code &ec]);
```

The `enum class directory_options` defines values that can be used to fine-tune the behavior of `recursive_directory_iterator` objects. It supports bitwise operators (the symbols' values are shown between parentheses). Here is an overview of all its symbols:

- `none` (0): directory symlinks are skipped, denied permission to enter a subdirectory generates an error;

- `follow_directory_symlink` (1): symlinks to subdirectories are followed;

- `skip_permission_denied` (2): directories that cannot be entered are silently skipped.

In addition to the members of the class `directory_iterator` the class `recursive_directory_iterator` provides these members:

- `int depth() const`:

  returns the current iteration depth. At the initial directory (specified at construction-time) `depth` returns 0;

- `void disable_recursion_pending()`:

  when called before calling the iterator's increment operator or member, the next entry is not recursed into if it is a sub-directory. Immediately after executing the increment operator recursion is allowed again, so if a recursion should end at a specific depth then this function must repeatedly be called for as long as `depth()` returns that specific depth;

- `recursive_directory_iterator &increment(error_code &ec)`:

  acts identically to the iterator's `operator++()`. However, when an error occurs `operator++` throws a `filesystem_error`, while `increment` assigns the appropriate error to `ec`;

- `directory_options options() const`:

  returns the option that was specified at construction-time;

- `void pop()`:

  ends processing of the current directory, and continues at the next entry in the current directory's parent or ends the directory processing if called at the initial directory;

- `bool recursion_pending() const`:

  `true` is returned if the entry the iterator currently points at is a directory into which directory processing will continue at the iterator's next increment;

Finally, a little program displaying all directory elements of a directory and all its immediate sub-directories:

```
int main()
{

    fs::recursive_directory_iterator base{ "/var/log" };

    for (auto entry = base, end = fs::end(base); entry != end; ++entry)
    {
        cout << entry.depth() << ": " << *entry << '\n';
        if (entry.depth() == 1)
            entry.disable_recursion_pending();
    }
}
```

18.9.5: Types (file_type) and permissions (perms) of file system entries: file_status

Objects of the class `file_status` contain a file system entries' types and permissions. The copy- and move- constructors and assignment operators are available. In addition it defines the constructor

```
explicit file_status(file_type type = file_type::none,
                     perms permissions = perms::unknown)
```

which can also be used as default constructor. In addition it defines the members

- `perms permissions() const` and `void permissions(perms newPerms)`:

  the former member returns the current set of permissions, the latter can be used to modify them;

- `file_type type() const` and `void type(file_type type)`:

  the former member returns the current type, the latter can be used to change the type.

The `enum class file_type` defines the following symbols:

- `not_found = -1` indicates that the file was not found (this is not considered an error);
- `none` indicates that the file status has not been evaluated yet, or an error occurred when evaluating it;
- `regular` a regular file;
- `directory` a directory;
- `symlink` a symbolic link;
- `block` a block device;
- `character` a character device;
- `fifo`: a named pipe;
- `socket`: a socket file;
- `unknown`: unknown file type

The `enum class perms` defines all access permissions of file system entries. The enumeration's symbols were selected so that their meanings should be more descriptive than the constants defined in the `<sys/stat.h>` header file, but other than that they have identical values. Also, all bitwise operators can be used by values of the `enum class perms`. Here is an overview of all its defined symbols:

Symbol	Value	sys/stat.h	Meaning
none	0000		No permission bits were set
owner_read	0400	S_IRUSR	File owner has read permission

```
owner_write   0200    S_IWUSR       File owner has write permission
owner_exec    0100    S_IXUSR       File owner has execute/search
                                    permissions
owner_all     0700    S_IRWXU       File owner has read, write, and
                                    execute/search permissions

group_read    0040    S_IRGRP       The file's group has read permission
group_write   0020    S_IWGRP       The file's group has write permission
group_exec    0010    S_IXGRP       The file's group has execute/search
                                    permissions
group_all     0070    S_IRWXG       The file's group has read, write, and
                                    execute/search permissions

others_read   0004    S_IROTH       Other users have read permission
others_write  0002    S_IWOTH       Other users have write permission
others_exec   0001    S_IXOTH       Other users have execute/search
                                    permissions
others_all    0007    S_IRWXO       Other users have read, write, and
                                    execute/search permissions

all           0777                  All users have read, write, and
                                    execute/search permissions

set_uid       04000   S_ISUID       Set user ID to file owner user ID on
                                    execution
set_gid       02000   S_ISGID       Set group ID to file's user group ID
                                    on execution
sticky_bit    01000   S_ISVTX       POSIX XSI specifies that when set on a
                                    directory, only file owners
                                    may delete files even if the
                                    directory is writeable to
                                    others (used with /tmp)

mask          07777                 All valid permission bits.
```

Here is a little program showing how file statuses can be determined and used:

```
namespace
{
    std::unordered_map<fs::file_type, char const *> map =
    {
        { fs::file_type::not_found, "an unknown file" },
        { fs::file_type::none,      "not yet or erroneously evaluated "
                                                "file type" },
        { fs::file_type::regular,   "a regular file" },
        { fs::file_type::directory, "a directory" },
        { fs::file_type::symlink,   "a symbolic link" },
        { fs::file_type::block,     "a block device" },
        { fs::file_type::character, "a character device" },
        { fs::file_type::fifo,      "a named pipe" },
        { fs::file_type::socket,    "a socket file" },
        { fs::file_type::unknown,   "an unknown file type" }
    };
    void status(fs::path const &path)
    {
        fs::file_status stat = fs::directory_entry{ path }.symlink_status();

        cout << path << " is " << map[stat.type()] << '\n';
    };
} // anon. namespace

int main()
{
    for (auto entry: fs::directory_iterator{"/home/frank"})
        ::status(entry);

    cout << "File status_known is " <<
            status_known( fs::file_status{} ) << '\n';
}
```

18.9.6: Information about the space of a file system entries: space_info

Every existing path lives in a particular file system. File systems can contain certain amounts of data (numbers of bytes) of which some amount already is in use and some amount is still available. These three pieces of information are made available by the function fs::space expecting a fs::path const &, and returning the information in a POD struct fs::space_info. This function throws a filesystem_error, receiving path as its first argument and the operating system's error code as its error_code argument. An overloaded function space expects as its second argument an error_code object, which is cleared if no error occurs, and which is set to the operating system's error code if an error occurred.

The returned `fs::space_info` has three fields:

```
uintmax_t capacity;     // total size in bytes
uintmax_t free;         // number of free bytes on the file system
uintmax_t available;    // free bytes for a non-privileged process
```

If a field cannot be determined it is set to -1 (i.e., the max. value of the type uintmax_t).

Here is a little program illustrating how `space` can be used:

```
int main()
{
    fs::path p{ "/tmp" };

    auto pod = fs::space(p);

    std::cout << "The filesystem containing /tmp has a capacity of " <<
                                        pod.capacity << " bytes,\n"
        "i.e., " << pod.capacity / 1024 << " KB.\n"
        "# free bytes: " << pod.free << "\n"
        "# available:  " << pod.available << '\n';
}
```

18.9.7: Free functions

Several functions are available that directly operate on the current file system.

Functions defining an optional `path const &base` parameter by default use `current_path`.

Some of them define an `error_code &ec` parameter. Those functions have a `noexcept` specification. If those functions cannot complete their task, then `ec` is set to the appropriate error code. Alternatively, `ec.clear()` is called if no error was encountered. If no `ec` argument is provided then those functions throw a `filesystem_error` if they cannot complete their tasks.

The following functions are available:

- `path absolute(path const &src, path const& base)`:

  a copy of `src` to which, unless already available in `src`, `absolute(base)'s` root name and root directory are prepended;

- `path canonical(path const &src [, path const &base [, error_code &ec]])`:

  returns `src's` canonical path (prefixing `base` if `src` is not an absolute path);

- `void copy(path const &src, path const &dest [, copy_options opts [, error_code &ec]])`:

  `src` must exist. Copies `src` to `dest` if the `cp` program would also succeed. Copy options can be specified to fine-tune its behavior: see below for the values that may be specified for the options.
  If `src` is a directory, and `dest` does not exist, `dest` is created. Directories are recursively copied if copy options `recursive` or `none` were specified;

- `bool copy_file(path const &src, path const &dest [, copy_options opts [, error_code &ec]])`:

  `src` must exist. Copies `src` to `dest` if the `cp` program would also succeed. Symbolic links are followed. Copy options can be `skip_existing`: `src` is not copied if `dest` already exists; `overwrite_existing`: a copy is forced; `update_existing`: `src` is copied if it is more recent than `dest`; `True` is returned if copying succeeded;

- `void copy_symlink(path const &src, path const &dest [, error_code &ec])`:

  creates the symlink `dest` as a copy of the symlink `src`;

- `bool create_directories(path const &dest [, error_code &ec])`:

creates each component of `dest`, unless already existing. `True` is returned if no errors were encountered. See also `create_directory` below;

- `bool create_directory(path const &dest [, path const &existing] [, error_code &ec])`:

  creates directory `dest` if it does not yet exist. It is not an error if a directory `dest` already exists. `Dest's` parent directory must exist. If `existing` is specified, then `dest` receives the same attributes as `existing`. `True` is returned if no errors were encountered;

- `bool create_directory_symlink(path const &dir, path const &link [, error_code &ec])`:

  like `create_symlink`, but should be used to create a symbolic link to a directory. See also `create_symlink` below;

- `bool create_hardlink(path const &dest, path const &link [, error_code &ec])`:

  creates a hard link from `link` to `dest`. `Dest` must exist;

- `bool create_symlink(path const &dest, path const &link [, error_code &ec])`:

  creates a symbolic (soft) link from `link` to `dest`; `dest` does *not* have to exist;

- `path current path([error_code &ec])`, `void current_path(path const &toPath [, error_code &ec])`:

  the former function returns the current working directory (cwd), the latter changes the cwd to `toPath`;

- `bool equivalent(path const &path1, path const &path2 [, error_code &ec])`:

  `true` is returned if `path1` and `path2` refer to the same file or directory, and have identical statuses. Both paths must exist;

- `bool exists(path const &dest [, error_code &ec])`, `exists(file_status status)`:

  `true` is returned if `dest` exists (actually: if `status(dest[, ec])` (see below) returns `true`). Note: when iterating over directories, the iterator usually provides the entries' statuses. In those cases calling `exists(iterator->status())` is more efficient than calling `exists(*iterator)`;

- `std::unintmax_t file_size(path const &dest [, error_code &ec])`:

  returns the size in bytes of a regular file (or symlink destination);

- `std::uintmax_t hard_link_count(path const &dest [, error_code &ec])`:

  returns the number of hard links associated with `dest`;

- `file_time_type last_write_time(path const &dest [, error_code &ec])`, `void last_write_time(path const &dest, file_time_type newTime [, error_code &ec])`:

  the former function returns `dest's` last modification time; the latter function changes `dest's` last modification time to `newTime`. The return type `file_time_type` is defined through a `using` alias for `chrono::time_point` (cf. section 20.1.4). The returned `time_point` is guaranteed to cover all file time values that may be encountered in the current file system;

- `void permissions(path const &dest, perms spec [, error_code &ec])`:

  sets `dest's` permissions to `spec`, unless `perms::add_perms` or `perms::remove_perms` was set. The permissions in `perms` are masked using `perms::mask`;

- `path read_symlink(path const &src [, error_code &ec])`:

  `src` must refer to a symbolic link or an error is generated. The link's target is returned;

- `bool remove(path const &dest [, error_code &ec])`, `std::uintmax_t remove_all(path const &dest [, error_code &ec])`:

  `remove` removes the file, symlink, or empty directory `dest`, returning `true` if `dest` could be removed; `remove_all` removes `dest` if it's a file (or symlink); and recursively removes directory `dest`, returning the number of removed entries;

- `void rename(path const &src, path const &dest [, error_code &ec])`:

  renames `src` to `dest`, as if using the standard **mv**(1) command;

- `void resize_file(path const &src, std::uintmax_t size [, error_code &ec])`:

`src's` size is changed to `size` as if using the standard **truncate**(1) command;

- `space_info space(path const &src [, error_code &ec])`:

  returns information about the file system in which `src` is located;

- `file_status status(path const &dest [, error_code &ec])`:

  returns type and attributes of `dest`. Use `symlink_status` if the type and attributes of a symbolic link are required;

- `bool status_known(file_status status)`:

  returns `true` if `status` refers to a determined status (which may indicate that the entity referred to by `status` does not exist). One way of receiving `false` is by passing it a default `file_status: status_known(file_status{})`;

- `path system_complete(path const &src[, error_code& ec])`:

  returns the absolute path matching `src`, using `current_path` as its base;

- `path temp_directory_path([error_code& ec])`:

  returns the path to a directory that can be used for temporary files. The directory is not created, but its name is commonly available from the environment variables `TMPDIR`, `TMP, TEMP`, or `TEMPDIR`. Otherwise, `/tmp` is returned.

File types can be interrogated using the following functions, all supporting the following signatures (where `WHATEVER` is the requested specification):

```
bool is_WHATEVER(file_status status)
bool is_WHATEVER(path const path &dest [, error_code &ec])
```

all functions return `true` if `dest` or `status` matches the requested type. Here are the available functions:

- `is_block_file`: the path refers to a block device;

- `is_character_file`: the path refers to a character device;

- `is_directory`: the path refers to a directory;

- `is_empty`: the path refers to an empty file or directory;

- `is_fifo`: the path refers to a named pipe;

- `is_other`: the path does not refer to a directory, regular file or symlink;
- `is_regular_file`: the path refers to a regular file;
- `is_socket`: the path refers to a named socket;
- `is_symlink`: the path refers to a symbolic link;

The `enum class copy_options` define symbolic constants that can be used to fine-tune the behavior of the `copy` and `copy_file` functions. The enumeration supports bitwise operators (the symbols' values are shown between parentheses). Here is an overview of all its defined symbols:

Options when copying files:

- `none` (0): report an error (default behavior);
- `skip_existing` (1): keep the existing file, without reporting an error;
- `overwrite_existing` (2): replace the existing file;
- `update_existing` (4): replace the existing file only if it is older; than the file being copied;

Options when copying subdirectories:

- `none` (0): skip subdirectories (default behavior);
- `recursive` (8): recursively copy subdirectories and their content;

Options when copying symlinks:

- `none` (0): follow symlinks (default behavior);
- `copy_symlinks` (16): copy symlinks as symlinks, not as the files they point to;
- `skip_symlinks` (32): ignore symlinks;

Options controlling `copy's` behavior itself:

- `none` (0): copy file content (default behavior);
- `directories_only` (64): copy the directory structure, but do not copy any non-directory files;

- `create_symlinks` (128): instead of creating copies of files, create symlinks pointing to the originals (the source path must be an absolute path unless the destination path is in the current directory);
- `create_hard_links` (256): instead of creating copies of files, create hardlinks that resolve to the same files as the originals.

- Table of Contents
- Previous Chapter
- Next Chapter