by Leedehai

Chapter 16(1) A Glimpse into The Standard Template Library

Good references:

Chapter 16 and Appendices F, G, *C++ Primer Plus*, 6th edition, by Stephen Prata

Part II, *C++ Primer*, 5th edition, by Stanley Lippman, et al.

The C++ Standard Library A Tutorial and Reference, 2nd edition, by Nicolai Josuttis.

Golden rule: **using makes familiar, practice makes perfect.**

16.1 What is STL?

- The *Standard Template Library* (STL) provides a collection of templates representing containers, iterators, function objects, and algorithms, and is part of the standard library that comes with C++.
    - A *container* is a unit that can hold several values, like an array. STL containers require homogeneous; that is, a container hold values all of the same kind.
    - A *iterator* is an object that let you move through a container mush as pointers let you move through an array; it is a generalization of a pointer.
    - A *function object* is an object that act like a function; they can be class objects or function pointers (including function names since a they act like pointers).
    - A *algorithm* is a recipe for accomplishing particular tasks, including sorting, searching, randomizing, etc.
- Alex Stepanov and Meng Lee developed STL at Hewlett-Packard Laboratories, releasing the implementation in 1994. The ISO/ANSI C++ committee voted to incorporated it into the C++ standard.
- The STL represents a programming paradigm called *generic programming*, in that it does not specify a particular type or types; rather, it consists of templates so as to accommodate any type.
- Object-oriented programming concentrates on the data aspect of programming, whereas generic programming concentrates on algorithms and pay little attention to data types. The main things these two paradigms have in common is that they emphasize on abstraction and code reusability.

16.2 The `string` class

- Constructing a string (NBTS: null-byte-terminated string, a.k.a. the C-style string, which is terminated by one or more trailing `\0`).
    - `string(const char *s)` - Initializes a `string` object to the NBTS pointed to by `s`.
    - `string(size_type n, char c)` - Creates a `string` object of `n` elements, each initialized to the same character `c`.
    - `string(const string &str)` - The copy constructor, initializes a `string` object to `str`.
    - `string()` - The default constructor, creates a 0-size `string` object.
    - `string(const char *s, size_t n)` - Initializes a `string` object to the NBTS pointed to by `s` and contains the first `n` characters.
    - `template<class Iter> string(Iter begin, Iter end)` - Initializes a `string` object to the values in the half-open range [`begin`, `end`), where `begin` and `end` act like pointers and specify locations.

    - `string(const string &str, size_t pos, size_t n = npos)` - Initializes a `string` object to the object `str`, starting at position `pos` in `str` and going to the end of `str` or using `n` characters, whichever comes first. `npos` is the `static` member constant representing the maximum value of a `size_t`.
    - `string(string &&str) noexcept` - The move constructor, initializes a `string` object to the `string` object `str`; `str` may be altered.
    - `string(initializer_list<char> il)` - Initializes a `string` object to the characters in the initializer list `il`, e.g. `string Lang {'A','d','a'};`.
- The `string` class input
    - For C-style strings, <u>within the pre-specified size</u>, you have three options:

```cpp
char info[50];
cin >> info;          // read up to a white space, leave it in queue
cin.getline(info,50);// read up to '\n', discard it from queue
cin.get(info,50);     // read up to '\n', leave it from queue
```

    - For `string` objects, which supports automatic sizing within the maximum allowable size (`string::npos` and memory allocable), you have two options:

```cpp
string stuff;
cin >> stuff;         // read up to a white space, leave it in queue
getline(cin,stuff);   // read up to '\n', discard it from queue
```

    - Both versions of `getline()` allow for an optional argument that specifies which character to use to delimit input:

```cpp
cin.getline(info, 50, ':');  // read up to ':', discard it from queue
getline(stuff,':');          // read up to ':', discard it from queue
```

    - The `getline()` function for the `string` class reads characters from the input and stores them in a `string` object until at least one of the following criteria is met:
        - The end-of-file is encountered, in which case `eofbit` of the input stream is set, and both the `fail()` and `eof()` methods will return `true`.
        - The delimiting character (`\n` by default) is reached, in which case it is not stored in the object and removed from the input stream.
        - The maximum allowable number of characters (the lesser of constant `string::npos` and the number of bytes in memory allocable) is read, in which case `failbit` of the input stream is set, and `fail()` method returns `true`.
    - The `operator>>()` function for the `string` class behaves similarly, except that instead of reading to and discarding a delimiting character, it reads up to a white space character and leaves it in the input queue.

        > An input stream object has an accounting scheme to keep track of the error state of the stream. In this scheme, setting `eofbit` registers detecting the end-of-file; setting `failbit` registers detecting an input error; setting `badbit` registers some unrecognized failure, such as a hardware failure; and setting `goodbit` indicates that all is well.

    - A `string` object does NOT take trailing `\0`s as C-style strings do.

- Working with strings
  - You can compare strings: `==`, `!=`, `<` and `>` (according to the encoding system the compiler adopts, most likely ASCII) are overloaded operators.
  - You can determine the size of a string, by using either `size()` or `length()`.

    > These two methods do the same job.

  - You can search for a given character or substring in a variety of ways:
    - `size_t find(const string &str, size_t pos = 0)` - Finds the first occurrence of the substring `str`, starting the search at location `pos` in the invoking string. Returns the index of the first character of the substring if found, and returns `string::npos` otherwise.
    - `size_t find(const char *s, size_t pos = 0)` - Finds the first occurrence of the substring `s`, starting the search at location `pos` in the invoking string. Returns the index of the first character of the substring if found, and returns `string::npos` otherwise.
    - `size_t find(const char *s, size_t pos = 0, size_t n)` - Finds the first occurrence of the substring consisting of the first `n` characters in `s`, starting the search at location `pos` in the invoking string. Returns the index of the first character of the substring if found, and returns `string::npos` otherwise.
    - `size_t find(char ch, size_t pos = 0) const` - Finds the first occurrence of the character `ch`, starting the search at location `pos` in the invoking string. Returns the index of the character if found, and returns `string::npos` otherwise.
    - There are also other related methods `rfind()` (finds the last occurrence), `find_first_of()` (finds the first occurrence of any of the character in the argument), `find_last_of` (finds the last occurrence of any of the character in the argument), `find_first_not_of()` (finds the first character that is not a character in the argument), `find_last_not_of()` (finds the last character that is not a character in the argument).
  - There are even more facilities offered by `string`.
- String varieties
  - The string library is actually based on a template class called `basc_string`: `template<class charT, class traits = char_traits<charT>, class Allocator = allocator<charT>> basic_string {...};`
  - The `basic_string` template comes with four explicit instantiations, each of which has an alias for convenience:

```
typedef basic_string<char> string;        // the commonly used.
typedef basic_string<wchar_t> wstring;
typedef basic_string<char16_t> u16string;
typedef basic_string<char32_t> u32string;
```

16.3 Smart pointer template classes

- A *smart pointer* is a class object that acts like a pointer but has additional features. There are four smart pointer templates: `auto_ptr` (deprecated in C++11), `unique_ptr`, `shared-ptr`, and `weak-ptr`. Each defines a pointer-like object intended to be assigned an address obtained by `new`. They are declared as part of the `std` namespace in the `<memory>` header.
  - `shared_ptr`: smart pointer that provides shared ownership. The object is deleted when the last `shared_ptr` pointing to that object is destroyed. It uses *reference count* mechanism to count how many users share a common object.

    - `unique_ptr`: smart pointer that provides single ownership. The object is deleted when the `unique_ptr` pointing to that object is destroyed. `unique_ptr` CANNOT be directly copied or assigned.
    - `weak_ptr`: smart pointer that points to an object managed by a `shared_ptr`. The `shared_ptr` does NOT count `weak_ptr`s in its reference count.
- When a smart pointer expires, its destructor uses `delete` to free the memory. Thus, if you assign an address returned by `new` to one of these smart pointers, <u>you do not have to remember to free the memory later</u>; it will be freed automatically when the smart pointer object expires.
- To create one of these smart pointer objects, you just need to use usual template syntax to instantiate the template and declare an object. For instance, the `auto_ptr` template is declared as:

```cpp
template<class X> class auto_ptr { ...
public: explicit auto_ptr(X *p = 0); ...
};
```

  Thus, an `auto_ptr` object of type `X` should be declared in this manner:

```cpp
auto_ptr<double> pd(new double);
auto_ptr<std::string> ps(new std::string);
auto_ptr<std::string> ps(new std::string("happy and joy"));
```

- A smart pointer acts like a real pointer. Naturally it overloads `*` so that you can use it to access the pointed-to object, and overloads `->` so that you can access the pointed-to object's members, if any.

```cpp
*pd = 2.5; int theSize = ps->size();
```

- However, you need to be aware of the restriction that a smart pointer CANNOT point to a non-heap memory block. When a smart pointer expires, its destructor will use `delete`/`delete[]` to free the pointed-to memory, and you certainly cannot free a non-heap memory block with `delete`/`delete[]`.

```cpp
std::string st("Customers' likes and dislikes make or break a product");
auto_ptr<std::string> ps(&str); // a runtime error when "ps" expires
```

- Choosing a smart pointer
    - If the program uses more than one pointer to an object, you should ise `shared_ptr`. Many STL algorithms include copy or assignment operations that worl with `shared_ptr` but NOT with `unique_ptr` (a compiler warning) or `auto_ptr` (undefined behavior).
    - If the program does not need multiple pointers to the same object, `unique_ptr` works.
    - Operations common to both `shared_ptr` and `unique_ptr`:
        - `shared_ptr<X> sp` and `unique_ptr<X> up` - null smart pointer that can point to objects of type `X`;
        - `p` - use `p` as a condition, as in `if(p)` and `while(p)`; `true` if `p` points to something other than null;
        - `*p` - dereference `p` to access the object to which `p` points;
        - `p->some_member` - synonym for `(*p).some_member`;
        - `p.get()` - returns the pointer in `p`; it should be used with caution, since the object to which

the returned pointer points will disappear when the smart pointer deletes it;
    - `swap(p,q)` and `p.swap(q)` - swaps the pointers in `p` and `q`.
  - Operations specific to `shared_ptr`:
    - `make_shared<X> (args)` - returns a `shared_ptr` pointing to a dynamically allocated object of type `X`; uses `args` to initialize that object;
    - `share_ptr<X> p(q)` - `p` is a copy of the `shared_ptr` `q`; increments the reference count in `q`; the pointer in `q` must be convertible to `X *`;
    - `p = q` - `p` and `q` are `shared_ptr`s holding pointers that can be converted to one another; decrements `p`'s reference count and increments `q`'s reference count;
    - `p.unique()` - returns `true` if `p.use_count()` is one; `false` otherwise;
    - `p.use_count()` - returns the number of objects sharing with `p`; may be a slow operation so it is intended primarily for debugging purposes.
  - For more details, see *C++ Primer*, 5th edition, by Stanley Lippman, et al.

16.4 The `vector` template class

- A `vector` object is a container. The template class is declared in `<vector>` as part of the `std` namespace.
- To create a `vector` object, you use `<type>` notation to instantiate the class template and declare an object, followed by a parentheses-pair enclosing the number of units, e.g. `std::vector<double> vec(5);` creates a vector containing 5 `double`s. The number of units does NOT need to be a constant.
- Things to do to `vector`s
  - All STL containers provide certain basic methods, such as `size()`, `swap()` (exchanges the contents of two containers), `begin()` (returns an iterator that refers to the first element in a container), and `end()` (returns an iterator that represents the past-the-end, namely, the element one past the last element, for the container).
  - The `<vector>` template class also has some methods that only some STL containers have, such as `push_back()` (adds an element to the end), `erase()` (removes a given close-open range of a vector), `insert()` (insert an element into the vector).
  - Suppose you have a function `showReview()` that takes in a dereferenced pointer to a `books` vector's element, and displays its content (a `Review` class object):

    ```
    std::vector<Review>::iterator pr;
    for (pr = books.begin(); pr != books.end(); pr++) showReview(*pr);
    ```

    You can use STL's `for_each()` (declared in `<algorithm>`), which works with any container types, to apply the designated function `showReview()` to each element in the given range. Here the function's pointer (its name) is passed to it:

    ```
    std::for_each(books.begin(), books.end(), showReview);
    ```

  - Function `random_shuffle()` in STL (declared in `<algorithm>`) rearranges elements in a given ranger in random order. Unlike `for_each()`, this function requires the container type allow random access `vector` supports it):

```
std::random_shuffle(books.begin(), books.end());
```

- STL function `sort()` (declared in `<algorithm>`), too, requires random access container types. It comes in two versions. The first version takes two iterators that define a range, and it sorts that range by using the `<` operator defined for the type element stored in the container, and it can be a built-in `<` operator or a overloaded `<` operator. The second version takes three arguments. The first two, again, are iterators that specify a range; the third is a function pointer that is used to replace `operator<()` for making the comparison, and that function's return value should be `bool`, with `false` meaning the two operand are in the wrong order.

```
std::sort(books.begin(), books.end()); // use `<`
std::sort(books.begin(), books.end(), betterThan);
```

- The "function pointers" used by `for_each()` and `sort()` above, do not need to be real pointers; generally they are *functor*s, as discussed in 16.8. And these three STL "functions" discussed above are examples of the STL's predefined algorithms, as discussed in 16.9, their real identities being function templates.

- For more containers, see *The C++ Standard Library A Tutorial and Reference*, 2nd edition, by Nicolai Josuttis.

16.5 Iterators: generalization of pointers

- Just as templates make algorithms independent of the type of data stored in a container, iterators make algorithms independent of the type of container itself.

> An example: finding a given element in an array and in a linked list certainly requires two different implementations - one uses array indexing to move through a list of items, and the other uses pointers to navigate. Thus you cannot generalize these two with one single template since their differences is not in data types. More broadly speaking, however, the thoughts behind these two are the same - sequentially compare the given element with each item in the container. That is where an iterator comes in.

- Four basic functionalities (I.D.E.A.) an iterator should have AT LEAST:
  - **Incrementation:** You should be able to move an iterator through all the elements of a container; that is, if `p` is an iterator, `p++` and `++p` should be defined and defined properly so as to traverse the container.
  - **Dereference:** You should be able to dereference an iterator in order to access the value to which it refers; that is, if `p` is an iterator, `*p` is defined;
  - **Equality Comparison:** You should be able to compare one iterator to another for equality; that is, if `p` and `q` are iterators, `p == q` and `p != q` are defined;
  - **Assignment:** You should be able to assign one iterator to another; that is, if `p` and `q` are iterators, `p = q` is defined;

  > There are more extra things an iterator is capable of. In reality the STL has several levels of iterators of increasing capabilities.
  >
  > An regular pointer meets the criteria above as an iterator. So an ordinary array's iterator is just a pointer.

> You can design your own iterator as well, by declaring a class possessing these capabilities as outlined above, utilizing operator overloading.

- How do the STL implement iterators?
    - Each container class (`vector` , `list` , etc.) defines an `iterator` type appropriate to the class: for one class, the iterator might be a pointer; for anther, it might be a class object. Whatever it is, the iterators need to have necessary operators such as `*` and `++` as listed above, and some may need more to accomplish more sophisticated tasks such as random access.
    - Each container class has a past-the-end marker, which is the value assigned to an iterator if it has been incremented one past the last item in the container.
    - Each container class has `begin()` and `end()` methods that return iterators to the first element in a container and to the past-the-end position. An *iterator range* corresponds to the close-open range [`begin` , `end`).
    - Each container has `++` operation take an iterator from the first element to past-the-end, visiting every container element en route (*traversal*).
- To use a container class, you do not need to know how its iterators are implemented nor how past-the-end is implemented.
    - For instance, suppose you want to print the values in a `vector<double>` class object:

    ```
    std::vector<double>::iterator pr;
    for (pr = myRecord.begin(); pr != myRecord.end(); pr++)
      std::cout << *pr << std::endl;
    ```

    - Suppose you want to print values in a `list<int>` class object instead:

    ```
    std::list<int>::iterator pr;
    for (pr = myRecord.begin(); pr != myRecord.end(); pr++)
      std::cout << *pr << std::endl;
    ```

    - With C++ automatic type deduction, you can even disregard the container type:

    ```
    // "myRecord" might be a vector, list, ... who cares?
    for (auto pr = myRecord.begin; pr != myRecord.end(); pr++)
      std::cout << *pr << std::endl;
    ```

    Plus, C++11 provides a even more concise way: range-based `for` loop (see 5.2):

    ```
    for (auto x : myRecord) std::cout << x << std::endl;
    ```

    - To summarize the STL approach of using iterator: you start with an algorithm for processing a container. You express the algorithm in as general term as possible, making it independent of data type (`int` , `SomeClass` , etc.) and container type (`vector` , `YourContainer` , etc.). To make the general algorithm work with specific cases, you define iterators that meet the needs of the algorithm and place requirements on the container design.

> Iterators are not confined to the STL containers; you can design your own iterator type (at least meets the I.D.E.A. criteria above) for your own containers. Sometimes the design of iterators can be not-so-trivial, e.g. an iterator type for a tree structure or a graph structure container.

- Kinds of iterators
    - Different algorithms have different requirements for iterators.

      > For instance, a finding algorithm needs the `++` operator so the iterator can step through the entire container (sequential access), and it just needs to permission to read, not write data. But a sorting algorithm, requires random access by defining the `+` operator so expressions like `iter + 10` is valid, and also the iterator needs to be able to both read and write data.

    - The STL defines five kinds of iterators and describes its algorithms in terms of which kinds of iterators it needs. The five kinds are: *input iterator*, *output iterator*, *forward iterator*, *bidirectional iterator*, and *random access iterator*.

      > For example, the `find()` prototype looks like this:
      >
      > ```cpp
      > template <class InputIterator, class T>
      > InputIterator find(InputIterator first, InputIterator last,
      >                    const T & value);
      > ```
      >
      > This tells you that this algorithm requires an input iterator. Similarly, the following prototype tells you that it requires a random access iterator:
      >
      > ```cpp
      > template <class RandomAccessIterator>
      > void sort(RandomAccessIterator first,
      >           RandomAccessIterator last);
      > ```

    - As with pointers, all five kinds of iterators can be dereferenced (using `*`) and can be compared for equality (using `==`) and inequality (using `!=`). If two iterators test as equal (`p` == `q`), then the results of dereferencing them should be the same (`*p` == `*q`).
    - Input iterators: <u>suitable for single-pass, read-only algorithms</u>
        - It can be used by a program to <u>read in values from a container</u>, but it need not allow a program to alter the value in the container.
        - And it supports `++` (both in prefix and postfix form) so as to <u>traverse a container sequentially in one direction</u>.
        - There is no guarantee that traversing a container a second time with an input iterator will move through the values in the same order; also after an input iterator has been incremented, there is no guarantee that its prior value can still be dereferenced. So algorithms using it should be single-pass.
    - Output iterators: <u>suitable for single-pass, write-only algorithms</u>
        - It can be used by a program to <u>write out values into a container</u>, but it need not allow a program to read in data from the container (like `cout`, which can modifies the stream of character sent to the screen but cannot read what is on screen.)
        - And it supports `++`, as with input iterators.
        - There is no guarantee that traversing a container a second time with an input iterator will move through the values in the same order; also after an input iterator has been

incremented, there is no guarantee that its prior value can still be dereferenced. So it suits single-pass algorithms.

- Forward iterators: suitable for single/multiple-pass, R/W algorithms
    - It can be used by a program to both read and write data.
    - And it supports `++`.
    - Unlike input and output iterators, it necessarily goes through a sequence of values in the same order each time you use it; after you increment it, you can still dereference the prior iterator value, you can still dereference the prior iterator value and get the same result. These properties make multiple-pass algorithms possible.
- Bidirectional iterators: suitable for single/multiple-pass, R/W algorithms
    - It can be used by a program to both read and write data.
    - And it supports `++` and `--`.
    - As with forward iterators, it necessarily goes through a sequence of values in the same order each time you use it; after you increment it, you can still dereference the prior iterator value and get the same result.
- Random access iterators: suitable for single/multiple-pass, R/W algorithms
    - It can be used by a program to both read and write data.
    - It supports `+`, `-`, `++`, `--`, `+=`, `-=`, `[]`, `<`, `>`, `<=`, `>=` so it permits random access (jump directly to an arbitrary element).
    - As with forward iterators, it goes through a sequence of values in the same order each time you use it; after you increment it, you can still dereference the prior iterator value and get the same result.

- Iterator hierarchy: a conceptual hierarchy can be summarized in according to an iterator's versatility.

capabilities	input	output	forward	bidirectional	rand. ac.
dereferencing read	yes	**NO**	yes	yes	yes
dereferencing write	**NO**	yes	yes	yes	yes
fixed traversal order	**NO**	**NO**	yes	yes	yes
`++i` `i++`	yes	yes	yes	yes	yes
`--i` `i--`	**NO**	**NO**	**NO**	yes	yes
`i[n]` (viz. `*(i+n)`)	**NO**	**NO**	**NO**	**NO**	yes
`i+n` `i-n` `i+=n` `i-=n`	**NO**	**NO**	**NO**	**NO**	yes

    - Note that the various iterator kinds are not defined types; rather, they are conceptual characterizations. As mentioned earlier, each container class defines a class scope `typedef` name called `iterator`. So the `vector<int>` class has iterators of type `vector<int>::iterator`. The documentation for this class would tell you that these iterators are random access iterators.

- The `<iterator>` header provides some predefined iterator types, including `ostream_iterator`, `istream_iterator`, `reverse_iterator`, `back_insert_iterator`, `front_insert_iterator`, and `insert_iterator`. These predefined iterators expand the generality of the STL.

- The iterator returned by the `end()` method is called the *off-the-end iterator*; the iterator returned by the `before_begin()` method is called the *off-the-beginning iterator*; it refers to the nonexistent "pseudo" element one before the first element and it is introduced to accommodate some algorithms (see your algorithm notes). `before_begin()` is not necessarily possessed by every container type.
- An example illustrating how iterators are used. There is an algorithm called `std::copy()` declared in `<algorithm>`, in the form of a function template:

  ```
  template <class InputIterator, class OutputIterator>  OutputIterator copy(InputIterator first, InputIterator last, OutputIterator result);
  ```

  Suppose we want to to copy an array into a vector. Of course, you can write your own function to perform the copying work. But it would be nice to utilize the STL algorithm directly:

  ```
  const int N = 10;
  int fibo[N] = {0,1,1,2,3,5,8,13,21,34}; std::vector<int> obif(N);
  std::copy(fibo, fibo+N, obif.begin()); // copy array to vector
  ```

  The first two iterator arguments to `copy()` represent a range to be copied, and the last iterator argument represents the location to which the first element is copied. The first two arguments must be input iterators (or better), and the final argument must be an output iterator (or better). The `copy()` function overwrites existing data in the destination container, and it assumes the container's size is big enough.

  > What if you want to copy the array's data to the display instead of to a vector? Another iterator for the last argument of `copy()` is needed, since a display's iterator is certainly not the same as a vector's. For more details, see 16.6.

16.6 STL predefined iterators

- The STL provides several predefined iterator types in `<iterator>` header file.
- `ostream_iteraotr` template class, for output streams
  - Using STL terminology, this template is a *model*, namely an implementation, of the *concept* of output iterators. It is also an example of *adapter* - a class or function that converts some other interface to an interface used by the STL.
  - You can instantiate an `ostream_iterator` class and declare an object like this:

    ```
    std::ostream_iterator<int, char> out_iter(std::cout, " ");
    ```

    - The `out_iter` iterator now becomes an interface that allows you to use `cout` to display information.
    - The first template argument `int` indicates the data type being sent to the output stream; and the second one, `char`, indicate the character type used by the output stream (I/O streams uses `char` or `wchar_t`).
    - The first constructor argument `std::cout` identifies the output stream being targeted (the argument slot could also be filled with a file output steam); and the second one designates a separating string (in this example, a blank space) to be displayed after each element sent to the output stream.

- You could use the iterator like this: `*out_iter++ = 18;`, which works like `std::cout << 18 << " ";`. For a regular pointer this would mean assigning the value `18` to the pointed-to object and then incrementing the pointer. For this iterator, however, it means sending `18` and then a blank space to the output stream (in this case, the display), and preparing the output stream for the next output operation.
- Following the example given at the end of 16.5, you could use the iterator for display with `std::copy()` (declared in `<algorithm>`) as follows:

  ```
  std::copy(fibo, fibo + N, out_iter); // array to display
  std::copy(obif.begin(), obif.end(), out_iter); // vector to display
  ```

  Or, you can skip creating a named iterator and construct an anonymous one:

  ```
  std::copy(fibo, fibo + N,
            std::ostream_iterator<int, char>(std::cout, " "));
  ```

* `istream_iterator` template class, for input streams
  - Using STL terminology, this template is a *model* of the *concept* of input iterators. It is also an example of *adapter*.
  - Similarly, You can instantiate and declare an `istream_iterator` object. You could use two `istream_iterator` objects to define an input range for `copy()`:

    ```
    std::copy(istream_iterator<int, char>(std::cin),
              istream_iterator<int, char>(), obif.begin());
    ```

    - The first template argument `int` indicated the data type to be read; and the second one, `char`, indicates the data type used by the input stream.
    - Using a constructor argument of `std::cin` means to use the input stream managed by `std::cin`, namely the display; omitting the constructor argument indicate input failure, so the code above means to read from the input stream until end-of-file, type mismatch, or some other input failures.
* Other useful iterators in `<iterator>`
  - In addition to `ostream_iterator` and `istream_iterator` as discussed before, there are `reverse_iterator`, `back_insert_iterator`, `front_insert_iterator`, and `insert_iterator`.
  - `reverse_iterator`: incrementing a reverse iterator causes it to move backward through a sequence - it exchanges the meaning of `++` and `--`.
    - Suppose you want to display the vector `obif`'s contents on the screen in a reverse order. Then the code looks like this, and you do not even have to declare a reverse iterator here:

      ```
      std::copy(obif.rbegin(), obif.rend(), out_iter);
      // rbegin() returns a reverse iterator pointing to the
      // past-the-end element, and rend() returns a reverse iterator
      // pointing to the first element.
      ```

        - Reverse pointers have to make a special compensation. Suppose `rp` is a reverse pointer initialized to `obif.rbegin()`. What should `*rp` be? Because `rbegin()` returns `past-the-end`, you should NOT try to dereference that address. Similarly, if `rend()` is really the location of the first element, `copy()` stops one location earlier because the end of the range is not in a range (discussed in 16.5). Reverse pointers solve both problems by decrementing first and then dereferencing. That is, `*rp` dereferences the iterator value immediately preceding the current value of `*rp`. If `rp` points to position six, `*rp` is the value of position five, and so on.
    - The three inset iterators, or collectively *inserter*s, converts a copying process to an insertion process; unlike copying, insertion adds new elements without overwriting existing data, and it uses automatic memory allocation to ensure that the new data fits.

      > You can use an `insert_iterator` to convert an algorithm that copies data into one that inserts data:
      >
      > ```
      > std::copy(fibo, fibo + N,
      >           std::insert_iterator<vector<int>>(obif,obif.begin()));
      > ```
      >
      > The reason you have to declare the container type `vector<int>` is that the iterator has use appropriate methods of the container. The first constructor argument `obif` indicates that the iterator is intended for a container named `obif`, and the second argument `obif.begin()` specifies the insertion location.

        - A `back_insert_iterator` inserts items at the end of a container, a `front_insert_iterator` inserts items at the front, and the `insert_iterator` inserts items in front of the location specified as an argument to the `insert_iterator` constructor.
        - There are restrictions, though. A `back_insert_iterator` can be used ONLY with containers that allow rapid back insertion (*rapid* refers to constant time complexity), such as `vector`s. A `front_insert_iterator` can be used ONLY with containers that allow rapid front insertion. The `insert_iterator` does not have these restrictions.

16.7 Containers

- The STL containers have two conceptual categories: *sequential container*s and *associative container*s.
- Basic container concept: no type corresponds to the basic container concept; rather, it describes elements common to all the container categories.
    - A *container* is an object that stores other objects, which are all of a single type. The stored objects can be class objects or built-in type objects.
    - Data stored in a container is *owned* by the container; that is, when a container expires, so does the data stored in it.

      > NOTE that if the stored data are pointers, the pointed-to data does not necessarily expire, since the pointed-to data is not owned by the container.

    - You cannot just store any kind of object in a container. The stored data should be of type that is *copy constructable* and *assignable*. Built-in types and class types satisfy these requirements, unless the class declaration makes one or both of the copy constructor and the assignment operator non-`public`. C++11 refines the requirements, with terms such as *copy insertable* and *move insertable*.

    - The basic container concept does NOT guarantee that the elements are stored in any particular order, or that the order is fixed.
    - Container operations: see p. 330, *C++ Primer*, 5th edition, by Stanley Lippman, et al.
- Container operations may invalidate iterators
    - Operations that add or remove elements from a container might invalidate pointers, references, or iterators, to container elements. An invalidated pointer, reference, or iterator is a serious runtime error source that is likely to lead to the same kinds of problems as using a uninitialized pointer.

      > Pointers, references, and iterators, are essentially addresses. Memory reallocation changes elements' addresses.
      >
      > In this note, pointers, references, and iterators, are collectively referred to as "generalized iterators".

    - After an operation that adds elements to a container,
        - Generalized iterators to a `vector` or `string` are invalid if the container was reallocated. If no reallocation happens, iterators before the insertion point remain valid.
        - Iterators to a `deque` are invalid if elements are added anywhere but at the front or back. If elements are added at the front or back, iterators are invalidated, but references and pointers to still valid.
        - Generalized iterators to a `list` or `forward_list` remain valid.
    - After an operation that removes elements from a container,
        - Generalized iterators to a `list` or `forward_list` remain valid.
        - Generalized iterators to a `deque` are invalidated if the removed elements are anywhere but the front or back. If elements at the front or back are removed, they remain valid, excluding the off-the-end iterator.
        - Generalized iterators to a `vector` or `string` remain valid for elements before the removal point.
    - When you use an generalized iterator, it is a good idea to minimize the part of program during which an iterator must stay valid. Because code that adds or removes elements to a container can invalidate generalized iterators, you need to ensure that these generalized iterator is repositioned before using them again. This advice is especially important for `vector`, `string`, and `deque`.
- Sequential containers (sequences)
    - The *sequence* is an important refinement of the basic container concept because several of the STL container types are sequences: `deque`, `forward_list`, `list`, `queue`, `priority_queue`, `stack`, `array` and `vector`.
    - The sequence requires that (1) the iterator be at least a forward iterator (or better in the hierarchy), (2) elements be arranged in a definite order so that it does not change from one cycle of iteration to the next, (3) elements be arranged in a strict linear order; that is, there is a first element and a last element, and each element but these two has exactly one element immediately ahead of it (immediate predecessor) and one element immediately after it (immediate successor).
    - `vector` declared in `<vector>` (a class representation of an array)
        - Elements in a `vector` can be accessed by their positional index (random access).
        - Elements can be efficiently added to the back or removed from the back.
        - Adding elements to a `vector` might cause it to be reallocated, invalidating all iterators into

the vector.
    - Adding or removing an element in the middle of a `vector` invalidates all iterators after the insertion/deletion point.
  - `deque` declared in `<deque>` ("double-end queue", because elements can be added to both the front and the back)
    - Elements in a `deque` can be accessed by their positional index (random access).
    - Elements can be added or deleted at the end or the back and does not reallocate existing elements.
  - `list` declared in `<list>` (a doubly linked list)
    - Elements in a `list` may be accessed only sequentially; starting from a given element, any element can be access by traversing each element in between.
    - Iterators on `list` supports both `++` and `--`.
    - Supports fast insertion and deletion anywhere in the `list`.
    - Iterators remain valid when new elements are added; when an element is removed, only the iterator to that element is invalidated.
  - `forward_list` (C++11): has the same facilities as `list` except that it represents a singly-linked list and thus the iterator is monodirectional, not bidirectional.
  - `queue` declared in `<queue>` (adapter) (represents a FIFO container)
    - Much more restrictive than `deque`.
    - Permits adding elements to the back and removing elements from the front.
  - `priority_queue` declared in `<queue>` (adapter) (represents a to-some-extent-FIFO container)
    - It supports the same facilities as `queue`.
    - The main difference is that the "largest" element, instead of the earliest-added one, gets moved to the front of the queue. The criteria of "being large" is according to numeric value by default: `priority_queue<int> pq;`, but it can be customized using `great<>()` function: `priority_queue<int> pq(greater<int>);`.
  - `stack` declared in `<stack>` (adapter) (represents a FILO container)
    - Permits adding elements and removing elements from one end only.
  - `array` (C++11) declared in `<array>` (represents a fixed-size array, just like an built-in array)
    - Elements in an `array` can be accessed by their positional index.
    - Supports fast random access.
- Associative containers
  - An *associative container* is another refinement of the container concept. It associates a value with a key, and uses the key to find the value.

    > For instance, the values could be structure variables representing employee information such as name, address, office number and phone number, etc., and the key could be a unique employee number. It is a bit like a database.

  - The strength of an associative container is that it provides rapid access to its elements. Like a sequence, an associative container allows you to insert new elements; however, you CANNOT specify a particular location for the inserted elements. The reason is that an associative container usually has a particular algorithm for determining where to place data so that it can retrieve information quickly.

   - Associative containers typically are implemented using some form of tree, a branching structure, making it relatively simple to add or remove a new data item, much as with a linked list. Compared to a list, a tree offers much faster search times.
   - The STL provides four associative containers: `set`, `multiset`, `map`, `multimap`.
   - `set` declared in `<set>`
     - The key type is the value type itself, and one element has one value.
     - It is reversible, sorted, and keys are unique, so it can hold no more than one of any given value.

       ```
       std::string s[6] = {"buffoon","thinkers","for",
                           "heavy","can","for"}; // "for" appears twice
       std::set<std::string> A(s1,s1+6); // can take a iterator range
                                         // [start,end)
       // print: buffoon can for heavy thinkers (sorted, unique)
       ```

     - Mathematics defines some standard operations for sets, such the union (implemented by STL algorithm `set_union()`).
   - `multiset` declared in `<set>`
     - It has the same facilities as `set`, but it allows a given key to appear more than once.
   - `map` declared in `<map>`
     - The key type is not necessarily the value type(s), and each key is associated with one value.
     - Each key can only appear once.
     - Dereferencing a `map` iterator yields a `pair` that holds a `const` key and its associated value.

       ```
       std::multimap<int, std:string> codes; // key type, value type
       std::pair<const int, std::string> item(213, "Los Alamos");
       // use: item.first (213), item.second ("Los Alamos")
       codes.insert(item);
       ```

   - `multimap` declared in `map`
     - It has the same facilities as `map`, but it allows a given key to appear more than once.
- Unordered Associative Containers (C++11)
   - An *unordered associative container* is yet another refinement of the container concept. Like an associative container, it associates a value with a key and uses the key to find to value.
   - The underlying difference is that while associative containers are based on tree structures, the unordered associative containers are based on hash table. The intent is to provide containers for which adding and deleting elements is relatively quick and for there there are efficient search algorithms.
   - The four unordered associative containers are `unordered_set`, `unordered_multiset`, `unordered_map`, and `unordered_multimap`.

     ☐