C++ Standard Template Library

Patrick Meier

C++ STL Overview

C++ Standard Template Library

What is the Standard Template Library?

The Standard Template Library (STL) is a general−purpose C++ library of algorithms and data structures, originated by Alexander Stepanov and Meng Lee. The STL, based on a concept known as ***generic programming***, is part of the standard ANSI C++ library. The STL is implemented by means of the C++ ***template*** mechanism, hence its name. While some aspects of the library are very complex, it can often be applied in a very straightforward way, facilitating reuse of the sophisticated data structures and algorithms it contains.

What about the `string` class?

Although not part of the STL, the `string` class is also part of the ANSI C++ standard library. Like the STL, it provides a commonly needed facility (character string handling).

A basic introduction to the STL and the `string` class

A complete introduction to the STL can be found by consulting the references below. For starters, though, a small number of classes and algorithms can be very useful:

- The ***find*** algorithm
- The ***list*** container template
- The ***map*** container template
- The ***iterator*** classes
- The ***string*** class
- The ***sort*** algorithm
- The ***vector*** container template

If you wish to use STL containers to hold objects of user−defined (class) types, as opposed to built−in types (e.g., ***int***), please refer to the requirements for these objects.

This short overview of the C++ STL is also available as a PDF−file.

Links to other resources

For more information on the STL, try the following links:

- Download the sgi reference implementation of the STL from: http://www.sgi.com/tech/stl
- Microsoft Visual C++ 5.0 has a reasonably good implementation of the STL and the string class as part of its standard library (see the MSVC 5.0 documentation for more information).
- Reference and tutorial information

- On−line reference: http://www.cs.rpi.edu/~musser/stl−book/
- Book: David R. Musser and Atul Saini, *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*, Addison−Wesley, 1996, ISBN 0−201−63398−1. More information is available by searhing for one of the authors' names at Addison−Wesley's web site.

• Newsgroups
- comp.lang.c++
- comp.std.c++

Last updated on 26.08.2002 by Patrick Meier

ANSI String Class

The ANSI string class implements a first−class character string data type that avoids many problems associated with simple character arrays ("C−style strings"). You can define a string object very simply, as shown in the following example:

```
#include <string>
using namespace std;
...
string first_name = "Bjarne";
string last_name;

last_name = "Stroustrup";

string names = first_name + " " + last_name;
cout << names << endl;

names = last_name + ", " + first_name;
cout << names << endl;
```

Member functions

The string class defines many member functions. A few of the basic ones are described below:

> ***Note***: The string class is based on a template class named ***basic_string***. Some of the member function declarations below may be a little confusing to those new to C++, even though they have been simplified somewhat. Fortunately, these functions are quite easy to use in practice.

Constructor	A `string` object may defined without an initializing value, in which case its initial value is an empty string (zero length, no characters):
	
	`string str1;`
	
	A `string` object may also be initialized with
	
	• a `string` expression:
	
	`string str2 = str1;`
	`string str3 = str1 + str2;`
	`string str4 (str2);  // Alternate form`
	• a character string literal:
	
	`string str4 = "Hello there";`
	`string str5 ("Goodbye");  // Alternate form`
	• a single character
	Unfortunately, the expected methods don't work:
	
	`string str6 = 'A';  // Incorrect`
	`string str7 ('A');  // Also incorrect`
	
	Instead, we must use a special form with two values:

```
string str7 (1,'A'); // Correct
```

The two values are the desired length of the string and a character to fill the string with. In this case, we are asking for a string of length one, filled with the character ***A***.

- a substring of another `string` object:

```
string str8 = "ABCDEFGHIJKL";
// Initialize str9 as "CDEFG"
// Starts at character 2 ('C')
// with a length of 5
// (or the rest of the string, if shorter)
string str9 (str8,2,5);
```

length

size

`size_type length() const;`
`size_type size() const;`
Both of these functions return the length (number of characters) of the string. The `size_type` return type is an unsigned integral type. (The type name usually must be scoped, as in `string::size_type`.)

```
string str = "Hello";
string::size_type len;
len = str.length(); // len == 5
len = str.size();   // len == 5
```

c_str

`const char* c_str() const;`
For compatibility with "older" code, including some C++ library routines, it is sometimes necessary to convert a `string` object into a character array ("C−style string"). This function does the conversion. For example, you might open a file stream with a user−specified file name:

```
string filename;
cout << "Enter file name: ";
cin >> filename;
ofstream outfile (filename.c_str());
outfile << "Data" << endl;
```

insert

`string& insert(size_type pos, const string& str);`
Inserts a string into the current string, starting at the specified position.

```
string str11 = "abcdefghi";
string str12 = "0123";
str11.insert (3,str12);
cout << str11 << endl; // "abc0123defghi"
str12.insert (1,"XYZ");
cout << str12 << endl; // "0XYZ123"
```

erase

`string& erase(size_type pos, size_type n);`
Delete a substring from the current string.

```
string str13 = "abcdefghi";
str12.erase (5,3);
cout << str12 << endl; // "abcdei"
```

replace

`string& replace(size_type pos, size_type n, const string& str);`
Delete a substring from the current string, and replace it with another string.

```
string str14 = "abcdefghi";
string str15 = "XYZ";
str14.replace (4,2,str15);
cout << str14 << endl; // "abcdXYZghi"
```

find

`size_type find (const string& str, size_type pos);`

rfind	Search for the ***first*** occurrence of the substring `str` in the current string, starting at position `pos`. If found, return the position of the first character. If not, return a special value (called `string::npos`). The member function rfind does the same thing, but returns the position of the ***last*** occurrence of the specified string.
	
	`string str16 = "abcdefghi";`
	`string str17 = "def";`
	`string::size_type pos = str16.find (str17,0);`
	`cout << pos << endl; // 3`
	`pos = str16.find ("AB",0);`
	`if (pos == string::npos) cout << "Not found" << endl;`

substr	`string substr (size_type pos, size_type n);`
	Returns a substring of the current string, starting at position `pos` and of length `n`:
	
	`string str18 = "abcdefghi"`
	`string str19 = str18.substr (6,2);`
	`cout << str19 << endl; // "gh"`

Non−member functions

In addition to member functions of the ***string*** class, some non−member functions are designed to work with strings; the most common of these is:

getline	`istream& getline (istream& is, string& str, char delim = '\n');`
	Reads characters from an input stream into a string, stopping when one of the following things happens:
	
	• An end−of−file condition occurs on the input stream
	• When the maximum number of characters that can fit into a string have been read
	• When a character read in from the string is equal to the specified delimiter (***newline*** is the default delimiter); the delimiter character is removed from the input stream, but not appended to the string.
	
	The return value is a reference to the input stream. If the stream is tested as a logical value (as in an ***if*** or ***while***), it is equivalent to ***true*** if the read was successful and ***false*** otherwise (e.g., end of file).
	
	The most common use of this function is to do "line by line" reads from a file. Remember that the normal extraction operator (>>) stops on white space, not necessarily the end of an input line. The ***getline*** function can read lines of text with embedded spaces.
	
	`vector<string> vec1;`
	`string line;`
	`vec1.clear();`
	`ifstream infile ("stl2in.txt");`
	`while (getline(infile,line,'\n'))`
	`{`
	`  vec1.push_back (line);`
	`}`

Operators

A number of C++ operators also work with `string` objects:

=	The assignment operator may be used in several ways:
	• Assigning one `string` object's value to another `string` object
	`string string_one = "Hello";`
	`string string_two;`
	`string_two = string_one;`
	• Assigning a C++ string literal to a `string` object
	`string string_three;`
	`string_three = "Goodbye";`
	• Assigning a single character (`char`) to a `string` object
	`string string_four;`
	`char ch = 'A';`
	`string_four = ch;`
	`string_four = 'Z';`

+	The "plus" operator concatenates:
	• two `string` objects
	`string str1 = "Hello ";`
	`string str2 = "there";`
	`string str3 = str1 + str2; // "Hello there"`
	• a `string` object and a character string literal
	`string str1 = "Hello ";`
	`string str4 = str1 + "there";`
	• a `string` object and a single character
	`string str5 = "The End";`
	`string str6 = str5 + '!';`

+=	The "+=" operator combines the above assignment and concatenation operations in the way that you would expect, with a `string` object, a string literal, or a single character as the value on the right−hand side of the operator.
	`string str1 = "Hello ";`
	`str1 += "there";`

== ***!=*** **<** **>** **<=** **>=**	The comparison operators return a Boolean (true/false) value indicating whether the specified relationship exists between the two operands. The operands may be: • two `string` objects • a `string` object and a character string literal

<<	The insertion operator writes the value of a `string` object to an output stream (e.g., `cout`).
	`string str1 = "Hello there";`
	`cout << str1 << endl;`

>>	The extraction operator reads a character string from an input stream and assigns the value to a `string` object.
	
	`string str1;`
	`cin >> str1;`

[]
 	
`(subscript)`
	The subscript operator accesses one character in a string:
	
	`string str10 = "abcdefghi";`
	`char ch = str10[3];`
	`cout << ch << endl; // 'd'`
	`str10[5] = 'X';`
	`cout << str10 << endl; // "abcdeXghi"`

Last updated on 26.08.2002 by Patrick Meier

Requirements for Objects in STL Containers

The use of STL containers places certain requirements on the objects that are stored in them. If you fail to meet these requirements, you will likely encounter compiler error messages that are very unhelpful in determining what the problems is.

Although the requirements on objects in STL containers vary by the specific container type (e.g., ***vector*** or ***list***) and by the algorithms that you apply, you can usually avoid trouble if you abide by the following rules when designing your "container contents" classes. Assume that ***MyObject*** is the class of the object that you wish to store in an STL container. Your class should include:

Member function	Example
"No argument" constructor	`MyObject::MyObject()`
Copy constructor	`MyObject::MyObject(const MyObject& m)`
Copy assignment operator	`const MyObject& MyObject::operator= (const MyObject& right)`

If you plan to use STL operations like ***find*** and ***sort***, you may need the following operations:

Member function	Example
Equality operator	`bool MyObject::operator== (const MyObject& right) const`
Inequality operator	`bool MyObject::operator!= (const MyObject& right) const`
"Less than" operator	`bool MyObject::operator< (const MyObject& right) const`
"Greater than" operator	`bool MyObject::operator> (const MyObject& right) const`

Note 1: Some systems, like Microsoft Visual C++ 5.0 (without service packs) may require the above relational operations to be declared, if not defined. This behavior seems inconsistent with the draft ANSI/ISO C++ standard.

Note2: It may only be necessary to define the equality and "less than" operations, as STL templates may be able to synthesize the inequality and "greater than" operations from them.

Although probably not necessary, you may wish to add the following operations, to complete the "relational operator" complement:

"Less than or equal" operator	`bool MyObject::operator<= (const MyObject& right) const`
"Greater than or equal" operator	`bool MyObject::operator>= (const MyObject& right) const`

Note that these examples are not the only way to provide the required operators. For example, you could implement the equality operator as the global function

```
bool operator== (const MyObject& left, const MyObject& right)
```

instead of the member function shown above. There are also some special templates in the `<utility>` library that can be used to supply missing operators, but they are a little tricky to apply correctly. It may be easier just to get in the habit of supplying the above operations for any class you wish to store in an STL container.

Last updated on 26.08.2002 by Patrick Meier

STL Vector Class

The `vector` container resembles a C++ array in that it holds zero or more objects of the same type, and that each of these objects can be accessed individually. (Be sure to study the requirements for objects that are stored in STL containers.) The `vector` container is defined as a template class, meaning that it can be customized to hold objects of any type. Here is a simple example:

```
#include <vector>
using namespace std;
...
// Define a vector of integers.
// The template name is "vector" and the type of object
// it contains is "int"; the fully specified container
// data type is "vector<int>".
vector<int> vec_one;

int a = 2;     // Some integer data objects
int b = -5;

vec_one.push_back(a); // Add item at end of vector
vec_one.push_back(9);
vec_one.push_back(b);

// vec_one now contains three int values: 2, 9, -5

unsigned int indx;
for (indx = 0; indx < vec_one.size(); indx++)
{
  cout << vec_one[indx] << endl; Write out vector item
}
```

Member functions

Some commonly used member functions of the `vector` class are:

size	`size_type size() const;` Returns the number of items (elements) currently stored in the vector. The `size_type` type is an unsigned integral value.
empty	`bool empty() const;` Returns a true value if the number of elements is zero, false otherwise.
push_back	`void push_back(const T& x);` Adds the element `x` at the end of the vector. (`T` is the data type of the vector's elements.)
begin	`iterator begin();` Returns an iterator (a special kind of object) that references the beginning of the vector. Although the iterator can be used for many things, for now we will just consider its use with `erase` and `sort`.
end	`iterator end();` Returns an iterator (a special kind of object) that references a position past the end of the vector. Like `begin()`, we will just consider its use with `erase` and `sort`.
erase	`void erase(iterator first, iterator last);` Erase (remove) elements from a vector. For now, we will consider only the case of removing all

	elements from a vector (see `clear()` for an alternate way to do the same thing):
	
	`vector<int> a;`
	`...`
	`a.erase(a.begin(),a.end()); // Remove all elements.`

clear	`void clear ();`
	Erase all elements from a vector.
	
	`vector<int> a;`
	`...`
	`a.clear(); // Remove all elements.`

Operators

Some of the operators defined for the `vector` container are:

=	The assignment operator replaces the target vector's contents with that of the source vector:
	
	`vector<int> a;`
	`vector<int> b;`
	
	`a.push_back(5);`
	`a.push_back(10);`
	
	`b.push_back(3);`
	
	`b = a;`
	`// The vector b now contains two elements: 5, 10`

==	Tests whether two vectors have the same content (element−by−element comparison for all elements).

[]	The subscript operator returns a reference to an element of the vector. A subscript value of zero returns a reference to the first element, and so on. The subscript must be between zero and `size()−1`. See the example above to see how the subscript operator is used in a loop to access elements of a vector. The subscripted vector may appear on the left or right sides of an assignment (the returned reference is an ***lvalue***):
	
	`vector<double> vec;`
	`vec.push_back(1.2);`
	`vec.push_back(4.5);`
	
	`vec[1] = vec[0] + 5.0;`
	`vec[0] = 2.7; // Vector now has two elements: 2.7, 6.2`

To sort a vector, see the STL sort algorithm.

Last updated on 26.08.2002 by Patrick Meier

STL List Class

The `list` container implements a classic list data structure; unlike a C++ array or an STL vector, the objects it contains cannot be accessed directly (i.e., by subscript). (Be sure to study the requirements for objects that are stored in STL containers.) The `list` container is defined as a template class, meaning that it can be customized to hold objects of any type. Here is a simple example:

```
#include <list>         // list class library
using namespace std;
...
  // Now create a "list" object, specifying its content as "int".
  // The "list" class does not have the same "random access" capability
  // as the "vector" class, but it is possible to add elements at
  // the end of the list and take them off the front.
  list<int> list1;

  // Add some values at the end of the list, which is initially empty.
  // The member function "push_back" adds at item at the end of the list.
  int value1 = 10;
  int value2 = -3;
  list1.push_back (value1);
  list1.push_back (value2);
  list1.push_back (5);
  list1.push_back (1);

  // Output the list values, by repeatedly getting the item from
  // the "front" of the list, outputting it, and removing it
  // from the front of the list.
  cout << endl << "List values:" << endl;
  // Loop as long as there are still elements in the list.
  while (list1.size() > 0)
  {
    // Get the value of the "front" list item.
    int value = list1.front();
    // Output the value.
    cout << value << endl;
    // Remove the item from the front of the list ("pop_front"
    // member function).
    list1.pop_front();
  }
```

Member functions

Some commonly used member functions of the `list` class are:

size	`size_type size() const;`
	Returns the number of items (elements) currently stored in the list. The `size_type` type is an unsigned integral value.
	
	`// Loop as long as there are still elements in the list.`
	`while (list1.size() > 0)`
	`{`
	`  ...`
	`}`

empty	

	`bool empty() const;`
	Returns a true value if the number of elements is zero, false otherwise.
	
	`if (list1.empty())`
	`{`
	`  ...`
	`}`

push_back
push_front
	`void push_back(const T& x);`
	`void push_front(const T& x);`
	Adds the element `x` at the end (or beginning) of the list. (`T` is the data type of the list's elements.)
	
	`list<int> nums;`
	`nums.push_back (3);`
	`nums.push_back (7);`
	`nums.push_front (10); // 10 3 7`

front
back
	`T& front();`
	`const T& front() const;`
	`T& back();`
	`const T& back() const;`
	Obtain a reference to the first or last element in the list (valid only if the list is not empty). This reference may be used to access the first or last element in the list.
	
	`list<int> nums;`
	`nums.push_back(33);`
	`nums.push_back(44);`
	`cout << nums.front() << endl; // 33`
	`cout << nums.back() << endl; // 44`

begin	`iterator begin();` Returns an iterator that references the beginning of the list.
end	`iterator end();` Returns an iterator that references a position just past the last element in the list.

insert	`iterator insert(iterator position, const T& x);`
	Insert the element `x` (type `T` is the type of a list element) into the list at the position specified by the iterator (before the element, if any, that was previously at the iterator's position). The return value is an iterator that specifies the position of the inserted element.
	
	`nums_iter = find (nums.begin(), nums.end(), 15);`
	`if (nums_iter != nums.end())`
	`{`
	`  nums_iter = nums.insert (nums_iter, -22);`
	`  cout << "Inserted element " << (*nums_iter) << endl;`
	`}`

erase	`void erase(iterator position);`
	`void erase(iterator first, iterator last);`
	Erase (remove) one element or a range of elements from a list. In the case of a range, this operation deletes elements from the first iterator's position up to, but not including, the second iterator's position. For an alternate way to erase all elements, see `clear()`.
	
	`nums.erase (nums.begin(), nums.end()); // Remove all elements;`
	`...`
	`nums_iter = find(nums.begin(), nums.end(), 3); // Search the list.`
	`// If we found the element, erase it from the list.`
	`if (nums_iter != nums.end()) nums.erase(nums_iter);`

clear	

	`void clear();`
	Erase all elements from a list.
	
	`nums.clear(); // Remove all elements;`

pop_front
pop_back
	`void pop_front();`
	`void pop_back();`
	Erases the first (or last) element from a list. These operations are illegal if the list is empty.
	
	`while (list1.size() > 0)`
	`{`
	`  ...`
	`  list1.pop_front();`
	`}`

remove	`void remove (const T& value);`
	Erases all list elements that are equal to value. The equality operator (==) must be defined for T, the type of element stored in the list.
	
	`nums.remove(15);`

sort	`void sort();`
	Sorts the list elements in ascending order. The comparison operator < ("less than") must be defined for the list element type. Note that the STL sort ***algorithm*** does NOT work for lists; that's why a sort ***member function*** is supplied.
	
	`nums.sort();`

reverse	`void reverse();`
	Reverses the order of elements in the list.
	
	`nums.reverse();`

In addition to these member functions, some STL algorithms (e.g., `find`) can be applied to the `list` container.

Operators

Some of the operators defined for the `list` container are:

=	The assignment operator replaces the target list's contents with that of the source list:
	
	`list<int> a;`
	`list<int> b;`
	
	`a.push_back(5);`
	`a.push_back(10);`
	
	`b.push_back(3);`
	
	`b = a;`
	`// The list b now contains two elements: 5, 10`

==	Tests whether two lists have the same content (element−by−element comparison for all elements).

Note that there is ***no subscript operator*** for the `list` container, since random access to elements is not

supported.

Using iterators with lists

More information on iterators is available. Here, we will consider only a few simple uses of iterators with list containers.

Traversing a list

A simple example of iterator use is traversing a list to print out its elements. The function `output_int_list` writes the elements of a list to an output stream and appends a specified delimiter string (here, just a newline character) after each element.

The list is passed by constant reference, instead of by value, so that it does not have to be copied (along with all its elements), but is still safe from modification. A `const` iterator (necessary because the argument is `const`) is used to traverse the list; each element is written to the specified stream.

```
#include <list>  // list class library
using namespace std;
...
  list<int> nums;

  nums.push_back (3);
  nums.push_back (7);
  nums.push_front (10);

  cout << endl << "List 'nums' now is:" << endl;
  output_int_list (nums, cout, "\n");
  cout << endl;
...
void output_int_list (const list<int>& lst,
                      ostream& out_stream,
                      const string& delim)
{
  // Create constant iterator for list.
  list<int>::const_iterator iter;
  // Iterate through list and output each element.
  for (iter=lst.begin(); iter != lst.end(); iter++)
  {
    out_stream << (*iter) << delim;
  }
}
```

Accessing adjacent elements

Just as the iterator's increment operator can be used to move forward in a list, the decrement operator may be used to "back up." Here, we search for the first occurrence of a certain element value, and then print the previous and following elements, if they exist. Note that we can copy the value of an iterator to "mark" a list

position and that more than one iterator can be associated with a single container.

```
#include <list>  // list class library
#include <algorithm>     // STL algorithms class library
using namespace std;
...
  list<int> nums;
  list<int>::iterator nums_iter;

  nums.push_back (3);
  nums.push_back (7);
  nums.push_front (10);

  nums_iter = find(nums.begin(), nums.end(), 3); // Search the list.
  if (nums_iter != nums.end())
  {
    cout << "Number " << (*nums_iter) << " found." << endl; // 3
    // If found element is not first, print out previous.
    if (nums_iter != nums.begin())
    {
        // Copy "found" iterator, and back up one position.
         list<int>::iterator prev_iter = nums_iter;
        cout << "Previous element is " << (*(--prev_iter)) << endl;
    }

    // Copy "found" iterator position, and move forward one position.
    list<int>::iterator next_iter = nums_iter;
    next_iter++;
    // If we didn't fall off the end, print out next element.
    if (next_iter != nums.end())
    {
        cout << "Following element is " << (*next_iter) << endl;
    }
  }
  else
  {
    cout << "Number not found." << endl;
  }
```

Iterator values may become invalid if the content of the associated container changes. For example, an iterator may specify the position of an element that is subsequently erased; in this case, the iterator value is no longer valid.

Last updated on 26.08.2002 by Patrick Meier

STL Map Class

The `map` container implements a classic map data structure. The object can be accessed fast directly. (Be sure to study the requirements for objects that are stored in STL containers.) The `map` container is defined as a template class, meaning that it can be customized to hold objects of any type. Here is a simple example:

```
#include<map>           // map class library
...
  // Now create a "map" type, specifying the key as "long" and the value as "std::string".
  typedef std::map<long, std::string, less<long>>MyMapType; // comparison object: less<long>
  typedef MyMapType::value_type ValuePair;
...
  MyMapType aMap;

  // Add some values
  aMap.insert( ValuePair(836361136, "Andrew") );
  aMap.insert( ValuePair(274635328, "Berni") );
  aMap.insert( ValuePair(534534345, "John") );
  aMap.insert( ValuePair(184092144, "Karen") );
  aMap.insert( ValuePair(785643523, "Thomas") );
  aMap.insert( ValuePair(923452344, "William") );

  // insetrion of Tom is not executed, because the key
  // already exists!
  aMap.insert( ValuePair(785643523, "Tom") );

  // Due to the underlying implementation, the output of the names
  // is sorted by numbers:

  std::out <<"Output: "  <<std::endl;
  MyMapType::const_iterator iter = aMap.begin();

  while( iter != aMap.end() )
  {
      std::out  <<(*iter).first  <<":"          // number
                <<(*iter).second  <<std::endl;  // name
      ++iter;
  }

  std::out  <<"Output of the name after entering the number: ";

  long number;
  std::cin >> number;

  iter = aMap.find( number ); // O(log N)

  if( iter!=aMap.find(number) )
  {
      std::cout  <<(*iter).second  <<' ' // O(1)
                 <<aMap[number]          // O(log N)
                 <<std::endl;
  }
  else
  {
      std::cout  <<"not found!"  <<std::endl;
  }
```

Member functions

Some commonly used member functions of the `map` class are:

size	`size_type size() const;`
	Returns the number of items (elements) currently stored in the map. The `size_type` type is an unsigned integral value.
	
	`// Loop as long as there are still elements in the map.`
	`while( map.size() > 0)`
	`{`
	`  ...`
	`}`

empty	`bool empty() const;`
	Returns a true value if the number of elements is zero, false otherwise.
	
	`if( map.empty() )`
	`{`
	`  ...`
	`}`

begin	`iterator begin();` Returns an iterator that references the beginning of the map.
end	`iterator end();` Returns an iterator that references a position just past the last element in the map.
rbegin	`iterator rbegin();` Returns an iterator that references the beginning of the map in reverse order.
rend	`iterator rend();` Returns an iterator that references a position just past the last element in the map in reverse order.

insert	`pair<iterator, bool> insert( const value_type &x );`
	Insert the element `x` into the map.
	
	`MyMapType aMap;`
	`aMap.insert( MyMapType::value_type(836361136, "Andrew") );`

erase	`void erase(iterator position);`
	`void erase(iterator first, iterator last);`
	Erase (remove) one element or a range of elements from a map. In the case of a range, this operation deletes elements from the first iterator's position up to, but not including, the second iterator's position. For an alternate way to erase all elements, see `clear()`.
	
	`nums.erase( nums.begin(), nums.end() ); // Remove all elements;`
	`...`
	`nums_iter = find( nums.begin(), nums.end(), 3 ); // Search the map.`
	`// If we found the element, erase it from the map.`
	`if( nums_iter != nums.end() ) nums.erase( nums_iter );`

clear	`void clear();`
	Erase all elements from a map.
	
	`nums.clear(); // Remove all elements;`

In addition to these member functions, some STL algorithms (e.g., `find`) can be applied to the `map` container.

Operators

Some of the operators defined for the `map` container are:

Using iterators with map

More information on iterators is available. Here, we will consider only a few simple uses of iterators with map containers.

Last updated on 26.08.2002 by Patrick Meier

STL Sort Algorithm

The `sort` algorithm is an operation (function) that can be applied to many STL containers (with the notable exception of the `list` container). For now, we will consider its use only with the `vector` container class template. It can be used in the following way:

```
#include <vector>
#include <algorithm>  // Include algorithms
using namespace std;

vector<int> vec;
vec.push_back (10);
vec.push_back (3);
vec.push_back (7);

sort(vec.begin(), vec.end()); // Sort the vector

// The vector now contains: 3, 7, 10
```

The `sort` algorithm orders the container's contents in ascending order, as defined by the "less than" (**<**) operator as applied to the vector elements. If this operator is defined for a programmer−defined type (as is the case with the `string` class), then the programmer−defined type can be sorted just as easily as a built−in type.

Last updated on 26.08.2002 by Patrick Meier

STL Find Algorithm

The `find` algorithm is an operation (function) that can be applied to many STL containers. It searches a subrange of the elements in a container (or all the elements), looking for an element that is "equal to" a specified value; the equality operator (`==`) must be defined for the type of the container's elements. The `find` algorithm can be used in the following way:

```
#include <vector>  // vector class library
#include <list>    // list class library
#include <algorithm>       // STL algorithms class library
using namespace std;
...
list<int> nums;
list<int>::iterator nums_iter;

nums.push_back (3);
nums.push_back (7);
nums.push_front (10);

nums_iter = find(nums.begin(), nums.end(), 3); // Search the list.
if (nums_iter != nums.end())
{
    cout << "Number " << (*nums_iter) << " found." << endl; // 3
}
else
{
    cout << "Number not found." << endl;
}

// If we found the element, erase it from the list.
if (nums_iter != nums.end()) nums.erase(nums_iter);
// List now contains: 10 7
```

The `find` algorithm searches the specified subrange of the container's elements and stops when it finds the ***first*** element equal to the specified value, as defined by the equality (**==**) operator as applied to the container's elements. If this operator is defined for a programmer−defined type (as is the case with the `string` class), then a search for the programmer−defined type can be done just as easily as for a built−in type.

Note that the search value (the third argument to the ***find*** function) must be of the same type as the elements stored in the container, or at least of a type that the compiler can automatically convert. In many cases, it will be necessary to create and initialize a temporary data object for this purpose.

The return value of the ***find*** function is an iterator specifying the position of the first matching element. If no matching element is found, the return value is equal to the iterator specifying the end of the element subrange (in the example above, the end of the list).

Last updated on 26.08.2002 by Patrick Meier

STL Iterator Classes

Iterators are special STL objects that represent ***positions*** of elements in various STL containers. Simplistically, they play a role similar to that of a subscript in a C++ array, permitting the programmer to access a particular element, and to traverse through the container. There are many different kinds of iterators, depending on the type of container with which they are associated. For the purposes of this discussion, only a few simple iterator capabilities will be considered.

Declaring an iterator

Because an iterator object is always associated with one specific type of container object, its type depends on the type of the container. For example:

```
#include <list>
#include <vector>
using namespace std;
...
list<int> nums;
list<int>::iterator nums_iter;
vector<double> values;
vector<double>::iterator values_iter;
vector<double>::const_iterator const_values_iter;
```

At any given time, an iterator object is associated with only one container object. There are several basic types of iterators:

Iterator type	Capabilities
Forward	Can specify the position of a single element in a container. Can move in one direction from element to element in a container.
Bidirectional	Same as forward iterator, but can move in two directions ("forward" and "reverse") from element to element.
Random access	Same as bidirectional iterator, but can also move in bigger steps (skipping multiple elements).

Iterators may be `const` (e.g., "`const_iterator`") or non−`const`. Constant iterators can be used to examine container elements, but not modify them. Non−constant iterators may not be used with constant container objects. Further, some specialized STL containers do not permit the use of non−constant iterators.

Iterator operators

Various operators can be applied to an iterator object, depending on its type:

Operator	Description	Forward	Bidirectional	Random access
==				

	Returns `true` if two iterator values specify the same element position, `false` otherwise. Valid only if both iterator values are associated with the same container object.			
!=	Returns `true` if two iterator values do NOT specify the same element position, `false` if they do. Valid only if both iterator values are associated with the same container object.	☑	☑	☑
*****	Returns a reference to the container element at the position specified by the iterator. Valid only if there is an element at that position. The reference can be used to examine or modify the element. For constant container objects (and for some specialized STL containers, constant or not), the reference is constant; in this case, it can only be used to examine the container element, not modify it. See the note below on invoking member functions of an object referenced by an iterator.	☑	☑	☑
++	Increments the iterator's value, so it specifies the ***next*** position in the associated container.	☑	☑	☑
--	Decrements the iterator's value, so it specifies the ***previous*** position in the associated container.		☑	☑
+= **-=** **+** **-**	Adds or subtracts an offset from the iterator's value, moving it forward or backward by a specified number of positions within the container.			☑
< **>** **<=** **>=**	Compares two iterator values and returns `true` or `false`, depending on whether the specified relationship is true. Valid only if both iterator values are associated with the same container.			☑

Iterator usage

Iterators have many uses. A few examples are included here.

List element access

After creating a list of integers and adding some elements, we initialize the iterator `nums_iter` to specify the first list position (`nums.begin()`). The loop runs until we reach the iterator value that represents the position "just beyond" the last list element (`nums.end()`). Each list element is accessed by the "`*`" operator, which returns a reference to the element. The increment operator (`++`) moves the iterator to the next position in the list.

Note that we are using the iterator to modify the list elements. If this were not the case, the iterator could have been declared as "`list<int>::const_iterator nums_iter;`".

```
#include <list>         // list class library
using namespace std;
...
  list<int> nums;
  list<int>::iterator nums_iter;
  nums.push_back (0);
  nums.push_back (4);
  nums.push_front (7);

  cout << endl << "List 'nums' now becomes:" << endl;
  for (nums_iter=nums.begin(); nums_iter != nums.end(); nums_iter++)
  {
    *nums_iter = *nums_iter + 3; // Modify each element.
    cout << (*nums_iter) << endl;
  }
  cout << endl;
```

Invoking member functions of an object referenced by an iterator

Very often, the elements stored in an STL container are objects of class type. In this case, we may want to invoke member functions of the object referenced by an iterator. To do so, we have to watch out for operator precedence, so that the compiler understands what we are trying to do. For example:

```
#include <list>         // list class library
#include <string> // string class library
using namespace std;
...
  list<string> words;
  list<string>::iterator words_iter;
...
  unsigned int total_length = 0;
  for (words_iter=words.begin(); words_iter != words.end(); words_iter++)
  {
    total_length += (*words_iter).length();  // correct
//  total_length += *words_iter.length();    // incorrect !!
  }
  cout << "Total length is " << total_length << endl;
```

The parentheses around "*words_iter" are required when we invoke the "length()" member function. Without them, the compiler would think that the "length()" function is a member of the iterator class, not of the string class, since the "." operator would otherwise be evaluated before the unary "*" operator.

As you might expect, the parentheses would also be required if we wish to access a data member of an object referenced by an iterator.

Using iterators with container member functions and STL algorithms

Some STL functions (both container member functions and algorithms) require iterator arguments. This following example illustrates iterator usage by:

- The ***sort*** algorithm as applied to a ***vector*** container
- The ***erase*** member function (***vector*** and ***list*** containers), specifying one element or a range of elements
- The ***find*** algorithm as applied to a ***list*** container

```
#include <vector>       // vector class library
#include <list>         // list class library
#include <algorithm>            // STL algorithms class library
using namespace std;
...
  vector<string> vec1;

  string state1 = "Wisconsin";
  string state2 = "Minnesota";
  vec1.push_back (state1);
  vec1.push_back (state2);
  vec1.push_back ("Illinois");
  vec1.push_back ("Michigan");
  sort(vec1.begin(),vec1.end()); // Sort the vector of strings.

  vec1.erase(vec1.begin(),vec1.end());
...
  list<int> nums;
  list<int>::iterator nums_iter;
...
  nums.erase (nums.begin(), nums.end()); // Remove all elements.
...
  nums_iter = find(nums.begin(), nums.end(), 3); // Search the list.
  if (nums_iter != nums.end())
  {
    cout << "Number " << (*nums_iter) << " found." << endl;
  }
...
  // If we found the element, erase it from the list.
  if (nums_iter != nums.end()) nums.erase(nums_iter);
```


Last updated on 26.08.2002 by Patrick Meier