THREE

STL <utility>, <functional>, <algorithm>, and <iterator> Templates

Just as every solid building starts with proper footers and a strong foundation, so too do C++ applications need the correct building blocks for an entire structure to stand. Before discussing the structural requirements for applications using the STL, you need to recognize the syntax evolution behind the new standard C++ libraries. These are separate library routines used by most C/C++ applications and have nothing directly to do with STL.

The `<utility>` template defines global versions of the **<=**, **>**, **>=**, and **!=** operators, which are all defined in terms of the **<** and **==**. Both the `<utility>` and `<functional>` templates are viewed as support libraries. You use the `<functional>` STL for defining several templates that help construct predicates for the templates defined in `<algorithm>` and `<numeric>`.

The `<algorithm>` template functions and classes work on containers. Although each individual container provides support for its own basic operations, the standard algorithms provide more extended or complex actions. They also allow you to work with two different types of containers at the same time.

The STL `<iterator>` template is an extremely important component of STL. While many of the other STL templates, such as container types, are easier to digest and use, in actuality STL is composed of many interrelated components. Iterators are every bit as important as containers, algorithms, and allocators.

Note The standard C++ library is a separate entity from the STL.

The standard C++ library encompasses all the latest ANSI C++, including the STL and a new iostream library. The standard C++ library provides new

functionality, such as numerous algorithms that manipulate C++ objects, and a migration path for developers who want to move to the standard iostream.

The standard C++ library is a set of 51 header files. The new header files do *not* have the `.h` extension. The standard C++ library also has 18 standard C headers with the `.h` extension, for example, `errno.h` and `stdio.h`.

The main difference between the standard C++ library and previous run-time libraries is in the iostream library. Details of the iostream implementation have changed, and you cannot mix calls to the old iostream library and to the new standard C++ library.

From iostream.h to <iostream>

Traditional C++ applications have always accessed library routines with the straightforward **`#include <filename.h>`** syntax, as in:

```
#include <fstream.h>
#include <iomanip.h>
#include <ios.h>
#include <iostream.h>
#include <istream.h>
#include <ostream.h>
#include <strstrea.h>
```

Under the new ANSI C/C++/ISO standard, to use the new standard C++ library, you include one or more of the standard C++ library header files in your code, with a different syntax, as in:

```
#include <iostream>
```

The new header files do not have the `.h` extension. You should *not* use the old iostream header files (`fstream.h`, `iomanip.h`, `ios.h`, `iostream.h`, `istream.h`, `ostream.h`, `streamb.h`, and `strstrea.h`). You cannot mix calls to the old iostream library and the new standard C++ library.

The good news is that many of the new standard C++ iostream header files, `fstream`, `iomanip`, `ios`, `iosfwd`, `iostream`, `istream`, `ostream`, `sstream`, `streambuf`, and `strstream`, have names that are the same as or similar to the old iostream header files, but without the `.h` extension. If you include the new standard C++ header files, the run-time library files that contain the standard C++ library will be the default libraries.

Many of the underlying details of the iostream implementation have changed. For some applications, you may have to rewrite parts of your code

that use iostream if you want to link with the Standard C++ library by removing any old iostream headers, such as `fstream.h`, `iomanip.h`, `ios.h`, `iostream.h`, `istream.h`, `ostream.h`, `streamb.h`, and `strstrea.h`, included in your code. They are replaced with one or more of the new standard C++ iostream headers, `fstream`, `iomanip`, `ios`, `iosfwd`, `iostream`, `istream`, `ostream`, `sstream`, `streambuf`, and `strstream`, all without the `.h` extension.

One option for older applications making heavy use of iostream is to simply link with the new standard C++ library. In this case, leave the old iostream headers in your code and the old iostream library will automatically be linked. However, you cannot include any of the new standard C++ library headers. You cannot mix calls to the old iostream library and the new standard C++ library.

The following section explains the differences in the new standard C++ iostream library and the old iostream library. In the new standard C++ iostream library:

- Open functions no longer have a third protection parameter.
- You can no longer create streams from file handles.
- There are no open ofstream objects with the ios::out flag by itself; instead, you combine the flag with another ios enumerator in a logical AND, for example, with `ios::in` or `ios::nocreate`.
- `ios::unsetf` returns void instead of the previous value.
- `istream::get( char& rchar )` does not assign to `rchar` if there is an error.
- `istream::get( char* pchar, int nCount, char cdelim )` is different in three ways:
  - When nothing is read, the failbit is set.
  - `istream::seekg` with an invalid parameter does not set the failbit.
  - The return type `streampos` is a class with overloaded operators. In functions that return a `streampos` value (such as `istream::tellg`, `ostream::tellp`, `strstreambuf::seekoff`, and `strstreambuf::seekpos`), you should cast the return value to the type required: `streamoff, fpos_t, or mbstate_t`.
- The first function parameter `falloc`, in `strstreambuf::strstreambuf`, takes a `size_t` argument, not the older type long.
- The following list enumerates elements of the old iostream library that are not elements of the new iostream library:
  - `Attach` member function of `filebuf`, `fstream ifstream`, and `ofstream`..
  - `filebuf::openprot` and `filebuf::setmode`.

- ios::bitalloc, ios::nocreate, ios::noreplace, and ios::sync_with_stdio
- streambuf::out_waiting, and streambuf::setbuf.

STL Syntax

Microsoft Visual C++ ships with a complete set of tools to help the STL beginner. Should you wish to view the formal definitions of any of the individual template libraries, follow these simple steps:

1. Click on the Visual C++ Help menu.
2. Choose the Search option.
3. When the Help window opens, make certain that the Index tab is selected.
4. Type in an STL name. For example, to find the definition for the <utility> template, type utility without the angle brackets.
5. Once the search has narrowed down to words close to your selection, search for the entry that uses the template's name, i.e., utility followed by the words header file.
6. Click on that entry and you'll launch yourself off into the template's formal syntax definition.
7. From there, hot-links can take you to any necessary related definitions and sample code.

STL <utility> and <functional> Templates

The <utility> and <functional> templates are very rarely used by themselves. Instead, they can be viewed as adding additional functionality to the other STLs. Since they are such a frequent component of mainstream STL applications, it is worthwhile to view their definitions and capabilities.

<utility> Template Syntax

The header file <utility> defines global versions of the <=, >, >=, and **!=** operators, which are all defined in terms of the < and ==. In principle, STL assumes that all object containers have at least the following:

- An assignment operator (=).
- An equality comparison operator (==).
- A less than comparison operator (<).
- A copy constructor.

Include the STL standard header `<utility>` to define several templates of general use throughout the STL. Four template operators, `operator!=`, `operator<=`, `operator>`, and `operator>=`, define a total ordering on pairs of operands of the same type, given definitions of `operator==` and `operator<`. If an implementation supports namespaces, these template operators are defined in the `rel_ops` namespace, nested within the `std` namespace.

To use the `<utility>` template operators, you use the following syntax:

```
using namespace std::rel_ops;
```

which promotes the template operators into the current namespace.

Those familiar to C/C++ programming know that frequently, to understand a particular definition, declaration, or statement, it is necessary to nest backwards through many levels of predefined identifiers (constants, variables, functions, and classes). The same holds true with the STL library. The following section saves you the time of doing this for all related support definitions.

STRUCT PAIR

```
template<class T, class U>
    struct pair {
    typedef T first_type;
    typedef U second_type
    T first;
    U second;
    pair();
    pair(const T& x, const U& y);
    template<class V, class W>
        pair(const pair<V, W>& pr);
    };
```

You use the pair template class to store pairs of objects. *`first`* is assigned the type *`T`*, and *`second`* type *`U`*. The type definition *`first_type`* is the same as the template parameter *`T`*, while *`second_type`* is the same as the template parameter *`U`*. The first (default) constructor initializes *`first`* to *`T()`* and `second` to *`U()`*. The second constructor initializes `first` to `x` and `second` to `y`. The third (template) constructor initializes `first` to `pr.first` and `second` to `pr.second`. `T` and `U` each need to supply only a single argument constructor and a destructor.

MAKE_PAIR

```
template<class T, class U>
    pair<T, U> make_pair(const T& x, const U& y);
```

The template function returns `pair<T, U>(x, y)`.

THE PAIRDATA.CPP APPLICATION

The first example application, `pairdata.cpp`, shows how the `<utility>` `make_pair()` method is used to create and use a pair of data types:

```
// pairdata.cpp
// Testing <utility>
// make_pair()
// Chris H. Pappas and William H. Murray, 1999


#include <utility>
#include <iostream>

using namespace std;

// STL id_PAIR, struct pair typedef using int and char
member types

typedef struct pair<int,char> ic_PAIR;

void main(void){

  // create storage for and initialize
  // an_int_char_pair using make_pair()

  ic_PAIR an_int_char_pair = make_pair(15,'c');


  // output the individual members
  cout << an_int_char_pair.first
       << "   "
       << an_int_char_pair.second
       << endl;


  // assign new member values
  an_int_char_pair.first  = 20;
  an_int_char_pair.second = 'a';


  // output the new values
  cout << an_int_char_pair.first
       << "   "
       << an_int_char_pair.second
       << endl;
}
```

The straightforward output from the program looks like:

```
15  c
20  a
```

THE UTILITY.CPP APPLICATION

This next example uses an `int, double` *pair* to test the overloaded `<utility>` operators:

```
// utility.cpp
// Testing <utility>
// ==, !=, <, <=, >, >=
// Chris H. Pappas and William H. Murray, 1999


#include <iostream>
#include <utility>

using namespace std ;

// STL id_PAIR, struct pair typedef usingint and double member types

typedef struct pair<int, double> id_PAIR;

void main(void)
{
  id_PAIR pair33_56a(33,5.6);
  id_PAIR pair22_56 (22,5.6);
  id_PAIR pair33_95 (33,9.5);
  id_PAIR pair33_56b(33,5.6);

// output original values

  cout << "pair33_56a = (  " << pair33_56a.first << " , " <<
pair33_56a.second << " )" << endl;
  cout << "pair22_56  = (  " << pair22_56.first  << " , " <<
pair22_56.second  << " )" << endl;
  cout << "pair33_95  = (  " << pair33_95.first  << " , " <<
pair33_95.second  << " )" << endl;
  cout << "pair33_56b = (  " << pair33_56b.first << " , " <<
pair33_56b.second << " )" << endl;

  cout << "\n\n\n";

// test for equality ==

  cout << ((pair33_56a == pair33_56b) ?
      "pair33_56a and pair33_56b are     equal \n" :
      "pair33_56a and pair33_56b are not equal \n");

// test for non-equality !=
```

```
  cout << ((pair22_56 != pair33_95) ?
      "pair22_56  and pair33_95  are not equal \n" :
      "pair22_56  and pair33_95  are     equal \n");

  cout << "\n\n";

// test for greater than >

  cout << ((pair33_56a > pair33_95) ?
       "pair33_56a is     greater than pair33_95 \n" :
       "pair33_56a is not greater than pair33_95 \n");

// test for greater than or equal >=

  cout << ((pair33_56a >= pair33_95) ?
       "pair33_56a is     greater than or equal to pair33_95 \n" :
       "pair33_56a is not greater than or equal to pair33_95 \n");

  cout << "\n\n";

// test for less than <

  cout << ((pair33_95 < pair33_56a) ?
      "pair33_95  is     less    than pair33_56a \n" :
      "pair33_95  is not less    than pair33_56a \n");

// test for less than or equal <=

  cout << ((pair33_95 < pair33_56a) ?
       "pair33_95  is     less    than or equal to pair33_56a \n" :
       "pair33_95  is not less    than or equal to pair33_56a \n");
}
```

The output from the program looks like:

```
pair33_56a = ( 33 , 5.6 )
pair22_56  = ( 22 , 5.6 )
pair33_95  = ( 33 , 9.5 )
pair33_56b = ( 33 , 5.6 )

pair33_56a and pair33_56b are     equal
pair22_56  and pair33_95  are not equal

pair33_56a is not greater than pair33_95
pair33_56a is not greater than or equal to pair33_95

pair33_95  is not less    than pair33_56a
pair33_95  is not less    than or equal to pair33_56
```

ch03 5/12/2000 7:59 AM Page 95

<functional> Template Syntax

As mentioned earlier, both the `<utility>` and `<functional>` templates are viewed as support libraries. You use the `<functional>` STL for defining several templates that help construct predicates for the templates defined in `<algorithm>` and `<numeric>`.

When you include the STL standard header `<functional>`, you gain access to several templates that help construct function objects. These are objects of a class that defines `operator()`. Later chapters will demonstrate how function objects behave much like function pointers, except that the objects can store additional information that can be used during a function call.

THE ITEMPLAT.CPP APPLICATION

The following program instantiates the `plus`, `minus`, `multiplies`, and `divides` templates for the standard C/C++ integer data type:

```
// itemplat.cpp
// Testing <functional>
// plus<>, minus<>, multiplies<>, divides<>
// Chris H. Pappas and William H. Murray, 1999

#include <iostream>
#include <functional>

using namespace std ;

class functional_tmplts :  public plus<int>, public minus<int>,
                 public multiplies<int>, public divides<int>
{
public:

  int iValue1;

  // Overloaded constructors
  functional_tmplts()            {iValue1 = 0      ;}
  functional_tmplts(int aValue1 ){iValue1 = aValue1;}

  // Overloaded operators
  result_type operator+(second_argument_type iValue2_to_add)
                                 {return iValue1 + iValue2_to_add;}
  result_type operator-(second_argument_type iValue2_to_sub)
                                 {return iValue1—iValue2_to_sub;}
  result_type operator*(second_argument_type iValue2_to_mul)
                                 {return iValue1 * iValue2_to_mul;}
  result_type operator/(second_argument_type iValue2_to_divby)
                                 {return iValue1 / iValue2_to_divby;}
```

ch03 5/12/2000 7:59 AM Page 96


```
};

ostream& operator<<(ostream& os, const functional_tmplts& obj )
{
       os << obj.iValue1 ;
       return os ;
}

void main(void)
{
  functional_tmplts iFirstResult,iSecondResult,
                   iThirdResult,iFourthResult,iFifthResult;

  cout << "Testing <functional> STL library \n\n";

  iFirstResult = 10;
  cout << "iFirstResult  = "
       <<  iFirstResult << endl ;

  iSecondResult = iFirstResult + 10;
  cout << "iSecondResult = iFirstResult  + 10 = "
       <<  iSecondResult << endl ;

  iThirdResult = iSecondResult–5;
  cout << "iThirdResult  = iSecondResult –  5 = "
       <<  iThirdResult << endl ;

  iFourthResult = iThirdResult * 2;
  cout << "iFourthResult = iThirdResult  *  2 = "
       <<  iFourthResult << endl ;

  iFifthResult = iFourthResult / 15;
  cout << "iFifthResult  = iFourthResult / 15 =  "
       <<  iFifthResult << endl ;
}
```

STL <algorithm> Template

The `<algorithm>` template functions and classes work on containers. Although each individual container provides support for its own basic operations, the standard algorithms provide more extended or complex actions. They also allow you to work with two different types of containers at the same time.

The `<algorithm>` template functions allow you to sort, compare, merge, insert, delete, swap, copy, fill, and do many other container element

manipulations. This section is used to introduce STL <algorithm> fundamentals and present several examples you can use to model additional applications. With over sixty `<algorithm>` template functions available, remembering some basic rules will help you to understand the algorithms themselves and how to use them.

The C++ STL provides generic, parameterized, iterator-based functions that implement some common array-based utilities, including searching, sorting, comparing, and editing. The default behavior of the STL algorithms can be changed by specifying a predicate, or modifying a function template. For example, the sort algorithm has a formal third argument specifying the sort type, as in `sort_descending`, or an STL template function like `greater< ... >()`.

Every `<algorithm>` template function operates on a sequence. A sequence is a range of elements in an array or container, or user-defined data structures delimited by a pair of iterators. The identifier *`first`* points to the first element in the sequence. The identifier *`last`* points one element beyond the end of the region you want the algorithm to process.

Many `<algorithm>` function templates use the `(...`*`first`*`,`*`last`*`)` sequence. This syntax implies that the sequence ranges from *`first`* to *`last`*, including *`first`* but not including *`last`*. `<algorithm>` template funtions use the increment operator until it equals `last`. The element pointed to by *`last`* will not be processed by the algorithm.

Certain STL algorithms will create an instance copy of a container. For example, the `reverse(first, last)` template function call adjusts container contents in the original container, while `reverse_copy (first, last)` generates a copy of the container with the adjusted element contents.

Sample Code

The following four programs, `algorth1.cpp`, `algorth2.cpp`, `algorth3.cpp`, and `algorth4.cpp`, demonstrate several of the `<algorithm>` template functions. These examples lay down the fundamental syntax requirements for using the `<algorithm>` template functions and STL iterators (see below).

THE ALGORTH1.CPP APPLICATION

The first example application, `algorth1.cpp`, shows how the `find()` function template can be used to locate the first occurrence of a matching element within a sequence. The syntax for `find()` looks like:

```
template<class InIt, class T>
    InIt find(InIt first, InIt last, const T& val);
```

ch03 5/12/2000 7:59 AM Page 98

`find()` expects two input iterators (discussed later in the chapter) and the address of the comparison value. It returns an input iterator. The program looks like:

```
// algorth1.cpp
// Testing <algorithm>
// find()
// Chris H. Pappas and William H. Murray, 1999

#include <iostream>
#include <algorithm>

using namespace std;

#define MAX_ELEMENTS 5

void main( void )
{
  // simple character array declaration and initialization
  char cArray[MAX_ELEMENTS] = { 'A', 'E', 'I', 'O', 'U' }
;

  char *pToMatchingChar, charToFind = 'I';

  // find() passed the array to search, length +1, and
charToFind ptr
  pToMatchingChar = find(cArray, cArray + MAX_ELEMENTS,
charToFind);

  if( pToMatchingChar!= cArray + MAX_ELEMENTS )
    cout << "The first occurrence of " << charToFind
         << " was at offset " << pToMatchingChar—cArray;
  else
    cout << "Match NOT found!";
};
```

Remember, all you need to do to use any STL template function is use the proper `include` statement:

```
#include <algorithm> // for this chapter
```

and a `using` statement:

```
using namespace std;
```

The program first defines the character array `cArray` and initializes it to uppercase vowels. The program then searches the array, using the `find()` function template for the letter `'I'`, and reports its offset into the `cArray` if found. The output from the program looks like:

```
The first occurrence of I was at offset 2
```

ch03 5/12/2000 7:59 AM Page 99

THE ALGORTH2.CPP APPLICATION

Randomization of data is an extremely important component of many applications, whether it's the random shuffle of a deck of electronic poker cards or truly random test data. The following application uses the `random_shuffle()` template function to randomize the contents of an array of characters. This simple example can be easily modified to work on any container element type.

Note Several of the applications use additional STL templates. Each chapter will emphasize, in the discussion, only those code segments relating to that chapter's STL template. Without this approach, each chapter would endlessly digress. With patience and practice, you will soon understand how the support STL templates work together, in much the same way someone learning to speak a new language may know ***how*** to use a verb without really knowing the details of sentence construction.

The syntax for `random_shuffle()` looks like:

```
template<class RanIt>
    void random_shuffle(RanIt first, RanIt last);
```

`random_shuffle` requires two random access iterator formal arguments (discussed below). The program looks like:

```
// algorth2.cpp
// Testing <algorithm>
// random_shuffle()
// Chris H. Pappas and William H. Murray, 1999

#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

#define MAX_ELEMENTS 5

void main( void )
{
  // typedef for char vector class and iterator
  typedef vector<char> cVectorClass;
  typedef cVectorClass::iterator cVectorClassIt;

  //instantiation of character vector
  cVectorClass cVowels(MAX_ELEMENTS);

  // additional iterators
  cVectorClassIt start, end, pToCurrentcVowels;

  cVowels[0] = 'A';
  cVowels[1] = 'E';
```

```
  cVowels[2] = 'I';
  cVowels[3] = 'O';
  cVowels[4] = 'U';

  start = cVowels.begin();   // location of first cVowels
  end = cVowels.end();       // one past the last cVowels

  cout << "Original order looks like: ";
  cout << "{ ";
  for(pToCurrentcVowels = start; pToCurrentcVowels != end;
      pToCurrentcVowels++)
        cout << *pToCurrentcVowels << " ";
    cout << "}\n" << endl;

    random_shuffle(start, end); // <algorithm> template function

    cout << "Shuffled order looks like: ";
    cout << "{ " ;
    for(pToCurrentcVowels = start; pToCurrentcVowels != end;
          pToCurrentcVowels++)
      cout << *pToCurrentcVowels << " ";
    cout << "}" << endl;
}
```

Notice that the application incorporates the STL templates `<vector>` and `<algorithm>`. The STL `<vector>` template provides the definitions necessary to create the character vector container, while `<algorithm>` defines the `random_shuffle()` template function. The application uses the `<vector>` templates `begin()` and `end()` to locate the front and back offset addresses into the *cVowels* container. These two parameters are then passed to the `random_shuffle()` template function so the algorithm knows where the container starts and ends in memory. The output from the program looks like:

```
Original order looks like: { A E I O U }

Shuffled order looks like: { U O A I E }
```

THE ALGORTH3.CPP APPLICATION

This next application uses the `remove_if()` template function along with the `<functional>` `less_equal()` template to remove any container elements matching the test value. The syntax for `remove_if()` looks like:

```
FwdIt remove_if(FwdIt first, FwdIt last, Pred pr);
template<class InIt, class OutIt, class T>
```

`remove_if()` is passed two forward iterators and a predicate telling `remove_if()` what comparison test to perform. The program looks like:

```
// algorth3.cpp
// Testing <algorithm>
// remove_if()
// Chris H. Pappas and William H. Murray, 1999

#include <iostream>
#include <algorithm>
#include <vector>
#include <functional>

using namespace std;

#define MAX_ELEMENTS 10

void main( void )
{
  typedef vector<int> cVectorClass ;
  typedef cVectorClass::iterator cVectorClassIt;

  cVectorClass iVector(MAX_ELEMENTS);

  cVectorClassIt start, end, pToCurrentint, last;

  start = iVector.begin();   // location of first iVector
  end = iVector.end();       // location of one past last iVector

  iVector[0] =  7;
  iVector[1] = 16;
  iVector[2] = 11;
  iVector[3] = 10;
  iVector[4] = 17;
  iVector[5] = 12;
  iVector[6] = 11;
  iVector[7] =  6;
  iVector[8] = 13;
  iVector[9] = 11;

  cout << "Original order: {";

  for(pToCurrentint = start; pToCurrentint != end;
      pToCurrentint++)
    cout << *pToCurrentint << " " ;
  cout << " }\n" << endl ;

  // call to remove all values less-than-or-equal to the value 11
  last = remove_if(start, end, bind2nd(less_equal<int>(), 11) ) ;

  cout << end-last << " elements were removed.\n" << endl;
```

ch03 5/12/2000 7:59 AM Page 102


```
  cout << "The " << MAX_ELEMENTS-(end-last)
       << " valid remaining elements are: { " ;
  for(pToCurrentint = start; pToCurrentint != last;
      pToCurrentint++)
    cout << *pToCurrentint << " " ;
    cout << " }\n" << endl ;

}
```

While many components of this application are similar to the two previous examples, this program uses the `<functional>` template function `bind2nd()` along with the `less_equal` template class, as the second argument to `remove_if()`, to find all occurrences with values less than or equal to the integer value 11. The output from the program looks like:

```
Original order: { 7 16 11 10 17 12 11 6 13 11 }
6 elements were removed.
The 4 valid remaining elements are: { 16 17 12 13 }
```

Notice that the values `7`, `11`, `10`, `11`, `6`, and `11` were removed, respectively.

THE ALGORTH4.CPP APPLICATION

This last application uses the `<algorithm>` `sort()` and `set_union()` template functions. First, the program instantiates two integer vectors, `iVector1` and `iVector2`, and a third `iUnionedVector` twice the length of the first two. Sorting is necessary for the `set_union()` template function to correctly locate and eliminate all duplicate values. The syntax for `set_union()` looks like:

```
template<class InIt1, class InIt2, class OutIt>
    OutIt set_union(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, OutIt x);
```

`set_union()` uses four input iterators and one output iterator to point to the comparison containers and output the results, respectively. The program looks like:

```
// algorth4.cpp
// Testing <algorithm>
// set_union()
// Chris H. Pappas and William H. Murray, 1999

#include <iostream>
#include <algorithm>
#include <vector>
```

```
#include <functional>

using namespace std;

#define MAX_ELEMENTS 10

void main( void )
{
  typedef vector<int> cVectorClass ;
  typedef cVectorClass::iterator cVectorClassIt;


  cVectorClass iVector1(MAX_ELEMENTS), iVector2(MAX_ELEMENTS),
               iUnionedVector(2 * MAX_ELEMENTS) ;

  cVectorClassIt start1, end1,
                 start2, end2,
                          pToCurrentint, unionStart;

  start1 = iVector1.begin();   // location of first iVector1
  end1 = iVector1.end();       // location of one past last iVector1

  start2 = iVector2.begin();   // location of first iVector2
  end2 = iVector2.end();       // location of one past last iVector2

  // locating the first element address of result union container
  unionStart = iUnionedVector.begin();

  iVector1[0] =  7; iVector2[0] = 14;
  iVector1[1] = 16; iVector2[1] = 11;
  iVector1[2] = 11; iVector2[2] =  2;
  iVector1[3] = 10; iVector2[3] = 19;
  iVector1[4] = 17; iVector2[4] = 20;
  iVector1[5] = 12; iVector2[5] =  7;
  iVector1[6] = 11; iVector2[6] =  1;
  iVector1[7] =  6; iVector2[7] =  0;
  iVector1[8] = 13; iVector2[8] = 22;
  iVector1[9] = 11; iVector2[9] = 18;

  cout << "iVector1 as is : { ";

  for(pToCurrentint = start1; pToCurrentint != end1;
      pToCurrentint++)
    cout << *pToCurrentint << " " ;
  cout << "}\n" << endl ;

  cout << "iVector2 as is : { ";

  for(pToCurrentint = start2; pToCurrentint != end2;
      pToCurrentint++)
```

```
    cout << *pToCurrentint << " " ;
  cout << "}\n" << endl ;

  // sort of both containers necessary for correct union
  sort(start1,end1);
  sort(start2,end2);

  cout << "\niVector1 sorted: { ";

  for(pToCurrentint = start1; pToCurrentint != end1;
      pToCurrentint++)
    cout << *pToCurrentint << " " ;
    cout << "}\n" << endl ;

  cout << "\niVector2 sorted: { ";

  for(pToCurrentint = start2; pToCurrentint != end2;
      pToCurrentint++)
    cout << *pToCurrentint << " " ;
    cout << "}\n" << endl;

  // call to set_union() with all necessary pointers
  set_union(start1,end1,start2,end2,unionStart);

  cout << "After calling set_union()\n" << endl ;

  cout << "iUnionedVector { " ;
  for(pToCurrentint = iUnionedVector.begin();
      pToCurrentint != iUnionedVector.end(); pToCurrentint++)
    cout << *pToCurrentint << " " ;
  cout << "}\n" << endl ;
}
```

The output from the program looks like:

```
iVector1 as is : { 7 16 11 10 17 12 11 6 13 11 }
iVector2 as is: { 14 11 2 19 20 7 1 0 22 18 }
iVector1 sorted: { 6 7 10 11 11 11 12 13 16 17 }
ivector2 sorted: { 0 1 2 7 11 14 18 19 20 22 }
After calling set_union():
iUnionedVector { 0 1 2 6 7 10 11 11 11 12 13 14 16 17 18 19 20 22 0 0 }
```

Notice that the union of `ivector1`'s and `iVector2`'s value of `11` removes their duplicate occurrences, explaining the last two 0s' in `iUnioned-Vector`, indicating null elements.

STL <iterator> Template

The STL `<iterator>` template is an extremely important component of STL. While many of the other STL templates, such as container types, are easier to digest and use, in actuality STL is composed of many interrelated components. Iterators are every bit as important as containers, algorithms, and allocators.

The ANSI/ISO Committee defines an iterator as a generalized pointer that allows a programmer to work with different data structures (or containers) in a uniform manner. Structurally, an iterator is a pointer data type with a few surprising subtleties of its own. Like all pointers, for example:

```
int * pi, or float * pf, or MYCLASS* pmyclass
```

iterators are defined in terms of what they point to. Their syntax:

```
container<Type>::iterator iterator_instance;
```

begins with the type of container they are associated with. The syntax looks slightly different than your standard pointer syntax because it is associated with a template type instead of a simple data type. In the example above, `iterator_instance` is going to be a pointer to elements in a container that holds objects of type `Type`. Dereferencing `iterator_instance` yields a reference to an object of type `Type`.

Since iterators allow algorithms to access the elements of any container type in a uniform way, you can generate code solutions that are easily ported to different application needs. For example, the following example uses the iterators `InIt` and `OutIt` to copy the elements of one container to another:

```
template <class InIt, class OutIt>
OutIt copy(InIt first, InIt last, OutIt copiedContainer)
  {
    while( first != last ) *copiedContainer++ = *first++;
    return copiedContainer;
  }
```

Here, the `copy()` template function works by using the iterator interface. It expects the input and output iterators to provide the basic set of three operations: `operator*()`, which returns a reference to the container's content_type; `operator!=()`, a template function that determines when it's time to exit the main loop; and `operarator++()`, which is used to move to the next container element in both the source and destination containers.

Since all the different iterator types used in the STL support these basic operations, the `<algorithm> copy()` template function can work on *any* container type. With iterators not caring about a container's type or element's type, you do not have to write custom versions of your algorithm for each container class.

Iterator Precedence

The following list of iterators begins with the simplest and most straightforward and progresses to iterators with the most capabilities. At the bottom of the tree are input and output iterators, followed by forward and bidirectional, and finally, at the top of the tree, random access iterators.

INPUT AND OUTPUT ITERATORS

Both input and output iterators are pointers to container elements. The only operations allowed on these iterators are pointer dereferencing and incrementing. Each specific location can only be dereferenced *once* to reference or store an individual container element.

FORWARD ITERATORS

Unlike input and output iterators, forward iterators return a true reference when the pointer is dereferenced so it can be read and written to multiple times.

BIDIRECTIONAL ITERATORS

Bidirectional iterators go forward iterators one better by allowing the pointer to be decremented, not just incremented. Any container that uses this iterator is no longer limited to single-pass algorithms.

RANDOM ITERATORS

As their name implies, random iterators behave most closely to true pointer variables in that they can be both incremented and decremented, or have a constant value added to or subtracted from them.

Iterator Range

If you have had a Data Structures course sometime during your software engineering career, you should be intimately aware of the debugging nightmare pointer variables can introduce into a program. Iterators are no exception.

The biggest problem associated with pointer variables is pointing with a garbage address. To avoid this disaster when you start working with iterators, you need to adopt the STL philosophy of using an interator's range. Most STL template functions that work on a container do so over the range of the container. A container's range is defined by the `begin()` and `end()` template functions, for example:

```
for( vector<float>::iterator pElement = ctr.begin(); pElementlll !=
  ctr.end(); pElement++ )
```

constructs a well-formed **for** loop with the iterator `pElement` always within a valid range.

One important concept to grasp when dealing with iterators, which remember are most closely related to pointer variable types, is that you do *not* compare their values with the standard pointer value NULL, or ask if one iterator is greater than or less than another. Instead, you use iterators in pairs to define the range over which the algorithm operates.

Important Naming Conventions

Not all algorithms support all iterator types. This could present a problem to you, the user of STL, were it not for certain naming conventions. For example, the `sort()` template function needs a random access iterator, not just one starting at `crt.begin()` and ending at `crt.end()`. In an attempt to avoid this type of confusion, the STL uses a standardized naming of the class arguments used to parameterize template functions.

Microsoft uses a very consistent naming convention that you will soon pick up on as you continue through the examples in this chapter and the remainder of the book. For example, look at the following STL definition for the `<algorithm> sort()`:

```
template<class RanIt> void sort(RanIt first, RanIt last);
```

In this definition, `first` and `last` are defined by a random iterator, `RanIt`. The name of an iterator type indicates the category of iterators required for that type. In order of increasing power, the categories are summarized here as:

- `OutIt`—An output iterator X can only have a value V stored indirectly on it, after which it must be incremented before the next reference.
- `InIt`—An input iterator X can represent a singular value that indicates end-of-sequence. If such an iterator does not compare

equal to its end-of-sequence value, it can have a value V accessed indirectly on it any number of times. To progress to the next value, or end-of-sequence, you increment it, as in ++X, X++. Once you increment any copy of an input iterator, none of the other copies can safely be compared, dereferenced, or incremented.

- `FwdIt`—A forward iterator X can take the place of an output iterator or an input iterator. You can read what you just wrote through a forward iterator. And, you can make multiple copies of a forward iterator, each of which can be dereferenced and incremented independently.
- `BidIt`—A bidirectional iterator X can take the place of a forward iterator. You can, however, also decrement a bidirectional iterator, as in – –X, X– –.
- `RanIt`—A random access iterator X can take the place of a bidirectional iterator. You can also perform much the same integer arithmetic on a random access iterator that you can on an object pointer.

Table 3.1 lists the iterators you can use to perform read, write, or read/write container element access.

Table 3.1 *Read, Write, and Read/Write Iterators*

Mode	Legal Iterator Type(s)
Write-only Iterators	Output Iterator
	(**FwdIt**) forward iterator
	(**BidIt**) bidirectional iterator
	(**RanIt**) random access iterator
Read-only Iterators	Input Iterator
	(**FwdIt**) forward iterator
	(**BidIt**) bidirectional iterator
	(**RanIt**) random access iterator
Read/Write Iterators	Forward Iterator
	(**BidIt**) bidirectional iterator
	(**RanIt**) random access iterator

The vectorit.cpp Application

The following application uses three iterators: **vector::begin**, which returns an iterator to start the traversal of the vector; **vector::end**, which returns an iterator for the last element of the vector; and **vector::traverse,** which traverses the vector. The program first creates a vector instance named **iVectorInstance**, then initializes the vector's contents with the offset address into the vector.

```
// vectorit.cpp
// Testing <iterator>
// ::iterator
// Chris H. Pappas and Willliam H. Murray, 1999

#include <iostream>
#include <vector>


using namespace std ;

#define MAX_ELEMENTS 10

typedef vector<int> iVector;

void ShowVector(iVector &iVectorInstance);

void main( void )
{
  iVector iVectorInstance;

  // Initialize vector elements with
  // the offset's value
  for (int offsetAndValue = 0; offsetAndValue < MAX_ELEMENTS;
       offsetAndValue++)
         iVectorInstance.push_back(offsetAndValue);

  // Iterator used to traverse vector elements
  iVector::iterator actualIterator;

  // Output original contents of iVectorInstance.
  cout << "iVectorInstance with the last element: ";
  for (actualIterator  = iVectorInstance.begin();
       actualIterator != iVectorInstance.end();
       actualIterator++)
         cout << *actualIterator << " ";

  cout << "\n\n";
```

ch03 5/12/2000 7:59 AM Page 110


```
  // Use <vector> erase() to delete last value
  iVectorInstance.erase(iVectorInstance.end() -1);

  // Output contents of iVectorInstance.minus last element
  cout << "iVectorInstance minus the last element: ";
  for (actualIterator = iVectorInstance.begin();
       actualIterator != iVectorInstance.end();
       actualIterator++)
         cout << *actualIterator << " ";
}
```

The program finishes by first printing the original vector's contents, then deleting the last element, and finally printing the modified container. The output from the program looks like:

```
iVectoInstance with the last element: 0 1 2 3 4 5 6 7 8 9
iVectorInstance minus the last element 0 1 2 3 4 5 6 7 8
```

The listit.cpp Application

For a change, the following application uses a `<list>` instead of a `<vector>` container to hold a series of words. The program uses a `list::iterator`, a `list::difference_type` (holding the type of element pointers it will be subtracting), and the `advance()` template function, which moves the input iterator *n* elements.

```
// listit.cpp
// Testing <iterator>
// ::iterator
// Chris H. Pappas and William H. Murray, 1999

#include <list>
#include <string>
#include <iostream>

using namespace std ;

typedef list<string> strClassList;

void main( void ) {

  strClassList List;

  // iterator used to traverse list of strings
  strClassList::iterator iteratorForListElements;
```

```
  // list::difference_type describes an object that represents
  // the difference between the addresses of any two elements.

  strClassList::difference_type distance_between;

  List.push_back("Sun,");
  List.push_back("Sand,");
  List.push_back("Ocean,");
  List.push_back("Beach,");
  List.push_back("Lotion,");
  List.push_back("equals ");
  List.push_back("Ahhhhhh!");

  // output the list
  iteratorForListElements = List.begin();
  cout << "The vacation begins: ";
  for( int i = 0; i < 7 ; i++, iteratorForListElements++ )
    cout << *iteratorForListElements  << "  ";

  // Find the first element
  iteratorForListElements=List.begin();

  cout << "\n\nUsing advance() to locate the KEY word :) \n";

  advance(iteratorForListElements,2); // move iterator two elements

  cout << "\nThe magic word is " << *iteratorForListElements << endl;

  // calculating the distance between first and third elements
  distance_between = distance( List.begin(), iteratorForListElements);

  // Output difference
  cout << "\nThe distance between the elements is : " << distance_between;
}
```

Notice that both programs declare container iterators using the identical syntax, just different container types:

```
iVector::iterator actualIterator;
strClassList::iterator iteratorForListElements;
```

The following statement defines what container element type `distance_between` will need to subtract:

```
strClassList::difference_type distance_between;
```

Next, the program uses the `list::push_back()` method to insert an element with the current string constant at the end of the list container. At this point, the list iterator needs the address to the first container element:

ch03 5/12/2000 7:59 AM Page 112


```
iteratorForListElements = List.begin();
```

before it can begin traversing the list and outputting its contents. Once the `for` loop executes, the iterator needs to be reset back to the beginning of the list container so that it may be advanced *n* elements from the beginning.

The `advance()` template function moves the list pointer two elements forward:

```
advance(iteratorForListElements,2); // move iterator two elements
```

The `distance()` template function uses the advanced iterator's address and compares it with the beginning of the list:

```
distance_between = distance( List.begin(), iteratorForListElements);
```

The output from `listit.cpp` looks like:

```
The vacation begins: Sun, Sand, Ocean, Beach, Lotion, equals Ahhhhhh!
Using advance() to locate the KEY word :)
The magic word is Ocean,
The distance between the elements is: 2
```

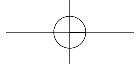
Summary

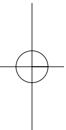
In this chapter, you learned to separate the standard C++ library from the C++ STL. You saw how to facilitate STL—basically, certain components of historic C++ need updating, in particular `iostream.h`. You also looked at the two "support" templates, `<utility>` and `<functional>`, which are frequently combined with additional templates to increase their usability.

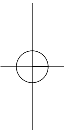
You also explored several of the STL `<algorithm>` template functions as used on integer `<vector>` classes. The applications demonstrated how to find a container element, randomly shuffle container contents, scan a container for certain comparison conditions (less-than-or-equal-to), and perform a union.

Finally, you looked at the similarities and differences between standard C/C++ pointers and STL iterators. The advantage of iterators involves their generic syntax that allows them to work on any container type/element type. You also discovered that when using template functions with iterator arguments, you must pay close attention to the required iterator type (input, output, random, etc.) and Microsoft's naming conventions for each category.

ch03 5/12/2000 7:59 AM Page 113







The good news is that with `<utility>`, `<functional>`, `<algorithm>`, and `<iterator>` STL routines behind you, you are ready to knowledgeably understand how the remaining STL definitions (`<vector>`, `<stack>`, `<queue>`, `<list>`, etc.) interface, providing extremely powerful, portable, and generic code solutions for today's applications.