PH019-Murray01 10/24/00 3:05 PM Page 1

CHAPTER 1

Easing into Standard Template Library (STL) Using Arrays—STL Vector

Congratulations! Yes, *you* deserve a word of praise. By picking up this book you have indirectly implied that you are serious about your understanding of C, C++, Object-Oriented Programming (OOP), and state-of-the-art programming with STL (ANSI/ISO C++ Standard Template Library). Kudos for your desire to learn the most exciting addition to ANSI/ISO C++ and the powerful programming solutions STL enables.

STL is as revolutionary to programming as is Object-Oriented Programming. But like OOP, STL has its own design roots, philosophies, and use. Understanding STL is like jumping into several hundred thousand lines of code, written by other programmers. Your task is to "jump onboard" with the logic, definitions, and syntax that make the whole package function.

What Do I Need to Know to Use This Book?

Simply, an understanding of C++ and familiarity with objects. Each chapter begins with a discussion of how a Data Structures concept is implemented using standard C++ syntax and then moves into the STL counterpart. The introductory section of each chapter reviews and bootstraps your understanding of the Data Structures concept and then rolls that understanding over into the STL equivalent.

Why Do I Need Calculus When I Can Buy a Calculator?

As a student, did you ever ask yourself the question: "Why do I need _____ - 101 (fill in the blank with any required course) when I can buy a piece of technology that does it all

PH019-Murray01 10/24/00 3:05 PM Page 2

for me? The answer you may have been given by your advisor might have gone along the lines, "This course will give you the fundamentals necessary to skillfully employ said technological device." You, on the other hand, may have heard them say: "Yada, yada, yada, just take it," and it wasn't until you obtained your degree and started working that you discovered just how important those fundamentals really are!

The Complexity of Multiplatform Target Environments

From a programming point-of-view, today's development environment is a hundred times more complex than a decade ago. Instead of PC application development targeting a standalone DOS text-mode environment, it must now deal with hundreds of PC clones and other popular competing platforms.

These new architectures have their own evolving operating systems and multitasking, multimedia capabilities. Add to this the typical Internet presence. In other words, today's programming environment, programmed by a single developer, was once the domain of systems, communications, security, networking, and utility specialists all working as a team to keep the "mother ship," or mainframe, up and running!

Something had to come along to enable application developers to keep pace with this ever-increasing resource management nightmare. Voila, enter C and C++. These new languages incorporated brand new programming capabilities to melt through this hidden iceberg of programming demands.

Unintentional Misuse or Ignorance of C/C++ Features

The biggest stumbling block to accessing these incredibly powerful C/C++ features is ignorance of their existence. In the real world, most experienced FORTRAN, COBOL, Pascal, PL/I, and Assembly Language programmers, when asked by their bosses to use a new language, taught themselves the new language! Why? Because, of course, the company wouldn't give them the time off. They diligently studied nights and weekends on their own, and mapped their understanding of whatever language they knew well to the new language's syntax.

This approach worked for decades as long as a programmer went from one "older high-level language," to the next. Unfortunately, when it comes to C/C++, this approach leaves the diligent, self-motivated, learn-on-their-own employee fired and wondering what went wrong *this time?*

Here's a very small example to illustrate the point. In COBOL, for instance, to increment a variable by 1, you would write:

```
accumulator = accumulator + 1;
```

Then one day the boss says you need to write the program in FORTRAN. You learn FORTRAN and rewrite the statement:

PH019-Murray01 10/24/00 3:05 PM Page 3


```
accumulator = accumulator + 1;
```

No problem. Then your company migrates to Pascal and once again you teach yourself the new syntax:

```
accumulator := accumulator + 1;
```

Ta da! Now your boss says that your million dollar code needs to be ported over to Microsoft Windows in C/C++. After a divorce, heart attack, and alcohol addiction you emerge feeling you have mastered Microsoft Windows/C/C++ logic and syntax and finally rewrite the statement:

```
iaccumulator = iaccumulator + 1;  //i for integer in Hungarian notation
```

and you get fired! The senior programmer, hired from a local two-year college, looks at your code and scoffs at your inept translation. Oh, sure, you got the idea behind Hungarian Notation (a C/C++ naming convention that precedes every variable's name with an abbreviation of its data type), but you created a literal statement *translation* instead of incorporating the efficiency alternatives available in C/C++.

Your senior programmer, green, twenty years younger than you, only knowing Microsoft Windows, C and C++ syntax, knew the statement should have been written:

```
iaccumulator++;
```

This statement, using the C/C++ increment operator, efficiently instructs the compiler to delete the double fetch/decode of the incorrectly written *translation*, and to treat the variable *iaccumulator* as its name implies—as an accumulator within a register, a much more efficient machine language encoding.

This extremely simple code example is only the beginning of hundreds of C/C++ language features waiting, like quicksand, to catch the unwary programmer.

Data Structures—The Course to Separate Hackers from Pros!

In a programmer's formal educational path, there stands a course typically called Data Structures, which statistically has an attrition rate of 50%. Why? Because it deals with two extremely efficient concepts—pointers, and dynamic memory allocation/deallocation—which when combined generate a geometric complexity in program development and debugging requirements. These concepts typically present such a steep learning curve that many programmers either avoid the course altogether, or lop along getting by, and then *never* use the concepts in the real world.

This is unfortunate since pointers and dynamic memory allocation present some of the most powerful and efficient algorithms available to a programmer. Enter the Standard Template Library!

PH019-Murray01 10/24/00 3:05 PM Page 4

So, Just What Is the Standard Template Library?

In a nutshell, STL encapsulates the pure raw horsepower of the C/C++ languages plus the advanced efficient algorithms engendered within a good Data Structures course, all bundled into a simple-to-use form! It is similar in a way to having struggled with years of pre-Calc and Calculus courses, only to be given an advanced portable calculator that does all the work for you.

You may view the Standard Template Library as an extensible framework which contains components for language support, diagnostics, general utilities, strings, locales, standard template library (containers, iterators, algorithms, numerics), and input/output.

The Origins of STL

With the ever-increasing popularity of C/C++ and Microsoft Windows-controlled environments, many third-party vendors evolved into extremely profitable commodities by providing libraries of routines designed to handle the storage and processing of data. In an ever-ongoing attempt to maintain C/C++'s viability as a programming language of choice, and to keep the ball rolling by maintaining a strict control of the languages' formal definition, the ANSI/ISO C++ added a new approach to defining these libraries called the Standard Template Library.

STL, developed by Alexander Stepanov and Meng Lee of Hewlett Packard, is expected to become the standard approach to storing and processing data. Major compiler vendors are beginning to incorporate the STL into their products. The Standard Template Library is more than just a minor addition to the world's most popular programming language; it represents a revolutionary new capability. The STL brings a surprisingly mature set of generic containers and algorithms to the C++ programming language, adding a dimension to the language that simply did not exist before.

What Do I Need to Know to Take Advantage of STL?

You have all you need to know right now, simply by picking up this book. Unlike many other STL books which simply enumerate endless lists of STL template names, functions, constants, etc., this book will begin by first teaching you the advanced C/C++ language fundamentals that make the Standard Template Library syntactically possible.

Along the way, this instructional section will show you the syntax that allows an algorithm to be generic; in other words, how C/C++ syntactically separate *what* a program does from the *data type(s)* it uses. You will learn about generic `void *` pointer's strengths and weaknesses, the "better way" with generic types, the "even better way" using templates, and finally, the "best way" with cross-platform, portable, Standard Templates.

The section on template development begins with simple C/C++ structures used syntactically to create *objects* (yes, you can create an object with this keyword; however it is a very

PH019-Murray01 10/24/00 3:05 PM Page 5

bad idea—you'll have to wait until the next chapter to see why)! The `struct` object definition is then evolved over, logically and syntactically, into the C++ `class`. Finally, the `class` object is mutated into a generic `template`. This progressive approach allows you to easily assimilate the new features of C/C++ and paves the way to technically correct use of the STL. With this under your belt, you will logically and syntactically understand how the STL works and begin to immediately incorporate this technology into your application development.

Generic programming is going to provide you with the power and expressiveness of languages like SmallTalk while retaining the efficiency and compatibility of C++. STL is guaranteed to increase the productivity of any programmer who uses it.

A High-Level View of STL

Although the STL is large and its syntax can be initially intimidating, it is actually quite easy to use once you understand how it is constructed and what elements it employs. At the core of the STL are three foundational items called *containers, algorithms*, and *iterators*. These libraries work together allowing you to generate, in a portable format, frequently employed algorithmic solutions, such as array creation, element insertion/deletion, sorting, and element output. But the STL goes even further by providing internally clean, seamless, and efficient integration of iostreams and exception handling.

Kudos to the ANSI/ISO C++ Committee

Multivendor implementations of C/C++ compilers would have long ago died on the vine were it not for the ANSI C/C++ Committees. They are responsible for giving us *portable* C and C++ code by filing in the missing details for the formal language descriptions of both C and C++ as presented by their authors, Dennis Ritchie and Bjarne Stroustrup, respectively. And to this day, it is the ANSI/ISO C++ Committee that continues to guarantee C++'s portability into the next millennium.

The ANSI/ISO C++ committee's current standards exceed their past recommendations which historically decided only to codify existing practice and resolve ambiguities and contradictions among existing translator implementations. The C++ committee's changes are innovations. In most cases, the changes implement features that committee members admired in other languages, features that they view as deficiencies in traditional C++, or simply features that they've always wanted in a programming language. A great deal of thought and discussion has been invested in each change and, consequently, the committee feels that the new C++ definition, along with the evolutionary definition of STL, is the best definition of C++ possible today.

Most of these recommended changes consist of language additions that should not affect existing code. Old programs should still compile with newer compilers as long as the old codes do not coincidentally use any of the new keywords as identifiers. However, even experienced C++ programmers may be surprised at how much C++ has evolved without

PH019-Murray01 10/24/00 3:05 PM Page 6

even discussing STL; take for example, the use of namespaces, new-style type casting, and runtime type information (discussed in detail in Chapter 2).

STL's Tri-Component Nature

Conceptually, STL encompasses three separate algorithmic problem solvers. The three most important are containers, algorithms, and iterators. A *container* is a way that stored data is organized in memory; for example, an array, stack, queue, linked list, or binary tree. However, there are many other kinds of containers, and the STL includes the most useful. The STL containers are implemented by template classes so they can be easily customized to hold different data types.

All the containers have common management member functions defined in their template definitions: `insert()`, `erase()`, `begin()`, `end()`, `size()`, `capacity()`, and so on. Individual containers have member functions that support their unique requirements.

Algorithms are behaviors or functionality applied to containers to process their contents in various ways. For example, there are algorithms to sort, copy, search, and merge container contents. In the STL, algorithms are represented by template functions. These are not member functions of the container classes; they are stand-alone functions. Indeed, one of the surprising characteristics of the STL is that its algorithms are so general. You can use them not only on STL containers, but also on ordinary C++ arrays or any other application-specific container.

A standard suite of algorithms provides searching for, copying, reordering, transforming, and performing numeric operations on the objects in the containers. The same algorithm is used to perform a particular operation for all containers of all object types.

Once you have decided on a container type and data behaviors, the only thing left is to interact the two with *iterators*. You can think of an iterator as a generalized pointer that points to elements within a container. You can increment an iterator, as you can a pointer, so it points in turn to each successive element in the container. Iterators are a key part of the STL because they connect algorithms with containers.

Latest C++ ANSI/ISO Language Updates

While the ANSI/ISO committee was busy incorporating STL, they took the opportunity to introduce modifications to the C++ language definition. These modifications, in most cases, implement features that the committee members admired in other languages, features that they viewed as deficiencies in traditional C++. These new changes, which consist of language additions, should not affect any previously written code.

Using `namespace`

We'll look at the definition for *namespace* from a bottom-up point of view. Namespaces control scope or identifier (constants, variables, functions, classes, etc.) visibility. The tight-

PH019-Murray01 10/24/00 3:05 PM Page 7

est scope is local—those identifiers declared within a function. Also at this level would be member function or method declarations. Higher up on the scale would be class scope.

There are visibility issues associated with file scope, for example, when `1.cpp`, `2.cpp`, and `3.cpp` are combined to generate `123.exe`. Identifiers declared in `1.cpp` are not visible (by default) in `2.cpp` and `3.cpp`.

At the highest level is program or workspace scope. Historically, this worked fine until the advent of today's complex programming environment where source files are coming at you from all directions. Today's programs are a combination of source files you write, those supplied by the compiler(s), some from the operating system itself, and third-party vendors. Under these circumstances, program scope is not sufficient to prevent identifier collisions between categories. Namespaces allow you to lock down all program identifiers, successfully preventing these types of collisions.

Collisions usually fall under the category of external, global identifiers used throughout a program. They are visible to all object modules in the application program, in third-party class and function libraries, and in the compiler's system libraries. When two variables in global scope have the same identifier, the linker generates an error.

Many compiler manufacturers initially solved this problem by assigning unique identifiers to each variable. For example, under standard C, the compiler system prefixes its internal global identifiers with underscore characters, and programmers are told to avoid that usage to avoid conflicts.

Third-party vendors perpended unique mnemonic prefixes to global identifiers in an attempt to prevent collisions. This failed, however, whenever two developers chose the same prefix. The problem is that the language had no built-in mechanism with which a library publisher could stake out a so-called namespace of its own, one that would insulate its global identifiers from those of other libraries being linked into the same application.

Traditionally, a programmer had three choices to eliminate collisions: they could get the source code, modify it, and rebuild it; have the authors of the offending code change *their* declarations; or select an alternate code source containing the same functionality. Not a very pleasant set of alternatives!

The C++ `namespace` keyword limits an identifier's scope to the namespace identifier. All references from outside the block to the global identifiers declared in the block must, in one way or another, qualify the global idenfier's reference with the namespace identifier. In actuality, this is logically similar to perpending prefixes, however, namespace identifiers tend to be longer than the typical two- or three-character prefixes and stand a better chance of working.

`namespace` Syntax

To define a `namespace`, encapsulate your declarations within a `namespace` block, as in:

PH019-Murray01 10/24/00 3:05 PM Page 8


```
namespace your_namespace_name {
  int ivalue;
  class my_class {/*....*/};
  // more declarations;
}
```

In the above example, any code statements within `your_namespace_name` have direct access to the namespace's declarations. However, any code statements outside of `your_namespace_name` must use a qualifying syntax. For example, from the `main()` function, accessing `ivalue` would look like:

```
void main ( void )
{
  your_namespace_name::ivalue++;
}
```

The `using namespace` Statement

If you do not like the idea of always having to qualify an identifier with its namespace everytime you access it, you can employ the `using` statement, as in:

```
using namespace your_namespace_name;
void main ( void )
{
  ivalue++;
}
```

However, this approach can be like giving a hotel guest the key to the entire hotel instead of a single room. The `using namespace` syntax provides access to all of the namespace's declarations; proceed with care. Each application will benefit from the best selection of these two approaches.

The Selective `using` Statement

Somewhere between a fully qualified namespace identifier (`your_namespace_name::ivaluel++;`) and the `using namespace your_namespace_name` syntax, there's the simpler `using` statement. The `using` directive tells the compiler that you intend to use specific identifiers within a namespace. Using the previous examples, this would look like:

```
using your_namespace_name::ivalue;
void main ( void )
{
  ivalue++;
}
```

PH019-Murray01 10/24/00 3:05 PM Page 9

Just as a programmer would not choose to always use `for` loops when there are `while` and `do-while` alternatives, so too a programmer should carefully select the best, application-specific approach to namespace identifier access.

Renaming namespaces

Sometimes third-party `namespace` names can get in your way because of their length; for example, `your_namespace_name` is quite long. For this reason the `namespace` feature allows a programmer to associate a new name with the `namespace` identifier, as in:

```
namespace YNN = your_namespace_name;
void main ( void )
{
  YNN::ivalue++;
}
```

`static` File Scope vs. Unnamed namespaces

One way to enforce file scope is with the keyword `static`. For example, if `1.cpp`, `2.cpp`, and `3.cpp` all have the external variable declaration int `ivalue;`, and you do not want internal linkage (meaning all three identifiers *share* the same storage location), precede all three declarations with the keyword `static`:

```
// 1.cpp                // 2.cpp                      // 3.cpp
static int ivalue;      static int ivalue;         static int ivalue;
void main ( void )      void some_funcs( void );   void more_funcs( void );
```

Unnamed namespaces provide the same capability, just a slightly different syntax:

```
// 1.cpp
namespace {
  int ivalue;
}
void main ( void )
{
  ivalue++;
}
```

To create an unnamed namespace, simply omit a namespace identifier. The compiler then generates an internal identifier that is unique throughout the program. All identifiers declared within an unnamed namespace are available only within the defining file. Functions in other files, within the program's workspace, cannot reference the declarations.

PH019-Murray01 10/24/00 3:05 PM Page 10

New Casting Operations

As traditional style casting proves to be unsafe, error-prone, and difficult to spot when reading programs, and even more challenging when searched for in large bodies of source code, the newer style cast is a huge improvement. There are four new types of casts. The general syntax looks like:

```
cast_operator <castType> (objectToCast)
```

Dynamic Casting

You use a `dynamic_cast` whenever you need to convert a base class pointer or reference to a derived class pointer or reference. The one restriction is that the base, parent, or root class must have at least one *virtual* function. The syntax for a `dynamic_cast` looks like:

```
dynamic_cast < castType > ( objectToCast );
```

This type of cast allows a program, at run time, to determine whether a base class pointer or reference points to an object of a specific derived class or to an object of a class derived from the specified class.

You can also use `dynamic_cast` to upcast a pointer or reference to a derived class to a pointer or reference to one of the base, parent, or root classes in the same hierarchy. Upcasting allows a program to determine, at run time, whether a pointer to a derived class really contains the address of an object of that class. At the same time, you want to force the address into a pointer of one of the object's ancestor classes.

Static Casting

A `static_cast` implicitly converts between types that are not in the same class hierarchy. The type-checking is static, where the compiler checks to ensure that the conversion is valid as opposed to the dynamic run-time type checking that is used with `dynamic_cast`s. The syntax for a `static_cast` looks like:

```
static_cast < castType > ( objectToCast );
```

The `static_cast` operator can be used for operations such as converting a pointer to a base class to a pointer to a derived class. Such conversions are not always safe. For example:

PH019-Murray01 10/24/00 3:05 PM Page 11


```
class typeA { ... };

class typeB : public typeA { ... };

void someFunction(typeA* ptypeA, typeB* ptypeB)
{
    typeB* ptypeB2 = static_cast<typeB*>(ptypeA);  // not safe, ptypeB may
                                                   // point to just typeB

    typeA* ptypeA2 = static_cast<typeA*>(ptypeB);  // BETTER - this is a
                                                   // safe conversion
    ...
}
```

In this code segment the object pointed to by `ptypeA` may not be an object of `typeB`, in which case the use of `*ptypeB2` could be disastrous. For instance, calling a function that is a member of the `typeB` class, but not the `typeA` class, could result in an access violation.

Newer C-Type Cast

The `reinterpret_cast` operator replaces many of the older C-type casts except those removing an identifier `const` restriction. The `reinterpret_cast` is capable of converting one pointer type into another—numbers into pointers, and pointers into numbers. The syntax looks like:

```
reinterpret_cast < castType > ( objectToCast );
```

The `reinterpret_cast` operator can be used for conversions such as `char*` to `int*`, or `Base_class*` to `anyOtherNONrelated_class*`, which are inherently unsafe.

The result of a `reinterpret_cast` cannot safely be used for anything other than being cast back to its original type. Other uses are, at best, nonportable. The following code segment demonstrates a pointer type cast in C, C++, and C++ using the `reinterpret_cast` syntax:

```
void main ( void )
{
  int * pointer_to_int;
  /* in C */
  pointer_to_int = malloc(100); /* implicit void * cast to int *,
                                   with warning */
  pointer_to_int = (int *) new int[100]; // C++ required cast of void *
                                            to int *
  pointer_to_int = reinterpret_cast<int *>( new int[100]); // new style cast
}
```

PH019-Murray01 10/24/00 3:05 PM Page 12

Constant Cast

The `const_cast` operator can be used to remove the `const`, `volatile`, and `__unaligned` attribute(s) from a class. The general syntax looks like:

```
const_cast < castType > ( objectToCast )
```

With a `const_cast`, your program can cast a pointer to any object type or a pointer to a data member, to a type that is identical except for the `const`, `volatile`, and `__unaligned` qualifiers. For pointers and references, the result will refer to the original object. For pointers to data members, the result will refer to the same member as the original (uncast) pointer to data member.

Run-Time Type Information (RTTI)

The `typeid` operator, found in `<typeinfo>`, supports the new C++ run-time type information feature. The operator returns a reference to a system-maintained object of the type `type_info`, which identifies the type of the argument. RTTI was added to the C++ language because many vendors of class libraries were implementing this functionality themselves. This caused incompatibilities between libraries. Thus, it became obvious that support for run-time type information was needed at the language level.

The following code segment demonstrates their straightforward syntax and logical use:

```
#include <typeinfo>
#include <iostream>
using namespace std;
// ...
class someClassType { };
// ...
  someClassType sCTinstance;
  int ivalue;
  cout << "object type = " << typeid(sCTinstance).name(); // type's name
  if ( typeid ( ivalue ) == typeid ( sCTinstance ) ) // type comparisons
    cout << "I DON'T BELIEVE IT!";
```

If this code section defined and then dynamically created a plethora of object types and instances, knowing which dynamically allocated type was in use at any point within an algorithm would be an extremely useful piece of run-time logic control.

Introduction to the Standard C++ Library

No matter how sophisticated your procedural or object-oriented, stand-alone, or Windows application is, as a programmer you must invariably resort to tried and true Data Structures algorithms for creating linked-lists, stacks, queues, and binary trees. By necessity, you have again reinvented the wheel for each application's unique user-defined data types. Along

PH019-Murray01 10/24/00 3:05 PM Page 13

with this application-specific proprietary code comes the developer's nightmare—design changes that are not easily added in such cases, and code maintenance.

Unlike many other object-oriented languages, such as SmallTalk, if these common programming components had been part of the C++ language, you could avoid re-inventing data-specific algorithms. In a nutshell, that is what STL is all about. Finally, the C++ language provides you with general purpose components for common programming tasks through the Standard C++ Library.

Some of the more interesting, reusable components of the Standard C++ Library include powerful and flexible containers and programmable algorithms. The facilities provided by the Standard C++ Library are as follows:

- **C++ Language Support—**includes common type definitions used throughout the library such as predefined types, C++ program start and termination function support, support for dynamic memory allocation, support for dynamic type identification, and support for exception processing.
- **Diagnostic Tools**—provides components for reporting several kinds of exceptional conditions, components for documenting program assertions, and a global variable for error number codes.
- **General C/C++ Utilities**—provides components used by other elements of the Standard C++ Library. This category also includes components used by the Standard Template Library (STL) and function objects, dynamic memory management utilities, and date/time utilities. These components may also be used by any C++ program. The general C/C++ utilities also include memory management components derived from the C library.
- **Strings—**includes components for manipulating sequences of characters, where characters may be of type `char`, `w_char` (used in UNICODE applications), or of a type defined in a C++ program. UNICODE uses a two-byte value to represent every known written language's symbol set versus the limiting one-byte ASCII code.
- **Cultural Formatting Support**—provides numeric, monetary, and date/time formatting and parsing and support for character classification and string collation.
- **STL (Standard Template Library)**—includes the most widely used algorithms and data structures in data-independent format. STL headers can be grouped into three major organizing concepts: containers, iterators, and algorithms. *Containers* are template classes that provide powerful and flexible ways to organize data; for example, vectors, lists, sets and maps. *Iterators* are the glue that paste together algorithms and containers.
- **Advanced Numerical Computation**—includes seminumerical operations and components for complex number types, numeric arrays, and generalized numeric algorithms.

PH019-Murray01 10/24/00 3:05 PM Page 14

- **Input/Output—**includes components for forward declarations of iostreams, predefined iostream objects, base iostream classes, stream buffering, stream formatting and manipulators, string streams, and file streams.
- The Standard C++ Library also incorporates the Standard C Library.

The Standard C++ Libraries

An application accesses Microsoft's Standard C++ Library facilities by using appropriate include files and associated static and dynamic libraries. Tables 1.1 and 1.2 list all the Standard C++ Library headers and the associated static and dynamic libraries provided by Visual C++.

Table 1.1 The Standard C++ Library Headers

algorithm	bitset	cassert	cctype	cerrno
cfloat	ciso646	climits	clocale	cmath
complex	csetjmp	csignal	cstdarg	cstddef
cstdio	cstdlib	cstring	ctime	cwchar
cwctype	deque	exception	fstream	functional
iomanip	ios	iosfwd	iostream	istream
iterator	limits	list	locale	map
memory	new	numeric	ostream	queue
set	sstream	stack	stdexcept	streambuf
string	strstream	utility	valarray	vector

Note

Remember, the Standard C++ Library headers have no .h file extension in accordance with the latest ANSI/ISO C++ standard.

Microsoft also provides the static and dynamic libraries as shown in Table 1.2.

PH019-Murray01 10/24/00 3:05 PM Page 15

Table 1.2 Static and Dynamic Libraries Included with Microsoft Visual C++

Library types	*C runtime library*	*Standard C++ Library*	*Old iostream library*
Single-Threaded	LIBC.LIB	LIBCP.LIB	LIBCI.LIB
Multithreaded	LIBCMT.LIB	LIBCPMT.LIB	LIBCIMT.LIB
Multithreaded DLL version	MSVCRT.LIB (uses MSVCRT.DLL)	MSVCPRT.LIB (uses MSVCRT.DLL)	MSVCIRT.LIB (uses MSVCIRT.DLL)
Debug Single-Threaded	LIBCD.LIB	LIBCPD.LIB	LIBCID.LIB
Debug Multithreaded	LIBCMTD.LIB	LIBCPMTD.LIB	LIBCIMTD.LIB
Debug Multithreaded	MSVCRTD.LIB (uses MSVCRT.DLL)	MSVCPRTD.LIB (uses MSVCRT.DLL)	MSVCIRTD.LIB (uses MSVCIRT.DLL)

Your First Standard C++ Library Application

To make you feel right at home, the following C++ program uses the Standard C++ Library `iostream` to print "Hello World!":

```
#include <iostream>
void main( void )
{
    cout << "Hello World!";
}
```

The code segment used the Standard C++ Library input/output component to print "Hello World!" by simply including the Standard C++ Library header `<iostream>`. It is important to remember that starting with Visual C++ 4.0, a C++ program, depending on the run-time library compiler option specified (/ML[d], /MT[d], or /MD[d]), will always link with one Basic C run-time library and, depending on headers included, will link with either a Standard C++ Library (as in the example program above) or the old `iostream` library (as in the following coded example):

```
#include <iostream.h>
void main( void )
{
  cout << "Hellow World!";
}
```

Implementing Your Own Template

Frequently, a C++ program uses common data structures such as stacks, queues, and linked-lists. Imagine a program that requires a queue of customers and a queue of messages. You

PH019-Murray01 10/24/00 3:05 PM Page 16

could easily implement a queue of customers, and then take the existing code and implement a queue of messages.

If the program grows and there is a need for a queue of orders you could take the queue of messages and convert it to a queue of orders. But what if you need to make some changes to the queue implementation? This would not be a very easy task because the code has been duplicated in many places. Reinventing source code is not an intelligent approach in an object-oriented environment that encourages reusability. It seems to make more sense to implement a queue that can contain any arbitrary type rather than duplicating code. How does one do that? The answer is to use *type parameterization*, more commonly referred to as *templates.*

Templates are very useful when implementing generic constructs such as vectors, stacks, lists, and queues that can be used with any arbitrary type. C++ templates provide a way to reuse source code, as opposed to inheritance and composition, which provide a way to reuse object code.

C++ provides two types of templates: *class templates* and *function templates*. Use function templates to write generic functions; for example, searching and sorting routines that can be used with arbitrary types. The Standard Template Library generic algorithms have been implemented as function templates and the containers have been implemented as class templates.

Your First `class` Template

The good news is that a `class` template definition looks very similar to a regular `class` definition, except for the prefix keyword `template`. The following example defines a `stack class` template, independent from any stack element type definitions:

```
template <class T>

class genericStack
{
public:
       genericStack( sizeOfStack = 25);
       ~genericStack() { delete[] stackPtr; }
       int pushElement(const T&);
       int popElement(T&) ;
       int isStackEmpty()const { return stackTop == -1 ; }
       int isStackFull() const { return stackTop == sizeOfStack - 1 ; }
private:
       int sizeOfStack ;          // number of stack elements
       int stackTop ;
       T* stackPtr ;
} ;
```

PH019-Murray01 10/24/00 3:05 PM Page 17

`T` represents any data type. It is important to note that `T` does not have to be a class type as implied by the keyword class. `T` can be anything from a simple data type like `int`, to a complex data structure like `pToArrayOfStructures *`.

Function Templates Requirements

Implementing template member functions is somewhat different than implementing the regular class member functions. The declarations and definitions of the class-template member functions should all be in the same header file. Why do the declarations and definitions need to be in the same header file? Consider the following:

```
//sample.h
template <class t>
class sample
{
public:
    sample() ;
    ~sample() ;
} ;
```

```
//sample.cpp
#include "sample.h"
template <class t>
sample<t>::sample()
{
}
template <class t>
sample<t>::~sample()
{
}
```

```
//main.cpp
#include "sample.h"
void main( void )
{
     sample<int> si ;
     sample<float> sf ;
}
```

When compiling `sample.cpp`, the compiler has both the declarations and the definitions available. At this point, the compiler does not need to generate any definitions for template classes, since there are no instantiations. When the compiler compiles `main.cpp`, there are two instantiations: template classes `sample<int>` and `sample<float>`. At this point, the compiler has the declarations but no definitions!

Using a `class` Template

Using a `class` template is very easy. Create the required classes by plugging in the actual type for the type parameters. This process is commonly known as *instantiating* a class. Here is a sample class that uses the `genericStack` `class` template:

```
#include <iostream>
#include "stack.h"
using namespace std;
void main( void )
{
  typedef genericStack<float> floatStack;
  typedef genericStack<int> intStack;
  FloatStack actualFloatStackInstance( 10 );
```

PH019-Murray01 10/24/00 3:05 PM Page 18

In the above example we defined a class template, `genericStack`. In the program we instantiated a `genericStack` of `float` (`floatStack`) and a `genericStack` of `int` (`intStack`). Once the template classes are instantiated, you can instantiate objects of that type (for example, `actualFloatStackInstance`).

It is good programming practice to use `typedef` while instantiating template classes. Then, throughout the program, you can use the `typedef` name. There are two advantages to this method. First, `typedef`s are helpful when nesting template definitions. For example, when instantiating an `int` STL vector, you could use:

```
typedef vector<int, allocator<int> > intVECTOR ;
```

Secondly, should the template definition change, simply change the `typedef` definition. This practice is especially helpful when using STL components.

`class` Template Parameters

The `genericStack` `class` template, described in the previous section, used only type parameters in the template header. It is also possible to use nontype parameters. For example, the template header could be modified to take an `int number_of_elements` parameter as follows:

```
#define MAX_ELEMENTS 32
template <class T, int number_of_elements>
class genericStack ;
```

Then, a declaration such as:

```
genericStack<float, MAX_ELEMENTS> currentStackSize;
```

could instantiate (at compile time) a `MAX_ELEMENTS` `genericStack` template class named `currentStackSize` (of float values); this template class would be of type `genericStack<float, 32>`.

Default Template Parameters

Let's look at the `genericStack` `class` template again:

```
template <class T, int number_of_elements> genericStack { ....};
```

C++ allows you to specify a *default template parameter*, so the definition could now look like:

```
template <class T = float, int number_of_elements = 10> genericStack { ....};
```

PH019-Murray01 10/24/00 3:05 PM Page 19

Then a declaration such as:

```
genericStack<> defaultStackSize;
```

would instantiate (at compile time) a 10 element `genericStack` template class named `defaultStackSize` (of float values); this template class would be of type `genericStack<float, 10>`.

If you specify a default template parameter for any formal parameter, the rules are the same as for functions and default parameters. Once you begin supplying a default parameter, all subsequent parameters must have defaults.

The Standard Template Library

The Standard Template Library is part of the Standard C++ Library. Every C++ programmer at one time or another has implemented common data structures such as a vector, list, or queue, and common algorithms such as binary search, sort, and so on. Through STL, C++ gives programmers a set of carefully designed generic data structures and algorithms.

These generic data structures and algorithms are parameterized types (templates) that require only plugging in of actual types to be ready for use. Finally, STL brings to C++ the long promised goal of reusable software components. More importantly, the STL substructure generates extremely efficient code size and performance.

STL Components

- **Containers** are objects that store other objects.

  sequential containers include:

  `vector`
  `list`
  `deque`

  associative containers include:

  `map`
  `multimap`
  `set`
  `multiset`

- **Algorithms** include generic functions that handle common tasks such as searching, sorting, comparing, and editing.
- **Iterators** are generic pointers used to interface containers and algorithms. STL algorithms are written in terms of iterator parameters, and STL containers provide iterators that can be plugged into the algorithms. Iterators include:

PH019-Murray01 10/24/00 3:05 PM Page 20


```
input
output
forward
bidirectional
random access
istream_iterator
ostream_iterator
```

- **Function templates** are objects of any class or struct that overload the function call `operator()`. Most STL algorithms accept a function object as a parameter that can change the default behavior of the algorithm. Function template argument types include:

```
plus
minus
times
divides
```

  The unary, logical, bitwise, and comparison operator object types include:

```
modulus
negate
equal_to
not_equal_to
greater
less
greater_equal
less_equal
logical_and
logical_or
logical_not
```

- **Adapters** modify the interface of other components. There are three kinds of STL adapters:

  Container Adapter: `stack`, `queue`, `priority_queue`

  Iterator Adapter: `reverse_bidirectional_iterator`, `back_insert_iterator`, `front_insert_iterator`, and `insert_iterator`

  Function Adapters: `not1`, `not2`, `bind1st`, and `bind2nd`

The individual STL libraries and glue components are designed to work together in useful ways to produce the kind of larger and more specialized algorithms needed in today's applications.

PH019-Murray01 10/24/00 3:05 PM Page 21

Rules for Using STL

The following sections highlight fundamental principles employed by the C++ compiler when using STL components. For example, when an object is used with an STL container, it is first copied with a call to the copy constructor, and the copy is what is actually inserted into the container. This means an object held by an STL container must have a copy constructor. If an application removes an STL container object, the object is destroyed with a call to the destructor. If the STL container is destroyed, it destroys all objects it currently holds.

Frequently, STL components use compare container objects with a complete set of logical tests such as `<`, `<=`, `>`, `>=`, `==`, and `!=`. This means the comparison operators must be defined for objects used with an STL component. Of course, some STL components modify the value of an object. This is accomplished using the assignment operator. This means the assignment operator, `=`, must be defined for objects used with an STL component. Your application can use the `<utility>` definitions of the `<=`, `>`, `>=`, and `!=` operators, which are all defined in terms of the `<` and `==`.

The simplest way for you to guarantee that your objects will sucessfully interact with the STL containers is to make certain your objects contain:

1. A copy constructor
2. An assignment operator, `=`
3. An equality comparison operator, `==`
4. A less than comparison operator, `<`

Function Objects

Function objects are relatively new to the C++ programming language. Their usage may seem odd at first glance, and the syntax may appear to be confusing. A function object is an object of a class or struct type that includes an `operator()` member function. An `operator()` member function allows the creation of an object that behaves like a function. For example, a two-dimensional `Array2D` class could overload `operator()` to access an element whose row and column index are specified as arguments to `operator()`.

```
class Array2D
{
  public:
    Array2D(int, int);
    int operator(int, int) const;
  private:
    Array<int> Array2Def;
    int rowOffset;
    int colOffset;
} ;
```

PH019-Murray01 10/24/00 3:05 PM Page 22


```
int Array2D::operator(int currentRow, int currentCol) const
{
  if ( currentRow > 0 && currentRow <= rowOffset && currentCol > 0 &&
currentCol <= colOffset)
    return Array2Def[currentRow, currentCol];
  else
    return (0);
}
Array2D intArray2D(10, 10);
int  intArray2D_element = intArray(5, 5);
```

It is important to note that function objects are objects that behave like functions, and, as such, they can be created and must always return a value.

STL Function Objects

The Standard Template Library provides function objects for standard math operations such as addition, subtraction, multiplication, and division. STL also provides function objects for unary operations, logical operations, bitwise operations, and comparison operations. Chapter 2 lists all the function objects defined in the STL header file `<functional>`.

The following example program demonstrates how these function objects can be used with STL algorithms to change their default behavior. The STL sort algorithm, by default, sorts in ascending order. However, by using the `greater(T)` function object, you can "trick" the sort algorithm to work in descending order:

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

void main( void )
{
  typedef vector<int, allocator<int> > iVECTOR;
  int iArray[5] = {10, 15, 22, 31, 18, 5};
  iVECTOR iVectorInstance(iArray, iArray + 5) ;

// default ascending sort
  copy(iVectorInstance.begin(), iVectorInstance.end(), out) ;
  sort(iVectorInstance.begin(), iVectorInstance.end()) ;
// use of function object to reverse sort to descinding

  copy(iVectorInstance.begin(), iVectorInstance.end(), out) ;
  sort(iVectorInstance.begin(), iVectorInstance.end(), greater<int>())
;
 }
```

PH019-Murray01 10/24/00 3:05 PM Page 23

Basically, it is the `greater<int>()` function object passed to `sort()` that inverts the comparison logic used by `sort()` thereby tricking it into a descending algorithm. You can also use STL function objects directly in a C++ program. The following two statements:

```
float floatCalculation = (times<float>())(1.1, 2.2);     // assigns 2.42
int intCalculation = (minus<int>())(15, 5);              // assigns 10
```

access the `times` and `minus` function objects directly.

STL Function Adapters

Function adapters help us construct a wider variety of function objects using existing function objects. Using function adapters is often easier than directly constructing a new function object type with a struct or class definition. STL provides three categories of function adapters: *binders*, *negators*, and *pointer-to-function adapters*.

Binders are function adapters that convert binary function objects into unary function objects by binding an argument to some particular value. STL provides two types of binder function objects: `binder1st<Operation>` and `binder2nd<Operation>`. A binder function object takes only a single argument. STL provides two template functions, `bind1st` and `bind2nd`, to create binder function objects.

The functions `bind1st` and `bind2nd` each take as arguments a binary function object *f* and a value *x*. `bind1st` returns a function object of type `binder1st<Operation>`, and `bind2nd` returns a function object of type `binder2nd<Operation>`. Here are the function prototypes for the `bind1st` and `bind2nd` functions:

```
template <class Operation, class T>
binder1st<Operation> bind1st(const Operation& f, const T& x) ;
template <class Operation, class T>
binder2nd<Operation> bind2nd(const Operation& f, const T& x) ;
```

Look at this first example:

```
int iGreater = (bind2nd(greater<int>(), 15))(actual_int_value)
```

If you assume `actual_int_value` is defined as type `int`, the above statement could have been rewritten as:

```
int iGreater = (greater<int>())(actual_int_value, 15)
```

Negators are the second type of function adaptors. Negators are used to return the complement of a result obtained by applying a provided unary or binary operation. STL provides two types of negator function objects: `unary_negate<Operation>` and `binary_negate<Operation>`. A negator function object takes only a single argument.

PH019-Murray01 10/24/00 3:05 PM Page 24

The two template functions, `not1` and `not2`, create negator function objects. The function `not1` takes a unary function object *f* as its argument and returns a function object of type `unary_negate<Operation>`. The function `not2` takes a binary function object *f* as its argument and returns a function object of type `binary_negate<Operation>`. Here are the function prototypes for the `not1` and `not2` functions:

```
template <class Operation>
unary_negate<Operation> not1(const Operation& f) ;
template <class Operation>
binary_negate<Operation> not2(const Operation& f) ;
```

The following code segment demonstrates `find_if()` used with the `not1` function used to locate the first element in the array that is not greater than 15:

```
int iArray[25] = {/* your initialization code here */};
int* offset = find_if(iArray, iArray+25,
not1(bind2nd(greater<int>(), 15)));
```

The function `bind2nd` creates a unary function object which returns the result of the comparison `int > 15`. The function `not1` takes the unary function object as an argument and creates another function object. This function object merely negates the results of the comparison `int > 15`. This next statement uses the `not2` function in a game playing match by causing the `greater` function object to trigger an ascending `sort` order:

```
sort(iVectorInstance.begin(), iVectorInstance.end(), not2(greater<int>())) ;
```

STL provides two types of *pointer-to-function objects*: `pointer_to_unary_function<Arg, Result>` and `pointer_to_binary_function<Arg1, Arg2, Result>`. An application can use adapters for pointers to functions to convert existing binary or unary functions to function objects. Adapters for pointers to functions allow the programmer to utilize the existing code to uniquely extend the library.

The `pointer_to_unary_function` function object takes one argument of type `Arg`, and `pointer_to_binary_function` takes two arguments of type `Arg1` and `Arg2`. STL provides two versions of the template function `ptr_fun` to create pointer-to-function function objects.

The first version of `ptr_fun` takes a unary function *f* as its argument and returns a function object of type `pointer_to_unary_function<Arg, Result>`. The second version of `ptr_fun` takes a binary function *f* as its argument and returns a function object of type `pointer_to_binary_function<Arg1, Arg2, Result>`. Here are the function prototypes for the `ptr_fun` functions:

PH019-Murray01 10/24/00 3:05 PM Page 25


```
template<class Arg, class Result>
    class pointer_to_unary_function
        : public unary_function<Arg, Result> {
public:
    explicit pointer_to_unary_function(Result (*pf)(Arg));
    Result operator()(const Arg x) const;
};
```

The template class stores a copy of `pf`. It defines its member function `operator()` as returning `(*pf)(x)`.

Standard Template Library Algorithms

This section serves to introduce the STL algorithm fundamentals and presents some examples. Remembering some basic rules will help you to understand the algorithms and how to use them. STL provides generic parameterized, iterator-based functions (a fancy description for template functions). These functions implement some common array-based utilities, including searching, sorting, comparing, and editing. The STL algorithms are user-programmable. What this means is that you can modify the default behavior of an algorithm to suit your needs, as in the example:

```
sort(first, last) ;          //sorts elements of a sequence
                             //in ascending order by default.
```

In this case, the STL algorithm assumes an operator == or operator < exists, and uses it to compare elements. The default behavior of the STL algorithms can be changed by specifying a predicate. The predicate function could be a C++ function. For example, `sort_descending` is a C++ function that compares two elements. In this case, the sort algorithm takes a function pointer, as follows:

```
sort(first, last, sort_descending);
```

Or, the predicate function could be a function object. Either define a function object, or use the function objects provided by STL. For example (as seen earlier):

```
sort(first, last, greater<int>());
```

Every algorithm operates on a range of 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 points last to the one element beyond the end of the region you want the algorithm to process. A common notation used to represent the sequence is `[first, last)`. This is a notation for an open interval. The notation `[first, last)` implies that the sequence ranges from first to last, including first but not including last. The algorithm will increment an internal iterator with the ++ operator until it equals last. The element pointed to by last will not be processed by the algorithm.

PH019-Murray01 10/24/00 3:05 PM Page 26

STL algorithms do not perform range or validity checking on the iterator or pointer values. Many algorithms work with two sequences. For example, the copy algorithm takes three parameters, as follows:

```
copy(firstValue1, lastValue1, firstValue2);
```

If the second sequence is shorter than the first, `copy` will blindly continue writing into unconnected areas of memory. Some STL algorithms also creates an in-place version and a copying version. For example:

```
reverse(first, last);            // places results in original container
reverse_copy(firstValue1, lastValue1, firstValue1);  // results in
                                                     // duplicate location
```

The STL generic algorithms can be divided into the following four main categories: *Nonmutating*—sequence algorithms operate on containers without modifying the contents of the container. *Mutating*—sequence algorithms typically modify the containers on which they operate. *Sorting*—related algorithms include sorting and merging algorithms, binary searching algorithms, and set operations on sorted sequences. Finally, there is a small collection of *generalized* numeric algorithms defined in the files: `<algorithm>`, `<functional>`, `<numeric>`.

Standard C++ Library Language Support

The language support section of the Standard C++ Library provides common type definitions used throughout the library, characteristics of predefined types, functions supporting start and termination of C++ programs, support for dynamic memory allocation, support for dynamic type identification, support for exception processing, and other run-time support.

`cstddef`

This header file basically includes `stddef.h`. There are two macros, `NULL` and `offsetof`, and two types, `ptrdiff_t` and `size_t`, specifically listed in this section of the standard. To determine the distance (or the number of elements) between two elements you can use the `distance()` function. If you pass it, an iterator pointing to the first element and one pointing to the third element, it will return a 2. The distance function is in the utility header file; it takes two iterators as parameters and returns a number of type `difference_type`. `Difference_type` maps is an `int`.

Implementation Properties: `limits`, `climits`, `cfloat`

The `numeric_limits` component provides information about properties of fundamental types. Specializations are provided for each fundamental type such as `int`, `floating point`,

PH019-Murray01 10/24/00 3:05 PM Page 27

and `bool`. The member, `is_specialized`, returns true for the specializations of `numeric_limits` for the fundamental types. The `numeric_limits` class is defined in the limits header file, as shown here:

```
template<class T>  class numeric_limits {
public:
static const bool has_denorm;
static const bool has_denorm_loss;
static const bool has_infinity;
static const bool has_quiet_NaN;
static const bool has_signaling_NaN;
static const bool is_bounded;
static const bool is_exact;
static const bool is_iec559;
static const bool is_integer;
static const bool is_modulo;
static const bool is_signed;
static const bool is_specialized;
static const bool tinyness_before;
static const bool traps;
static const float_round_style round_style;
static const int digits;
static const int digits10;
static const int max_exponent;
static const int max_exponent10;
static const int min_exponent;
static const int min_exponent10;
static const int radix;
static T denorm_min() throw();
static T epsilon() throw();
static T infinity() throw();
static T max() throw();
static T min() throw();
static T quiet_NaN() throw();
static T round_error() throw();
static T signaling_NaN() throw();
};
```

Exception Handling

The C++ Standard Library exception class defines the base class for the types of objects thrown as exceptions. The exception header file defines the exception class that is the base class for all exceptions thrown by the C++ Standard Library. The following code would catch any exception thrown by classes and functions in the Standard C++ Library:

PH019-Murray01 10/24/00 3:05 PM Page 28


```
try {
   // your code here
}
catch ( const exception &ex)
{
   cout << "exception: " << ex.what();
}
```

The exception class is defined in the header file exception, as follows:

```
class exception {
public:
    exception() throw();
    exception(const exception& rhs) throw();
    exception& operator=(const exception& rhs) throw();
    virtual ~exception() throw();
    virtual const char *what() const throw();
private:
// ...
};
```

Additional Support

Each of these headers files—`cstdarg`, `csetjmp`, `ctime`, `csignal`, and `cstdlib`—includes the corresponding C header file, `stdarg.h`, `setjmp.h`, `time.h`, `signal.h`, and `stdlib.h`. Macros, types, and functions for each of these in the Standard C++ Library are listed in Table 1.3.

Table 1.3 Macros, Types, and Functions

File	*Macros*	*Types*	*Functions*
cstdarg	va_arg, va_end,va_start	va_list	
csetjmp	Macro: setjmp	jmp_buf	longjmp
ctime	CLOCKS_PER_SEC	clock_t	clock
csignal	SIGABRT, SIGILL, SIGSEGV, SIG_DFL, SIG_IGN, SIGFPE, SIGINT, SIGTERM, SIG_ERR	sig_atomic_t	raise, signal
cstdlib	getenv, system		

PH019-Murray01 10/24/00 3:05 PM Page 29

STL Review

The following review is included to help you formalize the structural components of the Standard Template Library. You can logically divide the Standard Template Library into the following categories:

A) STL headers can be grouped into three major organizing concepts:
   1) *Containers* are template classes that support common ways to organize data: `<deque>`, `<list>`, `<map>`, `<multimap>`, `<queue>`, `<set>`, `<stack>`, and `<vector>`
   2) *Algorithms* are template functions for performing common operations on sequences of objects including: `<algorithm>`, `<functional>`, and `<numeric>`.
   3) *Iterators* are the glue that pastes together algorithms and containers and include: `<iterator>`, `<memory>`, and `<utility>`

B) Input Output includes components for:
   1) forward declarations of iostreams `<iosfwd>`
   2) predefined iostreams objects `<iostream>`
   3) base iostreams classes `<ios>`
   4) stream buffering `<streambuf>`
   5) stream formatting and manipulators: `<iosmanip>`, `<istream>`, and `<ostream>`
   6) string streams `<sstream>`
   7) file streams `<fstream>`

C) Other Standard C++ headers include:
   1) Language Support
      a) components for common type definitions used throughout the library `<cstddef>`
      b) characteristics of the predefined types `<limits>`, `<cfloat>`, and `<climits>`
      c) functions supporting start and termination of a C++ program `<cstdlib>`
      d) support for dynamic memory management `<new>`
      e) support for dynamic type identification `<typeinfo>`
      f) support for exception processing `<exception>`
      g) other run-time support, `<cstdarg>`, `<ctime>`, `<csetlmp>`, and `<csignal>`
   2) Diagnostics include components for:
      a) reporting several kinds of exceptional conditions `<stdexcept>`
      b) documenting program assertions `<cassert>`
      c) a global variable for error number codes `<cerrno>`
   3) Strings include components for:
      a) string classes `<string>`

PH019-Murray01 10/24/00 3:05 PM Page 30

b) null-terminated sequence utilities: <cctype>, <cwctype>, and <cwchar>

4) Cultural Language components include:

a) internationalization support for character classification, string collation, numeric, monetary, and date/time formatting and parsing, and message retrieval using <locale> and <clocale>

Sample Code

The following four programs—find.cpp, mdshfl.cpp, removif.cpp, and setunon.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.

The find.cpp Application

The first example application, find.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);
```

find() expects two input iterators (see Chapter 2) and the address of the comparison value, then returns an input iterator. The program looks like:

```
// find.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';
```

PH019-Murray01 10/24/00 3:05 PM Page 31


```
  // 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 is the proper include statement to use any STL template function:

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

and the 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
```

The `rndshfl.cpp` Application

Randomization of data is an extremely important component to many applications, whether it's the random shuffle of a deck of electronic poker cards, to 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.

PH019-Murray01 10/24/00 3:05 PM Page 32

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 in the next chapter). The program looks like:

```
// rndshuf.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;
```

PH019-Murray01 10/24/00 3:05 PM Page 33


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

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

Notice the application incorporates to 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 `cVowles` 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 `removif.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 to two forward iterators (discussed in chapter 2) and a predicate telling `remove_if()` what comparison test to perform. The program looks like:

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

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

using namespace std;
```

PH019-Murray01 10/24/00 3:05 PM Page 34


```
#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;

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

}
```

PH019-Murray01 10/24/00 3:05 PM Page 35

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, respectively, were removed.

The `setunon.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. The program looks like:

```
// setunon.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;
```

PH019-Murray01 10/24/00 3:05 PM Page 36


```
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: { ";
```

PH019-Murray01 10/24/00 3:05 PM Page 37


```
  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 `0`s in `iUnionedVector`, which indicate null elements.

Conclusion

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

PH019-Murray01 10/24/00 3:05 PM Page 38