

Module12

Exceptions, Templates, and Other Advanced Topics

Table of Contents

CRITICAL SKILL 12.1: Exception Handling	2
CRITICAL SKILL 12.2: Generic Functions	14
CRITICAL SKILL 12.3: Generic Classes	19
CRITICAL SKILL 12.4: Dynamic Allocation	26
CRITICAL SKILL 12.5: Namespaces	35
CRITICAL SKILL 12.6: static Class Members	42
CRITICAL SKILL 12.7: Runtime Type Identification (RTTI)	46
CRITICAL SKILL 12.8: The Casting Operators	49

You have come a long way since the start of this book. In this, the final module, you will examine several important, advanced C++ topics, including exception handling, templates, dynamic allocation, and namespaces. Runtime type ID and the casting operators are also covered. Keep in mind that C++ is a large, sophisticated, professional programming language, and it is not possible to cover every advanced feature, specialized technique, or programming nuance in this beginner's guide. When you finish this module, however, you will have mastered the core elements of the language and will be able to begin writing real-world programs.

CRITICAL SKILL 12.1: Exception Handling

An exception is an error that occurs at runtime. Using C++'s exception handling subsystem, you can, in a structured and controlled manner, handle runtime errors. When exception handling is employed, your program automatically invokes an error-handling routine when an exception occurs. The principal advantage of exception handling is that it automates much of the error-handling code that previously had to be entered "by hand" into any large program.

Exception Handling Fundamentals

C++ exception handling is built upon three keywords: try, catch, and throw. In the most general terms, program statements that you want to monitor for exceptions are contained in a try block. If an exception (that is, an error) occurs within the try block, it is thrown (using throw). The exception is caught, using catch, and processed. The following discussion elaborates upon this general description.

Code that you want to monitor for exceptions must have been executed from within a try block. (A function called from within a try block is also monitored.) Exceptions that can be thrown by the monitored code are caught by a catch statement that immediately follows the try statement in which the exception was thrown. The general forms of try and catch are shown here:

```
try {  
    // try block  
}  
catch (type1 arg) {  
    // catch block  
}  
catch (type2 arg) {  
    // catch block  
}  
catch (type3 arg) {  
    // catch block  
}  
// ...  
catch (typeN arg) {  
    // catch block  
}
```

The try block must contain the portion of your program that you want to monitor for errors. This section can be as short as a few statements within one function, or as all-encompassing as a try block that encloses the main() function code (which would, in effect, cause the entire program to be monitored).

When an exception is thrown, it is caught by its corresponding catch statement, which then processes the exception. There can be more than one catch statement associated with a try. The type of the exception determines which catch statement is used. That is, if the data type specified by a catch statement matches that of the exception, then that catch statement is executed (and all others are bypassed). When an exception is caught, arg will receive its value. Any type of data can be caught, including classes that you create.

The general form of the throw statement is shown here:

throw exception;

throw generates the exception specified by exception. If this exception is to be caught,

Exceptions, Templates, and Other Advanced Topics

then throw must be executed either from within a try block itself, or from any function called from within the try block (directly or indirectly).

If an exception is thrown for which there is no applicable catch statement, an abnormal program termination will occur. That is, your program will stop abruptly in an uncontrolled manner. Thus, you will want to catch all exceptions that will be thrown.

Here is a simple example that shows how C++ exception handling operates:

```
// A simple exception handling example.

#include <iostream>
using namespace std;

int main()
{
    cout << "start\n";

    try { // start a try block ←—— Begin a try block.
        cout << "Inside try block\n";
        throw 99; // throw an error ←—— Throw an exception.
        cout << "This will not execute";
    }
    catch (int i) { // catch an error ←—— Catch the exception.
        cout << "Caught an exception -- value is: ";
        cout << i << "\n";
    }

    cout << "end";

    return 0;
}
```

This program displays the following output:

```
start Inside
try block
Caught an exception -- value is: 99
end
```

Look carefully at this program. As you can see, there is a try block containing three statements and a catch(int i) statement that processes an integer exception. Within the try block, only two of the three statements will execute: the first cout statement and the throw. Once an exception has been thrown, control passes to the catch expression, and the try block is terminated. That is, catch is not called.

Rather, program execution is transferred to it. (The program's stack is automatically reset, as necessary, to accomplish this.) Thus, the `cout` statement following the `throw` will never execute.

Usually, the code within a `catch` statement attempts to remedy an error by taking appropriate action. If the error can be fixed, then execution will continue with the statements following the `catch`. Otherwise, program execution should be terminated in a controlled manner.

As mentioned earlier, the type of the exception must match the type specified in a `catch` statement. For example, in the preceding program, if you change the type in the `catch` statement to `double`, then the exception will not be caught and abnormal termination will occur. This change is shown here:

```
// This example will not work.

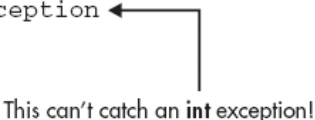
#include <iostream>
using namespace std;

int main()
{
    cout << "start\n";

    try { // start a try block
        cout << "Inside try block\n";
        throw 99; // throw an error
        cout << "This will not execute";
    }
    catch (double i) { // won't work for an int exception ←
        cout << "Caught an exception -- value is: ";
        cout << i << "\n";
    }

    cout << "end";

    return 0;
}
```



This can't catch an `int` exception!

This program produces the following output because the integer exception will not be caught by the `catch(double i)` statement. Of course, the final message indicating abnormal termination will vary from compiler to compiler.

```
start Inside
try block
Abnormal program termination
```

An exception thrown by a function called from within a `try` block can be handled by that `try` block. For example, this is a valid program:

```

/* Throwing an exception from a function called
   from within a try block. */

#include <iostream>
using namespace std;

void Xtest(int test)
{
    cout << "Inside Xtest, test is: " << test << "\n";
    if(test) throw test;
}

int main()
{
    cout << "start\n";

    try { // start a try block
        cout << "Inside try block\n";
        Xtest(0);
        Xtest(1);
        Xtest(2);
    }
    catch (int i) { // catch an error
        cout << "Caught an exception -- value is: ";
        cout << i << "\n";
    }

    cout << "end";

    return 0;
}

```

← This exception is caught by the **catch** statement in **main()**.

← Because **Xtest()** is called from within a **try** block, its code is also monitored for errors.

This program produces the following output:

```

start
Inside try block
Inside Xtest, test is: 0
Inside Xtest, test is: 1
Caught an exception -- value is: 1
end

```

As the output confirms, the exception thrown in `Xtest()` was caught by the exception handler in `main()`. A try block can be localized to a function. When this is the case, each time the function is entered, the exception handling relative to that function is reset. Examine this sample program:

```

// A try block can be localized to a function.

#include <iostream>
using namespace std;

// A try/catch is reset each time a function is entered.
void Xhandler(int test)
{
    try{ ←———— This try block is local to Xhandler().
        if(test) throw test;
    }
    catch(int i) {
        cout << "Caught One!  Ex. #: " << i << '\n';
    }
}

int main()
{
    cout << "start\n";

    Xhandler(1);
    Xhandler(2);

// A try block can be localized to a function.

#include <iostream>
using namespace std;

// A try/catch is reset each time a function is entered.
void Xhandler(int test)
{
    try{ ←———— This try block is local to Xhandler().
        if(test) throw test;
    }
    catch(int i) {
        cout << "Caught One!  Ex. #: " << i << '\n';
    }
}

int main()
{
    cout << "start\n";

    Xhandler(1);
    Xhandler(2);
    Xhandler(0);
    Xhandler(3);

    cout << "end";

    return 0;
}

```

This program displays the following output:

```
start
```

```

Caught One! Ex. #: 1
Caught One! Ex. #: 2
Caught One! Ex. #: 3
end

```

In this example, three exceptions are thrown. After each exception, the function returns. When the function is called again, the exception handling is reset. In general, a try block is reset each time it is entered. Thus, a try block that is part of a loop will be reset each time the loop repeats.



1. In the language of C++, what is an exception?
2. Exception handling is based on what three keywords?
3. An exception is caught based on its type. True or false?

Using Multiple catch Statements

As stated earlier, you can associate more than one catch statement with a try. In fact, it is common to do so. However, each catch must catch a different type of exception. For example, the program shown next catches both integers and character pointers.

```

// Use multiple catch statements.

#include <iostream>
using namespace std;

// Different types of exceptions can be caught.
void Xhandler(int test)
{
    try{
        if(test) throw test; // throw int
        else throw "Value is zero"; // throw char *
    }
    catch(int i) { ← This catches int exceptions.
        cout << "Caught One! Ex. #: " << i << '\n';
    }
    catch(char *str) { ← This catches char * exceptions.
        cout << "Caught a string: ";
        cout << str << '\n';
    }
}

```

```
int main()
{
    cout << "start\n";

    Xhandler(1);
    Xhandler(2);
    Xhandler(0);
    Xhandler(3);

    cout << "end";

    return 0;
}
```

In general, catch expressions are checked in the order in which they occur in a program. Only a matching statement is executed. All other catch blocks are ignored.

Catching Base Class Exceptions

There is one important point about multiple catch statements that relates to derived classes. A catch clause for a base class will also match any class derived from that base. Thus, if you want to catch exceptions of both a base class type and a derived class type, put the derived class first in the catch sequence. If you don't, the base class catch will also catch all derived classes. For example, consider the following program:


```
// Catching derived classes. This program is wrong!

#include <iostream>
using namespace std;

class B {
};

class D: public B {
};

int main()
{
    D derived;

    try {
        throw derived;
    }
    catch(B b) { ←———— This catch list is in the
        cout << "Caught a base class.\n";           wrong order! You must
    }                                                 catch derived classes
    catch(D d) { ←———— before base classes.
        cout << "This won't execute.\n";
    }

    return 0;
}
```

Here, because `derived` is an object that has `B` as a base class, it will be caught by the first catch clause, and the second clause will never execute. Some compilers will flag this condition with a warning message. Others may issue an error message and stop compilation. Either way, to fix this condition, reverse the order of the catch clauses.

Catching All Exceptions

In some circumstances, you will want an exception handler to catch all exceptions instead of just a certain type. To do this, use this form of catch:

```
catch(...) { // process all exceptions }
```

Here, the ellipsis matches any type of data. The following program illustrates `catch(...)`:

```

// This example catches all exceptions.

#include <iostream>
using namespace std;

void Xhandler(int test)
{
    try{
        if(test==0) throw test; // throw int
        if(test==1) throw 'a'; // throw char
        if(test==2) throw 123.23; // throw double
    }
    catch(...) { // catch all exceptions ← Catch all exceptions.
        cout << "Caught One!\n";
    }
}

int main()
{
    cout << "start\n";

    Xhandler(0);
    Xhandler(1);
    Xhandler(2);

    cout << "end";

    return 0;
}

```

This program displays the following output:

```

start
Caught One!
Caught One!
Caught One!
end

```

Xhandler() throws three types of exceptions: int, char, and double. All are caught using the catch(...) statement.

One very good use for catch(...) is as the last catch of a cluster of catches. In this capacity, it provides a useful default or “catch all” statement. Using catch(...) as a default is a good way to catch all exceptions that you don’t want to handle explicitly. Also, by catching all exceptions, you prevent an unhandled exception from causing an abnormal program termination.

Specifying Exceptions Thrown by a Function

You can specify the type of exceptions that a function can throw outside of itself. In fact, you can also prevent a function from throwing any exceptions whatsoever. To accomplish these restrictions, you must add a throw clause to a function definition. The general form of this clause is

ret-type func-name(arg-list) throw(type-list) { // ... }

Here, only those data types contained in the comma-separated type-list can be thrown by the function. Throwing any other type of expression will cause abnormal program termination. If you don't want a function to be able to throw any exceptions, then use an empty list.

NOTE: At the time of this writing, Visual C++ does not actually prevent a function from throwing an exception type that is not specified in the **throw** clause. This is nonstandard behavior. You can still specify a **throw** clause, but such a clause is informational only.

The following program shows how to specify the types of exceptions that can be thrown from a function:

```
// Restricting function throw types.

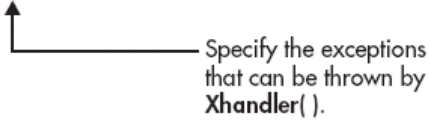
#include <iostream>
using namespace std;

// This function can only throw ints, chars, and doubles.

void Xhandler(int test) throw(int, char, double)
{
    if(test==0) throw test;    // throw int
    if(test==1) throw 'a';    // throw char
    if(test==2) throw 123.23; // throw double
}

int main()
{
    cout << "start\n";

    try{
        Xhandler(0); // also, try passing 1 and 2 to Xhandler()
```



Specify the exceptions
that can be thrown by
Xhandler().

```

    }
    catch(int i) {
        cout << "Caught int\n";
    }
    catch(char c) {
        cout << "Caught char\n";
    }
    catch(double d) {
        cout << "Caught double\n";
    }

    cout << "end";

    return 0;
}

```

In this program, the function `Xhandler()` can only throw integer, character, and double exceptions. If it attempts to throw any other type of exception, then an abnormal program termination will occur. To see an example of this, remove `int` from the list and retry the program. An error will result. (As mentioned, currently Visual C++ does not restrict the exceptions that a function can throw.)

It is important to understand that a function can only be restricted in what types of exceptions it throws back to the `try` block that has called it. That is, a `try` block within a function can throw any type of exception, as long as the exception is caught within that function. The restriction applies only when throwing an exception outside of the function.

Rethrowing an Exception

You can rethrow an exception from within an exception handler by calling `throw` by itself, with no exception. This causes the current exception to be passed on to an outer `try/catch` sequence. The most likely reason for calling `throw` this way is to allow multiple handlers access to the exception. For example, perhaps one exception handler manages one aspect of an exception, and a second handler copes with another aspect. An exception can only be rethrown from within a `catch` block (or from any function called from within that block). When you rethrow an exception, it will not be recaptured by the same `catch` statement. It will propagate to the next `catch` statement. The following program illustrates rethrowing an exception. It rethrows a `char *` exception.

```

// Example of "rethrowing" an exception.

#include <iostream>
using namespace std;

void Xhandler()
{
    try {
        throw "hello"; // throw a char *
    }
}

```

```

    catch(char *) { // catch a char *
        cout << "Caught char * inside Xhandler\n";
        throw ; // rethrow char * out of function ← Rethrow an exception.
    }
}

int main()
{
    cout << "start\n";

    try{
        Xhandler();
    }
    catch(char *) {
        cout << "Caught char * inside main\n";
    }

    cout << "end";

    return 0;
}

```

This program displays the following output:

```

start
Caught char * inside Xhandler
Caught char * inside main
End

```

Progress Check

1. Show how to catch all exceptions.
2. How do you specify the type of exceptions that can be thrown out of a function?
3. How do you rethrow an exception?

Templates

The template is one of C++'s most sophisticated and high-powered features. Although not part of the original specification for C++, it was added several years ago and is supported by all modern C++ compilers. Templates help you achieve one of the most elusive goals in programming: the creation of reusable code.

Using templates, it is possible to create generic functions and classes. In a generic function or class, the type of data upon which the function or class operates is specified as a parameter. Thus, you can use one function or class with several different types of data without having to explicitly recode specific versions for each data type. Both generic functions and generic classes are introduced here.

Ask the Expert

Q: It seems that there are two ways for a function to report an error: to throw an exception or to return an error code. In general, when should I use each approach?

A: You are correct, there are two general approaches to reporting errors: throwing exceptions and returning error codes. Today, language experts favor exceptions rather than error codes. For example, both the Java and C# languages rely heavily on exceptions, using them to report most types of common errors, such as an error opening a file or an arithmetic overflow. Because C++ is derived from C, it uses a blend of error codes and exceptions to report errors. Thus, many error conditions that relate to C++ library functions are reported using error return codes. However, in new code that you write, you should consider using exceptions to report errors. It is the way modern code is being written.

CRITICAL SKILL 12.2: Generic Functions

A generic function defines a general set of operations that will be applied to various types of data. The type of data that the function will operate upon is passed to it as a parameter. Through a generic function, a single general procedure can be applied to a wide range of data. As you probably know, many algorithms are logically the same no matter what type of data is being operated upon. For example, the Quicksort sorting algorithm is the same whether it is applied to an array of integers or an array of floats. It is just that the type of data being sorted is different. By creating a generic function, you can define the nature of the algorithm, independent of any data. Once you have done this, the compiler will automatically generate the correct code for the type of data that is actually used when you execute the function. In essence, when you create a generic function, you are creating a function that can automatically overload itself.

A generic function is created using the keyword `template`. The normal meaning of the word “template” accurately reflects its use in C++. It is used to create a template (or framework) that describes what a function will do, leaving it to the compiler to fill in the details as needed. The general form of a generic function definition is shown here:

```
template <class Ttype> ret-type func-name(parameter list) { // body of function }
```

Here, `Ttype` is a placeholder name for a data type. This name is then used within the function definition to declare the type of data upon which the function operates. The compiler will automatically replace `Ttype` with an actual data type when it creates a specific version of the function. Although the use of the keyword `class` to specify a generic type in a template declaration is traditional, you may also use the keyword `typename`.

The following example creates a generic function that swaps the values of the two variables with which it is called. Because the process of exchanging two values is independent of the type of the variables, it is a good candidate for being made into a generic function.

```
// Function template example.

#include <iostream>
using namespace std;

// This is a function template.
template <class X> void swapargs(X &a, X &b)
{
    X temp;
    temp = a;
    a = b;
    b = temp;
}

int main()
{
    int i=10, j=20;
    float x=10.1, y=23.3;
    char a='x', b='z';

    cout << "Original i, j: " << i << ' ' << j << '\n';
    cout << "Original x, y: " << x << ' ' << y << '\n';
    cout << "Original a, b: " << a << ' ' << b << '\n';

    swapargs(i, j); // swap integers
    swapargs(x, y); // swap floats
    swapargs(a, b); // swap chars

    cout << "Swapped i, j: " << i << ' ' << j << '\n';
    cout << "Swapped x, y: " << x << ' ' << y << '\n';
    cout << "Swapped a, b: " << a << ' ' << b << '\n';

    return 0;
}
```

A generic function that exchanges the values of its arguments. Here, **X** is the generic data type.

The compiler automatically creates versions of **swapargs()** that use the type of data specified by its arguments.

Let's look closely at this program. The line

```
template <class X> void swapargs(X &a, X &b)
```

tells the compiler two things: that a template is being created and that a generic definition is beginning. Here, **X** is a generic type that is used as a placeholder. After the template portion, the function **swapargs()** is declared, using **X** as the data type of the values that will be swapped. In **main()**, the **swapargs()** function is called using three different types of data: ints, floats, and chars. Because **swapargs()** is a generic function, the compiler automatically creates three versions of **swapargs()**: one that will exchange integer values, one that will exchange floating-point values, and one that will swap

characters. Thus, the same generic `swap()` function can be used to exchange arguments of any type of data.

Here are some important terms related to templates. First, a generic function (that is, a function definition preceded by a template statement) is also called a template function. Both terms are used interchangeably in this book. When the compiler creates a specific version of this function, it is said to have created a specialization. This is also called a generated function. The act of generating a function is referred to as instantiating it. Put differently, a generated function is a specific instance of a template function.

A Function with Two Generic Types

You can define more than one generic data type in the template statement by using a comma-separated list. For example, this program creates a template function that has two generic types:

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

template <class Type1, class Type2> ← Two generic types
void myfunc(Type1 x, Type2 y)
{
    cout << x << ' ' << y << '\n';
}

int main()
{
    myfunc(10, "hi");

    myfunc(0.23, 10L);

    return 0;
}
```

In this example, the placeholder types `Type1` and `Type2` are replaced by the compiler with the data types `int` and `char *`, and `double` and `long`, respectively, when the compiler generates the specific instances of `myfunc()` within `main()`.

Explicitly Overloading a Generic Function

Even though a generic function overloads itself as needed, you can explicitly overload one, too. This is formally called explicit specialization. If you overload a generic function, then that overloaded function overrides (or “hides”) the generic function relative to that specific version. For example, consider the following, revised version of the argument-swapping example shown earlier:


```
// Specializing a template function.  
  
#include <iostream>  
using namespace std;  
  
template <class X> void swapargs(X &a, X &b)  
{  
    X temp;
```

```

    temp = a;
    a = b;
    b = temp;
    cout << "Inside template swapargs.\n";
}

// This overrides the generic version of swapargs() for ints.
void swapargs(int &a, int &b) ← Explicit overload of swapargs()
{
    int temp;

    temp = a;
    a = b;
    b = temp;
    cout << "Inside swapargs int specialization.\n";
}

int main()
{
    int i=10, j=20;
    float x=10.1, y=23.3;
    char a='x', b='z';

    cout << "Original i, j: " << i << ' ' << j << '\n';
    cout << "Original x, y: " << x << ' ' << y << '\n';
    cout << "Original a, b: " << a << ' ' << b << '\n';

    swapargs(i, j); // calls explicitly overloaded swapargs()
    swapargs(x, y); // calls generic swapargs()
    swapargs(a, b); // calls generic swapargs() ← This calls the explicit
                                                    overload of swapargs().

    cout << "Swapped i, j: " << i << ' ' << j << '\n';
    cout << "Swapped x, y: " << x << ' ' << y << '\n';
    cout << "Swapped a, b: " << a << ' ' << b << '\n';

    return 0;
}

```

This program displays the following output:

```

Original i, j: 10 20
Original x, y: 10.1 23.3
Original a, b: x z
Inside swapargs int specialization.
Inside template swapargs.
Inside template swapargs.
Swapped i, j: 20 10
Swapped x, y: 23.3 10.1
Swapped a, b: z x

```

As the comments inside the program indicate, when `swapargs(i, j)` is called, it invokes the explicitly overloaded version of `swapargs()` defined in the program. Thus, the compiler does not generate this version of the generic `swapargs()` function, because the generic function is overridden by the explicit overloading.

Relatively recently, an alternative syntax was introduced to denote the explicit specialization of a function. This newer approach uses the `template` keyword. For example, using the newer specialization syntax, the overloaded `swapargs()` function from the preceding program looks like this:

```
// Use the newer-style specialization syntax.
template<> void swapargs<int>(int &a, int &b)
{
    int temp;

    temp = a;
    a = b;
    b = temp;
    cout << "Inside swapargs int specialization.\n";
}
```

As you can see, the new-style syntax uses the `template<>` construct to indicate specialization. The type of data for which the specialization is being created is placed inside the angle brackets following the function name. This same syntax is used to specialize any type of generic function. While there is no advantage to using one specialization syntax over the other at this time, the new-style syntax is probably a better approach for the long term.

Explicit specialization of a template allows you to tailor a version of a generic function to accommodate a unique situation—perhaps to take advantage of some performance boost that applies to only one type of data, for example. However, as a general rule, if you need to have different versions of a function for different data types, you should use overloaded functions rather than templates.

CRITICAL SKILL 12.3: Generic Classes

In addition to using generic functions, you can also define a generic class. When you do this, you create a class that defines all the algorithms used by that class; however, the actual type of data being manipulated will be specified as a parameter when objects of that class are created.

Generic classes are useful when a class uses logic that can be generalized. For example, the same algorithm that maintains a queue of integers will also work for a queue of characters, and the same mechanism that maintains a linked list of mailing addresses will also maintain a linked list of auto-part information. When you create a generic class, it can perform the operation you define, such as maintaining a queue or a linked list, for any type of data. The compiler will automatically generate the correct type of object, based upon the type you specify when the object is created.

The general form of a generic class declaration is shown here:

```
template <class Ttype> class class-name {
```

```
// body of class }
```

Here, Ttype is the placeholder type name, which will be specified when a class is instantiated. If necessary, you can define more than one generic data type using a comma-separated list.

Once you have created a generic class, you create a specific instance of that class using the following general form:

```
class-name <type> ob;
```

Here, type is the type name of the data that the class will be operating upon. Member functions of a generic class are, themselves, automatically generic. You need not use template to explicitly specify them as such.

Here is a simple example of a generic class:

```
// A simple generic class.
```

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

```
template <class T> class MyClass {
    T x, y;
public:
    MyClass(T a, T b) {
        x = a;
        y = b;
    }
    T div() { return x/y; }
};
```

← Declare a generic class.
Here, T is the generic type.

```
int main()
{
    // Create a version of MyClass for doubles.
    MyClass<double> d_ob(10.0, 3.0 );
    cout << "double division: " << d_ob.div() << "\n";

    // Create a version of MyClass for ints.
    MyClass<int> i_ob(10, 3);
    cout << "integer division: " << i_ob.div() << "\n";

    return 0;
}
```

← Create a specific instance
of a generic class.

The output is shown here:

```
double division: 3.33333
```

```
integer division: 3
```

As the output shows, the double object performed a floating-point division, and the int object performed an integer division.

When a specific instance of MyClass is declared, the compiler automatically generates versions of the div() function, and x and y variables necessary for handling the actual data. In this example, two different types of objects are declared. The first, d_ob, operates on double data. This means that x and y are double values, and the outcome of the division—and the return type of div()—is double. The second, i_ob, operates on type int. Thus, x, y, and the return type of div() are int. Pay special attention to these declarations:

Exceptions, Templates, and Other Advanced Topics

```
MyClass<double> d_ob(10.0, 3.0); MyClass<int> i_ob(10, 3);
```

Notice how the desired data type is passed inside the angle brackets. By changing the type of data specified when MyClass objects are created, you can change the type of data operated upon by MyClass.

A template class can have more than one generic data type. Simply declare all the data types required by the class in a comma-separated list within the template specification. For instance, the following example creates a class that uses two generic data types:

```
/* This example uses two generic data types in a
   class definition. */
#include <iostream>
using namespace std;

template <class T1, class T2> class MyClass
{
    T1 i;
    T2 j;
public:
    MyClass(T1 a, T2 b) { i = a; j = b; }
    void show() { cout << i << ' ' << j << '\n'; }
};

int main()
{
    MyClass<int, double> ob1(10, 0.23);
    MyClass<char, char *> ob2('X', "This is a test");

    ob1.show(); // show int, double
    ob2.show(); // show char, char *

    return 0;
}
```

This program produces the following output:

```
10 0.23
```

```
X This is a test
```

The program declares two types of objects. ob1 uses int and double data. ob2 uses a character and a character pointer. For both cases, the compiler automatically generates the appropriate data and functions to accommodate the way the objects are created.

Explicit Class Specializations

As with template functions, you can create a specialization of a generic class. To do so, use the `template<>` construct as you did when creating explicit function specializations. For example:

```
// Demonstrate class specialization.

#include <iostream>
using namespace std;

template <class T> class MyClass {
    T x;
public:
    MyClass(T a) {
        cout << "Inside generic MyClass\n";
        x = a;
    }
    T getx() { return x; }
};

// Explicit specialization for int.
template <> class MyClass<int> {
    int x;
public:
    MyClass(int a) {
        cout << "Inside MyClass<int> specialization\n";
        x = a * a;
    }
    int getx() { return x; }
};

int main()
{
    MyClass<double> d(10.1);
    cout << "double: " << d.getx() << "\n\n";

    MyClass<int> i(5);
    cout << "int: " << i.getx() << "\n";

    return 0;
}
```

This is an explicit specialization of **MyClass**.

This uses the explicit specialization of **MyClass**.

This program displays the following output:

```
Inside generic MyClass
double: 10.1
Inside MyClass<int> specialization
int: 25
```

In the program, pay close attention to this line:

```
template <> class MyClass<int> {
```

It tells the compiler that an explicit integer specialization of MyClass is being created. This same general syntax is used for any type of class specialization.

Explicit class specialization expands the utility of generic classes because it lets you easily handle one or two special cases while allowing all others to be automatically processed by the compiler. Of course, if you find that you are creating too many specializations, then you are probably better off not using a template class in the first place.



1. What keyword is used to declare a generic function or class?
2. Can a generic function be explicitly overloaded?
3. In a generic class, are all of its member functions also automatically generic?

Project 12-1 Creating a Generic Queue Class

In Project 8-2, you created a Queue class that maintained a queue of characters. In this project, you will convert Queue into a generic class that can operate on any type of data. Queue is a good choice for conversion to a generic class, because its logic is separate from the data upon which it functions. The same mechanism that stores integers, for example, can also store floating-point values, or even objects of classes that you create. Once you have defined a generic Queue class, you can use it whenever you need a queue.

Step by Step

1. Begin by copying the Queue class from Project 8-2 into a file called GenericQ.cpp.
2. Change the Queue declaration into a template, as shown here:

```
template <class QType> class Queue {
```

Here, the generic data type is called QType.
3. Change the data type of the q array to QType, as shown next:

QType q[maxQsize]; // this array holds the queue
 Because q is now generic, it can be used to hold whatever type of data an object of Queue declares.

4. Change the data type of the parameter to the put() function to QType, as shown here:

```
// Put a data into the queue.
void put(QType data) {
    if(putloc == size) {
        cout << " -- Queue is full.\n";
        return;
    }

    putloc++;
    q[putloc] = data;
}
```

5. Change the return type of get() to QType, as shown next:

```
// Get data from the queue.
QType get() {
    if(getloc == putloc) {
        cout << " -- Queue is empty.\n";
        return 0;
    }

    getloc++;
    return q[getloc];
}
```

6. The entire generic Queue class is shown here along with a main() function to demonstrate its use:

```
/*
    Project 12-1

    A template queue class.
*/
#include <iostream>
using namespace std;

const int maxQsize = 100;

// This creates a generic queue class.
template <class QType> class Queue {
    QType q[maxQsize]; // this array holds the queue
    int size; // maximum number of elements that the queue can store
    int putloc, getloc; // the put and get indices
public:

    // Construct a queue of a specific length.
    Queue(int len) {
        // Queue must be less than max and positive.
        if(len > maxQsize) len = maxQsize;
        else if(len <= 0) len = 1;
    }
};
```



```

    size = len;
    putloc = getloc = 0;
}

// Put data into the queue.
void put(QType data) {
    if(putloc == size) {
        cout << " -- Queue is full.\n";
        return;
    }

    putloc++;
    q[putloc] = data;
}

// Get data from the queue.
QType get() {
    if(getloc == putloc) {
        cout << " -- Queue is empty.\n";
        return 0;
    }

    getloc++;
    return q[getloc];
}

};

// Demonstrate the generic Queue.
int main()
{
    Queue<int> iQa(10), iQb(10); // create two integer queues

    iQa.put(1);
    iQa.put(2);
    iQa.put(3);

    iQb.put(10);
    iQb.put(20);
    iQb.put(30);

    cout << "Contents of integer queue iQa: ";
    for(int i=0; i < 3; i++)
        cout << iQa.get() << " ";

```

```

    cout << endl;

    cout << "Contents of integer queue iQb: ";
    for(int i=0; i < 3; i++)
        cout << iQb.get() << " ";
    cout << endl;

    Queue<double> dQa(10), dQb(10); // create two double queues

    Queue<double> dQa(10), dQb(10); // create two double queues

    dQa.put(1.01);
    dQa.put(2.02);
    dQa.put(3.03);

    dQb.put(10.01);
    dQb.put(20.02);
    dQb.put(30.03);

    cout << "Contents of double queue dQa: ";
    for(int i=0; i < 3; i++)
        cout << dQa.get() << " ";
    cout << endl;

    cout << "Contents of double queue dQb: ";
    for(int i=0; i < 3; i++)
        cout << dQb.get() << " ";
    cout << endl;

    return 0;
}

```

The output is shown here:

```

Contents of integer queue iQa: 1 2 3
Contents of integer queue iQb: 10 20 30
Contents of double queue dQa: 1.01 2.02 3.03
Contents of double queue dQb: 10.01 20.02 30.03

```

7. As the Queue class illustrates, generic functions and classes are powerful tools that you can use to maximize your programming efforts, because they allow you to define the general form of an object that can then be used with any type of data. You are saved from the tedium of creating separate implementations for each data type for which you want the algorithm to work. The compiler automatically creates the specific versions of the class for you.

CRITICAL SKILL 12.4: Dynamic Allocation

There are two primary ways in which a C++ program can store information in the main memory of the computer. The first is through the use of variables. The storage provided by variables is fixed at compile

time and cannot be altered during the execution of a program. The second way information can be stored is through the use of C++'s dynamic allocation system. In this method, storage for data is allocated as needed from the free memory area that lies between your program (and its permanent storage area) and the stack. This region is called the heap. (Figure 12-1 shows conceptually how a C++ program appears in memory.)

Dynamically allocated storage is determined at runtime. Thus, dynamic allocation makes it possible for your program to create variables that it needs during its execution. It can create as many or as few variables as required, depending upon the situation. Dynamic allocation is often used to support such data structures as linked lists, binary trees, and sparse arrays. Of course, you are free to use dynamic allocation wherever you determine it to be of value. Dynamic allocation for one purpose or another is an important part of nearly all real-world programs.

Memory to satisfy a dynamic allocation request is taken from the heap. As you might guess, it is possible, under fairly extreme cases, for free memory to become exhausted. Therefore, while dynamic allocation offers greater flexibility, it too is finite.

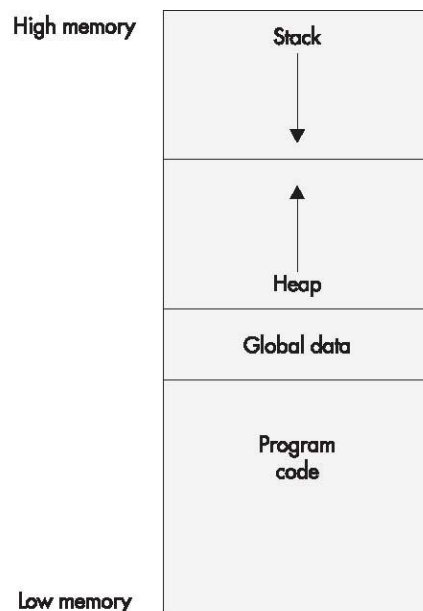


Figure 12-1 A conceptual view of memory usage in a C++ program

C++ provides two dynamic allocation operators: `new` and `delete`. The `new` operator allocates memory and returns a pointer to the start of it. The `delete` operator frees memory previously allocated using `new`. The general forms of `new` and `delete` are shown here:

```
p_var = new type; delete p_var;
```

Here, `p_var` is a pointer variable that receives a pointer to memory that is large enough to hold an item of type `type`.

Since the heap is finite, it can become exhausted. If there is insufficient available memory to fill an allocation request, then `new` will fail and a `bad_alloc` exception will be generated. This exception is

defined in the header `<new>`. Your program should handle this exception and take appropriate action if a failure occurs. If this exception is not handled by your program, then your program will be terminated.

The actions of `new` on failure as just described are specified by Standard C++. The trouble is that some older compilers will implement `new` in a different way. When C++ was first invented, `new` returned a null pointer on failure. Later, this was changed so that `new` throws an exception on failure, as just described. If you are using an older compiler, check your compiler's documentation to see precisely how it implements `new`.

Since Standard C++ specifies that `new` generates an exception on failure, this is the way the code in this book is written. If your compiler handles an allocation failure differently, then you will need to make the appropriate changes.

Here is a program that allocates memory to hold an integer:

```
// Demonstrate new and delete.

#include <iostream>
#include <new>
using namespace std;

int main()
{
    int *p;

    try {
        p = new int; // allocate space for an int
    } catch (bad_alloc xa) {
        cout << "Allocation Failure\n";
        return 1;
    }

    *p = 100;
    cout << "At " << p << " ";
    cout << "is the value " << *p << "\n";

    delete p;

    return 0;
}
```

Allocate an int.

Watch for an allocation failure.

Release the allocated memory.

This program assigns to `p` an address in the heap that is large enough to hold an integer. It then assigns that memory the value 100 and displays the contents of the memory on the screen. Finally, it frees the dynamically allocated memory.

The `delete` operator must be used only with a valid pointer previously allocated by using `new`. Using any other type of pointer with `delete` is undefined and will almost certainly cause serious problems, such as a system crash.

Initializing Allocated Memory

You can initialize allocated memory to some known value by putting an initializer after the type name in the new statement. Here is the general form of new when an initialization is included:

```
p_var = new var_type (initializer);
```

Of course, the type of the initializer must be compatible with the type of data for which memory is being allocated.

Ask the Expert

Q: I have seen some C++ code that uses the functions `malloc()` and `free()` to handle dynamic allocation. What are these functions?

A: The C language does not support the new and delete operators. Instead, C uses the functions `malloc()` and `free()` for dynamic allocation. `malloc()` allocates memory and `free()` releases it. C++ also supports these functions, and you will sometimes see `malloc()` and `free()` used in C++ code. This is especially true if that code has been updated from older C code. However, you should use new and delete in your code. Not only do new and delete offer a more convenient method of handling dynamic allocation, but they also prevent several types of errors that are common when working with `malloc()` and `free()`. One other point: Although there is no formal rule that states this, it is best not to mix new and delete with `malloc()` and `free()` in the same program. There is no guarantee that they are mutually compatible.

This program gives the allocated integer an initial value of 87:

```

// Initialize memory.

#include <iostream>
#include <new>
using namespace std;

int main()
{
    int *p;

    try {
        p = new int (87); // initialize to 87
    } catch (bad_alloc xa) {
        cout << "Allocation Failure\n";
        return 1;
    }

    cout << "At " << p << " ";
    cout << "is the value " << *p << "\n";

    delete p;

    return 0;
}

```

Initialize allocated memory.

Allocating Arrays

You can allocate arrays using new by using this general form:

```
p_var = new array_type [size];
```

Here, size specifies the number of elements in the array. To free an array, use this form of delete:

```
delete [] p_var;
```

Here, the [] informs delete that an array is being released. For example, the next program allocates a ten-element integer array:

```

// Allocate an array.

#include <iostream>
#include <new>

```

```

using namespace std;

int main()
{
    int *p, i;

    try {
        p = new int [10]; // allocate 10 integer array
    } catch (bad_alloc xa) {
        cout << "Allocation Failure\n";
        return 1;
    }

    for(i=0; i<10; i++ )
        p[i] = i;

    for(i=0; i<10; i++)
        cout << p[i] << " ";

    delete [] p; // release the array

    return 0;
}

```

Allocate an array of int.

Release the array.

Notice the delete statement. As just mentioned, when an array allocated by new is released, delete must be made aware that an array is being freed by using the []. (As you will see in the next section, this is especially important when you are allocating arrays of objects.)

One restriction applies to allocating arrays: They may not be given initial values. That is, you may not specify an initializer when allocating arrays.

Allocating Objects

You can allocate objects dynamically by using new. When you do this, an object is created, and a pointer is returned to it. The dynamically created object acts just like any other object. When it is created, its constructor (if it has one) is called. When the object is freed, its destructor is executed.

Here is a program that creates a class called Rectangle that encapsulates the width and height of a rectangle. Inside main(), an object of type Rectangle is created dynamically. This object is destroyed when the program ends.

```

// Allocate an object.

#include <iostream>

```

```

#include <new>
using namespace std;

class Rectangle {
    int width;
    int height;
public:
    Rectangle(int w, int h) {
        width = w;
        height = h;
        cout << "Constructing " << width <<
            " by " << height << " rectangle.\n";
    }

    ~Rectangle() {
        cout << "Destructing " << width <<
            " by " << height << " rectangle.\n";
    }

    int area() {
        return width * height;
    }
};

int main()
{
    Rectangle *p;

    try {
        p = new Rectangle(10, 8); ← Allocate a Rectangle object. This
    } catch (bad_alloc xa) {      calls the Rectangle constructor.
        cout << "Allocation Failure\n";
        return 1;
    }

    cout << "Area is " << p->area();

    cout << "\n";

    delete p; ← Release the object. This calls the Rectangle destructor.

    return 0;
}

```

The output is shown here:

```

Constructing 10 by 8 rectangle.
Area is 80
Destructing 10 by 8 rectangle.

```


Notice that the arguments to the object's constructor are specified after the type name, just as in other sorts of initializations. Also, because `p` contains a pointer to an object, the arrow operator (rather than the dot operator) is used to call `area()`.

You can allocate arrays of objects, but there is one catch. Since no array allocated by `new` can have an initializer, you must make sure that if the class defines constructors, one will be parameterless. If you don't, the C++ compiler will not find a matching constructor when you attempt to allocate the array and will not compile your program.

In this version of the preceding program, a parameterless constructor is added so that an array of `Rectangle` objects can be allocated. Also added is the function `set()`, which sets the dimensions of each rectangle.

```
// Allocate an array of objects.

#include <iostream>
#include <new>
using namespace std;

class Rectangle {
    int width;
    int height;
public:
    Rectangle() { ← Add a parameterless constructor.
        width = height = 0;
        cout << "Constructing " << width <<
            " by " << height << " rectangle.\n";
    }

    Rectangle(int w, int h) {
        width = w;
        height = h;
        cout << "Constructing " << width <<
            " by " << height << " rectangle.\n";
    }

    ~Rectangle() {
        cout << "Destructing " << width <<
            " by " << height << " rectangle.\n";
    }
}
```

```

void set(int w, int h) { ← Add the set() function.
    width = w;
    height = h;
}

int area() {
    return width * height;
}
};

int main()
{
    Rectangle *p;

    try {
        p = new Rectangle [3];
    } catch (bad_alloc xa) {
        cout << "Allocation Failure\n";
        return 1;
    }

    cout << "\n";

    p[0].set(3, 4);
    p[1].set(10, 8);
    p[2].set(5, 6);

    for(int i=0; i < 3; i++)
        cout << "Area is " << p[i].area() << endl;

    cout << "\n";

    delete [] p; ← This calls the destructor for
                  each object in the array.

    return 0;
}

```

The output from this program is shown here:

```

Constructing 0 by 0 rectangle.
Constructing 0 by 0 rectangle.
Constructing 0 by 0 rectangle.

Area is 12
Area is 80
Area is 30
Destructing 5 by 6 rectangle.
Destructing 10 by 8 rectangle.
Destructing 3 by 4 rectangle.

```

Because the pointer `p` is released using `delete []`, the destructor for each object in the array is executed, as the output shows. Also, notice that because `p` is indexed as an array, the dot operator is used to access members of `Rectangle`.



1. What operator allocates memory? What operator releases memory?
2. What happens if an allocation request cannot be fulfilled?
3. Can memory be initialized when it is allocated?

CRITICAL SKILL 12.5: Namespaces

Namespaces were briefly described in Module 1. Here they are examined in detail. The purpose of a namespace is to localize the names of identifiers to avoid name collisions. In the C++ programming environment, there has been an explosion of variable, function, and class names. Prior to the invention of namespaces, all of these names competed for slots in the global namespace and many conflicts arose. For example, if your program defined a function called `toupper()`, it could (depending upon its parameter list) override the standard library function `toupper()`, because both names would be stored in the global namespace. Name collision problems were compounded when two or more third-party libraries were used by the same program. In this case, it was possible—even likely—that a name defined by one library would conflict with the same name defined by the other library. The situation can be particularly troublesome for class names. For example, if your program defines a class called `Stack` and a library used by your program defines a class by the same name, a conflict will arise.

The creation of the namespace keyword was a response to these problems. Because it localizes the visibility of names declared within it, a namespace allows the same name to be used in different contexts without conflicts arising. Perhaps the most noticeable beneficiary of namespace is the C++ standard library. Prior to namespace, the entire C++ library was defined within the global namespace (which was, of course, the only namespace). Since the addition of namespace, the C++ library is now defined within its own namespace, called `std`, which reduces the chance of name collisions. You can also create your own namespaces within your program to localize the visibility of any names that you think may cause conflicts. This is especially important if you are creating class or function libraries.

Namespace Fundamentals


The namespace keyword allows you to partition the global namespace by creating a declarative region. In essence, a namespace defines a scope. The general form of namespace is shown here:

```
namespace name { // declarations }
```

Anything defined within a namespace statement is within the scope of that namespace.

Here is an example of a namespace. It localizes the names used to implement a simple countdown counter class. In the namespace are defined the counter class, which implements the counter, and the variables upperbound and lowerbound, which contain the upper and

```
// Demonstrate a namespace.

namespace CounterNameSpace {  Create a namespace called
    int upperbound;           CounterNameSpace.
    int lowerbound;

    class counter {
        int count;
    public:
        counter(int n) {
            if(n <= upperbound) count = n;
            else count = upperbound;
        }

        void reset(int n) {
            if(n <= upperbound) count = n;
        }

        int run() {
            if(count > lowerbound) return count--;
            else return lowerbound;
        }
    };
}
```

Here, upperbound, lowerbound, and the class counter are part of the scope defined by the CounterNameSpace namespace.

Inside a namespace, identifiers declared within that namespace can be referred to directly, without any namespace qualification. For example, within CounterNameSpace, the run() function can refer directly to lowerbound in the statement

```
if(count > lowerbound) return count--;
```

However, since namespace defines a scope, you need to use the scope resolution operator to refer to objects declared within a namespace from outside that namespace. For example, to assign the value 10 to upperbound from code outside CounterNameSpace, you must use this statement:

```
CounterNameSpace::upperbound = 10;
```

Or, to declare an object of type counter from outside CounterNameSpace, you will use a statement like this:

```
CounterNameSpace::counter ob;
```

In general, to access a member of a namespace from outside its namespace, precede the member's name with the name of the namespace followed by the scope resolution operator.

Here is a program that demonstrates the use of the CounterNameSpace:

```
// Demonstrate a namespace.

#include <iostream>
using namespace std;

namespace CounterNameSpace {
    int upperbound;
    int lowerbound;

    class counter {
        int count;
    public:
        counter(int n) {
            if(n <= upperbound) count = n;
            else count = upperbound;
        }

        void reset(int n) {
            if(n <= upperbound) count = n;
        }

        int run() {
            if(count > lowerbound) return count--;
            else return lowerbound;
        }
    };
}


int main()
{
    CounterNameSpace::upperbound = 100;
    CounterNameSpace::lowerbound = 0;

    CounterNameSpace::counter ob1(10);

    int i;

    do {
        i = ob1.run();
        cout << i << " ";
    } while(i > CounterNameSpace::lowerbound);
    cout << endl;

    CounterNameSpace::counter ob2(20);
}
```



Explicitly refer to members of **CounterNameSpace**. Note the use of the scope resolution operator.

```

do {
    i = ob2.run();
    cout << i << " ";
} while(i > CounterNameSpace::lowerbound);
cout << endl;

ob2.reset(100);
CounterNameSpace::lowerbound = 90;
do {
    i = ob2.run();
    cout << i << " ";
} while(i > CounterNameSpace::lowerbound);

return 0;
}

```

Notice that the declaration of a counter object and the references to upperbound and lowerbound are qualified by CounterNameSpace. However, once an object of type counter has been declared, it is not necessary to further qualify it or any of its members. Thus, ob1.run() can be called directly; the namespace has already been resolved.

There can be more than one namespace declaration of the same name. In this case, the namespaces are additive. This allows a namespace to be split over several files or even separated within the same file. For example:

```

namespace NS { int i;
}

// ...

namespace NS { int j;
}

```

Here, NS is split into two pieces, but the contents of each piece are still within the same namespace, that is, NS. One last point: Namespaces can be nested. That is, one namespace can be declared within another.

using

If your program includes frequent references to the members of a namespace, having to specify the namespace and the scope resolution operator each time you need to refer to one quickly becomes

tedious. The using statement was invented to alleviate this problem. The using statement has these two general forms:

```
using namespace name;
```

```
using name::member;
```

In the first form, name specifies the name of the namespace you want to access. All of the members defined within the specified namespace are brought into view (that is, they become part of the current namespace) and may be used without qualification. In the second form, only a specific member of the namespace is made visible. For example, assuming CounterNameSpace as just shown, the following using statements and assignments are valid:

```
using CounterNameSpace::lowerbound; // only lowerbound is visible
lowerbound = 10; // OK because lowerbound is visible
using namespace CounterNameSpace; // all members are visible
upperbound = 100; // OK because all members are now visible
```

The following program illustrates using by reworking the counter example from the

```
// Demonstrate using.

#include <iostream>
using namespace std;

namespace CounterNameSpace {
    int upperbound;
    int lowerbound;

    class counter {
        int count;
    public:
        counter(int n) {
            if(n <= upperbound) count = n;
            else count = upperbound;
        }

        void reset(int n) {
            if(n <= upperbound) count = n;
        }

        int run() {
            if(count > lowerbound) return count--;
            else return lowerbound;
        }
    };
}
```

```

int main()
{
    // use only upperbound from CounterNameSpace
    using CounterNameSpace::upperbound; ← Use a specific member of
                                         CounterNameSpace.

    // now, no qualification needed to set upperbound
    upperbound = 100;
    // qualification still needed for lowerbound, etc.
    CounterNameSpace::lowerbound = 0;
    CounterNameSpace::counter ob1(10);
    int i;

    do {
        i = ob1.run();
        cout << i << " ";
    } while(i > CounterNameSpace::lowerbound);
    cout << endl;

    // Now, use entire CounterNameSpace
    using namespace CounterNameSpace; ← Use the entire
                                         CounterNameSpace.

    counter ob2(20);

    do {
        i = ob2.run();
        cout << i << " ";
    } while(i > lowerbound);
    cout << endl;

    ob2.reset(100);
    lowerbound = 90;
    do {
        i = ob2.run();
        cout << i << " ";
    } while(i > lowerbound);

    return 0;
}

```

The program illustrates one other important point: using one namespace does not override another. When you bring a namespace into view, it simply adds its names to whatever other namespaces are currently in effect. Thus, by the end of the program, both `std` and `CounterNameSpace` have been added to the global namespace.

Unnamed Namespaces

There is a special type of namespace, called an unnamed namespace, that allows you to create identifiers that are unique within a file. It has this general form:


```
namespace {  
  
// declarations }
```

Unnamed namespaces allow you to establish unique identifiers that are known only within the scope of a single file. That is, within the file that contains the unnamed namespace, the members of that namespace may be used directly, without qualification. But outside the file, the identifiers are unknown. As mentioned earlier in this book, one way to restrict the scope of a global name to the file in which it is declared, is to declare it as static. While the use of static global declarations is still allowed in C++, a better way to accomplish this is to use an unnamed namespace.

The std Namespace

Standard C++ defines its entire library in its own namespace called `std`. This is the reason that most of the programs in this book have included the following statement:

```
using namespace std;
```

This causes the `std` namespace to be brought into the current namespace, which gives you direct access to the names of the functions and classes defined within the library without having to qualify each one with `std::`.

Of course, you can explicitly qualify each name with `std::` if you like. For example, you could explicitly qualify `cout` like this:

```
std::cout << "Explicitly qualify cout with std.";
```

You may not want to bring the standard C++ library into the global namespace if your program will be making only limited use of it, or if doing so will cause name conflicts. However, if your program contains hundreds of references to library names, then including `std` in the current namespace is far easier than qualifying each name individually.



1. What is a namespace? What keyword creates one?
2. Are namespaces additive?
3. What does `using` do?

CRITICAL SKILL 12.6: static Class Members

You learned about the keyword `static` in Module 7 when it was used to modify local and global variable declarations. In addition to those uses, `static` can be applied to members of a class. Both variables and function members can be declared `static`. Each is described here.

static Member Variables

When you precede a member variable's declaration with `static`, you are telling the compiler that only one copy of that variable will exist and that all objects of the class will share that variable. Unlike regular data members, individual copies of a static member variable are not made for each object. No matter how many objects of a class are created, only one copy of a static data member exists. Thus, all objects of that class use that same variable. All static variables are initialized to zero if no other initialization is specified. When you declare a static data member within a class, you are not defining it. Instead, you must provide a global definition for it elsewhere, outside the class. This is done by redeclaring the static variable using the scope resolution operator to identify which class it belongs to. This causes storage to be allocated for the static variable. Here is an example that uses a static member:

```
// Use a static instance variable.

#include <iostream>
using namespace std;

class ShareVar {
    static int num;
public:
    void setnum(int i) { num = i; };
    void shownum() { cout << num << " "; }
};

int ShareVar::num; // define num

int main()
{
    ShareVar a, b;

    a.shownum(); // prints 0
    b.shownum(); // prints 0

    a.setnum(10); // set static num to 10
}
```

Declare a **static** data member. It will be shared by all instances of **ShareVar**.

Define the **static** data member.

```

    a.shownum(); // prints 10
    b.shownum(); // also prints 10

    return 0;
}

```

The output is shown here:

```
0 0 10 10
```

In the program, notice that the static integer `num` is both declared inside the `ShareVar` class and defined as a global variable. As stated earlier, this is necessary because the declaration of `num` inside `ShareVar` does not allocate storage for the variable. C++ initializes `num` to 0 since no other initialization is given. This is why the first calls to `shownum()` both display 0. Next, object `a` sets `num` to 10. Then both `a` and `b` use `shownum()` to display its value. Because there is only one copy of `num` shared by `a` and `b`, both calls to `shownum()` display 10.

When a static variable is public, it can be referred to directly through its class name, without reference to any specific object. It can also be referred to through an object. For example:

```

// Refer to static variable through its class name.

#include <iostream>
using namespace std;

class Test {
public:
    static int num;
    void shownum() { cout << num << endl; }
};

int Test::num; // define num

int main()
{
    Test a, b;

    // Set num through its class name.
    Test::num = 100; ← Refer to num through
                        its class name Test.

    a.shownum(); // prints 100
    b.shownum(); // prints 100

    // Set num through an object.

```

```

a.num = 200;
a.shownum(); // prints 200
b.shownum(); // prints 200

return 0;
}

```

Refer to **num** through an object.

Notice how the value of num is set using its class name in this line:

```
Test::num = 100;
```

It is also accessible through an object, as in this line:

```
a.num = 200;
```

Either approach is valid.

static Member Functions

It is also possible for a member function to be declared as static, but this usage is not common. A member function declared as static can access only other static members of its class. (Of course, a static member function may access non-static global data and functions.) A static member function does not have a this pointer. Virtual static member functions are not allowed. Also, it cannot be declared as const or volatile. A static member function can be invoked by an object of its class, or it can be called independent of any object, using the class name and the scope resolution operator. For example, consider this program. It defines a static variable called count that keeps count of the number of objects currently in existence.

```

// Demonstrate a static member function.

#include <iostream>
using namespace std;

class Test {
    static int count;
public:

    Test() {
        count++;
        cout << "Constructing object " <<
            count << endl;
    }
};

```

```

    }

    ~Test() {
        cout << "Destroying object " <<
            count << endl;
        count--;
    }

    static int numObjects() { return count; }
};

int Test::count;

int main() {
    Test a, b, c;

    cout << "There are now " <<
        Test::numObjects() <<
        " in existence.\n\n";

    Test *p = new Test();

    cout << "After allocating a Test object, " <<
        "there are now " <<
        Test::numObjects() <<
        " in existence.\n\n";

    delete p;
    cout << "After deleting an object, " <<
        "there are now " <<
        a.numObjects() <<
        " in existence.\n\n";

    return 0;
}

```

A static member function

The output from the program is shown here:

```

Constructing object 1
Constructing object 2
Constructing object 3
There are now 3 in existence.

Constructing object 4
After allocating a Test object, there are now 4 in existence.
Destroying object 4
After deleting an object, there are now 3 in existence.

Destroying object 3
Destroying object 2
Destroying object 1

```

In the program, notice how the static function `numObjects()` is called. In the first two calls, it is called through its class name using this syntax:

```
Test::numObjects()
```

In the third call, it is invoked using the normal, dot operator syntax on an object.

CRITICAL SKILL 12.7: Runtime Type Identification (RTTI)

Runtime type information may be new to you because it is not found in non-polymorphic languages, such as C or traditional BASIC. In non-polymorphic languages there is no need for runtime type information, because the type of each object is known at compile time (that is, when the program is written). However, in polymorphic languages such as C++, there can be situations in which the type of an object is unknown at compile time because the precise nature of that object is not determined until the program is executed. As you know, C++ implements polymorphism through the use of class hierarchies, virtual functions, and base class pointers. A base class pointer can be used to point to objects of the base class or to any object derived from that base. Thus, it is not always possible to know in advance what type of object will be pointed to by a base pointer at any given moment. This determination must be made at runtime, using runtime type identification.

To obtain an object's type, use `typeid`. You must include the header `<typeinfo>` in order to use `typeid`. Its most commonly used form is shown here:

```
typeid(object)
```

Here, `object` is the object whose type you will be obtaining. It may be of any type, including the built-in types and class types that you create. `typeid` returns a reference to an object of type `type_info` that describes the type of object.

The `type_info` class defines the following public members:

```
bool operator==(const type_info &ob); bool operator!=(const type_info &ob); bool before(const  
type_info &ob); const char *name( );
```

The overloaded `==` and `!=` provide for the comparison of types. The `before()` function returns true if the invoking object is before the object used as a parameter in collation order. (This function is mostly for internal use only. Its return value has nothing to do with inheritance or class hierarchies.) The `name()` function returns a pointer to the name of the type.

Here is a simple example that uses `typeid`:

```
// A simple example that uses typeid.

#include <iostream>
#include <typeinfo>
using namespace std;

class MyClass {
    // ...
};

int main()
{
    int i, j;
    float f;
    MyClass ob;

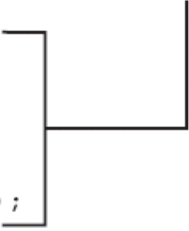
    cout << "The type of i is: " << typeid(i).name();
    cout << endl;
    cout << "The type of f is: " << typeid(f).name();
    cout << endl;
    cout << "The type of ob is: " << typeid(ob).name();
    cout << "\n\n";

    if(typeid(i) == typeid(j))
        cout << "The types of i and j are the same\n";

    if(typeid(i) != typeid(f))
        cout << "The types of i and f are not the same\n";

    return 0;
}
```

Use **typeid** to obtain the type of an object at runtime.



The output produced by this program is shown here:

```
The type of i is: int
The type of f is: float
The type of ob is: class MyClass
The types of i and j are the same
The types of i and f are not the same
```

Perhaps the most important use of `typeid` occurs when it is applied through a pointer of a polymorphic base class (that is, a class that includes at least one virtual function). In this case, it will automatically return the type of the actual object being pointed to, which may be a base class object or an object derived from that base. (Remember, a base class pointer can point to objects of the base class or of any class derived from that base.) Thus, using `typeid`, you can determine at runtime the type of the object that is being pointed to by a base class pointer. The following program demonstrates this principle:

```

// An example that uses typeid on a polymorphic class hierarchy

#include <iostream>
#include <typeinfo>
using namespace std;

class Base {
    virtual void f() {}; // make Base polymorphic
    // ...
};

class Derived1: public Base {
    // ...
};

class Derived2: public Base {
    // ...
};

int main()
{
    Base *p, baseob;
    Derived1 ob1;
    Derived2 ob2;

    p = &baseob;
    cout << "p is pointing to an object of type ";
    cout << typeid(*p).name() << endl;

    p = &ob1;
    cout << "p is pointing to an object of type ";
    cout << typeid(*p).name() << endl;

    p = &ob2;
    cout << "p is pointing to an object of type ";
    cout << typeid(*p).name() << endl;

    return 0;
}

```

The output produced by this program is shown here:

```

p is pointing to an object of type class Base
p is pointing to an object of type class Derived1
p is pointing to an object of type class Derived2

```

When typeid is applied to a base class pointer of a polymorphic type, the type of object pointed to will be determined at runtime, as the output shows.

In all cases, when `typeid` is applied to a pointer of a non-polymorphic class hierarchy, then the base type of the pointer is obtained. That is, no determination of what that pointer is actually pointing to is made. As an experiment, comment-out the virtual function `f()` in `Base` and observe the results. As you will see, the type of each object will be `Base` because that is the type of the pointer.

Since `typeid` is commonly applied to a dereferenced pointer (that is, one to which the `*` operator has been applied), a special exception has been created to handle the situation in which the pointer being dereferenced is null. In this case, `typeid` throws a `bad_typeid` exception.

References to an object of a polymorphic class hierarchy work the same as pointers. When `typeid` is applied to a reference to an object of a polymorphic class, it will return the type of the object actually being referred to, which may be of a derived type. The circumstance where you will most often make use of this feature is when objects are passed to functions by reference.

There is a second form of `typeid` that takes a type name as its argument. This form is shown here:

```
typeid(type-name)
```

For example, the following statement is perfectly acceptable:

```
cout << typeid(int).name();
```

The main use of this form of `typeid` is to obtain a `type_info` object that describes the specified type so that it can be used in a type comparison statement.



1. What makes a static member variable unique?
2. What does `typeid` do?
3. What type of object does `typeid` return?

CRITICAL SKILL 12.8: The Casting Operators

C++ defines five casting operators. The first is the traditional-style cast described earlier in this book. It has been part of C++ from the start. The remaining four were added a few years ago. They are `dynamic_cast`, `const_cast`, `reinterpret_cast`, and `static_cast`. These operators give you additional control over how casting takes place. Each is examined briefly here.

`dynamic_cast`

Perhaps the most important of the additional casting operators is the `dynamic_cast`. The `dynamic_cast` performs a runtime cast that verifies the validity of a cast. If at the time `dynamic_cast` is executed, the cast is invalid, then the cast fails. The general form of `dynamic_cast` is shown here:

```
dynamic_cast<target-type> (expr)
```

Here, `target-type` specifies the target type of the cast, and `expr` is the expression being cast into the new type. The target type must be a pointer or reference type, and the expression being cast must evaluate to a pointer or reference. Thus, `dynamic_cast` can be used to cast one type of pointer into another or one type of reference into another.

The purpose of `dynamic_cast` is to perform casts on polymorphic types. For example, given two polymorphic classes `B` and `D`, with `D` derived from `B`, a `dynamic_cast` can always cast a `D*` pointer into a `B*` pointer. This is because a base pointer can always point to a derived object. But a `dynamic_cast` can cast a `B*` pointer into a `D*` pointer only if the object being pointed to actually is a `D` object. In general, `dynamic_cast` will succeed if the pointer (or reference) being cast is pointing to (or referring to) either an object of the target type or an object derived from the target type. Otherwise, the cast will fail. If the cast fails, then `dynamic_cast` evaluates to null if the cast involves pointers. If a `dynamic_cast` on reference types fails, a `bad_cast` exception is thrown.

Here is a simple example. Assume that `Base` is a polymorphic class and that `Derived` is derived from `Base`.

```
Base *bp, b_ob;
Derived *dp, d_ob;

bp = &d_ob; // base pointer points to Derived object
dp = dynamic_cast<Derived *> (bp); // cast to derived pointer OK
if(dp) cout << "Cast OK";
```

Here, the cast from the base pointer `bp` to the derived pointer `dp` works because `bp` is actually pointing to a `Derived` object. Thus, this fragment displays `Cast OK`. But in the next fragment, the cast fails because `bp` is pointing to a `Base` object, and it is illegal to cast a base object into a derived object.

```
bp = &b_ob; // base pointer points to Base object
dp = dynamic_cast<Derived *> (bp); // error
if(!dp) cout << "Cast Fails";
```

Because the cast fails, this fragment displays `Cast Fails`.

const_cast

The `const_cast` operator is used to explicitly override `const` and/or `volatile` in a cast. The target type must be the same as the source type, except for the alteration of its `const` or `volatile` attributes. The most common use of `const_cast` is to remove `const`-ness. The general form of `const_cast` is shown here:

```
const_cast<type> (expr)
```

Here, `type` specifies the target type of the cast, and `expr` is the expression being cast into the new type. It must be stressed that the use of `const_cast` to cast away `const`-ness is a potentially dangerous feature. Use it with care.

One other point: Only `const_cast` can cast away `const`-ness. That is, `dynamic_cast`, `static_cast`, and `reinterpret_cast` cannot alter the `const`-ness of an object.

`static_cast`

The `static_cast` operator performs a non-polymorphic cast. It can be used for any standard conversion. No runtime checks are performed. Thus, the `static_cast` operator is essentially a substitute for the original cast operator. Its general form is

```
static_cast<type> (expr)
```

Here, `type` specifies the target type of the cast, and `expr` is the expression being cast into the new type.

`reinterpret_cast`

The `reinterpret_cast` operator converts one type into a fundamentally different type. For example, it can change a pointer into an integer and an integer into a pointer. It can also be used for casting inherently incompatible pointer types. Its general form is

```
reinterpret_cast<type> (expr)
```

Here, `type` specifies the target type of the cast, and `expr` is the expression being cast into the new type.

What Next?

The purpose of this book is to teach the core elements of the language. These are the features and techniques of C++ that are used in everyday programming. With the knowledge you now have, you can begin writing real-world, professional-quality programs. However, C++ is a very rich language, and it contains many advanced features that you will still want to master, including:

- The Standard Template Library (STL)
- Explicit constructors
- Conversion functions
- `const` member functions and the `mutable` keyword
- The `asm` keyword
- Overloading the array indexing operator `[]`, the function call operator `()`, and the dynamic allocation operators, `new` and `delete`

Of the preceding, perhaps the most important is the Standard Template Library. It is a library of template classes that provide off-the-shelf solutions to a variety of common data-storage tasks. For

example, the STL defines generic data structures, such as queues, stacks, and lists, which you can use in your programs.

You will also want to study the C++ function library. It contains a wide array of routines that will simplify the creation of your programs.

To continue your study of C++, I suggest reading my book C++: The Complete Reference, published by Osborne/McGraw-Hill, Berkeley, California. It covers all of the preceding, and much, much more. You now have sufficient knowledge to make full use of this in-depth C++ guide.

Module 12 Mastery Check

1. Explain how try, catch, and throw work together to support exception handling.
2. How must the catch list be organized when catching exceptions of both base and derived classes?
3. Show how to specify that a MyExcpt exception can be thrown out of a function called func() that returns void.
4. Define an exception for the generic Queue class shown in Project 12-1. Have Queue throw this exception when an overflow or underflow occurs. Demonstrate its use.
5. What is a generic function, and what keyword is used to create one?
6. Create generic versions of the quicksort() and qsort() functions shown in Project 5-1. Demonstrate their use.
7. Using the Sample class shown here, create a queue of three Sample objects using the generic Queue shown in Project 12-1:

```
class Sample {  
    int id;  
public:  
    Sample() { id = 0; }  
    Sample(int x) { id = x; }  
    void show() { cout << id << endl; }  
};
```

8. Rework your answer to question 7 so that the Sample objects stored in the queue are dynamically allocated.
9. Show how to declare a namespace called RobotMotion.
10. What namespace contains the C++ standard library?
11. Can a static member function access the non-static data of a class?
12. What operator obtains the type of an object at runtime?

13. To determine the validity of a polymorphic cast at runtime, what casting operator do you use?
14. What does `const_cast` do?
15. On your own, try putting the `Queue` class from Project 12-1 in its own namespace called `QueueCode`, and into its own file called `Queue.cpp`. Then rework the `main()` function so that it uses a `using` statement to bring `QueueCode` into view.
16. Continue to learn about C++. It is the most powerful computer language currently available. Mastering it puts you in an elite league of programmers.