

Chapter
1

INTRODUCTION

*Get on the
Fast Track!*



TM

**SYS-ED/
COMPUTER
EDUCATION
TECHNIQUES, INC.**

Objectives

You will learn:

- C Anonymous class types.
- C Nested class declarations.
- C Incomplete declarations.
- C Pointers to class members and member objects.
- C Conversion of pointers to classes.
- C Friend functions and declarations.
- C Templates.

1 Anonymous Class Types

An anonymous class is declared without an identifier. This is useful when a class name is replaced with a typedef name.

Example:

```
typedef struct
{
    unsigned x;
    unsigned y;
} POINT;
```

In some C code, the use of typedef in conjunction with anonymous structures is prevalent. Anonymous classes are also useful for referencing to a class member to appear as though it were not contained in a separate class.

Example:

```
struct PTValue
{
    POINT ptLoc;
    union
    {
        int  iValue;
        long lValue;
    };
};

PTValue ptv;
```

iValue can be accessed using the object member-selection operator (.) as follows:

```
int i = ptv.iValue;
```

Anonymous classes have the following restrictions:

- C Cannot have a constructor or destructor.
- C Cannot be passed as arguments to functions unless type checking is defeated using ellipses.
- C Cannot be returned as return values from functions.

2 Nested Class Declarations

A class can be declared within the scope of another class.

Nested classes are considered to be within the scope of the enclosing class and are available for use within that scope. In order to refer to a nested class from a scope other than its immediate enclosing scope, a fully qualified name must be used.

This code declares nested classes:

```
1 class BufferedIO
2 {
3 public:
4     enum IOError { None, Access, General };
5
6     // Declare nested class BufferedInput.
7     class BufferedInput
8     {
9     public:
10         int read();
11         int good() { return _inputerror == None; }
12     private:
13         IOError _inputerror;
14     };
15
16     // Declare nested class BufferedOutput.
17     class BufferedOutput
18     {
19         // Member list
20     };
21 };
```

BufferedIO::BufferedInput and BufferedIO::BufferedOutput are declared within BufferedIO. These class names are not visible outside the scope of class BufferedIO. However, an object of type BufferedIO does not contain any objects of types BufferedInput or BufferedOutput.

Nested classes can directly use names, type names, names of static members, and enumerators only from the enclosing class. In order to use names of other class members, pointers, references, or object names must be used.

In the BufferedIO example, the enumeration IOError can be accessed directly by member functions in the nested classes, BufferedIO::BufferedInput or BufferedIO::BufferedOutput, as shown in function good.

Nested classes declare only types within class scope. They do not cause contained objects of the nested class to be created. The preceding example declared two nested classes; but did not declare any objects of these class types.

Nesting a class within another class does not give special access privileges to member functions of the nested class. Similarly, member functions of the enclosing class have no special access to members of the nested class.

3 Incomplete Declarations

An incomplete type is a type that describes an identifier, but lacks information needed to determine the size of the identifier.

An "incomplete type" can be:

- C A structure type whose members have not yet been specified.
- C A union type whose members have not yet been specified.
- C An array type whose dimension has not yet been specified.

The void type is an incomplete type that cannot be completed.

3.1 Examples for Creating and Completing the Incomplete Type

In order to complete an incomplete type, specify the missing information.

The following examples demonstrate how to create and complete the incomplete types.

- C In order to create an incomplete structure type, declare a structure type without specifying its members. The `ps` pointer points to an incomplete structure type called `student`.

```
struct student *ps;
```

- C In order to complete an incomplete structure type, declare the same structure type later in the same scope with its members specified.

```
struct student
{
    int num;
} /* student structure now completed */
```

- C In order to create an incomplete array type, declare an array type without specifying its repetition count.

```
char a[]; /* a has incomplete type */
```

- C In order to complete an incomplete array type, declare the same name later in the same scope with its repetition count specified.

```
char a[25]; /* a now has complete type */
```

4 Pointers to Class Members and Member Objects

The use of pointers to class members enhances the type safety of the C++ language.

Three new operators and constructs are used with pointers to members.

| Operator or Construct | Syntax | Use |
|-----------------------|--------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ::* | type::*ptr-name | <p>Declaration of pointer to member. The type specifies the class name, and ptr-name specifies the name of the pointer to member. Pointers to members can be initialized.</p> <p>Example:</p> <pre>MyType::*pMyType = &MyType::i;</pre> |
| .* | obj-name.*ptr-name | <p>Dereference a pointer to a member using an object or object reference.</p> <p>Example:</p> <pre>int j = Object.*pMyType;</pre> |
| ->* | obj-ptr->*ptr-name | <p>Dereference a pointer to a member using a pointer to an object.</p> <p>Example:</p> <pre>int j = pObj->*pMyType;</pre> |

Example:

This code defines a class `TestClass` and the derived type `pValue`, which points to the member `iVar1`:

```
1 #include <iostream.h>
2
3 // Define class TestClass.
4 class TestClass
5 {
6 public:
7     int iVar1;
8     Show() { cout << iVar1 << "\n"; }
9 };
10
11 // Define a derived type pValue that points to iVar1 members of
12 // objects of type TestClass.
13 int TestClass::*pValue = &TestClass::iVar1;
14
15 void main()
16 {
17     TestClass TestClass;           // Define an object of type TestClass.
18     TestClass *pTestClass = &TestClass; // Define a pointer to that object.
19
20     int i;
21
22     TestClass.*pValue = 7777; // Assign to TestClass::iVar1 using .* operator.
23     TestClass.Show();
24
25     i = pTestClass->*pValue; // Dereference a pointer
26                             // using ->* operator.
27     cout << i << "\n";
28 }
```

The pointer to member `pValue` is a new type derived from class `TestClass`. It is more strongly typed than a "plain" pointer to `int` because it points only to `int` members of class `Aclass`.

Example:

Pointers to static members are plain pointers rather than pointers to class members.

```
class StaticMemClass
{
public:
    static int StaticMember;
};
int StaticMemClass::StaticMember = 0;

int *ptrStaticMember = &StaticMemClass::StaticMember;
```

Example:

The type of the pointer is "pointer to int," not "pointer to StaticMemClass::int." Pointers to members can refer to member functions as well as member data.

```
1 // Declare a base class, A, with a virtual function, Identify.
2 // (Note that in this context, struct is the same as class.)
3 struct A
4 {
5     virtual void Identify() = 0;    // No definition for class A.
6 };
7
8 // Declare a pointer to the Identify member function.
9 void (A::*ptrIdentify)() = &A::Identify;
10
11 // Declare class B derived from class A.
12 struct B : public A
13 {
14     void Identify();
15 };
16
17 // Define Identify functions for classe B
18 void B::Identify()
19 {
20     printf( "Identification is B::Identify\n" );
21 }
22
23 void main()
24 {
25     B BObject;                // Declare objects of type B
26     A *ptrA;                  // Declare pointer to type A.
27
28     ptrA = &BObject;         // Make ptrA point to an object of type B.
29     (ptrA->*ptrIdentify)();  // Call Identify function through pointer
30                             // to member ptrIdentify.
31 }
```

The output from this program is:

Identification is B::Identify

The function is called through a pointer to type A. However, because the function is a virtual function, the correct function for the object to which ptrA refers is called.

4.1 Initializing Pointers to Member Objects

A pointer to a const object can be initialized with a pointer to an object that is not const, but not vice versa.

The following initialization is allowed:

```
Window StandardWindow;  
const Window* pStandardWindow( &StandardWindow );
```

The pointer pStandardWindow is declared as a pointer to a const object. Although StandardWindow is not declared as const, the declaration is acceptable because it does not allow an object not declared as const access to a const object.

The reverse of this is:

```
const Window StandardWindow;  
Window* pStandardWindow( &StandardWindow );
```

The preceding code explicitly declares StandardWindow as a const object. Initializing the nonconstant pointer pStandardWindow with the address of StandardWindow generates an error because it allows access to the const object through the pointer. This serves to allow for the removal of the const attribute from the object.

5 Conversion of Pointers to Classes

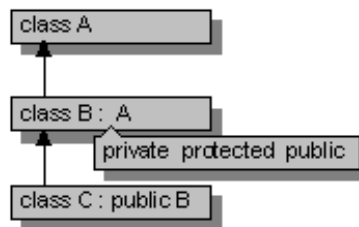
There are two cases in which a pointer to a class can be converted to a pointer to a base class.

Case 1:

The first case is when the specified base class is accessible and the conversion is unambiguous.

Whether a base class is accessible depends on the kind of inheritance used in derivation.

Consider the following inheritance:



This table lists the base-class accessibility for the situation illustrated in the figure.

| Type of Function | Derivation | Conversion from B* to A* Legal? |
|--------------------------------------|--------------------------------|---------------------------------|
| External function (not class-scoped) | Private Protected Public | No No Yes |
| B member function (in B scope) | Private Protected Public | Yes Yes Yes |
| C member function (in C scope) | Private Protected Public | No Yes Yes |

Case 2:

The second case in which a pointer to a class can be converted to a pointer to a base class is when an explicit type conversion is being used.

The result of such a conversion is a pointer to the "subobject," the portion of the object that is completely described by the base class.

The following code defines two classes, A and B, where B is derived from A. It then defines bObject, an object of type B, and two pointers (pA and pB) that point to the object.

```
class A
{
public:
    int AComponent;
    int AMemberFunc();
};

class B : public A
{
public:
    int BComponent;
    int BMemberFunc();
};

B bObject;
A *pA = &bObject;
B *pB = &bObject;

pA->AMemberFunc(); // OK in class A
pB->AMemberFunc(); // OK: inherited from class A
pA->BMemberFunc(); // Error: not in class A
```

The pointer pA is of type A *, which can be interpreted as meaning "pointer to an object of type A." Members of bObject (such as BComponent and BMemberFunc) are unique to type B and are therefore inaccessible through pA. The pA pointer allows access only to those characteristics (member functions and data) of the object that are defined in class A.

6 Friend Functions

In some circumstances, it is more convenient to grant member-level access to functions that are not members of a class or to all functions in a separate class.

With the friend keyword, programmers can designate:

C the specific functions.

OR

C the classes whose functions can access not only public members but also protected and private members.

Friend functions are not considered class members; they are normal external functions that are given special access privileges. Friends are not in the class's scope, and they are not called using the member-selection operators (. and ->) unless they are members of another class.

Example:

This code illustrates a Point class and an overloaded operator, operator+. It primarily illustrates friends, not overloaded operators.

```
1 #include <iostream.h>
2
3 // Declare class Point.
4 class Point
5 {
6 public:
7     // Constructors
8     Point() { _x = _y = 0; }
9     Point( unsigned x, unsigned y ) { _x = x; _y = y; }
10    // Accessors
11    unsigned x() { return _x; }
12    unsigned y() { return _y; }
13    void    Print() { cout << "Point(" << _x << ", " << _y << ")"
14                << endl; }
15    // Friend function declarations
16    friend Point operator+( Point& pt, int nOffset );
17    friend Point operator+( int nOffset, Point& pt );
18
19 private:
20    unsigned _x;
21    unsigned _y;
22 };
23
24 // Friend-function definitions
25 //
26 // Handle Point + int expression.
27 Point operator+( Point& pt, int nOffset )
28 {
```

```
29     Point ptTemp = pt;
30     // Change private members _x and _y directly.
31     ptTemp._x += nOffset;
32     ptTemp._y += nOffset;
33
34     return ptTemp;
35 }
36
37 // Handle int + Point expression.
38 Point operator+( int nOffset, Point& pt )
39 {
40     Point ptTemp = pt;
41
42     // Change private members _x and _y directly.
43     ptTemp._x += nOffset;
44     ptTemp._y += nOffset;
45
46     return ptTemp;
47 }
48
49 // Test overloaded operator.
50 void main()
51 {
52     Point pt( 10, 20 );
53     pt.Print();
54
55     pt = pt + 3;      // Point + int
56     pt.Print();
57
58     pt = 3 + pt;     // int + Point
59     pt.Print();
60 }
```

When the expression `pt + 3` is encountered in the `main` function, the compiler determines whether an appropriate user-defined `operator+` exists. In this case, the function `operator+(Point pt, int nOffset)` matches the operands, and a call to the function is issued.

In the second case (the expression `3 + pt`), the function `operator+(Point pt, int nOffset)` matches the supplied operands. Therefore, supplying these two forms of `operator+` preserves the commutative properties of the `+` operator.

A user-defined `operator+` can be written as a member function, but it takes only one explicit argument: the value to be added to the object. As a result, the commutative properties of addition cannot be correctly implemented with member functions; they must use friend functions instead.

7 Friend Declarations

If a friend function is declared that was not previously declared, that function is exported to the enclosing nonclass scope. Functions declared in a friend declaration are treated as if they had been declared using the `extern` keyword.

Although functions with global scope can be declared as friends prior to their prototypes, member functions cannot be declared as friends before the appearance of their complete class declaration.

This code shows why the fails occurs:

```
class ForwardDeclared;           // Class name is known.

class HasFriends
{
    friend int ForwardDeclared::IsAFriend(); // Error.
};
```

The preceding code enters the class name `ForwardDeclared` into scope, but the complete declaration -- specifically, the portion that declares the function `IsAFriend` -- is not known. Therefore, the friend declaration in class `HasFriends` generates an error.

In order to declare two classes that are friends of one another, the entire second class must be specified as a friend of the first class. The reason for this restriction is that the compiler has sufficient information to declare individual friend functions only at the point where the second class is declared.

Although the entire second class must be a friend to the first class, it is possible to select which functions in the first class will be friends of the second class.

8 Templates

Templates are a mechanism for generating functions and classes based on type parameters. They are sometimes called "parameterized types". The utilization of templates makes it possible to design a single class that operates on data of many types, instead of having to create a separate class for each type.

In order to create a type-safe function that returns the minimum of two parameters without using templates, it will be necessary to write a set of overloaded functions such as:

```
// min for ints
int min( int a, int b )
    return ( a < b ) ? a : b;

// min for longs
long min( long a, long b )
    return ( a < b ) ? a : b;

// min for chars
char min( char a, char b )
    return ( a < b ) ? a : b;

//etc...
```

By using templates, this duplication can be reduced to a single templated function:

```
template <class T> T min( T a, T b )
    return ( a < b ) ? a : b;
```

Templates can significantly reduce source code size and increase code flexibility without reducing type safety.

8.1 Function Templates

Class templates define a family of related classes that are based on the parameters passed to the class upon instantiation. Function templates are similar to class templates, but define a family of functions.

The template declaration specifies a set of parameterized classes or functions.

Syntax:

```
template-declaration:
    template < template-argument-list > declaration
template-argument-list:
    template-argument
    template-argument-list , template-argument
template-argument:
    type-argument
    argument-declaration
type-argument:
    class identifier
```

- C The declaration declares a function or a class.

With function templates, each template-argument must appear at least once in the template-argument-list of the function being declared.

- C The template-argument-list is a list of arguments used by the template function that specifies which parts of the following code will vary.

Example:

```
template< class T, int i > class MyStack...
```

In this case the template can receive a type (class T) and a constant parameter (int i). The template will use type T and the constant integer i upon construction. Within the body of the MyStack declaration, it will be necessary to refer to a T identifier.

- C A template declaration itself does not generate code; it specifies a family of classes or functions, one or more of which will be generated when referenced by other code. Template declarations have global scope.

Example:

This function template swaps two items:

```
template< class T > void MySwap( T& a, T& b )
{
    T c;
    c = a; a = b; b = c;
}
```

Although this function could be performed by a nontemplated function, using void pointers, the template version is type-safe.

Consider the following calls:

```
int j = 10;
int k = 18;
CString Hello = "Hello, Windows!";
MySwap( j, k );           //OK
MySwap( j, Hello );     //error
```

The second MySwap call triggers a compile-time error, since the compiler cannot generate a MySwap function with parameters of different types. If void pointers were used, both function calls would compile correctly, but the function would not work properly at run time.

8.2 Class Templates

Templates can be used for creating a family of classes that operate on a type.

Example:

```
template <class T, int i> class TempClass
{
public:
    TempClass( void );
    ~TempClass( void );
    int MemberSet( T a, int b );
private:
    T Tarray[i];
    int arraysize;
};
```

The templated class uses two parameters, a type T and an int i.

- C The T parameter can be passed any type, including structures and classes.
- C The i parameter has to be passed an integer constant.

Since i is a constant defined at compile time, a member array of size i can be defined using a standard automatic array declaration.

Unlike function templates, all template parameters do not have to be used in the definition of a templated class.