نموذج وصف المقرر

مراجعة أداء مؤسسات التعليم العالي ((مراجعة البرنامج الأكاديمي))

وصف المقرر

دراسة البرمجة المهيكلة والبرمجة الكيانية وما يعرف بالبرمجة الشيئية ومعرفة الايعازات والدوال لتهيئة الطالب لمعرفة كيفية كتابة مجموعة من الاوامرومعرفة ما هي الايعازات وكيفية بناء الاصناف والكائنات وما يحمله الصنف من خواص ودوال وكيفية بناء عدة اصناف وعدة كائنات وكيف يتم وراثة الخصائص بينهم .

1. المؤسسة التعليمية	جامعة الانبار / كلية علوم الحاسوب وتكنولوجيا المعلومات
2. القسم الجامعي / المركز	قسم نظم المعلومات
3. اسم / رمز المقرر	برمجة كيانية 2 / 02314
4. البرامج التي يدخل فيها	
5. أشكال الحضور المتاحة	دوام رسمي
6. الفصل / السنة	الفصل الثاني- 2023
7. عدد الساعات الدراسية (الكلي)	75
8. تاريخ إعداد هذا الوصف	
9-أهداف المقرر :	
أ- اكتساب الطالب لمفهوم البرمجة الكيانية والاصناف والكائنات وكيفية التعامل معها.	
ب- توضيح مفهوم الاصناف وما هي الدوال والخصائص الخاصة بيها والكائنات لكل صنف .	
ج- إعطاء الطالب خبرة في التعامل مع الكائنات والاصناف وتوزيع الخصائص والدوال .	

9. مخرجات التعلم وطرائق التعليم والتعلم والتقييم

أ- المعرفة والفهم
- اكتساب القدرة والمهارة في تمييز الايعازات البرمجية والدوال الخاصة بالبرمجة الكيانية والتعامل معها.
- اكتساب مهارة التمييز بين الكائنات والاصناف والدوال والربط بينهما.
-التعامل مع الصفات والخصائص الخاصة بكل صنف وبرمجة الدوال.

ب - المهارات الخاصة بالموضوع
–التدريب الصيفي
– بحوث تخرج
– تقارير علمية

طرائق التعليم والتعلم

- الاختبارات اليومية المفاجئة والاسبوعية المستمرة .
- التدريبات والأنشطة في قاعة الدرس .
- إرشاد الطلاب إلى بعض المواقع الالكترونية للإفادة منها.

طرائق التقييم

- المشاركة في قاعة الدرس.
- تقديم الأنشطة
- اختبارات فصلية ونهائية وأنشطة .

ج- مهارات التفكير
- تطوير قدرة الطالب للعمل على أداء الواجبات وتسليمها في الموعد المقرر.
- تحليل المشكلة بشكل برمجي وايجاد الحلول لها على اساس النتائج المتوقعة.
- تطوير قدرة الطالب على الحوار والمناقشة.

طرائق التعليم والتعلم
• إدارة المحاضرة على نحو تطبيقي مرتبط بواقع الحياة اليومية لجذب الطالب الى موضوع الدرس دون الابتعاد عن صلب الموضوع لتكون المادة مرنه قابله للفهم والتحليل . • تكليف الطالب ببعض الأنشطة والواجبات الجماعية. • تخصيص نسبة من الدرجة للواجبات اليومية والاختبارات .
طرائق التقييم
• المشاركة الفاعلة في قاعة الدرس دليل التزام الطالب وتحمله المسؤولية. • الالتزام بالموعد المحدد في تقديم الواجبات والبحوث. • تعبر الاختبارات الفصلية والنهائية عن الالتزام والتحصيل المعرفي والمهاري.
د - المهارات العامة والمنقولة (المهارات الأخرى المتعلقة بقابلية التوظيف والتطور الشخصي). - تنمية قدرة الطالب على التعامل مع وسائل التقنية. - تنمية قدرة الطالب على التعامل مع الإنترنت. - تنمية قدرة الطالب على التعامل مع الوسائل المتعددة. - تطوير قدرة الطالب على الحوار والمناقشة.

10. بنية المقرر

الأسبوع	الساعات	مخرجات التعلم المطلوبة	اسم الوحدة / المساق أو الموضوع	طريقة التعليم	طريقة التقييم
الاول	5	الفصل الاول	**Introduction to Operator Overloading**	نظري+عملي	اسئلة عامة ومناقشة
الثاني	5	**الدوال والاصناف والكائنات**	**Operator Overloading Using Member Functions**	نظري+عملي	اسئلة عامة ومناقشة او امتحان اني
الثالث	5	**الاصناف والعمليات**	**Unary Operators Overloading**	نظري+عملي	أسئلة عامة ومناقشة
الرابع	5	المواضيع السابقة	**Operator Overloading Tips and Restrictions**	نظري+عملي	الواجبات الجماعية+ مناقشة
الخامس	5	المواضيع السابقة	**Nonmember Operator Functions**	نظري+عملي	امتحان اني
السادس	5	المواضيع السابقة	**Using a Friend to Overload a Unary Operator**	نظري+عملي	أسئلة عامة ومناقشة او امتحان اني
السابع	5	المواضيع السابقة	**Overloading the Relational and Logical Operators**	نظري+عملي	اسئلة عامة و مناقشة
الثامن	5	الدوال والاصناف والكائنات	**Introducing Inheritance**	نظري+عملي	الواجبات الجماعية+ مناقشة
التاسع	5	الوراثة	**Base Class Access Control**	نظري+عملي	اسئلة عامة
العاشر	5	الوراثة	**Using protected Members**	نظري+عملي	الواجبات الجماعية
الحادي عشر	5	المواضيع السابقة	**Inheriting Multiple Base Classes**	نظري+عملي	اسئلة عامة
الثاني عشر	5	المواضيع السابقة	**Constructors, Destructors, and Inheritance**	نظري+عملي	امتحان شهري
الثالث عشر	5	المواضيع السابقة	**Passing Parameters to Base Class Constructors**	نظري+عملي	اسئلة عامة
الرابع عشر	5	المواضيع السابقة	**Virtual Base Classes**	نظري+عملي	الواجبات الجماعية+ مناقشة
الخامس عشر	5		**Final Exam**	نظري+عملي	امتحان شهري

11- القراءات المطلوبة : ▪ كتب المقرر ▪ اخرى	(1) **C++ from the Ground Up, Herbert Scheldt, Third Edition , McGraw-Hill/Osborne,2013.**
متطلبات خاصة	
الخدمات الاجتماعية (وتشمل على سبيل المثال محاضرات الضيوف والتدريب المهني والدراسات الميدانية)	التطبيق العملي في الشركات والدوائر ذات العلاقة ومشاريع بحوث التخرج.

11. القبول	
المتطلبات السابقة	لا توجد
أقل عدد من الطلبة	10
أكبر عدد من الطلبة	66

CHAPTER FOUR

Base Class Access Control

Base Class Access Control

When one class inherits another, the members of the base class become members of the derived class. The access status of the base class members inside the derived class is determined by the access specifier used for inheriting the base class. The base class access specifier must be public, private, or protected. If the access specifier is not used, then it is private by default if the derived class is a class. If the derived class isa struct, then public is the default in the absence of an explicit access specifier. Let's examine the ramifications of using public or private access. (The protected specifieris described in the next section.)

When a base class is inherited as public, all public members of the base class become public members of the derived class. In all cases, the private elements of the base class remain private to that class, and are not accessible by members of the derived class. For example, in the following program, the public members of base become public members of derived. Thus, they are accessible by other parts of the program.

```
#include <iostream.h>
class base
{
     inti, j;
  public:
     void set(int a, int b) { i = a; j = b; }
     void show() { cout<<i<< " " << j << "\n"; }
};

class derived : public base
{
    int k;
  public:
     derived(int x) { k = x; }
     void showk() { cout<< k << "\n"; }
```

```
};
intmain()
{
    derived ob(3);
    ob.set(1, 2); // access member of base
    ob.show();    // access member of base
    ob.showk();   // uses member of derived class
    return 0;
}
```

When a base class is inherited as private, its public members become private members of the derived class.Since set( ) and show( ) are inherited as public, they can be called on an object of type derived from within main( ). Since i and j are specified as private, they remain private to base.

The opposite of public inheritance is private inheritance. When the base class is inherited as private, then all public members of the base class become private members of the derived class. For example, the program shown next will not compile, because both set( )and show( ) are now private members of derived, and thus cannot be called from main( ).

```
// This program won't compile.
#include <iostream.h>
class base
{
     inti, j;
  public:
     void set(int a, int b) { i = a; j = b; }
     void show() { cout<<i<< " " << j << "\n"; }
};
// Public elements of base are private in derived.
class derived : private base
{
     int k;
```

```
   public:
      derived(int x) { k = x; }
      void showk() { cout<< k << "\n"; }
};
intmain()
{
   derived ob(3);
   ob.set(1, 2); // Error, can't access set()
   ob.show(); // Error, can't access show()
   return 0;
}
```

The key point to remember is that when a base class is inherited as private, public members of the base class become private members of the derived class. This means that they are still accessible by members of the derived class, but cannot be accessed by other parts of your program.

Using protected Members

In addition to public and private, a class member can be declared as protected. Further, a base class can be inherited as protected. Both of these actions are accomplished by using the protected access specifier. The protected keyword is included in C++ to provide greater flexibility for the inheritance mechanism.

When a member of a class is declared as protected, that member is not accessible to other, non-member elements of the program. With one important exception, access to a protected member is the same as access to a private member; it can be accessed only by other members of the class of which it is a part. The sole exception to this rule is when a protected member is inherited. In this case, a protected member differs substantially from a private one.

As you know, a private member of a base class is not accessible by any other partof your program, including any derived class. However, protected members behave differently. When a base class is

inherited as public, protected members in the base class become protected members of the derived class, and are accessible to the derived class. Therefore, by using protected, you can create class members that are private to their class, but that can still be inherited and accessed by a derived class.Consider this sample program:

```
  #include <iostream.h>
  class base
  {
    protected:
      inti, j; // private to base, but accessible to derived
    public:
      void set(int a, int b) { i = a; j = b; }
      void show() { cout<<i<< " " << j << "\n"; }
  };
class derived : public base
{
         int k;
       public:
         // derived may access base's i and j
         void setk() { k = i*j; }
         void showk() { cout<< k << "\n"; }
};
   intmain()
   {
derived ob;

       ob.set(2, 3); // OK, known to derived
       ob.show(); // OK, known to derived
       ob.setk();
       ob.showk();

       return 0;
    }
```

Here, because base is inherited by derived as public, and because i and j are declared as protected, derived's function setk( ) may access them. If i and j were declared as private by base, then derived would not have access to them, and the program would not compile.

When a derived class is used as a base class for another derived class, then any protected member of the initial base class that is inherited (as public) by the first derived class can be inherited again, as a protected member, by a second derived class. For example, the following program is correct, and derived2 does, indeed, have access to i and j:

```
#include <iostream.h>
class base
{
   protected:
      inti, j;
   public:
      void set(int a, int b) { i = a; j = b; }
      void show() { cout<<i<< " " << j << "\n"; }
};
// i and j inherited as protected.
class derived1 : public base
{
      int k;
   public:
      void setk() { k = i*j; } // legal
      void showk() { cout<< k << "\n"; }
};
// i and j inherited indirectly through derived1.
class derived2 : public derived1
{
      int m;
   public:
      void setm() { m = i-j; } // legal
      void showm() { cout<< m << "\n"; }
};
intmain()
{
   derived1 ob1;
   derived2 ob2;

   ob1.set(2, 3);
   ob1.show();
```

```
    ob1.setk();
    ob1.showk();

    ob2.set(3, 4);
    ob2.show();
    ob2.setk();
    ob2.setm();
    ob2.showk();
    ob2.showm();

    return 0;
}
```

When a base class is inherited as private, protected members of the base class become private members of the derived class. Therefore, in the preceding example, if basewere inherited as private, then all members of base would become private members of derived1, meaning that they would not be accessible to derived2. (However, i and j would still be accessible to derived1.) This situation is illustrated by the following program, which is in error (and won't compile). The comments describe each error.

```
// This program won't compile.
#include <iostream.h>
class base
{
   protected:
      inti, j;
   public:
      void set(int a, int b) { i = a; j = b; }
      void show() { cout<<i<< " " << j << "\n"; }
};
// Now, all elements of base are private in derived1.
   class derived1 : private base
   {
        int k;
   public:
      // This is legal because i and j are private to
```

```
        derived1. void setk() { k = i*j; }    // OK
        void showk() { cout<< k << "\n"; }
};
// Access to i, j, set(), and show() not inherited.
class derived2 : public derived1
{
   int m;
   public:
// Illegal because i and j are private to derived1.
   void setm() { m = i-j; }    // error
   void showm() { cout<< m << "\n"; }
};
intmain()
{
derived1 ob1;
   derived2 ob2;
   ob1.set(1, 2); // Error, can't use set()
   ob1.show();    // Error, can't use show()
   ob2.set(3, 4); // Error, can't use set()
   ob2.show();    // Error, can't use show()
   return 0;
}
```

Even though base is inherited as private by derived1, derived1 still has access to the public and protected elements of base. However, it cannot pass this privilege along. This is the reason that protected is part of the C++ language. It provides a means of protecting certain members from being modified by non-member functions, but allows them to be inherited.

The protected specifier can also be used with structures. It cannot be used with a union, however, because a union cannot inherit another class or be inherited. (Some compilers will accept its use in a union declaration, but because unions cannot participate in inheritance, protected is the same as private in this context.)

The protected access specifier may occur anywhere in a class declaration, although typically it occurs after the (default) private

members are declared, and before the public members. Thus, the most common full form of a class declaration is

```
class  class-name
{
   private members
   protected:
   protected members
   public:
   public members
};
```

Of course, the protected category is optional.

CHAPTERFOUR

Constructors, Destructors, and Inheritance

Constructors, Destructors, and Inheritance

There are two important questions that arise relative to constructors and destructors when inheritance is involved. First, when are base class and derived class constructors and destructors called? Second, how can parameters be passed to a base class constructor? This section answers these questions.

When Constructors and Destructors Are Executed

It is possible for a base class, a derived class, or both, to contain a constructor and/or destructor. It is important to understand the order in which these are executed when an object of a derived class comes into existence and when it goes out of existence.Examine this short program:

```
#include <iostream.h>
class base
{
   public:
      base() { cout<< "Constructing base\n"; }
      ~base() { cout<< "Destructing base\n"; }
};
class derived: public base
{
   public:
      derived() { cout<< "Constructing derived\n"; }
      ~derived() { cout<< "Destructing derived\n"; }
};
intmain()
{
   derived ob;
   // do nothing but construct and destruct ob
   return 0;
}
```

As the comment in main( ) indicates, this program simply constructs and then destroys an object called ob, which is of class derived. When executed, this program displays:

```
Constructing base
Constructing derived
Destructing derived
Destructing base
```

As you can see, the constructor of base is executed, followed by the constructor of derived. Next (since ob is immediately destroyed in this program), the destructor of derived is called, followed by that of base.

The results of the foregoing experiment can be generalized as follows: When an object of a derived class is created, the base class constructor is called first, followed by the constructor for the derived class. When a derived object is destroyed, its destructor is called first, followed by the destructor for the base class. Put differently, constructors are executed in the order of their derivation. Destructors are executed in reverse order of derivation.

If you think about it, it makes sense that constructor functions are executed in the order of their derivation. Because a base class has no knowledge of any derived class, any initialization it needs to perform is separate from, and possibly prerequisite to, any initialization performed by the derived class. Therefore, it must be executed first.

Likewise, it is quite sensible that destructors be executed in reverse order of derivation. Since the base class underlies a derived class, the destruction of the base class implies the destruction of the derived class. Therefore, the derived destructor must be called before the object is fully destroyed.In the case of a large class hierarchy (i.e., where a derived class becomes the base class for another derived class), the general rule applies: Constructors are called in order of derivation, destructors in reverse order. For example, this program

```
#include <iostream.h>
class base
{
   public:
      base() { cout<< "Constructing base\n"; }
      ~base() { cout<< "Destructing base\n"; }
};
class derived1 : public base
{
   public:
      derived1() { cout<< "Constructing derived1\n"; }
      ~derived1() { cout<< "Destructing derived1\n"; }
};
class derived2: public derived1
{
   public:
      derived2() { cout<< "Constructing derived2\n"; }
      ~derived2() { cout<< "Destructing derived2\n"; }
};
intmain()
{
  derived2 ob;// construct and destruct ob
  return 0;
}
```

displays this output:

```
Constructing base
Constructing derived1
Constructing derived2
Destructing derived2
Destructing derived1
Destructing base
```

The same general rule applies in situations involving multiple base classes. For example, this program

```
#include <iostream.h>
class base1
{
   public:
```

```
      base1() { cout<< "Constructing base1\n"; }
      ~base1() { cout<< "Destructing base1\n"; }
};
class base2
{
   public:
      base2() { cout<< "Constructing base2\n"; }
      ~base2() { cout<< "Destructing base2\n"; }
};
class derived: public base1, public base2
{
   public:
      derived() { cout<< "Constructing derived\n"; }
      ~derived() { cout<< "Destructing derived\n"; }
};
intmain()
{
   derived ob;// construct and destruct ob
   return 0;
}
```

produces this output:

```
Constructing base1
Constructing base2
Constructing derived
Destructing derived
Destructing base2
Destructing base1
```

As you can see, constructors are called in order of derivation, left to right, as specified in derived's inheritance list. Destructors are called in reverse order, right to left. This means that if base2 were specified before base1 in derived's list, as shown here:

```
class derived: public base2, public base1 {
```

then the output of the preceding program would look like this:

```
Constructing base2
Constructing base1
Constructing derived
```

```
Destructing derived
Destructing base1
Destructing base2
```

Copy construct

Creating and Using a Copy Constructor

One of the more important forms of an overloaded constructor is the copy constructor. As earlier examples have shown, problems can occur when an object is passed to, or returned from, a function. As you will learn in this section, one way to avoid these problems is to define a copy constructor, which is a special type of overloaded constructor.

To begin, let's restate the problems that a copy constructor is designed to solve. When an object is passed to a function, a bitwise (i.e., exact) copy of that object is made and given to the function parameter that receives the object. However, there are cases in which this identical copy is not desirable. For example, if the object contains a pointer to allocated memory, then the copy will point to the same memory as does the original object.

Therefore, if the copy makes a change to the contents of this memory, it willbe changed for the original object, too! Furthermore, when the function terminates, the copy will be destroyed, thus causing its destructor to be called. This may also have undesired effects on the original object.

A similar situation occurs when an object is returned by a function. The compiler will generate a temporary object that holds a copy of the value returned by the function. (This is done automatically, and is beyond your control.) This temporary object goes out of scope once the value is returned to the calling routine, causing the temporary object's destructor to be called. However, if the destructor destroys something needed by the calling routine, trouble will follow.

At the core of these problems is the creation of a bitwise copy of the object. To prevent them, you need to define precisely what occurs when a copy of an object is made so that you can avoid undesired side effects. The way you accomplish this is by creatinga copy constructor.

Before we explore the use of the copy constructor, it is important for you to understandthat C++ defines two distinct types of situations in which the value of one object is given to another. The first situation is assignment. The second situation is initialization, which can occur three ways:

- ***When one object explicitly initializes another, such as in a declaration***
- ***When a copy of an object is passed as a parameter to a function***
- ***When a temporary object is generated (most commonly, as a return value)***

The copy constructor applies only to initializations. It does not apply to assignments.The most common form of copy constructor is shown here:

```
classname (constclassname&obj) {
// body of constructor
}
```

Here, obj is a reference to an object that is being used to initialize another object. For example, assuming a class called myclass, and y as an object of type myclass, then the following statements would invoke the myclass copy constructor:

```
myclass x = y; // y explicitly initializing x
func1(y);     // y passed as a parameter
y = func2(); // y receiving a returned object
```

In the first two cases, a reference to y would be passed to the copy constructor. In the third, a reference to the object returned by func2( ) would be passed to the copy constructor.To fully explore the value of copy constructors, let's see how they impact each of the three situations to which they apply.

Copy Constructors and Parameters

When an object is passed to a function as an argument, a copy of that object is made. If a copy constructor exists, the copy constructor is called to make the copy. Here is a program that uses a copy constructor to properly handle objects of type myclass when they are passed to a function. (This is a corrected version of the incorrect program shown earlier in this chapter.)

```
// Use a copy constructor to construct a parameter.
#include <iostream.h>
#include <stdlib.h>
class myclass
{
     int *p;
  public:
     myclass(int i); // normal constructor
     myclass(constmyclass&ob); // copy constructor
     ~myclass();
     intgetval() { return *p; }
};
// Copy constructor.
myclass::myclass(constmyclass&obj)
{
  p = new int;
  *p = *obj.p; // copy value
  cout<< "Copy constructor called.\n";
}
// Normal Constructor.
myclass::myclass(int i)
{
  cout<< "Allocating p\n";
  p = new int;
  *p = i;
}
myclass::~myclass()
{
cout<< "Freeing p\n";
```

```
delete p;
}
// This function takes one object parameter.
void display(myclassob)
{
     cout<<ob.getval() << '\n';
}
int main()
{
   myclass a(10);
   display(a);
   return 0;
}
```

This program displays the following output:

```
Allocating p
Copy constructor called.
10
Freeing p
Freeing p
```

Here is what occurs when the program is run: When a is created inside main( ), the normal constructor allocates memory and assigns the address of that memory to a.p. Next, a is passed to ob of display( ). When this occurs, the copy constructor is called, and a copy of a is created. The copy constructor allocates memory for the copy, and a pointer to that memory is assigned to the copy's p member. Next, the value stored at the original object's p is assigned to the memory pointed to by the copy's p. Thus, the areas of memory pointed to by a.p and ob.p are separate and distinct, but the valuesthat they point to are the same. If the copy constructor had not been created, then the default bitwise copy would have caused a.p and ob.p to point to the same memory.

When display( ) returns, ob goes out of scope. This causes its destructor to be called, which frees the memory pointed to by ob.p. Finally, when main( ) returns, a goes out of scope, causing its destructor to free a.p. As you can see, the use of the copy constructor has eliminated the destructive side effects associated with passing an object to a function.

Copy Constructors and Initializations

The copy constructor is also invoked when one object is used to initialize another. Examine this sample program:

```
// The copy constructor is called for initialization.
   #include <iostream.h>
   #include <stdlib.h>
   class myclass
   {
        int *p;
     public:
        myclass(int i); // normal constructor
        myclass(constmyclass&ob); // copy constructor
        ~myclass();
        intgetval() { return *p; }
   };

   // Copy constructor.
   myclass::myclass(constmyclass&ob)
   {
      p = new int;
      *p = *ob.p; // copy value
      cout<< "Copy constructor allocating p.\n";
   }

   // Normal constructor.
   myclass::myclass(int i)
   {
      cout<< "Normal constructor allocating p.\n";
      p = new int;
      *p = i;
   }
   myclass::~myclass()
   {
      cout<< "Freeing p\n";
      delete p;
   }
```

```
int main()
{
myclass a(10); // calls normal constructor
myclass b = a; // calls copy constructor
return 0;
}
```

This program displays the following output:

```
Normal constructor allocating p.
Copy constructor allocating p.
Freeing p
Freeing p
```

As the output confirms, the normal constructor is called for object a. However, when a is used to initialize b, the copy constructor is invoked. The use of the copy constructor ensures that b will allocate its own memory. Without the copy constructor, b would simply be an exact copy of a, and a.p would point to the same memory as b.p.

Keep in mind that the copy constructor is called only for initializations. For example, the following sequence does not call the copy constructor defined in the preceding program:

```
myclass a(2), b(3);
// ... b = a;
```

In this case, b = a performs the assignment operation, not a copy operation.

Using Copy Constructors When an Object Is Returned

The copy constructor is also invoked when a temporary object is created as the result of a function returning an object. Consider this short program:

```
#include <iostream>
class myclass {
public:
   myclass() { cout<< "Normal constructor.\n"; }
   myclass(constmyclass&obj)
   { cout<< "Copy constructor.\n"; }
};

myclass f()
{
   myclassob; // invoke normal constructor
   return ob; // implicitly invoke copy constructor
}
int main()
{
myclass a; // invoke normal constructor
   a = f(); // invoke copy constructor
   return 0;
}
```

This program displays the following output:

```
Normal constructor.
Normal constructor.
Copy constructor.
```

Here, the normal constructor is called twice: once when a is created inside main( ), and once when ob is created inside f( ). The copy constructor is called when the temporary object is generated as a return value from f( ).Although copy constructors may seem a bit esoteric at this point, virtually every real-world class will require one, due to the side effects that often result from the default bitwise copy.

The this Keyword

Each time a member function is invoked, it is automatically passed a pointer, called ***this***, to the object on which it is called. The this pointer is an implicit parameter toall member functions. Therefore, inside a member function, this may be used to refer to the invoking object.As you know, a member function can directly access the private data of its class. For example, given this class,

```
class cl
{int i;
void f() { ... };
// ...
};
```

inside f( ), the following statement can be used to assign i the value10:

```
i = 10;
```

In actuality, the preceding statement is shorthand for this one:

```
this->i = 10;
```

To see how the this pointer works, examine the following short program:

```
#include <iostream.h>
class cl
{int i;
  public:
     void load_i(intval) { this->i = val; }
     // same as i = val
     intget_i() { return this->i; }// same as return i
} ;
int main()
{cl o;
  o.load_i(100);
  cout<<o.get_i();
return 0;
}
```

This program displays the number

```
100.
```

Friend function

2.1 Introduction

This chapter continues the discussion of the class begun in Lecture 1. It discusses friend functions, overloading constructors, passing objects to functions, andreturning objects. It also examines a special type of constructor, called the copy constructor, which is used when a copy of an object is needed. The chapter concludes with a description of the this keyword.

2.2 Friend Functions

It is possible to allow a non-member function access to the private members of a class by declaring it a friend of the class. To make a function a friend of a class, include its prototype in the public section of the class declaration and precede it with the friend keyword. For example, in this fragment frnd( ) is declared to be a friend of the class cl:

```
class cl
{
// ... public:
friend void frnd(cl ob);
};
```

The friend keyword gives a non- member function access to the private membersof a class.As you can see, the keyword friend precedes the rest of the prototype. A function may be a friend of more than one class.Here is a short example that uses a friend function to access the private members of myclass:

```
// Demonstrate a friend function.
#include <iostream.h>
class myclass
{
int a, b;
public:
    myclass(int i, int j) { a=i; b=j; }
```

```
        friend int sum(myclass x);
        // sum() is a friend of myclass
    };
 // Note: sum() is not a member function of any class.
 int sum(myclass x)
{
    /* Because sum() is a friend of myclass,
    it can directly access a and b. */
    return x.a + x.b;
}
    int main()
    {
    myclass n(3, 4);
    cout<< sum(n);
    return 0;
    }
```

In this example, the sum( ) function is not a member of myclass. However, it still has full access to the private members of myclass. Specifically, it can access x.a and x.b. Notice also that sum( ) is called normally—not in conjunction with an object and the dot operator. Since it is not a member function, it does not need to be qualified with an object's name. (In fact, it cannot be qualified with an object.) Typically, a friend function is passed one or more objects of the class for which it is a friend, as is the case with sum( ).

While there is nothing gained by making sum( ) a friend rather than a member function of myclass, there are some circumstances in which friend functions are quite valuable. First, friends can be useful for overloading certain types of operators. Second, friend functions simplify the creation of some types of I/O functions. Both of these uses are discussed later in this course.

Thethird reason that friend functions may be desirable is that, in some cases, two or more classes may contain members that are interrelated relative to other parts of your program. For example,

imagine two different classes that each display a pop-up message on the screen when some sort of event occurs. Other parts of your program that are designed to write to the screen will need to know whether the pop-up message is active, so that no message is accidentally overwritten. It is possible to create a member function in each class that returns a value indicating whether a message is active or not; however, checking this condition involves additional overhead (i.e., two function calls, not just one). If the status of the pop-up message needs to be checked frequently, the additional overhead may not be acceptable. However, by using a friend function, it is possible to directly check the status of each object by calling only one function that has access to both classes. In situations like this, a friend function helps you write more efficient code. The following program illustrates this concept.

```
// Use a friend function.
#include <iostream.h>
constint IDLE=0;
constint INUSE=1;
class C2;    // forward declaration
class C1
{
int status;  // IDLE if off, INUSE if on screen
public:
void set_status(int state);
friend int idle(C1 a, C2 b);
};
class C2
{int status; // IDLE if off, INUSE if on screen
public:
void set_status(int state);
friend int idle(C1 a, C2 b);
};
void C1::set_status(int state)
{
status = state;
}
```

```
void C2::set_status(int state)
{
status = state;
}
// idle( ) is a friend of C1 and C2.
int idle(C1 a, C2 b)
{
   if(a.status || b.status) return 0;
   else return 1;
}

int main()
{
     C1 x; C2 y;
     x.set_status(IDLE);
     y.set_status(IDLE);

     if(idle(x, y)) cout<< "Screen Can Be Used.\n";
     else cout<< "Pop-up In Use.\n";

     x.set_status(INUSE);

     if(idle(x, y)) cout<< "Screen Can Be Used.\n";
     else cout<< "Pop-up In Use.\n";

     return 0;
}
```

The output produced by this program is shown here:

```
Screen Can Be Used.
Pop-up In Use.
```

Because idle( ) is a friend of both C1 and C2 it has access to the private status memberdefined by both classes. Thus, a single call to idle() can simultaneously check the status of an object of each class.

NOTE: A forward declaration declares a class type-name prior to the definition of the class.

Notice that this program uses a forward declaration (also called a forward reference) for the class C2. This is necessary because the declaration of idle( ) inside C1 refers to C2 before it is declared. To create a forward declaration to a class, simply use the form shown in this program.A friend of one class can be a member of another. For example, here is the preceding program rewritten so that idle( ) is a member of C1. Notice the use of the scope resolution operator when declaring idle( ) to be a friend of C2.

```
/* A function can be a member of one class and a
friend of another. */
#include <iostream.h>
constint IDLE=0;
constint INUSE=1;
class C2;    // forward declaration
class C1
{
     int status;  // IDLE if off, INUSE if on screen
public:
void set_status(int state);
     int idle(C2 b);  // now a member of C1
};
class C2
{
     int status;  // IDLE if off, INUSE if on screen
  public:
     void set_status(int state);
     friend int C1::idle(C2 b); // a friend, here
};
void C1::set_status(int state)
{
status = state;
}
void C2::set_status(int state)
{
status = state;
}
```

```
// idle() is member of C1, but friend of C2.
int C1::idle(C2 b)
{
   if(status || b.status) return 0;
   else return 1;
}
int main()
{
   C1 x; C2 y;
   x.set_status(IDLE);
   y.set_status(IDLE);
   if(x.idle(y)) cout<< "Screen Can Be Used.\n";
   else cout<< "Pop-up In Use.\n";

   x.set_status(INUSE);

   if(x.idle(y)) cout<< "Screen Can Be Used.\n";
   else cout<< "Pop-up In Use.\n";

   return 0;
}
```

Since idle( ) is a member of C1, it can access the status variable of objects of typeC1 directly. Thus, only objects of type C2 need be passed to idle( ).

CHAPTER FOUR

Granting Access

Granting Access

When a base class is inherited as private, all members of that class (public, protected, or private) become private members of the derived class. However, in certain circumstances, you may want to restore one or more inherited members to their original access specification. For example, you might want to grant certain public members of the base class public status in the derived class, even though the base class is inherited as private. You have two ways to accomplish this. First, you may use a using declaration within the derived class. This is the method recommended by Standard C++ for use in new code. However, a discussion of using is deferred until later in this book when namespaces are examined. (The primary reason for using is to provide support for namespaces.) The second way to adjust access to an inherited member is to employ an access declaration. Access declarations are still supported by Standard C++, but they have recently been deprecated, which means that they should not be used for new code. Since they arestill used in existing code, a discussion of access declarations is presented here. An access declaration takes this general form:

```
base-class::member;
```

The access declaration restores the access level of an inherited member to what it was in the base class.

The access declaration is put under the appropriate access heading in the derivedclass. Notice that no type declaration is required (or allowed) in an access declaration. To see how an access declaration works, let's begin with this short fragment:

```
class base
{
   public:
   int j;// public in base
```

```
};
// Inherit base as private.
class derived: private base
{
public:
// here is access declaration base::j;
// make j public again
// ...
};
```

Because base is inherited as private by derived, the public variable j is made a private variable of derived. However, the inclusion of this access declaration

```
base::j;
```

under derived's public heading restores j to its public status.You can use an access declaration to restore the access rights of public and protected members. However, you cannot use an access declaration to raise or lower a member's access status. For example, a member declared as private within a base class cannot be made public by a derived class. (Allowing this would destroy encapsulation!), The following program illustrates the use of access declarations:

```
#include <iostream.h>
class base
{
inti;    // private to base
public:
   int j, k;
   void seti(int x) { i = x; }
   intgeti() { return i; }
};
// Inherit base as private.
class derived: private base
{
public:
/* The next three statements override base's
inheritance as privateand restore j, seti()
```

```
and geti() to public access. */
base::j; // make j public again - but not k
base::seti; // make seti() public
base::geti; // make geti() public
//       base::i;
// illegal, you cannot elevate access
int a; // public
};
intmain()
{
   derived ob;
   //ob.i = 10;//illegal because i is private in derived
   ob.j = 20; // legal because j is made public in derived
   //ob.k = 30;   // illegal because k is private in derived
   ob.a = 40; // legal because a is public in derived
   ob.seti(10);
   cout<<ob.geti() << " " <<ob.j<< " " <<ob.a;
   return 0;
}
```

Notice how this program uses access declarations to restore j, seti(), and geti( ) to public status. The comments describe various other access restrictions.C++ provides the ability to adjust access to inherited members to accommodate those special situations in which most of an inherited class is intended to be made private, but a few members are to retain their public or protected status. It is best to use this feature sparingly.

CHAPTERFOUR

Inheritance

4.1 Introduction

Inheritance is one of the cornerstones of OOP because it allows the creation of hierarchical classifications. With inheritance, it is possible to create a general classthat defines traits common to a set of related items. This class may then be inherited by other, more specific classes, each adding only those things that are unique to the inheriting class.

In standard C++ terminology, a class that is inherited is referred to as a base class. The class that does the inheriting is called the derived class. Further, a derived class can be used as a base class for another derived class. In this way, a multilayered class hierarchy can be achieved.

4.2 Introducing Inheritance

C++ supports inheritance by allowing one class to incorporate another class into its declaration. Before discussing the theory and details, let's begin with an example of inheritance. The following class, called road_vehicle, very broadly defines vehicles that travel on the road. It stores the number of wheels a vehicle has and the number of passengers it can carry.

```
class road_vehicle
{
   int wheels;
   int passengers;
public:
   void set_wheels(intnum) { wheels = num; }
   intget_wheels() { return wheels; }
   void set_pass(intnum) { passengers = num; }
   intget_pass() { return passengers; }
};
```

You can use this broad definition of a road vehicle to help define specific types of vehicles. For example, the fragment shown here inherits road_vehicle to create a class called truck.

```
class truck : public road_vehicle
{
     int cargo;
  public:
     void set_cargo(int size) { cargo = size; }
     intget_cargo() { return cargo; }
     void show();
};
```

Because truck inherits road_vehicle, truck includes all of road_vehicle. It then adds cargo to it, along with the supporting member functions.Notice how road_vehicle is inherited. The general form for inheritance is shown here:

```
class  derived-class  : access base-class
{
     body of new class
}
```

Here, access is optional. However, if present, it must be either public, private, orprotected. You will learn more about these options later in this chapter. For now,all inherited classes will use public. Using public means that all the public members of the base class will also be public members of the derived class. Therefore, in the preceding example, members of truck have access to the public member functions of road_vehicle, just as if they had been declared inside truck. However, truck does not have access to the private members of road_vehicle.

For example, truck does not have access to wheels.Here is a program that uses inheritance to create two subclasses of road_vehicle. One is truck and the other is automobile.

```
// Demonstrate inheritance.
#include <iostream.h>
// Define a base class for vehicles.
class road_vehicle
{
     int wheels;
```

```
      int passengers;
   public:
      void set_wheels(intnum) { wheels = num; }
      intget_wheels() { return wheels; }
      void set_pass(intnum) { passengers = num; }
      intget_pass() { return passengers; }
};
// Define a truck.
class truck : public road_vehicle
{
      int cargo;
   public:
      void set_cargo(int size) { cargo = size; }
      intget_cargo() { return cargo; }
      void show();
};
enum type {car, van, wagon};
// Define an automoble.
class automobile : public road_vehicle
{
      enum type car_type;
   public:
      void set_type(type t) { car_type = t; }
      enum type get_type() { return car_type; }
      void show();
};
void truck::show()
{
   cout<< "wheels: " <<get_wheels() << "\n";
   cout<< "passengers: " <<get_pass() << "\n";
   cout<< "cargo capacity in cubic feet: "
   << cargo << "\n";
}

void automobile::show()
{
   cout<< "wheels: " <<get_wheels() << "\n";
   cout<< "passengers: " <<get_pass() << "\n";
   cout<< "type: ";
```

```
   switch(get_type())
   {
      case van: cout<< "van\n";break;
      case car: cout<< "car\n";break;
      case wagon: cout<< "wagon\n";
   }
}
intmain()
{
   truck t1, t2;
   automobile c;

   t1.set_wheels(18);
   t1.set_pass(2);
   t1.set_cargo(3200);

   t2.set_wheels(6);
   t2.set_pass(3);
   t2.set_cargo(1200);

   t1.show();
   cout<< "\n"; t2.show();
   cout<< "\n";

   c.set_wheels(4);
   c.set_pass(6);
   c.set_type(van);

   c.show();
   return 0;
}
```

The output from this program is shown here:

```
wheels: 18 passengers: 2
cargo capacity in cubic feet: 3200

wheels: 6 passengers: 3
cargo capacity in cubic feet: 1200

wheels: 4 passengers: 6 type: van
```

When a base class is inherited as public, its public members become public members of the derived class.As this program shows, the major advantage of inheritance is that it lets you create a base class that can be incorporated into more specific classes. In this way, each derived class can be precisely tailored to its own needs while still being part of a general classification.

One other point: Notice that both truck and automobile include a member function called show( ), which displays information about each object. This illustrates another aspect of polymorphism. Since each show( ) is linked with its own class, the compiler can easily tell which one to call for any given object.Now that you have seen the basic procedure by which one class inherits another, let's examine inheritance in detail.

Operator Overloading

3.1 Introduction

InC++, operators can be overloaded relative to class types that you define. The principal advantage to overloading operators is that it allows you to seamlesslyintegrate new data types into your programming environment.

Operator overloading allows you to define the meaning of an operator for a particular class. For example, a class that defines a linked list might use the + operator to add an object to the list. A class that implements a stack might use the + to push an object onto the stack. Another class might use the + operator in an entirely different way. When an operator is overloaded, none of its original meaning is lost. It is simply that a new operation, relative to a specific class, is defined. Therefore, overloading the + to handle a linked list, for example, does not cause its meaning relative to integers (i.e., addition) to change.

Operator overloading is closely related to function overloading. To overload an operator, you must define what the operation means relative to the class to which it is applied. To do this, you create an operator function, which defines the action of the operator. The general form of an operator function is

```
type classname::operator#(arg-list)
{
operation  relative to the class
}
```

Operators are overloaded using an operator function.Here, the operator that you are overloading is substituted for the #, and type is the type of value returned by the specified operation. Although it can be of any type you choose, the return value is often of the same type as the class for which the operator is being overloaded. This correlation facilitates the use of the overloaded operator in compound expressions.

The specific nature of arg-list is determined by several factors, as you will soon see.

Operator functions can be either members or nonmembers of a class. Nonmember operator functions are often friend functions of the class, however. Although similar, there are some differences between the way a member operator function is overloaded and the way a nonmember operator function is overloaded. Each approach is described here.

3.2 Operator Overloading Using Member Functions

To begin our examination of operator overloading using member functions, we will start with a simple example. The following program creates a class called three_d, which maintains the coordinates of an object in three-dimensional space. This program overloads the + and the = operators relative to the three_d class. Examine it closely:

```
    // Overload operators using member functions.
    #include <iostream.h>
    class three_d
    {
         int x, y, z; // 3-D coordinates
public:
          three_d() { x = y = z = 0; }
          three_d(inti, int j, int k)
          {x = i; y = j; z = k; }
    three_d operator+(three_d op2);
    three_d operator=(three_d op2);
    void show() ;
    };

    // Overload +.
    three_dthree_d::operator+(three_d op2)
    {
       three_d temp;
       temp.x = x + op2.x; // These are integer additions
       temp.y = y + op2.y; // and the + retains itsoriginal
```

```
   temp.z = z + op2.z; // meaning relative to them.
   return temp;
}

// Overload assignment.
three_dthree_d::operator=(three_d op2)
{
   x = op2.x; // These are integer assignments
   y = op2.y; // and the = retains its original
   z = op2.z; // meaning relative to them.
   return *this;
}
// Show X, Y, Z coordinates.
void three_d::show()
{
   cout<< x << ", ";
   cout<< y << ", ";
   cout<< z << "\n";
}
intmain()
{
   three_da(1, 2, 3), b(10, 10, 10), c;

   a.show();
   b.show();

   c = a + b; // add a and b together
   c.show();
   c = a + b + c; // add a, b and c together
   c.show();

   c = b = a; // demonstrate multiple assignment
   c.show();
   b.show();
return 0;
}
```

This program produces the following output:

```
1, 2, 3
10, 10, 10
11, 12, 13
22, 24, 26
1, 2, 3
1, 2, 3
```

As you examined the program, you may have been surprised to see that both operatorfunctions have only one parameter each, even though they overload binary operations. Thereason for this apparent contradiction is that when a binary operator is overloaded using a member function, only one argument is explicitly passed to it. The other argument is implicitly passed using the this pointer. Thus, in the line

```
temp.x = x + op2.x;
```

the x refers to this–>x, which is the x associated with the object that invokes the operator function. In all cases, it is the object on the left side of an operation that causes the call to the operator function. The object on the right side is passed to the function.

In general, when you use a member function, no parameters are used when overloading a unary operator, and only one parameter is required when overloading a binary operator. (You cannot overload the ternary ? operator.) In either case, the object that invokes the operator function is implicitly passed via the this pointer.

To understand how operator overloading works, let's examine the preceding program carefully, beginning with the overloaded operator +. When two objects of type three_d are operated on by the + operator, the magnitudes of their respective coordinates are added together, as shown in operator+( ). Notice, however, that this functiondoes not modify the value of either operand. Instead, an object of type three_d, which contains the result of the operation, is returned by the function. To understand whythe + operation does not change the contents of either object, think about the standard arithmetic + operation, as applied like this: 10 + 12. The outcome of this operation is22, but neither 10 nor 12 is changed by it. Although there is no rule that prevents an overloaded operator from altering the value of one of its operands, it is best for the

actions of an overloaded operator to be consistent with its original meaning.

Notice that operator+( ) returns an object of type three_d. Although the function could have returned any valid C++ type, the fact that it returns a three_d object allows the + operator to be used in compound expressions, such as a+b+c. Here, a+b generates a result that is of type three_d. This value can then be added to c. Had any other type of value been generated by a+b, such an expression would not work.

In contrast with the + operator, the assignment operator does, indeed, cause one of its arguments to be modified. (This is, after all, the very essence of assignment.) Since the operator=( ) function is called by the object that occurs on the left side of the assignment, it is this object that is modified by the assignment operation. Most often, the return value of an overloaded assignment operator is the object on the left, after the assignment has been made. (This is in keeping with the traditional action of the = operator.) For example, to allow statements like

```
a = b = c = d;
```

it is necessary for operator=( ) to return the object pointed to by this, which will be the object that occurs on the left side of the assignment statement. This allowsa string of assignments to be made. The assignment operation is one of the most important uses of the this pointer.

```
// This program uses friend operator++() functions.
#include <iostream.h>
class three_d
{
   int x, y, z; // 3-D coordinates
public:
   three_d() { x = y = z = 0; }
   three_d(inti, int j, int k)
   {x = i; y = j; z = k; }

   friend three_d operator+(three_d op1, three_d op2);
   three_d operator=(three_d op2);

   // use a reference to overload the ++
   friend three_d operator++(three_d&op1);
   friend three_d operator++(three_d&op1, intnotused);

   void show() ;
 } ;

// This is now a friend function.
three_d operator+(three_d op1, three_d op2)
{
   three_d temp;
   temp.x = op1.x + op2.x;
   temp.y = op1.y + op2.y;
   temp.z = op1.z + op2.z;
   return temp;
}
// Overload the =.
three_dthree_d::operator=(three_d op2)
{
   x = op2.x;
   y = op2.y;
   z = op2.z;
   return *this;
}
```

```
/* Overload prefix ++ using a friend function.
This requires the use of a reference parameter. */
three_d operator++(three_d&op1)
{
     op1.x++; op1.y++; op1.z++; return op1;
}

/* Overload postfix ++ using a friend function.
This requires the use of a reference parameter. */
three_d operator++(three_d&op1, intnotused)
{
   three_d temp = op1;
   op1.x++; op1.y++; op1.z++;
   return temp;
}
// Show X, Y, Z coordinates.
void three_d::show()
{
   cout<< x << ", ";
   cout<< y << ", ";
   cout<< z << "\n";
}
intmain()
{
   three_da(1, 2, 3), b(10, 10, 10), c;

   a.show();
   b.show();

   c = a + b; // add a and b together c.show();

   c = a + b + c; // add a, b and c together
   c.show();

   c = b = a; // demonstrate multiple assignment
   c.show();
   b.show();

   ++c; // prefix increment c.show();
```

```
    c++; // postfix increment c.show();

    a = ++c; // a receives c's value after increment
    a.show( ); // a and c
    c.show( ); // are the same
    a = c++; // a receives c's value prior to increment
    a.show( ); // a and c
    c.show( ); // now differ

    return 0;
}
```

3.8 Overloading the Relational and Logical Operators

Overloading a relational or logical operator, such as ==, <, or && is a straightforward process. However, there is one small distinction. As you know, an overloaded operator function usually returns an object of the class for which it is overloaded. However, an overloaded relational or logical operator typically returns a true or false value. This is in keeping with the normal usage of these operators, and allows them to be used in conditional expression.Here is an example that overloads the = = relative to the three_d class:

```
//overload ==.
bool three_d::operator==(three_d op2)
{
   if((x == op2.x) && (y == op2.y) && (z == op2.z))
   return true;
   else
   return false;
}
```

Once operator==( ) has been implemented, the following fragment is perfectly valid:

```
three_d a, b;
// ...
if(a == b) cout<< "a equals b\n";
```

```
else cout<< "a does not equal b\n";
```

Because == returns a bool result, its outcome can be used to control an if statement. As an exercise, try implementing several of the relational and logical operators relative to the three_d class.

Order Matters

Order Matters

When overloading binary operators, remember that in many cases, the orderof the operands does make a difference. For example, while A + B is commutative,A – B is not. (That is, A – B is not the same as B – A!) Therefore, when implementing overloaded versions of the non-commutative operators, you must remember which operand is on the left and which is on the right. For example, in this fragment, subtraction is overloaded relative to the three_d class:

```
// Overload subtraction.
three_dthree_d::operator-(three_d op2)
{
three_d temp;
   temp.x = x - op2.x;
   temp.y = y - op2.y;
   temp.z = z - op2.z;
   return temp;
}
```

Remember, it is the operand on the left that invokes the operator function.The operand on the right is passed explicitly. This is why x – op2.x is the proper order for the subtraction.For example, in the following program, a friend is used instead of a member function to overload the + operation:

```
// Overload + using a friend.
#include <iostream.h>
class three_d
{
   int x, y, z; // 3-D coordinates
public:
   three_d() { x = y = z = 0; }
   three_d(inti, int j, int k)
   { x = i; y = j; z = k;}

   friend three_d operator+(three_d op1, three_d op2);
   three_d operator=(three_d op2);
```

```
   void show() ;
} ;

// This is now a friend function.
three_d operator+(three_d op1, three_d op2)
{
   three_d temp;

   temp.x = op1.x + op2.x;
   temp.y = op1.y + op2.y;
   temp.z = op1.z+ op2.z;
   return temp;
}

// Overload assignment.
three_dthree_d::operator=(three_d op2)
{
   x = op2.x; y = op2.y; z = op2.z;
   return *this;
}
// Show X, Y, Z coordinates.
void three_d::show()
{
   cout<< x << ", ";
   cout<< y << ", ";
   cout<< z << "\n";
}

intmain()
{
   three_da(1, 2, 3), b(10, 10, 10), c;

   a.show();
   b.show();

   c = a + b; // add a and b together
   c.show();

   c = a + b + c; // add a, b and c together
   c.show();
```

```
    c = b = a; // demonstrate multiple assignment
    c.show();
    b.show();

    return 0;
}
```

As you can see by looking at operator+( ), now both operands are passed to it. The left operand is passed in op1, and the right operand in op2.In many cases, there is no benefit to using a friend function rather than a member function when overloading an operator. However, there is one situation in which a friend function is quite useful: when you want an object of a built-in type to occur on the left side of a binary operator. To understand why, consider the following.

As you know, a pointer to the object that invokes a member operator function is passed in this. In the case of a binary operator, it is the object on the left that invokes the function. This is fine, provided that the object on the left defines the specified operation. For example, assuming some object called Ob, which has integer addition defined for it, then the following is a perfectly valid expression:

```
Ob + 10; // will work
```

Because the object Ob is on the left side of the + operator, it invokes its overloaded operator function, which (presumably) is capable of adding an integer value to some element of Ob. However, this statement won't work:

```
10 + Ob; // won't work
```

The problem with this statement is that the object on the left of the + operator is an integer, a built-in type for which no operation involving an integer and an object of Ob's type is defined.

The solution to the preceding problem is to overload the + using two friend functions. In this case, the operator function is explicitly

passed both arguments, and it is invoked like any other overloaded function, based upon the types of its arguments. One version of the + operator function handles object + integer, and the other handles integer + object. Overloading the + (or any other binary operator) using friend functions allows a built-in type to occur on the left or right side of the operator. The following sample program shows you how to accomplish this:

```
#include <iostream.h>
class CL
{
public:
   int count;
   CL operator=(CL obj);
   friend CL operator+(CL ob, inti);
   friend CL operator+(inti, CL ob);
};


CL CL::operator=(CL obj)
{
   count = obj.count;
   return *this;
}
// This handles ob + int.
CL operator+(CL ob, inti)
{
   CL temp;

   temp.count = ob.count + i;
   return temp;
}
// This handles int + ob.
CL operator+(inti, CL ob)
{
   CL temp;

   temp.count = ob.count + i;
```

```
   return temp;
}
intmain()
{
   CL O;

   O.count = 10;
   cout<<O.count<< " "; // outputs 10

   O = 10 + O; // add object to integer
   cout<<O.count<< " "; // outputs 20

   O = O + 12; // add integer to object
   cout<<O.count; // outputs 32

   return 0;
}
```

As you can see, the operator+( ) function is overloaded twice, to accommodate the two ways in which an integer and an object of type CL can occur in the addition operation.

CHAPTERFOUR

Passing Parameters to Base Class Constructors

Passing Parameters to Base Class Constructors

So far, none of the preceding examples have included constructors requiring arguments. In cases where only the constructor of the derived class requires one or more arguments, you simply use the standard parameterized constructor syntax. But how do you pass arguments to a constructor in a base class? The answer is to use an expanded form of the derived class' constructor declaration, which passes arguments along to one or more base class constructors. The general form of this expanded declaration is shown here:

```
derived-constructor(arg-list) : base1(arg-list),
base2(arg-list), ... baseN(arg-list)
{
body of derived constructor
}
```

Here, base1 throughbaseN are the names of the base classes inherited by the derived class. Notice that a colon separates the constructor declaration of the derived class from the base classes, and that the base classes are separated from each other by commas, in the case of multiple base classes.Consider this sample program:

```
#include <iostream.h>
class base
{
protected:
inti;
public:
base(int x)
{ i = x; cout<< "Constructing base\n"; }
~base() { cout<< "Destructing base\n"; }
};
class derived: public base
{
     int j;
```

```
public:
          // derived uses x; y is passed along to base.
          derived(int x, int y): base(y)
          { j = x; cout<< "Constructing derived\n"; }
          ~derived() { cout<< "Destructing derived\n"; }
          void show() { cout<<i<< " " << j << "\n"; }
    };
    intmain()
    {
       derived ob(3, 4);
       ob.show(); // displays 4 3
       return 0;
    }
```

Here, derived's constructor is declared as taking two parameters, x and y. However,derived( ) uses only x; y is passed along to base( ). In general, the constructor of thederived class must declare the parameter(s) that its class requires, as well as any required by the base class. As the preceding example illustrates, any parameters required by the base class arepassed to it in the base class' argument list, specified after the colon.Here is a sample program that uses multiple base classes:

```
    #include <iostream.h>
    class base1
    {
       protected:
       inti;
       public:
       base1(intx){i = x;cout<<"Constructing base1\n";}
       ~base1() { cout<< "Destructing base1\n"; }
    };
    class base2
    {
    protected:
    int k;
    public:
       base2(intx){k = x;cout<<"Constructing base2\n";}
~base2() { cout<< "Destructing base2\n"; }
```

```
};
class derived: public base1, public base2
{
   int j;
   public:
   derived(int x, int y, int z): base1(y), base2(z)
   { j = x; cout<< "Constructing derived\n"; }
   ~derived() { cout<< "Destructing derived\n"; }
   void show()
   { cout<<i<< " " << j << " " << k << "\n"; }
};
intmain()
{
derived ob(3, 4, 5);
ob.show();   // displays 4 3 5
return 0;
}
```

It is important to understand that arguments to a base class constructor are passed via arguments to the derived class' constructor. Therefore, even if a derived class' constructor does not use any arguments, it still must declare one or more arguments if the base class takes one or more arguments. In this situation, the arguments passed to the derived class are simply passed along to the base. For example, in the following program, the constructor of derived takes no arguments, but base1( ) and base2( ) do:

```
#include <iostream.h>
class base1
{
protected:
     inti;
public:
base1(int x)
{ i=x; cout<< "Constructing base1\n"; }
~base1() { cout<< "Destructing base1\n"; }
};
```

```
class base2
{
   protected:
      int k;
   public:
      base2(int x)
      { k = x; cout<< "Constructing base2\n"; }
      ~base2() { cout<< "Destructing base2\n"; }
};
class derived: public base1, public base2
{
   public:
   /* Derived constructor uses no parameters,
   but still must be declared as taking them to pass
   them along to base classes.*/
   derived(int x, int y): base1(x), base2(y)
   { cout<< "Constructing derived\n"; }
   ~derived() { cout<< "Destructing derived\n"; }
   void show() { cout<<i<< " " << k << "\n"; }
};

intmain()
{
   derived ob(3, 4);
   ob.show(); // displays 3 4
   return 0;
}
```

The constructor of a derived class is free to use any and all parameters that it is declared as taking, whether or not one or more are passed along to a base class. Put differently, just because an argument is passed along to a base class does not preclude its use by the derived class as well. For example, this fragment is perfectly valid:

```
class derived: public base
{
   int j;
   public:
   // derived uses both x and y
```

```
    derived(int x, int y): base(x, y)
    { j = x*y; cout<< "Constructing derived\n"; }
    // ...
}
```

One final point to keep in mind when passing arguments to base class constructors: An argument being passed can consist of any expression valid at the time, including function calls and variables. This is in keeping with the fact that C++ allows dynamic initialization.

Using Member Functionsto Overload Unary Operators

Using Member Functionsto Overload Unary Operators

You may also overload unary operators, such as ++, – –, or the unary – or +. As stated earlier, when a unary operator is overloaded by means of a member function, no object is explicitly passed to the operator function. Instead, the operation is performed on the object that generates the call to the function through the implicitly passed this pointer.For example, here is an expanded version of the previous example program. This version defines the increment operation for objects of type three_d.

```
// Overload a unary operator.
#include <iostream.h>
class three_d
{
      int x, y, z; // 3-D coordinates
public:
      three_d() { x = y = z = 0; }
      three_d(inti, int j, int k)
      {x = i; y = j; z = k; }
      three_d operator+(three_d op2);
      three_d operator=(three_d op2);
      three_d operator++(); // prefix version of ++
      void show() ;
} ;
// Overload +.
three_dthree_d::operator+(three_d op2)
{
   three_d temp;
   temp.x = x + op2.x; // These are integer additions
   temp.y = y + op2.y;// and the + retains its original
   temp.z = z + op2.z; // meaning relative to them.
   return temp;
}
// Overload assignment.
three_dthree_d::operator=(three_d op2)
{
   x = op2.x; // These are integer assignments
```

```
   y = op2.y; // and the = retains its original
   z = op2.z; // meaning relative to them.
   return *this;
}
// Overload the prefix version of ++.
three_dthree_d::operator++()
{
   x++; // increment x, y, and z
   y++;
   z++;
   return *this;
}
// Show X, Y, Z coordinates.
void three_d::show()
{
   cout<< x << ", ";
   cout<< y << ", ";
   cout<< z << "\n";
}

intmain()
{
   three_da(1, 2, 3), b(10, 10, 10), c;
   a.show();
   b.show();

   c = a + b; // add a and b together
   c.show();

   c = a + b + c; // add a, b and c together
   c.show();

   c = b = a; // demonstrate multiple assignment
   c.show();
   b.show();
   ++c;  // increment c
   c.show();

   return 0;
}
```

The output from the program is shown here.

```
1, 2, 3
10, 10, 10
11, 12, 13
22, 24, 26
1, 2, 3
1, 2, 3
2, 3, 4
```

As the last line of the output shows, operator++( ) increments each coordinate in the object and returns the modified object. Again, this is in keeping with the traditional meaning of the ++ operator.As you know, the ++ and – – have both a prefix and a postfix form. For example, both++O;andO++;are valid uses of the increment operator. As the comments in the preceding program state, the operator++( ) function defines the prefix form of ++ relative to the three_d class. However, it is possible to overload the postfix form as well. The prototype for the postfix form of the ++ operator, relative to the three_d class, is shown here:

```
three_dthree_d::operator++(intnotused);
```

The increment and decrement operators have both a prefix and postfix form. The parameternotused is not used by the function, and should be ignored. This parameter is simply a way for the compiler to distinguish between the prefix and postfix forms of the increment operator. (The postfix decrement uses the same approach.)Here is one way to implement a postfix version of ++ relative to the three_d class:

```
// Overload the postfix version of ++.
three_dthree_d::operator++(intnotused)
{
   three_d temp = *this; // save original value

   x++;  // increment x, y, and z
   y++;
   z++;
   return temp; // return original value
}
```

Notice that this function saves the current state of the operand by using the statement

```
three_d temp = *this;
```

and then returns temp. Keep in mind that the traditional meaning of a postfix increment is to first obtain the value of the operand, and then to increment the operand. Therefore,it is necessary to save the current state of the operand and return its original value, before it is incremented, rather than its modified value.

The following version of the original program implements both forms of the ++operator:

```
// Demonstrate prefix and postfix ++.
#include <iostream.h>
class three_d
{
     int x, y, z; // 3-D coordinates
public:
      three_d() { x = y = z = 0; }
      three_d(inti, int j, int k)
      {x = i; y = j; z = k; }

      three_d operator+(three_d op2);
      three_d operator=(three_d op2);
      three_d operator++(); // prefix version of ++
      three_d operator++(intnotused);
      // postfix version of ++
      void show() ;
};

// Overload +.
three_dthree_d::operator+(three_d op2)
{
   three_d temp;

   temp.x = x + op2.x; // These are integer additions
   temp.y = y + op2.y; // and the + retains itsoriginal
```

```
      temp.z = z + op2.z; // meaning relative to them.
      return temp;
   }
   // Overload assignment.
   three_dthree_d::operator=(three_d op2)
   {
      x = op2.x; // These are integer assignments
      y = op2.y; // and the = retains its original
      z = op2.z; // meaning relative to them.
      return *this;
   }

   // Overload the prefix version of ++.
   three_dthree_d::operator++()
   {
      x++;  // increment x, y, and z
      y++;
      z++;
      return *this; // return altered value
   }
   // Overload the postfix version of ++
three_dthree_d::operator++(intnotused)
   {
      three_d temp = *this; // save original value

      x++; // increment x, y, and z
      y++;
      z++;
      return temp; // return original value
   }
   // Show X, Y, Z coordinates.
   void three_d::show( )
   {
      cout<< x << ", ";
      cout<< y << ", ";
      cout<< z << "\n";
   }
   intmain()
   {
   three_da(1, 2, 3), b(10, 10, 10), c;
```

```
a.show();
b.show();

c = a + b; // add a and b together c.show();
c = a + b + c; // add a, b and c together c.show();

c = b = a; // demonstrate multiple assignment
c.show();
b.show();

++c; // prefix increment c.show();
c++; // postfix increment c.show();

a = ++c; // a receives c's value after increment
a.show(); // a and c
c.show(); // are the same 13
a = c++; // a receives c's value prior to increment
a.show(); // a and c c.show(); // now differ

return 0;
}
```

The output is shown here.

```
1, 2, 3
```

As the last four lines show, the prefix increment increases the value of c before its value is assigned to a, and the postfix increment increases c after its value is assigned to a.

Remember that if the ++ precedes its operand, the operator++( ) is called. If it follows its operand, the operator++(intnotused) function is called. This same approach is also used to overload the prefix and postfix decrement operator relative to any class. Youmight want to try defining the decrement operator relative to three_d as an exercise.

Operator Overloading Tips and Restrictions

The action of an overloaded operator, as applied to the class for which it is defined, need not bear any relationship to that operator's

default usage, as applied to C++'s built-in types. For example, the << and >> operators, as applied to cout and cin, have little in common with the same operators applied to integer types. However, to maintain the transparency and readability of your code, an overloaded operator should reflect, when possible, the spirit of the operator's original use. For example, the + relative to three_dis conceptually similar to the + relative to integer types. There would be little benefit in defining the + operator relative to some class in such a way that it acts more the way you would expect the || operator, for instance, to perform. The central concept here is that, while you can give an overloaded operator any meaning you like, for clarity, it is best when its new meaning is related to its original meaning.

There are some restrictions to overloading operators. First, you cannot alter the precedence of any operator. Second, you cannot alter the number of operands required by the operator, although your operator function could choose to ignore an operand. Finally, except for the function call operator (discussed later), operator functions cannot have default arguments.The only operators that you cannot overload are shown here: `(.  ::  .*  ?`). Nonmember binary operator functions have two parameters. Nonmember unary operator functions have one parameter.

Nonmember Operator Functions

You can overload an operator for a class by using a nonmember function, which is often a friend of the class. As you learned earlier, nonmember functions, including friend functions, do not have a this pointer. Therefore, when a friend is used to overload an operator, both operands are passed explicitly when a binary operator is overloaded, and a single operand is passed when a unary operator is overloaded. The only operators that cannot be overloaded using nonmember functions are =, ( ), [ ], and –>.

overload ++ operator

Using a Friend to Overload a Unary Operator

You can also overload a unary operator by using a friend function. However,doing so requires a little extra effort. To begin, think back to the original version of the overloaded ++ operator relative to the three_d class that was implemented as a member function. It is shown here for your convenience:

```
// Overload the prefix form of ++.
three_dthree_d::operator++()
{
   x++; y++; z++;
   return *this;
}
```

As you know, every member function receives as an implicit argument this, which is a pointer to the object that invokes the function. When a unary operator is overloaded by use of a member function, no argument is explicitly declared. The only argument needed in this situation is the implicit pointer to the invoking object. Any changes made to the object's data will affect the object on which the operator function is called. Therefore, in the preceding function, x++ increments the x member of the invoking object.

Unlike member functions, a nonmember function, including a friend, does not receive a this pointer, and therefore cannot access the object on which it was called. Instead, a friend operator function is passed its operand explicitly. For this reason, trying to create a friend operator++( ) function, as shown here, will not work:

```
// THIS WILL NOT WORK
three_d operator++(three_d op1)
{
     op1.x++; op1.y++; op1.z++; return op1;
}
```

This function will not work because only a copy of the object that activated the call to operator++( ) is passed to the function in parameter op1. Thus, the changes inside operator++( ) will not affect the calling object, only the local parameter.

If you want to use a friend function to overload the increment or decrement operators, you must pass the object to the function as a reference parameter. Since a reference parameter is an implicit pointer to the argument, changes to the parameter will affect the argument. Using a reference parameter allows the function to increment or decrement the object used as an operand.

When a friend is used for overloading the increment or decrement operators, the prefix form takes one parameter (which is the operand). The postfix form takes two parameters. The second is an integer, which is not used.Here is the entire three_d program, which uses a friend operator++( ) function. Notice that both the prefix and postfix forms are overloaded.

```
// This program uses friend operator++() functions.
#include <iostream.h>
class three_d
{
   int x, y, z; // 3-D coordinates
public:
   three_d() { x = y = z = 0; }
   three_d(inti, int j, int k)
   {x = i; y = j; z = k; }

   friend three_d operator+(three_d op1, three_d op2);
   three_d operator=(three_d op2);

   // use a reference to overload the ++
   friend three_d operator++(three_d&op1);
   friend three_d operator++(three_d&op1, intnotused);

   void show() ;
 } ;
```

```
// This is now a friend function.
three_d operator+(three_d op1, three_d op2)
{
   three_d temp;
   temp.x = op1.x + op2.x;
   temp.y = op1.y + op2.y;
   temp.z = op1.z + op2.z;
   return temp;
}
// Overload the =.
three_dthree_d::operator=(three_d op2)
{
   x = op2.x;
   y = op2.y;
   z = op2.z;
   return *this;
}

/* Overload prefix ++ using a friend function.
This requires the use of a reference parameter. */
three_d operator++(three_d&op1)
{
     op1.x++; op1.y++; op1.z++; return op1;
}

/* Overload postfix ++ using a friend function.
This requires the use of a reference parameter. */
three_d operator++(three_d&op1, intnotused)
{
   three_d temp = op1;
   op1.x++; op1.y++; op1.z++;
   return temp;
}
// Show X, Y, Z coordinates.
void three_d::show()
{
   cout<< x << ", ";
   cout<< y << ", ";
   cout<< z << "\n";
```

```
}
intmain()
{
   three_da(1, 2, 3), b(10, 10, 10), c;

   a.show();
   b.show();

   c = a + b; // add a and b together c.show();

   c = a + b + c; // add a, b and c together
   c.show();

   c = b = a; // demonstrate multiple assignment
   c.show();
   b.show();

   ++c; // prefix increment c.show();

   c++; // postfix increment c.show();

   a = ++c; // a receives c's value after increment
   a.show( ); // a and c
   c.show( ); // are the same
   a = c++; // a receives c's value prior to increment
   a.show( ); // a and c
   c.show( ); // now differ

   return 0;
}
```

Overloading the Relational and Logical Operators

Overloading a relational or logical operator, such as ==, <, or && is a straightforward process. However, there is one small distinction. As you know, an overloaded operator function usually returns an object of the class for which it is overloaded. However, an overloaded relational or logical operator typically returns a true or false value. This is in keeping with the normal usage of these operators, and allows them to be

used in conditional expression.Here is an example that overloads the = = relative to the three_d class:

```
//overload ==.
bool three_d::operator==(three_d op2)
{
   if((x == op2.x) && (y == op2.y) && (z == op2.z))
   return true;
   else
   return false;
}
```

Once operator==( ) has been implemented, the following fragment is perfectly valid:

```
three_d a, b;
// ...
if(a == b) cout<< "a equals b\n";
else cout<< "a does not equal b\n";
```

Because == returns a bool result, its outcome can be used to control an if statement. As an exercise, try implementing several of the relational and logical operators relative to the three_d class.

CHAPTER FOUR

Using protected for Inheritance of a Base Class

Using protected for Inheritance of a Base Class

In addition to specifying protected status for members of a class, the keyword protected can also be used to inherit a base class. When a base class is inherited as protected, all public and protected members of the base class become protected members of the derived class. Here is an example:

```
// Demonstrate inheriting a protected base class.
#include <iostream.h>
class base
{
   inti;
   protected:
   int j;
   public:
   int k;
      void seti(int a) { i = a; }
      intgeti() { return i; }
};
// Inherit base as protected.
class derived : protected base
{
public:
     void setj(int a) { j = a; } // j is protected here
     void setk(int a) { k = a; } // k is also protected
     intgetj() { return j; }
intgetk() { return k; }
};
intmain()
{
derived ob;
/* This next line is illegal because seti() is
a protected member of derived, which makes it
        inaccessible outside of derived. */
// ob.seti(10);
```

```
//cout<<ob.geti();illegal -- geti() is protected
//ob.k = 10;  also illegal because k is protected

// these next statements are OK
   ob.setk(10);
   cout<<ob.getk() << ' ';
   ob.setj(12);
   cout<<ob.getj() << ' ';

   return 0;
}
```

As you can see by reading the comments in this program, k, j, seti(), and geti( )in base become protected members of derived. This means that they cannotbe accessed by code outside of derived. Thus, inside main( ), references to these members through ob are illegal.

Reviewing public, protected, and private

Because the access rights as defined by public, protected, and private are fundamental to C++ programming, let's review their meanings.

When a class member is declared as public, it can be accessed by any other part of a program. When a member is declared as private, it can be accessed only by members of its class. Further, derived classes do not have access to private base class members. When a member is declared as protected, it can be accessed only by members of its class, or by derived classes. Thus, protected allows a member to be inherited, but to remain private within a class hierarchy.

When a base class is inherited by use of public, its public members become public members of the derived class, and its protected members become protected members of the derived class.

When a base class is inherited by use of protected, its public and protected members become protected members of the derived class.

When a base class is inherited by use of private, its public and protected members become private members of the derived class.

In all cases, private members of a base class remain private to the base class, and are not inherited.As you become more familiar with C++, the meaning of public, protected, and private will become second nature. For now, if you are unsure what precise effect an access specifier has, write a short sample program as an experiment and observe the results.

Inheriting Multiple Base Classes

It is possible for a derived class to inherit two or more base classes. For example, in this short program, derived inherits both base1 and base2:

```
// An example of multiple base classes.
#include <iostream.h>
class base1
{
   protected:
   int x;
   public:
   void showx() { cout<< x << "\n"; }
};
class base2
{
   protected:
   int y;
   public:
   void showy() { cout<< y << "\n"; }
};
// Inherit multiple base classes.
class derived: public base1, public base2
{
public:
     void set(inti, int j) { x = i; y = j; }
};
```

```
intmain()
{
   derived ob;

   ob.set(10, 20); // provided by derived
   ob.showx();    // from base1
   ob.showy();    // from base2

   return 0;
}
```

As this example illustrates, to cause more than one base class to be inherited, you must use a comma-separated list. Further, be sure to use an access specifier for each base class inherited.