Chapter 11: More About Classes and Object-Oriented Programming

Starting Out with C++
Early Objects
Global Edition

by Tony Gaddis, Judy Walters, and Godfrey Muganda

Addison-Wesley is an imprint of PEARSON

Topics

Topics (continued)

11.9 Type Conversion Operators

11.10 Convert Constructors

11.11 Aggregation and Composition

11.12 Inheritance

11.13 Protected Members and Class Access

11.14 Constructors, Destructors, and Inheritance

11.15 Overriding Base Class Functions

11.1 The **`this`** Pointer and Constant Member Functions

- **`this`** pointer:
  - Implicit parameter passed to a member function
  - points to the object calling the function
- **`const`** member function:
  - does not modify its calling object

Using the **`this`** Pointer

Can be used to access members that may be hidden by parameters with the same name:

```
class SomeClass
{
  private:
    int num;
  public:
    void setNum(int num)
    { this->num = num; }
};
```

P739 Program 11-1 eP697

Constant Member Functions

- A parameter that is passed to a function by reference or through a pointer may be modified by that function.
  - The **`const`** keyword is used with a parameter to prevent the called function from modifying it.
- When **`const`** appears in the parameter list,

```
int setNum (const int num)
```

the parameter is read-only.

- When **`const`** follows the parameter list,

```
int getX()const
```

the function is prevented from modifying the object.

P741 example eP699-700

11.2 Static Members

- Static member variable:
  - One instance of variable for the entire class
  - Shared by all objects of the class
- Static member function:
  - Can be used to access static member variables
  - Can be called before any class objects are created

Static Member Variables

1) Must be declared in class with keyword **`static`**:

```
class IntVal
{
 public:
     IntVal(int val = 0)
     { value = val; valCount++ }
     int getVal();
     void setVal(int);
  private:
     int value;
     static int valCount;
 };
```

Static Member Variables

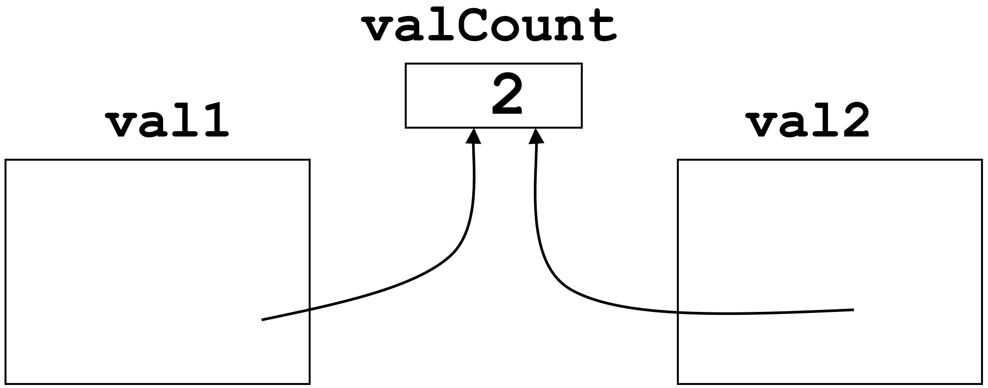
2) Must be defined outside of the class:

```
class IntVal
{
    //In-class declaration
    static int valCount;
    //Other members not shown
};
//Definition outside of class
int IntVal::valCount = 0;
```

Static Member Variables

3) Can be accessed or modified by any object of the class: Modifications by one object are visible to all objects of the class:

```
IntVal val1, val2;
```



P 744 Program 11-2 eP702

Static Member Functions

1)Declared with **`static`** before return type:

```
class IntVal
{ public:
      static int getValCount()
      { return valCount; }
    private:
      int value;
      static int valCount;
};
```

Static Member Functions

2) Can be called independently of class objects, through the class name:

```
cout << IntVal::getValCount();
```

3) Because of item 2 above, the **`this`** pointer cannot be used

4) Can be called before any objects of the class have been created

5) Used primarily to manipulate static member variables of the class

P 746 Program 11-3 eP704

11.3 Friends of Classes

- Friend function: a function that is not a member of a class, but has access to private members of the class
- A friend function can be a stand-alone function or a member function of another class
- It is declared a friend of a class with the **`friend`** keyword in the function prototype

Friend Function Declarations

1) Friend function may be a stand-alone function:

```
class aClass
{
  private:
    int x;
    friend void fSet(aClass &c, int a);
};

void fSet(aClass &c, int a)
{
   c.x = a;
}
```

Friend Function Declarations

2) Friend function may be a member of another class:

```
class aClass
{ private:
    int x;
    friend void OtherClass::fSet
                          (aClass &c, int a);
};
class OtherClass
{ public:
     void fSet(aClass &c, int a)
     { c.x = a; }
};
```

Friend Class Declaration

3) An entire class can be declared a friend of a class:

```
class aClass
{private:
   int x;
   friend class frClass;
};

class frClass
{public:
   void fSet(aClass &c,int a){c.x = a;}
   int fGet(aClass c){return c.x;}
};
```

Friend Class Declaration

- If **`frClass`** is a friend of **`aClass`**, then all member functions of **`frClass`** have unrestricted access to all members of **`aClass`**, including the private members.
- In general, restrict the property of Friendship to only those functions that must have access to the private members of a class.

P750 Program 11-4 eP708

11.4 Memberwise Assignment

- Can use **=** to assign one object to another, or to initialize an object with an object's data (field-to-field copy)
- Examples (assuming class **V**):

```
V v1, v2;
… // statements that assign
… // values to members of v1
v2 = v1;     // assignment
V v3 = v2;   // initialization
```

P754 Program 11-5 eP712

11.5 Copy Constructors

- Special constructor used when a newly created object is initialized to the data of another object of same class
- Default copy constructor copies field-to-field, using memberwise assignment
- The default copy constructor works fine in most cases

P756 Program 11-6 eP714

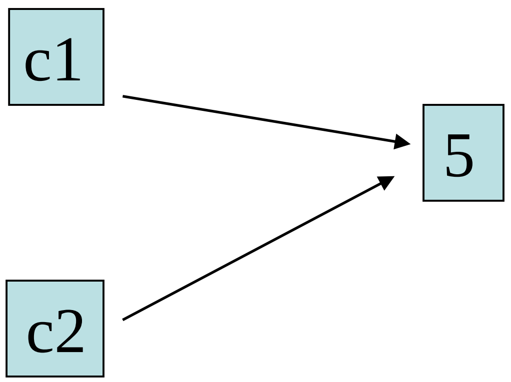
Copy Constructors

Problems occur when objects contain pointers to dynamic storage:

```
class CpClass
{
 private:
  int *p;
 public:
  CpClass(int v=0)
     { p = new int; *p = v;}
  ~CpClass(){delete p;}
};
```

Default Constructor Causes Sharing of Storage

```
CpClass c1(5);
if (true)
{
   CpClass c2=c1;
}
// c1 is corrupted
// when c2 goes
// out of scope and
// its destructor
// executes
```



Problems of Sharing Dynamic Storage

- Destructor of one object deletes memory still in use by other objects

- Modification of memory by one object affects other objects sharing that memory

P758 Program 11-7 eP716

Programmer-Defined Copy Constructors

- A copy constructor is one that takes a reference parameter to another object of the same class
- The copy constructor uses the data in the object passed as parameter to initialize the object being created
- Reference parameter should be **`const`** to avoid potential for data corruption

Programmer-Defined Copy Constructors

- The copy constructor avoids problems caused by memory sharing
- Can allocate separate memory to hold new object's dynamic member data
- Can make new object's pointer point to this memory
- Copies the data, not the pointer, from the original object to the new object

Copy Constructor Example

```
class CpClass
{
    int *p;
  public:
    CpClass(const CpClass &obj)
    { p = new int; *p = *obj.p; }
    CpClass(int v=0)
    { p = new int; *p = v; }
    ~CpClass(){delete p;}
};
```

P 760 Program 11-8   eP718

Copy Constructor – When Is It Used?

A copy constructor is called when

- An object is initialized from an object of the same class
- An object is passed by value to a function
- An object is returned using a `return` statement from a function

P763 example eP721

11.6 Operator Overloading

- Operators such as **=**, **+**, and others can be redefined for use with objects of a class

- The name of the function for the overloaded operator is **`operator`** followed by the operator symbol, *e.g.*,

  `operator+` is the overloaded **+** operator and

  `operator=` is the overloaded **=** operator

Operator Overloading

- Operators can be overloaded as
  - instance member functions, or as
  - friend functions
- The overloaded operator must have the same number of parameters as the standard version. For example, **`operator=`** must have two parameters, since the standard = operator takes two parameters.

Overloading Operators as Instance Members

A binary operator that is overloaded as an instance member needs only one parameter, which represents the operand on the right:

```
class OpClass
{
 private:
    int x;
 public:
    OpClass operator+(OpClass right);
};
```

Overloading Operators as Instance Members

- The left operand of the overloaded binary operator is the calling object
- The implicit left parameter is accessed through the **`this`** pointer

```
OpClass OpClass::operator+(OpClass r)
{  OpClass sum;
   sum.x = this->x + r.x;
   return sum;
}
```

Invoking an Overloaded Operator

- Operator can be invoked as a member function:

```
OpClass a, b, s;
s = a.operator+(b);
```

- It can also be invoked in the more conventional manner:

```
OpClass a, b, s;
s = a + b;
```

Overloading Assignment

- Overloading the assignment operator solves problems with object assignment when an object contains pointer to dynamic memory.
- Assignment operator is most naturally overloaded as an instance member function
- It needs to return a value of the assigned object to allow cascaded assignments such as

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

Overloading Assignment

Assignment overloaded as a member function:

```
class CpClass
{
      int *p;
   public:
      CpClass(int v=0)
      { p = new int; *p = v;
      ~CpClass(){delete p;}
      CpClass operator=(CpClass);
};
```

Overloading Assignment

Implementation returns a value:

```
CpClass CpClass::operator=(CpClass r)
{
  *p = *r.p;
  return *this;
};
```

Invoking the assignment operator:

```
CpClass a, x(45);
a.operator=(x); // either of these
a = x;          // lines can be used
```

Operator Overloading

- P 767 Program 11-9 eP724
- Multiple assignment a=b=c ?
- P769 +P770 example eP727

Notes on Overloaded Operators

- Overloading can change the entire meaning of an operator
- Most operators can be overloaded

Table 11-1 Operators That Can Be Overloaded

+	-	*	/	%	^	&	|	~	!	=	<
>	+=	-=	*=	/=	%=	^=	&=	|=	<<	>>	>>=
<<=	==	!=	<=	>=	&&	||	++	--	->*	,	->
[ ]	( )	new	delete								

- Cannot change the number of operands of the operator
- Cannot overload the following operators:

`?:   .   .*   ::  sizeof`

Overloading Types of Operators

- Overloaded relational operators should return a **`bool`** value

  P773 Program11-10 eP731

- **`++`, `--`** operators overloaded differently for prefix vs. postfix notation

  P776 +777 eP734+735

Overloading Types of Operators

- Overloaded stream operators **`>>`**, **`<<`** must return **`istream`**, **`ostream`** objects and take **`istream`**, **`ostream`** objects as parameters

P778~781 example eP735~739

11.7 Rvalue References and Move Operations

Introduced in C++ 11:

- An rvalue is a temporary value that is unnamed.

ex: `double x; x = pow(3.0, 2);`

`pow(3.0, 2)` is an rvalue. It is used for the assignment statement, then the memory is deallocated and is inaccessible.

Rvalue Reference

Introduced in C++ 11:

- An rvalue reference is a reference variable that refers to an rvalue. It is declared with the && operator:

```
double && powRef = pow(3.0, 2);
cout << powRef << endl;
```

- Declaring the rvalue reference assigns a name to the temporary location holding the value of the expression, making it no longer an rvalue

Move Assignment, Move Constructor

Introduced in C++ 11:

•Copy assignments and copy constructors are used when objects contain dynamic memory. Deallocating memory in the target object, then allocating memory for the copy, then destroying the temporary object, is resource-intensive.

P 790~791

•Move assignment and move constructors, which use rvalue references, are much more efficient.

P 793

Move Assignment/Constructor Details

- Move assignment (overloaded = operator) and move constructor use an rvalue reference for the parameter
- The dynamic memory locations can be "taken" from the parameter and assigned to the members in the object invoking the assignment.
- Set dynamic fields in the parameter to **`nullptr`** before the function ends and the parameter's destructor executes.

Moving is Not New

- Though introduced in C++ 11, move operations have already been used by the compiler:
  - when a non-void function returns a value
  - when the right side of an assignment statement is an rvalue
  - on object intialization from a temporary object

Default Class Operations

- Managing the details of a class implementation is tedious and potentially error-prone.
- The C++ compiler can generate a default constructor, copy constructor, copy assignment operator, move constructor, and destructor.
- If you provide your own implementation of any of these functions, you should provide your own implementation for all of them.

11.8 Function Objects and Lambda Expressions

- The function operator, `()`, can be overloaded.
- An object of a class that overloads the function operator is a function object, or a functor.

Function Object Example

A class that overloads **()**

```
class Multiply
{
  public:
      int operator() (int x, int y)
          {
              return x * y;
          }
} // end class Multiply
```

Function Object Example

Create and use objects:

```
Multiply prod;
int product = prod(4,3);
```

You can create and use in the same step:

```
int product = Multiply()(4,3);
```

The object created in the previous example is anonymous.

Predicates

- Predicate: A function that returns a Boolean value
- Function objects can be created to perform tasks associated with predicates.
- Unary Predicate: takes one argument
- Binary Predicate: takes two arguments

Unary Predicate

- Unary Predicate: A predicate that takes a single argument

```
class IsPositive
{
  public:
      bool operator() (int x)
            {
                return x > 0;
            }
} // end class IsPositive
```

Binary Predicate

- Binary Predicate: A predicate that takes two arguments

```
class GreaterThan
{
  public:
      bool operator() (int x, int y)
          {
              return x > y;
          }
} // end class GreaterThan
```

Passing Function Objects to Functions

- Function objects can be passed as parameters to functions.

- Many C++ library functions have function objects as parameters

Passing Function Objects to Functions

- Examples:
  - **sort()**, used for sorting an array or vector. The function parameter is used to determine how to compare two values in the array and determine how to order them.

    void sort(begin, end, compare)

    example:

```
int arr[ ] {12, 89, 34, 15, 11};
sort(arr, arr+5, LessThan()};
```

```
Class LessThan
{
  Public:
  Bool operator()(int a, int b)
  {
     return a<b;
  }
}
```

Passing Function Objects to Functions

- Examples:
  - **`remove_if()`,** used for removing elements from an array or vector that meet a criteria. The function parameter is used to define the criteria.
    void remove_if(begin, end, unary_predicate)
    example:
    vector<int> vec {12, 25, 36, 8, 11, 15, 89, 32, 71};
    auto rem_start=remove_if(begin(vec), end(vec), IsEven()};
    {25, 11, 15, 89, 71, 15, 89, 32, 71}
    vec.erase(rem_start, end(vec));

Passing a Parameter to a Function Object's Constructor

- The constructor of a class for a function object can take a parameter.
- The argument to the parameter can be stored in a member variable.
- This can make the class more general and allow function objects to be created with some flexibility.

Example of Passing a Parameter to a Function Object's Constructor

```
class HasAFactor
{
        int factor;
   public:
        HasAFactor(int f) { factor = f; }
        bool operator() (int x)
                {
                    return x % factor == 0;
                }
} // end class HasAFactor
```

HasAFactor(f) : a function object

HasAFactor(f)(x) : is the Boolean result of calling the function object with argument x

Unary Functions

- A unary function takes a single parameter and has a `void` return type.

```
class Display
{
  public:
      void operator() (int x)
          {
              cout << x << " ";
          }
} // end class Display
```

P 801 Program 11-17

Predefined Functional Classes

- Requires the **<functional>** header file
- Some examples: **P 802 Program 11-18**

Function Object	Description
less<T>	less<T>()(T a, T b) is true if and only if a < b
less_equal<T>	less_equal()(T a, T b) is true if an only if a <= b
greater<T>	greater<T>()(T a, T b) is true if and only if a > b
greater_equal<T>	greater_equal<T>()(T a, T b) is true if and only if a >= b

Lambda Expression

- A lambda expression provides a simplified way to create a function object.
- Examples:

```
// Multiply
[] (int x, int y){ return x * y; }
// IsPositive
[] (int x){ return x > 0; }
```

- Assigning a name to a lambda expression: P804

```
auto isPositive = [](auto x)
                      {return x>0;};
```

11.9 Type Conversion Operators

- Conversion Operators are member functions that tell the compiler how to convert an object of the class type to a value of another type
- The conversion information provided by the conversion operators is automatically used by the compiler in assignments, initializations, and parameter passing

Syntax of Conversion Operators

- Conversion operator must be a member function of the class you are converting from
- The name of the operator is the name of the type you are converting to
- The operator does not specify a return type

Conversion Operator Example

- To convert from a class **`IntVal`** to an integer:

```
class IntVal
{
    int x;
  public:
    IntVal(int a = 0){x = a;}
    operator int(){return x;}
};
```

- Automatic conversion during assignment:

```
IntVal obj(15); int i;
i = obj;  cout << i; // prints 15
```

P805~806 ePP747~748

11.10 Convert Constructors

Convert constructors provide a way for the compiler to convert a value of a given type to an object of the class.

```
class CCClass
{  int x;
  public:
    CCClass();              //default
    CCClass(int a, int b);
    CCClass(int a);         //convert
    CCClass(string s);      //convert
};
```

Example of a Convert Constructor

The C++ **`string`** class has a convert constructor that converts from C-strings:

```
class string
{
  public:
    string(char *);  //convert
    …
};
```

Uses of Convert Constructors

- They are automatically invoked by the compiler to create an object from the value passed as parameter:

```
string s("hello");   //convert C-string
CCClass obj(24);     //convert int
```

- The compiler allows convert constructors to be invoked with assignment-like notation:

```
string s = "hello"; //convert C-string
CCClass obj = 24;    //convert int
```

Uses of Convert Constructors

- Convert constructors allow functions that take the class type as parameter to take parameters of other types:

```
void myFun(string s); // needs string
                      // object
myFun("hello");       // accepts C-string

void myFun(CCClass c);
myFun(34);            // accepts int
```

P808~809, ePP750~751

11.11 Aggregation and Composition

- Class aggregation: An object of one class owns an object of another class
- Class composition: A form of aggregation where the enclosing class controls the lifetime of the objects of the enclosed class
- Supports the modeling of 'has-a' relationship between classes – enclosing class 'has a(n)' instance of the enclosed class

Object Composition

```
class StudentInfo
{
   private:
      string firstName, LastName;
      string address, city, state, zip;
    ...
};
class Student
{
   private:
      StudentInfo personalData;
    ...
};
```

Member Initialization Lists

- Used in constructors for classes involved in aggregation.
- Allows constructor for enclosing class to pass arguments to the constructor of the enclosed class
- Notation:

```
owner_class(parameters):owned_class(parameters);
```

Member Initialization Lists

Use:

```
class StudentInfo
{
      ...
};
class Student
{
   private:
     StudentInfo personalData;
   public:
    Student(string fname, string lname):
    StudentInfo(fname, lname);
};
```

Member Initialization Lists

- Member Initialization lists can be used to simplify the coding of constructors
- Should keep the entries in the initialization list in the same order as they are declared in the class

P811-812 eP753-754

Aggregation Through Pointers

- A ‘has-a’ relationship can be implemented by owning a pointer to an object
- Can be used when multiple objects of a class may ‘have’ the same attribute for a member
  - ex: students who may have the same city/state/zipcode
- Using pointers minimizes data duplication and saves space

P812~813 eP754~755

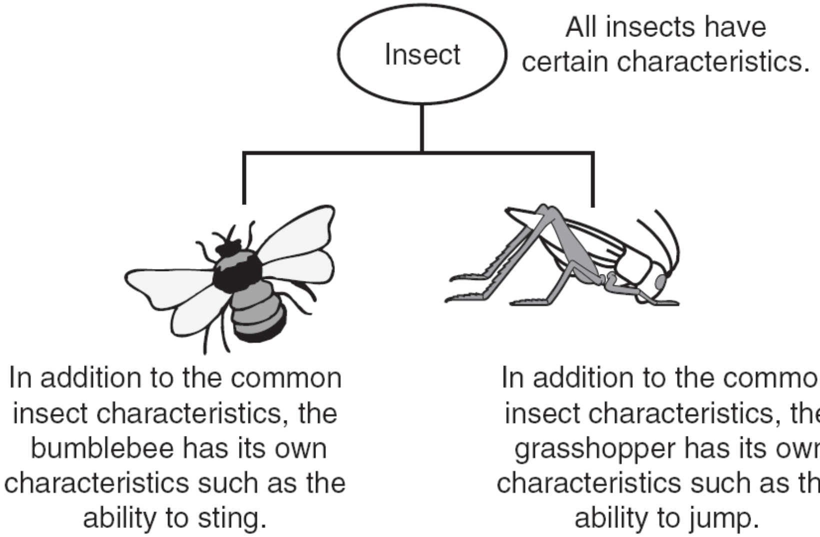
Aggregation, Composition, and Object Lifetimes

- Aggregation represents the owner/owned relationship between objects.
- Composition is a form of aggregation in which the lifetime of the owned object is the same as that of the owner object
- Owned object is usually created as part of the owning object's constructor, destroyed as part of owning object's destructor

P813 program 11-21, eP755~757

11.12 Inheritance

- Inheritance is a way of creating a new class by starting with an existing class and adding new members
- The new class can replace or extend the functionality of the existing class
- Inheritance models the 'is-a' relationship between classes



Inheritance - Terminology

- The existing class is called the base class
  – Alternates: parent class, superclass
- The new class is called the derived class
  – Alternates: child class, subclass

Inheritance Syntax and Notation

```
// Existing class
class Base
{
};
// Derived class
class Derived : public Base
{
};
```

Inheritance of Members

```
class Parent
{
   int a;
   void bf();
};
```

```
Objects of Parent have
Members:int a; void bf();
```

```
class Child : public Parent
{
   int c;
   void df();
};
```

```
Objects of Child have
Members:int a; void bf();
        int c; void df();
```

P819 program 11-22, eP761~762

11.13 Protected Members and Class Access

- protected member access specification: A class member labeled **`protected`** is accessible to member functions of derived classes as well as to member functions of the same class
- Like **`private`**, except accessible to members functions of derived classes

Base Class Access Specification

Base class access specification determines how `private`, `protected`, and `public` members of base class can be accessed by derived classes

Base Class Access

C++ supports three inheritance modes, also called base class access modes:

- public inheritance

```
class Child : public Parent { };
```

- protected inheritance

```
class Child : protected Parent{ };
```

- private inheritance

```
class Child : private Parent{ };
```

Base Class Access vs. Member Access Specification

Base class access is not the same as member access specification:

- Base class access: determine access for inherited members
- Member access specification: determine access for members defined in the class

Member Access Specification

Specified using the keywords

`private`, **`protected`**, **`public`**

```
class MyClass
{
  private: int a;
  protected: int b; void fun();
  public: void fun2();
};
```

Base Class Access Specification

```
class Child : public Parent
{
   protected:
      int a;
   public:
      Child();
};
```

base access

member access

Base Class Access Specifiers

1) **`public`** – object of derived class can be treated as object of base class (not vice-versa)
2) **`protected`** – more restrictive than **`public`**, but allows derived classes to know some of the details of parents
3) **`private`** – prevents objects of derived class from being treated as objects of base class.

Effect of Base Access

Base class members		How base class members appear in derived class

`private: x`
`protected: y`
`public: z`
	`private`
	base class
		`x` inaccessible
		`private: y`
		`private: z`

`private: x`
`protected: y`
`public: z`
	`protected`
	base class
		`x` inaccessible
		`protected: y`
		`protected: z`

`private: x`
`protected: y`
`public: z`
	`public`
	base class
		`x` inaccessible
		`protected: y`
		`public: z`

P823 program 11-23, eP763~765

繼承的模式

- 子類別繼承父類別的模式有 3 種：public， protected 與 private。
- 設有父類別 CA 及其子類別 CB， 其繼承的模式的作用乃在限制子類別 CB 的子類別 CC 使用父類別 CA 成員的權限。
- 以現實的財產繼承來做類似的說明如下，當 CA 有 3 份財產 a (public)、b (protected) 與 c (private)。 財產 c 自己使用，不讓任何其他人使用，財產 b 留給自己的孩子 CB 使用， 財產 a 則任何人都可以碰。
- CB 擁有自己的 3 份財產 x、 y 與 z， 並且繼承了父親的財產 a 與 b。CB 的孩子 CC 可以繼承 CB 自己的財產 x 與 y 這兩份， 財產 z 則 CB 自己使用。 對於 CB 所擁有的繼承財產 a 與 b， 如何讓自己的孩子 CC 去經營，則由 CB 繼承 CA 的繼承模式來決定。
- 如果 CB 覺得自己的孩子 CC 不錯可以去經營財產 a 與 b，而且財產 a 可供給任何人使用，則 CB 繼承 CA 的繼承模式為 public。
- 如果 CB 覺得自己的孩子 CC 相當好， 可以去經營財產 a 與 b，而且任何他人都不得使用財產 a 與 b，則 CB 繼承 CA 的繼承模式為 protected。
- 如果 CB 覺得自己的孩子 CC 不好， 無法經營財產 a 與 b， 則 CB 繼承 CA 的繼承模式為 private。

11.14 Constructors, Destructors and Inheritance

- By inheriting every member of the base class, a derived class object contains a base class object
- The derived class constructor can specify which base class constructor should be used to initialize the base class object

Order of Execution

- When an object of a derived class is created, the base class's constructor is executed first, followed by the derived class's constructor
- When an object of a derived class is destroyed, its destructor is called first, then that of the base class

Order of Execution

```
// Student – base class
// UnderGrad – derived class
// Both have constructors, destructors
int main()
{
   UnderGrad u1;
   ...
   return 0;
}// end main
```

Execute **`Student`** constructor, then execute **`UnderGrad`** constructor

Execute **`UnderGrad`** destructor, then execute **`Student`** destructor

P827 Program 11-24 eP768~769

Passing Arguments to Base Class Constructor

- Allows selection between multiple base class constructors
- Specify arguments to base constructor on derived constructor heading
- Can also be done with inline constructors
- Must be done if base class has no default constructor

Passing Arguments to Base Class Constructor

```
class Parent {
    int x, y;
    public: Parent(int,int);
};
class Child : public Parent {
    int z
    public:
    Child(int a): Parent(a,a*a)
    {z = a;}
};
```

P828~829 example eP770~771

11.15 Overriding Base Class Functions

- Overriding: function in a derived class that has the *same name and parameter list* as a function in the base class
- Typically used to replace a function in base class with different actions in derived class
- Not the same as overloading – with overloading, the parameter lists must be different

Access to Overridden Function

- When a function is overridden, all objects of derived class use the overriding function.
- If necessary to access the overridden version of the function, it can be done using the scope resolution operator with the name of the base class and the name of the function:

```
Student::getName();
```

P832 example eP773~774

Homework 5

TEXT BOOK CH. 11 (P846) Programming Challenges

7. `Rectangle` Class (Aggregation)

Design a `Length` class having two member variables: `centimeters` and `millimeters`. The class should have constructors and member functions to input and return member variables. It should also overload the operators ==, + and *.

Design another class `Rectangle` that should comprise two `Length` objects, representing the two adjacent sides of a rectangle. The class should have the following member functions:

`setSides`: This member function should ask the user to input the values of the sides of the rectangle and accordingly set the value of the lengths by calling the `set` function of the `Length` class.

`getSides`: This member function should call the `get` function of the `Length` class to display the sides of the rectangle.

`isSquare`: This function should return a Boolean value `True` if the rectangle is a square, that is, if. its sides are equal; otherwise it should return `False`.

`getArea` : This should return the area of the rectangle, which is. the product of its two adjacent sides.

`getPerimeter`: This should return the perimeter of the rectangle, which is. the sum of all its four sides.

Demonstrate the classes in a program that creates a `Rectangle` object and calls all its member functions.

Homework 6

TEXT BOOK CH. 11 (P845) Programming Challenges

5. Book, `Journal` and `Magazine` Classes

A certain publisher publishes both books and journals. Design a class `Publication`, which has member variables, `title`, `volume` and `year`. Publicly derive three classes from this class:

`Book` class: that adds member variables `author`, `ISBN` and `price`.

`Journal` class: that adds member variables `month`, `ISSN`, `impactFactor` and `annualSubscription`

`Magazine` class: that adds member variables `month`, `editor` and `annualSubscription`

Include appropriate member functions in each class to input the data members in each class ando display them. Demonstrate the classes in a complete program. The main program should declare objects of classes `Book`, `Journal` and `Magazine` and call their public member functions.