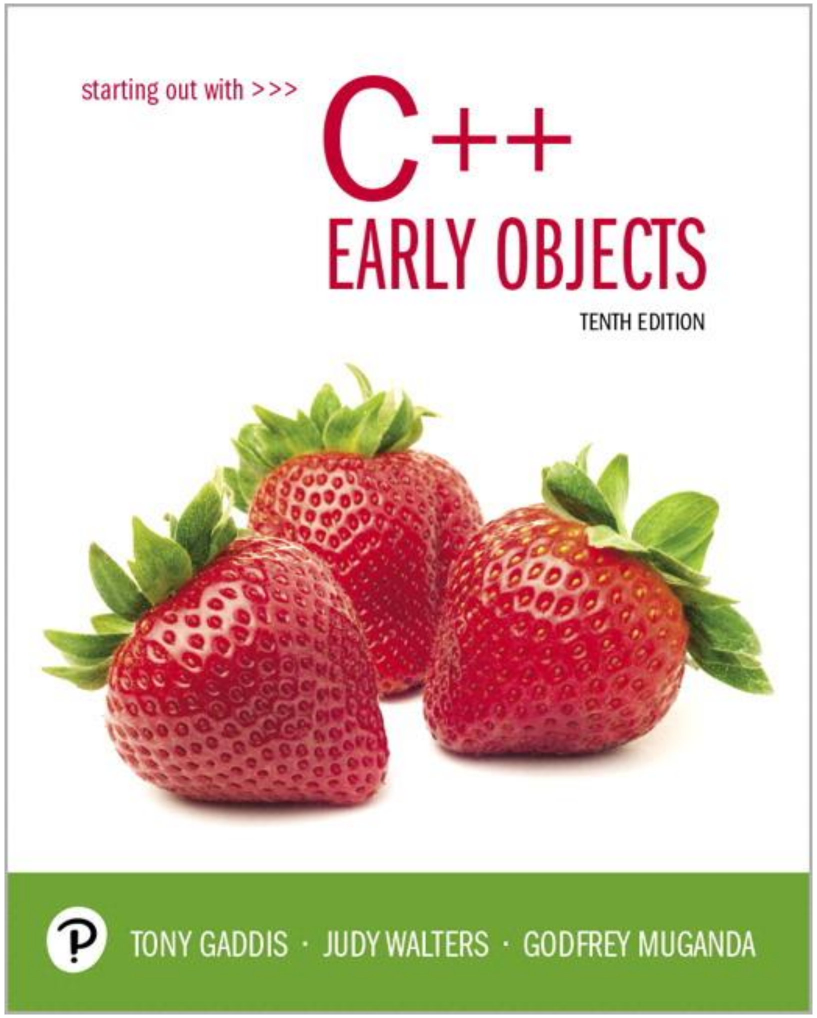
Starting Out with C++ Early Objects

Tenth Edition



Chapter 11

More about Classes and Object-Oriented Programming

Topics 1 of 2

Topics 2 of 2

11.1 The `this` Pointer and Constant Member Functions

- `this` pointer:

  - Implicit parameter passed to every member function

  - it 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; }
};
```

Hands-On Pg. 719

- Listing 11-1

Constant Member Functions

- Declared with keyword **`const`**
- When **`const`** appears in the parameter list,

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

  the function is prevented from modifying the parameter. The parameter is read-only.

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

```
int getX()const
```

  the function is prevented from modifying the object.

11.2 Static Members

- Static member variable:
  - There is one instance of the variable for the entire class
  - It is 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 of 3

1) Must be declared in the class with the 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 of 3

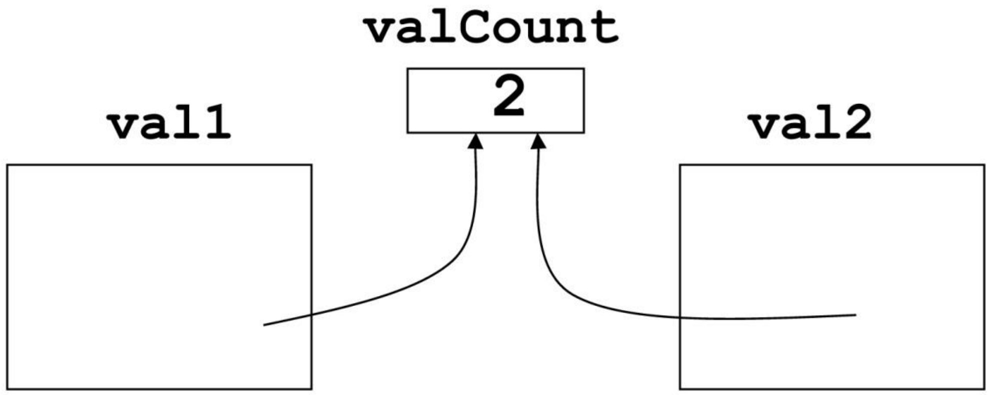
2) Must be defined outside of the class:

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

Static Member Variables 3 of 3

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;
```



Hands-On Pg. 724

- Listing 11-2

Static Member Functions 1 of 2

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

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

Static Member Functions 2 of 2

2) Can be called independently of any 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

Hands-On Pg. 726

- Listing 11-3

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 of 2

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 of 2

2) A 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 1 of 2

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 2 of 2

- 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, you should restrict the property of Friendship to only those functions that must have access to the private members of a class.

Hands-On Pg. 730

- Listing 11-4

11.4 Memberwise Assignment

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

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

Hands-On Pg. 734

- Listing 11-5

11.5 Copy Constructors 1 of 2

A copy constructor

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

Copy Constructors 2 of 2

Problems occur when objects contain pointers to dynamic storage:

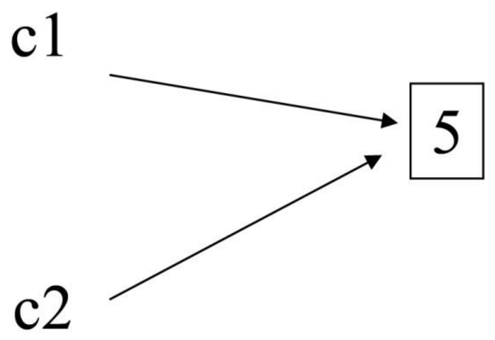
```
class CpClass
{
  private:
   int *p;
  public:
   CpClass(int v=0)
      { p = new int; *p = v;}
   ~CpClass(){delete p;}
};
```

Hands-On Pg. 736

- Listing 11-6

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

Hands-On Pg. 738

- Listing 11-7

Programmer-Defined Copy Constructors 1 of 2

- 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
- The reference parameter should be `const` to avoid potential for data corruption

Programmer-Defined Copy Constructors 2 of 2

- The copy constructor avoids problems caused by memory sharing
- It can allocate separate memory to hold new object's dynamic member data
- It can make the new object's pointer point to the new object's own dynamic memory
- It 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;}
};
```

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

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 1 of 2

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 2 of 2

- 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

- An overloaded 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 1 of 3

- Overloading the assignment operator solves problems with object assignment when an object contains a pointer to dynamic memory.
- The 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 2 of 3

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 3 of 3

The implementation returns a value:

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

Invoking the assignment operator:

```
CpClass a, x(45);


a = x;            // lines can be used
```

Notes on Overloaded Operators

- Overloading can change the entire meaning of an operator
- Most operators can be overloaded
- You cannot change the number of operands of the operator
- Cannot overload the following operators:

```
    ?:   .   .*   sizeof
```

Overloaded Operator – Where Does It Go?

- Overloaded operator as a member function of a class
  - Access to private members
  - Use of the **`this`** pointer
- Overloaded operator as a separate function outside of a class
  - Must declare the operator function as a friend of the class to have access to private members

Overloading Types of Operators

- **`++`, `--`** operators overloaded differently for prefix vs. postfix notation
- Overloaded relational operators should return a **`bool`** value
- Overloaded stream operators **`>>`**, **`<<`** must return **`istream`**, **`ostream`** objects and take **`istream`**, **`ostream`** objects as parameters. These are best overloaded as standalone operator functions.

Overloaded `[ ]` Operator

- Can be used to create classes that behave like arrays
- Can provide bounds-checking on subscripts
- Overloaded `[]` returns a reference to an object, not the object itself

Hands-On Pg. 747

- Listing 11-9
- Table 11-1, Pg. 751

Hands-On Pg. 754

- Listing 11-10

Read-Only Pg. 758

- Length overloading

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

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 initialization from a temporary object

Hands-On Pg. 768

- Listing 11-14

Read-Only Pg. 770

- Overload3.h

Hands-On Pg. 773

- Listing 11-15

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 <u>any</u> of these functions, you should provide your own implementation for <u>all</u> of them.

11.8 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
```

Hands-On Pg. 777

- Listing 11-16

11.9 Convert Constructors

Convert constructors are constructors that take a single parameter of a type other than the class in which they are defined

```
 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
    …
};
```

Pearson

Uses of Convert Constructors 1 of 2

- 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 2 of 2

- 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
```

11.10 Aggregation and Composition

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

Object Composition

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

class Student
{
  private:
        StudentInfo personalData;
     ...
};
```

Member Initialization Lists 1 of 2

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

```
owner_class(parameters):owned_object(parameters);
```

Member Initialization Lists 2 of 2

Use:

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

Aggregation Through Pointers

- A ‘has-a’ relationship can be implemented by owning a pointer to an object
- This 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

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
- The owned object is usually created as part of the owning object's constructor, and destroyed as part of owning object's destructor

Hands-On Pg. 784

- Listing 11-18

11.11 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

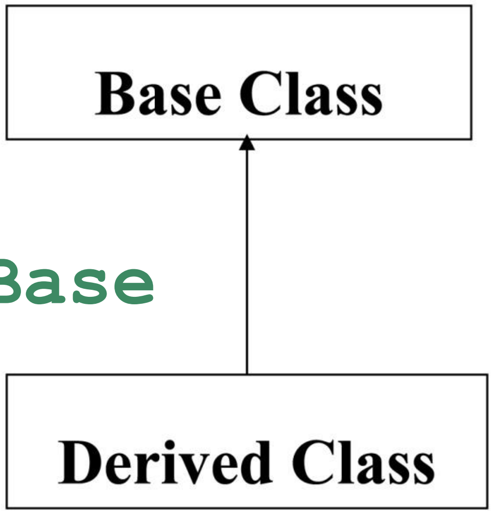
- The existing class is called the base class
  - Alternates: parent class, superclass

- 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 Class Diagram



Inheritance of Members

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

Objects of Parent have members

```
int a; void bf();
```

Objects of Child have members

```
int a; void bf();
int c; void df();
```

Access specifiers will determine availability

Hands-On Pg. 791

- Listing 11-19

Hands-On Pg. 794

- Listing 11-20

Dynamic Allocation of Objects

- When dealing with inheritance, objects should be created using dynamic memory and accessed via pointers to the allocated memory.
- A method that expects an argument of a base class may not have the space needed to accept an argument of a derived class, even though an object of a derived class "is-a" object of the base class.
- Use pointers to objects instead of objects for parameter list and arguments.

Hands-On Pg. 798

- Listing 11-21

11.12 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
- Similar to **`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

Hands-On Pg. 804

- Listing 11-22

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 And Member Access Specification

```
class Child : public Parent // base access
{
  protected:                    // member access
    int a;
  public:                       // member access
    Child();
 };
```

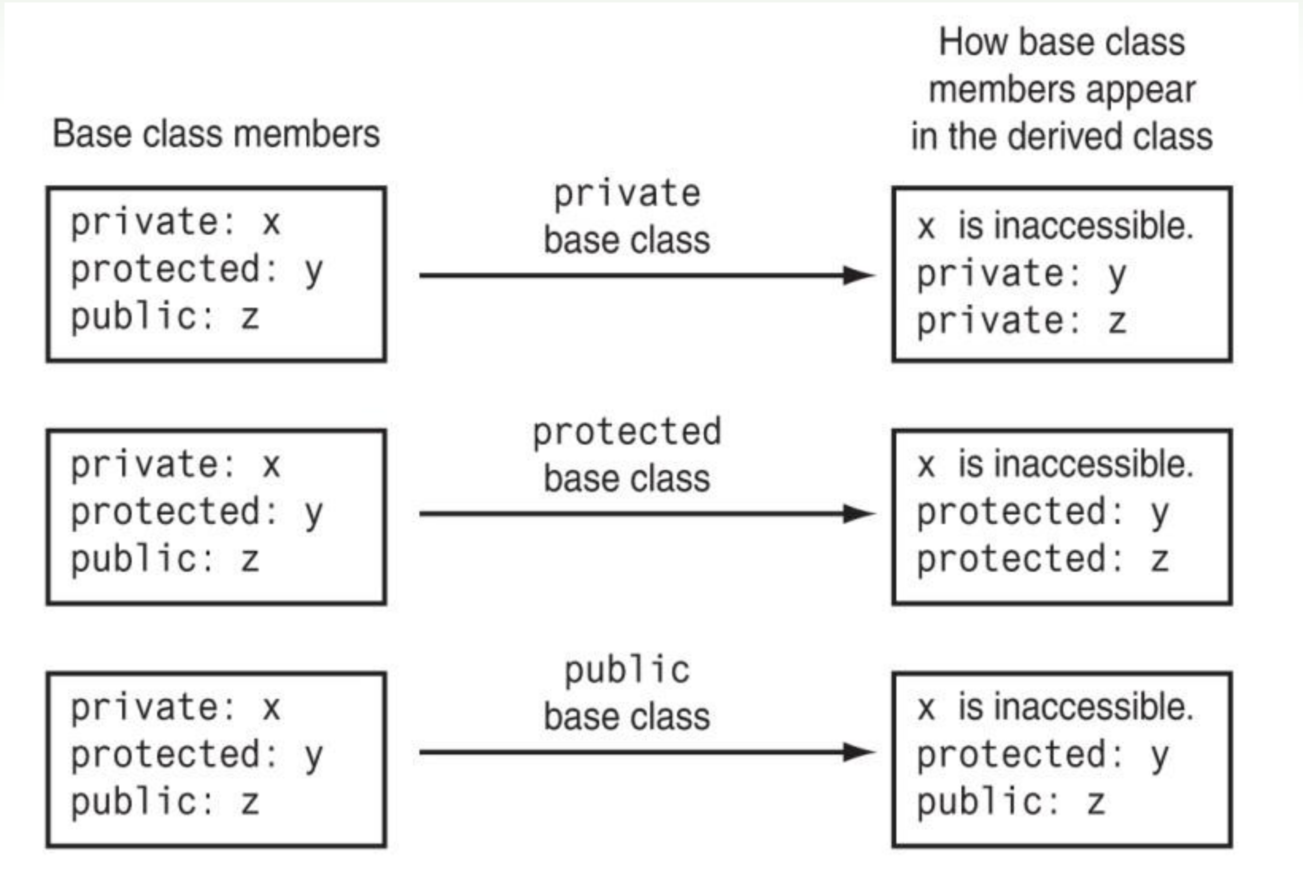
Base Class Access Specifiers

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



11.13 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 1 of 2

- 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 2 of 2

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

Pearson

Passing Arguments to Base Class Constructor 1 of 2

- This allows selection between multiple base class constructors
- You can specify arguments to a base constructor on the derived constructor heading
- Can also be done with inline constructors
- This must be done if the base class has no default constructor

Passing Arguments to Base Class Constructor 2 of 2

```
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;}
};
```

11.15 Overriding Base Class Functions

- Overriding: a 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
- This is 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 a derived class use the overriding function.
- If it is 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();
```

Copyright