UNIT - 3

Operator Overloading & Inheritance concepts

By A. Vijay Bharath
Operator Overloading

- Operator overloading is one of the many exciting features of C++ language.

- It is an important technique that has enhanced the power of C++.

- C++ permits us to add two variables of user-defined types with the same syntax that is applied to basic types.

- This means that C++ has the ability to provide the operators with a special meaning of data type.

- The mechanism of giving such special meanings to an operator is known as operator overloading.
Operator Overloading

Consider the following statements:

```c
int a, b, c;
float x, y, z;
c = a + b;     //integer addition and assignment
z = x + y;    //floating-point addition and assignment
x = a + b;    //integer addition and floating-point assignment
```

For instance, the statement

```
c3 = addcomplex(c1, c2);
```

performs the addition of operands c1 and c2 belonging to the user defined data type and assigns the result to c3.

By A. Vijay Bharath
Operator Overloading

- In C++, by overloading the ‘+’ operator, the above statement can be changed to an easily readable form:

  \[ c3 = c1 + c2; \]

- The operator overloading feature of C++ is one of the methods of realizing *polymorphism*.

- The word *polymorphism* is derived from the Greek words *poly* and *morphism* (polymorphism = poly + morphism).

- Here, *poly* refers to many or multiple and *morphism* refers to action, i.e. performing many actions with a single operator.

By A. Vijay Bharath
Overloadable Operators

- C++ provides a wide variety of operators to perform operations on various operands.

- The operators are classified into *unary and binary* operators based on the number of arguments on which they operate.

- C++ allows almost all operators to be overloaded in which case at least one operand must be an instance of a class.

- It allows overloading of the operators listed below:

By A. Vijay Bharath
### C++ Overloadable Operators

<table>
<thead>
<tr>
<th>Operator Category</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>+, -, *, /, %</td>
</tr>
<tr>
<td>Bit-wise</td>
<td>&amp;,</td>
</tr>
<tr>
<td>Logical</td>
<td>&amp;&amp;,</td>
</tr>
<tr>
<td>Relational</td>
<td>&gt;, &lt;, ==, !=, &lt;=, &gt;=</td>
</tr>
<tr>
<td>Assignment or Initialization</td>
<td>=</td>
</tr>
<tr>
<td>Arithmetic Assignment</td>
<td>+=, -=, *=, /=, %=, &amp;=,</td>
</tr>
<tr>
<td>Shift</td>
<td>&lt;&lt;, &gt;&gt;, &lt;&lt;=, &gt;&gt;=</td>
</tr>
<tr>
<td>Unary</td>
<td>++, --</td>
</tr>
<tr>
<td>Subscripting</td>
<td>[]</td>
</tr>
<tr>
<td>Function Call</td>
<td>( )</td>
</tr>
<tr>
<td>Dereferencing</td>
<td>-&gt;</td>
</tr>
<tr>
<td>Unary Sign Prefix</td>
<td>+, -</td>
</tr>
<tr>
<td>Allocate and Free</td>
<td>new, delete</td>
</tr>
</tbody>
</table>

Table 3.1

By A. Vijay Bharath
Unary Operator Overloading

Consider an example of class Index which keeps track on the index value. The program index1.cpp having class members to maintain the index value is listed below:

```cpp
Class Index
{
  Private: int value;
  Public:
    Index() { value = 0; }  // no argument constructor
    int Getindex()  // index access
    {
      return value;
    }
    void Nextindex()  // Advance index
    {
      value = value + 1;
    }
};
void main()
{
  Index idx1, idx2;
  cout<<"n Index 1=<<idx1.Getindex();
  cout<<"n Index 2=<<idx2.Getindex();
  // Display index values
  cout<<"n Index 1 = " <<idx1.Getindex();
  cout<<"n Index 2 = " <<idx2.Getindex();
  // Advance index objects
  idx1.Nextindex();
  idx2.Nextindex();
  Idx2.Nextindex();
}
```

By A. Vijay Bharath
The program index2.cpp illustrates overloading of ‘++’ operator

Class index
{
Private: int value;
Public:
Index() { value = 0; } //no argument constructor
int Getindex() //index access
{
    return value;
}
void operator ++() //prefix or postfix increment operator
{
    value = value + 1;  // value++;
}
};

void main()
{
    index idx1, idx2;
    cout<<\n Index 1=<<idx1.Getindex();
    cout<<\n Index 1 = " <<idx1.Getindex();
    cout<<\n Index 2=<<idx2.Getindex();
    cout<<\n Index 2 = " <<idx2.Getindex();

    // Display index values
    cout<<\n Index 1 = “ <<idx1.Getindex();
    cout<<\n Index 2 = “ <<idx2.Getindex();

    //advance index objects with ++ operators
    ++idx1; //equivalent to idx1.operator++();
    idx2++;
    idx2++;
Index class and ++ operator overloading

By A. Vijay Bharath
The keyword *operator* facilitates overloading of the C++ operators.

The general format of operator is shown below:

```plaintext
ReturnType    operator     Operatorsymbol ( [arg1, [arg2] ] )
{
    // body of Operator function
}
```

Syntax of operator overloading

By A. Vijay Bharath
‘operator’ keyword

- The keyword operator indicates that the operator symbol following it, is the C++ operator to be overloaded to operate on member of its class.

- The operator overloaded in a class is known as ‘overloaded operator function’.

- Overloading without explicit arguments to an operator function is known as unary operator overloading.

- Overloading with single explicit argument is known as binary operator overloading.
- Overloaded operator member function can be either defined within the body of a class or outside the body of a class.

- The following defines inside the body of class:

```cpp
class MyClass
{
    // class data or function stuff
    int operator ++( ) // member function definition
    {
        // body of a function
    }
};
```

- The same class having the operator member function defined outside its body is as follows: By A. Vijay Bharath
The syntax of overloading the unary operator is shown below:

```
ReturnType operator Operatorsymbol ( )
{
    // body of Operator function
}
```

No explicit arguments

The following example illustrates the overloading of unary operators:

1. Index operator +( )
2. int operator – ( )
3. void operator ++( )
4. void operator --( )
5. int operator *( )

By A. Vijay Bharath
The process of overloading involves the following steps

1. First, create a class that defines the data type that is to be used in the overloading operations.
2. Declare the operator function `operator op()` in public part of the class. It maybe either a member function or a `friend` function.
3. Define the operator function to implement the required operations.

By A. Vijay Bharath
class index
{
private: int value;
public:
index() { value = 0; } // no argument constructor
int Getindex() // index access
{
    return value;
}
index operator ++() // return index object
{
    index temp; // temp object
    value = value + 1; // update index value
    temp.value = value; // initialize temp object
    return temp; // return temp object
}
};

void main()
{
    index idx1, idx2;
    cout<<"\n Index 1=<<idx1.Getindex();
    cout<<"\n Index 2=<<idx2.Getindex();
    idx1 = idx2++; // returned object of idx2++ is assigned to idx1
    idx2++;
    cout<<"\n Index 1 = " <<idx1.Getindex();
    cout<<"\n Index 2 = " <<idx2.Getindex();
}
Overloading unary operators

- Let us consider the unary minus operator. A minus operator, when used as a unary, takes just one operand.

- We know that this operator changes the sign of an operand when applied to a basic data item.

- The unary minus when applies to an object should change the sign of each of its data items.

By A. Vijay Bharath
```cpp
class space
{
  int x, y, z;
public:
  void getdata(int a, int b, int c);
  void display(void);
  void operator -();
};

void space::getdata(int a, int b, int c)
{
  x = a;
  y = b;
  z = c;
}

void space::display(void)
{
  cout << x << "t" << "";
  cout << y << "t" << "";
  cout << z << "n";
}

void space::operator -()
{
  x = -x;
  y = -y;
  z = -z;
}

main()
{
  space s;
  s.getdata(10, -20, 30);
  cout << " S :";
  s.display();
  -s;
  cout << "S:";
  s.display();
}
```

By A. Vijay Bharath
// index class with unary operator -, ++, --
class index
{
private: int value;
public:
index() { value = 0; } //no argument constructor
index(int val) { value = val; } //one arg. constructor
int Getindex() //index access
{
    return value;
}

index  operator -( )
{
    return index(-value); }
index  operator ++ ()
{
    ++value;
    return index(value); }
index  operator ++ ()
{
    value++;  
    return index(value); }
index  operator -- ()
{
    --value; 
    return index(value); }
};
void main()
{
    index idx1, idx2;
cout<<“\n Index 1 = “ <<idx1.Getindex();  
++idx2;
+cout<<“\n Index 2 = “ <<idx2.Getindex();
cout<<“\n Index 1 = “ <<idx1.Getindex();  
--idx2;
cout<<“\n Index 2 = “ <<idx2.Getindex();
}
// index class with unary operator -, ++, - -
class index {
    private: int value;
    public:
        index() { value = 0; }  // no argument constructor
        index(int val) { value = val; }  // one arg. constructor
        int Getindex() { return value; }  // index access
        index operator++() { return index(++value); }
        index operator++(int) { return index(value++); }
};

void main() {
    index idx1(2), idx2(2), idx3, idx4;
    cout << "Index 1 = " << idx1.Getindex();
    cout << "Index 2 = " << idx2.Getindex();
    idx3 = idx1++;
    idx4 = ++idx2;
    cout << "Index 1 = " << idx1.Getindex();
    cout << "Index 3 = " << idx3.Getindex();
    cout << "Index 2 = " << idx2.Getindex();
    cout << "Index 4 = " << idx4.Getindex();
}
Overloading Binary Operators

- The same mechanism can be used to overload a binary operator.

- A statement like `C = sum(A, B);` //functional notation
  can be replaced by `C = A + B;` //arithmetic notation

  By overloading the `+` operator using `operator+( )` function
class complex
{
    float x, y;  // real, imag.

public:
    complex() {}
    complex(float real, float imag) // constructor two
    {
        x = real; y = imag;
    }
    complex operator+(complex); // constructor one
    void display(void);
};

complex complex : : operator+(complex c)
{
    complex temp;
    temp.x = x + c.x;
    temp.y = y + c.y;
    return(temp);
}

void complex : : display(void)
{
    cout<<x <<"+j"<<y<<"\n";
}

main()
{
    Complex c1, c2, c3; // invokes constructor 1
    C1 = complex(2.5, 3.5); // invokes constructor 2
    C2 = complex(1.6, 2.7); // invokes constructor 2
    C3 = C1 + C2;
    cout<< "C1 = " ;
    C1.display();
    cout<< "C2 = " ;
    C2.display();
    cout<< "C3 = " ;
    C3.display();
}
complex operator+(complex c) {
    complex temp;
    temp.x = c.x + x;
    temp.y = c.y + y;
    return (temp);
}

return C3 = C1 + C2

4.10 6.20 x 2.50 3.50 y 1.60 2.70 x y

By A. Vijay Bharath
Complex c1, c2, c3

\{ 
  \text{C1} = \text{complex}(2.5, 3.5); 
  \text{C2} = \text{complex}(1.6, 2.7); 
\}

\text{C3} = \text{c1 + c2};

\text{C3.display}();

\text{private member variables}

float real;
float imag;

\text{complex( ) (constructor)}

\text{Operator +(complex) (addition)}

\text{getdata( ) (initialize)}

\text{outdata( )}
Comparison Operator

- Similar to arithmetic operators, the relational operators can be overloaded for comparing the magnitudes of the operands.

- The program `idxcmp.cpp` demonstrates the overloading of the comparison operator `<` to compare indexes.

By A. Vijay Bharath
#include<iostream.h>
enum boolean {FALSE, TRUE};
class index
{
private : int value;
public:
    index()
    { value = 0; }
    index(int val)
    {
        value = val;
    }
    int getindex()
    {
        return value;
    }
    boolean operator<(index idx)
    {
        return(value < idx.value ? TRUE : FALSE);
    }
};
void main()
{
    index idx1=5;
    index idx2=10;
    cout<<"\n Index1="<<idx1.getindex();
    cout<<"\n Index2="<<idx2.getindex();
    if(idx1 < idx2)
        cout<<"\n Index1 is less than Index2";
    else
        cout<<"\n Index1 is not less than Index2";
}
Conversion between Basic Data Types

- Consider the statement
  
  ```
  weight = age;  //weight is of float type and age is of integer type
  ```

  where `weight` is of type float and `age` is of type integer.

- Here, the compiler calls a special routine to convert the value of `age`, which is represented in an integer format, to a floating-point format, so that it can be assigned to `weight`.

- The compiler has several built-in routines for the conversion of basic data types such as `char to int`, `float to double`.

By A. Vijay Bharath
• This feature of the compiler, which performs conversion of data without the user intervention is known as implicit type conversion.

• The compiler can be instructed explicitly to perform type conversion using the type conversion operators known as typecast operators. For instance to convert int to float, the statement is

\[
\text{weight} = (\text{float}) \text{age};
\]

where the keyword float enclosed between braces is the type cast operator. In C++ the above statement can also be expressed in a more readable form as

\[
\text{weight} = \text{float} (\text{age});
\]

By A. Vijay Bharath
Conversion between Objects and Basic Types

- To convert data from a basic type to a user-defined type, the conversion function should be defined in user-defined object’s class in the form of constructor.

- This constructor function takes a single argument of basic-type as shown in Figure.

```
Constructor (BasicType)
{
  // steps for converting
  //BasicType to Object attributes
}
Conversion function: basic to user-defined
```
Operator BasicType()
{
    // steps for converting
    // Object attributes to BasicType
}

Conversion function: user-defined to basic
Inheritance

- C++ strongly supports the concept of reusability.

- Once a class has been written and tested, it can be adapted by other programmers to suit their requirements.

- This is basically done by creating new classes, reusing the properties of the existing ones.

- The mechanism of deriving a new class from an old one is called inheritance. The old class is referred to as the base class and the new one is called the derived class.

By A. Vijay Bharath
- The derived class inherits some or all of the traits from the base class.

- A class can also inherit properties from more than one class or from more than one level.

- A derived class with only one base class is called single inheritance.

- A derived class with several base classes is called multiple inheritance.

- On the other hand, the traits of one class may be inherited by more than one class. This process is known as hierarchical inheritance.

By A. Vijay Bharath
The mechanism of deriving a class from another derived class is known as **multilevel inheritance**.

The below fig. shows the various forms of inheritance that could be used for writing extensible programs. The direction of arrow indicates the direction of inheritance.
a) Single inheritance

b) Multiple inheritance

c) Hierarchical inheritance

Forms of inheritance

By A. Vijay Bharath
d) Multilevel inheritance

e) Hybrid inheritance

Forms of inheritance
By A. Vijay Bharath
● Inheritance is a technique of organizing information in a hierarchical form.

● It is like a child inheriting the features of its parents (such as beauty of the mother and intelligence of the father).

● In real world, an object is described by using inheritance. It derives general properties of an object by tracing an inheritance tree from one specific instance, upwards towards the primitive concepts at the root.

By A. Vijay Bharath
The derived class inherits all the capabilities of the base class and can add refinements and extensions of its own. The base class remains unchanged.

The below fig. represents the derived class inherits the feature of the base class (A, B, C) and add its own feature (D).

The arrow in the diagram represents derived from.

Its direction from the derived class towards the base class, represents that the derived class accesses features of the base class and not vice versa.
Base class and derived class relationship

Ancestor, parent, superclass

Descendent, child, subclass

Defined in derived class

Defined in base class and also accessible from derived class

By A. Vijay Bharath

Base class

Derived class

Feature A

Feature B

Feature C

Feature D
Defining Derived Class

- A derived class is defined by specifying its relationship with the base class in addition to its own details. The general form of defining a derived class is:

  ```
  class derived-class-name : visibility-mode base-class-name
  {
     . . . .
     . . . . // members of derived class
     . . . .
  }
  ```

- The colon indicates that the `derived-class-name` is derived from the `base-class-name`. 

By A. Vijay Bharath
Class Revisited

- C++, not only supports the access specifiers **private** and **public** but also an important access specifier, **protected**.

- A class can use all the three visibility modes as illustrated below:

```plaintext
class class-name
{
    private:
    ........    // visible to member function within
    ........    // its class but not in derived class

    protected:
    ........    // visible to member function within
    ........    // its class and derived class

    public:
    ........    // visible to member function within
    ........    // its class, derived classes and through object
};
```

By A. Vijay Bharath
Similar to the private members of a class, the protected members can be accessed only within the class.

That is, in the hierarchy of access, privilege code (members and friend) can see the whole structure of an object whereas, the external code can see only the public features.

Consider the following definition of a class to illustrate the visibility limit of the various class members.
Class x
{
    \textbf{Private:}
    \begin{itemize}
    \item int a;
    \item void f1()
    \end{itemize}
    \begin{itemize}
    \item \{ // . . . Can refer to members a, b, c and functions f1, f2, f3
    \item \}
    \end{itemize}

    \textbf{Protected:}
    \begin{itemize}
    \item int b;
    \item void f2()
    \end{itemize}
    \begin{itemize}
    \item \{ // . . . Can refer to members a, b, c and functions f1, f2, f3
    \item \}
    \end{itemize}

    \textbf{Public:}
    \begin{itemize}
    \item int c;
    \item void f3()
    \end{itemize}
    \begin{itemize}
    \item \{ // . . . Can refer to members a, b, c and functions f1, f2, f3
    \item \}
    \end{itemize}
\};

By A. Vijay Bharath
● The data member \( a \) is \textit{private} to class \( x \) and is accessible only to members of its own class, that is member function \( f1() \), \( f2() \), \( f3() \) can access directly.

● However, statements outside and even member function of the derived class are not allowed to access \( a \) directly.

● In addition, the member function \( f1() \) can be called by other members of class \( x \). The statements outside the class cannot call \( f1() \).

```cpp
Class x
{
    Private:
    int a;
    void f1( )
    {
    }

    Protected:
    int b;
    void f2( )
    {
    }

    Public:
    int c;
    void f3( )
    {
    }
};
```

By A. Vijay Bharath
The data member b and the member function f2() are protected. These members are accessible to other member functions of the class x and member functions in a derived class.

However, outside the class protected members have private status. The statements outside the class cannot directly access members b or f2() using the class.

By A. Vijay Bharath
The data member \( c \) and the member function \( f3() \) are \textit{public}, and may be accessed directly by all the members of class \( x \), or by the members in a derived class, or by the objects of the class.

Public members are always accessible to all users of the class.

---

By A. Vijay Bharath
The following statements,

\[
x \ objx; \quad //objx \ is \ an \ object \ of \ class \ x \\
int \ d; \quad //temporary \ variable \ d
\]

Define the object \texttt{objx} of the class \texttt{x} and the integer variable \texttt{d}. The member access privileges are illustrated by the following statements referring to object \texttt{objx}.

1. **Accessing private members of the class \texttt{X}**

\[
d = \ objx.a; \quad //Error: ‘x: : a’ is not accessible \\
\texttt{objx.f1();} \quad //Error: ‘x::f1() is not accessible
\]

Both the statements are invalid b’cas the private members are not accessible. 

By A. Vijay Bharath
2. Accessing protected members of the class x

   \[
   d = \text{objx.b}; \quad // \text{Error: ‘x::b’ is not accessible}
   \text{objx.f2();} \quad // \text{Error: ‘x::f2()’ is not accessible}
   \]

Both the statements are invalid b’cas the protected members of a class are inaccessible since they are private to the class X.

3. Accessing public members of the class x

   \[
   d = \text{objx.c;} \quad // \text{OK}
   \text{objx.f3();} \quad // \text{OK}
   \]

Both the statements are valid b’cas the public members of a class are accessible to statements outside the scope of class.

By A. Vijay Bharath
Class B
{ int a;
 Public: int b;
 Void get_ab();
 Int get_a(void);
 Void show_a(void);
};
Class D:public B
{
 int c;
 Public: void mul(void);
 Void display(void);
};

int b::get_ab(void)
{ a = 5; b = 10; }
int b::get_a()
{return a;}
void B:: show_a()
{ cout<<“a=“<<a<<“\n”;}
void D::mul()
{ 
 C=b*get_a(); }
void D:: display()
{ cout<<“a=“<<get_a();
 cout<<“b=“<<b;
 cout<<“c=“<<c;  }

Main()
{ 
 D d;
 d.get_ab();
 d.mul();
 d.show_a();
 d.display();
 d.b=20;
 d.mul();
 d.display();
}

By A. Vijay Bharath
The following are the 3 possible styles of derivation:

1. Class D : public B    //public derivation
   {
       // members of D
   }

2. Class D : private B    //private derivation
   {
       //members of D
   }

3. Class D : B            //private derivation by default
   {
       //members of D
   }

By A. Vijay Bharath
- Inheritance of a base class with visibility mode public, by a derived class, causes public members of the base class to become public members of the derived class and protected members of the base class become protected members of the derived class.

- Hence, the objects of a derived class can access public members of the base class that are inherited as public using dot operator.

- However, protected members cannot be accessed with the dot operator.
class Base
private
protected
public

class Derv : public Base
private
protected
public

Derv objD
Base objB

By A. Vijay Bharath
Inheritance of a base class with visibility mode `private` by a derived class, causes public members of the base class to become private members of the derived class and the protected members of the base class become private members of the derived class.

The visibility of base class members undergoes modifications in a derived class as summarized in below Table:

```cpp
Class D:private B //private derivation
{
    //members of D
};
```
The private members of the base class remain private to the base class, whether the base class is inherited publicly or privately.

By A. Vijay Bharath
Let us consider for **private** derivation

Class B

```cpp
{  
    int a;
    public:
        int b;
    Void get_ab();
    void get_a();
    void show_a();
};
```

Class D: **private** B // private derivation

```cpp
{  
    int c;
    public:
        void mul();
        void display();
};
```
In private derivation, the public members of the base class become private members of the derived class.

Therefore, the objects of D cannot have direct access to the public members functions of B.

The statements such as

```
d.get_ab();
d.get_a();
d.show_a()
```

will not work. However, these functions can be used inside `mul()` and `display()` like the normal function as shown below:

```
Void mul()
{
    get_ab();
    c = b * get_a();
}

void display()
{
    show_a(); //outputs value of ‘a’
    cout << "b = " << b << "\n";
    cout << "c = " << c << "\n";
}
```

By A. Vijay Bharath
Class B
{ int a; //private
 public: int b;
 void get_ab();
 int get_a(void);
 soid show_a(void); 
};

class D: private B //private derivation
{
 int c;
 public: void mul(void);
 void display(void);
};

void B::get_ab(void)
{ cout<<“enter values for a and b:”;
  cin>>a>>b; }

int b::get_a()
{ return a; }
void B:: show_a()
{ cout<<“a=“<<a<<“\n”; }
void D::mul()
{ get_ab();
 c=b*get_a(); } 
void D:: display()
{ 
 show_a(); //outputs value of ‘a’
 cout<<“b=“<<b;
 cout<<“c=“<<c; }

main()
{
 D d;
 d.mul(); // d.get_ab(); wont work
 d.display(); // d.show_a(); wont work
 //d.b=20; wont work; b has become private
 d.mul();
 d.display();
} 

By A. Vijay Bharath
The following rules are to be borne in mind while deciding whether to define members as private, protected, or public.

1. A private member is accessible only to members of the class in which the private member is declared. They cannot be inherited.

2. A private member of the base class can be accessed in the derived class through the member functions of the base class.

3. A protected member is accessible to members of its own class and any of the members in a derived class.

4. If a class is expected to be used as a base class in future, then members which might be needed in the derived class should be declared protected rather than private.

By A. Vijay Bharath
The following rules are to be borne in mind while deciding whether to define members as private, protected, or public.

5. A public member is accessible to members of its own class, members of the derived class, and outside users of the class.

6. The visibility mode in the derivation of a new class can be either private or public.

7. Constructors of the base class and the derived class are automatically invoked when the derived class is instantiated. If a base class has constructors with arguments, then their invocations must be explicitly specified in the derived class’s initialization section. Remember that, constructors must be defined in public section of a class (base and derived).
Consider the following declarations of the base class to illustrate public and private inheritance.

**Public inheritance**

Consider the following declarations to illustrate the derivation of a new class D from the base class B publicly declared earlier:

Class D:

```cpp
public B
{
Private: int privateD;
Protected: int protectedD;
Public: int publicD;
void myfunc()
{
    int a;
    a=privateB;       //error: B::privateB is not accessible
    a=getBprivate();  //OK, inherited member accesses private data
    a=protectedD;     //OK
    a=publicD;        //OK
}
};
```

Class B //base class
{
Private:    int privateB;
Protected:  int protectedB;
Public:     int publicB;
Int getBprivate()
{
    Return privateB;
    }
};

By A. Vijay Bharath
The member function `myfunc()` of the class D can access protectedB and publicB inherited from the base class B.

Since the class B is inherited as public by the derived class D, the status of members protectedB and publicB, getBprivate() remain unchanged in the derived class D.

The statements

```cpp
D objd;     //objd is a object of class D
int d;      //temporary variable d
```

consider the following statements referring to the object objd. Access to the protected member of the base class B in the statement

```cpp
d = objd.protectedB;  //error: ‘B::protectedB’ is not accesible
```

is invalid; protectedB has protected visibility status in class D.

By A. Vijay Bharath
Statements like
\[ d = \text{objd.publicB}; \quad \text{//OK is valid (publicB has public visibility in class D)} \]

The inherited member function getBprivate() in the statement
\[ d = \text{objd.getBprivate();} \quad \text{//OK, inherited member access private data} \]
access the private data member of the base class.

Ex. Class X : public D
{
    public :
    void g();
};

The member function g() in the derived class X may still access members protectedB and publicB and even retains the original protected and public status.

By A. Vijay Bharath
private inheritance

Consider the following declarations to illustrate the derivation of a new class D from the base class B privately:

Class D: private B

{
Private: int privateD;
Protected: int protectedD;
Public: int publicD;

void myfunc()
{
    int a;
    a=privateB;       //error: B::privateB is not accessible
    a=getBprivate();  //OK, inherited member accesses private data
    a=protectedB;     //OK
    a=publicB;        //OK
}

Class B //base class
{
Private: int privateB;
Protected: int protectedB;
Public: int publicB;

Int getBprivate()
{
    Return privateB;
}

};

By A. Vijay Bharath
The member function `myfunc()` of the class D can access `protectedB` and `publicB` inherited from the base class B.

Since the base class B is inherited as private base class of the derived class D, the status of members `protectedB`, `publicB`, and `getBprivate()` become private in the derived class D. The statements

```cpp
D objd; // objd is a object of class D
int d; // temporary variable d
```

consider the following statements referring to the object `objd`. Access to the protected member of the base class B in the statement

```cpp
d = objd.protectedB; // error: ‘B::protectedB’ is not accessible
```

is invalid; `protectedB` has private visibility status in class D.
Statements like
\[ d = \text{objd.publicB}; \quad \text{//error: B::publicB is not accessible} \]

The inherited member function getBprivate() in the statement
\[ d = \text{objd.getBprivate()}; \quad \text{//error} \]

it has become private to the derived class D, however, a member function of the derived class can access – myfunc() accesses getBprivate() function.

Ex. Class X : public D

```cpp
{ 
     public :
     void g();
};
```

The member function g() in X cannot access members protectedB and publicB since these members have gained private visibility status in class D

By A. Vijay Bharath
ACCESS MECHANISM IN CLASSES

Class B
- Private
- Protected
- Public

Class D1: public B
- Private
- Protected
- Public

Class D2: private B
- Private
- Protected
- Public

Class X: public D1, public D2
- Private
- Protected
- Public

By A. Vijay Bharath
Member functions accessibility

- The various categories of functions which have access to the private and protected members could be any of the following:
  - A member function of a class
  - A member function of a derived class
  - A friend function of a class
  - A member function of a friend class

By A. Vijay Bharath
The below table shows the access control to class members

<table>
<thead>
<tr>
<th>Function Type</th>
<th>Access directly to</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Private</td>
</tr>
<tr>
<td>Class member</td>
<td>yes</td>
</tr>
<tr>
<td>Derived class member</td>
<td>No</td>
</tr>
<tr>
<td>Friend</td>
<td>Yes</td>
</tr>
<tr>
<td>Friend class member</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Constructors in Derived Classes

- The constructors play an important role in initializing an object’s data members and allocating required resources such as memory.

- The derived class need not have a constructor as long as the base class has a no-argument constructor.

- However, if the base class has constructors with arguments (one or more), then it is mandatory for the derived class to have a constructor and pass the arguments to the base class constructor.

- When an object of a derived class is created, the constructor of the base class is executed first and later the constructor of the derived class.

By A. Vijay Bharath
The following examples illustrate the order of invocation of constructors in the base class and the derived class

1. No-constructor in the base class and derived class

When there are no constructors either in the base or derived classes, the compiler automatically creates objects of classes without any error when the class is instantiated.

//cons1.cpp: no constructor in base and derived class
#include<iostream.h>
Class B //base class
{
    //body of base class, without constructors
}
Class D : public B //publicly derived class
{
    //body of derived base class, without constructors
    Public:
    void msg()
    {
        cout<<“No constructors exists in base and derived class”<<endl;
    }
};

void main()
{
    D objd; //base constructor
    objd.msg( );
}

Run: No constructors exists in base and derived class

By A. Vijay Bharath
2. Constructors only in the base class
//cons2.cpp: constructor in base class only

#include<iostream.h>
Class B //base class
{
    public:
    B()
    {
        Cout<<“No-argument constructor of the base class B is executed”;
    }
};
Class D : public B //publicly derived class
{
    Public:
};

void main()
{
    D objd; // accesses base constructor
}

Run: No-argument constructor of the base class B is executed

By A. Vijay Bharath
3. Constructors only in the Derived class

//cons3.cpp: constructor in derived class only

#include<iostream.h>
Class B    //base class
{
  public:
  //body of base class, without constructors
  
};
Class D : public B    //publicly derived class
{
  //body of derived base class, without constructors
  Public:
  D( )
  {
  cout<<“constructors exists in only in derived class”<<endl;
  }
  
};

void main()
{
  D objd;    // accesses derived constructor
}

Run: constructors exists in only in derived class

By A. Vijay Bharath
4. Constructors only in both Base and Derived class
//cons4.cpp: constructor in both based and derived class

#include<iostream.h>

Class B  //base class
{
  public:
    B()
    {
      cout<<"No-argument constructors of the base class B executed first "<<endl;
    
    
    
};

Class D : public B  //publicly derived class
{
  Public:
    D()
    {
      cout<<"No-argument constructors of the derived class D executed next"<<endl;
    }
  
};

void main()
{
  D objd;   // accesses both constructor
}

Run:
No-argument constructors of the base class B executed first
No-argument constructors of the derived class D executed next

By A. Vijay Bharath
5. Multiple Constructors in base class and a single constructor in Derived class

//cons5.cpp: Multiple Constructors in base class and a single in Derived class

#include<iostream.h>
Class B   //base class
{
  public:
B( ) { cout<<"No-argument constructors of the base class B"; }
B( int a) { cout<<"One-argument constructors of the base class B"; }
};
Class D : public B   //publicly derived class
{
  Public:
D( int a)
{
  cout<<"One-argument constructors of the derived class D;"
}
};

void main()
{
  D objd(3);
}

Run:
No-argument constructors of the base class B
One-argument constructors of the derived class D

By A. Vijay Bharath
6. Constructors in base and derived classes without default constructor

//cons6.cpp: Constructors in base and derived class

#include<iostream.h>

Class B //base class
{
    public:
    B( int a) { cout<<"One-argument constructors of the base class B"; }
};

Class D : public B //publicly derived class
{
    Public:
    D( int a)
    {
    cout<<"One-argument constructors of the derived class D;"
    }
};

void main()
{
    D objd(3);
}

Run:
The compilation of the above program generates the following error:

cannot find ‘default’ constructor to initialize base class ‘B’

This can be overcome by explicit invocation of a constructor of base class by cons7.cpp as follows

By A. Vijay Bharath
7. Explicit invocation in the absence of default constructor

//cons7.cpp: Constructors in base and derived class

#include<iostream.h>
Class B  //base class
{
  public:
  B( int a) { cout<"One-argument constructors of the base class B"; }
}

Class D : public B  //publicly derived class
{
  Public:
  D( int a) : B(a)
  {
    cout<"One-argument constructors of the derived class D; 
  }
}

void main()
{
  D objd(3);
}

Run:
One-argument constructors of the base class B
One-argument constructors of the derived class D

By A. Vijay Bharath
By A. Vijay Bharath
By A. Vijay Bharath
By A. Vijay Bharath
By A. Vijay Bharath