\

UNIT-II

String class

Strings are objects that represent sequences of characters.

The standard string class provides support for such objects with an interface similar to that of a standard container of bytes, but adding features specifically designed to operate with strings of single-byte characters.

The string class is an instantiation of the basic_string class template that uses char (i.e., bytes) as its *character type*, with its default char_traits and allocator types (see basic_string for more info on the template).

Member functions

(constructor)

Construct string object (public member function)

Construct string object

Constructs a string object, initializing its value depending on the constructor version used:

(1) empty string constructor (default constructor)

Constructs an empty string, with a length of zero characters.

(2) copy constructor

Constructs a copy of *str*.

(3) substring constructor

Copies the portion of *str* that begins at the character position *pos* and spans *len* characters (or until the end of *str*, if either *str* is too short or if *len* is string::npos).

Iterators:

begin

Return iterator to beginning (public member function)

end

Return iterator to end (public member function)

size

Return length of string (public member function)

length

Return length of string (public member function)

max_size

Return maximum size of string (public member function)

resize

Resize string (public member function)
capacity
Return size of allocated storage (public member function)
reserve
Request a change in capacity (public member function)

clear
Clear string (public member function)
empty
Test if string is empty (public member function)
Non-member function overloads
operator+
Concatenate strings (function)
relational operators
Relational operators for string (function)
swap
Exchanges the values of two strings (function)
operator>>
Extract string from stream (function)
operator<<
Insert string into stream (function)
getline
Get line from stream into string (function)

Example:

```
#include <iostream>
#include <string>
using namespace std;

int main ()
{
   char *line = "short line for testing";

   // with no arguments
   string s1;
   s1 = "Anatoliy";
   cout << "s1  is: " << s1 << endl;

   // copy constructor
   string s2 (s1);
```

```
    cout << "s2  is: " << s2 << endl;

    // one argumen
    string s3 (line);
    cout << "s3  is: " << s3 << endl;

    // first argumen C string
    // second number of characters
    string s4 (line,10);
    cout << "s4  is: " << s4 << endl;

    // 1 - C++ string
    // 2 - start position
    // 3 - number of characters
    string s5 (s3,6,4); // copy word 'line' from s3
    cout << "s5  is: " << s5 << endl;

    // 1 - number characters
    // 2 - character itself
    string s6 (15,'*');
    cout << "s6  is: " << s6 << endl;

    // 1 - start iterator
    // 2 - end iterator
    string s7 (s3.begin(),s3.end()-5);
    cout << "s7  is: " << s7 << endl;

    // you can instantiate string with assignment
    string s8 = "Anatoliy";
    cout << "s8  is: " << s8 << endl;

    return 0;
}
OUTPUT:
// s1  is: Anatoliy
// s2  is: Anatoliy
// s3  is: short line for testing
// s4  is: short line
// s5  is: line
// s6  is: ***************
```

```
// s7  is: short line for te
// s8  is: Anatoliy
```

USING GETLINE MEMBER FUNCTION

```
#include <iostream>
#include <string>
#include <vector>
#include <fstream>
using namespace std;

int main ()
{
   string str;
   cout << "Enter string (EOL = $) : ";
   getline (cin, str, '$');
   cout << "Str is : " << str << endl;

   ifstream In("data.dat");
   vector v;

   cout << endl << "Read data from file" << endl;
   while ( ! In.eof() )
   {
         getline (In, str);
         v.push_back(str);
   }

   copy (v.begin(),v.end(),
                  ostream_iterator(cout,"\n"));
   cout << endl;


   return 0;
}
```

OUTPUT:

```
// Enter string (EOL = $) : Str is : first line
// second line$
//
// Read data from file
// file: "data.dat"
```

```
// second line
// last line
```

SWAPPING OF TWO STRING

```
#include <iostream>
#include <string>
using namespace std;

int main ()
{
   string str;

   cout << "Enter string for testing : ";
   cin  >> str;
   cout << "\nString is : " << str << endl;

   cout << "Enter string for testing "
           << "(d to quit) : ";
   while ( cin >> str )
   {
           cout << endl;
           cout << "String is : " << str << endl;
           cout << "Enter string for testing "
           << "(d to quit) : ";
   }

   return 0;
}
```

OUTPUT:

```
// Enter string for testing : first
// String is : first
// Enter string for testing (d to quit) : second
// String is : second
// Enter string for testing (d to quit) : third
// String is : third
// Enter string for testing (d to quit) :
```

+ += = operators 1.

COPY CONSTRUCTOR

The **copy constructor** is a constructor which creates an object by initializing it with an object of the same class, which has been created previously. The copy constructor is used to:

- Initialize one object from another of the same type.
- Copy an object to pass it as an argument to a function.
- Copy an object to return it from a function.

If a copy constructor is not defined in a class, the compiler itself defines one.If the class has pointer variables and has some dynamic memory allocations, then it is a must to have a copy constructor. The most common form of copy constructor is shown here:

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

Here, **obj** is a reference to an object that is being used to initialize another object.

```
#include <iostream>

using namespace std;

class Line
{
  public:
    int getLength( void );
    Line( int len );            // simple constructor
    Line( const Line &obj);  // copy constructor
    ~Line();                    // destructor

  private:
    int *ptr;
};

// Member functions definitions including constructor
Line::Line(int len)
{
   cout << "Normal constructor allocating ptr" << endl;
   // allocate memory for the pointer;
   ptr = new int;
   *ptr = len;
}
```

```
Line::Line(const Line &obj)
{
   cout << "Copy constructor allocating ptr." << endl;
   ptr = new int;
  *ptr = *obj.ptr; // copy the value
}

Line::~Line(void)
{
   cout << "Freeing memory!" << endl;
   delete ptr;
}
int Line::getLength( void )
{
   return *ptr;
}

void display(Line obj)
{
  cout << "Length of line : " << obj.getLength() <<endl;
}

// Main function for the program
int main( )
{
  Line line(10);

  display(line);

  return 0;
}
```
When the above code is compiled and executed, it produces the following result:
```
Normal constructor allocating ptr
Copy constructor allocating ptr.
Length of line : 10
Freeing memory!
Freeing memory!
```

Polymorphism

Polymorphism means many forms (ability to take more than one form). In Polymorphism poly means "multiple" and morph means "forms" so polymorphism means many forms.

In polymorphism we will declare methods with same name and different parameters in same class or methods with same name and same parameters in different classes. Polymorphism has ability to provide different implementation of methods that are implemented with same name.

In Polymorphism we have 2 different types those are

- **Compile Time Polymorphism** (Called as Early Binding or Overloading or static binding)

- **Run Time Polymorphism** (Called as Late Binding or Overriding or dynamic binding)

Compile Time Polymorphism

Compile time polymorphism means we will declare methods with same name but different signatures because of this we will perform different tasks with same method name. This compile time polymorphism also called as **early binding** or **method overloading**.

Method Overloading or compile time polymorphism means same method names with different signatures (different parameters)

Run Time Polymorphism

Run time polymorphism also called as **late binding** or **method overriding** or **dynamic polymorphism**. Run time polymorphism or method overriding means same method names with same signatures.

In this run time polymorphism or method overriding we can override a method in base class by creating similar function in derived class this can be achieved by using inheritance principle and using "**virtual** & **override**" keywords.

1) Run Time polymorphism: a) Virtual function
2) Compile Time Polymorphism : a) function overloading
b) operator overloading

Example of compile time polymorphism:
Example 1: example of compile time polymorphism; static time binding

```
void f(int i){cout<<"int";}
void f(char c){cout<<"char";}
int main()
{
  f(10);
  return 0;
}
```

Output: int

Example of run time polymorphism:

```
class Base
{
public:

   virtual void display(int i)
   { cout<<"Base::"<<i; }
};

class Derv: public Base
{
public:

   void display(int j)
   { cout<<"Derv::"<<j; }
};

int main()
{
   Base *ptr=new Derv;
   ptr->display(10);
   return 0;
}
```

Output: Derv::10

Function overloading in C++:

You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You can not overload function declarations that differ only by return type.

Following is the example where same function **print()** is being used to print different data types:

Function Overloading

A single function name can be used to perform different types of tasks. The same function name can be used to handle different number and different types of arguments. This is known as function overloading or function polymorphism.

```
#include<iostream.h>
#include<conio.h>
void swap(int &x,int &y)
{
int t;
t=x;
x=y;
y=t;
}
void swap(float &p,float &q)
{
float t;
t=p;
p=q;
q=t;
}
void swap(char &c1,char &c2)
{
char t;
t=c1;
c1=c2;
c2=t;
}
void main()
{
int i,j;
float a,b;
```

```
char s1,s2;
clrscr();
cout<<"\n Enter two integers : \n";
cout<<" i = ";
cin>>i;
cout<<"\n j = ";
cin>>j;
swap(i,j);
cout<<"\n Enter two float numbers : \n";
cout<<" a = ";
cin>>a;
cout<<"\n b = ";
cin>>b;
swap(a,b);
cout<<"\n Enter two Characters : \n";
cout<<" s1 = ";
cin>>s1;
cout<<"\n s2 = ";
cin>>s2;
swap(s1,s2);
cout<<"\n After Swapping \n";
cout<<" \n Integers i = "<<i<<"\t j =  "<<j;
cout<<" \n Float Numbers a= "<<a<<"\t b =  "<<b;
cout<<" \n Characters s1 = "<<s1<<"\t s2 =  "<<s2;
getch();
}
```

Operators overloading in C++:

You can redefine or overload most of the built-in operators available in C++. Thus a programmer can use operators with user-defined types as well.

Overloaded operators are functions with special names the keyword operator followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list.

```
Box operator+(const Box&);
```

declares the addition operator that can be used to **add** two Box objects and returns final Box object. Most overloaded operators may be defined as ordinary non-member functions or as class member functions. In case we define above function as non-member function of a class then we would have to pass two arguments for each operand as follows:

```
Box operator+(const Box&, const Box&);
```

Following is the example to show the concept of operator over loading using a member function. Here an object is passed as an argument whose properties will be accessed using this object, the object which will call this operator can be accessed using **this** operator as explained below:

```
#include <iostream>
using namespace std;

class Box
{
  public:

    double getVolume(void)
    {
      return length * breadth * height;
    }
    void setLength( double len )
    {
       length = len;
    }

    void setBreadth( double bre )
    {
       breadth = bre;
    }

    void setHeight( double hei )
    {
       height = hei;
    }
    // Overload + operator to add two Box objects.
    Box operator+(const Box& b)
    {
      Box box;
      box.length = this->length + b.length;
      box.breadth = this->breadth + b.breadth;
      box.height = this->height + b.height;
      return box;
    }
  private:
    double length;      // Length of a box
```

```
    double breadth;     // Breadth of a box
    double height;      // Height of a box
};
// Main function for the program
int main( )
{
  Box Box1;               // Declare Box1 of type Box
  Box Box2;               // Declare Box2 of type Box
  Box Box3;               // Declare Box3 of type Box
  double volume = 0.0;     // Store the volume of a box here

  // box 1 specification
  Box1.setLength(6.0);
  Box1.setBreadth(7.0);
  Box1.setHeight(5.0);

  // box 2 specification
  Box2.setLength(12.0);
  Box2.setBreadth(13.0);
  Box2.setHeight(10.0);

  // volume of box 1
  volume = Box1.getVolume();
  cout << "Volume of Box1 : " << volume <<endl;

  // volume of box 2
  volume = Box2.getVolume();
  cout << "Volume of Box2 : " << volume <<endl;

  // Add two object as follows:
  Box3 = Box1 + Box2;

  // volume of box 3
  volume = Box3.getVolume();
  cout << "Volume of Box3 : " << volume <<endl;

  return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Volume of Box1 : 210
```

Volume of Box2 : 1560
Volume of Box3 : 5400

Overloadable/Non-overloadableOperators:

Following is the list of operators which can be overloaded:

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

Following is the list of operators, which can not be overloaded:

:: .* . ?:

UNARY OPERATOR OVERLOADING

The unary operators operate on a single operand and following are the examples of Unary operators:

- The increment (++) and decrement (--) operators.
- The unary minus (-) operator.
- The logical not (!) operator.

The unary operators operate on the object for which they were called and normally, this operator appears on the left side of the object, as in !obj, -obj, and ++obj but sometime they can be used as postfix as well like obj++ or obj--.

Following example explain how minus (-) operator can be overloaded for prefix as well as postfix usage.

```
#include <iostream>
using namespace std;

class Distance
{
  private:
    int feet;             // 0 to infinite
    int inches;           // 0 to 12
  public:
    // required constructors
    Distance(){
      feet = 0;
      inches = 0;
    }
```

```
    Distance(int f, int i){
      feet = f;
      inches = i;
    }
    // method to display distance
    void displayDistance()
    {
      cout << "F: " << feet << " I:" << inches <<endl;
    }
    // overloaded minus (-) operator
    Distance operator- ()
    {
      feet = -feet;
      inches = -inches;
      return Distance(feet, inches);
    }
};
int main()
{
  Distance D1(11, 10), D2(-5, 11);

  -D1;                  // apply negation
  D1.displayDistance();  // display D1

  -D2;                  // apply negation
  D2.displayDistance();  // display D2

  return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
F: -11 I:-10
F: 5 I:-11
```

BINARY OPERTOR OVERLOADING

the unary operators take two arguments and following are the examples of Binary operators. You use binary operators very frequently like addition (+) operator, subtraction (-) operator and division (/) operator.

Following example explains how addition (+) operator can be overloaded. Similar way, you can overload subtraction (-) and division (/) operators.

```
#include<iostream.h>
#include<conio.h>
```

```
class complex
{
        int a,b;
   public:
        void getvalue()
        {
          cout<<"Enter the value of Complex Numbers a,b:";
          cin>>a>>b;
        }
        complex operator+(complex ob)
        {
                complex t;
                t.a=a+ob.a;
                t.b=b+ob.b;
                return(t);
        }
        complex operator-(complex ob)
        {
                complex t;
                t.a=a-ob.a;
                t.b=b-ob.b;
                return(t);
        }
        void display()
        {
                cout<<a<<"+"<<b<<"i"<<"\n";
        }
};

void main()
{
  clrscr();
  complex obj1,obj2,result,result1;

  obj1.getvalue();
  obj2.getvalue();

  result = obj1+obj2;
  result1=obj1-obj2;
```

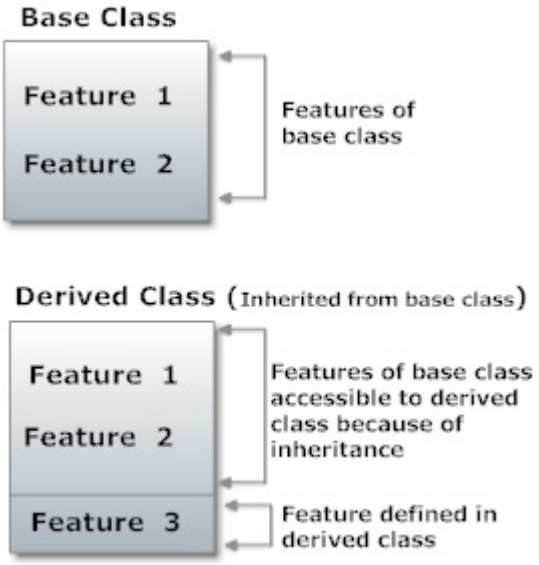
```
  cout<<"Input Values:\n";
  obj1.display();
  obj2.display();

  cout<<"Result:";
  result.display();
 result1.display();

  getch();
}
```

INHERITANCE

Inheritance is one of the key feature of object-oriented programming including C++ which allows user to create a new class(derived class) from a existing class(base class). The derived class inherits all feature from a base class and it can have additional features of its own.



Concept of Inheritance in OOP

Suppose, you want to calculate either area, perimeter or diagonal length of a rectangle by taking data(length and breadth) from user. You can create three different objects(*Area*, *Perimeter* and *Diagonal*) and asks user to enter length and breadth in each object and calculate corresponding data. But, the better approach would be to create a additional object *Rectangle* to store value of length and breadth from user and derive objects *Area*, *Perimeter* and *Diagonal* from *Rectangle* base class. It is because, all three objects *Area*, *Perimeter* and *diagonal* are related to object *Rectangle* and you don't need to ask user the input data from these three derived objects as this feature is included in base class.

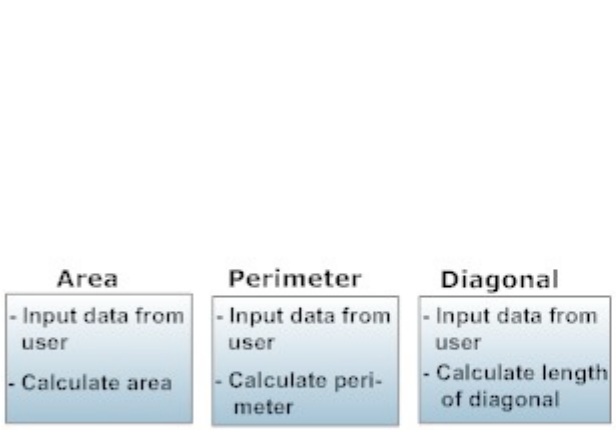


Figure: Visualization of problem without using Inheritance

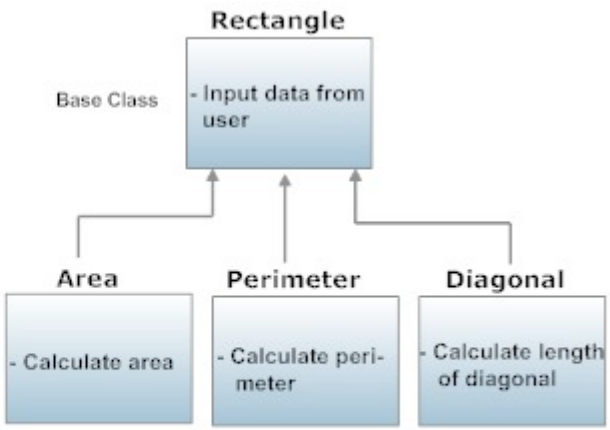


Figure: Visualization of problem using Inheritance

Note: Arrow is pointing from derived from to base class

Implementation of Inheritance in C++ Programming

```
class Rectangle
{
  ... .. ...
};

class Area : public Rectangle
{
  ... .. ...
};

class Perimeter : public Rectangle
{
  .... .. ...
};
```

In the above example, class *Rectangle* is a base class and classes *Area* and *Perimeter* are the derived from *Rectangle*. The derived class appears with the declaration of class followed by a colon, the keyword *public* and the name of base class from which it is derived.

Since, *Area* and *Perimeter* are derived from *Rectangle, all data member and member function of base class Rectangle can be accessible from derived class.*

Note: Keywords private and protected can be used in place of public while defining derived class(will be discussed later).

Source Code to Implement Inheritance in C++ Programming

This example calculates the area and perimeter a rectangle using the concept of inheritance.

```
/* C++ Program to calculate the area and perimeter of rectangles using concept of
inheritance. */

#include <iostream>
using namespace std;
class Rectangle
{
   protected:
      float length, breadth;
   public:
       Rectangle(): length(0.0), breadth(0.0)
       {
          cout<<"Enter length: ";
          cin>>length;
          cout<<"Enter breadth: ";
          cin>>breadth;
       }

};

/* Area class is derived from base class Rectangle. */
class Area : public Rectangle
{
   public:
      float calc()
        {
           return length*breadth;
        }

};

/* Perimeter class is derived from base class Rectangle. */
class Perimeter : public Rectangle
{
   public:
      float calc()
        {
           return 2*(length+breadth);
        }
};
```

```
int main()
{
    cout<<"Enter data for first rectangle to find area.\n";
    Area a;
    cout<<"Area = "<<a.calc()<<" square meter\n\n";

    cout<<"Enter data for second rectangle to find perimeter.\n";
    Perimeter p;
    cout<<"\nPerimeter = "<<p.calc()<<" meter";
    return 0;
}
```

Output

```
Enter data for first rectangle to find area.
Enter length: 5
Enter breadth: 4
Area = 20 square meter

Enter data for second rectangle to find perimeter.
Enter length: 3
Enter breadth: 2
Area = 10 meter
```

Explanation of Program

In this program, classes *Area* and *Perimeter* are derived from class *Rectangle*. Thus, the object of derived class can access the public members of *Rectangle*. In this program, when objects of class *Area* and *Perimeter* are created, constructor in base class is automatically called. If there was public member function in base class then, those functions also would have been accessible for objects *a* and *p*.

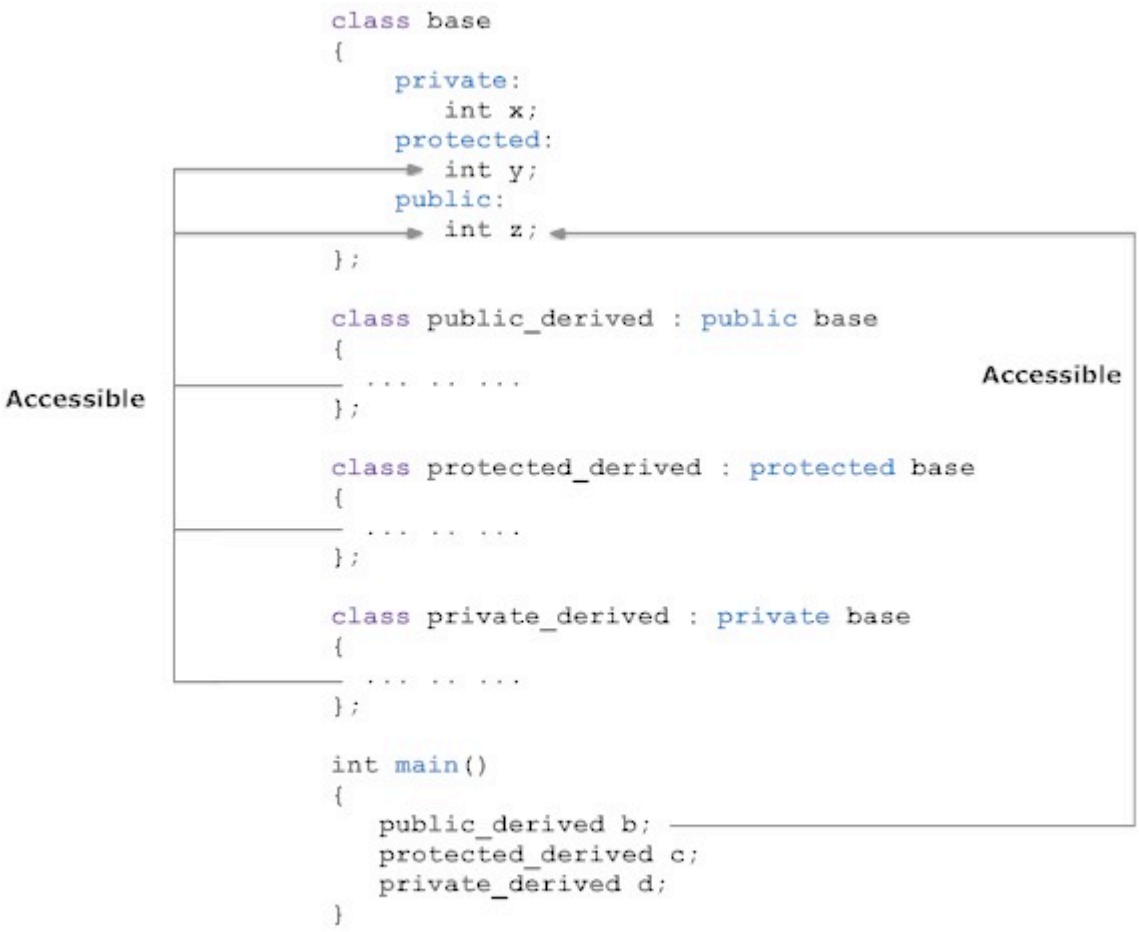
Keyword protected

In this program, *length* and *breadth* in the base class are protected data members. These data members are accessible from the derived class but, not accessible from outside it. This maintains the feature of data hiding in C++ programming. If you defined *length* and *breadth* as private members then, those two data are not accessible to derived class and if defined as public members, it can be accessible from both derived class and from main( ) function.

Accessbility	private	protected	public
Accessible from own class ?	yes	yes	yes
Accessible from dervied class ?	no	yes	yes
Accessible outside dervied class ?	no	no	yes

Things to remember while Using Public, Protected and Private Inheritance

1. Protected and public members(data and function) of a base class are accessible from a derived class(for all three: public, protected and private inheritance).
2. Objects of derived class with private and protected inheritance cannot access any data member of a base class.
3. Objects of derived class with public inheritance can access only public member of a base class.



This diagram shows all the access combination for public, private and protected inheritance. The arrow shows the position from where that particular member can be accessed.

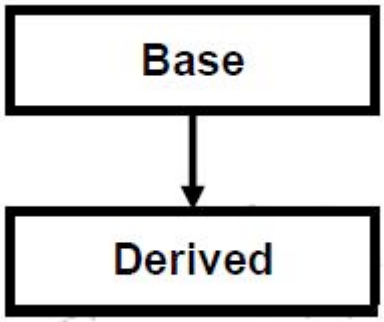
<u>Inheritance can be classified to 5 types.</u>

1. Single Inheritance
2. Hierarchical Inheritance
3. Multi Level Inheritance
4. Hybrid Inheritance

5. Multiple Inheritance

1. Single Inheritance

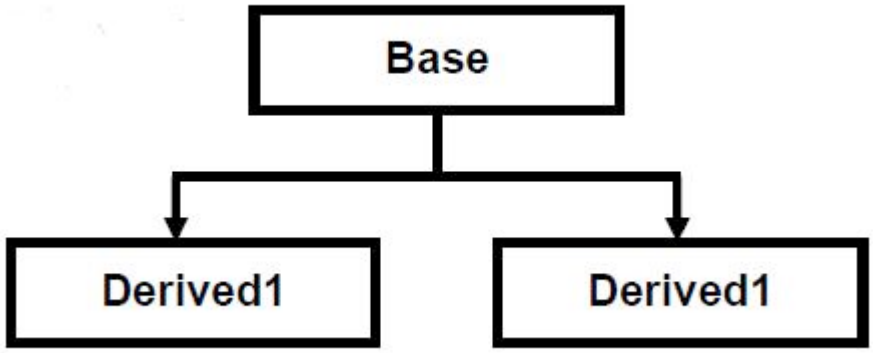
when a single derived class is created from a single base class then the inheritance is called as single inheritance.



Single Inheritance Example Program

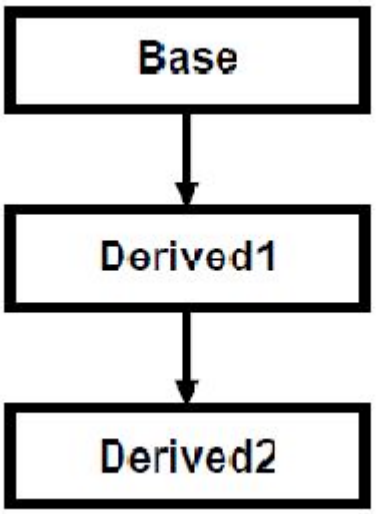
2. Hierarchical Inheritance

when more than one derived class are created from a single base class, then that inheritance is called as hierarchical inheritance.



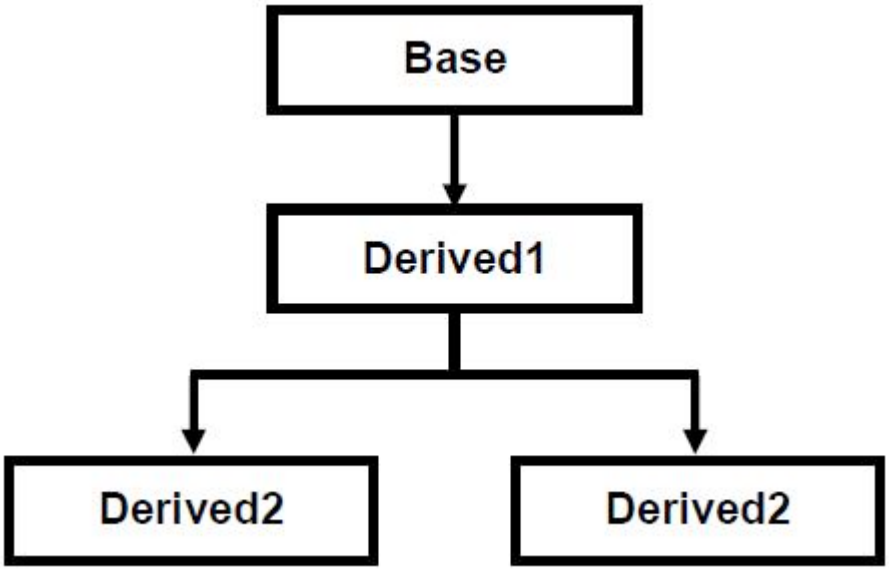
3. Multi Level Inheritance

when a derived class is created from another derived class, then that inheritance is called as multi level inheritance.



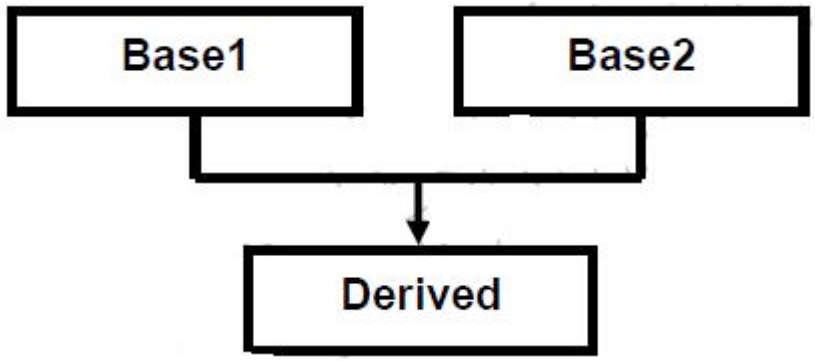
4. Hybrid Inheritance

Any combination of single, hierarchical and multi level inheritances is called as hybrid inheritance.



5. Multiple Inheritance

when a derived class is created from more than one base class then that inheritance is called as multiple inheritance. But multiple inheritance is not supported by .net using classes and can be done using interfaces.



MULTIPLE INHERITANCE

```
#include<iostream.h>
#include<conio.h>
class add
{
protected:
int val;
public:
void sum(int a,int b)
{
val=a+b;
}
};
class sub
{
protected:
int res;
public:
void minus(int a,int b)
{
res=a-b;
}
};
class mul:public add,public sub
{
```

```
private:
int prod;
public:
void display()
{
cout<<"\n OUTPUT";
cout<<"\n~~~~~~~~";
cout<<"\n\n Added Value = "<<val;
cout<<"\n\n Subtracted Value = "<<res;
prod=val*res;
cout<<"\n\n Product = "<<prod;
}
};
void main()
{
clrscr();
int x,y;
mul s;
cout<<"\n Enter 2 numbers : \n";
cin>>x>>y;
s.sum(x,y);
s.minus(x,y);
s.display();
getch();
   }
```

Virtual Function

The two types of polymorphism are,

• Compile time polymorphism – The compiler selects the appropriate function for a particular call at the compile time itself. It can be achieved by function overloading and operator overloading.

• Run time Polymorphism - The compiler selects the appropriate function for a particular call at the run time only. It can be achieved using virtual functions

Program to implement runtime polymorphism:

```
   #include<iostream.h>
   #include<conio.h>
   class squareint
   {
   public:
   int s;
```

```
squareint()
{
}
squareint(int a)
{
s=a*a;
}
virtual void display()
{
cout<<"\nSquare of an integer = "<<s;
}
};
class squareflt:public squareint
{
public:
float s1;
squareflt(int a,float b):squareint(a)
{
s1=b*b;
}
void display()
{
cout<<"\n Square of a float number ="<<s1;
}
};
void main()
{
int x;
float y;
clrscr();
cout<<"\n Enter an Integer : ";
cin>>x;
cout<<"\n Enter a float : ";
cin>>y;
squareflt obj(x,y);
squareint obj1;
squareint *p;
obj1=obj;
p=&obj1;
p->display();
```

```
p=&obj;
p->display();
getch();
}
```

An ambiguity can arise when several paths exist to a class from the same base class. This means that a child class could have duplicate sets of members inherited from a single base class.
C++ solves this issue by introducing a virtual base class. When a class is made virtual, necessary care is taken so that the duplication is avoided regardless of the number of paths that exist to the child class.

C++ - What is a virtual base class? -

What is Virtual base class? Explain its uses.

When two or more objects are derived from a common base class, we can prevent multiple copies of the base class being present in an object derived from those objects by declaring the base class as virtual when it is being inherited. Such a base class is known as virtual base class. This can be achieved by preceding the base class' name with the word virtual.
Consider following example:

```
class A
{
    public:
        int i;
};
class B : virtual public A
{
     public:
            int j;
};
class C: virtual public A
{
      public:
          int k;
};
class D: public B, public C
{
public:
     int sum;
};
```

```
int main()
{
        D ob;
        ob.i = 10; //unambiguous since only one copy of i is inherited.
        ob.j = 20;
        ob.k = 30;
        ob.sum = ob.i + ob.j + ob.k;
        cout << "Value of i is : "<< ob.i<<"\n";
        cout << "Value of j is : "<< ob.j<<"\n"; cout << "Value of k is :"<<
ob.k<<"\n";
        cout << "Sum is : "<< ob.sum <<"\n";

return 0;
}.
```

Dynamic memory ALLOCATION

In the programs seen in previous chapters, all memory needs were determined before program execution by defining the variables needed. But there may be cases where the memory needs of a program can only be determined during runtime. For example, when the memory needed depends on user input. On these cases, programs need to dynamically allocate memory, for which the C++ language integrates the operators new and delete.

Operators new and new[]

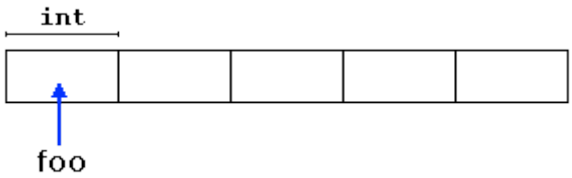
Dynamic memory is allocated using operator new. new is followed by a data type specifier and, if a sequence of more than one element is required, the number of these within brackets []. It returns a pointer to the beginning of the new block of memory allocated. Its syntax is:

```
pointer = new type
pointer = new type [number_of_elements]
```

The first expression is used to allocate memory to contain one single element of type type. The second one is used to allocate a block (an array) of elements of type type, where number_of_elements is an integer value representing the amount of these. For example:

```
1 int * foo;
2 foo = new int [5];
```

In this case, the system dynamically allocates space for five elements of type int and returns a pointer to the first element of the sequence, which is assigned to foo (a pointer). Therefore, foo now points to a valid block of memory with space for five elements of type int.



Here, foo is a pointer, and thus, the first element pointed to by foo can be accessed either with the expression foo[0] or the expression *foo (both are equivalent). The second element can be accessed either with foo[1] or *(foo+1), and so on...

There is a substantial difference between declaring a normal array and allocating dynamic memory for a block of memory using new. The most important difference is that the size of a regular array needs to be a *constant expression*, and thus its size has to be determined at the moment of designing the program, before it is run, whereas the dynamic memory allocation performed by new allows to assign memory during runtime using any variable value as size.

The dynamic memory requested by our program is allocated by the system from the memory heap. However, computer memory is a limited resource, and it can be exhausted. Therefore, there are no guarantees that all requests to allocate memory using operator new are going to be granted by the system.

C++ provides two standard mechanisms to check if the allocation was successful:

One is by handling exceptions. Using this method, an exception of type bad_alloc is thrown when the allocation fails. Exceptions are a powerful C++ feature explained later in these tutorials. But for now, you should know that if this exception is thrown and it is not handled by a specific handler, the program execution is terminated.

This exception method is the method used by default by new, and is the one used in a declaration like:

```
foo = new int [5];  // if allocation fails, an exception is thrown
```

The other method is known as nothrow, and what happens when it is used is that when a memory allocation fails, instead of throwing a bad_alloc exception or terminating the program, the pointer returned by new is a *null pointer*, and the program continues its execution normally.

This method can be specified by using a special object called nothrow, declared in header <new>, as argument for new:

```
foo = new (nothrow) int [5];
```

In this case, if the allocation of this block of memory fails, the failure can be detected by checking if foo is a null pointer:

```
1 int * foo;
2 foo = new (nothrow) int [5];
3 if (foo == nullptr) {
4   // error assigning memory. Take measures.
5 }
```

This nothrow method is likely to produce less efficient code than exceptions, since it implies explicitly checking the pointer value returned after each and every allocation. Therefore, the exception mechanism is generally preferred, at least for critical allocations. Still, most of the coming examples will use the nothrow mechanism due to its simplicity.

Operators delete and delete[]

In most cases, memory allocated dynamically is only needed during specific periods of time within a program; once it is no longer needed, it can be freed so that the memory becomes available again for other requests of dynamic memory. This is the purpose of operator delete, whose syntax is:

```
1 delete pointer;
2 delete[] pointer;
```

The first statement releases the memory of a single element allocated using new, and the second one releases the memory allocated for arrays of elements using new

and a size in brackets ([]).

The value passed as argument to delete shall be either a pointer to a memory block previously allocated with new, or a *null pointer* (in the case of a *null pointer*, delete produces no effect).

```
   // rememb-o-matic
1  #include <iostream>
2  #include <new>
3  using namespace std;
4
5  int main ()
6  {
7    int i,n;
8    int * p;
9    cout << "How many numbers would
10 you like to type? ";
11   cin >> i;
12   p= new (nothrow) int[i];
13   if (p == nullptr)
14     cout << "Error: memory could not
15 be allocated";
16   else
17   {
18     for (n=0; n<i; n++)
19     {
20       cout << "Enter number: ";
21       cin >> p[n];
22     }
23     cout << "You have entered: ";
24     for (n=0; n<i; n++)
25       cout << p[n] << ", ";
26     delete[] p;
27   }
28   return 0;
   }
```

```
How many numbers would
you like to type? 5
Enter number : 75
Enter number : 436
Enter number : 1067
Enter number : 8
Enter number : 32
You have entered: 75, 436,
1067, 8, 32,
```

Edit & Run

Notice how the value within brackets in the new statement is a variable value entered by the user (i), not a constant expression:

p= *new* (nothrow) *int*[i];

There always exists the possibility that the user introduces a value for i so big that the system cannot allocate enough memory for it. For example, when I tried to give a value of 1 billion to the "How many numbers" question, my system could not allocate that much memory for the program, and I got the text message we prepared for this case (Error: memory could not be allocated).

Nested class is a class defined inside a class, that can be used within the scope of the class in which it is defined. In C++ nested classes are not given importance because of the strong and flexible usage of inheritance. Its objects are accessed using "Nest::Display".

Example:

```
#include <iostream.h>
class Nest
 {
   public:
    class Display
     {
      private:
       int s;
      public:
       void sum( int a, int b)
         {  s =a+b; }
       void show( )
         { cout << "\nSum of a and b is:: " << s;}
     };
 };
void main()
 {
    Nest::Display x;
    x.sum(12, 10);
    x.show();
 }
```

Result:

Sum of a and b is::22

In the above example, the nested class "Display" is given as "public" member of the class "Nest".

Local class is a class defined inside a function. Following are some of the rules for using these classes.

- Global variables declared above the function can be used with the scope operator "::".
- Static variables declared inside the function can also be used.
- Automatic local variables cannot be used.
- It cannot have static data member objects.
- Member functions must be defined inside the local classes.
- Enclosing functions cannot access the private member objects of a local class.

Example:

```
#include <iostream.h>
int y;
void g();
int main()
 {
        g();
        return 0;
 }
void g()
{
 class local
   {
    public:
      void put( int n) {::y=n;}
      int get() {return ::y;}
     }ab;
  ab.put(20);
  cout << "The value assigned to y is::"<< ab.get();
 }
```

The value assigned to y is::20

In the above example, the local class "local" uses the variable "y" which is declared globally. Inside the function it is used using the "::" operator. The object "ab" is used to set, get the assigned values in the local class.