The use of objects in an OOP defines the way programs are designed and written. Classes are a software construct that can be used to emulate a real world object. Any real world object can be defined by its attributes and by its actions. For instance, a cat has attributes such as its age, weight and color. It also has a set of actions that it can perform such as sleep, eat and complain (meow). Obviously, no software construct can perfectly emulate a real object, but the goal in designing classes is to have all the relevant attributes and actions wrapped by the class. This way, objects of a class are easily created and used.

An Object has the similar relationship to a class that a variable has to a fundamental (built-in) data type. A class may contain data as well as functions. Contents of a class are known as **members** of the class; data declared inside the class are known as **data members** and functions included inside the class are known as **member functions**. Data members of a class are also known as **attributes** and member functions are called **methods**.

Data members (Attributes)

Member functions (Methods)

Class

Any member of a class has a visibility label. A member of a class can be **public** or **protected** or **private**. By default (if no visibility label is specified) members of the class are assigned visibility label **private**. A **public** member can be used inside (inside the member function of a class) as well outside the class. Where as **private** and **protected** members can only be used inside the member functions of a class. Any member function of a class may access any data member or other member function of the same class. An object is said to be an instance of a class, in the same way **man** and **woman** are instances of a **human**. Properties associated with an object signify the **state** of that object. The operations (functions or methods) associated with the object exhibit the **behaviour** of the object.

A class is declared or defined by using the keyword **class** followed by a programmer-specified name (class name is an identifier name) followed by the class declaration (definition) within pair of curly braces. Closing curly brace is terminated by semi-colon (;). A class declaration (definition) contains the class members - its data members and its member functions. An example of a general class declaration is given below:

**class** ClassName

{

**private**:

PriDataMember1, PriDataMember2, …

PriMemFunction1()**,**PriMemFunction2(), …

**protected:**

ProDataMember1, ProDataMember2, …

ProMemFunction1()**,**ProMemFunction2(), …

**public:**

PubDataMember1, PubDataMember2, …

PubMemFunction1()**,**PubMemFunction2(), …

};

**void** main()

{

ClassName ObjectName1, ObjectName2,…

//More C++ statements

}

A programming example is given below.

Class name is **student**. Data members **roll**, **name** and **fees** are declared under the visibility label **private**. Member functions **input**(), **display**(), **retroll**() and **setfees**() are declared under the visibility label **public**. If the visibility label **public** is removed then all the members of the class (data and functions) would become **private** members of class. Declaring a class with only **private** members is syntactically correct but this type of classes are of no use. It is just like building a room without either a door or a window. Note that the member functions of **student** are defined inside the class.

**class** student

{

**private:**

**int** roll;

**char** name[20];

**double** fees;

**public:**

**void** input()

{

cout<<"Roll? "; cin>>roll;

cout<<"Name? "; gets(name);

cout<<"Fees? "; cin>>fees;

}

**void** display()

{

cout<<"Roll= "<<roll<<endl;

cout<<"Name= "<<name<<endl;

cout<<"Fees= "<<fees<<endl;

}

**int** retroll() { **return** roll; }

**void** setfees(**double** amt) { fees=amt; }

}**;**

Member functions of class can be defined outside the class also. First member functions are declared inside class and then member functions are defined outside the class. When a member function is defined outside the class, it must be identified as belonging to a particular class. This is done with the class name and the scope resolution operator (**::**). The general rule for defining a member function outside class is:

**DataType CName::FName()**

**{**

**//C++ Code**

**}**

**DataType** is the return value of the function, **CName** is the class name and **FName** is the member function name. Scope resolution operator is used as a binary operator. It is also called membership operator of a class.

**class** student

{

**int** roll;

**char** name[20];

**double** fees;

**public:**

**void** input();

**void** display();

**int** retroll();

**void** setfees(**double**);

}**;**

**void** student::input()

{

cout<<"Roll? "; cin>>roll;

cout<<"Name? "; gets(name);

cout<<"Fees? "; cin>>fees;

}

**void** student::display()

{

cout<<"Roll= "<<roll<<endl;

cout<<"Name= "<<name<<endl;

cout<<"Fees= "<<fees<<endl;

}

**int** student::retroll() { **return** roll; }

**void** student::setfees(**double** amt) { fees=amt; }

**void** main()

**stuobj** is an instance of **student**. Member function **input**() takes input from keyboard for the data members of the object **stuobj**. Member function **display**() displays values stored in data members of the object **stuobj**. Member function **retroll**() returns value stored in data member **roll**. Member function **setfees**() updates value stored in data member **fees**.

{

student stuobj;

stuobj.input();

stuobj.display();

cout<<"Roll Number="

<<stuobj.retroll()

<<endl;

stuobj.setfees(900);

stuobj.display();

}

**Some key concept related to classes and objects**:

1. Memory is allocated to object but not to a class. No memory is allocated to class student because student is a data type. Memory will be allocated to an object stuobj – an instance of student (a variable of the type student). Just like an array variable or a structure variable, memory is allocated contiguously to an object stuobj which is an instance of student.
2. Private members can only be accessed through member function of the same class. The keyword **private** indicates that the three data members, roll, name and fees, cannot be accessed directly from outside the class student.
3. Technique of accessing and manipulation of **private** data members only through member functions of the same class is referred to as **data** **hiding**.
4. Public members can be accessed from anywhere in the program. The keyword **public** indicates that the four member functions, input(), display(), retroll() and setfees() can be called from code outside of the class. That is, they may be called (accessed) from other parts of a program using objects of the class student.
5. Member function retroll() is called an **access** **function**. An access function is a **public** member function of a class returning value stored in a **private** data member of the class.
6. To access the **public** members of an object we use dot (.) operator. Dot (.) is a membership operator of an object and it is a binary operator. General rule to access a **public** member of an object is:

ObjectName.PublicMemberName

Examples are given below:

stuobj.input();

stuobj.display();

cout<<stuobj.retroll()<<endl;

stuobj.setfees(900);

stuobj.display();

1. To store values in the object stuobj, member function input() is invoked. Member function display(), displays values stored in the object stuobj on the screen.
2. **Private** and **protected** members of a class cannot be used outside the class. If a program tries to access **private** or **protected** members from outside the class, then compiler will flag a syntax error. To access or update **private** or **protected** data members from outside the class, **public** member functions are to be used. For example, in class student **public** member function retroll() returns the value stored in the **private** data member roll. Also **public** member function setfees() assigns new value to the **private** data member fees.
3. The size of an object, calculated in term of bytes – is the sum total of bytes allocated to the data members of the object. For example, the object stuobj has three data members, that is, **int** roll, **char** name[20] and **double** fees. Data type int is allocated 4 bytes, an array of 20 characters is allocated 20 bytes and data type **double** is allocated 8 bytes. They all add up to 32 bytes. So an instance of student will be allocated 32 bytes. **Member functions do not add to the memory allocated to an object**.

**class** student

Kindly note that data members **int** **roll**, **char** **name**[20] and **double** **fees** are declared without any visibility label since default visibility label of a member of a class is **private**.

{

**int** roll;

**char** name[20];

**double** fees;

**public:**

**void** input()

{

cout<<"Roll? "; cin>>roll;

cout<<"Name? "; gets(name);

cout<<"Fees? "; cin>>fees;

}

**void** display()

{

cout<<"Roll= "<<roll<<endl;

cout<<"Name= "<<name<<endl;

cout<<"Fees= "<<fees<<endl;

}

**int** retroll() { **return** roll; }

**void** setfees(**double** amt) { fees=amt; }

}**;**

**void** main()

{

student s1, s2;

s1.input();

s2.input();

s1.display();

s2.display();

}

Objects s1 and s2 will have roll, name and fees as data members. Object s1 and s2 will be allocated two different memory locations of 32 bytes each to store values for their respective data members. But a very interesting point to be noted is that both the objects s1 and s2 (there could more objects belonging to the same class) shares the same member functions. That is, there will be only one copy member functions in the computer’s main storage. So the question that arises is how does the function invocation of s1.input() know that it has to access and store values in s1.roll, s1.name and s1.fees? Or how does function s2.display() know that it has to display s2.roll, s2.name and s2.fees? There is an **implicit** **pointer** called **this** pointer which stores the address of the object when a member function of an object is invoked. That is when s1.input() is invoked then **this** pointer stores address of object s1 and s1.roll becomes **this**->roll, s1.name becomes **this**->name and s1.fees becomes **this**->fees. Inside the member function s1.input() code changes from:

**void** input()

Inside the function **s1**.**input**(), **roll** is replaced by **this->roll**, **name** is replaced by **this->name** and **fees** is replaced by **this->fees**. So when **s1**.**input**() is invoked, **this->roll**, **this->name** and **this->fees** represent **s1**.**roll**, **s1**.**name** and **s1**.**fees** respectively.

{

cout<<"Roll? "; cin>>roll;

cout<<"Name? "; gets(name);

cout<<"Fees? "; cin>>fees;

}

To:

**void** input()

{

cout<<"Roll? "; cin>>**this**->roll;

cout<<"Name? "; gets(**this**->name);

cout<<"Fees? "; cin>>**this**->fees;

}

**Class**: A class is a user defined data type which is used to represent objects of one type.

**Object**: An instance of a class is called an object. Also variable of the type class can also be called an object.

**Encapsulation:** Wrapping up of data members and associated member functions in a single unit is known as encapsulation. In a class, we pack the data and member functions together as a capsule.

**Data Hiding:** Keeping the data in **private** area or **protected** area of a class to prevent it from accidental modification (change) from outside the class is known as data hiding. **Private** or **protected** data members can only be inputted, updated or accessed by using member functions of the same class. Any attempt to use **private** or **protected** data members outside the class will be flagged as syntax error. An example is given below:

Syntax error since **private** members of a class cannot be used with an object (cannot be used outside the class). But if the visibility labels of data members are changed from **private** (**protected**) to **public**, then all the statements will treated as valid by the compiler.

student obj;

cout<<"Roll? "; cin>>obj.roll;

cout<<"Name? "; gets(obj.name);

cout<<"Fees? "; cin>>obj.fees;

cout<<"Roll= "<<obj.roll<<endl;

cout<<"Name= "<<obj.name<<endl;

cout<<"Fees= "<<obj.fees<<endl;

Given below is an example of incorrect class declaration:

Syntax error since class **student** is not a variable so no values can be assigned to data members of class **student**. Kindly note that to save space member functions are declared only.

**class** student

{

**int** roll=21;

**char** name[20]="Atul Gupta";

**double** fees=900;

**public:**

**void** input();

**void** display();

};

In a class, data members are either **private** or **protected** and member functions are **public**, but it is always possible to declare data members which are **public** and member functions which are **private** or **protected**. An example is given below:

Class student has 3 **private** members: data members **tot**, **avg** and member function **calc**(). It has 5 **public** members: data members **roll**, **name**, **marks** and member functions **input**() and **display**().

**class** student

{

**double** tot, avg;

**void** calc()

{

tot=marks[0]+marks[1]+marks[2]+marks[3]+marks[4];

avg=tot/5;

}

**public**:

**int** roll;

**char** name[20];

**double** marks[5];

**void** input();

**void** display();

};

**void** student::input()

{

cout<<"Roll? "; cin>>roll;

cout<<"Name? "; gets(name);

**for** (**int** k=0; k<5; k++)

{

cout<<"Marks of Subject "<<(k+1)<<" ?"; cin>>marks[k];

}

}

**void** student::display()

{

calc();

cout<<"Roll="<<roll<<endl;

cout<<"Name="<<name<<endl;

**for** (**int** k=0; k<5; k++)

cout<<"Marks of Subject "<<(k+1)<<"="<<marks[k]<<endl;

cout<<"Total Marks ="<<tot<<endl;

cout<<"Average Marks="<<avg<<endl;

}

**Public** data members **roll**, **name** and **marks** can be used inside the class as well as outside the class. Since **calc**() is a **private** member function of class **student**, it can only invoked from a **public** member function of **student**.

**void** main()

{

student s;

s.input();

s.display();

cout<<"Roll? "; cin>>s.roll;

cout<<"Name? "; gets(s.name);

**for** (**int** k=0; k<5; k++)

{

cout<<"Marks of Subject "<<(k+1)<<" ?"; cin>>s.marks[k];

}

s.display();

}

Running of the program produces following output:

Roll? 21

Input of **roll**, **name** and **marks** through **public** member function **input**().

Name? Sumit Gupta

Marks of Subject 1 ?67

Marks of Subject 2 ?78

Marks of Subject 3 ?89

Marks of Subject 4 ?93

Marks of Subject 5 ?75

Roll=21

Calculation of **tot** and **avg**; display of **roll**, **name**, **marks**, **tot** and **avg** through **public** member function **display**().

Name=Sumit Gupta

Marks of Subject 1=67

Marks of Subject 2=78

Marks of Subject 3=89

Marks of Subject 4=93

Marks of Subject 5=75

Total Marks =402

Average Marks=80.4

Roll? 15

Input of **roll**, **name** and **marks** from outside the class **student** (through an object **s**).

Name? Dinesh Kapoor

Marks of Subject 1 ?89

Marks of Subject 2 ?98

Marks of Subject 3 ?93

Marks of Subject 4 ?95

Marks of Subject 5 ?96

Calculation of **tot** and **avg**; display of **roll**, **name**, **marks**, **tot** and **avg** through **public** member function **display**().

Roll=15

Name=Dinesh Kapoor

Marks of Subject 1=89

Marks of Subject 2=98

Marks of Subject 3=93

Marks of Subject 4=95

Marks of Subject 5=96

Total Marks =471

Average Marks=94.2

A class is declared before the main() function as a global identifier so that the same class declaration can be used to create objects in the main() function and in any other functions in the program. An example is given below:

**class** teacher

Class **teacher** is declared globally. Generally a class is declared as a global identifier so that it can be used throughout the program.

{

**char** name[20];

**char** subject[20];

**double** salary;

**public:**

**void** tinput()

{

cout<<"Full Name? "; gets(name);

cout<<"Subject ? "; gets(subject)

cout<<"Salary ? "; cin>>salary

}

**void** tdisplay()

{

cout<<"Full Name="<<name<<endl;

cout<<"Subject ="<<subject<<endl;

cout<<"Salary ="<<salary<<endl;

}

}**;**

**void** funct()

Global class **teacher** creates two local objects **a** and **b** inside the function **funct**().

{

teacher a, b;

//more C++ statements

}

**void** main()

Global class **teacher** creates two local objects **t1** and **t2** inside the **main**() function.

{

teacher t1, t2;

//more C++ statements

}

But a class can be declared locally inside a block. In that case, class declaration can create instance of that class inside that particular block and blocks nested below. An example is given below:

**void** main()

Class **teacher** is declared locally inside the **main**() function. Class **teacher** can be used to create objects inside **main**() function and block nested below **main**() function.

{

**class** teacher

{

**char** name[20];

**char** subject[20];

**double** salary;

**public:**

**void** tinput()

{

cout<<"Full Name? "; gets(name);

cout<<"Subject ? "; gets(subject)

cout<<"Salary ? "; cin>>salary

}

**void** tdisplay()

{

cout<<"Full Name="<<name<<endl;

cout<<"Subject ="<<subject<<endl;

cout<<"Salary ="<<salary<<endl;

}

}**;**

teacher a1, a2;

Local class **teacher** creates two local objects **a1** and **a2** inside the **main**() function. It also creates two local objects **b1** and **b2** inside the nested block.

//more C++ statements

//Nested block below

{

teacher b1, b2;

//more C++ statements

}

}

**void** funct()

Syntax error because the class **teacher** is declared locally inside the **main**() function.

{

teacher a, b;

//more C++ statements

}

Normally class declaration starts with keyword **class** followed by a class name; then a block containing all members (data and functions) and block is terminated by a semi-colon. The class name becomes a data type. Class name is used to create objects in the program. But it is possible to create class without a class name. An example is given below:

**class**

Perfectly correct class declaration but is of no use in the program since the class is without any name.

{

**char** name[20], subject[20]; double salary;

**public:**

**void** tinput()

{

cout<<"Full Name? "; gets(name);

cout<<"Subject ? "; gets(subject)

cout<<"Salary ? "; cin>>salary

}

**void** tdisplay()

{

cout<<"Full Name="<<name<<endl;

cout<<"Subject ="<<subject<<endl;

cout<<"Salary ="<<salary<<endl;

}

}**;**

But class declaration without a name (type less class) is of no use in a program unless object(s) is (are) created along with the class declaration. An example is given below:

**class**

A nameless (type less) class is created and at the end of class declaration, objects **t1** and **t2** are also created so that they can used in the program.

{

**char** name[20], subject[20]; **double** salary;

**public:**

**void** tinput()

{

cout<<"Full Name? "; gets(name);

cout<<"Subject ? "; gets(subject)

cout<<"Salary ? "; cin>>salary

}

**void** tdisplay()

{

cout<<"Full Name="<<name<<endl;

cout<<"Subject ="<<subject<<endl;

cout<<"Salary ="<<salary<<endl;

}

} t1, t2;

**Global and Local Objects**: Generally a class is created as a global identifier but instances of classes are created locally (inside a block). In all our previous examples, objects are created locally. Using a global variable goes against the paradigm of Object Oriented Programming. But that does not stop a programmer to create global object. Some examples are given below showing how to create global object.

Example #1:

**class** teacher

{

**char** name[20], subject[20];

**double** salary;

**public:**

**void** tinput()

{

cout<<"Full Name? "; gets(name);

cout<<"Subject ? "; gets(subject);

cout<<"Salary ? "; cin>>salary;

}

**void** tdisplay()

{

cout<<"Full Name="<<name<<endl;

cout<<"Subject ="<<subject<<endl;

cout<<"Salary ="<<salary<<endl;

}

};

teacher t1;

**void** main()

{

teacher t2;

//more C++ statements

}

Example #2:

**class** teacher

{

**char** name[20], subject[20];

**double** salary;

**public:**

**void** tinput()

{

cout<<"Full Name? "; gets(name);

cout<<"Subject ? "; gets(subject)

cout<<"Salary ? "; cin>>salary

}

**void** tdisplay()

{

cout<<"Full Name="<<name<<endl;

cout<<"Subject ="<<subject<<endl;

cout<<"Salary ="<<salary<<endl;

}

} t1;

**void** main()

{

teacher t2;

//more C++ statements

}

Example #3:

**class**

{

**char** name[20], subject[20];

**double** salary;

**public:**

**void** tinput()

{

cout<<"Full Name? "; gets(name);

cout<<"Subject ? "; gets(subject)

cout<<"Salary ? "; cin>>salary

}

**void** tdisplay()

{

cout<<"Full Name="<<name<<endl;

cout<<"Subject ="<<subject<<endl;

cout<<"Salary ="<<salary<<endl;

}

} t1, t2;

**void** main()

{

//more C++ statements

}

**Objects as parameters**: An object may be passed as a parameter to a function. A function’s **return value could be an object** as well. An example is given below:

**class** Time

{

**int** hh, mm;

**public**:

**void** InputTime()

{

cout<<"Hour ? "; cin>>hh;

cout<<"Minute? "; cin>>mm;

}

**void** DisplayTime() { cout<<"Time: "<<hh<<'.'<<mm<<endl; }

**void** AddTime(Time T1, Time T2)

{

Time t;

t.mm=(T1.mm+T2.mm)%60;

t.hh=T1.hh+T2.hh+(T1.mm+T2.mm)/60;

t.DisplayTime();

}

};

**void** main()

{

Time a, b;

a.InputTime();

b.InputTime();

a.AddTime(a, b); //Or, b.AddTime(a, b);

}

Execution of the program produces following output:

Hour ? 2

Minute? 45

Hour ? 3

Minute? 35

Time: 6.20

Given below is the same program but done it in a little different way.

**class** Time

{

**int** hh, mm;

**public**:

**void** InputTime()

{

cout<<"Hour ? "; cin>>hh;

cout<<"Minute? "; cin>>mm;

}

**void** DisplayTime() { cout<<"Time: "<<hh<<'.'<<mm<<endl; }

**void** AddTime(Time t)

{

**int** Min=mm+t.mm;

mm=Min%60;

hh+=t.hh+Min/60;

}

};

|  |  |
| --- | --- |
| **void** main()  {  Time a, b;  a.InputTime();  b.InputTime();  a.AddTime(b);  a.DisplayTime();  b.DisplayTime();  } | **void** main()  {  Time a, b;  a.InputTime();  b.InputTime();  b.AddTime(a);  a.DisplayTime();  b.DisplayTime();  } |

Execution of the program produces following output:

|  |  |
| --- | --- |
| Hour ? 2  Minute? 45  Hour ? 3  Minute? 35  Time: 6.20  Time: 3.45 | Hour ? 2  Minute? 45  Hour ? 3  Minute? 35  Time: 2.45  Time: 6.20 |

**Function returning object**: A function can return an object. An example is given below:

**class** Time

{

**int** hh, mm;

**public**:

**void** InputTime()

{

cout<<"Hour ? "; cin>>hh;

cout<<"Minute? "; cin>>mm;

}

**void** DisplayTime() { cout<<"Time: "<<hh<<'.'<<mm<<endl; }

Time AddTime(Time T1, Time T2)

{

Time t;

t.mm=(T1.mm+T2.mm)%60;

t.hh=T1.hh+T2.hh+(T1.mm+T2.mm)/60;

**return** t;

}

};

**void** main()

{

Time a, b;

a.InputTime();

b.InputTime();

Time c=a.AddTime(a, b); //Or, Time c=b.AddTime(a, b);

c.DisplayTime();

}

Execution of the program produces following output:

Hour ? 2

Minute? 45

Hour ? 3

Minute? 35

Time: 6.20

In the above example, object is passed as parameter to a member function. But object can be passed as parameter to a stand-alone function (not a member function) also. In the program given below, AddTime() is a stand-alone function. Accordingly class Time has to be modified. Access function Rethr() returns value stored in hh, access function Retmi() returns value stored in mm and member function UpdateTime() assigns values to data members hh and mm. A complete example is given below:

**class** Time

{

**int** hh, mm;

**public**:

**int** Rethr() { **return** hh; }

**int** Retmi() { **return** mm; }

**void** UpdateTime(**int** h, **int** m) { hh=h; mm=m; }

**void** DisplayTime() { cout<<"Time: "<<hh<<'.'<<mm<<endl; }

**void** InputTime()

{

cout<<"Hour ? "; cin>>hh;

cout<<"Minute? "; cin>>mm;

}

};

Time AddTime(Time T1, Time T2)

{

**int** mi=T1.Retmi()+T2.Retmi(), hr=T1.Rethr()+T2.Rethr();

Time t;

t.UpdateTime(hr+mi/60, mi%60);

**return** t;

}

**void** main()

{

Time a, b;

a.InputTime();

b.InputTime();

Time c=AddTime(a, b);

c.DisplayTime();

}

Execution of the program produces following output:

Hour ? 2

Minute? 45

Hour ? 3

Minute? 35

Time: 6.20

**Array of Objects**: So far we have created one or two objects in our programs (in our examples). Suppose we want to represent many objects of same class in computer’s main storage, then we have to create an array of objects. To create an array of object, first we have to create a class and then we will create an array of that class. Each element of the array will be an object.

Name of the class is **ClassName**. Name of the array is **ArrObj**. **ArrObj** is an array of object. **SIZE** is a constant representing number of elements in the array. **SIZE** could either be a user defined constant or a literal constant like **10** or **20**. Every elements of array **ArrObj** will represent an object.

**class** ClassName

{

//DataMembers MemberFunctions

};

ClassName ArrObj[SIZE];

Syntax to access an element of an array of objects is:

ArrObj[Index]

ArrObj[0], ArrObj[1], ArrObj[2], …

are the elements of the array ArrObj. Each element of ArrObj represents an object. With an object only public members of the object can be accessed. Syntax to access public members of an array of objects is given below:

ArrObj[Index].PublicDataMember

Or,

ArrObj[Index].PublicMemberFunction()

An attempt to access private member of an object through an element of ArrObj will result in syntax error. An example is given below explaining the concept of array of objects:

**class** student

Class student has 3 data members and 6 member functions. 6 member functions are: **input**() to input value in the data members, **display1**() to display values stored data members in a tabular (column) format, **display2**() to display values stored data members in a vertical format (one line for each data member), 3 access functions to return value stored in private data members. Immediately access functions do not play any role but they will be needed later. Class **student** will be used to explain the concepts related to array of objects. Array **arr[5]** is an array of **student**.

{

**int** rno; //Roll Number

**char** snam[20]; //Student Name

**double** subm; //Subject Marks

**public:**

**void** input()

{

cout<<"Roll? "; cin>>rno;

cout<<"Name? "; gets(snam);

cout<<"Mark? "; cin>>subm;

}

**void** display1()

{

printf("%2i %-19s %5.1lf\n",

rno, snam, subm);

}

**void** display2()

{

cout<<"Roll= "<<rno<<endl;

cout<<"Name= "<<snam<<endl;

cout<<"Mark= "<<subm<<endl;

}

**int** roll() { **return** rno; }

**char**\* name() { **return** snam; }

**double** marks() { **return** subm; }

}**;**

student arr[5];

arr[0], arr[1], arr[2], arr[3] and arr[4] are the 5 elements of the array. Each element of the array arr is an object. With an element of arr only public members can be used. Given below are the lists of public members that can be used with an element of the array arr.

|  |  |  |
| --- | --- | --- |
| arr[0].input()  arr[0].display1()  arr[0].display2()  arr[0].roll()  arr[0].name()  arr[0].marks() | arr[1].input()  arr[1].display1()  arr[1].display2()  arr[1].roll()  arr[1].name()  arr[1].marks() | arr[2].input()  arr[2].display1()  arr[2].display2()  arr[2].roll()  arr[2].name()  arr[2].marks() |

But an element of the array arr cannot access private members. For example arr[0].rno, arr[0].snam and arr[0].subm will flag syntax errors.

A complete program is given below showing use of array of objects.

**const** MAX=20;

**void** main()

{

student arr[MAX];

**for** (**int** k=0; k<MAX; k++)

arr[k].input();

**double** hm=0, sum=0;

**for** (**int** x=0; x<MAX; x++)

{

arr[x].display1();

sum+=arr[x].marks();

**if** (hm>arr[x].marks())

hm=arr[x].marks();

}

**double** avg=sum/MAX;

cout<<"Highest="<<hm<<" & Average="<<avg<<endl;

}

Or,

**void** arrinput(student a[], **int** n)

{

**for** (**int** k=0; k<n; k++)

arr[k].input();

}

**void** arrdisplay(student a[], **int** n)

{

**for** (**int** k=0; k<n; k++)

arr[k].display1();

}

**void** arrhiav(student a[], **int** n)

{

**double** hm=0, sum=0;

**for** (**int** k=0; k<MAX; k++)

{

sum+=arr[k].marks();

**if** (hm>arr[k].marks())

hm=arr[k].marks();

}

**double** avg=sum/n;

cout<<"Highest="<<hm<<" & Average="<<avg<<endl;

}

**void** main()

{

student arr[MAX];

arrinput(arr, MAX);

arrdisplay(arr, MAX);

arrhiav(arr, MAX);

}

**void** bubblesortroll(student arr[], **int** n)

{

**for**(**int** x=1; x<n; x++)

Function to sort an array of **student** on **roll** using **bubble** sort.

**for**(**int** k=0; k<n-x; k++)

**if** (arr[k].roll()>arr[k+1].roll())

{

student t=arr[k];

arr[k]=arr[k+1];

arr[k+1]=t;

}

}

**void** selectionsortroll(student arr[], **int** n)

{

**for**(**int** x=0; x<n-1; x++)

Function to sort an array of **student** on **roll** using **selection** sort.

{

student min=arr[x];

**int** pos=x;

**for**(**int** k=x+1; k<n; k++)

**if** (arr[k].roll()<min.roll())

{

min=arr[k];

pos=k;

}

arr[pos]=arr[x];

arr[x]=min;

}

}

**void** insertionsortroll(student arr[], **int** n)

{

Function to sort an array of **student** on **roll** using **insertion** sort.

**for**(**int** x=1; x<n; x++)

{

student t=arr[x];

**int** k=x-1;

**while**(k>=0 && t.roll()<arr[k].roll())

{

arr[k+1]=arr[k];

k--;

}

arr[k+1]=t;

}

}

**int** linearsearchroll(student arr[], **int** n, **int** roll)

{

Function to locate for a **roll** in an array of **student** using **linear** search. Return value of the function is **int**. If search is successful function returns value **1** otherwise **0**.

**int** x=0, found=0;

**while** (x<n && found==0)

**if** (roll==arr[x].roll())

found=1;

**else**

x++;

**return** found;

}

**void** linearsearchroll(student arr[], **int** n, **int** roll)

{

Function to locate for a **roll** in an array of **student** using **linear** search. Return value of the function is **void**. Status of search is displayed in the function.

**int** x=0, found=0;

**while** (x<n && found==0)

**if** (roll ==arr[x].roll())

found=1;

**else**

x++;

**if** (found==1)

cout<<roll<<" Found in the array\n";

**else**

cout<<roll<<" Does not Exist in the array\n";

}

**int** binarysearchroll(student arr[], **int** n, **int** roll)

{

Function to locate for a **roll** in a sorted array of **student** using **binary** search. Return value of the function is **int**. If search is successful function returns value **1** otherwise **0**.

**int** lb=0, ub=n-1;

**int** found=0, mid;

**while** (lb<=ub && found==0)

{

mid=(ub+lb)/2;

**if** (roll<arr[mid].roll())

ub=mid-1;

**else**

**if** (roll>arr[mid].roll())

lb=mid+1;

**else**

found=1;

}

**return** found;

}

**void** binarysearchroll(student arr[], **int** n, **int** roll)

{

Function to locate for a **roll** in a sorted array of **student** using **binary** search. Return value of the function is **void**. Status of search is displayed in the function.

**int** lb=0, ub=n-1;

**int** found=0, mid;

**while** (lb<=ub && found==0)

{

mid=(ub+lb)/2;

**if** (roll<arr[mid].roll())

ub=mid-1;

**else**

**if** (roll>arr[mid].roll())

lb=mid+1;

**else**

found=1;

}

**if** (found==1)

cout<<roll<<" Found in the array\n";

**else**

cout<<roll<<" Does not Exist in the array\n";

}

**void** bubblesortname(student arr[], **int** n)

{

**for**(**int** x=1; x<n; x++)

**for**(**int** k=0; k<n-x; k++)

**if** (strcmp(arr[k].name(), arr[k+1].name())>0)

{

Function to sort an array of **student** on **name** using **bubble** sort.

student t=arr[k];

arr[k]=arr[k+1];

arr[k+1]=t;

}

}

**void** selectionsortname(student arr[], **int** n)

{

Function to sort an array of **student** on **name** using **selection** sort.

**for**(**int** x=0; x<n-1; x++)

{

student min=arr[x];

**int** pos=x;

**for**(**int** k=x+1; k<n; k++)

**if** (strcmp(arr[k].name(), min.name())<0)

{

min=arr[k];

pos=k;

}

arr[pos]=arr[x];

arr[x]=min;

}

}

**void** insertionsortname(student arr[], **int** n)

{

Function to sort an array of **student** on **name** using **insertion** sort.

**for**(**int** x=1; x<n; x++)

{

student t=arr[x];

**int** k=x-1;

**while**(k>=0 && strcmp(t.name(), arr[k].name())<0)

{

arr[k+1]=arr[k];

k--;

}

arr[k+1]=t;

}

}

**int** linearsearchname(student arr[], **int** n, **char**\* name)

{

Function to locate for a **name** in an array of **student** using **linear** search. Return value of the function is **int**. If search is successful function returns value **1** otherwise **0**.

**int** x=0, found=0;

**while** (x<n && found==0)

**if** (strcmp(name, arr[x].name())==0)

found=1;

**else**

x++;

**return** found;

}

**void** linearsearchname(student arr[], **int** n, **char**\* name)

{

Function to locate for a **name** in an array of **student** using **linear** search. Return value of the function is **void**. Status of search is displayed in the function.

**int** x=0, found=0;

**while** (x<n && found==0)

**if** (strcmp(name, arr[x].name())==0)

found=1;

**else**

x++;

**if** (found==1)

cout<<name<<" Found in the array\n";

**else**

cout<<name<<" Does not Exist in the array\n";

}

**int** binarysearchname(student arr[], **int** n, **char**\* name)

{

Function to locate for a **name** in a sorted array of **student** using **binary** search. Return value of the function is **int**. If search is successful function returns value **1** otherwise **0**.

**int** lb=0, ub=n-1;

**int** found=0, mid;

**while** (lb<=ub && found==0)

{

mid=(ub+lb)/2;

**if** (strcmp(name, arr[mid].name())<0)

ub=mid-1;

**else**

**if** (strcmp(name, arr[mid].name())>0)

lb=mid+1;

**else**

found=1;

}

**return** found;

}

**void** binarysearchname(student arr[], **int** n, **char**\* name)

{

Function to locate for a **name** in a sorted array of **student** using **binary** search. Return value of the function is **void**. Status of search is displayed in the function.

**int** lb=0, ub=n-1;

**int** found=0, mid;

**while** (lb<=ub && found==0)

{

mid=(ub+lb)/2;

**if** (strcmp(name, arr[mid].name())<0)

ub=mid-1;

**else**

**if** (strcmp(name, arr[mid].name())>0)

lb=mid+1;

**else**

found=1;

}

**if** (found==1)

cout<<name<<" Found in the array\n";

**else**

cout<<name<<" Does not Exist in the array\n";

}

**void** mergeroll(student a[],student b[],student c[],**int** n1,**int** n2)

{

**int** i=0, j=0, k=0;

**while** (i<n1 && j<n2)

Function to **merge** two arrays of **student** sorted on **roll** to obtain the third array also sorted on **roll**. All three arrays are sorted in **ascending** order on **roll**.

**if** (a[i].roll()<b[j].roll())

c[k++]=a[i++];

**else**

c[k++]=b[j++];

**while** (i<n1)

c[k++]=a[i++];

**while** (j<n2)

c[k++]=b[j++];

}

**void** mergename(student a[],student b[],student c[],**int** n1,**int** n2)

Function to **merge** two arrays of **student** sorted on **name** to obtain the third array also sorted on **name**. All three arrays are sorted in **ascending** order on **name**.

{

**int** i=0, j=0, k=0;

**while** (i<n1 && j<n2)

**if** (strcmp(a[i].name(),b[j].name())<0)

c[k++]=a[i++];

**else**

c[k++]=b[j++];

**while** (i<n1)

c[k++]=a[i++];

**while** (j<n2)

c[k++]=b[j++];

}

**void** arrinsert(**student** arr[], **int**& n, **int** pos, **student** item)

{

Function to **insert** an object in an array of **student**. **MAX** is a user defined constant representing size of the array. Array name, number of elements currently in the array, position for insertion and the object that is to be inserted are passed as parameters to the function.

**if** (n==MAX)

cout<<"Overflow\n";

**else**

{

**for** (**int** x=n-1; x>=pos; x--)

arr[x+1]=arr[x];

arr[pos]=item;

n++;

item.display2();

cout<<"Inserted in the array\n";

}

}

**void** arrdelete(**student** arr[], **int**& n, **int** pos)

{

Function to **delete** an object from an array of **student**. Parameter **n** represents number elements currently in the array. Array name, number of elements currently in the array and position for deletion are passed as parameters to the function.

**if** (n==0)

cout<<"Underflow\n";

**else**

{

**for** (**int** x=pos+1; x<n; x++)

arr[x-1]=arr[x];

n--;

}

}

**const** MAX=20;

**class** array

{

**int** array[MAX];

**public**:

**void** arrinput();

**void** arrdisplay();

**void** arrsort();

**void** arrsearch();

};

**void** array::arrinput()

{

**for** (**int** k=0; k<MAX; k++)

array[k]=random(1000);

}

**void** array::arrdisplay()

{

**for** (**int** k=0; k<MAX; k++)

cout<<array[k]<<endl;

}

**void** array::arrsort()

{

**for** (**int** k=1; k<MAX; k++)

**for** (**int** x=0; x<MAX-k; x++)

**if** (array[x]>array[x+1];

{

**int** t=array[k];

array[k]=array[k+1];

array[k]=t;

}

}

**void** array::arrsearch()

{

**int** lb=0, ub=MAX-1, found=0, item, mid;

cout<<"Input integer value to search? "; cin>>item;

**while** (lb<=ub && found==0)

{

mid=(lb+ub)/2;

**if** (item<array[mid])

ub=mid-1;

**else**

**if** (item>array[mid])

lb=mid+1;

**else**

found=1;

}

**if** (found==1)

cout<<item<<" found in the array\n";

**else**

cout<<item<<" not found in the array\n";

}

**void** main()

{

**int** ch;

array arrobj;

randomize();

**do**

{

cout<<"1. Input Array\n";

cout<<"2. Display Array\n";

cout<<"3. Sort Array\n";

cout<<"4. Search Array\n";

cout<<"0. Quit\n";

cout<<"Input Choice[0-4]? "; cin>>ch;

**switch** (ch)

{

**case** 1:arrobj.arrinput();**break**;

**case** 2:arrobj.arrdisplay();**break**;

**case** 3:arrobj.arrsort();**break**;

**case** 4:arrobj.arrsearch();**break**;

}

}

**while** (ch!=0);

}

**Constructor**: A variable of the fundamental type can be created and a value can be assigned to it in one single C++ statement. An array variable or a structure variable can be created and value can be assigned to it by using an initializer. Examples are given below:

Variables **cvar**, **ivar** and **dvar** are also assigned values at the point of creation. Structure variable **b1**, array variables **ar1**, **ar2** and **ar3** are initialized using initializers (values are assigned at the point of creation).

**char** cvar='T';

**int** ivar=20;

**double** dvar=30.25;

**char** ar1[10]="Careless";

**int** ar2[5]={23, 45, 67, 89, 17};

**double** ar3[5]={1.7, 3.8, 9.1, 4.5, 2.3};

**struct** batsman

{

**char** name[15];

**int** runs;

};

batsman b1={"Karthik", 46};

Till now, when working with classes and objects, first we create a class (declare a class with data members and member functions), then we create an object and to store values in the data member of the object, a **public** member function is invoked to either assign values or input values to **private** data members of the object. Now we will learn how to initialize data members of an object when the object is getting created. To do that we will introduce a special **public** member function of the class called **constructor** function. In fact a **constructor** function does much more than that. We will learn more about **constructor** function later. An example is given showing use of **constructor** function to assign values to data members of objects:

**class** cric

Class **cric** has a special **public** member function called **cric**() and it has 2 formal parameters. Values stored in the parameters are assigned to the **private** data members of the class **cric**. Member function **cric**() is the **constructor** function of the class **cric**. Class and constructor function have same name.

{

**char** name[15];

**int** runs;

**public:**

cric(**char**\* n, **int** r)

{

strcpy(name, n);

runs=r;

}

**void** input()

{

cout<<"Name? "; gets(name);

cout<<"Runs? "; cin>>runs;

**cric c1=cric(**"**Amar**"**,103);**

statement invokes the constructor function and assigns values to data members of **c1**. This way of invoking constructor function is called **explicit** call to a constructor function.

**cric c2(**"**Kunal**"**,88);**

statement invokes the constructor function and assigns values to data members of **c2**. This way of invoking constructor function is called **implicit** call to a constructor function.

}

**void** display()

{

cout<<"Name="<<name<<endl;

cout<<"Runs="<<runs<<endl;

}

}**;**

**void** main()

{

cric c1=cric("Amar", 103);

cric c2("Kunal", 88);

c1.display();

c2.display();

}

But if we add the following statement in the main() function then the compiler will flag a **syntax** error.

cric c3;

Compiler will flag **syntax** error because it looks for a constructor function like cric(){ }, that is a constructor **without** any parameter. Since no such constructor exists, compiler flags a syntax error. So constructor function cric(){ } has to be added to the class cric. Edited class cric and the complete program is given below:

Now class **cric** has two constructors, **cric**() and **cric**(**char**\* **n**, **int** **r**). A class can have many constructors. The concept of a class having many constructors is called **constructor overloading**. A constructor function **without** any parameter is called **default** **constructor**. A default constructor may or may not contain any statement (code). A constructor with **parameter** is called **parameterized** **constructor**.

**class** cric

{

**char** name[15];

**int** runs;

**public:**

cric() { }

cric(**char**\* n, **int** r)

{

strcpy(name, n);

runs=r;

}

**void** input()

{

cout<<"Name? "; gets(name);

cout<<"Runs? "; cin>>runs;

}

**void** display()

{

cout<<"Name="<<name<<endl;

cout<<"Runs="<<runs<<endl;

}

}**;**

**void** main()

{

Object **c1** is created by **default** constructor and object **c2** is created by **parameterized** constructor.

cric c1, c2("Kunal", 88);

c1.input();

c1.display();

c2.display();

}

What is the role of constructor cric(){ }? It just creates an object. As mentioned earlier, a constructor function does more than just initialize values to data members of an object. **In fact the most important role of a constructor function is to create an object**. Every object is created by constructor. But classes declared earlier, did not have any constructor. So in that case how were objects created without any constructor? **When a class is created without any constructor, compiler adds a default constructor** (a constructor without any parameter), so that instance of that class can be created. But once a class has a constructor, compiler does not add default constructor and in that case default constructor has to be added by the programmer. A constructor function can be defined inside the class and outside the class as well. An example is given below:

**class** cric

{

**char** name[15];

**int** runs;

**public:**

All the member functions of class **cric**, including constructor functions are **declared** **inside** the class, but all the member functions are **defined** **outside** the class.

cric();

cric(**char**\*, **int**);

**void** input();

**void** display();

}**;**

cric::cric() { }

cric::cric(**char**\* n, **int** r)

{

strcpy(name, n);

runs=r;

}

**void** cric::input()

{

cout<<"Name? "; gets(name);

cout<<"Runs? "; cin>>runs;

}

**void** cric::display()

{

cout<<"Name="<<name<<endl;

cout<<"Runs="<<runs<<endl;

}

Constructor function is a special **public** member function of a class that creates an object. A Constructor functions have the following unique characteristics:

* Name of a constructor is same as that of a class name. If class name is cric then constructor name is cric(), if class name is student then constructor name is student().
* A constructor function is defined / declared only in the **public** area of a class, that is, a constructor is defined / declared with visibility label **public** only.
* A constructor function **does not have any return type**, not even **void**.
* A class can have **many** constructors.
* A constructor function is not inherited.

All types of constructors will be discussed in details. Our discussions regarding constructor will be using class cric as reference. Only relevant constructor will be defined and main() function will be added whenever necessary.

* **Default constructor** is constructor without any parameter. Also it is a constructor added by the compiler when there are no constructor functions defined / declared in a class. Example of default constructor is given:

cric()

**First** definition of **default** constructor **cric**() contains a empty block. **Second** definition of **default** constructor **cric**() contains statements to initialize data member **name** to "" (**nul** string) and data member **runs** to **0** (zero).

{ //No statement }

Or,

cric()

{

strcpy(name, "");

runs=0;

}

* **Parameterised constructor** is a constructor with parameter and it is always created (defined) by the programmer. Example of parameterized constructor is given below:

cric(**char**\* na, **int** ru) //1st Constructor

{

strcpy(name, na);

runs=ru;

}

cric(**char**\* na) //2nd Constructor

{

strcpy(name, na);

runs=0;

}

cric(**int** ru) //3rd Constructor

{

strcpy(name, "NONAME");

runs=ru;

}

**void** main()

Class **cric** has **three** parameterized constructors. Object **c1** is created with **1st** constructor, object **c2** is created with **2nd** constructor and object **c3** is created with **3rd** constructor.

{

cric c1("Kunal", 88);

cric c2("Kunal"), c3(88);

//More C++ statement

}

* **Constructor with default parameter** is a parameterised constructor with default values assigned to the formal parameters of the constructor function. Default values to the formal parameters should always be assigned from right. Advantage of constructor with default parameter is that, single constructor can replace **default** constructor and **parameterized** constructor. Example of constructor with default parameter is given below:

cric(**char**\* na="", **int** ru=0)

{

strcpy(name, na);

Object **c1**’s data members **name** and **runs** are assigned values "**Kunal**" and **88**. Object **c2**’s data members **name** and **runs** are assigned values "**Kunal**" and **0**. Object **c3**’s data members **name** and **runs** are assigned values "" and **0**.

runs=ru;

}

**void** main()

{

cric c1("Kunal", 88);

cric c2("Kunal"), c3;

//More C++ statement

}

* **Copy constructor** is a constructor function in which an object of the same class is passed by **reference** to the constructor function. Copy constructor is used to duplicate an object when the object is getting created. **A copy constructor is added by the compiler when there is no copy constructor defined in the class**. Example of a copy constructor is given below:

cric(**char**\* na="", **int** ru=0)

Objects **c1** and **c2** are created by **parameterized** constructor, objects **c3** and **c4** are created by **copy** constructor and object **c5** is created by **parameterized** constructor. Object **c3** is a duplicate of object **c2** and object **c4** is a duplicate of object **c1**. But object **c5** obtains the values of object **c1** through assignment operator because object **c5** has been created already. Constructor is invoked only once to create an object. There are **two** ways to invoke **copy** constructor:

1. **cric c3=c2;**

2. **cric c4(c1);**

{

//Parameterized Constructor

strcpy(name, na);

runs=ru;

}

cric(cric& c)

{

//Copy Constructor

strcpy(name, c.name);

runs=c.runs;

}

**void** main()

{

cric c1("Kunal", 88);

cric c2("Afzal", 42);

cric c3=c2, c4(c1);

cric c5;

c5=c1;

//More C++ statement

}

**Constructor Overloading**: A class can have more than one constructor functions, that is, objects can be created in more than one ways. Example of constructor overloading is given below:

cric(**char**\* na="", **int** ru=0)

{

strcpy(name, na);

runs=ru;

}

cric(cric& c)

{

strcpy(name, c.name);

runs=c.runs;

}

Or,

cric()

{

strcpy(name, "");

runs=0;

}

cric(**char**\* na, **int** ru)

{

strcpy(name, na);

runs=ru;

}

cric(**char**\* na)

{

strcpy(name, na);

runs=0;

}

cric(**int** ru)

{

strcpy(name, "NONAME");

runs=ru;

}

cric(cric& c)

{

strcpy(name, c.name);

runs=c.runs;

}

Constructor function is a special **public** member function of a class, used to create an object. Once an object is created, it must be destroyed, so we have another special **public** member function called **destructor** function of class to reclaim resources allocated to an object (destroys an object). A destructor function is invoked automatically when an object goes out of scope.

Every object is created by a constructor and destroyed by a destructor. **If a class is created without any destructor, then the compiler adds one**. In most case destructor added by the compiler is sufficient to destroy an object but for some special cases, destructor function has to be added by the programmer.

**Destructor** is a special **public** member function of a class that destroys an object, when an object goes out of scope. A complete program is given below showing use destructor:

**class** cric

Constructor of class **cric** is **cric**(). Destructor of class **cric** is ~**cric**(). Destructor of class starts with ~ (tilde character) followed by class name (**cric**). Destructor ~**cric**() displays a message on the screen. Object **obj** is created by constructor **cric**(). Statement **obj**.**display**(), displays value stored in the data members of object **obj**. When program terminates, object **obj** goes out of scope, that is, destructor ~**cric**() is invoked and **Object Destroyed** is displayed on the screen.

Execution of the program produces following output on the screen:

**Name=Rajat**

**Runs=103**

**Object Destroyed**

{

**char** name[15];

**int** runs;

**public:**

cric(**char**\* n="", **int** r=0)

{

strcpy(name, n);

runs=r;

}

~cric()

{

cout<<"Object Destroyed\n";

}

**void** input()

{

cout<<"Name? "; gets(name);

cout<<"Runs? "; cin>>runs;

}

**void** display()

{

cout<<"Name="<<name<<endl;

cout<<"Runs="<<runs<<endl;

}

}**;**

**void** main()

{

cric obj("Rajat", 103);

obj.display();

}

**Destructor** functions have the following unique characteristics:

* Destructor function name begin with ~ (tilde character) and followed by class name
* A destructor function is defined / declared only in the **public** area of a class, that is, a constructor is defined / declared with visibility label **public** only.
* A destructor function **does not have any return type**, not even **void**.
* A class can have **only one** destructor.
* A destructor function does not have any parameter.
* A destructor function is invoked automatically.
* A destructor function is not inherited

**Visibility** **Label** **protected**: Till now we have only discussed visibility labels **public** and **private**. Now is the right time to introduce third visibility label, **protected**. A **private** member can be used inside the class where as a **public** member can used inside and as well as outside the class. A **protected** member can be used inside the class – similar to a **private** member but it can be inherited – similar to a **public** member. **Private** member cannot be inherited. Inheritance means creating a new class from an old class such that, new class can access **public** and **protected** members of the old class (**public** and **protected** members of the old class become members of new class).

**Inheritance**: Inheritance is a process of creating a new class (**derived** classes or **sub** classes) from old class (**base** class or **super** class). The **derived** class acquires some properties of **base** class and **derived** class also adds new features of its own. Acquiring certain properties from **base** class means that **public** and **protected** members of the **base** class also become members of a **derived** class. It also implies **public** and **protected** members of the **base** class can be directly accessed from a **derived** class. **Private** members of base class cannot be accessed by derived class (**private** members cannot be inherited). The process of Inheritance does not affect the **base** class. A **derived** class is created by specifying its relationship with the **base** class in addition to its own details.

**Rule**: **class** DerivedClassName: AccSpec BaseClassName

{

//Members of Derived Class

};

The colon indicates that the derived class DerivedClassName is created (derived / defined) from the base class BaseClassName. Access specifier (AccSpec)can either be **public** or **protected** or **private**. Access specifier is optional. The default access specifier is **private**. Access specifier specifies whether features of base class are **publicly** or **protectedly** or **privately** derived. An example is given below:

**Base** class **oldclass**

* **Private** member: **a**
* **Inheritable** members
  + **Protected member: b, ocp()**
  + **Public** member: **c**, **ocf**()

**class** oldclass

{

**int** a;

**protected**:

**char** b;

**void** ocp();

**public**:

**double** c;

**Derived** class **newclass**

* **Private** member: **x**
* **Protected** members
  + **Own**: **y**, **ncp**()
  + **Inherited**: **b**, **ocp**()
* **Public** members
  + **Own: z, ncf()**
  + **Inherited**: **c**, **ocf**()

Arrow head pointing down implies that the **newclass** is **derived** from **oldclass**.

**void** ocf();

};

**class** newclass: **public** oldclass

{

**double** x;

**protected**:

**int** y;

**void** ncp();

**public**:

**char** z;

**void** ncf();

};

Base class is oldclass, from base class oldclass a new class is created named newclass (derived class). **Public** members of oldclass (c, and ocf()) and **protected** members of oldclass (b, ocp()) become members of newclass. But **private** members of oldclass, a remain private to oldclass, that is, they are not inherited. Since AccessSpecifier is **public**, therefore, **public** and **protected** members of base class retain their base class visibility label in the derived class as well (**public** members of base class remains **public** and **protected** members of base class remain **protected** in the **derived** class). A complete programming example is given below:

**class** person

{

**char** name[20];

**int** mobile;

**public**:

**void** inputperson()

{

cout<<"Name ? "; gets(name);

cout<<"Mobile ? "; cin>>mobile;

}

**void** displayperson()

{

cout<<"Name : "<<name<<endl;

cout<<"Mobile : "<<mobile<<endl;

}

};

**class** employee: **public** person

{

**int** eno;

**char** designation[40]; **double** basic;

**public**:

**void** inputemp()

{

cout<<"Emp Number ? "; cin>>eno;

cout<<"Designation? "; gets(designation);

cout<<"Basic Pay ? "; cin>>basic;

}

**void** displayemp()

{

cout<<"Emp Number : "<<eno<<endl;

cout<<"Designation: "<<designation<<endl;

cout<<"Basic Pay : "<<basic<<endl;

}

};

**void** main()

{

employee obj;

obj.inputperson();

obj.inputemp();

obj.displayperson();

obj.displayemp();

}

Running of the program produces following output:

Name ? Shankar N Gupta

Mobile ? 99637248

Emp Number ? 21

Designation? Network Administrator

Basic Pay ? 2100

Name : Shankar N Gupta

Mobile : 99637248

Emp Number : 21

Designation: Network Administrator

Basic Pay : 2100

* When memory is allocated to an instance of derived class (in our example employee is the **derived** class and obj is an instance of employee), memory is allocated for data members of **base** class (including memory allocated for the **private** members of **base** class) plus data members of **derived** class.
* In our example memory allocated to instance of employee is 80 bytes (20 bytes for name, 4 bytes each for phone and mobile plus 4 bytes for eno, 40 bytes for designation and 8 bytes for basic).
* Even though the memory is allocated for the **private** members of the **base** class, member functions of **derived** class cannot access **private** members of the **base** class.
* Private members of the **base** class are accessed by member functions of **base** class.
* In our example, inputperson() and displayperson() are the **public** member functions of **base** class, person. Through inheritance, inputperson() and displayperson() become member of **derived** class employee.
* Member functions inputperson() and displayperson() are invoked from an instance of employee to access the **private** members of **base** class person.

What happens when access specifier is changed from **public** to **private** (or **protected**)? As mentioned earlier, if access specifier is **public**, then inherited members in the **derived** class retains the visibility label of the **base** class. But if access specifier is **private** (or **protected**) then all the inherited members become **private** (or **protected**) member in the **derived** class. A **private** (or **protected**) member cannot be used outside the class. Consider the main() function given below:

**void** main()

{

employee empobj;

empobj.inputperson();

empobj.inputemp();

empobj.displayperson();

empobj.displayemp();

}

Assuming that access specifier is either **private** or **protected**, statements obj.inputperson(); and obj.displayperson(); would flag syntax errors. Therefore member functions of class person, inputperson() and displayperson() are to be invoked from member functions class employee. Edited program is given in the next page:

**class** person

{

**char** name[20];

**int** mobile;

**public**:

**void** inputperson()

{

cout<<"Name ? "; gets(name);

cout<<"Mobile ? "; cin>>mobile;

}

**void** displayperson()

{

cout<<"Name : "<<name<<endl;

cout<<"Mobile : "<<mobile<<endl;

}

};

**class** employee: **private** person

{

**int** eno;

**char** designation[40];

**double** basic;

**public**:

**void** inputemp()

{

inputperson(); //Base class member function

cout<<"Emp Number ? "; cin>>eno;

cout<<"Designation? "; gets(designation);

cout<<"Basic Pay ? "; cin>>basic;

}

**void** displayemp()

{

displayperson(); //Base class member function

cout<<"Emp Number : "<<eno<<endl;

cout<<"Designation: "<< designation <<endl;

cout<<"Basic Pay : "<<basic<<endl;

}

};

**void** main()

{

employee obj;

obj.inputemp();

obj.displayemp();

}

Running of the program produces following output:

Name ? Shankar N Gupta

Mobile ? 99637248

Emp Number ? 21

Designation? Network Administrator

Basic Pay ? 2100

Name : Shankar N Gupta

Mobile : 99637248

Emp Number : 21

Designation: Network Administrator

Basic Pay : 2100

* Access specifier is **private** (or **protected** but the code would have remained the same)
* **Base** class member function inputperson() is invoked from **derived** class member function inputemp()
* **Base** class member function displayperson() is invoked from **derived** class member function displayemp()

**Types of Inheritance**

1. **Single Level Inheritance**: A **derived** class has one **base** class (or a **base** class has only one **derived** class).

**Rule**: **class** BaseClassName

{

//Members of Base Class

};

**class** DerivedClassName: AccSpec BaseClassName

{

//Members of Derived Class

};

Only **public** and **protected** members of **base** class are inherited by **derived** class (**public** and **protected** members of **base** class become members of **derived** class). Access specifier (AccSpec) is either **public** or **protected** or **private**.

**Base Class**

**Example**:

**class** person

{

**char** name[20]; **int** mobile;

**Derived Class**

**public**:

**void** inputperson()

{

cout<<"Name ? "; gets(name);

cout<<"Mobile ? "; cin>>mobile;

}

**void** displayperson()

{

cout<<"Name : "<<name<<endl;

cout<<"Mobile : "<<mobile<<endl;

}

};

**class** employee: **public** person

{

**int** eno;

**char** designation[40];

**double** basic, hra, gross;

**public**:

**void** inputemp()

{

cout<<"Emp Number ? "; cin>>eno;

cout<<"Designation? "; gets(designation);

cout<<"Basic Pay ? "; cin>>basic;

hra=0.35\*basic; gross=basic+hra;

}

**void** displayemp()

{

cout<<"Emp Number : "<<eno<<endl;

cout<<"Designation: "<<designation<<endl;

cout<<"Basic Pay : "<<basic<<endl;

cout<<"House Rent : "<<hra<<endl;

cout<<"Gross Pay : "<<gross<<endl;

}

};

**void** main()

{

employee obj;

obj.inputperson();

obj.inputemp();

obj.displayperson();

obj.displayemp();

}

2. **Multi-level Inheritance**: A new class is derived from a derived class, that is, between top most base class and bottom most derived there is at least one intermediate class.

**Rule**: **class** BaseClass

{

//Members of Base Class

};

**class** InterDerClass1: AccSpec1 BaseClass

{

//Members of 1st Intermediate Class

};

**class** InterDerClass2: AccSpec2 InterClass1

{

//Members of 2nd Intermediate Class

};

**class** DeriveClass: AccSpec3 InterClass2

{

//Members of Derived Class

};

Classes InterDerClass1 and InterDerClass2 represent **intermediate** **derived** classes between top most **base** class (BaseClass) and bottom most **derived** class (DeriveClass). If access specifier (AccSpec1, AccSpec2, AccSpec3) is either **public** or **protected** then members inherited by an **intermediate** **derived** class can be further inherited by classes derived from **intermediate** **derived** class.

**Example**:

**Top Most**

**Base Class**

**Intermediate**

**Derived Class 1**

**Intermediate**

**Derived Class 2**

**Bottom Most**

**Derived Class**

**class** person

{

**char** name[20];

**int** mobile;

**public**:

**void** inputperson()

{

cout<<"Name ? "; gets(name);

cout<<"Mobile ? "; cin>>mobile;

}

**void** displayperson()

{

cout<<"Name : "<<name<<endl;

cout<<"Mobile : "<<mobile<<endl;

}

};

**class** employee: **public** person

{

**int** eno;

**char** designation[40];

**public**:

**void** inputemp()

{

cout<<"Emp Number ? "; cin>>eno;

cout<<"Designation? "; gets(designation);

}

**void** displayemp()

{

cout<<"Emp Number : "<<eno<<endl;

cout<<"Designation: "<<designation<<endl;

}

};

**class** salary: **public** employee

{

**double** basic, hra, da, gross;

**public**:

**void** inputsal()

{

cout<<"Basic Sal ? "; cin>>basic;

hra=0.3\*basic; da=0.4\*basic;

gross=basic+hra+da;

}

**void** displaysal()

{

cout<<"Basic Sal : "<<basic<<endl;

cout<<"House Rent : "<<hra<<endl;

cout<<"DA : "<<da<<endl;

cout<<"Gross Sal : "<<gross<<endl;

}

};

**void** main()

{

salary obj;

obj.inputperson();

obj.inputemp();

obj.inputsal();

obj.displayperson();

obj.displayemp();

obj.displaysal();

}

Running of the program produces following output:

Name ? Sushant Gupta

Mobile ? 93150342

Emp Number ? 45

Designation? Business Manager

Basic Sal ? 65000

Name : Sushant Gupta

Mobile : 93150342

Emp Number : 45

Designation: Business Manager

Basic Sal : 65000

House Rent : 19500

DA : 26000

Gross Sal : 110500

Intermediate **derived** class employee is derived **publically** from **class** person, therefore members inherited by class employee are ready to be inherited further. **Derived** class salary further inherits members inherited by class employee (from class person). Member functions inputperson() and displayperson() are inherited by class employee from base class person. Class salary further inherits these two member functions of class employee through class person. That is the reason it is possible to invoke functions inputperson() and displayperson() from an instance of salary.

If **intermediate** **derived** class was derived **protectedly** (**private**) from class person, class salary would still have inherited functions inputperson() and displayperson() but as **protected** (**private**) members. Hence they had to be invoked either from member functions of class employee. Edited program is given below:

**class** person

{

**char** name[20];

**int** mobile;

**public**:

**void** inputperson()

{

cout<<"Name ? "; gets(name);

cout<<"Mobile ? "; cin>>mobile;

}

**void** displayperson()

{

cout<<"Name : "<<name<<endl;

cout<<"Mobile : "<<mobile<<endl;

}

};

**class** employee: **protected** person

{

**int** eno;

**char** designation[40];

**public**:

**void** inputemp()

{

inputperson();

cout<<"Emp Number ? "; cin>>eno;

cout<<"Designation? "; gets(designation);

}

**void** displayemp()

{

displayperson();

cout<<"Emp Number : "<<eno<<endl;

cout<<"Designation: "<<designation<<endl;

}

};

**class** salary: **public** employee

{

**double** basic, hra, da, gross;

**public**:

**void** inputsal()

{

cout<<"Basic Sal ? "; cin>>basic;

hra=0.3\*basic;

da=0.4\*basic;

gross=basic+hra+da;

}

**void** displaysal()

{

cout<<"Basic Sal : "<<basic<<endl;

cout<<"House Rent : "<<hra<<endl;

cout<<"DA : "<<da<<endl;

cout<<"Gross Sal : "<<gross<<endl;

}

};

**void** main()

{

salary obj;

obj.inputemp();

obj.inputsal();

obj.displayemp();

obj.displaysal();

}

Member function inputperson() and displayperson() are invoked from member function of class employee – inputemp() and displayemp() respectively because inputperson() and displayperson() are **protected** members of class employee. If access specifier was **private** at every level, then all the inherited member functions had to be invoked from member functions of derived class. An example is given:

**class** person

{

**char** name[20];

**int** mobile;

**public**:

**void** inputperson()

{

cout<<"Name ? "; gets(name);

cout<<"Mobile ? "; cin>>mobile;

}

**void** displayperson()

{

cout<<"Name : "<<name<<endl;

cout<<"Mobile : "<<mobile<<endl;

}

};

**class** employee: **private** person

{

**int** eno;

**char** designation[40];

**public**:

**void** inputemp()

{

inputperson();

cout<<"Emp Number ? "; cin>>eno;

cout<<"Designation? "; gets(designation);

}

**void** displayemp()

{

displayperson();

cout<<"Emp Number : "<<eno<<endl;

cout<<"Designation: "<<designation<<endl;

}

};

**class** salary: **private** employee

{

**double** basic, hra, da, gross;

**public**:

**void** inputsal()

{

inputemp();

cout<<"Basic Sal ? "; cin>>basic;

hra=0.3\*basic; da=0.4\*basic;

gross=basic+hra+da;

}

**void** displaysal()

{

displayemp();

cout<<"Basic Sal : "<<basic<<endl;

cout<<"House Rent : "<<hra<<endl;

cout<<"DA : "<<da<<endl;

cout<<"Gross Sal : "<<gross<<endl;

}

};

**void** main()

{

salary obj;

obj.inputsal();

obj.displaysal();

}

Since classes are derived **privately**, inherited members become **private** members in the **derived** class, meaning, those inherited members has to be used inside (if data member) or invoked from (if member function) member functions of **derived** class. Also those inherited members cannot be inherited further. But those members can be accessed indirectly through the member functions of immediate base class. In our example, inputperson() is invoked from inputemp() and inputemp() is invoked from inputsal() and displayperson() is invoked from displayemp() and displayemp() is invoked from displaysal(). From main() function, only two functions are invoked, inputsal() and displaysal() – the public member functions of object obj. Input of data is done by invoking member function inputsal(), inputsal() invokes inputemp() and inputemp() invokes inputperson(). Output is displayed on the screen by invoking member function displaysal(), displaysal() invokes displayemp() and displayemp() invokes displayperson().

3. **Multiple Inheritance**: A new class is derived from two or more (at least two) base classes (or a derived class has two or more base classes – at least two base classes).

**Rule**: **class** BaseClass1

{

//Members of 1st Base Class

};

**class** BaseClass2

{

//Members of 2nd Base Class

};

**class** DeriveClass: AccSpec1 BaseClass1, AccSpec2 BaseClass2

{

//Members of Derived Class

};

Classes BaseClass1 and BaseClass2 represent two base classes. Class DeriveClass is derived from BaseClass1 and BaseClass2. If access specifier (AccSpec1, AccSpec2) is either **public** or **protected** then members inherited by DeriveClass class can be further inherited by classes derived from DeriveClass.

**Derived Class**

**Base Class1**

**Base Class2**

**Example**:

**class** student

{

**int** sroll;

**char** sname[20];

**public**:

**void** inputstud()

{

cout<<"Student Roll ? "; cin>>sroll;

cout<<"Student Name ? "; gets(sname);

}

**void** displaystud()

{

cout<<"Student Roll : "<<sroll<<endl;

cout<<"Student Name : "<<sname<<endl;

}

};

**class** teacher

{

**int** tcode;

**char** tname[20];

**public**:

**void** inputteach()

{

cout<<"Teacher Code ? "; cin>>tcode;

cout<<"Teacher Name ? "; gets(tname);

}

**void** displayteach()

{

cout<<"Teacher Roll : "<<tcode<<endl;

cout<<"Teacher Name : "<<tname<<endl;

}

};

**class** school: **public** student, **public** teacher

{

**char** schname[40];

**char** priname[20];

**char** board[20];

**public**:

**void** inputscho()

{

cout<<"School Name ? "; gets(schname);

cout<<"Principal Name? "; gets(priname);

cout<<"Board ? "; cin>>board;

}

**void** displayscho()

{

cout<<"School Name : "<<schname<<endl;

cout<<"Principal Name: "<<priname<<endl;

cout<<"Board : "<<board<<endl;

}

};

**void** main()

{

school obj;

obj.inputstud();

obj.inputteach();

obj.inputscho();

obj.displaystud();

obj.displayteach();

obj.displayscho();

}

Running of the program produces following output:

Student Roll ? 19

Student Name ? Sudhir Kr Jain

Teacher Code ? 1023

Teacher Name ? Vimal Garg

School Name ? Elite International School

Principal Name? Suman Rani Bhalla

Board ? CBSE

Student Roll : 19

Student Name : Sudhir Kr Jain

Teacher Roll : 1023

Teacher Name : Vimal Garg

School Name : Elite International School

Principal Name: Suman Rani Bhalla

Board : CBSE

4. **Hierarchical Inheritance**: Two or more classes (at least two derived classes) are derived from single base class (or two or more derived classes have same base class).

**Rule**: **class** BaseClass

{

//Members of Base Class

};

**class** DeriveClass1: AccSpec1 BaseClass

{

//Members of 1st Derive Class

};

**class** DeriveClass2: AccSpec2 BaseClass

{

//Members of 2nd Derive Class

};

Classes DeriveClass1 and DeriveClass2 are two **derived** classes derived from single **base** class BaseClass. If access specifier (AccSpec1, AccSpec2) is either **public** or **protected** then members inherited by classes DeriveClass1 and DeriveClass2 can be further inherited by classes derived from DeriveClass1 and DeriveClass2.

**Base Class**

**Derived Class1**

**Derived Class2**

**Example**:

**class** person

{

**char** name[30];

**int** phone;

**char** address[70];

**public**:

**void** inputpers()

{

cout<<"Name ? "; gets(name);

cout<<"Phone ? "; cin>>phone;

cout<<"Address ? "; gets(address);

}

**void** showpers()

{

cout<<"Name : "<<name<<endl;

cout<<"Phone : "<<phone<<endl;

cout<<"Address : "<<address<<endl;

}

};

**class** staff: **public** person

{

**int** eno;

**char** designation[30];

**double** salary;

**public**:

**void** inputstaff()

{

cout<<"Emp Number ? "; cin>>eno;

cout<<"Designation? "; gets(designation);

cout<<"Monthly Pay? "; cin>>salary;

}

**void** showstaff()

{

cout<<"Emp Number : "<<eno<<endl;

cout<<"Designation: "<<designation<<endl;

cout<<"Monthly Pay: "<<salary<<endl;

}

};

**class** tempstaff: **public** person

{

**char** particular[30];

**double** hrwage;

**public**:

**void** inputtemp()

{

cout<<"Particular ? "; gets(particular);

cout<<"Hourly Wage? "; cin>>hrwage;

}

**void** showtemp()

{

cout<<"Particular : "<<particular<<endl;

cout<<"Hourly Wage: "<<hrwage<<endl;

}

};

**void** main()

{

staff obj1;

obj1.inputpers();

obj1.inputstaff();

obj1.showpers();

obj1.showstaff();

cout<<endl;

tempstaff obj2;

obj2.inputpers();

obj2.inputtemp();

obj2.showpers();

obj2.showtemp();

}

Running of the program produces following output:

Name ? Dinesh John Mathew

Phone ? 23921834

address ? Building-6, Flat-12, Street-28, Mangaf-3

Emp Number ? 1017

Designation? Marketing Manager

Monthly Pay? 2500

Name : Dinesh John Mathew

Phone : 23921834

Address : Building-6, Flat-12, Street-28, Mangaf-3

Emp Number : 1017

Designation: Marketing Manager

Monthly Pay: 2500

Name ? Lakshman Singh

Phone ? 23725692

address ? House-45, Street-39, North Ahmadi

Particular ? Part time worker

Hourly Wage? 2.5

Name : Lakshman Singh

Phone : 23725692

Address : House-45, Street-39, North Ahmadi

Particular : Part time worker

Hourly Wage: 2.5

### Abstract class and Concrete class: A class without an instance is called Abstract class where as a class with an instance is called Concrete class. An Example is given below:

**class** bclass

{

**int** a, b;

**public**:

**void** inputbc()

{

cout<<"Input a? "; cin>>a;

cout<<"Input b? "; cin>>b;

}

**void** showbc()

{

cout<<"a="<<a<<endl;

cout<<"b="<<b<<endl;

}

};

**class** dclass: **public** bclass

{

**double** x, y;

**public**:

**void** inputdc()

{

cout<<"Input x? "; cin>>x;

cout<<"Input y? "; cin>>y;

}

**void** showdc()

{

cout<<"x="<<x<<endl;

cout<<"y="<<y<<endl;

}

Class **bclass** is **base** class and class **dclass** is **derived** from **bclass**. In the **main**() function there is an instance of **dclass**, **obj**. Hence **bclass** is **Abstract** class (without any instance) and **dclass** is a **Concrete** class (has an instance).

};

**void** main()

{

dclass obj;

obj.inputbc();

obj.inputdc();

obj.showbc();

obj.showdc();

}

**Inheritance, Constructor and Destructor**: Constructor and destructor functions are never inherited. But constructor and destructor play a very important role in creating object and deleting object, that is, base class constructor will create base class object (resources will be allocated for base class members) and derived class constructor will create derived class objects (resources will be allocated for derived class members). In case of **multi**-**level** inheritance, the constructors will be executed **in the order** of inheritance. While in case of **multiple** inheritance, the constructors will be executed in the **order in which they appear in the declaration** of the derived class. Order of execution of **destructor** will be the **reverse** order of execution of **constructor**.

//Constructor and Destructor in multi-level inheritance

**class** AAA

{

**char** strA[5];

**public**:

AAA()

{

cout<<"Constructor of AAA\n"; strcpy(strA, "AAA");

}

**void** showAAA() { cout<<"strA = "<<strA<<endl; }

~AAA() { cout<<"Destructor of AAA\n"; }

};

**class** BBB: **public** AAA

{

**char** strB[5];

**public**:

BBB()

{

cout<<"Constructor of BBB\n"; strcpy(strB, "BBB");

}

**void** showBBB() { cout<<"strB = "<<strB<<endl; }

~BBB() { cout<<"Destructor of BBB\n"; }

};

**class** CCC: **public** BBB

{

**char** strC[5];

**public**:

CCC()

{

cout<<"Constructor of CCC\n"; strcpy(strC, "CCC");

}

**void** showCCC() { cout<<"strC = "<<strC<<endl; }

~CCC() { cout<<"Destructor of CCC\n"; }

};

**void** main()

{

CCC obj;

obj.showAAA(); obj.showBBB(); obj.showCCC();

}

Running of the program produces following output:

Constructor of AAA

Class **BBB** is derived from class **AAA** and class **CCC** is derived from class **BBB**. Therefore **constructors** of three classes are executed in that order, that is, first **constructor** of **AAA** executed, then **constructor** of **BBB** is executed and finally **constructor** of **CCC** is executed. **Destructors** are invoked in the reverse order. First **destructor** of **CCC** is invoked, then **destructor** of **BBB** is invoked and finally **destructor** of **AAA** is invoked.

Constructor of BBB

Constructor of CCC

strA = AAA

strB = BBB

strC = CCC

Destructor of CCC

Destructor of BBB

Destructor of AAA

//Constructor and Destructor in multiple inheritance #1

**class** AAA

{

**char** strA[5];

**public**:

AAA()

{

cout<<"Constructor of AAA\n"; strcpy(strA, "AAA");

}

**void** showAAA() { cout<<"strA = "<<strA<<endl; }

~AAA() { cout<<"Destructor of AAA\n"; }

};

**class** BBB

{

**char** strB[5];

**public**:

BBB()

{

cout<<"Constructor of BBB\n"; strcpy(strB, "BBB");

}

**void** showBBB() { cout<<"strB = "<<strB<<endl; }

~BBB() { cout<<"Destructor of BBB\n"; }

};

**class** CCC: **public** AAA, **public** BBB

{

**char** strC[5];

**public**:

CCC()

{

cout<<"Constructor of CCC\n"; strcpy(strC, "CCC");

}

**void** showCCC() { cout<<"strC = "<<strC<<endl; }

~CCC() { cout<<"Destructor of CCC\n"; }

};

**void** main()

{

CCC obj;

obj.showAAA(); obj.showBBB(); obj.showCCC();

}

Running of the program produces following output:

Class **CCC** is declared as:

**class** CCC**: public** AAA**, public** BBB

{ //Members of CCC };

**Base** class constructors are executed in the order in which they appear in the declaration of the **derived** class. **Constructor** of **AAA** is executed first, then **constructor** of **BBB** is executed and **constructor** of **CCC** is executed last. **Destructors** are invoked in the reverse order – **CCC**'s **destructor** first, **BBB**'s **destructor** next and last **AAA**'s **destructor**.

Constructor of AAA

Constructor of BBB

Constructor of CCC

strA = AAA

strB = BBB

strC = CCC

Destructor of CCC

Destructor of BBB

Destructor of AAA

//Constructor and Destructor in multiple inheritance #2

**class** AAA

{

**char** strA[5];

**public**:

AAA()

{

cout<<"Constructor of AAA\n"; strcpy(strA, "AAA");

}

**void** showAAA() { cout<<"strA = "<<strA<<endl; }

~AAA() { cout<<"Destructor of AAA\n"; }

};

**class** BBB

{

**char** strB[5];

**public**:

BBB()

{

cout<<"Constructor of BBB\n"; strcpy(strB, "BBB");

}

**void** showBBB() { cout<<"strB = "<<strB<<endl; }

~BBB() { cout<<"Destructor of BBB\n"; }

};

**class** CCC: **public** BBB, **public** AAA

{

**char** strC[5];

**public**:

CCC()

{

cout<<"Constructor of CCC\n"; strcpy(strC, "CCC");

}

**void** showCCC() { cout<<"strC = "<<strC<<endl; }

~CCC() { cout<<"Destructor of CCC\n"; }

};

**void** main()

{

CCC obj;

obj.showAAA(); obj.showBBB(); obj.showCCC();

}

Running of the program produces following output:

Class **CCC** is declared as:

**class** CCC**: public** BBB**, public** AAA

{ //Members of CCC };

**Base** class constructors are executed in the order in which they appear in the declaration of the **derived** class. **Constructor** of **BBB** is executed first, then **constructor** of **AAA** is executed and **constructor** of **CCC** is executed last. **Destructors** are invoked in the reverse order – **CCC**'s **destructor** first, **AAA**'s **destructor** next and last **BBB**'s **destructor**.

Constructor of BBB

Constructor of AAA

Constructor of CCC

strA = AAA

strB = BBB

strC = CCC

Destructor of CCC

Destructor of AAA

Destructor of BBB

**Polymorphism:** The process of using an operator or a function in different ways for different set of inputs given is known as polymorphism (or functions and operators have same name but perform different actions depending on formal parameters).

In C++ **polymorphism** is implemented through **Function** **Overloading**, **Operator** **Overloading** and **Virtual** **Function**. But **Operator** **Overloading** and **Virtual** **Function** are not included in the syllabus. When two or more functions with the same name work with different set of parameters to perform different operations is known as **Function** **Overloading**, that is, functions are differentiated on the basis of formal parameters.

**Example** **1**:

**void** print() { cout<<"Summer Break is Over!"<<endl; }

**void** print(**int** n) { cout<<"Integer = "<<n<<endl; }

**void** print(**double** m) { cout<<"Double = "<<m<<endl; }

**void** print(**char** s[]){ cout<<"String = "<<s<<endl; }

**void** main()

Function **print**() is an **Overloaded** **Function**, differentiate on the basis of **data** **type** of **formal** parameter. There are 4 definitions of function **print**() – without any parameter, **int** parameter, **double** parameter and string parameter.

{

print(90);

print("Back to School!");

print(12.75);

print();

}

Running of the program produces following output:

Integer = 90

String = Back to School

Double = 12.75

Happy Holidays!

**Example** **2**:

**void** area(**int** si)

{

**int** sa=si\*si;

cout<<"Area of Square = "<<sa<<endl;

}

**void** area(**double** ra)

{

**double** ca=3.14159\*ra\*ra;

cout<<"Area of Circle = "<<ca<<endl;

}

**void** area(**int** le, **int** br)

{

**int** ra=le\*br;

cout<<"Area of Rectangle = "<<ra<<endl;

}

**void** area(**double** hi, **double** ba)

{

**double** ta=0.5\*hi\*ba;

cout<<"Area of Triangle = "<<ta<<endl;

}

**void** main()

Function **area**() is an **Overloaded** **Function**, differentiate on the basis of **data** **type** of **formal** parameter and **number** of **formal** parameters. There are 4 definitions of function **area**() – one **int** parameter, two **int** parameters, one **double** parameter and two **double** parameters.

{

area(7.5);

area(10, 5);

area(12.5, 3.5);

area(15);

}

Running of the program produces following output:

Area of Circle = 176.714

Area of Rectangle = 50

Area of Triangle = 21.875

Area of Square = 225

**Reusability**: is another important feature of OOP. It is always nice if we could to reuse something that already exists rather than trying to create the same all over again. It would not only save time and money but also reduce frustration and increase reliability. For example, the reuse of a class that has already been tested, debugged and used many times can save us effort of developing and testing the same. C++ strongly supports the concept of reusability. The C++ classes can be reused in several ways. Once a class has been created 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. This means that we can add additional features to an existing class without modifying it. This is possible by deriving a new class from an existing one. The new class will have the combined features of both the classes.

**Data Abstraction**: refers to the act of representing the essential features without including the background details. Class is defined as a list of abstract attributes (data members) and the member functions to operate on these attributes (data members). They encapsulate all the essential properties of the object that are to be created. Since the classes use the concept of data abstraction, they are known as Abstract Data Type (ADT). An example is given below:

**class** student

Data members **roll**, **name** and **marks** of class **student** represent abstract attributes. Why abstract attributes? Because with these 3 data members we are trying to represent a student – and missing so many other attributes which cannot be represented or may not be relevant for the program. For example class **student** does not include the likes and dislikes of the student. It does not mention whether he / she uses school transport or private transport to travel to and fro between his / her house and school since this bit of information is not relevant for the program.

{

**int** roll, mark;

**char** name[20];

**public**:

**void** input()

{

cout<<"Roll? "; cin>>roll;

cout<<"Name? "; gets(name);

cout<<"Mark? "; cin>>marks;

}

**void** display()

{

cout<<"Roll= "<<roll<<endl;

cout<<"Name= "<<name<<endl;

cout<<"Mark= "<<mark<<endl;

}

};

|  |  |
| --- | --- |
| **Object Oriented Program** | **Procedural Oriented Program** |
| * Equal emphasis is on data and functions | * Emphasis is on doing things (Functions) |
| * Concept of Data Hiding prevents accidental change in the data | * Excessive use of global variables may result in accidental change of data. |
| * Supports Polymorphism, Inheritance and Encapsulation | * Does not support Polymorphism, Inheritance and Encapsulation |

|  |  |  |
| --- | --- | --- |
| **Visibility Label** | **Accessibility** | **Inheritable** |
| Public | Can be used inside the class as well as outside the class | Yes |
| Protected | Can be used inside the class only | Yes |
| Private | Can be used inside the class only | No |

|  |  |  |  |
| --- | --- | --- | --- |
| **Base Class Visibility** | **Access Specifier** | **Derived Class Visibility** | **Further Inheritable** |
| Public | Public | Public | Yes |
| Protected | Protected |
| Public | Protected | Protected | Yes |
| Protected | Protected |
| Public | Private | Private | No |
| Protected | Private |