Unit 2

Classes and Objects

Introduction to C++

Origin

C++ is an object-oriented Programming Language.

It was developed by Bjarne Stroustrup at AT &T Bell Laboratories in Murry Hill, New Jersey, USA in the early 1980's.

Stroustroup, an admirer of simua67 and strong supporter of C, wanted to combine the best of both the languages and create a more powerful language that could support Object Oriented Programming features still retain the power and elegance of C. The result was C++.Therefore, C++ is an extension of C with a major addition of class construct feature of simula67.

Since class was the major addition to the Original C language, Stroustrup initially called the new language 'C with classes'. However, later in 1983, the name was changed to C++. The idea of C++ comes from the C increment operator ++, thereby suggesting that C++ is an augmented (incremented) version of C.

C++ is superset of C. Most of what we already know about C applies to C++ also. Therefore, almost all C programs are also C++ programs. However, there are a few minor differences that will prevent C program to run under C++ compiler.

The most important facilities that C++ adds on to C are classes, inheritance, function overloading and operator overloading. These features enable creation of abstract data types, inherit properties from existing data types and support polymorphism, thereby making C++ a truly Object Oriented Language.

Basic C++ program structure

Let us begin with a simple example of a C++ program that prints a string on the screen.

```
#include<iostream>
using namespace std;
int main()
{
        cout<<"C++ is better than C";
        return 0;
    //End of example
}
```

When we run a C++ program, the first statement executed will be at the beginning of a function called main(). The program may consist of many functions, classes, and other program elements, but on startup, control always goes to main(). If there is no function called main() in your program, an error will be reported when we run the program.

The first statement **cout<<"C++ is better than C**"; tells the computer to display the quoted phrase.

A semicolon signals the end of the statement. This is a crucial part of the syntax.

The last statement in the function body is **return 0**; This tells main() to return the value 0 to whoever called it, in this case the operating system or compiler.

- **Preprocessor directives and header file**

The first line of the program

```
#include <iostream>
```

might look like a program statement, but it's not. It isn't part of a function body and doesn't end with a semicolon, as program statements must. Instead, it starts with a number sign (#). It's called a preprocessor directive.

A preprocessor directive, on the other hand, is an instruction to the compiler. A part of the compiler called the preprocessor deals with these directives before it begins the real compilation process. The preprocessor directive #include tells the compiler to insert another file into your source file. In effect, the #include directive is replaced by the contents of the file indicated. Using an #include directive to insert another file into your source file is similar to pasting a block of text into a document with your word processor.

#include is only one of many preprocessor directives, all of which can be identified by the initial # sign.

This directive causes the preprocessor to add the contents of the iostream file to the program. It contains declarations for the identifier cout and the operator <<. Some old versions of C++ use a header file called iostream.h. This one is changes introduced by ANSI C++. (we should use iostream.h if the compiler does not support ANSI C++ features).

- **namespace**

A namespace is a part of the program in which certain names are recognized; outside of the namespace they're unknown. The directive

using namespace std;

says that all the program statements that follow are within the std namespace. Various program components such as cout are declared within this namespace. If we didn't use the using directive, we would need to add the std name to many program elements.

For example, in above program, we need to write

std::cout << "C++ is better than C";

To avoid adding std:: dozens of times in programs we use the using directive instead.

- **comments**

Comments are an important part of any program. They help the person writing a program, and anyone else who must read the source file, understand what's going on. The compiler ignores comments, so they do not add to the file size or execution time of the executable program.

Comments start with a double slash symbol (//) and terminate at the end of the line. A comment may start anywhere in the line and the whatever follows till the end of line is ignored. Note that there is no closing symbol.

```
/*This is an example of c++
program to use of comments*/

#include <iostream> //preprocessor directive
using namespace std;
int main() //function name "main"
{
 //start function body
cout <<"C++ is better than C"; //statement
return 0;
//end function body
}
```

The C comments symbols /* */ are still valid and more suitable for multiline comments

Console Input/Output Streams

Every program takes some data as input and generates processed data as an output following the familiar input process output cycle. It is essential to know how to provide the input data and present the results in the desired form. The use of the **cin** and **cout** is already known with the operator >> and << for the input and output operations. To control the way the output is printed, C++ supports a rich set of I/O functions and operations.

C++ supports a rich set of I/O functions and operations. These functions use the advanced features of C++ such as classes, derived classes, and virtual functions.

It also supports all of C's set of I/O functions and therefore can be used in C++ programs, but their use should be restrained due to two reasons.

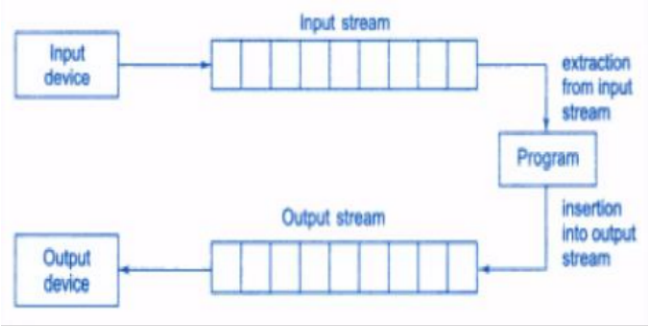
- ✓ I/O methods in C++ support the concept of OOPs.
- ✓ I/O methods in C cannot handle the user-defined data type such as class and object.

C++ uses the concept of stream and stream classes to implement its I/O operations with the console and disk files.

C++ Streams

The I/O system in C++ is designed to work with a wide variety of devices including terminals, disks, and tape drives. Although each device is very different, the I/O system supplies an interface to the programmer that is independent of the actual device being accessed. This interface is known as the stream.

- ✓ A stream is a sequence of bytes.
- ✓ The source stream that provides the data to the program is called the input stream.
- ✓ The destination stream that receives output from the program is called the output stream.
  In other words, a program extracts the bytes from an input stream and insert bytes into an output stream as illustrated in figure.



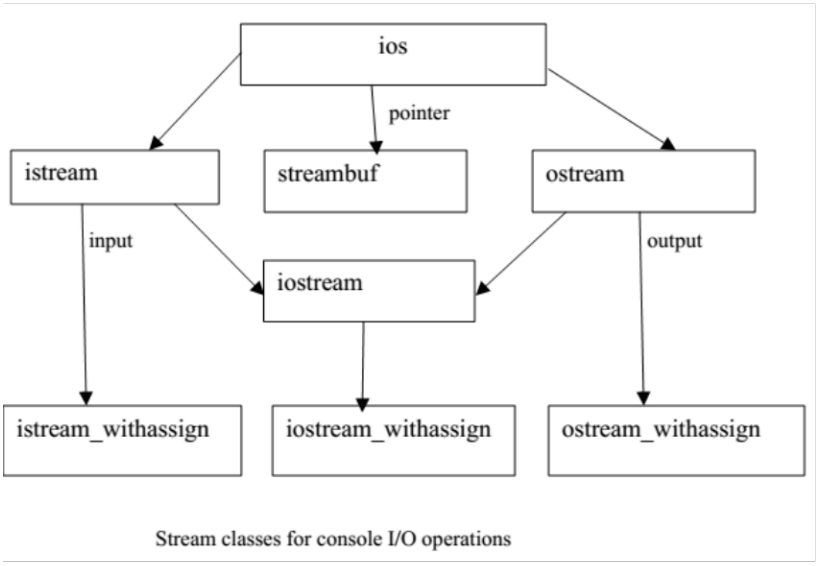
- ✓ The data in the input stream can come from the keyboard or any other input device.
- ✓ The data in the output stream can go to the screen or any other output device.

C++ contains several pre-defined streams that are automatically opened when a program begins its execution. These include **cin** and **cout**. It is known that **cin** represents the input stream connected to the standard input device (usually the keyboard) and **cout** represents the output stream connected to the standard output device (usually the screen).

C++ Stream Classes

The C++ I/O system contains a hierarchy of classes that are used to define various streams to deal with both the console and disk files. These classes are called stream classes. Figure below shows the hierarchy of the stream classes used for input and output operations with the console unit. These classes are declared in the header file iostream. This file should be included in all the programs that communicate with the console unit.

As seen in the figure ios is the base class for **istream**(input stream) and **ostream** (output stream) which are in turn, base classes for iostream(input/output stream). The class ios is declared as the virtual base class so that only one copy of its members are inherited by the **iostream.**



Stream classes for console I/O operations

The class **ios** provides the basic support for formatted and unformatted I/O operations. The class **istream** provides the facilities for formatted and unformatted input while the class **ostream**(through inheritance) provides the facilities for formatted output. The class **iostream** provides the facilities for handling both input and output streams. Three classes, namely **istream_withassign**, **ostream_withassign** and **iostream_withassign** add assignment operators to these classes. Table gives the details of these classes.

Class name	Contents
ios(General input/output stream class)	Contains basic facilities that are usedby all other input and output classes Also contains a pointer to buffer object(streambuf object) Declares constants and functions that are necessary for handling formatted input and output operations
istream(input stream)	Inherits the properties of ios Declares input functions such as get(),getline() and read() Contains overloaded extraction operator>>
ostream(output stream)	Inherits the property of ios Declares output functions put() and write() Contains overloaded insertion operator <<
iostream (input/output stream)	Inherits the properties of ios stream and ostream through multiple inheritance and thus contains all the input and output functions
streambuf	Provides an interface to physical devices through buffer Acts as a base for filebuf class used ios files

Manipulators

Manipulators are operators that are used to format the data display. The most commonly used manipulators are endl and setw.

The endl manipulator ,when used in an output statement, causes linefeed to be inserted. It has the same effet as using the newline character "\n". For example the statement

. . . . . .

cout<<"m= "<<m<<endl

<<"n= "<<n<<endl

<<"p= "<<p<<endl;

. . . . . .

. . ..

would cause three lines of output, one for each variable. If we assume the values of the variables as 2597 ,14 and 175 respectively, the output will appear as follows:

m=	2	5	9	7
n=	1	4		
p=	1	7	5	

It is important to note that this form is not the ideal output. It should rather appear as under:

m= 2597

n= 14

p= 175

Here the numbers are right justified. This form of output is possible only if we can specify a common field width for all the numbers and force them to be printed right-justified. The **setw**

manipulator does this job. It is used as follows.

cout<<setw(5)<<sum<<endl;

The manipulator setw(5) specifies a field width 5 for printing the value of the variable sum. This value is right justified within the field as shown below.

		3	4	5

Program illustrates the use of endl and setw

```
#include<iostream>
#include<iomanip>
using namespace std;
int main()
{
        int basic=950,allowance=95,total=1045;
        cout<<setw(10)<<"Basic"<<setw(10)<<basic<<endl;
        cout<<setw(10)<<"Allowance"<<setw(10)<<allowance<<endl;
        cout<<setw(10)<<"Total"<<setw(10)<<total<<endl;
        return 0;
}
```

output

```
     Basic       950
 Allowance        95
     Total      1045
```

Structure in C++

- Structure is the collection of variables of different types under a single name for better visualization of problem.
- Structure can hold data of one or more types.

The **struct** keyword defines a structure type followed by an identifier (name of the structure). Then inside the curly braces, you can declare one or more members (declare variables inside curly braces) of that structure. For example:

```
struct Person
{
   char name[50];
   int age;
   float salary;
};
```

Here a structure person is defined which has three members: name, age and salary.

Once you declare a structure person as above. You can define a structure variable as:
Person p1;
Here, a structure variable p1 is defined which is of type structure Person.
When structure variable is defined, only then the required memory is allocated by the compiler. Considering having 16 bit system, the memory of float is 4 bytes, memory of int is 2 bytes and memory of char is 1 byte. Hence, 56 bytes of memory is allocated for structure variable p1.

Accessing members of structures

The members of structure variable is accessed using a dot (.) operator.
Suppose we want to access age of structure variable p1 and assign it 25 to it. You can perform this task by using following code below:
p1.age = 25;

Example: C++ Structure

C++ Program to assign data to members of a structure variable and display it.

```
#include <iostream>
using namespace std;
struct Person
{
   char name[50];
   int age;
   float salary;
};
int main()
{
   Person p1;
   cout << "Enter Full name: ";
   cin.get(p1.name, 50);
   cout << "Enter age: ";
   cin >> p1.age;
   cout << "Enter salary: ";
   cin >> p1.salary;
   cout << "Displaying Information." << endl;
   cout << "Name: " << p1.name << endl;
   cout <<"Age: " << p1.age << endl;
   cout << "Salary: " << p1.salary;
   return 0;
}
```

Output

```
Enter Full name: Ram Karki
Enter age: 45
Enter salary: 56750.8
Displaying Information
Name: Ram Karki
Age: 45
Salary: 56750.8
--------------------------------
```

To read the text containing blank space, cin.get function can be used. This function takes two arguments. First argument is the name of the string (address of first element of string) and second argument is the maximum size of the array.

Differences Between the C and C++ Structures

C Structures	C++ Structures
It can have only data members and not member functions.	It can have both data members and functions
Cannot have a constructor inside a structure.	Constructor creation is allowed.
Direct Initialization of data members is not possible.	Direct Initialization of data members is possible.
Writing the 'struct' keyword is necessary to declare structure-type variables.	Writing the 'struct' keyword is not necessary to declare structure-type variables.
Do not support access modifiers.	Supports access modifiers.
Only pointers to struct are allowed.	Can have both pointers and references to the struct.
Sizeof operator will generate 0 for an empty structure.	Sizeof operator will generate 1 for an empty structure.
Data Hiding is not possible.	Data Hiding is possible.
Cannot have static members.	Can have static members.

Example 1:
Structures in C cannot have member functions inside a structure but Structures in C++ can have member functions along with data members.

```
#include <iostream>
using namespace std;
struct marks
 {
        int num;
        void Set(int temp)
         {
          num = temp;
         }

        void display()
         {
         cout << "num=" << num;
         }
};

int main()
{
        marks m1;
        m1.Set(9);
        m1.display();
        return 0;
}
```

Output

```
num=9
```

This will generate an error in C.

Example 2:

We cannot directly initialize structure data members in C but we can do it in C++.

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

struct Record
{
   int x = 7;
};
int main()
{
   Record s;
   cout << s.x << endl;
   return 0;
}
```

output

```
7
```

This will generate an error in C.

Class

A class serves as a blueprint or a plan or a template for an object. It specifies what data and what functions will be included in objects of that class. Once a class has been defined, we can create any number of objects belonging to that class. An object is an instance of a class. A class binds the data and its associated functions together. It allows the data (and function) to be hidden, if necessary, from external use.

Example:

Class Car	Class Computer
Properties Company, Model Color, Capacity	**Properties** Brand, Price, Screen Resolution HDD size, RAM size
Behavior Speed(), Acceleration(), Break()	**Behavior** Processing(), Display(), Printing()

Declaration of class

Class declaration describes the type and scope of its members. The general form of class declaration is:

```
class class_name
{
private:
variable declaration;
function declaration;
public:
variable declaration;
function declaration;
};
```

Example:

```
class student
{
private:
char name[20];
int roll;
public:
void getinfo();
void display();
};
```

- Here, **class** is C++ keyword; **class_name** is name of class defined by user.
- The body of class enclosed within braces and terminated by a semicolon.
- The class body contains the declaration of variables and functions. The variables declared inside the class are known as data members and the functions are known as member functions. These functions and variables are collectively called as class members.
- Keyword private and public within class body are known as visibility labels. i.e. they specify which of the members are private and which of them are public.
- The class members that have been declared as private can be accessed only from within the class but public members can be accessed from outside of the class also. Using private declaration, data hiding is possible in C++ such that it will be safe from accidental manipulation.
- The use of keyword private is optional. By default, all the members are private. *Usually, the data within a class is private and the functions are public.*
- C++ provides a third visibility modifier, protected, which is used in inheritance. A member declared as protected is accessible by the member functions within its class and any class immediately derived from it.
- **The binding of a data and functions together into a single class-type variable is referred to as encapsulation.**

```
#include<iostream>
using namespace std;
class student
{
private:
char name[20];
int roll;
public:
void getinfo()
{
cout<<"Enter name"<<endl;
cin>>name;
cout<<"Enter Rollno"<<endl;
cin>>roll;

}
void display()
{
cout<<"Name:"<<name<<endl;
cout<<"Roll No:"<<roll<<endl;
}
};
int main()
{
student st;
st.getinfo();
st.display();
return 0;
}
```

Objects:

An **Object** is an instance of a Class. When a class is defined, no memory is allocated but when it an object is created memory is allocated. To use the data and access functions defined in the class, we need to create objects.

Syntax for creating an object is

class_name object_name;

e.g.

Student st; // here Student is assumed to be a class name

The statement student st; creates a variable st of type Student. This variable st is known as object of the class Student. Thus, class variables are known as objects.

Key differences between a class and its objects

- Once we have defined a class, it exists all the time a program is running whereas objects may be created and destroyed at runtime.
- For single class there may be any number of objects.
- A class has unique name, attributes, and methods. An object has identity, state, and behavior.

Accessing the class member

The data member functions of class can be accessed using the **dot(.)** operator with the object. For example if the name of object is ***st*** and you want to access the member function with the name ***getInfo()*** then you will have to write ***st.getinfo();***
The public data members are also accessed in the same way given however the private data members are not allowed to be accessed directly by the object. Accessing a data member depends solely on the access control of that data member. This access control is given by **Access modifiers in C++.**

```
class Student
{
private:
int age;
public:
int rollno;
getInfo();
displayInfo();
. . . . . .
}
. . .. . . . . . .. . . .
int main()
{
Student st;
st.age=25;   // error, age is private
st.rollno=5; //ok, roll number is public
st.getinfo(); //Ok
. . . .. . . . .. . . . .
}
```

Defining Member functions

Member function can be defined in two places:

- Outside the class definition
- Inside the class definition

Defining Member function inside the class definition	Defining Member function Outside the class definition
In this case, a member function is defined at the time of its declaration. The function declaration is replaced by the function definition inside the class body.	In this technique, only declaration is done within class body. The member function that has been declared inside a class has to be defined separately outside the class. The syntax for defining the member function is as **return_type class_name::function_name(argument_list)** **{** **//function body** **}** Here, membership label class-name :: tells the compiler that the function function_name belongs to the class class-name. The symbol **::** is called scope resolution operator.

class Item
{
private:
int number;
float cost;
public:
void getdata()
{
. . . . . .
}
void display()
{
.. . .
}
};
	class Item
	{
	private:
	int number;
	float cost;
	public:
	void getdata();
	void display();
	};
	void Item::getdata()
	{
	. . . . . .
	}
	void Item::display()
	{
	. . . .
	}

#include<iostream>
using namespace std;
class Item
{
private:
int number;
float cost;
public:
void getdata()
{
cout<<"Enter the Item number"<<endl;
cin>>number;
cout<<"Enter the cost of item"<<endl;
cin>>cost;
}
void display()
{
cout<<"Item number="<<number<<endl;
cout<<"Cost of item="<<cost<<endl;
}
};
 	
int main()
{
Item x;
x.getdata();
x.display();
return 0;
}
	#include<iostream>
	using namespace std;
	class Item
	{
	private:
	int number;
	float cost;
	public:
	void getdata();
	void display();
	};
	void Item::getdata()
	{
	cout<<"Enter the Item number"<<endl;
	cin>>number;
	cout<<"Enter the cost of item"<<endl;
	cin>>cost;
	}
	void Item::display()
	{
	cout<<"Item number="<<number<<endl;
	cout<<"Cost of item="<<cost<<endl;
	}
	
	int main()
	{
	Item x;
	x.getdata();
	x.display();
	return 0;
	}

In this example, the functions getdata() and display() are defined within body of class item.	In this example, getdata() and display() functions are member functions of class item. They are declared within class body and they are defined outside the class body. While defining member functions, item:: (membership identity label) specifies that member functions belong to the class item. Other are similar to normal function definition.

Some characteristics of the member functions

- Several different classes can use the same function name (function overloading). The membership level will resolve their scope.
- Member functions can access the private data of the class. A non-member function cannot do so.
- A member function can call another member function directly, without using the dot operator.
- A private member function cannot be called by other function that is not a member of its class.

Array of objects

An object of class represents a single record in memory, if we want more than one record of class type, we have to create an array of object. As we know, an array is a collection of similar date type, we can also have arrays of variables of type class. Such variables are called array of object. First we define a class and then array of objects are declared. An array of objects is stored inside the memory in the same way as multidimensional array.

```
#include<iostream>
using namespace std;
class Employee
{
   char name[25];
   int age;
   float salary;
   public:
   void getdata();
   void display();
};
 void Employee::getdata()
   {
     cout<<"Enter Employee Name:";
     cin>>name;
     cout<<"Enter Employee Age :";
     cin>>age;
     cout<<"Enter Employee Salary:";
     cin>>salary;
   }
   void Employee::display()
   {
    cout<<"Name:"<<name<<endl;
    cout<<"Age:"<<age<<endl;
    cout<<"Salary:"<<salary<<endl;
   }
```

```
int main()
{
int i;
Employee e[10];         //Creating Array of 10 employees
for(i=0;i<10;i++)
{
 cout<<"Enter details of "<<i+1<<" Employee"<<endl;
 e[i].getdata();
}
for(i=0;i<10;i++)
{
cout<<"Details of Employee"<<i+1<<endl;
e[i].display();
}
return 0;
}
```

WAP to define the class in C++ as shown in class diagram .

Student
Name Roll Address Percentage
input() display()

input():to input initial values
display():to display the record of students who passed
Note: 45% is pass percentage and Perform above operations for 5 students.

```
#include<iostream>
using namespace std;
class student
{
private:
char name[20],address[20];
int roll;
float per;
public:
void input()
{
cout<<"Enter the name";
cin>>name;
cout<<"Enter the address";
cin>>address;
cout<<"Enter the roll";
cin>>roll;
cout<<"Enter the percentage";
cin>>per;
}
```

```
void display()
{
if(per>=45)
{
cout<<"Name="<<name<<endl;
cout<<"Roll="<<roll<<endl;
cout<<"Address="<<address<<endl;
cout<<"Percentage="<<per<<endl;
}
}
};

int main()
{
int i;
student st[5];
for(i=0;i<5;i++)
{
cout<<"Enter the information of student"<<i+1<<endl;
st[i].input();
}
for(i=0;i<5;i++)
{
st[i].display();
}
return 0;
}
```

Class Diagram and Object Diagram

Class diagram

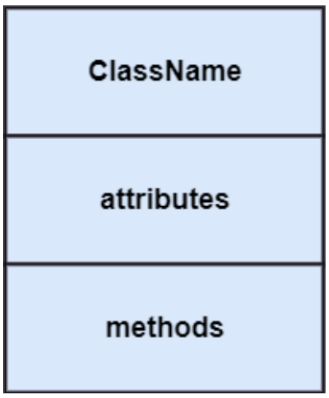
class diagram in the Unified Modeling Language (UML) is a type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects.

The class diagram is made up of three sections.

Upper section: The upper section contains the name of the class.

Middle section: The middle section contains the attributes of the class. The attribute type is shown after colon. Attributes are mapped onto data members in code. The attributes are written along with its visibility factors, which are public (+), private (-), protected (#), and package (~). The accessibility of an attribute class is illustrated by the visibility factors.

Lower section: Lower section contains the methods and operations. The methods are represented in the form of a list, where each method is written in a single line. The return type of a method is shown after the colon at the end of the method signature. The data type of method parameters is shown after the colon following the parameter name. Operations map onto class methods in code.



Class Relationships

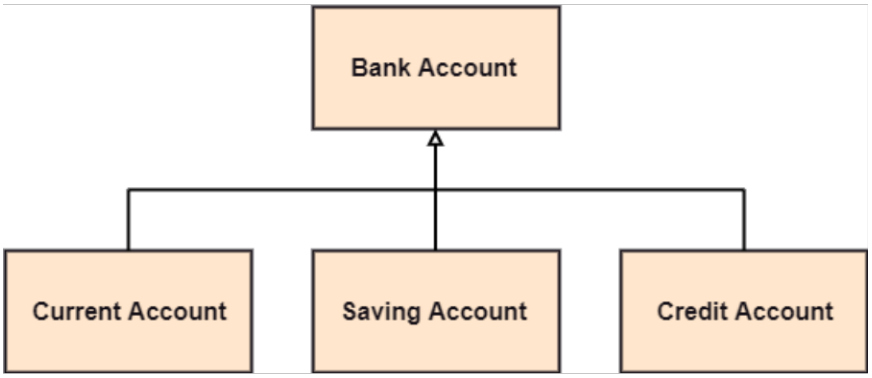
A class may be involved in one or more relationships with other classes. A relationship can be one of the following types.

Dependency: A dependency is a semantic relationship between two or more classes where a change in one class causes changes in another class. It forms a weaker relationship.

In the following example, Student_Name is dependent on the Student_Id.

Generalization: A generalization is a relationship between a parent class (superclass) and a child class (subclass). In this, the child class is inherited from the parent class.

For example, The Current Account, Saving Account, and Credit Account are the generalized form of Bank Account.



Association: It describes a static or physical connection between two or more objects. It depicts how many objects are there in the relationship.

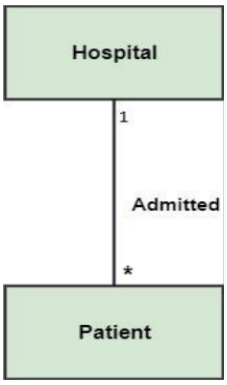
For example, a department is associated with the college.

- **Multiplicity:** It defines a specific range of allowable instances of attributes. In case if a range is not specified, one is considered as a default multiplicity.

  How many objects of each class take part in the relationships and multiplicity can be expressed as:

  - Exactly one - 1
  - Zero or one - 0..1
  - Many - 0..* or *
  - One or more - 1..*
  - Exact Number - e.g. 3..4 or 6

- Or a complex relationship - e.g. 0..1, 3..4, 6.* would mean any number of objects other than 2 or 5



For example, multiple patients are admitted to one hospital.

Aggregation: An aggregation is a subset of association, which represents has a relationship. It is more specific then association. It defines a part-whole or part-of relationship. In this kind of relationship, the child class can exist independently of its parent class.

The company encompasses a number of employees, and even if one employee resigns, the company still exists.

Composition: The composition is a subset of aggregation. It portrays the dependency between the parent and its child, which means if one part is deleted, then the other part also gets discarded. It represents a whole-part relationship.

A contact book consists of multiple contacts, and if you delete the contact book, all the contacts will be lost.

Example:

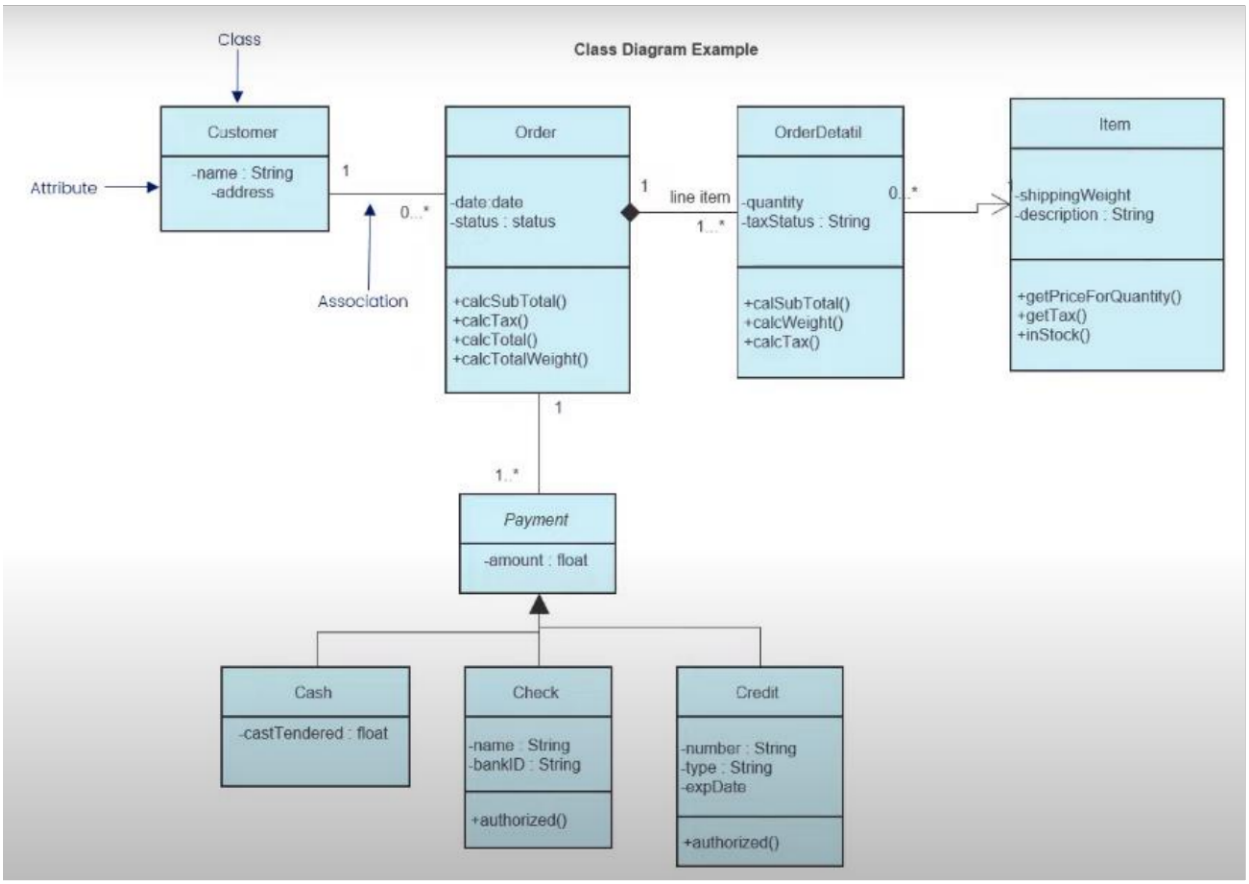


figure: class diagram for online order system

Example 2:

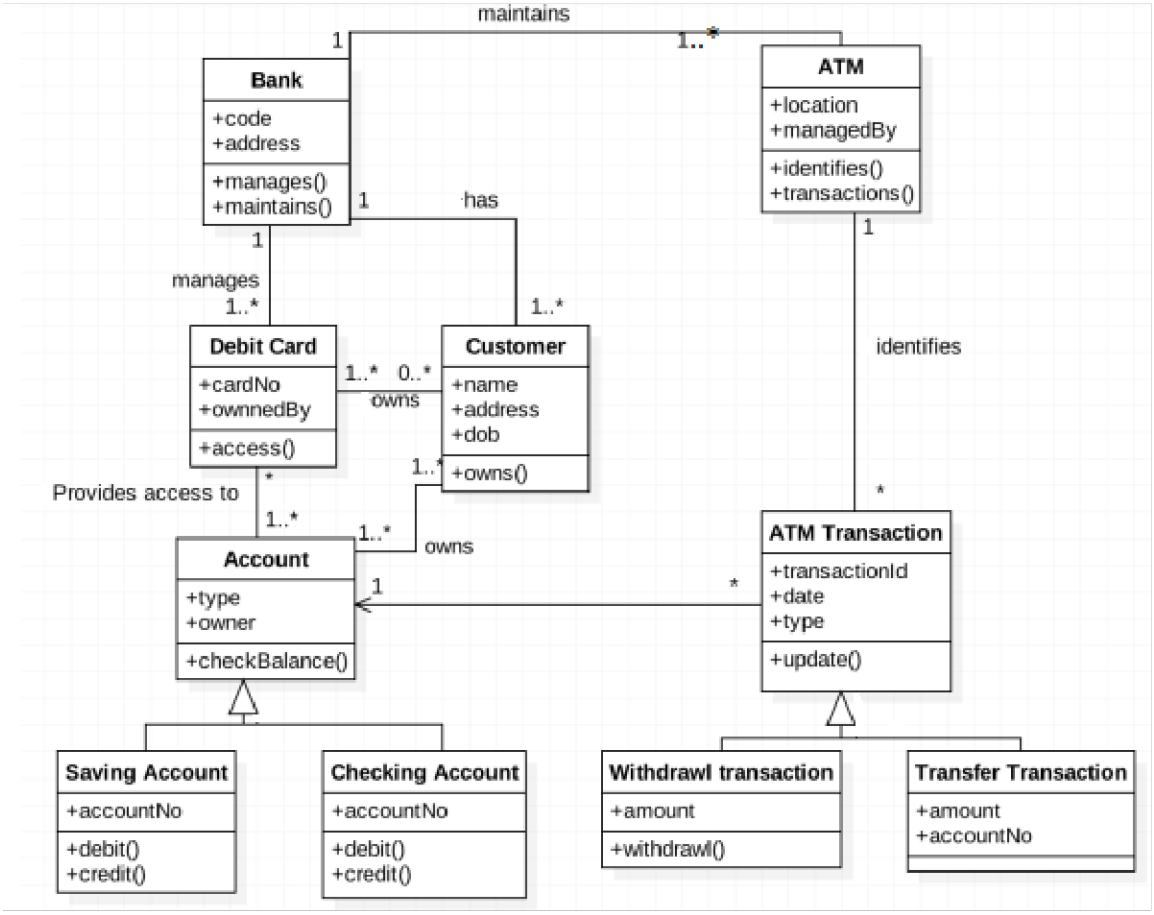


figure: class diagram for ATM system

Assignment:

Draw class diagram for

- ❖ Library management system
- ❖ Banking system
- ❖ Hospital Management system

Object diagram

Object is an instance of a class in a particular moment in runtime that can have its own state and data values. Likewise a static UML object diagram is an instance of a class diagram; it shows a snapshot of the detailed state of a system at a point in time, thus an object diagram encompasses objects and their relationships which may be considered a special case of a class diagram or a communication diagram.

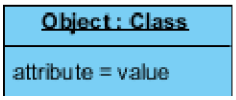
Basic Object Diagram Symbols and Notations

Object Names:

- Every object is actually symbolized like a rectangle, that offers the name from the object and its class underlined as well as divided with a colon.

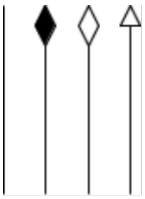
Object Attributes:

- Similar to classes, you are able to list object attributes inside a separate compartment. However, unlike classes, object attributes should have values assigned for them.



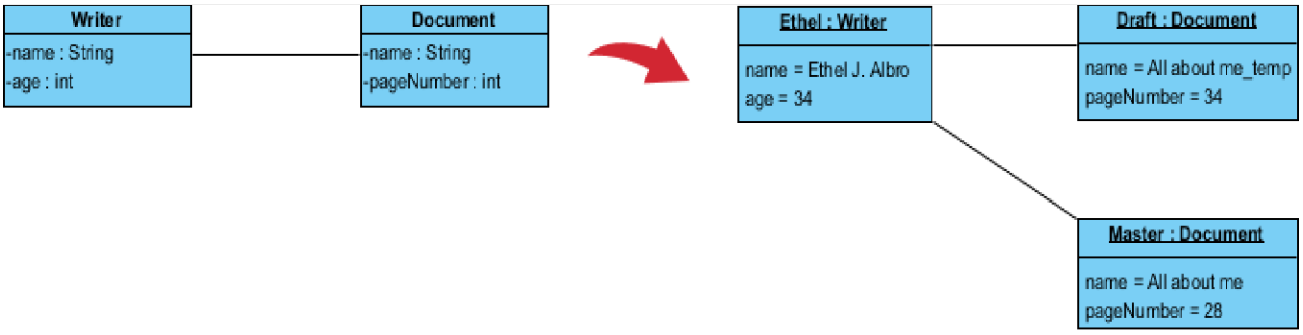
Links:

- Links tend to be instances associated with associations. You can draw a link while using the lines utilized in class diagrams.



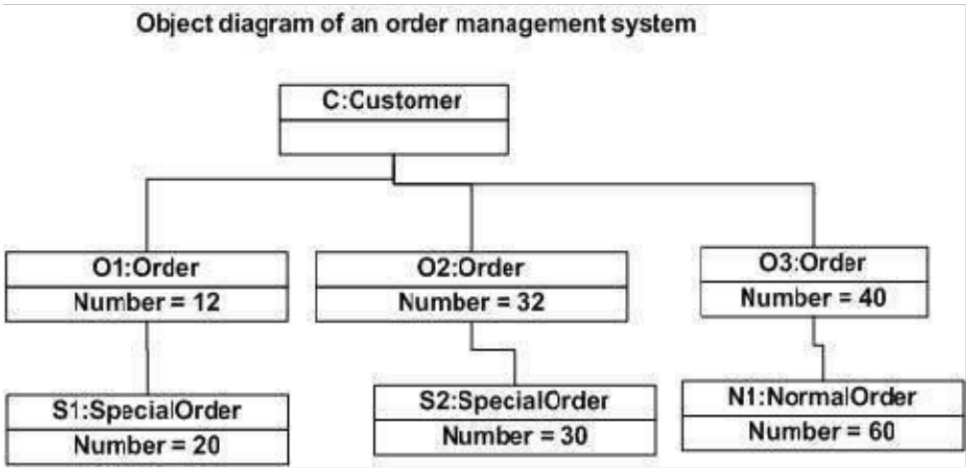
Example 1:

Object Diagram of writer



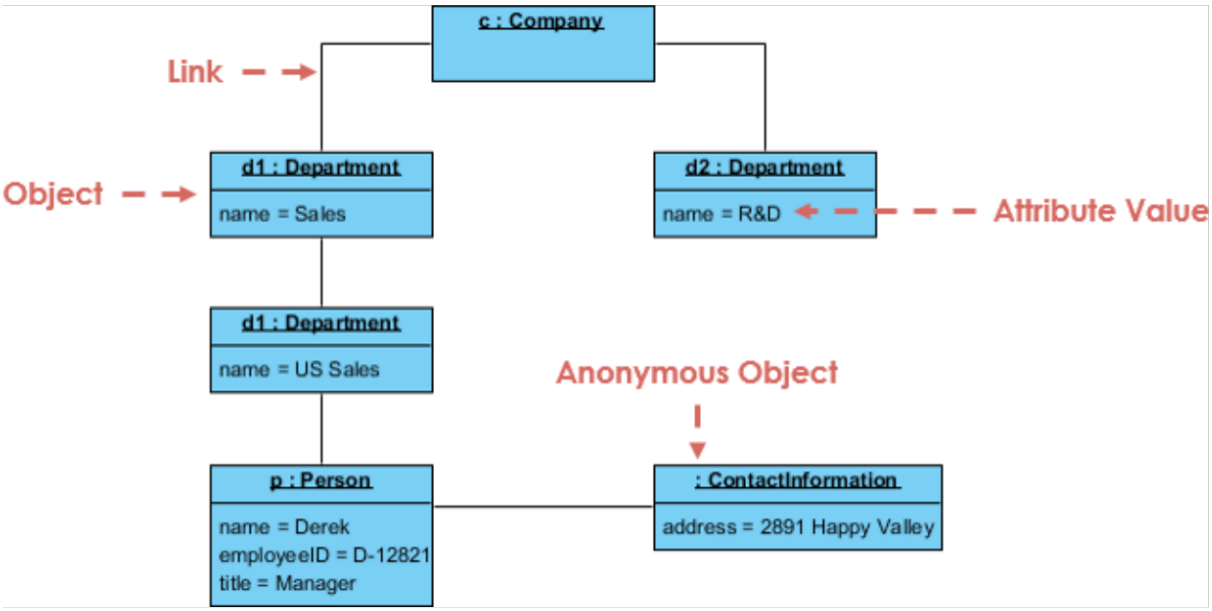
Example 2:

Object Diagram of order management system



Example 3:

Object Diagram of company structure



Access specifiers and visibility modes

Access specifiers

C++ access specifiers are used for determining or setting the boundary for the availability of class members (data members and member functions) beyond that class. That is, it sets some restrictions on the class members so that they cannot be accessed outside the class. By default, the class members are private. So, if the access specifiers are missing then by default all the class members are private.

There are 3 types of access specifiers available in C++.

They are:

1) Public:

All the class members declared under public will be available to everyone. The data members and member functions declared public can be accessed by other classes too. The public members of a class can be accessed from anywhere in the program using the direct member access operator (.) with the object of that class.

2) Private:

The class members declared as private can be accessed only by the functions inside the class. They are not allowed to be accessed outside the class. Only the member functions of that class or the friend functions are allowed to access the private data members of a class.

```
#include<iostream>
using namespace std;
class Rectangle
{
private:
float length,breadth,area;
public:
void setdata(float l,float b)
{
length=l;
breadth=b;
}
void disp_area()
{
area=length*breadth;
cout<<"Area="<<area<<endl;
}
};
```

```
int main()
{
Rectangle r;
r.setdata(10.5,6);
r.disp_area();
return 0;
}
```

Here, member function disp_area() is public so it is accessible outside class .But data member length ,breadth and area being private they are not accessible outside class. Only member function of that class disp_area() can access it.

3) Protected

class member declared as Protected are inaccessible outside the class but they can be accessed within class and by any subclass(derived class) of that class

Example:

```
#include <iostream>
using namespace std;
class A
{
protected:
int a, b;
public:
void setdata(int x, int y)
{
a = x;
b = y;
}
};
class B : public A
{
public:
void display()
{
cout << "value of a=" << a << endl;
cout << "value of b=" << b << endl;
cout << "Sum of two protected variables a and b = "<< a + b << endl;
}
};
int main()
{
B obj;
obj.setdata(4, 5);
obj.display();
return 0;
}
```

In above example , protected variable a and b is accessed by the derived class B.

From above discussion we can conclude that:

Access specifier	Accessible from own class	Accessible from derived class	Accessible from objects outside the class
private protected public	YES YES YES	NO YES YES	NO NO YES

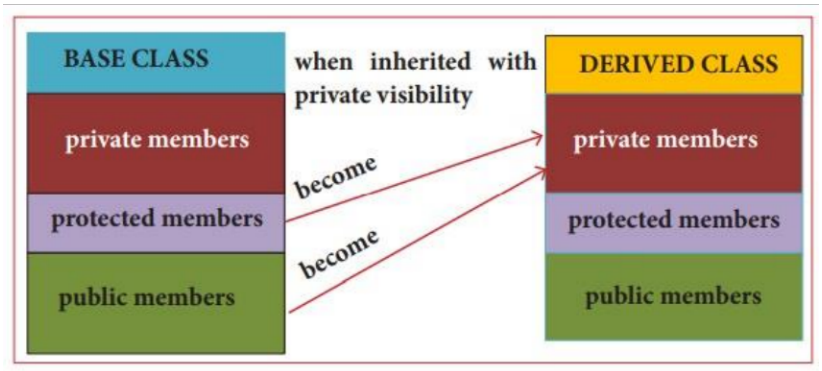
Visibility modes

An important feature of Inheritance is to know which member of the base class will be acquired by the derived class. This is done by using visibility modes.

- ✓ The accessibility of base class by the derived class is controlled by visibility modes.
- ✓ The three visibility modes are private, protected and public. The default visibility mode is private.
- ✓ Though visibility modes and access specifiers look similar, the main difference between them is **Access specifiers control the accessibility of the members within the class whereas visibility modes control the access of inherited members within the class**.

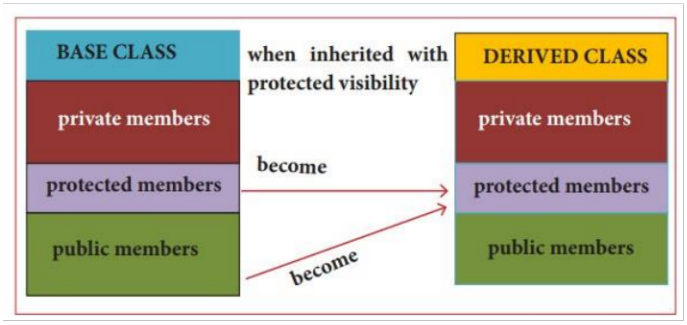
Private visibility mode

When a base class is inherited with private visibility mode the public and protected members of the base class become 'private' members of the derived class.



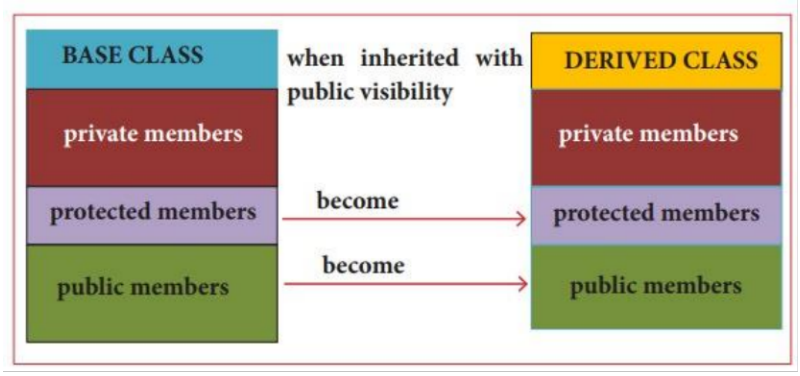
protected visibility mode

When a base class is inherited with protected visibility mode the protected and public members of the base class become 'protected members 'of the derived class.



public visibility mode

When a base class is inherited with public visibility mode, the protected members of the base class will be inherited as protected members of the derived class and the public members of the base class will be inherited as public members of the derived class**.**



//Implementation of Single Inheritance using public visibility mode

```
#include <iostream>
using namespace std;
class Shape
{
private:
    int count;
protected:
    int width;
    int height;

public:
    void setWidth(int w)
    {
    width = w;
    }
        void setHeight(int h)
        {
         height = h;
        }
};
```

```
class Rectangle: public Shape
{
public:
int getArea()
{
return (width * height);
}
};

int main()
{
Rectangle r1;
r1.setWidth(5);
r1.setHeight(7);
cout<< "Total area: "<<r1.getArea() <<endl;
// Print the area of the object.
return 0;
}
```

Output

```
Total area: 35
```

The following table contain the members defined inside each class before inheritance

MEMBERS of class	visibility modes		
	Private	Protected	Public
Shape(base class)	int count;	int width; int height;	void setWidth(int) void setHeight(int)
Rectangle (derived class only with its defined members)			intgetArea();

The following table contain the details of members defined after inheritance

MEMBERS of class	visibility modes –public for acquiring the properties of the base class		
	Private	Protected	Public
Shape(base class)	int count;	int width; int height;	void setWidth(int) void setHeight(int)
Rectangle (derived class acquired the properties of base class with public visibility)	Private members of base classes are not directly accessible by the derived class	int width; int height;	int getArea(); void setWidth(int) void setHeight(int)

Suppose the class **rectangle is derived with protected visibility** then the properties of class rectangle will change as follows:

MEMBERS of class	visibility modes –protected for acquiring the properties of the base class		
	Private	Protected	Public
Shape(base class)	int count;	int width; int height;	void setWidth(int) void setHeight(int)
Rectangle (derived class acquired the properties of base class with protected visibility)	Private members of base classes are not directly accessible by the derived class	int width; int height; void setWidth(int) void setHeight(int)	intgetArea();

In case the class rectangle is derived with private visibility mode from its base class shape then the property of class rectangle will change as follows

MEMBERS of class	visibility modes –private for acquiring the properties of the base class		
	Private	Protected	Public
Shape(base class)	int count;	int width; int height;	void setWidth(int) void setHeight(int)
Rectangle (derived class acquired the properties of base class with private visibility)	Private members of base classes are not directly accessible by the derived class	int width; int height; void setWidth(int) void setHeight(int)	int getArea();

When you derive the class from an existing base class, it may inherit the properties of the base class based on its visibility mode. So one must give appropriate visibility mode depends up on the need.

- ✓ Private inheritance should be used when you want the features of the base class to be available to the derived class but not to the classes that are derived from the derived class.
- ✓ Protected inheritance should be used when features of base class to be available only to the derived class members but not to the outside world.
- ✓ Public inheritance can be used when features of base class to be available the derived class members and also to the outside world.

State and Behavior of Object

State

- State tells us about the type or the value of that object**.**
- It tells, "**What the objects have?"**
  Example: Student have a first name, last name, age, etc...
- An objects state is defined by the attributes (i.e. data members or variables) of the object.
- In OOP state is defined as data members.
- It is determined by the values of its attributes.

Behavior

- Behavior tells us about the operations or things that the object can perform.
- It tells **"What the objects do"**
- Example Student attend a course "OOP", "C programming" etc…
- An objects behavior is defined by the methods or action (i.e. Member functions) of the object.
- In OOP behaviors are defined as member function
- It determines the actions of an object.

Let us consider another example

Imagine a dog. This is an example of an object.

A state can be on or off. If it's asleep, its state is off. If it's awake, the state is on.

Behavior: If the dog is awake, what is it doing? Examples of behavior might be barking, sniffing, eating or drinking, pawing, jumping, playing, to name a few.

Responsibility of Object

An object must contain the data (attributes) and code (methods) necessary to perform any and all services that are required by the object. This means that the object must have the capability to perform required services itself or at least know how to find and invoke these services. Rather than attempt to further refine the definition, take a look at an example that illustrates this responsibility concept.

Consider standalone application when looking for an example to illustrate object responsibility. One of my favorite examples is that of a generic paint program that allows a user to select a shape from a pallet and then drag it to a canvas, where the shape is dropped and displayed. From the user's perspective, this activity is accomplished by a variety of mouse actions. First, the user hovers over a specific shape with the mouse (perhaps a circle), right clicks, drags the shape to a location on the canvas and then lifts his/her finger off the mouse.

This last action causes the shape to be placed at the desired location. In fact, what is really happening when our user's finger is lifted from the mouse button? We can use this action to illustrate our object responsibility concept.

Let's assume that when the finger is lifted from the mouse button that the resulting message from this event is simply: draw. Consider the mouse as an object and that the mouse is only responsible for itself as well. The mouse can, and should only do mouse things. In this context, when the finger is lifted from the mouse button, the mouse thing to do is to draw. But draw what? Actually, the mouse doesn't care. It is not the mouse's responsibility to know how to draw anything. Its responsibility is to signal when to draw, not what to draw.
If this is the case (remember that a circle was selected), how does the right thing get drawn? The answer is actually pretty straightforward; the circle knows how to draw itself. The circle is responsible for doing circle things—just like the mouse is responsible for doing mouse things. So, if you select a circle, the circle object that is created knows everything possible about how to use a circle in the application.

This model has a major implication. Because the mouse does not need to know anything about specific shape objects, all it needs to do to draw a shape is to send the message draw. Actually, this will take the form of a method called draw( ). Thus, more shapes can be added without having to change any of the mouse's behavior. The only constraint is that any shape object installed in the application must implement a draw( ) method. If there is no draw( ) method, an exception will be generated when the mouse mouse-up action attempts to invoke the object's non-existent draw( ) method. In this design methodology, there is a de facto contract between the mouse class and the shape class and these contracts are made possible by the powerful object-oriented design technique called polymorphism

Data Abstraction

Data abstraction refers to providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details. Data abstraction is a programming (and design) technique that relies on the separation of interface and implementation
Let's take a real-life example of AC, which can be turned ON or OFF, change the temperature, change the mode, and other external components such as fan, swing. But, we don't know the internal details of the AC, i.e., how it works internally. Thus, we can say that AC separates the implementation details from the external interface.

Data Abstraction can be achieved in following ways:
• Abstraction using classes
• Abstraction in header files
• Abstraction using access specifiers

Abstraction using classes: An abstraction can be achieved using classes. A class is used to group all the data members and member functions into a single unit by using the access specifiers. A class has the responsibility to determine which data member is to be visible outside and which is not.

Abstraction in header files: Another type of abstraction is header file. For example, pow() function available is used to calculate the power of a number without actually knowing which algorithm function uses to calculate the power. Thus, we can say that header files hides all the implementation details from the user.

Abstraction using Access specifiers: Access specifiers are the main pillar of implementing abstraction in C++. We can use access specifiers to enforce restrictions on class members.
For example:
•Members declared as public in a class, can be accessed from anywhere in the program.
• Members declared as private in a class, can be accessed only from within the class. They are not allowed to be accessed from any part of code outside the class.

We can easily implement abstraction using the above two features provided by access specifiers. Say, the members that defines the internal implementation can be marked as private in a class. And the important information needed to be given to the outside world can be marked as public. And these public members can access the private members as they are inside the class.

Example:

```
#include <iostream>
using namespace std;
class AbstractionExample
{
private:
int a, b;
public:
void setdata(int x, int y)
{
a = x;
b = y;
}
void display()
{
cout<<"a = " <<a << endl;
cout<<"b = " << b << endl;
}
};
```

```
int main()
{
AbstractionExample obj;
obj.setdata(10, 20);
obj.display();
return 0;
}
```

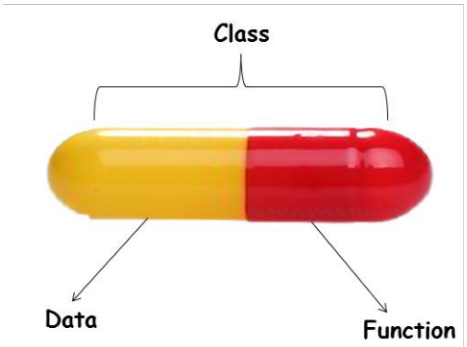
In above program ,By making data members private, we have hidden them from outside world. These data members are not accessible outside the class. The only way to set and get their values is through the public functions.

The main use of abstraction mechanism in C++ are as follows:

- To manage complexity that arises due to interconnections of software components using abstraction.
- Set of procedure written by one programmer can be used by many other programmers without knowing the exact details of implementation. They needed only the necessary interface. The program can use the function sort_name() to sort names in alphabetical order without knowing whether implementation uses bubble sort, merge sort, quick sort algorithms.
- To change the Internal implementation without affecting the user level code.
- To avoids the code duplication, i.e., programmer does not have to undergo the same tasks every time to perform the similar operation.
- To increase the readability of the code as it eliminates the possibility of displaying the complex working of the code.
- To increase security of an application or program as only important details are provided to the user.

Encapsulation

Encapsulation is a process of combining member functions and data members in a single unit called a class. The purpose is to prevent access to the data directly. Access to them is provided through the functions of the class. It is one of the popular features of Object-Oriented Programming (OOPs), which helps in data hiding.



Suppose we go to an automatic teller machine (ATM) and request money. The machine processes our requests and gives us money.

Here ATM is a class. It takes data from the user (money amount and PIN) and displays data in the form of icons and options. It processes the request(functions). So, it contains both data and functions wrapped/integrated under a single ATM. This is called Encapsulation.

Encapsulation can be enforced by:

- ✓ First, make all the data members private.
- ✓ Then getter(gets the value of data member) and setter (sets the value of data member)functions should be performed for each data member.

```
#include<iostream>
using namespace std;
class Square
{
private:
int num;
public:
void setnum(int x)
{
num=x;
}
int getnum()
{
return (num*num);
}
};
int main()
{
Square obj;
obj.setnum(5);
cout<<"Square of a number="<<obj.getnum()<<endl;
return 0;
}
```

In this program we are finding square of a given number. We have a data member num declared as private. That means it can be used within that class only and not even in the main (). If we want to use this function in the main function, we should use the getter and setter functions. So here we have a getter function getnum() and setter function setnum().

Inside main(), we create an object of the Square class. Now we can use the setnum() method to set the value of the number. Then we call the getnum() method on the object to return the square of the given number.

Data Hiding

Data Hiding means protecting the data members of a class from an illegal or unauthorized access from outside class. It ensures exclusive data access to class members and protects object integrity by preventing unintended or intended changes.
By declaring the data member private in a class, the data members can be hidden from outside the class. Those private data members cannot be accessed by the object directly.

```
#include<iostream>
using namespace std;
class Square
{
private:
int num;
public:
void getdata()
{
cout << "Enter the number"<<endl;;
cin>>num;
}
void display()
{
cout << "Square of a given number= "<< num*num<<endl;
}
};
int main()
{
Square obj;
obj.getdata();
obj.display();
return 0;
}
```

In the above example, the variable “num” is private. Hence this variable can be accessed only by the member function of the same class and is not accessible from anywhere else. Hence outside the classes will be unable to access this variable which is called data hiding. In this way data hiding can be achieved.

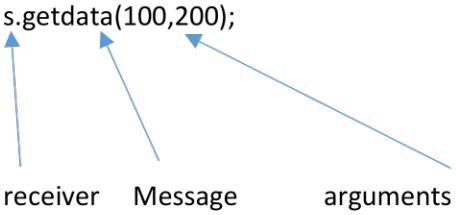
Message Passing

Message passing (sometimes also called method lookup) means the dynamic process of asking an object to perform a specific action.

- A message is always given to some object, called the receiver.
- The action performed in response to the message is not fixed but may be differ, depending on the class of the receiver .That is different objects may accept the same message the and yet perform different actions.

There are three identifiable parts to any message-passing expression. These are

1) Receiver: the object to which the message is being sent
2) Message selector: the text that indicates the particular message is being sent.
3) Arguments used in responding the message.



Example of message passing

```
#include<iostream>
using namespace std;
class student
{
int roll;
public:
void getdata(int x)
{
roll=x;
}
void display()
{
cout<<"Roll number="<<roll;
}
};
int main()
{
student s;
s.getdata(325); //objects passing message
s.display(); //objects passing message
return 0;
}
```

Message passing vs procedure calls

The message passing and procedure calls are differ in following aspects.

Message passing

- In a message passing there is a designated receiver for that message; the receiver is some object to which message is sent.
- Interpretation of the message (that is method is used to respond the message) is determined by the receiver and can vary with different receivers. That is, different object receive the same message and yet perform different actions.
- Usually, the specific receiver for any given message will not be known until run time, so determination of which method cannot be made until then. Thus, we say there is a late binding between the message (function or procedure name) and the code fragment (method) used to respond to the message.

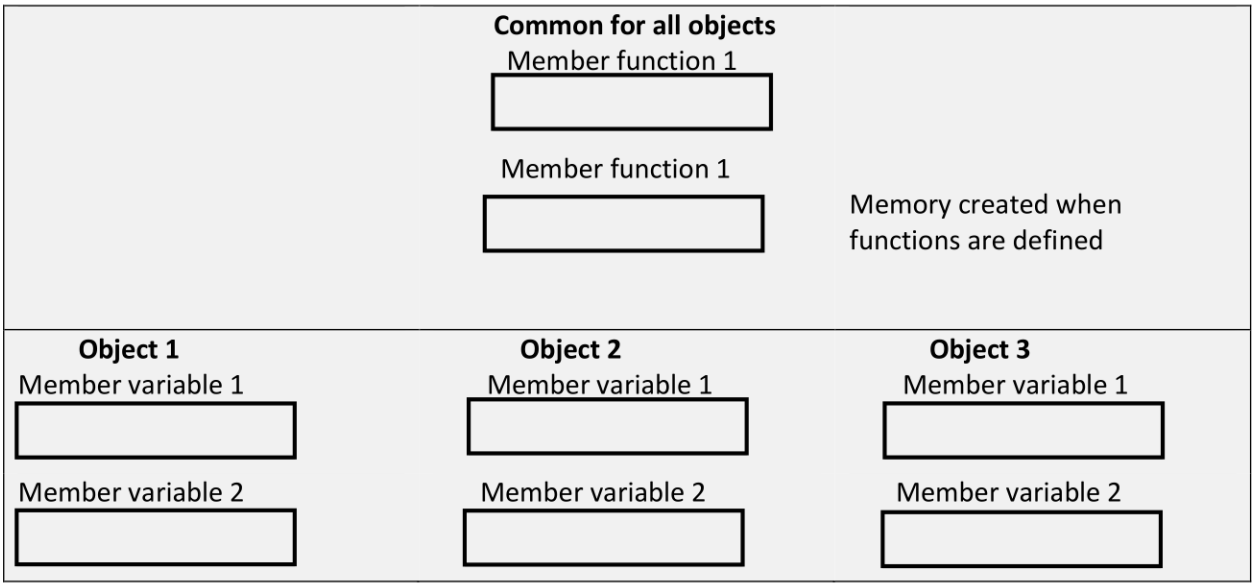
Procedure calls

- In a procedure call, there is no designated receiver.
- Determination of which method to invoke is very early(compile-time or link time) binding of a name to fragment in procedure calls.

Memory allocation for objects

For each object memory spaces for member variables is allocated separately, because the member variables will hold different data value for different objects.

The member functions are created and placed in the memory space only once when they are defined as a part of class specification. Since, all the objects belonging to that class use the same member functions, no separate space is allocated for member functions when objects are created.



Constructor

- A constructor is a special member function which initializes the objects of its class. It is called special because its name is same as that of the class name.
- The constructor is automatically executed whenever an object is created. Thus, a constructor helps to initialize the objects without making a separates call to a member function.
- It is called constructor because it constructs values of data members of a class.

Characteristics of constructor

- A constructor has same name as class name.
- They are invoked automatically when the objects are created.
- They should be declared in the public section.
- They do not have return types.
- They cannot be inherited, though a derived class can call the base class constructor.
- Like other C++ functions, they can have default arguments.
- Constructors cannot be **virtual**.
- We cannot refer to their addresses.
- An object with a constructor (or destructor) cannot be used as a member of a union.
- They make 'implicit calls' to the new and delete operators when a memory allocation is required.

Types of Constructors:

1. Default constructor

- A constructor that accepts no arguments (parameters) is called the default constructor.
- If a class does not include any constructor, the compiler supplies a default constructor.

The default Constructor is declared and defined as follows:

```
class Complex
{
private:
int real,imag;
public:
Complex
{
real=0;
imag=0;
}
. . . . . .
. . . . . .
};
```

When a class contains a constructor like the defined above, it is guaranteed that an object by class will be initialized automatically. For example, the declaration
Complex c1; not only creates the object **c1** of type **Complex** but also initializes its data members real and imag to zero. There is no need to write any statement to invoke the constructor function (as we do with normal member functions).

Program:

```
#include <iostream>
using namespace std;
class Complex
{
private:
int real,imag ;
public:
Complex()
{
        real=0;
        imag=0;
}
void display( )
{
 cout<<"Real part="<<real<<endl;
 cout<<"Imaginary part="<<imag<<endl ;
}
} ;
int main( )
{
Complex c1;
c1.display();
return 0;
}
```

Output:

```
Real part=0
Imaginary Part=0
```

2. Parameterized Constructor

The constructor that can take arguments are called parameterized constructor.
Arguments are passed when the objects are created. This can be done in two ways.

- by calling the constructor explicitly
- by calling the constructor implicitly

For example, the constructor used in the following example is the parameterized constructor and takes two arguments both of type int.

```
class Complex
{
private:
int real,imag;
public:
Complex (int r, int i) //parameterized constructor
{
real=r;
imag=i;
}
. . . . . .
. . . . . .
};
int main()
{
Complex c1(5,2); //implicit call
Complex c2 =Compex(7,4); //explicit call
. . . . . .
}
```

Example:

```
#include <iostream>
using namespace std;
class Complex
{
private:
int real,imag;
public:
Complex(int r,int i)
{
        real=r;
        imag=i;
}
 void display( )
  {
 cout<<"Real part="<<real<<endl;
 cout<<"Imaginary part="<<imag<<endl ;
  }
};
```

```
int main( )
{
Complex c1(5,2);
c1.display();
return 0;
}
```
Output:

```
Real part=5
Imaginary part=2
```

3) Copy Constructor

A copy constructor is used to declare and initialize an object with another object of the same type.
For example, the statement
Complex c2(c1);
Creates new object c2 and performs member-by-member copy of c1 into c2. Another form of this statement is
Complex c2 = c1;
The process of initializing through assignment operator is known as copy initialization.
A copy constructor takes reference to an object of the same class as its argument. For example,

```
Complex (Complex &x)
{
real=x.real;
Imag=x.imag;
}
```

Remember: ***We cannot pass the argument by value to a copy constructor***

When no copy constructor is defined, the compiler supplies its own copy constructor.

```
#include<iostream>
using namespace std;
class Complex
{
private:
int real, imag ;
public:
Complex( )
{
}
```

```
Complex (int r,int i)
{
real=r ;
imag=i ;
}
Complex (Complex &x)
{
real=x.real ;
imag=x.imag ;
}
void display( )
{
cout<<"Real part="<<real<<endl;
cout<<"Imaginary part="<<imag<<endl;
}
};


int main( )
{
Complex c1(5,10) ;
Complex c2(c1) ; // copy constructor called
Complex c3;
c3=c1;
cout<<"Details of c1"<<endl;
c1.display( ) ;
cout<< "Details of c2"<<endl;
c2.display( ) ;
cout<< "Details of c3"<<endl;
c3.display( ) ;
return 0;
}
```

Constructor Overloading

We can have more than one constructor in a class with same name and different number of arguments. This concept is known as Constructor Overloading.

- Overloaded constructors essentially have the same name (name of the class) and different number of arguments.
- A constructor is called depending upon the number and type of arguments passed.
- While creating the object, arguments must be passed to let compiler know, which constructor needs to be called.

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

```
class Complex
{
int real,imag;
public:
Complex() //Default Constructor
{
real=0;
imag=0;
}
Complex(int r,int i)  //Parameterized Constructor
{
real=r;
imag=i;
}
Complex(Complex &x) //Copy Constructor
{
real=x.real;
imag=x.imag;
}

void display()
{
cout<<"Real part:"<<real<<endl;
cout<<"Imaginary part"<<imag<<endl;
}
};
int main()
{
Complex c1;
Complex c2(5,2);
Complex c3(c2);
c1.display();
c2.display();
c3.display();
return 0;
}
```

In the above example of class Complex, we have defined three constructors. The first one is invoked when we don't pass any arguments. The second gets invoked when we supply two argument, while the third one gets invoked when an object is passed as an argument.

Example

Complex c1;

This statement would automatically invoke the first one.

Complex c2(102,300); invokes 2nd constructor

Complex c3(c2); invokes 3rd constructor

Destructors

- Destructor is a member function that destroys the object that have been created by a constructor.
- Destructor name is similar to class name but preceded by a tilde sign (~).
- They are also defined in the public section.
- Destructor never takes any argument, nor does it return any value. So, they cannot be Overloaded.
- Destructor gets invoked, automatically, when an object goes out of scope (i.e. exit from the program, or block or function).

  Example:
  A destructor for the class test will look like

  ~test()
  {
  - - - - - -
  - - - - - -
  }

```
#include<iostream>
using namespace std;
class test
{
static int count;
public:
test()
{
count++;
cout<<"object:"<<count<<"created"<<endl;
}
~test()
{
cout<<"object:"<<count<<"destroyed"<<endl;
count--;
}
};
int test::count;
```

```
int main()
{
cout<<"Inside the main block"<<endl;
test t1;
{
        //Block 1
        cout<<"Inside Block 1"<<endl;
        test t2,t3;
        cout<<"Leaving Block 1"<<endl;
}
cout<<"Back to main Block"<<endl;
return 0;
}
```

Output:

```
Inside the main block
object:1created
Inside Block 1
object:2created
object:3created
Leaving Block 1
object:3destroyed
object:2destroyed
Back to main Block
object:1destroyed
```

Dynamic Memory Allocation

The process of allocating or de-allocating a block of memory during the execution of a program is called Dynamic Memory Allocation. The operators new and delete are utilized for dynamic memory allocation in C++.

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.

***Syntax*:** pointer-variable = new data-type;

This expression is used to allocate memory to contain one single element of type data-type.

For Example:

```
int *ptr;
ptr=new int;
```

Similarly,

Syntax: pointer-variable = new data-type [size];
This one is used to allocate a block (an array) of elements of type data-type,
where size is an integer value representing the number of elements.

For Example:

```
int *ptr;
ptr = new int [5];
```

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.
- **delete** operator free the memory allocated to the variable i.e. it deletes the variables memory.

Syntax is:

```
delete pointer-variable;
delete [size]  pointer-variable;
```

- The first statement releases the memory of a single element allocated using new
- The second one releases the memory allocated for arrays of elements using new and a size in brackets ([ ]).

Note: The size specifies the number of elements in the array to be freed. The problem with this form is that programmer should remember the size of the array. Recent version of c++ do not require the size to be specified. for example,

```
delete[] ptr;
```

Example1:

```
#include<iostream>
using namespace std;
int main()
{
        int *ptr;
        ptr=new int;
        cout<<"Enter any number"<<endl;
        cin>>*ptr;
        cout<<"Entered number="<<*ptr<<endl;
        return 0;
}
```

Example2:

```
#include <iostream>
using namespace std;
int main ()
{
int i,size;
int *ptr;
cout << "How many numbers would you like to Enter? ";
cin >>size;
ptr= new int[size];
for (i=0; i<size; i++)
{
cout << "Enter number:"<<i+1<<endl;
cin >> ptr[i];
}
cout << "You have entered:"<<endl;
for (i=0; i<size; i++)
{
cout << ptr[i]<<endl ;
}
delete[] ptr;
return 0;
}
```

Dynamic Constructor

Dynamic constructor is used to allocate the memory to the objects at the run time. Memory is allocated at run time with the help of 'new' operator .This will enable the system to allocate right amount of memory for each object when objects are not of the same size, thus resulting in the saving of memory.
Allocation of memory to objects at the time of their construction is known as dynamic constructions of objects. The memory is allocated with the help of new operator**.**

Example:

```
#include<iostream>
using namespace std;
class sample
{
	private:
		int size;
		int *ptr;
	public:
		sample(int s)
		{
			size=s;
			ptr=new int[size];
		}
		void input()
		{
			cout<<"Enter the values"<<endl;
			for(int i=0;i<size;i++)
			{
				cin>>ptr[i];
			}
		}
		void output()
		{
			cout<<"The values entered by user are"<<endl;
			for(int i=0;i<size;i++)
			{
				cout<<ptr[i]<<endl;
			}
		}
		~sample()
		{
			delete[] ptr;
		}
};
int main()
{
	sample s1(5);
	sample s2(10);
	s1.input();
	s1.output();
	s2.input();
	s2.output();
	return 0;
}
```

Write a program to find the sum of two Complex number using dynamic constructor.

```
#include<iostream>
using namespace std;
class complex
{
int *real,*imag;
public:
complex()
{
real=new int;
imag=new int;
*real=0;
*imag=0;
}
complex(int r,int i)
{
real=new int;
*real=r;
imag=new int;
*imag=i;
}
void display()
{
cout<<*real<<"+i"<<*imag<<endl;
}
void addcomplex(complex c1,complex c2)
{
*real=*c1.real+*c2.real;
*imag=*c1.imag+*c2.imag;
}
~complex()
{
        delete real;
        delete imag;
}
};
```

```
int main()
{
complex c1(5,10);
complex c2(2,4);
complex c3;
cout<<"First complex number=";
c1.display();
cout<<"Second complex number=";
c2.display();
cout<<"Sum =";
c3.addcomplex(c1,c2);
c3.display();
return 0;
}
```

Write a program to concatenate two string using the concept of dynamic constructor.

```
#include<iostream>
#include<string.h>
using namespace std;
class stringc
{
char *name;
int length;
public :
stringc()
{
length=0;
name=new char[length + 1];
}
stringc(char s[])
{
length=strlen(s);
name=new char[length+1];
strcpy(name,s);
}

void join(stringc s1,stringc s2)
{
length=s1.length+s2.length;
delete name;
name=new char[length+1];
strcpy(name,s1.name);
strcat(name,s2.name);
}
```

```
void display()
{
cout<<name<<endl;
}
~stringc()
{
        delete[] name;
}
};
int main ()
{
stringc s1("pokhara");
stringc s2("university");
stringc s3;
s3.join(s1,s2);
s3.display();
return 0;
}
```

Inline functions

When function is called, it takes a lot of extra time in executing a series of instructions for tasks such as jumping to the function, saving register, pushing arguments into the stack, and returning to the calling function. When a function is small, a substantial percentage of execution time may be spent in such overheads.
To eliminate the cost of calls to small functions, C++ proposes a new feature called inline function. An inline function is a function that is expanded in line when it is invoked. That is, the compiler replaces the function call with the corresponding function code.

A function is made inline by using a keyword **inline** before the function definition. Inline functions must be defined before they are called.

Example:

```
#include<iostream>
using namespace std;
inline int max(int a, int b)
{
return (a>b)?a:b;
}
int main()
{
int x,y;
cout<<"Enter x and y:"<<endl;
cin>>x>>y;
cout<<"Maximum  value is:"<<max(x,y);
return 0;
}
```

Inline functions provide following advantages:

1) Function call overhead doesn't occur.
2) It also saves the overhead of push/pop variables on the stack when function is called.
3) It also saves overhead of a return call from a function.
4) When you inline a function, you may enable compiler to perform context specific optimization on the body of function. Such optimizations are not possible for normal function calls. Other optimizations can be obtained by considering the flows of calling context and the called context

Note:
Some of the situations where inline expansion may not work are

- For functions returning values, if a loop, a switch or a goto exists.
- For functions not returning values, if a return statement exists.
- If function contain static variables.
- If inline functions are recursive.

Default argument

C++ allows a function to assign a parameter the default value in case no argument for that parameter is specified in the function call. Default values are specified when the function is declared. Default value is specified in a manner syntactically similar to a variable initialization. If a value for that parameter is not passed when the function is called, the default value is used, but if a value is specified, this default value is ignored and the passed value is used instead.

For example:

float amount(float principal, int period, float rate=0.15);

The above prototype declares a default value of 0.15 to the argument rate.
A subsequent function call like
value=amount(5000,7); // one argument missing
Passes the value 5000 to principal and 7 to period and then lets a function use default value of 0.15 for rate,
The call
Value=amount(5000,5,0.12); // no missing argument
Passes the explicit value of 0.12 to rate.

Note:

- Default argument are useful in situations where some arguments always have the same value. Eg. bank interest may remain the same for all customers for a particular period of deposit.
- Only the trailing arguments can have default values and therefore we must add defaults from right to left.
- We cannot provide a default value to a particular argument in the middle of an argument list.

```
int add (int a, int b = 5, int c); // illegal
int add (int a = 5, int b, int c = 6); //  illegal
int add (int a  int b , int c = 5); // legal
int add (int a = 5, int b = 6, int c = 7); // legal
```

Example:

```
#include<iostream>
using namespace std;
int add(int a=1, int b=2, int c=3);
int main()
{
int x=5,y=10,z=15;
cout<<"Sum="<<add(x,y,z)<<endl;
cout<<"Sum="<<add(x,y)<<endl;
cout<<"Sum="<<add(x)<<endl;
cout<<"Sum="<<add()<<endl;
return 0;
}
int add(int a, int b, int c)
{
return (a+b+c);
}
```

Output:

```
Sum=30
Sum =18
Sum =10
Sum =6
```

Reference Variable

A reference variable provides an alias(alternative name) for a previously defined variable.
A reference variable can be created as follows:

data-type &reference-name = variable name;

```
eg  float total = 100;
float &sum = total; // creating reference variable for 'total'.
```

Example:

```
#include<iostream>
using namespace std;
int main()
{
        float total=100;
        float &sum=total;
        cout<<"Total="<<total<<endl;
        cout<<"Sum="<<sum<<endl;
        total=total+100;
        cout<<"Total="<<total<<endl;
        cout<<"Sum="<<sum<<endl;
        return 0;
}
```

Output:

```
Total=100
Sum=100
Total=200
Sum=200
```

In the above example, we are creating a reference variable 'sum' for an existing variable 'total'. Now these can be used interchangeably. Both of these names refer to same data object in memory. If the value is manipulated and changed using one name then it will change for another also. Eg- the statement
total = total + 100; will change value of 'total' to 200. And it will also change for 'sum'. So the statements
cout<<total;
cout<<sum; both will print 200. This is because both the variables use same data object in memory.

- A reference variable must be initialized at the time of declaration, since this will establish correspondence between the reference and the data object which it names.
- The symbol & is not an address operator here. The notation float & means reference to float type data.

Major application of reference variable is in passing arguments to function. Consider the following example.

```
#include<iostream>
using namespace std;
void fun(int &x);
```

```
int main()
{
int m;
m=10;
fun(m);
cout<<"Updated value="<<m;
return 0;
}
void fun(int &x)
{
x=x+10;
}
```

When the function call fun(m) is executed, the following initialization occurs:
int &x=m;
Thus x becomes an alias of m after executing the statement.
fun(m);

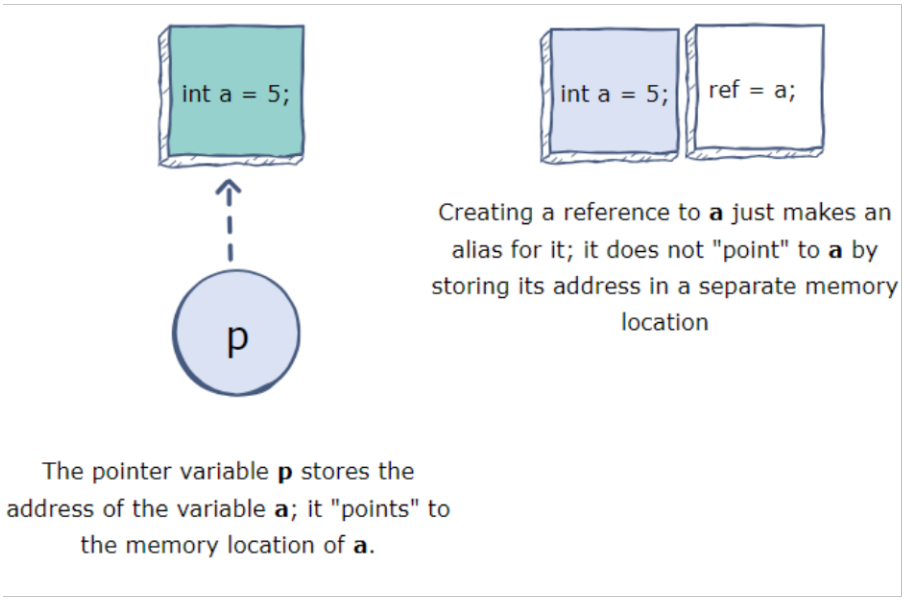
such function calls are known as call by reference. Since the variables x and m are aliases, when the function increments **x, m** is also incremented. The value of m becomes 20 after the function is executed.

Pointer and reference

C++ reference and pointer seem to be similar, but there are some differences that exist between them.

A pointer in C++ is a variable that holds the memory address of another variable.

A reference is an alias(another name) for an already existing variable. Once a reference is initialized to a variable, it cannot be changed to refer to another variable.



Key Differences

Pointer	Reference

It is not necessary to initialize it with a value during declaration.
 	
`int a = 5;`
`int *p;`
`p = &a;`
	It is necessary to initialize it with a value during declaration.
	
	`int a = 5;`
	`int &ref = a;`

It can be assigned a NULL value.	It cannot be NULL.
It must be dereferenced with a * to access the variable's value.	It does not need to be dereferenced and can be used simply by name.
Arithmetic operations can be performed on it.	Arithmetic operations can not be performed on it.

After declaration, it can be re-assigned to any other variable of the same type.
 	
`int a = 5;`
`int *p;`
`p = &a;`
`int b = 6;`
`p = &b;`
	It cannot be re-assigned to any other variable after its initialization.

Static data member

A data member of class can be qualified as static using the prefix static. A static member variable has following characteristics. They are:

- Static member variable of a class is **initialized to zero** when the first object of its class is created.
- **Only one copy** of that member is created for the entire class and is shared by all the objects of that class, no matter how many object are created.
- It is **visible only within the class** but its **life time is the entire program.**

The type and scope of each static member variable must be defined outside the class definition.

Static variables are normally used to maintain values common to the entire class. For example, a static data member can be used as a counter that records the occurrences of all the objects.

Example:

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

class item
{
static int count ; // static member variable
int number ;
public:
void getdata (int a)
{
number=a;
count++ ;
}
void displaycount()
{
cout<<"count:"<<count<<endl;
}
};
int item ::count;   //static member definition

int main()
{
item x,y,z ;                //count is initialized to zero
x.displaycount();          // display count
y.displaycount();
z.displaycount();
x.getdata(10) ;            //getting data into object x
y.getdata(20) ;             //getting data into object y
z.getdata(30) ;            //getting data into object z
cout<<"After reading data"<<endl ;
x.displaycount();         // display count
y.displaycount();
z.displaycount();
return 0;
}
```

Output:

```
Count:0
Count:0
Count:0
After reading data
Count:3
Count:3
Count:3
```

Static Member functions

Like static member variables, we can also have static member functions.
A member function that is declared static has the following properties.

- A static function can have access to only other static members (functions or variables) declared in the same class.
- A static member function can be called using the class name(instead of objects) as follows.

  class-name::function name;

```
#include<iostream>
using namespace std;
class test
{
private:
int code;
static int count;
public:
void setcode()
{
code=++count;
}
void showcode()
{
cout<<"code="<<code<<endl;
}
static void showcount()
{
cout<<"count="<<count<<endl;
}
};
int test::count;
int main()
{
        test t1,t2;
        t1.setcode();
        t2.setcode();
        test::showcount();
        test t3;
        t3.setcode();
        test::showcount();
        t1.showcode();
        t2.showcode();
        t3.showcode();
        return 0;
}
```

Output:

```
Count=2
Count=3
Code=1
Code=2
Code=3
```

Object as function arguments

Like any other data type, an object can be used as function argument.

This can be done in two ways:

- A copy of the entire object is passed to the function *(pass-by-value)*
- Only the address of the object is transferred to the function*(pass-by-reference)*

Let us illustrate use of objects as function arguments with the help of program which performs the addition of complex numbers.

```
#include<iostream>
using namespace std;
class Complex
{
private:
int real, imag;
public:
void getcomplex();
void addcomplex(Complex, Complex);
void display();
};

void Complex:: getcomplex()
{
cout<<"Enter real part:"<<endl;
cin>>real;
cout<<"Enter imaginary part:"<<endl;
cin>>imag;
}
void Complex::display()
{
cout<<real<<"+"<<imag<<"i"<<endl;
}
void Complex::addcomplex( Complex c1, Complex c2)
{
real=c1.real+c2.real;
imag=c1.imag+c2.imag;
}
```

```
int main()
{
Complex c1,c2,c3;
cout<<"For first complex number"<<endl;
c1.getcomplex();
cout<<"For second complex number"<<endl;
c2.getcomplex();
c3.addcomplex(c1,c2); //objects first and second passed as argument
cout<<"sum of two complex number=";
c3.display();
return 0;
}
```

Since the member function **addcomplex ()** is invoked by the object **c3** with the object **c1** and **c2** as arguments, it can directly access the real and imag variables of **c3**. But, the member of **c1** and **c2** can be accessed by using the dot operator (like c1.real and c1.imag). Therefore, inside the function **addcomplex(),** the variable **real** and **imag** refers to **c3**, c1.real and c1.imag refers to **c1** and c2.real and c2.imag refers to **c2.**

Output:

```
For first complex number
Enter real part:
2
Enter imaginary Part:
4
For second complex number
Enter real part:
1
Enter imaginary Part:
8
Sum of two complex number=3+12i
```

Let us consider another example which uses objects as function arguments with the help of program which performs the addition of time in the hour and minute format.

```
#include<iostream>
using namespace std;
class Time
{
private:
int hours;
int minutes;
```

```
public:
void gettime(int h,int m)
{
hours=h;
minutes=m;
}
void display()
{
cout<<hours<<"Hours and";
cout<<minutes<<"Minutes"<<endl;
}
void sum(Time t1,Time t2)
{
minutes = t1.minutes + t2.minutes;
hours = minutes/60;
minutes = minutes%60;
hours = hours + t1.hours + t2.hours;
}
};

int main()
{
Time t1,t2,t3;
t1.gettime(2,45);
t2.gettime(3,30);
t3.sum(t1,t2);
cout<<"First time=";
t1.display();
cout<<"Second time=";
t2.display();
cout<<"sum of two times=";
t3.display();
return 0;
}
```

Since the member function sum () is invoked by the object t3, with the object t1 and t2 as arguments, it can directly access the hours and minutes variables of t3. But, the member of t1and t2 can be accessed by using the dot operator (like t1.hours and t1.minutes). Therefore, inside the function sum(), the variable hours and minutes refers to t3, t1.hours and t1.minutes refers to t1 and t2.hours and t2.minutes refers to t2.

Output:

```
First time=2 Hours and 45 Minutes
Second time=3 Hours and 30 Minutes
Total time=6 Hours and 15 Minutes
```

Returning Object

A function cannot only receive object as arguments but also return them. The example in program below illustrates how an object can be created (within function) and returned to another function.

Now, let us consider the program to perform addition of complex number by passing object as arguments and returning object.

```
#include<iostream>
using namespace std;
class Complex
{
private:
int real;
int imag;
public:
void getdata();
Complex addcomplex(Complex,Complex);
void display();
};

void Complex::getdata()
{
cout<<"Enter the real part:"<<endl;
cin>>real;
cout<<"Enter the imaginary part :"<<endl;
cin>>imag;
}

void Complex::display()
{
cout<<real<<"+"<<imag<<"i"<<endl;
}
Complex Complex::addcomplex( Complex  c1, Complex c2)
{
Complex temp;
temp.real=c1.real+c2.real;
temp.imag=c1.imag+c2.imag;
return temp;
}
```

```
int main()
{
Complex c1,c2,c3,result;
cout<<"For first complex number"<<endl;
c1.getdata();
cout<<"For second complex number"<<endl;
c2.getdata();
result=c3.addcomplex(c1,c2);
cout<<"sum of two complex number=";
result.display();
return 0;
}
```

WAP to perform the addition of distance with data members feet and inches by passing object as argument and returning object .

```
#include<iostream>
using namespace std;
class Distance
{
private:
int feet;
int inches;
public:
void getdata();
Distance add(Distance,Distance);
void display();
};
void Distance::getdata()
{
cout<<"Enter feet:"<<endl;
cin>>feet;
cout<<"Enter inches:"<<endl;
cin>>inches;
}
void Distance::display()
{
cout<<feet<<"feet and"<<inches<<"inches"<<endl;
}
Distance Distance::add(Distance d1, Distance d2)
{
Distance temp;
temp.inches=d1.inches+d2.inches;
temp.feet=temp.inches/12;
temp.inches=temp.inches%12;
temp.feet=temp.feet+d1.feet+d2.feet;
return temp;
}
```

```
int main()
{
Distance d1,d2,d3,result;
cout<<"Enter information of first distance"<<endl;
d1.getdata();
cout<<"Enter information of second distance"<<endl;
d2.getdata();
result=d3.add(d1,d2);
cout<<"sum of distance=";
result.display();
return 0;
}
```

Function call by one object passing second object as function argument and return third object adding two objects.

WAP to perform the addition of two objects of complex numbers with data members real and imag and a function call by one object passing second object as function argument and return third object adding two objects.[Hint: c3=c1.addcomplex(c2)]

```
#include<iostream>
using namespace std;
class Complex
{
private:
int real;
int imag;
public:
void getdata();
Complex addcomplex(Complex);
void display();
};
void Complex::getdata()
{
cout<<"Enter the real part:"<<endl;
cin>>real;
cout<<"Enter the imaginary part :"<<endl;
cin>>imag;
}
void Complex::display()
{
cout<<real<<"+"<<imag<<"i"<<endl;
}
```

```
Complex Complex::addcomplex( Complex  c2)
{
Complex temp;
temp.real=real+c2.real;
temp.imag=imag+c2.imag;
return temp;
}

int main()
{
Complex c1,c2,c3;
cout<<"For first Complex number"<<endl;
c1.getdata();
cout<<"For second Complex number"<<endl;
c2.getdata();
c3=c1.addcomplex(c2);
cout<<"sum of two Complex number=";
c3.display();
return 0;
}
```

Assignment:

- WAP to perform the addition of time in hours, minutes and seconds format by passing object as an argument.
- WAP to perform the addition of time in hours, minutes and seconds format by returning object as an argument.
- WAP to create two time objects with data members hours, minutes and seconds a function call by one object passing second object as function argument and return third object adding two objects. *Hint:t3=t1.adddistance(t2);*
- WAP to create two distance objects with data members feet, inches and a function call by one object passing second object as function argument and return third object adding two objects. *Hint:d3=d1.adddistance(d2);*

Friend function

The private member of a class cannot be accessed from outside the class i.e. a nonmember function cannot access to the private data of a class. But may be in some situation one class wants to access private data of second class and second wants to access private data of first class, or may be an outside function wants to access the private data of a class. However, we can achieve this by using friend functions.

The non-member function that is "friendly" to a class, has full access rights to the private members of the class.

To make an outside function friendly to a class, we have to declare this function as a friend function. Declaration of friend function should be preceded by the keyword **friend**.

The friend function declares as follows:

```
class ABC
{
- - - - - -
- - - - - -
public :
- - - - - -
- - - - - -
friend void xyz() ;     //declaration
} ;
```

We give a keyword friend in front of any members function to make it friendly.

```
void xyz() //function definition
{
//function body
}
```

It should be noted that a function definition does not use friend keyword or scope resolution operator (: :). A function can be declared as friend in any number of classes.

Characteristics of a Friend Function:

- It is not in the scope of the class to which it has been declared as friend. That is why, it cannot be called using object of that class.
- It can be invoked like a normal function without the help of any object.
- It cannot access member names (member data) directly and has to use an object name and dot membership operator with each member name.(eg.A.x)
- It can be declared in the public or the private part of a class without affecting its meaning.
- Usually, it has the objects as arguments

Program to illustrate the use of friend function

```
#include <iostream>
using namespace std;
class sample
{
private:
int a,b;
public:
void setvalue(int x,int y )
{
a=x;
b=y;
}
void dispay()
{
        cout<<"value of a="<<a<<endl;
        cout<<"value of b="<<b<<endl;
}
friend void sum(sample s) ;
};
void  sum(sample s)
{
cout<<"sum of two numbers="<<(s.a+s.b);
}

int main( )
{
sample s1 ;
s1.setvalue(5,10) ;
s1.dispay();
sum(s1);
return 0 ;
 }
```

Create classes called class1 and class2 with each of having one private member .Add member function to set a value(say setvalue) one each class. Add one more function max() that is friendly to both classes. max() function should compare two private member of two classes and show maximum among them. Create one-one object of each class and then set a value on them. Display the maximum number among them.[PU:2015 fall,2016 fall

```
#include <iostream>
using namespace std;
class class2;
class class1
{
private:
int x;
public:
void setvalue (int num)
{
x=num ;
}
friend void max (class1, class2);
};

class class2
{
private:
int y ;
public:
void setvalue(int num)
{
 y=num ;
}
friend void max(class1, class2) ;
} ;

void max(class1 m, class2 n)
{
if (m.x>n.y)
cout<<"Maximum value="<<m.x ;
else
cout<<"Maximum value="<<n.y ;
}
int main( )
{
class1 p;
class2 q;
p.setvalue(10) ;
q.setvalue(20) ;
max(p, q);
return 0;
}
```

Write a program to perform addition of two private data of different classes using friend function.

```
#include <iostream>
using namespace std;
class ABC;        // Forward declaration
class XYZ
{
private:
int x;
public:
void setdata (int num)
{
x=num ;
}
friend void add (XYZ,ABC);
};
class ABC
{
private:
int y ;
public:
void setdata (int num)
{
 y=num ;
}
friend void add(XYZ,ABC) ;
} ;

void add (XYZ m, ABC n)         // Definition of friend
{
cout<<"Sum ="<<(m.x+n.y);
}
int main( )
{
XYZ p;
ABC q;
p.setdata(15) ;
q.setdata(20) ;
add(p,q);
return 0;
}
```

Output:

Sum=35

Alternative solution

```
#include <iostream>
using namespace std;
class ABC; // Forward declaration
class XYZ
{
public:
int x;
void getdata ()
{
cout<<"Enter Value of class XYZ"<<endl;
cin>>x;
}
friend void add (XYZ,ABC);
};

class ABC
{
int y ;
public:
void getdata ()
{
cout<<"Enter the value of class ABC"<<endl;
cin>>y;
}
friend void add(XYZ,ABC) ;
} ;
void add (XYZ m, ABC n)
{
cout<<"Sum ="<<(m.x+n.y);
}
int main( )
{
XYZ p;
ABC q;
p.getdata() ;
q.getdata() ;
add(p,q);
return 0;
}
```

Write a program to perform swapping of private data of two different classes using friend function.

```
#include <iostream>
using namespace std;
class ABC; // Forward declaration
class XYZ
{
private:
int x;
public:
void getdata ()
{
cout<<"Enter Value of class XYZ"<<endl;
cin>>x;
}
void display()
{
cout<<"value1="<<x<<endl;
}
friend void swap (XYZ &,ABC &);
};
class ABC
{
private:
int y ;
public:
void getdata ()
{
cout<<"Enter the value of class ABC"<<endl;
cin>>y;
}
void display()
{
cout<<"value2="<<y<<endl;
}
friend void swap(XYZ &,ABC &) ;
};
void swap (XYZ &m, ABC &n) // Definition of friend
{
int temp;
temp=m.x;
m.x=n.y;
n.y=temp;
}
```

```
int main( )
{
XYZ p;
ABC q;
p.getdata() ;
q.getdata() ;
cout<<"Value before swapping"<<endl;
p.display();
q.display();
swap(p,q);
cout<<"Value after swapping"<<endl;
p.display();
q.display();
return 0;
}
```

Friend class

A friend class can access private and protected members of other classes in which it is declared as a friend.
For example:

```
class B;
class A
{
…….
friend class B;
};
```

Here, class B can use all the data member of class A.

Example:

```
#include<iostream>
using namespace std;
class B;            //forward declaration
class A
{
int x,y;
public:
void getdata()
{
cout<<"Enter the value of x and y:"<<endl;
cin>>x>>y;
}
friend class B;
};
```

```
class B
{
int z;
public:
void getdata()
{
cout<<"Enter the value of z:";
cin>>z;
}
void sum(A t)
{
cout<<"Sum of x,y and z ="<<(t.x+t.y+z)<<endl;
}
};

int main()
{
A  p;
B q;
p.getdata();
q.getdata();
q.sum(p);
return 0;
}
```

1)**WAP to find the area of rectangle using the concept of constructor**

```
#include<iostream>
using namespace std;
class Rectangle
{
  private:
  float length,breadth,area;
  public:
    Rectangle(float l,float b)
    {
       length = l;
       breadth = b;
    }
     void calc( )
     {
           area= length * breadth;
     }
     void display( )
     {
       cout << "Area = " << area << endl;
     }
};
```

```
int main()
{
   Rectangle r(2,4);
   r.calc();
   r.display();
   return 0;
 }
```

2)WAP to initialize name, roll and marks of student using constructor and display then obtained information.

```
#include<iostream>
#include<string.h>
using namespace std;
class Student
{
private:
char name[20];
int roll;
float marks;
public:
Student(char n[],int r,float m)
{
strcpy(name,n);
roll=r;
marks=m;
}
void display()
{
cout<<"Name:"<<name<<endl;
cout<<"Roll:"<<roll<<endl;
cout<<"Marks:"<<marks<<endl;
}
};
int main()
{
char name[20];
int roll;
float marks;
cout<<"Enter name"<<endl;
cin>>name;
cout<<"Enter marks"<<endl;
cin>>marks;
cout<<"Enter Rollno"<<endl;
cin>>roll;
Student st(name,marks,roll);
st.display();
return 0;
}
```

3)Create a class student with data members (name, roll, marks in English, Maths and OOPs(in 100), total, average) a constructor that initializes the data members to the values passed to it as parameters, A function called calculate ( ) that calculates the total of the marks obtained and average. And function print() to display name, roll no, total and average.

```
#include<iostream>
#include<string.h>
using namespace std;
class student
{
   private:
   int roll;
   float me,mm,mo,total,avg;
   char name[30];
   public:
   student(char n[],int r,float e,float m,int o)
   {
     strcpy(name,n);
     roll=r;
     me=e;
     mm=m;
     mo=o;
   }
   void  calculate()
   {
       total= me+mm+mo;
       avg=total/3;
   }
  void print()
  {
   cout<<"Name="<<name<<endl;;
   cout<<"Roll No="<<roll<<endl;;
   cout<<"Total="<<total<<endl;
   cout<<"Average="<<avg<<endl;
  }
};

int main()
{
   char name[30];
   int roll;
   float meng,mmath,moops;
   cout<<"Enter the name"<<endl;
   cin>>name;
   cout<<"Enter the roll"<<endl;
   cin>>roll;
```

```
    cout<<"Enter the marks in english"<<endl;
    cin>>meng;
     cout<<"Enter the marks in maths"<<endl;
    cin>>mmath;
    cout<<"Enter marks in oops"<<endl;
    cin>>moops;
    student st(name,roll,meng,mmath,moops);
    st.calculate();
    st.print();
    return 0;
}
```

4)Write a Program to add two complex number using the concept of Constructor/ Constructor overloading.[PU:2015 spring]

```
#include<iostream>
using namespace std;
class Complex
{
private:
int real,imag;
public:
Complex()
{
real=2;
imag=4;
}
Complex(int r,int i)
{
real=r;
imag=i;
}
void addcomplex(Complex c1,Complex c2)
{
        real=c1.real+c2.real;
        imag=c1.imag+c2.imag;
}
void display()
{
 cout<<real<<"+"<<imag<<"i"<<endl;
}
};
```

```
int main()
{
Complex c1;
Complex c2(10,20);
Complex c3;
cout<<"First complex number=";
c1.display();
cout<<"Second complex number=";
c2.display();
c3.addcomplex(c1,c2);
cout<<"sum of complex number=";
c3.display();
return 0;
}
```

5)Create a class person with data members Name, age, address and citizenship number. Write a constructor to initialize the value of the person. Assign citizenship number if the age of the person is greater than 16 otherwise assign value zero to citizenship number. Also create a function to display the values.[PU:2013 fall]

```
#include<iostream>
#include<string.h>
using namespace std;
class Person
{
  private:
   char name[20];
   char address[20];
   int age;
   long int citzno;
public:
   Person(char n[],int a,char addr[],long int  c)
   {
     strcpy(name,n);
     age=a;
     strcpy(address,addr);
     citzno=c;
   }
void display()
{
cout<<"Name:"<<name<<endl;
cout<<"Address:"<<address<<endl;
cout<<"Age:"<<age<<endl;
cout<<"Citizenship no:"<<citzno<<endl;
}
};
```

```
int main()
{
   char name[20];
   char address[20];
   int age;
   long int citzno;
   cout<<"Enter the name"<<endl;
   cin>>name;
   cout<<"Enter the address"<<endl;
   cin>>address;
   cout<<"Enter the age"<<endl;
  cin>>age;
 if (age>16)
  {
    cout<<"Enter the citizenship number"<<endl;
    cin>>citzno;
    Person p(name,age,address,citzno);
    p.display();
  }
  else
  {
  Person p(name,age,address,0);
  p.display();
  }
  return 0;
}
```

6)Create a class Mountain with data members name, height, location, a constructor that initializes the members to the values passed it to its parameters, a function called CmpHeight() to compare two objects and DispInf() to display the information of mountain. In main create two objects of the class mountain and print the information of the mountain which is greatest height.[PU:2016 spring]

```
#include<iostream>
#include<string.h>
using namespace std;
class Mountain
{
private:
char name[20],location[20];
float height;
public:
Mountain(char n[],float h,char l[])
{
 strcpy(name,n);
 height=h;
 strcpy(location,l);
}
```

```
void DispInf()
{
        cout<<"Name:"<<name<<endl;
        cout<<"Height:"<<height<<endl;
        cout<<"Location:"<<location<<endl;
}
friend void CmpHeight(Mountain,Mountain);
};

void CmpHeight(Mountain m1,Mountain m2)
{
        if(m1.height>m2.height)
        {
          m1.DispInf();
        }
        else
        {
           m2.DispInf();
        }
}

int main()
{
char name1[20],name2[20];
float height1,height2;
char location1[20],location2[20];
cout<<"Enter Name,Height and location of first mountain"<<endl;
cin>>name1>>height1>>location1;
cout<<"Enter Name,Height and location of second mountain"<<endl;
cin>>name2>>height2>>location2;
Mountain m1(name1,height1,location1);
Mountain m2(name2,height2,location2);
cout<<"Information of mountain with greatest height"<<endl;
CmpHeight(m1,m2);
return 0;
}
```

7)Create a class time constructor having hour, minute and second as an arguments is use to take two time data from user. The add function that takes two class objects an arguments add them respectively then display the aggregate result? (Apply 60 second =1 minutes and 60 minutes =1 hour) [PU: 2017 spring]

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

class time
{
	private:
	int hour,minute,second;
	public:
		time()
		{

		}
		time(int h,int m,int s)
		{
		hour=h;
		minute=m;
		second=s;
		}
		void display()
		{
			cout<<hour<<":"<<minute<<":"<<second<<endl;
		}
		void sum(time,time);
};

void time::sum(time t1,time t2)
{
	second=t1.second+t2.second;
	minute=second/60;
	second=second%60;
	minute=minute+t1.minute+t2.minute;
	hour=minute/60;
	minute=minute%60;
	hour=hour+t1.hour+t2.hour;
}
```

```
int main()
{
int hr1,min1,sec1,hr2,min2,sec2;
cout<<"Enter first hour,minutes and seconds"<<endl;
cin>>hr1>>min1>>sec1;
cout<<"Enter second hour,minutes and seconds"<<endl;
cin>>hr2>>min2>>sec2;
time t1(hr1,min1,sec1);
time t2(hr2,min2,sec2);
time t3;
cout<<"The 1st time is"<<endl;
t1.display();
cout<<"The 2nd time is"<<endl;
t2.display();
t3.sum(t1,t2);
cout<<"The resultant time is"<<endl;
t3.display();
return 0;
}
```

8)Create a class called employee with data member Code, Name, address, salary. Create a constructor to initialize the member of the class. Also create the another constructor so that we can create an object from another object. Define member function display() to display the information of the class.[PU:2018 fall]

```
#include<iostream>
#include<string.h>
using namespace std;
class employee
{
        private:
        int code;
        char name[20];
        char address[20];
        float salary;

        public:
          employee(int c, char n[],char addr[],float s)
           {
                code=c;
                strcpy(name,n);
                strcpy(address,addr);
                salary=s;
          }
```

```
  employee(employee &e)
        {
                code=e.code;
                strcpy(name,e.name);
                strcpy(address,e.address);
                salary=e.salary;
        }
void display()
 {
        cout<<"Code="<<code<<endl;
        cout<<"Name="<<name<<endl;
        cout<<"Address="<<address<<endl;
        cout<<"Salary="<<salary<<endl;
  }
};
 int main()
{
         int code;
        char name[20];
        char address[20];
        float salary;
                 cout<<"Enter Code:"<<endl;
                cin>>code;
                cout<<"Enter Name:"<<endl;
                cin>>name;
                cout<<"Enter Address:"<<endl;
                cin>>address;
                cout<<"Enter Salary:"<<endl;
                cin>>salary;
                employee e1(code,name,address,salary);
                employee e2(e1);
                cout<<"Values in object e1"<<endl;
                e1.display();
                cout<<"Values in object e2"<<endl;
                e2.display();
                return 0;
         }
```

9) Write a program using of dynamic memory allocation which should include calculation of marks of 3 subjects of n students and displaying the result as pass or fail & name, roll. Pass mark is 45 out of 100 in each subject

```
#include<iostream>
using namespace std;
class student
{
private:
char name[20];
int roll;
float m1,m2,m3;
public:
void getdata()
{
cout<<"enter the name"<<endl;
cin>>name;
cout<<"enter the roll"<<endl;
cin>>roll;
cout<<"Enter marks of 3 subjects"<<endl;
cin>>m1>>m2>>m3;
}
void display()
{
cout<<"Name:"<<name<<endl;
cout<<"Roll:"<<roll<<endl;
if(m1>=45&&m2>=45&&m3>=45)
cout<<"Result:Pass"<<endl;
else
cout<<"Result:Fail"<<endl;
}
};

int main()
{
        int n,i;
        student *ptr;
        cout<<"Enter the number of students:";
        cin>>n;
        ptr= new student[n];
        for(i=0;i<n;i++)
        {
        cout<<"Enter the information of student"<<i+1<<endl;
        ptr[i].getdata();
        }
```

```
        for(i=0;i<n;i++)
        {
        cout<<"Information of student"<<i+1<<endl;
        ptr[i].display();
        }
        delete[] ptr;
        return 0;
}
```

10)Write a program using of dynamic memory allocation which should include calculation of marks of 5 subjects of students and displaying the result of pass or fail.(Pass marks is 45 out of 100 in each subject) [PU: 2010 fall 3(b)]

```
#include<iostream>
using namespace std;
class Student
{
private:
float sub1,sub2,sub3,sub4,sub5;
public:
void getdata()
{
cout<<"Enter marks of 5 subjects"<<endl;
cin>>sub1>>sub2>>sub3>>sub4>>sub5;
}
void display()
{
if(sub1>=45&&sub2>=45&&sub3>=45&&sub4>=45&&sub5>=45)
{
cout<<"Student is passed"<<endl;
}
else
{
cout<<"Student is failed"<<endl;
}
}
};
```

```
int main()
{
	int n,i;
	Student *ptr;
	cout<<"Enter the number of students:";
	cin>>n;
	ptr= new Student[n];
	for(i=0;i<n;i++)
	{
	cout<<"Enter marks of student"<<i+1<<endl;
	ptr[i].getdata();
	ptr[i].display();
	}
	delete[] ptr;
	return 0;
}
```

11)WAP to find the sum of n numbers Entered by user using Dynamic Memory Allocation in C++.

```
	#include <iostream>
	using namespace std;
	int main ()
	{
	int i,size,sum=0;
	int *ptr;
	cout << "How many numbers would you like to Enter?"<<endl;
	cin >>size;
	ptr= new int[size];
	for (i=0; i<size; i++)
	{
	cout << "Enter number"<<i+1<<endl;
	cin >> ptr[i];
	}
	cout << "You have entered:"<<endl;
	for (i=0; i<size; i++)
	{
	cout << ptr[i] << endl;
	}
	for (i=0; i<size; i++)
	{
	sum=sum+ptr[i];
	}
	cout<<"sum of numbers="<<sum<<endl;
	delete[] ptr;
	return 0;
	}
```

Previous Board Exam Questions

1) Declare a C++ structure (Program) to contain the following piece of information about cars on a used car lot: [**PU:2013 spring]**

i. Manufacturer of the car
ii. Model name of the car
iii. The asking price for the car
iv. The number of miles on odometer

2) Differentiate between class and structure. Explain them with example.**[PU:2010 spring]**
3) What sorts of shortcomings of structure are addressed by classes? Explain giving appropriate example.**[PU:2014 fall]**
4) Differentiate between structure and class. Why Class is preferred over structure? Support your answer with suitable examples.**[PU:2016 fall]**
5) Differentiate between structure and class in C++?What are the various access specifiers in C++?[**PU:2019 spring]**
6) Explain the various access specifiers used in C++ with an example. **[PU:2010 fall][PU:2016 spring]**
7) What is information hiding? What are the access mode available in C++ to implement different levels of visibility? Explain through example.**[PU:2014 spring][PU:2016 fall]**
8) What is data hiding? How do you achieve data hiding in C++? Explain with suitable program.**[PU:2019 fall]**
9) What is encapsulation? How can encapsulation enforced in C++?Explain with suitable example code. **[PU:2017 spring]**
10) What are the common typed of function available in C++? Define the 3 common types of functions in C++ with a program. **[PU:2015 spring]**
11) What is a function? Discuss the use of friend function taking into consideration the concept of data hiding in object oriented programming.**[PU:2009 spring]**
12) Does friend function violate the data hiding? Explain briefly**.[PU:2017 fall]**
13) "Friend function breaches the encapsulation." Justify. Also mention the use of friend function. **[PU:2015 spring]**
14) Where do you use friend function? **[PU:2014 spring]**
15) What are the merits and demerits of friend function? **[PU:2009 fall]**
16) Private data and function of a class cannot be accessed from outside function. Explain how is it possible to access them with reference of an example. **[PU:2018 fall]**
17) What are the advantages of using friend function? List different types of classes and explain any two. **[PU:2010 spring]**
18) What are the advantages and disadvantages of using friend function? Explain with example program.**[PU:2018 fall]**
19) Illustrate the role of friend function in object oriented programming with its pros and cons. Also write suitable program.**[PU:2019 spring]**
20) Friend function is the non-member of the class. Justify this statement with suitable example.**[PU:2020 fall]**

21) What is inline function? Explain its importance with the help of example program. **[PU:2015 fall]**
22) What is the role of static data in C++ classes? Give example.**[PU:2006 spring]**
23) What do you mean by static member of a class ?Explain the characteristic of static data member.**[PU:2013 fall][PU:2017 fall]**
24) When and how do we make use of static data members of a class? Differentiate between virtual functions, friend functions and static member functions**.[PU:2013 spring]**
25) What are the static data member and static member functions? Show their significance giving examples.**[PU:2014 fall]**
26) What is message passing? Describe with example**.[PU:2014 fall]**
27) What is the difference between message passing and function call? Explain the basic message formalization. **[PU:2006 spring]**
28) Differentiate message passing and procedure call with suitable example. What are the possible memory errors in programming.**[PU:2014 spring]**
29) Explain the following term with suitable examples. Agents ,Responsibility, Messages and methods.**[PU:2009 fall]**
30) Explain message passing formalism with syntax in C++.What is stack Vs heap memory allocation? **[PU:2016 spring]**
31) What does constructor mean? Explain different types of constructor with suitable examples. **[PU:2005 fall]**
32) What is constructor? Explain copy constructor with suitable examples.[PU:2009 fall]
33) What is constructor? Write an example of Copy constructor and explain each line of code. **[PU:2017 spring**]
34) "A constructor is a special member function that automatically initializes the objects of its class", support this statement with a program of all types of constructors. Also enlist the characters of constructors. **[PU:2015 spring]**

    What is constructor? Is it mandatory to use constructor in class. Explain**[PU:2006 spring]**
35) What are default and parametrized constructors? Write a program to illustrate parametrized constructor.**[PU:2020 fall]**
36) What is copy constructor in C++? Is it possible to pass object as argument in Copy constructor? Explain with suitable program**.[PU:2020 fall]**
37) Differentiate between constructor and destructor. Can there be more than one destructor in a program for destroying a same object. Illustrate you answer.**[PU:2010 fall]**
38) What are constructors and destructors? Explain their types and uses with good illustrative example? What difference would be experienced if the features of constructors and destructors were not available in C++. **[PU:2009 spring]**
39) What is de-constructor? can you have two destructors in a class? Give example to support your reason.**[PU:2014 spring]**
40) Discuss the various situations when a Copy constructor is automatically invoked. How a default constructor can be equivalent to a constructor having default arguments. **[PU:2013 spring]**

41) Can we have more than one destructor in a class? Write a Program to add two complex numbers using the concept of constructor. **[PU:2015 spring]**
42) What do you mean by dynamic constructor? Explain its application by a program to compute complex numbers. **[PU:2016 fall]**
43) Differentiate methods of argument passing in constructor and destructor.**[PU:2017 fall]**
44) Why destructor function is required in class? Can a destructor accept arguments? **[PU:2017 spring]**
45) What is constructor? Can constructor can be overloaded? If yes how that is possible with reference of an example? **[PU:2018-fall][PU:2017 fall]**
46) What are the advantages of dynamic memory allocation? Explain with suitable example. **[PU:2016 spring]**
47) What is dynamic memory allocation? How memory is allocated and de-allocated in C++?Explain with examples**.[PU:2019 spring]**
48) What is memory recovery? How does stack differ from heap memory allocation. **[PU:2005 fall]**
49) Write a short notes on:
    - Copy constructor**[PU:2014 spring]**
    - Stack vs Heap based Allocation
    - Memory Recovery**[PU:2015 spring]**
    - Message passing in C++**[PU:2019 sprintg]**
    - Friend function **[PU:2006 spring][PU:2013 spring] [PU:2017 spring][PU:2005 fall]**
    - Inline function**[PU:2009 fall][PU:2010 spring][PU:2013 spring][PU:2019 spring]**
    - Reference variable
    - Default argument

Programs

1. Declare a C++ structure (Program) to contain the following piece of information about cars on used car lot. [PU:2013 spring]

i. Manufacturer of the car
ii. Model name of the car
iii. The asking price of car
iv. The number of miles on odometer

2. Create a class called Employee with three data members (empno , name, address), a function called readdata() to take in the details of the employee from the user, and a function called displaydata() to display the details of the employee. In main, create two objects of the class Employee and for each object call the readdata() and the displaydata() functions. **[PU:2005 fall]**

3. Create a class called student with three data members (stdnt_name[20],faculty[20],roll_no), a function called readdata() to take the details of the students from the user and a function called displaydata() to display the details the of the students. In main, create two objects of the class student and for each object call both of the functions. **[PU:2010 fall]**
4. Modify the **Que.no 2** for 20 students using array of object.
5. WAP to perform the addition of time in hours, minutes and seconds format.
6. WAP to perform the addition of time using the concept of returning object as argument.
7. WAP to create two time objects with data members hours, minutes and seconds a function call by one object passing second object as function argument and return third object adding two objects. *Hint:t3=t1.adddistance(t2);*
8. WAP to create two distance objects with data members feet, inches and a function call by one object passing second object as function argument and return third object adding two objects. *Hint:d3=d1.adddistance(d2);*
9. Create a class called Rational having data members nume and deno and using friend function find which one is greater.
10. WAP to add the private data of three different classes using friend function.
11. Write a program to find the largest of four integers .your program should have three classes and each classes have one integer number.[**PU:2014 spring]**
12. WAP to swap the contents of two variables of 2 different classes using friend function.
13. WAP to add two complex numbers of two different classes using friend function.
14. WAP to add complex numbers of two different classes using friend class.
15. Using class write a program that receives inputs principle amount, time and rate. Keeping rate 8% as the default argument, calculate simple interest for three customers.**[PU:2019 fall]**
16. Create a new class named City that will have two member varibles CityName (char[20]), and DistFromKtm (float).Add member functions to set and retrieve the CityName and DistanceFromKtm separately. Add new member function AddDistance that takes two arguments of class City and returns the sum of DistFromKtm of two arguments. In the main function, Initialize three city objects .Set the first and second City to be pokhara and Dhangadi. Display the sum ofDistFromKtm of Pokhara and Dhangadi calling AddDistance function of third City object. **[PU: 2010 Spring]**
17. Create a class called Volume that uses three Variables (length, width, height) of type distance (feet and inches) to model the volume of a room. Read the three dimensions of the room and calculate the volume it represent, and print out the result .The volume should be in (feet3) form ie. you will have to convert each dimension into the feet and fraction of foot. For instance , the length 12 feet 6 inches will be 12.5 ft)
    [PU: 2009 spring]
18. WAP to read two complex numbers and a function that calls by passing references of two objects rather than values of objects and add into third object and returns that object.
19. WAP in C++ to calculate simple interest from given principal, time and rate. Set the rate to 15 % as default argument when rate is not provided.

20. Create a class student with six data members (roll no, name, marks in English,math, science and total).Write a program init() to initializes necessary data members calctotal() and display().Create a program for one student (i.e one object only necessary)