Chapter-3
Message, Instance & Initialization

Message Passing Formalization

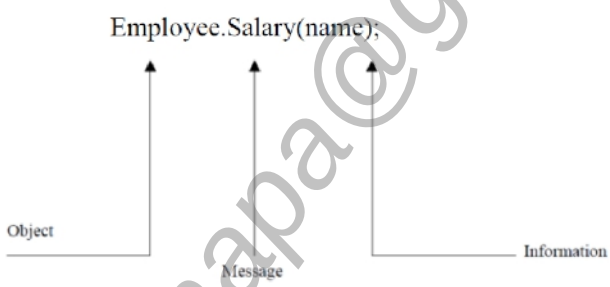
- Procedural programming languages have function driven communication. That is a function is invoked for a piece of data.
- Object oriented language have message driven communication. A message for an object is a request for execution of a procedure and therefore will invoke a function or procedure in receiving object that generate the desired result for the given argument.
- An Object-Oriented program consists of set of objects that communicate with each other. Object communicates with each other by sending and receiving message.
- Communications among the objects are analogous to exchanging messages among people. To make possible message passing, the following things have to be followed to be done :
    - We have to create classes that define objects and its behaviour.
    - Then Creating the objects from class definitions
    - Calling and connecting the communication among objects

Message Passing Syntax in c++

- Message passing involves specifying the name of the object name of the function (message) and the information (arguments to function) to be sent.
  Syntax

  Object_Name.Function_Name(Parameters)

  Example:



- In above example Employee is regarded as an object sending the message Salary to be paid by the Employee with the given name.
- Communication between the objects takes place as long as their existence. Objects are created and destroyed automatically whenever needed.

Message Passing Implementation in c++

```
#include<iostream>
class Employee
{
   public :
   void Salary(char *name)
   {
      std::cout<<"Salary Paid to "<<name;
   }
};
int main()
{
   char *name="Ram Thapa";
   Employee f;
   f.Salary(name);
};
```

In the above code passing information`(name) to Method (Salary) is said to be Message Passing.

Message Passing vs Procedure Call

- In both, there is a set of well-defined steps that will be initiated following the request. But, there are two important distinctions.
- The first is that in a message there is a designated receiver for that message; the receiver is some object to which the message is sent. In a procedure call, there is no designated receiver.
- The second is that the interpretation of the message (that is, the method used to respond to the message) is dependent on the receiver and can vary with different receivers.

Object Initialization using Constructor

- Instead of using function like setData() which has to be called manually to initialize object, an OOP has provision of automatic object initialization.
- A class in C++ contains two special member function: Constructor and Destructor(will be discussed later).
- A constructor is a special member function having the same name as that of the class which is used to automatically initialize the objects of the class type with legal initial values.
- The constructor is invoked automatically whenever an object of its associated class is created.
- It is called constructor because it constructs the values of data members of the class.

Properties of constructor

- They are also used to allocate memory for a class object.
- They execute automatically when an object of a class is created.
- Constructor's name is same as that of class name.
- They should be declared in the "public" section.
- They do not have return types, not even void and therefore, and they cannot return any values.
- Like C++ functions, they can have default arguments.
- Constructor is NOT called when a pointer of a class is created.
- Constructer can't be virtual and inherited.

A constructor is declared and defined as below

```
class Demo
{
        private:
        ----------
        ----------
        public:
        Demo(); //constructor
};
Demo ::Demo() //note no return type required
{
        //Body of constructor if defined outside the class
}
```

Types of Constructor

1. **Default Constructor**

- A constructor that does not take any parameter is called default constructor.
- This constructor is always called by compiler if no user-defined constructor is provided.
- The following Class definition shows a default constructor.

```
class Demo
{
        public:
        Demo()  //Default Constructor
```

```
        {

        }
};
```

- This constructor is also called **as implicit constructor** because if we do not provide any constructor with a class, the compiler provided one would be the default constructor.
- And it does not do anything other than allocating memory for the class object.

```
class A
{
        //no constructor
};
```

- Also if we provide constructor with all default arguments, then that can also be considered as the default constructor or **constructor with default argument.**

```
class Demo
{
        public:
        Demo (int a=7)
        {

        }
};
```

Example:
Create a class Demo with two data member (id, name), a default constructor to initialize these fields and a member function display() to show student details. Write a main program to test your class.

```
#include<iostream.>
#include<string.h>
using namespace std;
class Demo
{
        int id;
        char name[10];
        public:
                Demo() // Default constructor
                {
                        id=5;
                        strcpy(name,"Bhesh");
                }
                void display()
                {
                        cout<<"ID= "<<id<<endl;
                        cout<<"Name= "<<name<<endl;
                }
};
int main()
{
        Demo d;
        d.display();
}
```

Output:

```
ID= 5
Name= Bhesh
```

2. Parameterized constructor

- The constructors that can take arguments or parameters are called parameterized constructor.
- In this, we pass the initial value as arguments to the constructor function when the object is declared.
- The argument to be passed must match the signature i.e. no or parameters and types of constructor defined.
- This can be done in two ways.

i. By calling the constructor explicitly.
For calling the constructor explicitly, we use the following declaration.
Class_name object_name =Class_name (Argument1, Argument2,……Argumentn);
Example:
Demo d=Demo(6);

ii. By calling the constructor implicitly.
For calling the constructo implicitly, we use the following declaration.
Class_name object_name(Argument1, Arguemnt2,……Argumentn);
Example:
Demo d(6);

The following Class definition shows a parameterized constructor.

```
class Demo
{
        public:
        Demo(int x, int y)  //Parameterized Constructor
        {

        }
};
```

WAP to find the area of rectangle using the concept of parameterized constructor

```
#include<iostream.>
using namespace std;
class Demo
{
        int l , b;
        public:
                Demo(int length, int breadth)
                {
                        l=length;
                        b=breadth;
                }
                void area()
                {
                        int a;
                        a=l*b;
                        cout<<"Area of rectangle= "<<a<<endl;
                }
};
int main()
{
        Demo d(5,6);
        d.area();
}
```

Output:

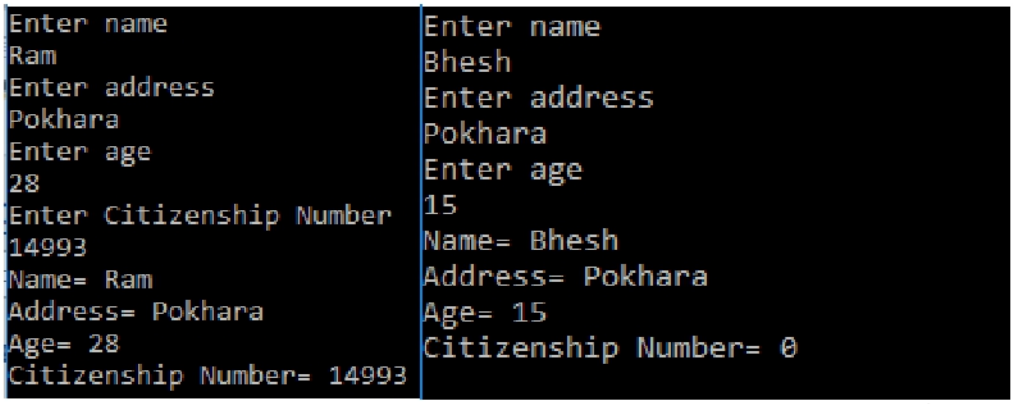
```
Area of rectangle= 30
```

Create a class Person with data member Name, age, address and citizenship_number. Write a constructor to initialize the value of a person. Assign citizenship number if the age of the person is greater than 16 otherwise assign values zero to citizenship number. Also, create a function to display the values. Write a main program to test your class.

```
#include<iostream.>
#include<string.h>
using namespace std;
class Demo
{
        char name[50],address[50];
        int citizenship_number,age;
        public:
                Demo(char n[],char ad[],int a, int cn)
                {
                        strcpy(name,n);
                        strcpy(address,ad);
                        citizenship_number=cn;
                        age=a;
                }
                void display()
                {
                        cout<<"Name= "<<name<<endl;
                        cout<<"Address= "<<address<<endl;
                        cout<<"Age= "<<age<<endl;
                        cout<<"Citizenship Number= "<<citizenship_number<<endl;
                }

};
int main()
{
        char nm[50],ad[50];
        int cn,a;
        cout<<"Enter name"<<endl;
        cin>>nm;
        cout<<"Enter address"<<endl;
        cin>>ad;
        cout<<"Enter age"<<endl;
        cin>>a;
        cn=0;
        if(a>16)
        {
                cout<<"Enter Citizenship Number"<<endl;
                cin>>cn;
        }
        Demo d(nm,ad,a,cn);
        d.display();
}
```

Output:



3. Copy Constructor

- The copy constructor is a constructor which creates an object by initializing it with an object of the same class, which has been created previously.
- Parameterized constructors having object of the same class as parameter or argument is called copy constructor.
- The copy constructor is used to:
  i. Initialize one object from another of the same type.
  ii. Copy an object to pass it as an argument to a function.
  iii. Copy an object to return it from a function.
- **If a copy constructor is not defined in a class, the compiler itself defines one.** A copy constructor is called when an object is created by copying an existing object.
- Example:

  Class_name obj2(obj1);

  The above statement would define the object obj2 and initialize it to the values of obj1.

  The above statement can also be written as,

  Class_name obj2=obj1;
- The process of initializing through a copy constructor is known as copy initialization.
- A copy constructor takes a reference to an object of the same class as itself (as an argument).
- We cannot pass the argument by value to a copy constructor.
- When no copy constructor is defined, the compiler supplies its own copy constructor. This is called as default copy constructor.
- Remember the statement

  obj2 = obj1;

  will not invoke the copy constructor. However, if obj1 and obj2 are objects, this statement is legal and simply assigns the values of obj1 to obj2, member by member. This is the task of the overloaded assignment operator (=).

A program to demonstrate the concept of copy constructor

```
#include<iostream.>
using namespace std;
class Demo
{
        int a;
        public:
                Demo()//default constructor
                {}
                Demo (int b)
                {
                        a=b;
                }
```

```
        void display()
        {
                cout<<a<<endl;
        }

        //the below is the definition of copy constructor and this is optional
        Demo (Demo &x)
        {
                a=x.a;
        }
};
int main()
{
        Demo d(5);
        Demo x(d); // copy constructor will called and if no copy constructor available the compiler will
                create one
        Demo y=d; // copy constructor will called and if no copy constructor available the compiler will
                create one
        Demo z;
        z=d; // It will not call copy constructor but assigns value member by member
        d.display();
        x.display();
        y.display();
        z.display();
}
```

Output:

```
5
5
5
5
```

Note:// we also can write statiement like Demo x(&d) and Define copy constructor Demo as

```
Demo(Demo *x)
{
        a= x->a;
}
```

Note:

A constructor can be invoked in two ways: implicitly and explicitly. The following program demonstrates the concept.

```
#include<iostream.>
using namespace std;
class Demo
{
        int l , b;
        public:
                Demo(int length, int breadth)
                {
                        l=length;
                        b=breadth;
                }
                void area()
                {
                        int a;
                        a=l*b;
```

```
                cout<<"Area of rectangle= "<<a<<endl;
        }
};
int main()
{
        Demo d(5,6); // this type of constructor invoking technique  is called implicit
        d.area();

        Demo p=Demo(7,8); // this type of constructor invoking technique is called explicit.
        p.area();
}
```

The Second One actually creates a temporary using the two-argument constructor, which is then used to create p using the copy constructor, whereas the first one only needs the two-argument constructor

Default vs parameterized constructor

Default Constructor	Parameterized Constructor
A constructor that has no parameter is called default constructor.	A constructor that has paramteter(s) is called parameterized constructor.
Default constructor is used to initialize object with same default value like 0, null.	Parameterized constructor is used to initialize each object with different values.
When data is not passed at the time of creating an object, default constructor is called but not parameterized.	When data is passed at the time of creating an object, default constructor as well as parameterized constructor is called.

Constructor vs Method/Function

Constructor	Methods
Constructor is used to initialize the state of an object.	Method is used to expose behaviour of an object.
Constructor must not have return type.	Method must have return type.
Constructor is invoked implicitly.	Method is invoked explicitly.
The java compiler provides a default constructor if you don't have any constructor.	Method is not provided by compiler in any case.
Constructor name must be same as the class name.	Method name may or may not be same as class name.

Constructor Overloading

- When more than one constructor functions are defined in the same class then we say the constructor is overloaded.
- All the constructors have the same name as the corresponding class, and they differ in terms of number of arguments, data types of argument or both.
- This makes the creation of object flexible.

Example

```
class Demo
{
        public:
                Demo()  //Default Constructor
                {

                }
                Demo(parameter1) // Parameterized constructor with one parameter
                {

                }
```

```
        Demo(parameter1,parameter2)//Parameterized constructor with two parameters
        {

        }
};
```

- In the above example, the constructor is overloaded because we have three different definitions for same constructor name with different signature.

WAP to find the area of circle and rectangle using the concept of constructor overloading

```
#include<iostream.>
using namespace std;
class Demo
{
    public:
        Demo(float radius)
        {
            cout<<"Area of circle= "<<(3.1416*radius*radius)<<endl;
        }
        Demo(float length, float breadth)
        {
            cout<<"Area of rectangle= "<<(length * breadth)<<endl;
        }
};
int main()
{
    Demo d(4);// invokes constructor with one parameter
    Demo d1(5,6); //invokes constructor with two parameter
}
```

Output:

```
Area of circle= 50.2656
Area of rectangle= 30
```

Constructor with Default Argument

- We can define constructor with default argument unlike function with default argument.
- The following program demonstrates the concept of default constructor.

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

class Demo
{
    int l,b;
    public:
        Demo(int length, int breadth=20)
        {
            l=length;
            b=breadth;
        }
        void area()
        {
            cout<<"Area of rectangle= "<<(l*b)<<endl;
        }
};
int main()
{
    Demo d(4);
```

```
        Demo d1(4,5);
        d.area();
        d1.area();
    }
```
Output:
```
Area of rectangle= 80
Area of rectangle= 20
```

Again consider the following program to find the square of number

```
#include<iostream.>
using namespace std;
class Demo
{
        int n;
        public:
                Demo(int a=5) //Default constructor with default argument
                {
                        n=a;
                }
                void square()
                {
                        cout<<(n*n)<<endl;
                }
};
int main()
{
        Demo a;
        a.square();
        Demo a1(5);
        a1.square();
}
```
Output:
```
25
25
```

- The default argument constructor can be called with either one argument or no argument as shown in above example. But if we use both default constructor as well as above default argument constructor then the declaration of statement

  Demo a;

  Will be ambiguous for whether to call Demo() or Demo(int=0).

Private Constructor

- A private constructor is a special instance constructor.
- It is generally used in classes that contain static members only.
- If a class has one or more private constructors and no public constructors, other classes cannot create instances of this class.
- The following program demonstrates the concept of private constructor

```
#include<iostream>
using namespace std;
class Demo
{
        private:
                Demo()
                {
                }
```

```
        public:
                static void display()
                {
                        cout<<"Hello";
                }
};
int main()
{
        //Demo d; //this statement is illegal now because the constructor is private
        Demo::display(); //Directly access Static member via class name
}
```

Destructors

- Destructors are the special function that destroys the object that has been created by a constructor.
- In other words, they are used to release dynamically allocated memory and to perform other "cleanup" activities. Destructors, too, have special name, a class name preceded by a tilde sign (~).

  Example:

  ~Demo(){}

- Destructor gets invoked, **automatically,** when an object goes **out of scope** (i.e. exit from the program, or block or function).
- They are also defined in the public section.
- Destructor never takes any argument, nor does it return any value. So, they cannot be overloaded.

  Note: whenever 'new' is used to allocate memory in the constructors, we should use 'delete' to free the memory.

  For example:

```
Demo:: ~Demo()
{
        Delete obj;
}
```

- When multiple object goes out of scope at same time then the destructor of the object whose constructor is called last is called first i.e. destructor of the object whose constructor is called first is called at last

Characteristic of Destructor

- It can't take any argument so can't be overloaded.
- It has same name as the class name but preceded by a tilde symbol.
- It can't be declared statics.
- It can't return a value.
- It should have public or protected access specifiers.
- Only one destructor exists in a class.

Example:

The below program demonstrates the concept of constructor and destructor

```
#include<iostream>
using namespace std;
class Demo
{
        static int count;
```

```
        public:
        Demo()
        {
                count++;
                cout<<"Object created= "<<count<<endl;
        }
        ~Demo()
        {
                cout<<"Object Destroyed= "<<count<<endl;
                count--;
        }
};
int Demo::count;
int main()
{
        Demo a1,a2,a3;
        {
                Demo a4;
        }
}
```

Output:

```
Object created= 1
Object created= 2
Object created= 3
Object created= 4
Object Destroyed= 4
Object Destroyed= 3
Object Destroyed= 2
Object Destroyed= 1
```

Constructor vs Destructor

- Constructors **guarantee** that the member **variables are initialized when an object is declared**.
- Constructors automatically execute when a class object enters its scope.
- The **name of a constructor** is **the same as the name of the class**.
- A class can **have more than one** constructor.
- A constructor without parameters is called **the default constructor**.

- Destructor **automatically execute when a class object goes out of scope.**
- The name of a destructor is the tilde (~), followed by the class name (no spaces in between).
- A class **can have only one destructor.**
- The destructor has no parameters.

Dynamic Initialization of Object

- Dynamic initialization of object refers to initializing the objects at a run time i.e., the initial value of an object is provided during run time.
- It can be achieved by using constructors and by passing parameters to the constructors.
- This comes in really handy when there are multiple constructors of the same class with different inputs. So this allows various initialization format using constructor overloading.

WAP to demonstrate dynamic initialization of object

```
#include<iostream>
using namespace std;
class Shape
{
        private:
        float len,bre,rad;
        public:
                Shape(float len, float bre)
                {
                        this->len=len;
                        this->bre=bre;
                }
                Shape(float rad)
                {
                        this->rad=rad;
                }
                void areaofrectangle()
                {
                        float area=len*bre;
                        cout<<"Area of rectangle="<<area<<endl;
                }
                void areaofcircle()
                {
                        float area=3.14 * rad * rad;
                        cout<<"Area of circle="<<area<<endl;
                }
};
int main()
{
        Shape circle(1.5);//dynamic initialization
        circle.areaofcircle();
        Shape rect(3,4);//Dynamic initialization with different format using overloaded constructor
        rect.areaofrectangle();
}
```

Memory Allocation Method

- Memory allocation is the process of reserving computer memory for the execution of programs.
- Memory in the C++ program is divided into two parts:
  1. Stack: All variables declared inside any function take up the stack's memory.
  2. Heap: It is the unused memory of the program and can be used to allocate the memory at runtime dynamically.
- There are two types of memory allocation in C ++.

1. Static memory allocation/Compile Time Allocation

- In static memory allocation, size and location where variable will be stored is fixed during compile time.
- In this, the memory allocation is at compile time. Therefore, it is also called compile time memory allocation.
- Here the compiler allocates memory for named variables so the exact size and storage must be known at compile time. So, the memory size is fixed in it and doesn't grow or shrink at run time and hence it is also called static memory allocation.
- The memory allocated by static memory allocation is de -locate or free when the program terminates.

Example:

```
#include <iostream>
using namespace std;
int main()
{
   int x;//2 bytes
   char ch; // 1 bytes
   int a[100]; // 200 bytes
}
```

In the above code there are three variables, **x** is a integer variable, **ch** is a character variable and **a** is a integer array. In static memory allocation, during compilation, compiler calculates how much memory (here 203 bytes) these variables will need and finally find a location in computer memory with that size where these variables will be stored.

2. Dynamic memory allocation/Run Time Allocation

- In dynamic memory allocation, it allows us to define memory requirement during execution of the program.
- In this, the memory allocation is done dynamically at run time within the program run. The memory segment is known as a heap or the free store. Therefore, it is also called run time memory allocation.
- In this case, the exact space or number of the item does not have to be known by the compiler in advance.
- Memory de-allocation is also a part of this concept where the "clean-up" of space is done for variables or other data storage. It is the job of the programmer to de-allocate dynamically created space.

Dynamic Memory Allocation using new and delete operator

- Dynamic memory allocation allows our program to obtain more memory space while running, or to release it if it's not required.
- So the process of allocating and freeing memory at runtime is known as dynamic memory allocation. If the required size is not known until run-time (for example, if data of arbitrary size is being read from the user or from a disk file), then using fixed-size data objects is inadequate.
- **C++ provide two dynamic memory allocation operator: new and delete**
    1. New Operator
        - **New operator** is used to allocate memory at runtime.
        - The new operator denotes a request for memory allocation on the heap memory and if the requested memory is available the new operator allocates memory and returns a pointer to start of it.

          Syntax for new operator:

          Type *var=new Type; or Type *var=new Type(initial_value);

          Where Type can be any built in type or user defined type including structure and class.

          ‘

          Example:

          int *p=new int; //dynamically allocate an integer(2-byte) for loading the address in p. Here p is the pointer of type int

          *p=5; Store value using pointer

          Or above two statement can be written as

          int *p=new int(5);

- We also can use new operator to allocate block(array) of memory of some specified data types.

Syntax: Type * var=new Type[Size], where size specifies the number of items in an array

Example:

int *p=new int[50]; //dynamically allocate block of memory (200 bytes) of 50 integers and return the pointer to the first element which is assigned to p. Here p is the pointer of type int.

2. Delete Operator
   - It is mandatory to deallocate the dynamically allocated memory using new operator. It is the responsibility of programmer to deallocate dynamically allocated memory.
   - **So, Delete operator** is used to free or release the previously dynamically allocated memory using new.
   - Once the task with the dynamic allocated memory is done, it should be freed using delete operator so that it may be reused for some by dynamic memory request.

Syntax for delete operator:

delete var or delete (var);

Example:

```
int *p=new int; //dynamically allocate an integer(2-byte) for loading the address in
                p. Here p is the pointer of type int
*p=5; Store value using pointer
delete(p) // memory release
```

WAP to demonstrate dynamic memory allocation using new and delete operator

```
#include<iostream>
using namespace std;
int main()
{
    int *num=new int;//memory allocated at runtime
    *num=10; //assign value
    cout<<"Dynamically allocated address="<<num<<endl;
    cout<<"Value="<<*num;
    delete(num); //memory release
}
```

Or

```
#include<iostream>
using namespace std;
int main()
{
    int *num=new int(10);//memory allocated and initialized with 10 at runtime
    cout<<"Dynamically allocated address="<<num<<endl;
    cout<<"Value="<<*num;
    delete(num); //memory release
}
```

- In the above program, num is assigned an address that is large enough to hold an integer using new operator. The value 10 is stored in that address. Finally, using delete operator, the dynamically allocated memory is freed.

WAP to demonstrate dynamic memory allocation while creating object using new and delete operator

```
#include<iostream>
using namespace std;
class Demo
{
        private:
                int n1,n2;
        public:
                void setData()
                {
                        cout<<"Enter two numbers";
                        cin>>n1>>n2;
                }
                void displaySum()
                {
                        int sm;
                        sm=n1+n2;
                        cout<<"Sum of two number="<<sm;
                }
};

int main()
{
        Demo *d=new Demo();//pointer object
        d->setData();
        d->displaySum();
        delete(d);
}
```

Example 3:

WAP to find the greatest among N numbers entered by using dynamic memory allocation in c++.

```
#include<iostream>
using namespace std;
int main()
{
        int n,*data;
        cout<<"Enter value of N"<<endl;
        cin>>n;
        data=new int[n]; //Dynamic Memory Allocation
        cout<<"Enter "<<n<< " Numbers"<<endl;
        for(int i=0;i<n;i++)
        {
                cin>>*(data+i);
        }
        int max=*data;
        for(int i=0;i<n;i++)
        {
```

```
            if(*(data+i)>max)
            {
                max=*(data+i);
            }
        }
        cout<<"Largest Number="<<max;
        delete(data); //Releasing the dynamically allocated memory
}
```

Advantage/Disadvantage of DMA

Advantage

- The advantage of dynamic memory allocation is that *reuse* of memory is possible.
- Offers flexible memory allocation because size of memory can grow or shrink during runtime.
- Memory efficient program, can be developed as we can allocate exact amount of memory that is required.

Disadvantage

- Compiler does not help with allocation and deallocation. Programmer needs to both allocate and deallocate the memory.
- Memory leak is possible. Programmer needs to be careful while re-assigning memory to another variable.
- Slower as compared to static memory allocation since memory allocation is performed at runtime.

Static Allocation vs Dynamic Allocation

Static Allocation	Dynamic Allocation
Performed at static or compile time	Performed at dynamic or run time
Assigned to stack	Assigned to heap
Size must be know at compile time	Size may be unknown at compile time
First in last out	No particular order of assignment
It is best if required size of memory known in advance	It is best if we don't have idea about how much memory require

Pointer Object

- We can create pointers pointing to a class by using the class name as type of pointer.
- Example:

  Demo *d;

  In the above statement, d is pointer to an object of class Demo.
- We should use new operator to allocate memory for the object of Demo class as given below.

  d=new Demo();
- Also we can combine above two statements into single statements as

  Demo *d = new Demo();
- Similar to structure, we have to use arrow operator(→) to access the class Member by pointer to an object.

WAP to demonstrate the concept of pointer object.

```
#include<iostream>
using namespace std;
class Demo
{
        private:
                int n1,n2;
        public:
                void setData()
                {
                        cout<<"Enter two numbers";
                        cin>>n1>>n2;
                }
                void displaySum()
                {
                        int sm;
                        sm=n1+n2;
                        cout<<"Sum of two number="<<sm;
                }
};

int main()
{
        Demo *d=new Demo();//pointer object
        d->setData();
        d->displaySum();
        free(d);
}
```

Dynamic Constructor

- Dynamic constructor is used to allocate the memory to the objects at the run time.
- Allocation of memory to objects at the time of their construction is called dynamic construction/initialization of objects.
- The memory is allocated at runtime using a **new** operator and similarly, memory is deallocated at runtime using the **delete** operator.
- Using new operator, we can allocate right amount of memory for each object when they are not of same size, thus resulting memory utilization.

Example:

```
#include<iostream>
using namespace std;
#include<string.h>
class Demo
{
        char *name;
        public:

                Demo(char *str)
                {
                        int len;
                        len=strlen(str);
                        name=new char[len+1];
                        strcpy(name,str);
                }
```

```
        void display()
        {
                cout<<"Name= "<<name<<endl;
        }
};
int main()
{
        Demo d1("Bheeshma"), d2("Bhesh");
        d1.display();
        d2.display();
}
```

- In the above program the constructor function initializes the string dynamically at runtime.

WAP to initialize a field named name using dynamic constructor. Also write destructor to release the dynamically allocated memory.

```
#include<iostream>
using namespace std;
#include<string.h>
class Demo
{
        char *name;
        public:
                Demo(char *str)
                {
                        int len;
                        len=strlen(str);
                        name=new char[len+1];//name=new char(len+1);

                        strcpy(name,str);
                }
                void display()
                {
                        cout<<"Name= "<<name<<endl;
                }
                ~Demo()
                {
                        delete(name);
                }
};
int main()
{
        Demo d1("Bheeshma"), d2("Bhesh");
        d1.display();
        d2.display();
}
```

Issues in Creation and Initialization

Memory Map

Fig: Memory Map

- During the execution, a c++ Program is mapped into four different regions of memory each with different functionality. These includes
  1. Code/Program: This region of memory holds the compiled code of the program
  2. Static/Global Variable: This region of memory holds the static and global variable throughout the life time of program.
  3. Heap: This region of memory is free store of memory which can be allocated using DMA.
  4. Stack: This region of memory stores all the variables declared inside function, return address and arguments passed to the function and temporary results.

Memory Allocation: Stack Based and Heap Based Memory Allocation

- Memory in the C++ program is divided into two parts:
  1. Stack:
     - All variables declared inside any function take up the stack's memory.
     - Stack memory stores variables declared inside function call and is generally smaller than heap memory.
     - The size of memory to be allocated is known to the compiler and whenever a function is called, its variables get memory allocated on the stack. And whenever the function call is over, the memory for the variables is de-allocated.
     - This all happens using some predefined routines in the compiler.
     - A programmer does not have to worry about memory allocation and de-allocation of stack variables.
     - This kind of memory allocation is also known as Temporary memory allocation because as soon as the method finishes its execution all the data belonging to that method flushes out from the stack automatically.
     - This means any value stored in the stack memory scheme is accessible as long as the method hasn't completed its execution and is currently in a running state.

  2. Heap:
     - It is the unused memory of the program and can be used to allocate the memory at runtime dynamically.
     - The memory is allocated during the execution of instructions at runtime.
     - It is called a heap because it is a pile of memory space available to programmers to allocate and de-allocate.
     - Here programmer is responsible to handle the memory. C++ provides new and delete to allocate and release memory locations.

- Every time when we made an object, it always creates in Heap-space and the referencing information to these objects is always stored in Stack-memory.
- Heap memory allocation isn't as safe as Stack memory allocation if a programmer does not handle this memory well, a memory leak can happen in the program.
- Heap memory is accessible or exists as long as the whole application runs.

Stack vs Heap Based memory Allocation

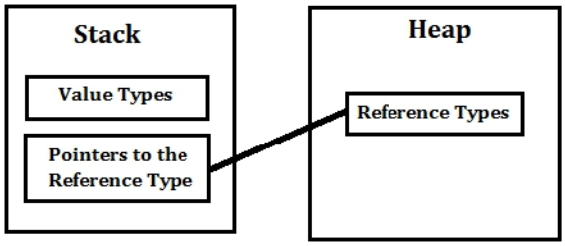


Fig: Stack Vs Heap

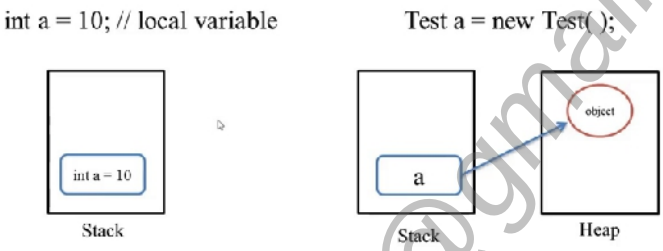


Fig: Example, Stack vs Heap

- Stack memory is automatically managed by OS and programmer has nothing to do but in heap memory allocation programmer is responsible for memory management.
- Stack memory allocation is considered safer as compared to heap memory allocation because in stack based allocation, programmer has nothing to do with allocation and deallocation operation.
- Memory allocation and de-allocation are faster as compared to Heap-memory allocation.
- Stack memory has limited storage(depends on OS) space as compared to Heap-memory.
- A stack is not flexible, the memory size allotted cannot be changed whereas a heap is flexible, and the allotted memory can be altered.
- Memory is allocated in a contiguous block in stack but non-contiguous block in Heap.
- Memory shortage problem is more likely to happen in stack whereas the main issue in heap memory is fragmentation.
- Stack implementation is easy as compared to Heap.
- Stack memory allocation is preferred in array while Heap memory allocation is preferred in linked list.

Memory Leak

- A memory can be dynamically allocated using new operator and released using delete operator in c++.
- It's programmer responsibility to release the dynamically allocated memory.
- Memory leak occurs when programmers create a memory in heap and forget to delete it.
- The consequences of memory leak is that it reduces the performance of the computer by reducing the amount of available memory. Eventually, in the worst case, too much of the available memory may become allocated and all or part of the system or device stops working correctly, the application fails, or the system slows down vastly.
- The below program is subjected to memory leak.

```
int main()
{
```

```
  int *ptr = new int[500];//dynamically allocate memory
  /* Do some work */

  return 0 ; /* Return without freeing ptr*/
}
```

Memory Recovery

- In heap based memory allocation, if the dynamically allocated memory is of no more use then we can recover it using the **delete** operator.
- The delete operator free up the previously allocated memory and make it reusable for future dynamic memory request.
- Also, to avoid memory leaks, memory allocated on heap should always be freed when no longer needed
- The following program demonstrate how we can recover the memory on memory heap.

```
int main()
{
  int *ptr = new int[500];
  /* Do some work */

  delete(ptr);// Memory Recovery
  return 0 ; /* Return without freeing ptr*/
}
```