Unit 3: Message, Instance and Initialization

3.1 Message Passing

- In OOP, objects communicate with one another by sending and receiving information with each other which is known as message passing.
- A message for an object is a request for execution of one of its operations or behavior (function/method)
- Sender object invokes a function from the receiving object which generates desired results.

Message Passing Formalization:

The formal process of message passing involves specifying the following:

- the name of the receiver object
- the name of the function (method) within the receiver object
- additional information to be sent (as arguments)

Message Passing Syntax in C++

Syntax

In Sender object's code:

```
receiver.function(arguments)
```

Example

```
Car myCar;              // myCar is an object of class Car
myCar.ChangeGear(3);    // Request myCar to change to 3rd gear
```

Sender Object : Object that invokes the method (function) of another object
Receiver Object : Object that performs the requested task
Dot operator : Enables access to the method (function) of the receiver object
Arguments : Additional information for the method being invoked

3.2 Mechanism for Instantiation (Creation) and Initialization

Instantiation:

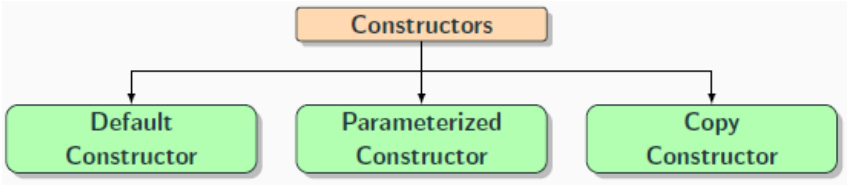
- When an object of a class is created, it is known as an instance of that class.
- The process of creating an object of a class is known as instantiation.

Initialization:

- Initialization involves assigning attributes to the properties of an object
  - properties = data members
  - attributes = values
- Initialization of properties can be done in either of the following ways:
  - after instantiation ➔ using member functions
  - during instantiation ➔ using Constructors

Constructors and its types

- Constructors are special public member functions of a class which is called automatically whenever an instance (object) of that class is created or constructed otherwise.
- Constructors can be implicitly supplied by the compiler or they can be explicitly written by the programmer.
- In C++, explicit constructors take the name of the class.
- Constructors can take any number of arguments and hence, they can be overloaded .i.e., be polymorphic.
- Constructors never return any value (not even void).
- Constructors are called automatically, and hence generally not called manually like other member functions.
- There are three types of Constructors:
  a) Default Constructor
  b) Parameterized Constructor
  c) Copy Constructor



a) Default Constructor:

- Default constructor is the most basic kind of constructor.
- Neither does it take any argument nor does it return any value.
- For every class, an implicit default constructor is automatically provided by the compiler which is responsible for carrying on the necessary actions during creation of an object from a class.
- The default constructor can also be overridden i.e., supplied explicitly by the programmer.
- It is generally used to initialize an object with some predefined default values or perform some pre-defined tasks at the time of instantiation.
- In C++, explicit default constructor is written as a member function using the name of the class.

b) Parameterized Constructor

- Constructors which take some arguments/parameters are called parameterized Constructors.
- The parameters passed to the constructor are generally used to initialize the properties of the object at the time of instantiation.
- Constructors can be overloaded by varying the types and/or number of arguments across different versions.

c) Copy Constructor

- A copy constructor is used to copy the contents from another object of the same class to the object being created.
- A copy constructor is automatically called when
    - An object is instantiated by initializing it using another object of the same class.
    - An object is passed by value as an argument to a function.
- A copy constructor is implicitly provided by the compiler. However, it can also be explicitly supplied by the programmer.
- A copy constructor is a special version of parameterized constructor which takes the reference to a source object of the same class as an argument from which the properties needs to be copied.

Example: Showing the usage of different types of constructors in a single program.

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

class Student{
  private:
    int    Roll;
    string Name;

  public:
    // default Constructor
    Student( )
    {
        Roll = 0;
        Name = "Unknown";
    }

    // Parameterized Constructor
    Student(int r, string n)
    {
        Roll = r;
        Name = n;
    }

    // Copy Constructor
    Student(const Student & s)
    {
        Roll = s.Roll + 1;
        Name = s.Name;
    }

    void Display()
    {
        cout << endl << "Roll : " << Roll;
        cout << endl << "Name : " << Name;
    }
};
```

```
int main()
{
    // Use of Default Constructor

    Student s1;
    s1.Display();             // Output: 0 and Unknown


    // Use of Parameterized Constructor

    Student s2(101, "Ram");
    s2.Display();             // Output: 101, Ram


    // Use of Copy Constructor

    Student s3(s2);
    s3.Display();             // Output: 102, Ram

    return 0;
}
```

3.3 Issues in Creation and Initialization

Memory Map

- Memory or RAM (Random Access Memory) is a crucial resource while running a program in computer. Hence, it should be managed efficiently.
- A program stored in secondary storage devices (like hard drive) must first be loaded into primary memory (RAM) so that the CPU can run the instructions one by one and store and retrieve the data during computation quickly and efficiently.
- As a programmer, it is necessary to have some knowledge on how the operating system manages memory and the way memory is accessible to our programs.
- When a program written in any high level language like C/C++ is executed, the memory management is quite complex. However, it can be divided into following four segments:
    - Code Segment
    - Global Segment
    - Stack Segment
    - Heap Area

a) Code Segment:

- Also called the Program Segment, it holds the compiled code (instructions) of the program.
- Codes of different functions/modules are stored in different memory address blocks.
- It is automatically freed after the program is terminated.

b) Static Segment

- Global variables and Static variables which must be stored throughout the lifetime of the entire program are stored in this segment.
- It is automatically freed after the program is terminated.

c) Stack Segment

- Last in First out type of memory responsible for keeping track of the current state of the CPU
    - Push operation= Insert New Data at the top of stack
    - Pop operation = Take the topmost data out from stack

- It is used for storing local (auto) variables of a function
- It is also responsible to store the return address when calling a function
- Arguments passed during a function call are also stored in Stack segment
- As soon as we return from a function, associated items from the stack are removed
- Only a limited amount of memory is reserved for Stack segment which is decided at compile time
- It is automatically freed after the program is terminated

d) Heap Area

- Heap is a large area with plenty of free memory chunks.
- It is generally used for Objects/variables created using DMA (Dynamic Memory Allocation).
- Memory occupied in Heap may not be freed automatically after the program is terminated. Hence, the programmer must manually de-allocate the memory when not required anymore.

Memory Allocation Methods

Static Memory Allocation

When a variable is declared normally, memory is allocated automatically. This is also termed as static memory allocation (not to be confused with static data type).

Dynamic Memory Allocation (DMA)

Memory can be allocated for variables as and when required dynamically during the execution of a program. This is done using the new operator in C++. For dynamic memory allocation, a pointer variable is required and the memory for actual data is allocated in the heap area.

DMA is especially useful when an arbitrarily sized array needs to be created whose size is determined only during the execution of a program. This prevents the array from being undersized or oversized, thus making optimum use of the computer's memory.

Memory Recovery

When memory is allocated dynamically, it must be freed manually when not required anymore using the delete operator. This is known as memory recovery.

In C++, when memory for data members of a class are created dynamically using a constructor, such a constructor is termed as ***dynamic constructor*** and such memory must be freed using a special member function called destructor.

Destructor

- A destructor is a special member function of a class which is called automatically just before an object gets destroyed.
- Generally, a destructor is used for the purpose of performing clean-up tasks before the object is destroyed.
- Destructors are implicitly provided by the compiler which can be overridden or explicitly written by the programmer.
- A destructor supplied explicitly by the programmer takes the name of the class preceded by the tilde sign (~).
- A destructor does not take any argument. So, it cannot be overloaded.
- Like constructor, a destructor does not return any value at all (not even void).

Example 1: Dynamic memory allocation for a single variable

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

int main()
{
    int* ptr;              // ptr is a pointer to int

    ptr  = new int;        // dynamic memory allocation
    *ptr = 555;
    cout << endl << "value = "<< *ptr;    // free the memory

    delete ptr;
    return 0;
}
```

Example 2: Dynamic Memory Allocation for an array

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

int main()
{
    int n;
    float *ptr;
    float  sum=0;

    cout << "How many data ?";
    cin >> n;
    ptr  = new float[n];    // Dynamic memory allocation

    cout << endl << "Enter " << n << " data:" << endl;
    for(int i=0; i<n; i++){
      cin >> ptr[i];
      sum += ptr[i];
    }

    cout << endl << "-----------------";
    for(int i=0; i<n; i++){
      cout << endl << "Elem[" << i << "] = " << ptr[i];
    }

    cout << endl << "-----------------";
    cout << endl << "Sum is: " << sum;
    cout << endl << "-----------------";

    delete [] ptr;      // free the memory
    return 0;
}
```

Example 3: Dynamic memory allocation and memory recovery inside a class using Dynamic Constructor and Destructor

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

class Student{
private:
    int     Roll;
    string  Name;
    int     NumSub;
    int     *Marks;
    int     Total;

public:
    Student(int num=1)        // dynamic constructor
    {
        NumSub = num;
        Marks  = new int[NumSub];
    }

    void SetData()
    {
        cout << endl << "Enter roll no.:" ;
        cin >> Roll;
        cout << endl << "Enter Name:" ;
        cin >> Name;
        cout << "Marks in " << NumSub << " Subjects: ";

        Total = 0;
        for(int i = 0; i < NumSub; i++)
        {
            cin >> Marks[i];
            Total += Marks[i];
        }
    }
```

```
    void ShowTotal()
    {
        cout << endl << "Roll : " << Roll;
        cout << endl << "Name : " << Name;
        cout << endl << "Total: " << Total;
    }

    ~Student()                          // destructor
    {
        // free the marks array
        // allocated dynamically by the constructor
        delete [] Marks;
    }
}; // end of student class

int main()
{
    Student s1(3);
    s1.SetData();
    s1.ShowTotal();


    return 0;
}
```

Example 4: Dynamic Memory Allocation for a single object

```
#include <iostream>
#define pi 3.14159265
using namespace std;

class Cylinder{
private:
    float radius;
    float height;
    float volume;
    float area;

public:
    void InputData()
    {
        cout << endl << "Enter radius :";
        cin  >> radius;
        cout << endl << "Enter height :";
        cin  >> height;
    }

    void ShowArea()
    {
        area = 2*pi*radius*(radius+height);
        cout<< endl << "Total Surface area = " << area;
    }

    void ShowVolume()
    {
        volume = pi*radius*radius*height;
        cout << endl << "Volume = " << volume;
    }
};  // end of class Cylinder
```

```
int main()
{
    Cylinder *p;        // p is a pointer to a Cylinder
    p = new Cylinder;   // DMA for a Cylinder object

    p->InputData();     // -> is pointer to member access
    p->ShowArea();
    p->ShowVolume();

    delete p;       // free the memory allocated via DMA
    return 0;
}
```

Example 5: Dynamic Memory Allocation for an array of objects

```
// Code of Cylinder class to be replicated from Example 4

int main()
{
    // ptr is a pointer array
    // which can point to an array of Cylinder objects
    Cylinder * ptr[5];
    ptr = new Cylinder[5];          // DMA

    for(int i = 0; i < 5; i++)
    {
        ptr[i]->InputData();
        ptr[i]->ShowVolume();
        ptr[i]->ShowArea();
    }

    free [] ptr;
    return 0;
}
```