

# **OBJECT ORIENTED PROGRAMMING IN C++ [UNIT-II]**

A class that has neither data nor code.

e.g;

```
class xyz
```

```
{
```

```
};
```

```
class abc
```

```
{
```

```
};
```

At initial level of project, some of the classes are not fully identified or not fully implemented, hence they are implemented as empty class.

Important for Exception Handling.

## Example: Access Specifiers

```
class MyClass
{
    public:           //access from anywhere
        int x;
    private:         //only access from within a class
        int y;
    protected:      //access from within a class ,or derived class
        int z;
};

void main()
{
    MyClass  CopyClass;
    CopyClass.x = 1;   //OK, Public Access.
    CopyClass.y = 2;   //Error! Y isn't a member of MyClass
    CopyClass.z = 3;   //Error! Z isn't a member of MyClass
}
```

There are three kind of access control specifiers:

- Private
- Public
- Protected

```
#include <iostream>
class student
{private:
int rollno;
char name[20];
void setdata(int rn, char* namein );
void outdata();
};
void main(){
student s1;
s1.setdata(1, "xx"); //can't access
s1.outdata(); //can't access
}
```

Access control is similar to private members, has more significance in Inheritance.

```
#include <iostream>
class student
{
protected:
int rollno;
char name[20];
void setdata(int rn, char *namein);
void outdata();
};
void main()
{
student s1;
s1.setdata(1, "xx"); //can't access protected (same as private)
s1.outdata(); //can't access
}
```

```
#include <iostream>
class student
{public:
int rollno;
char name[20];
void setdata(int rn, char *namein);
void outdata();
};
void main()
{student s1;
s1.rollno; // can access public data
s1.setdata(1, "xx"); //can access public function
s1.outdata(); //can access public function
}
```



# Encapsulation

- Encapsulation is nothing new to what we have read. **It is the method of combining the data and functions inside a class. Thus the data gets hidden from being accessed directly from outside the class.**
- This is similar to a **capsule** where several medicines are kept inside the capsule thus hiding them from being directly consumed from outside.
- **All the members of a class are private by default, thus preventing them from being accessed from outside the class.**

# Encapsulation

- Encapsulation is necessary to keep the details about an object hidden from the users of that object. **Details of an object are stored in its data members .This is the reason we make all the member variables of a class private and most of the member functions public.**
- **Member variables are made private so that these cannot be directly accessed from outside the class (to hide the details of any object of that class like how the data about the object is implemented) and so most member functions are made public to allow the users to access the data members through those functions.**

# Encapsulation

There are various benefits of encapsulated classes.

- Encapsulated classes reduce complexity.
- Help protect our data. A client cannot change an Account's balance if we encapsulate it.
- Encapsulated classes are easier to change. We can change the privacy of the data according to the requirement without changing the whole program by using access modifiers (public, private, protected). For example, if a data member is declared private and we wish to make it directly accessible from anywhere outside the class, we just need to replace the specifier private by public.

# Encapsulation

```
using namespace std;
```

```
class Rectangle
```

```
{
```

```
    int length;
```

```
    int breadth;
```

```
    public:
```

```
        void setDimension(int l, int b)
```

```
        {
```

```
            length = l;
```

```
            breadth = b;
```

```
        }
```

```
        int getArea()
```

```
        {
```

```
            return length * breadth;
```

```
        }
```

```
};
```

```
int main()
```

```
{
```

```
    Rectangle rt;
```

```
    rt.setDimension(7, 4);
```

```
    cout << rt.getArea() << endl;
```

```
    return 0;
```

```
}
```

**The member variables length and breadth are encapsulated inside the class Rectangle. Since we declared these private, so these variables cannot be accessed directly from outside the class.**

**Therefore , we used the functions 'setDimension' and 'getArea' to access them.**

The data is hidden, so that it will be safe from accidental manipulation.

**Private members/methods** can only be accessed by methods defined as part of the class. Data is most often defined as private to prevent direct outside access from other classes. Private members can be accessed by members of the class.

**Public members/methods** can be accessed from anywhere in the program. Class methods are usually public which is used to manipulate the data present in the class. As a general rule, data should not be declared public. Public members can be accessed by members and objects of the class.

**Protected member/methods** are private within a class and are available for private access in the derived class.

## REAL LIFE EXAMPLE OF INFORMATION HIDING:

My Name and personal Information is stored in My Brain, no body can access this information directly. For getting this information you need to ask me about it and it will be up to myself that how much details I would like to share with you.

An Email Server may have account information of millions of people but it will share only my account information with me if I request it to send anyone else accounts information, my request will be refused.

Everything in C++ is associated with classes and objects, along with its attributes and methods. For example: in real life, a car is an object. The car has attributes, such as weight and color, and methods, such as drive and brake.

Attributes and methods are basically variables and functions that belongs to the class. These are often referred to as "class members".

A class is a user-defined data type that we can use in our program, and it works as an object constructor, or a "blueprint" for creating objects.



A Class is a user defined data-type which has data members and member functions.

Data members are the data variables and member functions are the functions used to manipulate these variables and together these data members and member functions defines the properties and behavior of the objects in a Class.

In the above example of class Car, the data member will be speed limit, mileage etc and member functions can be apply brakes, increase speed etc.

An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

**To access public members of a class, we use the (.)dot operator.**

```
#include <iostream>
using namespace std;
class Phone {
public:
    double cost;
    int slots;
};
int main() {
    Phone Y6;
    Phone Y7;

    Y6.cost = 100.0;
    Y6.slots = 2;

    Y7.cost = 200.0;
    Y7.slots = 2;
    cout << "Cost of Huawei Y6 : " << Y6.cost << endl;
    cout << "Cost of Huawei Y7 : " << Y7.cost << endl;

    cout << "Number of card slots for Huawei Y6 : " << Y6.slots << endl;
    cout << "Number of card slots for Huawei Y7 : " << Y7.slots << endl;

    return 0;
}
```

## You can create multiple objects of one class:

```
// Create a Car class with some attributes
```

```
class Car {  
    public:  
        string brand;  
        string model;  
        int year;  
};
```

```
int main() {
```

```
    // Create an object of Car
```

```
    Car carObj1;  
    carObj1.brand = "BMW";  
    carObj1.model = "X5";  
    carObj1.year = 1999;
```

```
    // Create another object of Car
```

```
    Car carObj2;  
    carObj2.brand = "Ford";  
    carObj2.model = "Mustang";  
    carObj2.year = 1969;
```

```
    // Print attribute values
```

```
    cout << carObj1.brand << " " << carObj1.model << " " << carObj1.year << "\n";  
    cout << carObj2.brand << " " << carObj2.model << " " << carObj2.year << "\n";  
    return 0;  
}
```

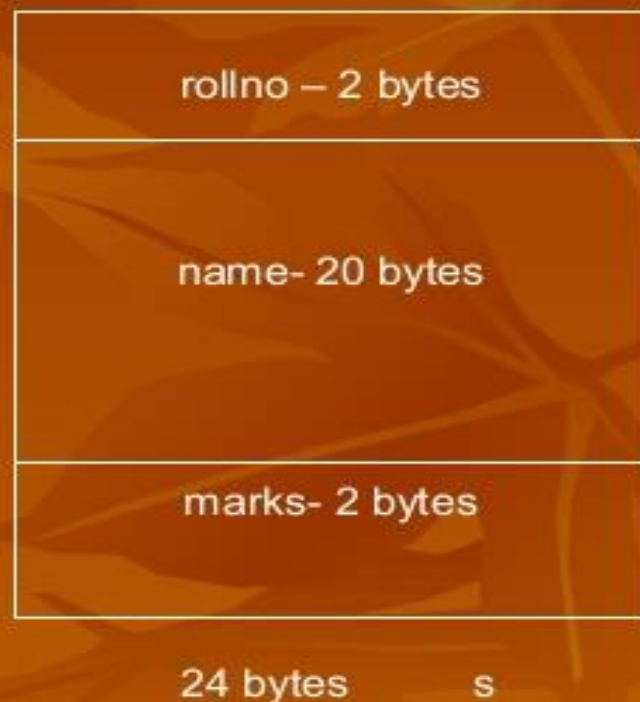
An object is an entity which has some properties and behavior associated with it. Objects are the basic run time entities in an object oriented system. programming problems are analyzed in terms of objects.

The main purpose of using objects are following.

- They correspond to the real life entities.
- They provide interactions with the real world.
- They provide practical approach for the implementation of the solution.

# Memory Allocation of Object

```
class student  
{  
    int rollno;  
    char name[20];  
    int marks;  
};  
student s;
```



# Object (state, behavior and identity)

**All the objects have a state, behavior and identity.**

State of an object - The **state or attributes** are the built in characteristics or properties of an object. For example, a T.V **has the size, colour, model etc.**

Behaviour of the object - The **behavior or operations** of an object are its functions. For example, a T.V. **can show picture , change channels, tune for a channel etc.** in object oriented programming terminology the behavior is implemented through methods.

Object identity - Each object is uniquely identifiable. For example, the **fridge can not become the T.V.**

An object consists of the following.

- **Data members** - Data members are the variables. They establish the state or attributes for the object.
- **Member functions** - Member functions represent the code to manipulate the data. The behaviour of the object is determined by the member functions.

```
class triangle
{
    private :
        float side1;
        float side2;
        float side3;
        float area;
    public :
        void read_data ( );
        void area_triangle ( );
        void display ( );
};
```

**Objects of the class triangle can be declared as:**

**Triangle t;**

Here t is object of the class triangle. A number of objects can also be declared as:

**Triangle t1 ,t2, t3;**

Here t1 ,t2, t3 are three objects of the class triangle. Each object has separate copy of data members.

Object can also be created when a class is defined by placing their names immediately after the closing brace. For example,

```
class triangle  
{  
...  
} t1 ,t2, t3 ;
```

Here t1 ,t2, t3 are objects of class triangle.



# What is 'Friend'

- Friend declarations introduce extra coupling between classes
  - Once an object is declared as a friend, it has access to all non-public members as if they were public
- Access is unidirectional
  - If B is friend of A, B can access A's non-public members; A cannot access B's
- A friend function of a class is defined outside of that class's scope
- The major use of friends is
  - to provide more efficient access to data members than the function call
  - to accommodate operator functions with easy access to private data members
- Friends have access to everything, which defeats data hiding, so use them carefully
- Friends have permission to change the internal state from outside the class. Always recommend use member functions instead of friends to change state

## Need

- a data is declared as private inside a class, then it is not accessible from outside the class.
- A function that is not a member or an external class will not be able to access the private data.
- A programmer may have a situation where he or she would need to access private data from non-member functions and external classes. For handling such cases, the concept of Friend functions is a useful tool.

What is a Friend Function?

- A friend function is used for accessing the non-public members of a class.
- A class can allow non-member functions and other classes to access its own private data, by making them friends.
- Thus, a friend function is an ordinary function or a member of another class.

## General syntax

Class ABC

{

.....

..... public :

.....

.....

Friend void func1(void);

};

## Special Characteristics

- The **keyword friend** is placed only in the function declaration of the friend function and **not in the function definition**.
- It is not in the scope of the class to which it has been declared as friend.
- It is possible to declare a function as friend in any number of classes.
- When a class is declared as a friend, the friend class has access to the private data of the class that made this a friend.
- A friend function, even though it is not a member function, would have the rights to access the private members of the class.
- It is possible to declare the friend function as either private or public without affecting meaning.
- The function can be invoked without the use of an object.
- It has object as argument.

```
#include <iostream>
using namespace std;
class exforsys
{
private:
int a,b;
public:
void test()
{
a=100; b=200;
}
friend int compute(exforsys e1);
//Friend Function Declaration with
keyword friend and with the object of
class exforsys to which it is friend
passed to it
};
int compute(exforsys e1)
{
//Friend Function Definition
which has access to private data
```

```
return int(e1.a+e1.b)-5;
}
void main()
{
exforsys e;
e.test();
cout << "The result is:" <<
compute(e);
//Calling of Friend Function with
object as argument.
}
```

A structure or an array in C++ can be initialized at the time of their declaration. For example;

```
struct student
{
int rollno;
float marks;
};

int main()
{student s1= {0, 0.0};
}
```

But **such initialization does not work** for a class because class members have their associated access specifier. They might not be available to the outside world (outside their class). The following code snippet is invalid in C++.

# Need for Constructors

```
class student { private:
    int rollno;
    float marks;
    public:
        : // public members
};

int main()
{ student senior = { 0, 0.0}; // illegal ! Data members are not
accessible.
    :
    :
};
```

The private data members are inaccessible by a non-member function in main().



- There can be an alternative solution to it. If we define a member function (say `init()`) inside a class to provide the initial values, as it is shown below:

```
class student { private:
    int rollno; float marks;
public:
    void init()
    { rollno = 0; marks=0.0;      }
    : //other public members
};
```

```
int main()
{ student senior ; // create an object
  senior.init();   // initialize it
  :
}
```

# Need for Constructors

- Here the programmer has to explicitly invoke the function called `init()`, in order to initialise the object.
- Thus the responsibility of initialization lies solely with the programmer . What if, the programmer fails to invoke `init()`? The object in such case is full of garbage and might cause havoc to your program.
- Thus the responsibility of initialisation is taken away from the programmer and given to the compiler because the compiler exactly knows when objects are created.
- Therefore every time an object is created , the compiler will automatically initialise it by invoking the initialisation function but if and only if the initialization function bears the same name as that of the class.
- And, Obviously this function is known as a constructor.

- A constructor is a **special member function** whose task is to initialize the data members of an objects of its class.
- It is special because it has **same name as its class name**.
- It **invokes automatically** whenever a new object of its associated class is created.
- It is called constructor because it constructs the initial values of data members and build your programmatic object.

# Introduction

- It is very common for some part of an object to require initialization before it can be used.
- Suppose you are working on 100's of objects and the default value of a particular data member is needed to be zero.
- Initialising all objects manually will be very tedious job.
- Instead, you can define a constructor function which initialises that data member to zero. Then all you have to do is declare object and constructor will initialise object automatically.

# Constructors

- There is no need to write any statement to invoke the constructor function.
- If a 'normal' member function is defined for initialization, we need to invoke that function for each and every objects separately.
- A constructor that accepts no parameters is called the default constructor.
- The default constructor for class A will be `A : : A ( )`

# Characteristics of Constructor

- They must be declared in the public scope.
- They are invoked automatically when the objects are created.
- They do not have return types, not even void and they cannot return values.
- They cannot be inherited, though a derived class can call the base class constructor.
- Like other C++ functions, Constructors can have default arguments.
- Constructors can not be virtual.

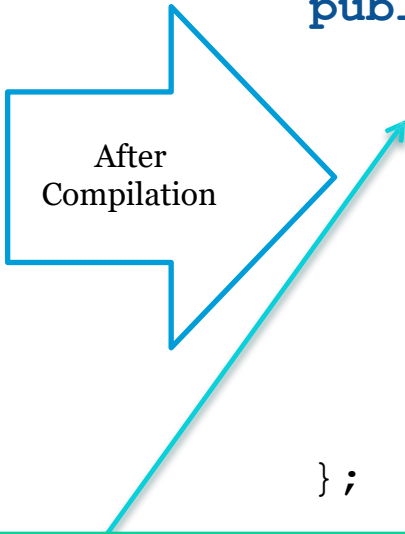
# Constructor

- The constructor function is responsible for creation of object.
- But in previous examples, we have not defined any constructor in class, so how come the objects were created of those classes?
- The answer is, If no constructor is defined in the class in such situation the compiler implicitly provides a constructor, which is called as **default constructor**.

# Constructor

```
class sample
{
    int someDataMember;
public:
    void someFunction ()
    {
        ..
        ..
    }
};
```

After  
Compilation



```
class sample
{
    int someDataMember;
public :
    sample ()
    {
    }
    void someFunction ()
    {
        ..
        ..
    }
};
```

Compiler has implicitly added a constructor to the class, which has empty body, because compiler is not supposed to put any logic in that.



# Types of Constructor

- ☐ Default Constructor/Non-Argument Constructor
- ☐ Parameterized Constructor
- ☐ Copy Constructor

# Default Constructor

- A constructor without any parameter is known as non-argument constructor or simply default constructor.
- If no constructor is defined in a class, then compiler implicitly provides an empty body constructor which is called as default constructor.

# Non-Argument Constructor Example

```
class circle
{
    float radius;
public:
    circle()
    {
        radius = 0;
    }
};
```

- In example beside, the constructor function takes no argument, and simply initializes radius to zero.
- Non-arg constructor is also called as default constructor.

# Parameterised Constructors

- Sometimes it becomes necessary to initialize the various data elements of an objects with different values when they are created.
- This is achieved by passing arguments to the constructor function when the objects are created.
- The constructors that can take arguments are called parameterized constructors.

# Parameterised Constructors

```
class circle
{
    float radius;
public:
    circle()
    {
        radius = 0;
    }

    circle(float r)
    {
        radius = r;
    }
};
```

Non-Arg (Default) constructor,  
which takes no arguments

Parametirised constructor, which  
takes 1 arguments as radius.

# Parameterised Constructors

```
class circle
{
    float radius;
public:
    circle()
    {
        radius = 0;
    }

    circle(float r)
    {
        radius = r;
    }
};
```

- When a constructor is parameterized, we must pass the arguments to the constructor function when an object is declared.
- Consider following declaration  

```
circle obj1;

circle sobj2(15);
```

# Two Ways of calling a Constructor

```
class circle
{
    float radius;
public:
    circle()
    {
        radius = 0;
    }

    circle(float r)
    {
        radius = r;
    }
};
```

- Implicit call (shorthand method)

- circle ob(7.6);

- Explicit call

- circle ob;

- ob = circle(7.6);

# Multiple Constructors in a Class

- C++ permits to use more than one constructors in a single class.
- `Add( ) ;` // No arguments
- `Add (int, int) ;` // Two arguments



# Multiple Constructors in a Class

```
class add
{
int m, n ;
public :
add ( )
{
m = 0 ; n = 0 ;

}
add (int a, int b)
{
m = a ; n = b ;
}
add (add & i)
{
m = i.m ; n = i.n ;
}
```

- The first constructor receives no arguments.
- The second constructor receives two integer arguments.
- The third constructor receives one add object as an argument.

# Multiple Constructors in a Class

```
class add
{
int m, n ; public :
add ( )
{
m = 0 ; n = 0 ;
}
add (int a, int b)
{
m = a ; n = b ;
}
add (add & i)
{
m = i.m ; n = i.n ;
}
};
```

- Add a1;
  - Would automatically invoke the first constructor and set both m and n of a1 to zero.
- Add a2(10,20);
  - Would call the second constructor which will initialize the data members m and n of a2 to 10 and 20 respectively.

# Multiple Constructors in a Class

```
class add
{
int m, n ; public :
add ( ) {m = 0 ; n = 0 ;}
add (int a, int b)
{m = a ; n = b ;}
  add (add & i)
  {m = i.m ; n = i.n ;}

};
```

- Add a3(a2);
  - Would invoke the third constructor which copies the values of a2 into a3.
  - This type of constructor is called the “copy constructor”.
- Construction Overloading
  - More than one constructor function is defined in a class.

# Constructors with Default Arguments

- It is possible to define constructors with default arguments.
- Consider `complex (float real, float imag = 0);`
  - The default value of the argument `imag` is zero.
  - `complex C1 (5.0)` assigns the value 5.0 to the real variable and 0.0 to `imag`.
  - `complex C2(2.0,3.0)` assigns the value 2.0 to real and 3.0 to `imag`.

# Constructors with Default Arguments

- `A :: A ( )` → Default constructor
- `A :: A (int = 0)` → Default argument constructor
- The default argument constructor can be called with either one argument or no arguments.
- When called with no arguments, it becomes a default constructor.

# Copy Constructor

The copy constructor is, however, defined in the class as a parameterized constructor receiving an **object** as **argument passed by reference**.

## Copy Constructor

A copy constructor is used to declare and initialize an object from another object.

```
class student
{
    .....;
public:
    student(student &obj)    // copy constructor
    {
        .....;
    }
};
```

# Copy Constructor

- The **copy constructor** is a constructor which creates an object by initializing it with an object of the same class, which has been created previously. The copy constructor is used to:
- Initialize one object from another of the same type.
- Copy an object to pass it as an argument to a function.
- Copy an object to return it from a function.
- If a copy constructor is not defined in a class, the compiler itself defines one. If the class has pointer variables and has some dynamic memory allocations, then it is a must to have a copy constructor.
- A copy constructor is a constructor of the form **classname(classname &)**.

# Copy Constructor

- In the above given class, two constructors have been used; a parameterized and a copy constructor sharing a common class name i.e. address. Thus, constructor can be overloaded.
- It may be noted that the code for copy constructor has to be written by the programmer.
- It may be further noted that the argument to a copy constructor has been passed by reference.
- **We can not pass the argument by value because this will result in copy calling itself on and on until the compiler runs out of memory.**
- Let us now write a program that uses this class to create an object called obj1 with some initial values. It also creates another object called obj2 and copies the contents of obj1 into obj2. It then displays the contents of obj2.



- An object called **obj1** of class **address** is created with some initial values. Another object **obj2**, a copy of **obj1**, is created with the help of the copy constructor.
- At this stage, a question which arises is that, this job could have been done by simple assignment of objects i.e. **obj1 = obj2**, then why to use a copy constructor?
- The need of a copy constructor is felt when the class includes pointers which need to be properly initialized. A simple assignment will fail to do this, **because both the copies will hold pointers to same memory location.**

# Copy Constructor

```
{  
int data;  
public:  
example()           // Default Constructor  
{ }  
  
example(int x)       // Parameterise Constructor  
{  
    data=x;  
}  
void display()       // To display value in data  
{  
    cout<<" data ="<<data<<endl;  
}
```

# Copy Constructor

```
int main()  
{  
    Values in obj1 is copied in obj2  
    example obj1(50);  
    example obj2(obj1);  
    return 0; // This is called as Copy Constructor  
}
```



```
class example  
{  
    int data;  
    public:  
    example()  
    { }
```

```
    example(int x)  
    {  
        data=x;  
    }  
    void display()  
    {  
        cout<<" data = "<<d  
    }
```



# Copy Constructor

*But we have not written any code for this..??*

*This is because compiler automatically makes copy constructor*

*But it is Good Practice if we manually made Copy Constructor*

*Let's See Various Calling Forms of Copy Constructor*

# Copy Constructor

## Call Copy Constructor

Syntax : -

```
Class_name target_object(source_object);
```

```
Class_name target_object = source_object;
```

Ex: -

```
student target(source);
```

```
student target = source;
```

# Copy Constructor

*Also be written as*

*example obj1(50);  
example obj2(obj1);*

*example obj1(50);  
example obj2;  
obj2=obj1;*

*Also be written as*

*example obj1(50);  
example obj2=obj1;*

*So these are the three forms*



# Copy Constructor

Start here x c++.cpp x this is the same program as we have seen

```

1  #include <iostream>
2  using namespace std;
3  class example
4  {
5      int data;
6  public:
7      example()
8      { }
9      example(int x)
10     {
11         data=x;
12     }
13     void display()
14     {
15         cout<<"data ="<<data<<endl;
16     }
17 };
18
19 int main()
20 {
21     example obj1(50);
22     example obj2(obj1);
23     obj1.display();
24     obj2.display();
25     return 0;
26 }
27

```

//in this program compiler will make  
copy constructor automatically

D:\codeblock\c++\c++.exe

```

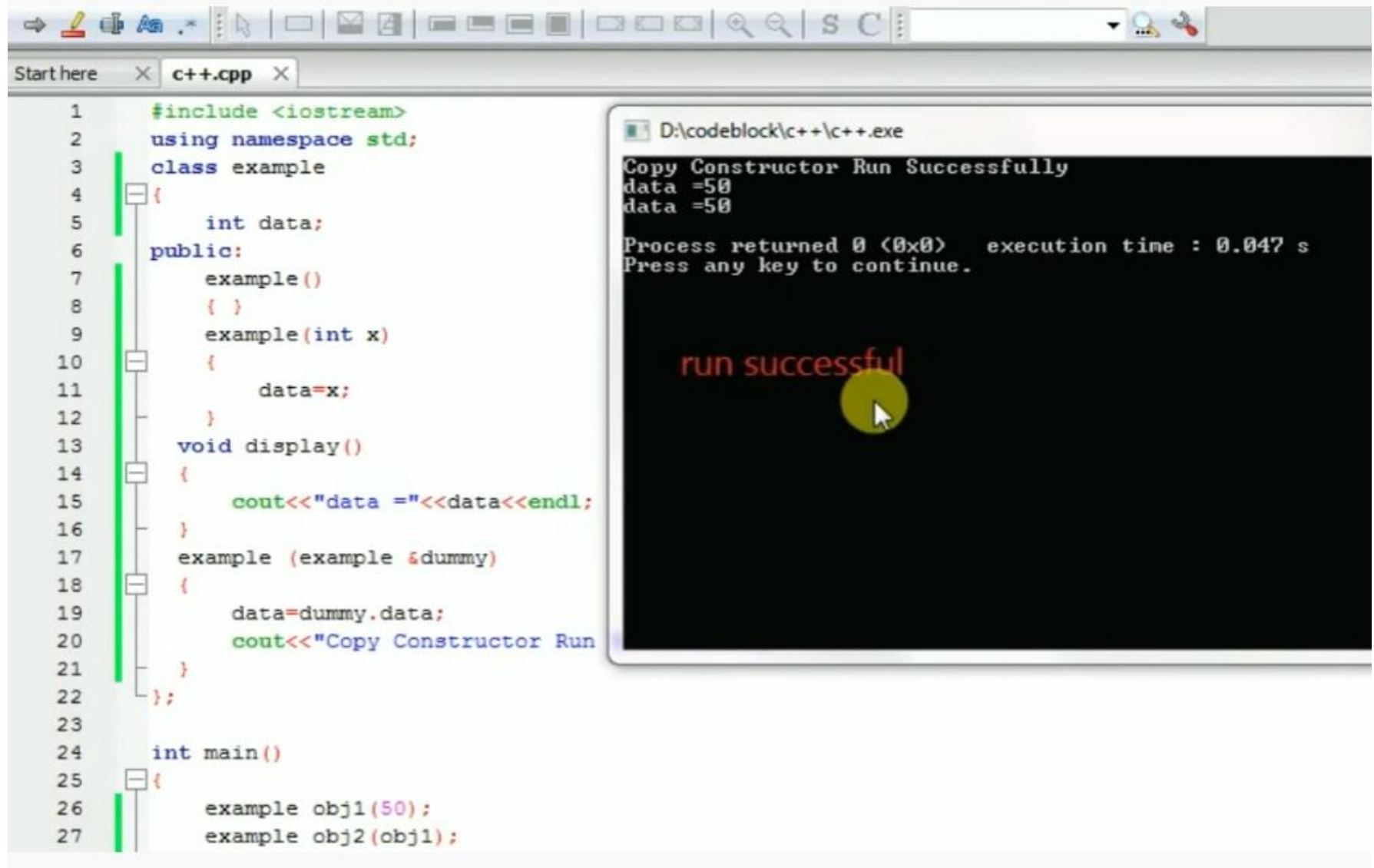
data =50
data =50
Process returned 0 (0x0)   execution time : 0.018 s
Press any key to continue.

```

//value copied successfully

let's make manual copy constructor|

# Copy Constructor



The screenshot shows the Code::Blocks IDE with a C++ file named `c++.cpp` open. The code defines a class `example` with an integer member `data`. It includes a default constructor, a parameterized constructor, and a copy constructor. The `main` function creates two objects: `obj1` with value 50, and `obj2` as a copy of `obj1`. A terminal window on the right shows the successful execution of the program, displaying the output "Copy Constructor Run Successfully" and "data =50" twice, with a process return code of 0 and an execution time of 0.047 seconds.

```
1  #include <iostream>
2  using namespace std;
3  class example
4  {
5      int data;
6  public:
7      example()
8      { }
9      example(int x)
10     {
11         data=x;
12     }
13     void display()
14     {
15         cout<<"data ="<<data<<endl;
16     }
17     example (example &dummy)
18     {
19         data=dummy.data;
20         cout<<"Copy Constructor Run
21     }
22 };
23
24 int main()
25 {
26     example obj1(50);
27     example obj2(obj1);
```

Terminal Output:

```
D:\codeblock\c++\c++.exe
Copy Constructor Run Successfully
data =50
data =50
Process returned 0 (0x0)   execution time : 0.047 s
Press any key to continue.

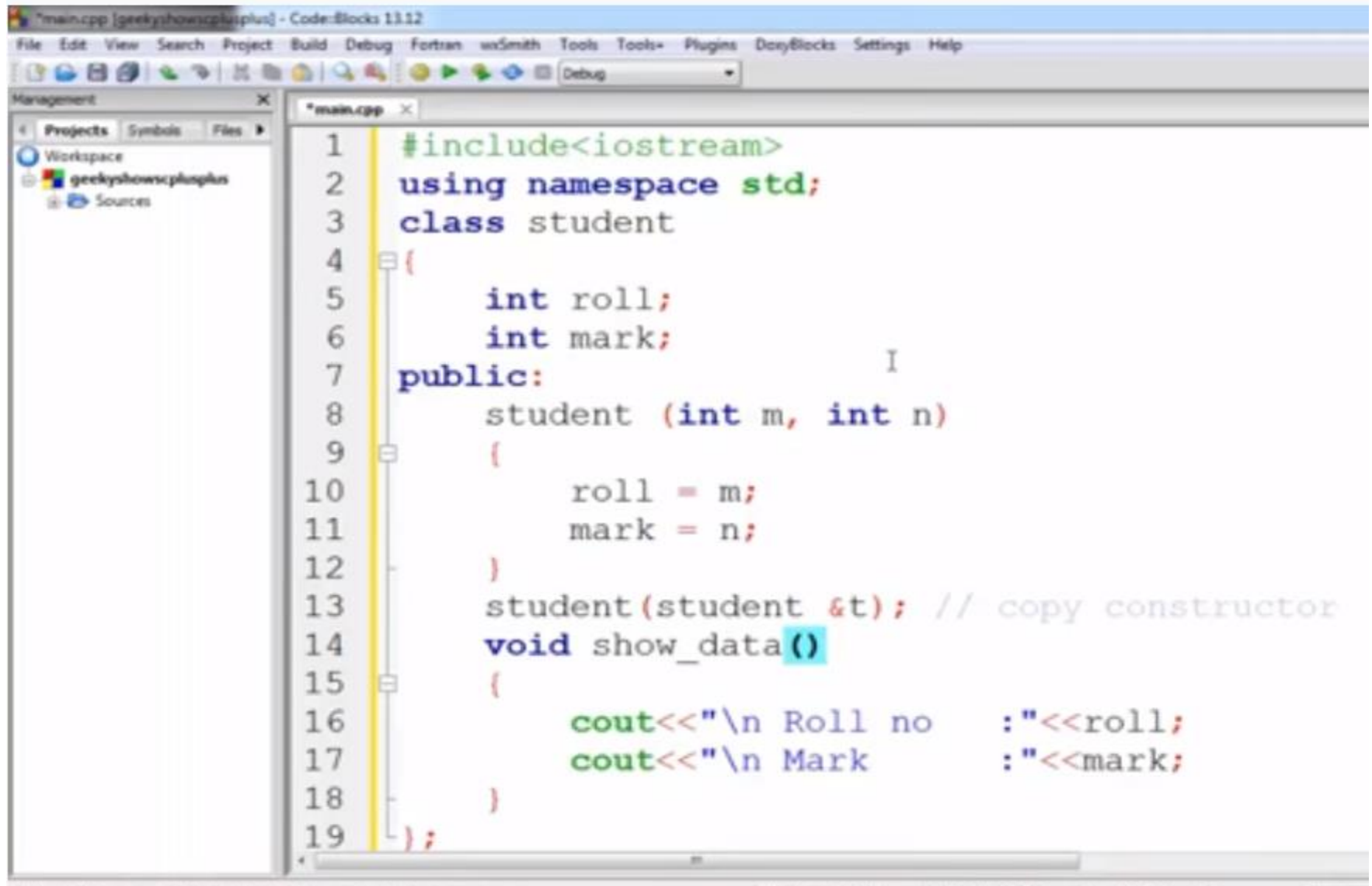
run successful
```



# Copy Constructor

```
class student
{
    .....;
public:
    student (student &obj); //copy constructor
};
student :: student(student &obj)
{
    .....;
}
```

# Copy Constructor



```

1  #include<iostream>
2  using namespace std;
3  class student
4  {
5      int roll;
6      int mark;
7  public:
8      student (int m, int n)
9      {
10         roll = m;
11         mark = n;
12     }
13     student(student &t); // copy constructor
14     void show_data()
15     {
16         cout<<"\n Roll no    : "<<roll;
17         cout<<"\n Mark      : "<<mark;
18     }
19 };
    
```

# Copy Constructor

```
student :: student(student &t) // copy constructor def
{
    roll = t.roll;
    mark = t.mark;
}

int main()
{
    cout << "\nParameterized constructor output r\n";
    student r(60, 130); // parameterized constructor
    r.show_data();      // parameterized constructor o
    cout << "\n\nCopy constructor output stu(r)\n";
    student stu(r);     // copy constructor
    stu.show_data();    // copy constructor output
    return 0;
}
```



# Deep & Shallow Copying

• **Shallow copy:** when we don't define our copy constructor and assignment operator, then compiler defines copy constructor and assignment operator for us and it provides a copying method known as a **shallow copy**, also known as a **memberwise copy**.

A **shallow copy** copies all of the member variable values. This works efficient if all the member variables are values, but may not work well if member variable point to dynamically allocated memory. The pointer will be copied but the memory it points to will not be copied, the variable in both the original object and the copy will then point to the same dynamically allocated memory, which is not usually what you want. The default copy constructor and assignment operator make shallow copies.

# Deep & Shallow Coping

- **Deep copy:** A deep copy copies all member variables, and makes copies of dynamically allocated memory pointed to by the variables. To make a **deep copy**, we have to define our copy constructor and overload the assignment operator.

## Requirements for deep copy in the class

- A destructor is required to delete the dynamically allocated memory.
- A copy constructor is required to make a copy of the dynamically allocated memory.
- An overloaded assignment operator is required to make a copy of the dynamically allocated memory.

# Copy Constructor

**Why Copy Constructor Take Argument As Reference?**

<https://www.youtube.com/watch?v=QgY7ro4RkK8>

# Destructors

- Whenever an object is created within a program, it also needs to be destroyed.
- If a class has constructor to initialize members, it should also have a destructor to free up the used memory.
- A destructor, as the name suggest, destroys the values of the object created by the constructor when the object goes out of scope.
- A destructor is also a member function whose name is the same name as that of a class, but is preceded by tilde (‘~’).
- For example, the destructor of class **salesperson** will be **~salesperson()**.
- A destructor does not take any arguments nor does it return any value. The compiler automatically calls them when the objects are destroyed.

# Destructors

A destructor is a special member function that works just opposite to constructor, unlike constructors that are used for initializing an object, destructors destroy (or delete) the object.

## Syntax of Destructor

```
~class_name()  
{  
    //Some code  
}
```

Similar to constructor, the destructor name should exactly match with the class name. A destructor declaration should always begin with the tilde(~) symbol as shown in the syntax above.



# Destructors

## When does the destructor get called?

A destructor function is called automatically when the object goes out of scope:

- The program finished execution/the program ends .
- The function ends
- A block containing local variables ends. When a scope (the { } parenthesis) containing local variable ends.
- A delete operator is called

### Can there be more than one destructor in a class?

No, there can only one destructor in a class with classname preceded by ~, no parameters and no return type.

## Destructor rules

- 1) Name should begin with tilde sign(~) and must match class name.
- 2) There cannot be more than one destructor in a class.
- 3) Unlike constructors that can have parameters, destructors do not allow any parameter.
- 4) They do not have any return type, just like constructors.
- 5) When you do not specify any destructor in a class, compiler generates a default destructor and inserts it into your code.

# Destructors

```
#include <iostream>
using namespace std;
class HelloWorld{
public:
    //Constructor
    HelloWorld(){
        cout<<"Constructor is called"<<endl;
    }
    //Destructor
    ~HelloWorld(){
        cout<<"Destructor is called"<<endl;
    }
    //Member function
    void display(){
        cout<<"Hello World!"<<endl;
    }
};
```

```
int main(){
    //Object created
    HelloWorld obj;
    //Member function called
    obj.display();
    return 0;
}
```

## Output:

Constructor is called  
Hello World!  
Destructor is called

# Destructors

```
#include<iostream>
using namespace std;
class Demo {
private:
int num1, num2;
public:
Demo(int n1, int n2) {
    cout<<"Inside Constructor"<<endl;
    num1 = n1;
    num2 = n2;
}
void display() {
    cout<<"num1 = "<< num1 <<endl;
    cout<<"num2 = "<< num2 <<endl;
}
~Demo() {
    cout<<"Inside Destructor";
}
};
```

```
int main() {
    Demo obj1(10, 20);
    obj1.display();
    return 0;
}
```

Output

```
Inside Constructor
num1 = 10
num2 = 20
Inside Destructor
```

# Destructors

In the above program, the class Demo contains a parameterized constructor that initializes num1 and num2 with the values provided by n1 and n2. It also contains a function display() that prints the value of num1 and num2. There is also a destructor in Demo that is called when the scope of the class object is ended.

# Destructors

```
#include <iostream>

using namespace std;
class Line {
public:
    void setLength( double len );
    double getLength( void );
    Line(); // This is the constructor declaration
    ~Line(); // This is the destructor: declaration

private:
    double length;
};

// Member functions definitions including constructor
Line::Line(void) {
    cout << "Object is being created" << endl;
}
Line::~~Line(void) {
    cout << "Object is being deleted" << endl;
}
void Line::setLength( double len ) {
    length = len;
}
double Line::getLength( void ) {
    return length;
}
```

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

    // set line length
    line.setLength(6.0);
    cout << "Length of line : " << line.getLength() << endl;

    return 0;
}
```

Output:

Object is being created  
Length of line : 6  
Object is being deleted

# Conclusion

- Constructors are normally used to initialise variables and to allocate memory.
- Similar to normal functions, constructors can be overloaded.
- When an object is created and initialized at the same time, a copy constructor gets called.
- C++ also provides destructor that destroys the objects when they are no longer required.
- Allocation of memory objects to objects at the times of their construction is known as dynamic construction of objects with the help of new operator and can be deleted through delete operator.

# Summary

- A class is a data type defined by the user where class describes the data and its behavior or functionality. It serves as a template to create objects.
- The objects can be passed to as well as returned from functions. Inline, constant, nested, friend and static functions add new dimension to the flexibility of C++ program.
- A constructor is a member function with the same name as that of its class and is used to initialise the objects of that class with a legal initial value.



# Summary

- A constructor may be called explicitly as well as implicitly. It has various types depending on the basic demand of the program.
- Objects can be initialized dynamically also with new operator. Make sure the memory allocated through new must be properly deleted through delete. Improper use of new and delete may lead to memory leak.
- A destructor deinitializes an object before it goes out of scope and follows the same access rules of class members.

# Instantiation in C++

- In OOP (object-oriented programming), a class of object may be defined. ... An instance of that object may then be declared, giving it a unique, named identity so that it may be used in the program. This process is called "instantiation."
- Used to create an object (class instance) from a class.

## Syntax

- `className objectName(parameters);`
- Input requirements are taken from the constructor. A class can have multiple constructors with different numbers of input parameters and types, to create different objects.
- An instance of a class is called an object. Instantiation is also known as an instance.

# Instantiation in C++

- Until an object becomes instantiated, none of the code within the relevant class declarations is used.
- Prior to modern OOP methods, instantiate had a similar meaning in relation to the creation of data within an empty template. For example, the entry of a record into a database was considered to be instantiation.

# Default Arguments (Parameters)

- In C++ programming, we can provide default values for function parameters.
- If a function with default arguments is called without passing arguments, then the default parameters are used.
- However, if arguments are passed while calling the function, the default arguments are ignored.

# Default Arguments (Parameters)

- The default arguments are used when you provide no arguments or only few arguments while calling a function. The default arguments are used during compilation of program.
- For example, let's say you have a user-defined function `sum` declared like this: **`int sum(int a=10, int b=20)`**, now while calling this function you do not provide any arguments, **simply called `sum()`**; then in this case the **result would be 30**, compiler used the **default values 10 and 20** declared in function signature.
- If you pass only one argument like this: **`sum(80)`** then the result would be **100**, using the passed argument 80 as first value and 20 taken from the default argument.


# Default Arguments (Parameters)

## Case 1 : No argument is passed

```
void temp(int = 10, float = 8.8);

int main() {
    ... ..
    temp();
    ... ..
}

void temp(int i, float f) {
    // code
}
```




## Case 2 : First argument is passed

```
void temp(int = 10, float = 8.8);

int main() {
    ... ..
    temp(6);
    ... ..
}

void temp(int i, float f) {
    // code
}
```




## Case 3 : All arguments are passed

```
void temp(int = 10, float = 8.8);

int main() {
    ... ..
    temp(6, -2.3);
    ... ..
}

void temp(int i, float f) {
    // code
}
```




## Case 4 : Second argument is passed

```
void temp(int = 10, float = 8.8);

int main() {
    ... ..
    temp(3.4);
    ... ..
}

void temp(int i, float f) {
    // code
}
```



# Default Arguments (Parameters)

We can understand the working of default arguments from the image above:

1. When `temp()` is called, both the default parameters are used by the function.
2. When `temp(6)` is called, the first argument becomes 6 while the default value is used for the second parameter.
3. When `temp(6, -2.3)` is called, both the default parameters are overridden, resulting in `i = 6` and `f = -2.3`.
4. When `temp(3.4)` is passed, the function behaves in an undesired way because the **second argument cannot be passed without passing the first argument**.

**Therefore, 3.4 is passed as the first argument. Since the first argument has been defined as int, the value that is actually passed is 3.**

# Default Arguments (Parameters)

- `#include <iostream>`
- `using namespace std;`
- `int sum(int a, int b=10, int c=20);`
- `int main(){`
- `/* In this case a value is passed as`
- `* 1 and b and c values are taken from`
- `* default arguments.`
- `*/`
- `cout<<sum(1)<<endl;`
- `/* In this case a value is passed as`
- `* 1 and b value as 2, value of c values is`
- `* taken from default arguments.`
- `*/`
- `cout<<sum(1, 2)<<endl;`
- `/* In this case all the three values are`
- `* passed during function call, hence no`
- `* default arguments have been used.`
- `*/`
- `cout<<sum(1, 2, 3)<<endl;`
- `return 0;`
- `}`

```
int sum(int a, int b, int c)
{
    int z;
    z = a+b+c;
    return z;
}
```

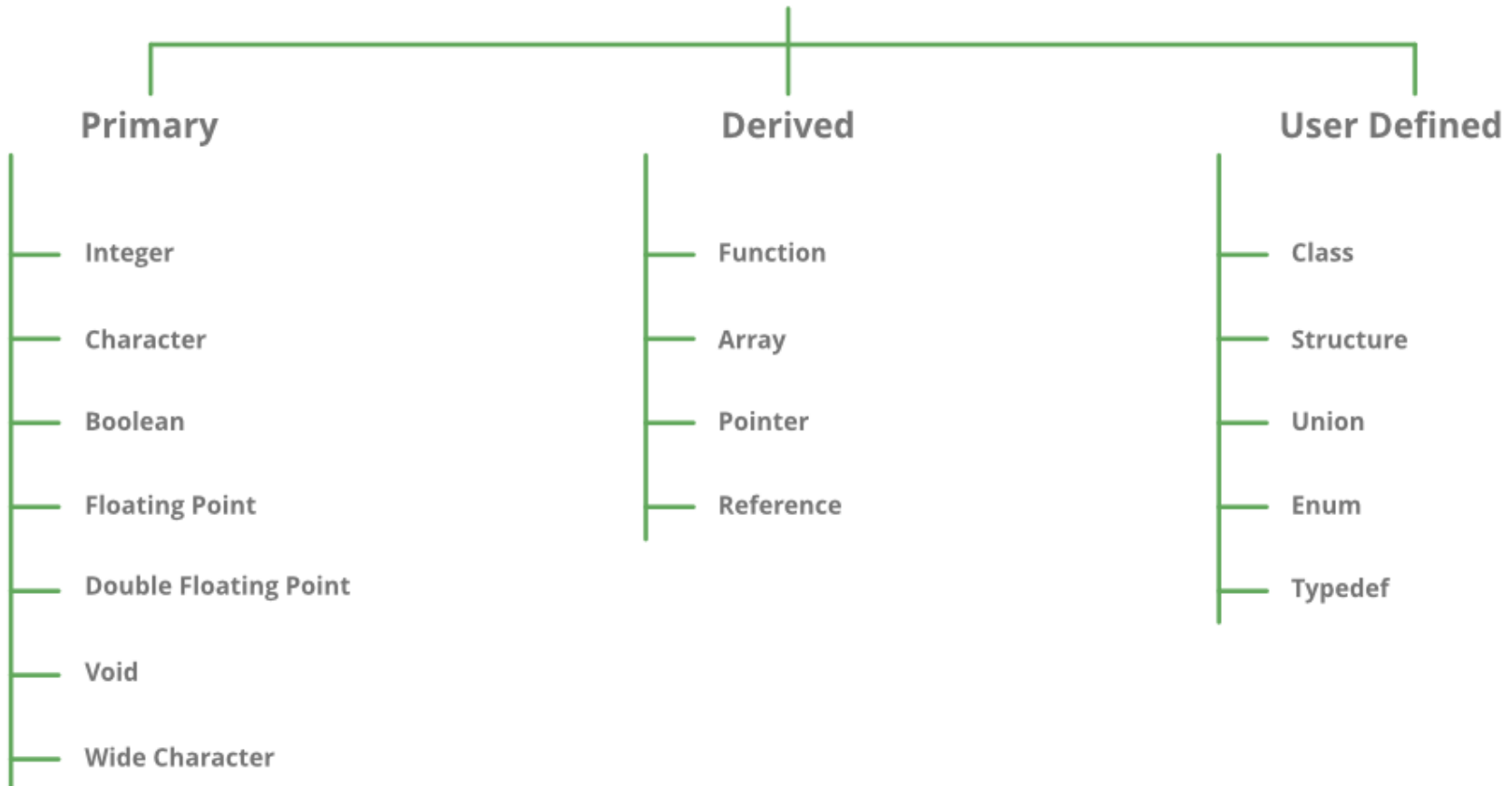
Output:

31  
23  
6



- All variables use data-type during declaration to restrict the type of data to be stored. Therefore, we can say that data types are used to tell the variables **the type of data it can store**.
- Whenever a variable is defined in **C++**, the **compiler allocates some memory for that variable based on the data-type with which it is declared**.
- **Every data type requires a different amount of memory**.

## DataTypes in C / C++



**Primitive Data Types:** These data types are built-in or predefined data types and **can be used directly by the user to declare variables**. example: int, char , float, bool etc. Primitive data types available in C++ are:

- Integer
- Character
- Boolean
- Floating Point
- Double Floating Point
- Valueless or Void
- Wide Character

**Derived Data Types:** The data-types that are **derived from the primitive or built-in datatypes** are referred to as Derived Data Types. These can be of four types namely:

- Function
- Array
- Pointer
- Reference

- **Abstract Data Types (ADT) or User-Defined Data Types:** These data types are defined by user itself. Like, defining a class in C++ or a structure. C++ provides the following user-defined datatypes:
  - Class
  - Structure
  - Union

```
#include <iostream>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    cout << "Size of char: " << sizeof(char) << " byte" << endl;
```

```
    cout << "Size of int: " << sizeof(int) << " bytes" << endl;
```

```
    cout << "Size of float: " << sizeof(float) << " bytes" << endl;
```

```
    cout << "Size of double: " << sizeof(double) << " bytes" << endl;
```

```
    return 0;
```

```
}
```

Output

Size of char: 1 byte

Size of int: 4 bytes

Size of float: 4 bytes

Size of double: 8 bytes

# C++ garbage collection

- In computer science, **garbage collection (GC)** is a form of **automatic memory management**. The garbage collector, or just collector, attempts to reclaim garbage, or memory occupied by objects that are no longer in use by the program.
- **C++** doesn't need a **garbage collector**, because it has no **garbage**. In modern **C++** you use smart pointers and therefore have no **garbage**.
- **Smart pointers** are objects which store **pointers** to dynamically allocated (heap) objects. They behave much like built-in **C++ pointers** except that they automatically delete the object pointed to at the appropriate time. ... They can also be used to keep track of dynamically allocated objects shared by multiple owners.

# C++ garbage collection

- In the **manual memory management** method, the user is required to mention the memory which is in use and which can be deallocated, whereas the **garbage collector** collects the memory which is occupied by variables or objects which are no more in use in the program.
- Only memory will be managed by garbage collectors, other resources such as destructors, user interaction window or files will not be handled by the garbage collector.



# C++ garbage collection

- Few languages need garbage collectors as part of the language for good efficiency. These languages are called as **garbage-collected languages**. For **example, Java, C#** and most of the scripting languages needs garbage collection as part of their functioning.
- Whereas languages such as **C and C++ support manual memory management** which works similar to the garbage collector.

## Manual Memory Management

- Dynamically allocated memory during run time from the heap needs to be released once we stop using that memory. Dynamically allocated memory takes memory from the heap, which is a free store of memory.
- **In C++ this memory allocation and deallocation are done manually using commands like new, delete.** Using 'new' memory is allocated from the heap. After its usage, this memory must be cleared using the 'delete' command.

```
n = new sample_object;
```

```
***** usage is implemented here*****
```

```
delete n;
```

- Advantages of **manual memory management** are that the **user would have complete control over both allocating and deallocating** operations and also know when a new memory is allocated and when it is deallocated or released.
- Also in the case of manual memory management, the **destructor will be called** at the same moment **when we call the 'delete' command**. But in case of **garbage collector that is not implemented**.

# C++ garbage collection

- **Garbage collection** is a form of **automatic memory management**. The garbage collector attempts to reclaim garbage, or memory used by objects that will never be accessed or changed again by the application.
- Tracing garbage collectors require some implicit **runtime overhead** that may be beyond the control of the programmer, and can sometimes lead to **performance problems**.
- **Java** applications obtain objects in memory as needed. It is the task of **garbage collection (GC)** in the **Java** virtual machine (**JVM**) to automatically determine what memory is no longer being used by a **Java** application and to recycle this memory for other uses.

# C++ garbage collection

- The biggest **benefit** of Java **garbage collection** is that it automatically handles the deletion of unused objects or **some** objects that are inaccessible to free up memory resources. **Garbage Collection** is now a new standard component of many popular programming languages. It makes Java memory-efficient.
- The main difference is the Garbage Collection. **Java runs all the memory by the Garbage Collector. C++ developers should keep watch of their memory management.** A complex C++ application can cause computer memory shortage. In a word, it's not easy to implement memory management in C++, as a result, there're constant issues with memory leaks there. **Garbage collection isn't part of C++ language specification. This was done to avoid memory overhead.**

1. There are two ways that memory gets allocated for data storage:

## Compile Time (or static) Allocation

1. Memory for named variables is allocated by the compiler
2. Exact size and type of storage must be known at compile time
3. For standard array declarations, this is why the size has to be constant

## 2. Dynamic Memory Allocation

1. Memory allocated "on the fly" during run time
2. dynamically allocated space usually placed in a program segment known as the *heap* or the *free store*
3. Exact amount of space or number of items does not have to be known by the compiler in advance.
4. For dynamic memory allocation, pointers are crucial

## **Dynamic allocation requires two steps:**

1. Creating the dynamic space.
  2. Storing its address in a pointer (so that the space can be accessed)
- **To dynamically allocate memory in C++, we use the new operator.**

## **De-allocation:**

- Deallocation is the "clean-up" of space being used for variables or other data storage
- Compile time variables are automatically deallocated based on their known extent (this is the same as scope for "automatic" variables)
- It is the programmer's job to deallocate dynamically created space
- To de-allocate dynamic memory, we use the delete operator

## Allocating space with new

- To allocate space dynamically, use the unary operator new, followed by the type being allocated.

```
new int;    // dynamically allocates an int
```

```
new double; // dynamically allocates a double
```

## If creating an array dynamically, use the same form, but put brackets with a size after the type:

```
new int[40]; // dynamically allocates an array of 40 ints
```

```
new double[size]; // dynamically allocates an array of size doubles
```

```
// note that the size can be a variable
```

**These statements above are not very useful by themselves, because the allocated spaces have no names! BUT, the new operator returns the starting address of the allocated space, and this address can be stored in a pointer:**

```
int * p;    // declare a pointer p
```

```
p = new int; // dynamically allocate an int and load address into p
```

```
double * d; // declare a pointer d
```

```
d = new double; // dynamically allocate a double and load address into d
```

```
// we can also do these in single line statements
```

```
int x = 40;
```

```
int * list = new int[x];
```

```
float * numbers = new float[x+10];
```

Notice that this is one more way of initializing a pointer to a valid target (and the most important one).



## Accessing dynamically created space

- So once the space has been dynamically allocated, how do we use it?
- For single items, we go through the pointer. Dereference the pointer to reach the dynamically created target:

```
int * p = new int;    // dynamic integer, pointed to by p
```

```
*p = 10;              // assigns 10 to the dynamic integer
```

```
cout << *p;           // prints 10
```

**For dynamically created arrays, you can use either pointer-offset notation, or treat the pointer as the array name and use the standard bracket notation:**

```
double * numList = new double[size];    // dynamic array
```

```
for (int i = 0; i < size; i++)
```

```
    numList[i] = 0;                      // initialize array elements to 0
```

```
numList[5] = 20;                        // bracket notation
```

```
*(numList + 7) = 15;                    // pointer-offset notation
```

```
// means same as numList[7]
```

## Deallocation of dynamic memory

To deallocate memory that was created with `new`, we use the unary operator `delete`. The one operand should be a pointer that stores the address of the space to be deallocated:

```
int * ptr = new int;           // dynamically created int
// ...
delete ptr;                   // deletes the space that ptr points to
```

Note that the pointer `ptr` still exists in this example. That's a named variable subject to scope and extent determined at compile time. It can be reused:

```
ptr = new int[10];            // point p to a brand new array
```

**To deallocate a dynamic array, use this form:**

```
delete [] name_of_pointer;
```

Example:

```
int * list = new int[40];      // dynamic array

delete [] list;               // deallocates the array
list = 0;                    // reset list to null pointer
```

After deallocating space, it's always a good idea to reset the pointer to null unless you are pointing it at another valid target right away.

- Sometimes implementation of all function cannot be provided in a base class because we don't know the implementation. Such a class is called abstract class. For example, let **Shape** be a base class. We cannot provide implementation of function **draw()** in Shape, but we know every derived class must have implementation of draw().
- **We cannot create objects of abstract classes.**
- ***A class is abstract if it has at least one pure virtual function.***  
Abstract class can have normal functions and variables along with a pure virtual function.
- A pure virtual function is implemented by classes which are derived from a Abstract class.

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

class Base
{
    int x;
public:
    virtual void fun() = 0;
    int getX() { return x; }
};

// This class inherits from Base and implements fun()
class Derived: public Base
{
    int y;
public:
    void fun()
    {
        cout << "fun() called";
    }
};

int main(void)
{
    Derived d;
    d.fun();
    return 0;
}
```

Output:

fun() called

1. How can we call member functions from outside the class?
2. Point out the reasons why using new is a better idea than malloc()?
3. What is the difference between the following two statements if a is pointer to an array allocated dynamically?  
delete a;  
delete [ ]a;
4. How can we initialize a const data members?

5. What is an anonymous class?
6. What is dangling pointer? Give example.
7. Is it necessary to accept a reference in the copy constructor?
8. Can a non-static member function access the static data?
9. Can we use the renew operator in C++ to reallocate memory?

1. Design a class from which we can create objects by passing one, two or three arguments. The class should not have more than constructor function.
2. How can we initialize an array of objects?
3. Can we initiate an object `s` of class `sample` through a statement like `sample s = 10; ?`
4. Write a snippet to show memory leak in C++?
5. What is the size of an object of an empty class? And Why?
6. Does the delete operator call the destructor of the class?

7. How can one return an error value from the constructor?
8. The this pointer always contain the address of the object using which the member function is being called. Illustrate the concept with the help of an object.
9. Can we modify the this pointer?
10. Discuss the various situations when a copy constructor is automatically invoked.



1. Write a program using a class to store marks list of 15 students and to print the highest marks as well as the average of all marks.
2. Illustrate the scope rules of global class, global object, local class and local object through some code fragment in C++.
3. How nesting of member functions work in a class? Discuss.
4. When declaration of static member and static functions become necessary ? Give example.

5. What should we keep in mind before the objects of an inner class can be declared? Show the validity of your remark in the perspective of a Nested class program.
6. How the working of inline function is different from other functions? Is it also different from #define value replacement directive or macro concept?
7. Define a situation that requires a function to operate on objects of two different classes. Write a program to signify the friend function as bridge.

8. Write a program to maintain a bank account class where you can pass objects as parameters to functions.
9. Write the syntax for creating nameless objects and how they will be destroyed in a program?
10. Create a vector class where array can be dynamically allocated?

## **TEXT:**

1. A..R.Venugopal, Rajkumar, T. Ravishanker “Mastering C++”, TMH, 2009.
2. S. B. Lippman & J. Lajoie, “C++ Primer”, 6th Edition, Addison Wesley, 2006.

## **REFERENCE:**

1. R. Lafore, “Object Oriented Programming using C++”, Galgotia Publications, 2008.
2. D . Parsons, “Object Oriented Programming with C++”, BPB Publication.
3. Steven C. Lawlor, “The Art of Programming Computer Science with C++”, Vikas Publication.
4. Schildt Herbert, “C++: The Complete Reference”, 7th Ed., Tata McGraw Hill, 2008.