UNIT 3:

Classes and Objects

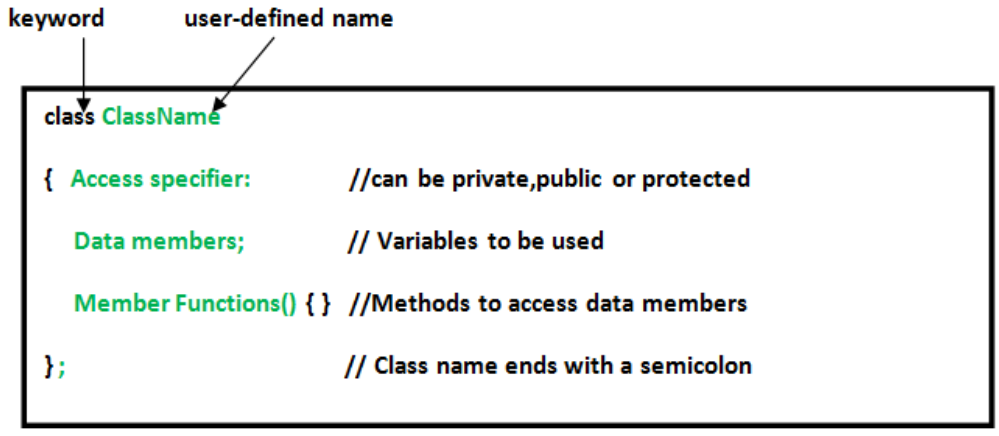
Class in C++ is the building block that leads to Object-Oriented programming. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A C++ class is like a blueprint for an object. For Example: Consider the Class of **Cars**. There may be many cars with different names and brands but all of them will share some common properties like all of them will have *4 wheels*, *Speed Limit*, *Mileage range,* etc. So here, Car is the class, and wheels, speed limits, and mileage are their properties.

- A Class is a user-defined data type that has data members and member functions.
- Data members are the data variables and member functions are the functions used to manipulate these variables together, these data members and member functions define 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 *applying brakes*, *increasing 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.

Defining Class and Declaring Objects

A class is defined in C++ using the keyword class followed by the name of the class. The body of the class is defined inside the curly brackets and terminated by a semicolon at the end.



Declaring Objects

When a class is defined, only the specification for the object is defined; no memory or storage is allocated. To use the data and access functions defined in the class, you need to create objects.

Syntax

ClassName ObjectName;

Accessing data members and member functions: The data members and member functions of the class can be accessed using the dot('.') operator with the object. For example, if the name of the object is *obj* and you want to access the member function with the name *printName()* then you will have to write *obj.printName()*.

Accessing Data Members

The public data members are also accessed in the same way given however the private data members are not allowed to be accessed directly by the object. Accessing a data member depends solely on the access control of that data member. This access control is given by Access modifiers in C++. There are three access modifiers: **public, private, and protected**.

C++ date and time

In this article, we will learn about the date and time formats in C++. There is no complete format in C++ for date and time so we inherit it from the c language. To use date and time in c++, **<ctime>** header file is added in the program.

<ctime>

This header file has four time-related types as follows -

- **Clock_t -** It stands for clock type which is an alias of arithmetic type. It represents clock tick counts(units of a time of a constant with system-specific length). Clock_t is the type returned by **clock()/.**
- **Time_t -** It stands for time_type. It represents the time that is returned by function **time().** It outputs an integral value as the number of seconds elapsed when time passes by 00:00 hours.
- **Size_t -** It is an alias for an unsigned integer type and represents the size of any object in bytes. Size_t is the result of the **sizeof()** operator which prints sizes and counts.
- **tm -** The tm structure holds date and time in the C structure. It is defined as follows -

```
struct tm {
    int tm_sec; // seconds of minutes from 0 to 61
    int tm_min; // minutes of hour from 0 to 59
    int tm_hour; // hours of day from 0 to 24
    int tm_mday; // day of month from 1 to 31
    int tm_mon; // month of year from 0 to 11
    int tm_year; // year since 1900
    int tm_wday; // days since sunday
    int tm_yday; // days since January 1st
    int tm_isdst; // hours of daylight savings time
}
```

Date and time functions in c++

Name of the function	Prototype of the function	Description About the function
mktime	time_t mktime(struct tm *time);	This function converts mktime to time_t or calendar date and time.
ctime	char *ctime(const time_t *time);	It returns the pointer to a string of the format - day month year hours: minutes: seconds year.
difftime	double difftime (time_t time2, time_t time1);	It returns the difference of two-time objects t1 and t2.
gmtime	struct tm *gmtime(const time_t *time);	This function returns the pointer of the time in the format of a structure. The time is in UTC.
clock	clock_t clock(void);	It returns an approximated value for the amount of time the calling program is being run. The value .1 is returned if not available.
localtime	struct tm *localtime(const time_t *time);	This function returns the pointer to the tm structure representing local time.

time	time_t time(time_t *time);	It represents current time.
strftime	size_t strftime();	With the help of this function, we can format date and time in a specific manner.
asctime	char * asctime (const struct tm * time);	The function converts the type object of tm to string and returns the pointer to that string.

Example to print current date and time

Below is the example to print the current date and time in the UTC format.

Code

```
#include <ctime>
#include <iostream>


using namespace std;

int main()
{

   time_t now = time(0); // get current dat/time with respect to system

   char* dt = ctime(&now); // convert it into string

   cout << "The local date and time is: " << dt << endl; // print local date and time

   tm* gmtm = gmtime(&now); // for getting time to UTC convert to struct
   dt = asctime(gmtm);
   cout << "The UTC date and time is:" << dt << endl; // print UTC date and time
}
```

Output

```
The local date and time is: Wed Sep 22 16:31:40 2021

The UTC date and time is: Wed Sep 22 16:31:40 2021
```

Objects as Function Arguments

The objects of a class can be passed as arguments to member functions as well as nonmember functions either by value or by reference. When an object is passed by value, a copy of the actual object is created inside the function. This copy is destroyed when the function terminates. Moreover, any changes made to the copy of the object inside the function are not reflected in the actual object. On the other hand, in pass by reference, only a reference to that object (not the entire object) is passed to the function. Thus, the changes made to the object within the function are also reflected in the actual object.

Whenever an object of a class is passed to a member function of the same class, its data members can be accessed inside the function using the object name and the dot operator. However, the data members of the calling object can be directly accessed inside the function without using the object name and the dot operator.

To understand how objects are passed and accessed within a member function, consider this example.

Example: A program to demonstrate passing objects by value to a member function of the same class

```
1   #include<iostream.h>
2   class weight {
3     int kilogram;
4     int gram;
5     public:
6       void getdata ();
7       void putdata ();
8       void sum_weight (weight,weight) ;
9   } ;
10  void weight :: getdata() {
11     cout<<"/nKilograms:";
12     cin>>kilogram;
13     cout<<"Grams:";
14     cin>>gram;
15  }
16  void weight :: putdata () {
17    cout<<kilogram<<" Kgs. and"<<gram<<" gros.\n";
18  }
19  void weight :: sum_weight(weight w1,weight w2) {
    gram = w1.gram + w2.gram;
    kilogram=gram/1000;
    gram=gram%1000;
    kilogram+=w1.kilogram+w2.kilogram;
```

```
}
int main () {
 weight w1,w2 ,w3;
   cout<<"Enter weight in kilograms and grams\n";
   cout<<"\n Enter weight #1" ;
   w1.getdata();
   cout<<" \n Enter weight #2" ;
   w2.getdata();
   w3.sum_weight(wl,w2);
   cout<<"/n Weight #1 = ";
   w1.putdata();
   cout<<"Weight #2 = ";
   w2.putdata();
   cout<<"Total Weight = ";
   w3.putdata();
   return 0;
}
```

The output of the program is

Enter weight in kilograms and grams

Enter weight #1

Kilograms: 12

Grams: 560

Enter weight #2

Kilograms: 24

Grams: 850

Weight #1 = 12 Kgs. and 560 gms.

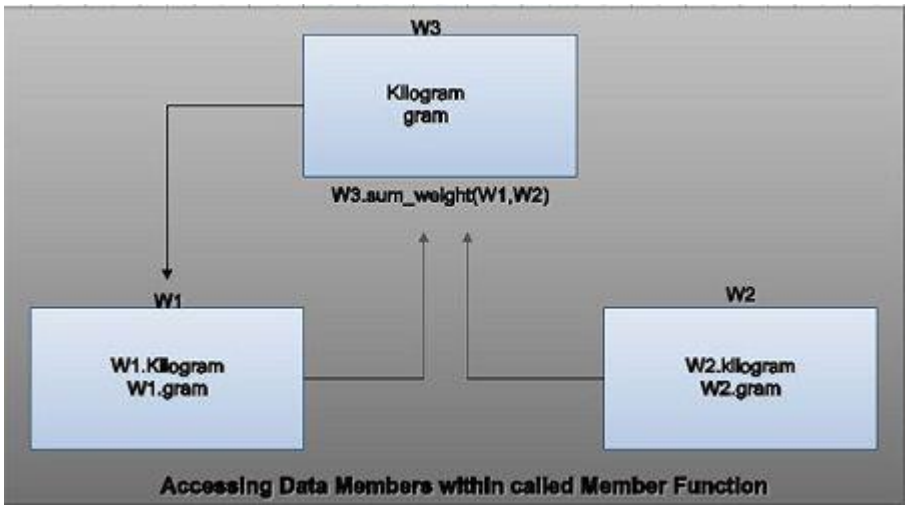
Weight #2 = 24 Kgs. and 850 gms.

Total Weight = 37 Kgs. and 410 gms.

In this example, the sum_weight () function has direct access to the data members of calling object (w3 in this case). However, the members of the objects passed as arguments (w1 and w2) can be accessed within function using the object name and dot operator. Note that, the objects w1 and w2 are passed by value, however, they can re passed by reference also. For example, to pass w1 and w2 by reference to the function sum_weight the function will be declared and defined as

```
1    void sum_weight (weight &,weight &) ;
2    void weight :: sum_weight (weight & w1, weight & w2) {
3      // body of function
4    }
```

Figure shows accessing of member variables inside the sum_weight function;



Array of Objects in C++ with Examples

An array in C/C++ or be it in any programming language is a collection of similar data items stored at contiguous memory locations and elements can be accessed randomly using indices of an array. They can be used to store the collection of primitive data types such as int, float, double, char, etc of any particular type. To add to it, an array in C/C++ can store derived data types such as structures, pointers, etc. Given below is the picture representation of an array.

Example:

Let's consider an example of taking random integers from the user.

34	67	78	32	78	98	89

0	1	2	3	4	5	6

←Array Indices →

Array

Array of Objects

When a class is defined, only the specification for the object is defined; no memory or storage is allocated. To use the data and access functions defined in the class, you need to create objects.

Syntax:

ClassName ObjectName[number of objects];

The Array of Objects stores *objects*. An array of a class type is also known as an array of objects.

Example#1:

Storing more than one Employee data. Let's assume there is an array of objects for storing employee data emp[50].

Objects	Employee Id	Employee name
emp[0]→		
emp[1]→		
emp[2]→		
emp[3]→		

Below is the C++ program for storing data of one Employee:

- C++

```
// C++ program to implement
```

```
// the above approach

#include<iostream>

using namespace std;



class Employee

{

  int id;

  char name[30];

  public:

  void getdata();//Declaration of function

  void putdata();//Declaration of function

};

void Employee::getdata(){//Defining of function

  cout<<"Enter Id : ";

  cin>>id;

  cout<<"Enter Name : ";

  cin>>name;

}

void Employee::putdata(){//Defining of function

  cout<<id<<" ";
```

```
  cout<<name<<" ";

  cout<<endl;

}

int main(){

  Employee emp; //One member

  emp.getdata();//Accessing the function

  emp.putdata();//Accessing the function

  return 0;



}
```

Let's understand the above example –

- In the above example, a class named Employee with id and name is being considered.
- The two functions are declared-
    - **getdata():** Taking user input for id and name.
    - **putdata():** Showing the data on the console screen.

This program can take the data of only one Employee. What if there is a requirement to add data of more than one Employee. Here comes the answer Array of Objects. An array of objects can be used if there is a need to store data of more than one employee. Below is the C++ program to implement the above approach-

- C++

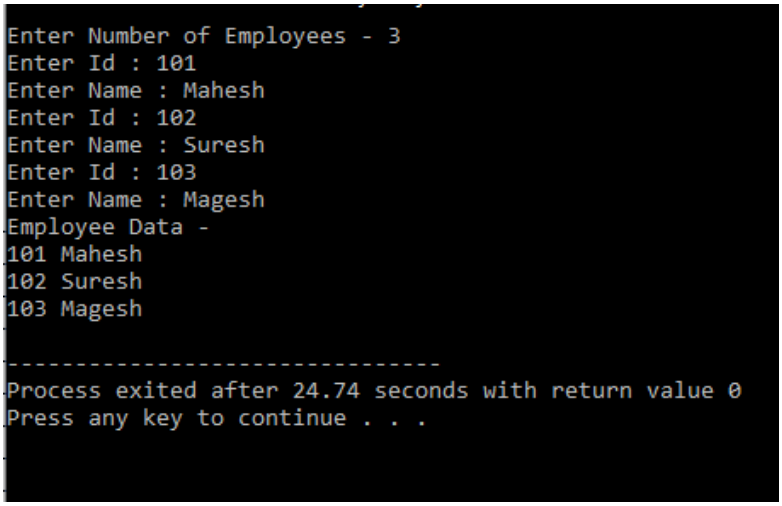
```
// C++ program to implement

// the above approach
```

```
#include<iostream>

using namespace std;



class Employee

{

 int id;

 char name[30];

 public:



 // Declaration of function

 void getdata();



 // Declaration of function

 void putdata();

};



// Defining the function outside

// the class

void Employee::getdata()
```

```
{
  cout << "Enter Id : ";
  cin >> id;
  cout << "Enter Name : ";
  cin >> name;
}

// Defining the function outside
// the class
void Employee::putdata()
{
  cout << id << " ";
  cout << name << " ";
  cout << endl;
}

// Driver code
int main()
{
  // This is an array of objects having
```

```
  // maximum limit of 30 Employees

  Employee emp[30];

  int n, i;

  cout << "Enter Number of Employees - ";

  cin >> n;



  // Accessing the function

  for(i = 0; i < n; i++)

    emp[i].getdata();



  cout << "Employee Data - " << endl;



  // Accessing the function

  for(i = 0; i < n; i++)

    emp[i].putdata();

}
```

Output:



Explanation:

In this example, more than one Employee's details with an Employee id and name can be stored.

- Employee emp[30] – This is an array of objects having a maximum limit of 30 Employees.
- Two for loops are being used-
    - First one to take the input from user by calling emp[i].getdata() function.
    - Second one to print the data of Employee by calling the function emp[i].putdata() function.

Returning object from function

A function can also return objects either by value or by reference. When an object is returned by value from a function, a temporary object is created within the function, which holds the return value. This value is further assigned to another object in the calling function.

The syntax for defining a function that returns an object by value is

```
1    class_name function_name (parameter_list) {
2     // body of the function
3    }
```

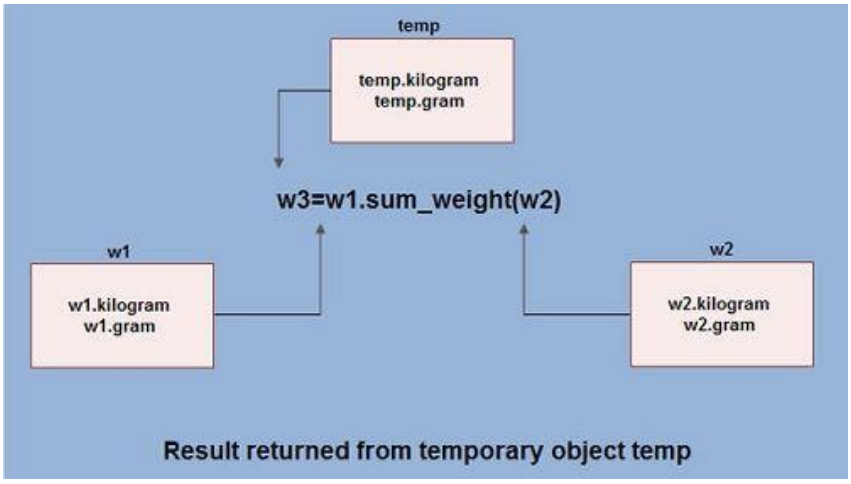
To understand the concept of returning an object by value from a function, consider this example.

Example: A program to demonstrate the concept of returning objects from a function

```
1
2   #include<iostream.h>
3   class weight {
4     int kilogram;
5     int gram;
6     public:
7       void getdata ();
8       void putdata ();
9       void sum_weight (weight,weight) ;
10      weight sum_weight (weight) ;
11  } ;
12  void weight :: getdata() {
13    cout<<"/nKilograms:";
14    cin>>kilogram;
15    cout<<"Grams:";
16    cin>>gram;
17  }
18   void weight :: putdata () {
19     cout<<kilogram<<" Kgs. and"<<gram<<" gros.\n";
20   }
21   weight weight :: sum_weight(weight w2) {
22    weight temp;
23    temp.gram = gram + w2.gram;
24    temp.kilogram=temp.gram/1000;
25    temp.gram=temp.gram%1000;
26    temp.kilogram+=kilogram+w2.kilogram;
27    return(temp);
28  }
29  int main () {
30    weight w1,w2 ,w3;
31    cout<<"Enter weight in kilograms and grams\n";
32    cout<<"\n Enter weight #1" ;
33    w3 = w1.sum_weight (w2);
34    w1.getdata();
35    cout<<" \n Enter weight #2" ;
36    w2.getdata();
37    w3.sum_weight(wl,w2);
38    cout<<"/n Weight #1 = ";
    w1.putdata();
    cout<<"Weight #2 = ";
    w2.putdata();
    cout<<"Total Weight = ";
    w3.putdata();
    return 0;
  }
```

As the return type of function is weight (an object of class weight), a temporary object temp is created within the function for holding return values. These values are accessed as temp kilogram and temp gram by the function. When the control returns to main (), the object temp is assigned to the object w3 in main( ).

Figure represents accessing of member variables within the function and returning a result from the temporary object.



In the case of returning an object by reference, no new object is created, rather a reference to the original object in the called function is returned to the calling function.

The syntax for defining a function that returns an object by reference is

```
class_name&amp;amp; function_name (parameter_list) {
  //body of the function
}
```

However, a reference to the local object (object declared inside the function) cannot be returned from the function since a reference to local data points to data within the function. Hence, when the function terminates and local data is destroyed, this reference points to nothing.

```

weight& weight::sum_weight(weight& w2) {
  weight temp; //object local to function
  temp.gram=gram+w2.gram;
  temp.kilogram=temp.gram/1000;
  temp.gram=temp.gram%1000;
  temp.kilogram+=kilogram+w2.kilogram;
  return temp; //invalid
}
```

To understand this concept, consider the function sum_weight () that is modified as shown in this code segment..

In this code segment, an attempt has been made to return the reference to an object of type weight (that is temp). However, it is not possible as the object temp is local to the sum_weight()function and a reference to this object remains effective only within the function. Thus, returning the reference to temp object outside the function generates a compile-time error.

Difference between Structure and Class in C++

In C++, the structure is the same as the class with some differences. Security is the most important thing for both structure and class. A structure is not safe because it could not hide its implementation details from the end-user, whereas a class is secure as it may hide its programming and design details. In this article, we are going to discuss the difference between a structure and class in C++. But before discussing the differences, we will know about the structure and class in C++.

What is the structure in C++?

A structure is a **grouping** of variables of various **data types** referenced by the same name. A structure declaration serves as a template for creating an instance of the structure.

Syntax:

The syntax of the structure is as follows:

```
1. Struct Structurename
2. {
3. Struct_member1;
4. Struct_member2;
5. Struct_member3;
6. .
7. .
8. .
9. Struct_memberN;
10. };
```

The **"struct"** keyword indicates to the compiler that a structure has been declared.
The **"structurename"** defines the name of the structure. Since the structure declaration is treated as a statement, so it is often ended by a semicolon.

What is Class in C++?

A class in C++ is similar to a C structure in that it consists of a list of **data members** and a set of operations performed on the class. In other words, a class is the **building block** of Object-Oriented programming. It is a user-defined object type with its own set of data members and member functions that can be accessed and used by creating a class instance. A C++ class is similar to an object's blueprint.

Syntax:

The structure and the class are syntactically similar. The syntax of class in C++ is as follows:

```
1. class class_name
2. {
3. // private data members and member functions.
4. Access specifier;
5. Data member;
6. Member functions (member list){ . . }
7. };
```

In this syntax, the class is a keyword to indicate the compiler that a class has been declared. OOP's main function is data hiding, which is achieved by having three access specifiers: **"public", "private"**, and **"safe"**. If no access specifier is specified in the class when declaring data members or member functions, they are all considered private by default.

The public access specifier allows others to access program functions or data. A member of that class may reach only the class's private members. During inheritance, the safe access specifier is used. If the access specifier is declared, it cannot be changed again in the program.

Main differences between the structure and class

Here, we are going to discuss the main differences between the structure and class. Some of them are as follows:

- By default, all the members of the structure are public. In contrast, all members of the class are private.
- The structure will automatically initialize its members. In contrast, constructors and destructors are used to initialize the class members.

- When a structure is implemented, memory allocates on a stack. In contrast, memory is allocated on the heap in class.
- Variables in a structure cannot be initialized during the declaration, but they can be done in a class.
- There can be no null values in any structure member. On the other hand, the class variables may have null values.
- A structure is a value type, while a class is a reference type.
- Operators to work on the new data form can be described using a special method.

Head-to-head comparison between the structure and class

Here, we are going to discuss a head-to-head comparison between the structure and class. Some of them are as follows:

Features	Structure	Class
Definition	A structure is a grouping of variables of various data types referenced by the same name.	In C++, a class is defined as a collection of related variables and functions contained within a single structure.
Basic	If no access specifier is specified, all members are set to 'public'.	If no access specifier is defined, all members are set to 'private'.

Declaration	struct structure_name{
	type struct_member 1;
	type struct_member 2;
	type struct_member 3;
	.
	type struct_memberN;
	};
		class class_name{
		data member;
		member function;
		};

Instance	Structure instance is called the 'structure variable'.	A class instance is called 'object'.
Inheritance	It does not support inheritance.	It supports inheritance.

Memory Allocated	Memory is allocated on the stack.	Memory is allocated on the heap.
Nature	Value Type	Reference Type
Purpose	Grouping of data	Data abstraction and further inheritance.
Usage	It is used for smaller amounts of data.	It is used for a huge amount of data.
Null values	Not possible	It may have null values.
Requires constructor and destructor	It may have only parameterized constructor.	It may have all the types of constructors and destructors.

Similarities

The following are similarities between the structure and class:

- Both class and structure may declare any of their members private.
- Both class and structure support inheritance mechanisms.
- Both class and structure are syntactically identical in C++.
- A class's or structure's name may be used as a stand-alone type.

Conclusion

Structure in C has some limitations, such as the inability to hide data, the inability to treat 'struct' data as built-in types, and the lack of inheritance support. The C++ structure overcame these drawbacks.

The extended version of the structure in C++ is called a class. The programmer makes it easy to use the class to hold both the data and functions, whereas the structure only holds data.

Constructors in C++

Constructor in C++ is a special method that is invoked automatically at the time of object creation. It is used to initialize the data members of new objects generally. The constructor in C++ has the same name as the class or structure. Constructor is invoked

at the time of object creation. It constructs the values i.e. provides data for the object which is why it is known as constructors.

• Constructor is a member function of a class, whose name is same as the class name.

• Constructor is a special type of member function that is used to initialize the data members for an object of a class automatically, when an object of the same class is created.

• Constructor is invoked at the time of object creation. It constructs the values i.e. provides data for the object that is why it is known as constructor.

• Constructor do not return value, hence they do not have a return type.

The prototype of the constructor looks like

<class-name> (list-of-parameters);

Constructor can be defined inside the class declaration or outside the class declaration

a. Syntax for defining the constructor within the class

```
<class-name>(list-of-parameters)
{
        //constructor definition
}
```

b. Syntax for defining the constructor outside the class

```
<class-name>: :<class-name>(list-of-parameters)
{
      //constructor definition
}
```

Characteristics of constructor

• The name of the constructor is same as its class name.

• Constructors are mostly declared in the public section of the class though it can be declared in the private section of the class.

- Constructors do not return values; hence they do not have a return type.
- A constructor gets called automatically when we create the object of the class.
- Constructors can be overloaded.
- Constructor can not be declared virtual.

Types of constructor

- Default constructor
- Parameterized constructor
- Overloaded constructor
- Constructor with default value
- Copy constructor
- Inline constructor

Constructor does not have a return value, hence they do not have a return type.

The prototype of Constructors is as follows:

<class-name> (list-of-parameters);

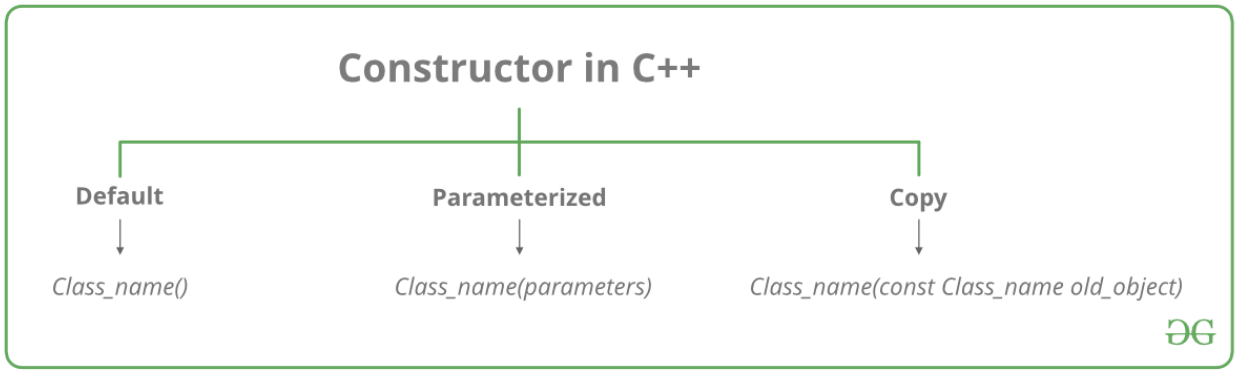
Constructors can be defined inside or outside the class declaration:-

The syntax for defining the constructor within the class:

<class-name> (list-of-parameters) { // constructor definition }

The syntax for defining the constructor outside the class:

<class-name>: :<class-name> (list-of-parameters){ // constructor definition}



1. Default Constructors: Default constructor is the constructor which doesn't take any argument. It has no parameters. It is also called a zero-argument constructor.

- CPP

```
// Cpp program to illustrate the
// concept of Constructors
#include <iostream>
using namespace std;

class construct {
public:
   int a, b;

   // Default Constructor
   construct()
   {
      a = 10;
      b = 20;
   }
};

int main()
```

```
{

   // Default constructor called automatically

   // when the object is created

   construct c;

   cout << "a: " << c.a << endl << "b: " << c.b;

   return 1;

}
```

Output

a: 10

b: 20

Note: *Even if we do not define any constructor explicitly, the compiler will automatically provide a default constructor implicitly.*

2. Parameterized Constructors: It is possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor's body, use the parameters to initialize the object.

Note: *when the parameterized constructor is defined and no default constructor is defined explicitly, the compiler will not implicitly call the default constructor and hence creating a simple object as*

Student s;

Will flash an error

- CPP

```
// CPP program to illustrate
```

```
// parameterized constructors

#include <iostream>

using namespace std;



class Point {

private:

   int x, y;



public:

   // Parameterized Constructor

   Point(int x1, int y1)

   {

      x = x1;

      y = y1;

   }



   int getX() { return x; }

   int getY() { return y; }

};
```

```
int main()

{

   // Constructor called

   Point p1(10, 15);



   // Access values assigned by constructor

   cout << "p1.x = " << p1.getX()

       << ", p1.y = " << p1.getY();



   return 0;

}
```

Output

p1.x = 10, p1.y = 15

When an object is declared in a parameterized constructor, the initial values have to be passed as arguments to the constructor function. The normal way of object declaration may not work. The constructors can be called explicitly or implicitly.

```
Example e = Example(0, 50); // Explicit call


Example e(0, 50);          // Implicit call
```

- **Uses of Parameterized constructor:**
  1. It is used to initialize the various data elements of different objects with different values when they are created.
  2. It is used to overload constructors.

- **Can we have more than one constructor in a class?**
  Yes, It is called Constructor Overloading.

3. Copy Constructor:

A copy constructor is a member function that initializes an object using another object of the same class. A detailed article on Copy Constructor.
Whenever we define one or more non-default constructors(with parameters) for a class, a default constructor(without parameters) should also be explicitly defined as the compiler will not provide a default constructor in this case. However, it is not necessary but it's considered to be the best practice to always define a default constructor.

Copy constructor takes a reference to an object of the same class as an argument.

```
Sample(Sample &t)
    {
        id=t.id;
    }
```

- CPP

```
// Illustration

#include <iostream>

using namespace std;



class point {

private:

   double x, y;



public:
```

```
    // Non-default Constructor &

    // default Constructor

    point(double px, double py) { x = px, y = py; }

};



int main(void)

{



    // Define an array of size

    // 10 & of type point

    // This line will cause error

    point a[10];



    // Remove above line and program

    // will compile without error

    point b = point(5, 6);

}
```

Output:

Error: point (double px, double py): expects 2 arguments, 0 provided

Default Arguments in C++

A default argument is a value provided in a function declaration that is automatically assigned by the compiler if the calling function doesn't provide a value for the argument. In case any value is passed, the default value is overridden.

1) The following is a simple C++ example to demonstrate the use of default arguments. Here, we don't have to write 3 sum functions; only one function works by using the default values for 3rd and 4th arguments.

- CPP

```
// CPP Program to demonstrate Default Arguments

#include <iostream>

using namespace std;



// A function with default arguments,

// it can be called with

// 2 arguments or 3 arguments or 4 arguments.

int sum(int x, int y, int z = 0, int w = 0) //assigning default values to z,w as 0

{

   return (x + y + z + w);

}
```

```
// Driver Code

int main()

{

   // Statement 1

   cout << sum(10, 15) << endl;



   // Statement 2

   cout << sum(10, 15, 25) << endl;



   // Statement 3

   cout << sum(10, 15, 25, 30) << endl;

   return 0;

}
```

Output

25

50

80

Time Complexity: O(1)
Space Complexity: O(1)
Explanation: In statement 1, only two values are passed, hence the variables z and w take the default values of 0. In statement 2, three values are passed, so the value of z is overridden with 25. In statement 3, four values are passed, so the values of z and w are overridden with 25 and 30 respectively.

2) If function overloading is done containing the default arguments, then we need to make sure it is not ambiguous to the compiler, otherwise it will throw an error. The following is the modified version of the above program:

- CPP

```
// CPP Program to demonstrate Function overloading in
// Default Arguments
#include <iostream>
using namespace std;


// A function with default arguments, it can be called with
// 2 arguments or 3 arguments or 4 arguments.
int sum(int x, int y, int z = 0, int w = 0)
{
   return (x + y + z + w);
}
int sum(int x, int y, float z = 0, float w = 0)
{
   return (x + y + z + w);
}
// Driver Code
```

```
int main()

{

   cout << sum(10, 15) << endl;

   cout << sum(10, 15, 25) << endl;

   cout << sum(10, 15, 25, 30) << endl;

   return 0;

}
```

Error:

```
prog.cpp: In function 'int main()':

prog.cpp:17:20: error: call of overloaded

'sum(int, int)' is ambiguous

  cout << sum(10, 15) << endl;

                    ^

prog.cpp:6:5: note: candidate:

int sum(int, int, int, int)

 int sum(int x, int y, int z=0, int w=0)

     ^

prog.cpp:10:5: note: candidate:

int sum(int, int, float, float)

 int sum(int x, int y, float z=0, float w=0)

     ^
```

3) A constructor can contain default parameters as well. A default constructor can either have no parameters or parameters with default arguments.

- C++

```
// CPP code to demonstrate use of default arguments in
// Constructors

#include <iostream>
using namespace std;

class A {
   private:
      int var = 0;
   public:
      A(int x = 0): var(x){}; // default constructor with one argument
                        // Note that var(x) is the syntax in c++ to do : "var = x"
      void setVar(int s){
         var = s; // OR => this->var = s;
         return;
      }
      int getVar(){
         return var; // OR => return this->var;
      }
```

```
};

int main(){

   A a(1);



   a.setVar(2);



   cout << "var = " << a.getVar() << endl;



   /* ANOTHER APPROACH:

   A *a = new A(1);



   a->setVar(2);



   cout << "var = " << a->getVar() << endl;

   */

}



// contributed by Francisco Vargas #pt
```

Explanation: Here, we see a default constructor with no arguments and a default constructor with one default argument. The default constructor with argument has a default parameter x, which has been assigned a value of 0.

Key Points:

- Default arguments are different from constant arguments as constant arguments can't be changed whereas default arguments can be overwritten if required.
- Default arguments are overwritten when the calling function provides values for them. For example, calling the function sum(10, 15, 25, 30) overwrites the values of z and w to 25 and 30 respectively.
- When a function is called, the arguments are copied from the calling function to the called function in the order left to right. Therefore, sum(10, 15, 25) will assign 10, 15, and 25 to x, y, and z respectively, which means that only the default value of w is used.
- Once a default value is used for an argument in the function definition, all subsequent arguments to it must have a default value as well. It can also be stated that the default arguments are assigned from right to left. For example, the following function definition is invalid as the subsequent argument of the default variable z is not default.

```
// Invalid because z has default value, but w after it doesn't have a default value

int sum(int x, int y, int z = 0, int w).
```

Advantages of Default Arguments:

- Default arguments are useful when we want to increase the capabilities of an existing function as we can do it just by adding another default argument to the function.
- It helps in reducing the size of a program.
- It provides a simple and effective programming approach.
- Default arguments improve the consistency of a program.

Disadvantages of Default Arguments:

- It increases the execution time as the compiler needs to replace the omitted arguments by their default values in the function call.

Dynamic initialization of object in C++

In this article, we will discuss the Dynamic initialization of objects using Dynamic Constructors.

- 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.

Dynamic Constructor:

- The constructor used for allocating the memory at runtime is known as the **dynamic constructor**.
- The memory is allocated at runtime using a new operator and similarly, memory is deallocated at runtime using the delete operator.

Dynamic Allocation:

Approach:

1. In the below example, **new** is used to dynamically initialize the variable in default constructor and memory is allocated on the heap.
2. The objects of the class geek calls the function and it displays the value of dynamically allocated variable i.e ptr.

Below is the program for dynamic initialization of object using **new** operator:

- C++

```
// C++ program for dynamic allocation

#include <iostream>

using namespace std;



class geeks {

   int* ptr;



public:

   // Default constructor

   geeks()

   {

      // Dynamically initializing ptr
```

```
    // using new
    ptr = new int;
    *ptr = 10;
  }

  // Function to display the value
  // of ptr
  void display()
  {
    cout << *ptr << endl;
  }
};

// Driver Code
int main()
{
  geeks obj1;

  // Function Call
```

```
    obj1.display();



    return 0;

}
```

Output
10

Dynamic Deallocation:
Approach:

1. In the below code, **delete** is used to dynamically free the memory.
2. The contents of obj1 are overwritten in the object obj2 using assignment operator, then obj1 is deallocated by using **delete** operator.

Below is the code for dynamic deallocation of the memory using **delete** operator.

- C++

```
// C++ program to dynamically

// deallocating the memory

#include <iostream>

using namespace std;



class geeks {

    int* ptr;



public:
```

```
    // Default constructor
    geeks()
    {
        ptr = new int;
        *ptr = 10;
    }

    // Function to display the value
    void display()
    {
        cout << "Value: " << *ptr
             << endl;
    }
};

// Driver Code
int main()
{
    // Dynamically allocating memory
    // using new operator
```

```
    geeks* obj1 = new geeks();

    geeks* obj2 = new geeks();



    // Assigning obj1 to obj2

    obj2 = obj1;



    // Function Call

    obj1->display();

    obj2->display();



    // Dynamically deleting the memory

    // allocated to obj1

    delete obj1;



    return 0;

}
```

Output

Value: 10

Value: 10

Below C++ program is demonstrating dynamic initialization of objects and calculating bank deposit:

- C++

```
// C++ program to illustrate the dynamic

// initialization as memory is allocated

// to the object

#include <iostream>

using namespace std;



class bank {

    int principal;

    int years;

    float interest;

    float returnvalue;



public:

    // Default constructor

    bank() {}



    // Parameterized constructor to

    // calculate interest(float)
```

```
bank(int p, int y, float i)

{

   principal = p;

   years = y;

   interest = i/100;

   returnvalue = principal;

   cout << "\nDeposited amount (float):";



   // Finding the interest amount

   for (int i = 0; i < years; i++) {

      returnvalue = returnvalue

                  * (1 + interest);

   }

}



// Parameterized constructor to

// calculate interest(integer)

bank(int p, int y, int i)

{

   principal = p;
```

```
    years = y;
    interest = float(i)/100;
    returnvalue = principal;
    cout << "\nDeposited amount"
        << " (integer):";

    // Find the interest amount
    for (int i = 0; i < years; i++) {
       returnvalue = returnvalue
                * (1 + interest);
    }
  }

  // Display function
  void display(void)
  {
    cout << returnvalue
        << endl;
  }
};
```

```
// Driver Code
int main()
{
    // Variable initialization
    int p = 200;
    int y = 2;
    int I = 5;
    float i = 2.25;

    // Object is created with
    // float parameters
    bank b1(p, y, i);

    // Function Call with object
    // of class
    b1.display();

    // Object is created with
```

```
    // integer parameters
    bank b2(p, y, I);



    // Function Call with object
    // of class
    b2.display();



    return 0;
}
```

Output:
Deposited amount (float):209.101

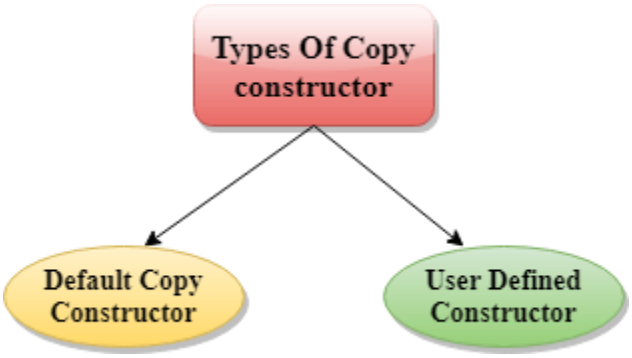
Deposited amount (integer):220.5

Copy Constructor

A Copy constructor is an **overloaded** constructor used to declare and initialize an object from another object.

Copy Constructor is of two types:

- **Default Copy constructor:** The compiler defines the default copy constructor. If the user defines no copy constructor, compiler supplies its constructor.
- **User Defined constructor:** The programmer defines the user-defined constructor.



Syntax Of User-defined Copy Constructor:

1. Class_name(**const** class_name &old_object);

Consider the following situation:

```
1. class A
2. {
3.    A(A &x) //  copy constructor.
4.    {
5.       // copyconstructor.
6.    }
7. }
```

In the above case, **copy constructor can be called in the following ways:**

- **A a2(a1);**
- **A a2 = a1;**

} **a1 initialises the a2 object.**

Let's see a simple example of the copy constructor.

// program of the copy constructor.

```
1. #include <iostream>
2. using namespace std;
3. class A
4. {
5.   public:
6.    int x;
7.    A(int a)            // parameterized constructor.
```

```
8.      {
9.        x=a;
10.     }
11.     A(A &i)            // copy constructor
12.     {
13.       x = i.x;
14.     }
15. };
16. int main()
17. {
18.   A a1(20);            // Calling the parameterized constructor.
19.  A a2(a1);             //  Calling the copy constructor.
20.  cout<<a2.x;
21.   return 0;
22. }
```

Output:

```
20
```

When Copy Constructor is called

Copy Constructor is called in the following scenarios:

- When we initialize the object with another existing object of the same class type. For example, Student s1 = s2, where Student is the class.
- When the object of the same class type is passed by value as an argument.
- When the function returns the object of the same class type by value.

Two types of copies are produced by the constructor:

- Shallow copy
- Deep copy

Shallow Copy

- The default copy constructor can only produce the shallow copy.

- A Shallow copy is defined as the process of creating the copy of an object by copying data of all the member variables as it is.

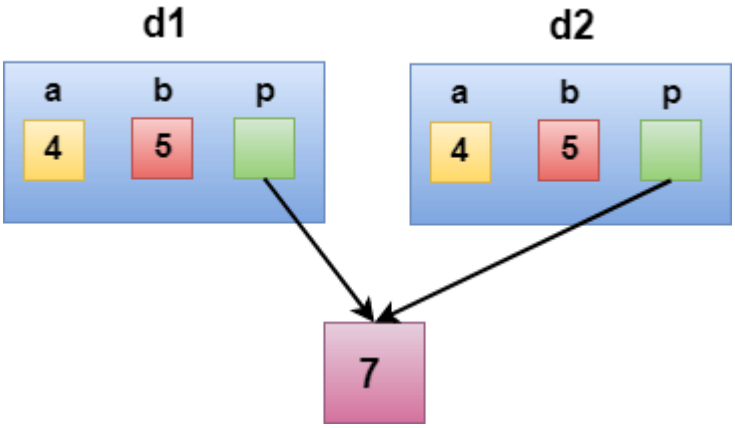
Let's understand this through a simple example:

```
1.  #include <iostream>
2.
3.  using namespace std;
4.
5.  class Demo
6.  {
7.     int a;
8.     int b;
9.     int *p;
10.    public:
11.    Demo()
12.    {
13.       p=new int;
14.    }
15.    void setdata(int x,int y,int z)
16.    {
17.       a=x;
18.       b=y;
19.       *p=z;
20.    }
21.    void showdata()
22.    {
23.       std::cout << "value of a is : " <<a<< std::endl;
24.       std::cout << "value of b is : " <<b<< std::endl;
25.       std::cout << "value of *p is : " <<*p<< std::endl;
26.    }
27. };
28. int main()
29. {
30.   Demo d1;
```

```
31.   d1.setdata(4,5,7);
32.   Demo d2 = d1;
33.   d2.showdata();
34.    return 0;
35. }
```

Output:

```
value of a is : 4
value of b is : 5
value of *p is : 7
```



In the above case, a programmer has not defined any constructor, therefore, the statement **Demo d2 = d1;** calls the default constructor defined by the compiler. The default constructor creates the exact copy or shallow copy of the existing object. Thus, the pointer p of both the objects point to the same memory location. Therefore, when the memory of a field is freed, the memory of another field is also automatically freed as both the fields point to the same memory location. This problem is solved by the **user-defined constructor** that creates the **Deep copy**.

Deep copy

Deep copy dynamically allocates the memory for the copy and then copies the actual value, both the source and copy have distinct memory locations. In this way, both the source and copy are distinct and will not share the same memory location. Deep copy requires us to write the user-defined constructor.

Let's understand this through a simple example.

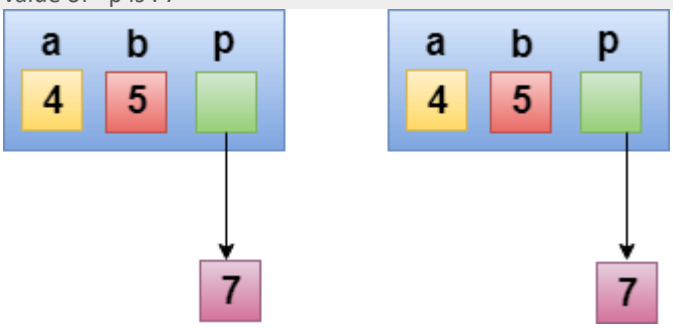
```
1. #include <iostream>
2. using namespace std;
```

```
3.  class Demo
4.  {
5.     public:
6.     int a;
7.     int b;
8.     int *p;
9.
10.    Demo()
11.    {
12.       p=new int;
13.    }
14.    Demo(Demo &d)
15.    {
16.       a = d.a;
17.       b = d.b;
18.       p = new int;
19.       *p = *(d.p);
20.    }
21.    void setdata(int x,int y,int z)
22.    {
23.       a=x;
24.       b=y;
25.       *p=z;
26.    }
27.    void showdata()
28.    {
29.       std::cout << "value of a is : " <<a<< std::endl;
30.       std::cout << "value of b is : " <<b<< std::endl;
31.       std::cout << "value of *p is : " <<*p<< std::endl;
32.    }
33. };
34. int main()
35. {
36.   Demo d1;
```

```
37.   d1.setdata(4,5,7);
38.   Demo d2 = d1;
39.   d2.showdata();
40.   return 0;
41. }
```

Output:

```
value of a is : 4
value of b is : 5
value of *p is : 7
```



In the above case, a programmer has defined its own constructor, therefore the statement **Demo d2 = d1;** calls the copy constructor defined by the user. It creates the exact copy of the value types data and the object pointed by the pointer p. Deep copy does not create the copy of a reference type variable.

Dynamic Constructor in C++ with Examples

When allocation of memory is done dynamically using dynamic memory allocator new in a constructor, it is known as **dynamic constructor**. By using this, we can dynamically initialize the objects.
Example 1:

- CPP14

```
#include <iostream>

using namespace std;
```

```
class geeks {
    const char* p;

public:
    // default constructor
    geeks()
    {

        // allocating memory at run time
        p = new char[6];
        p = "geeks";
    }

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

```
int main()

{

   geeks obj;

   obj.display();

}
```

Output:

Explanation: In this we point data member of type **char** which is allocated memory dynamically by **new** operator and when we create dynamic memory within the constructor of class this is known as dynamic constructor.

What is a destructor?

Destructor is an instance member function that is invoked automatically whenever an object is going to be destroyed. Meaning, a destructor is the last function that is going to be called before an object is destroyed.

- A destructor is also a special member function like a constructor. Destructor destroys the class objects created by the constructor.
- Destructor has the same name as their class name preceded by a tilde (~) symbol.
- It is not possible to define more than one destructor.
- The destructor is only one way to destroy the object created by the constructor. Hence destructor can-not be overloaded.
- Destructor neither requires any argument nor returns any value.
- It is automatically called when an object goes out of scope.
- Destructor release memory space occupied by the objects created by the constructor.
- In destructor, objects are destroyed in the reverse of an object creation.

The thing is to be noted here if the object is created by using new or the constructor uses new to allocate memory that resides in the heap memory or the free store, the destructor should use delete to free the memory.

Syntax

The syntax for defining the destructor within the class:

```
~ <class-name>() {
   // some instructions
}
```

The syntax for defining the destructor outside the class:

```
<class-name> :: ~<class-name>() {
   // some instructions
}
```

Properties of Destructor

The following are the main properties of Destructor:

- The destructor function is automatically invoked when the objects are destroyed.
- It cannot be declared static or const.
- The destructor does not have arguments.
- It has no return type not even void.
- An object of a class with a Destructor cannot become a member of the union.
- A destructor should be declared in the public section of the class.
- The programmer cannot access the address of the destructor.

When is the destructor called?

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

1. the function ends
2. the program ends
3. a block containing local variables ends
4. a delete operator is called

Note: destructor *can also be called explicitly for an object.*

How to call destructors explicitly?

We can call the destructors explicitly using the following statement:

```
object_name.~class_name()
```

How are destructors different from normal member functions?

- Destructors have the same name as the class preceded by a tilde (~)

- Destructors don’t take any argument and don’t return anything

Constraints on constructors and destructors

Like other member functions, constructors and destructors are declared within a class declaration. They can be defined inline or external to the class declaration. Constructors can have default arguments. Unlike other member functions, constructors can have member initialization lists. The following restrictions apply to constructors and destructors:

- Constructors and destructors do not have return types nor can they return values.
- References and pointers cannot be used on constructors and destructors because their addresses cannot be taken.
- Constructors cannot be declared with the keyword virtual.
- Constructors and destructors cannot be declared static, const, or volatile.
- Unions cannot contain class objects that have constructors or destructors.