Constructors and destructors

Object Initialization and Clean-up : Constructor

- A constructor is a special member function whose main operation is to allocate the required resources such as memory and initialize the objects of a class
- A constructor is also a member function that has the same name as its class and is invoked automatically.
- It can be overloaded, no return type.
- Must be declared in the public portion of a class.
- It is automatically called as and when an object of a class is created.
- If we do not define an explicit constructor the C++ compiler creates a default constructor.

Types of Constructors

1. Default Constructors – Which doesn't have any arguments / Parameters.
2. Parameterized Constructors – Which has arguments
3. Copy Constructors – is a member function which initializes its objects using another object of same class.

Syntax

```
class classname {

.....
.....
public:
classname(){ definition inside}
.....
......
};

classname::classname() {definition outside }
```

Simple Example

```
class Counter
{
private: unsigned int count;
public: Counter() //Constructor
{count=0}
void inc_count()
{
 count++;
}
int get_count()
{
 return count;
}
 };
```

```
void main()
{
Counter c1, c2;
cout<<"\nc1="<< c1.get_count();
cout<<"\nc2="<<c2.get_count();
c1.inc_count(); //increment c1
c2.inc_count(); //increment c2
c2.inc_count(); //increment c2
cout <<"\nc1="<<c1.get_count();
cout <<"\nc2="<<c2.get_count();
}
c1=0
c2=0
c1=1
c2=2
```

Simple Example

```
Class integer
{
    int m,n;
public:
    integer() //Constructor
    ..............
    ..............
};
Integer::integer()
{
    m=0; n=0;
}
```

```
void main()
{
    integer c1, c2;
    cout<<"\nc1="<< c1.m<<c1.n;
    cout<<"\nc2="<<c2.m<<c2.n;
}
```

```
c1=0 0
c2=0 0
```

Scope of Constructor (Implicit and explicit)

```
Class Test
{
public:
Test();
};
Test::Test() {
cout<<“constructor of class
test called \n ”;
}
Test G;
void f1() {
Test L;
cout<<“here’s function f1()\n”;
}
```

```
void main()
{
Test X;
cout<“main() function\n”;
f1();
};
```

```
Constructor of class Test called
(global object G)
Constructor of class Test called
(object X in main)
main() function
Constructor of class Test called
(object L in func())
here’s function f1()
```

Parameterized Constructors

- **A constructor that can take arguments are called as *Parameterized Constructors***
- **Used to initialize various data elements of different objects with different values when they are created.**

Parameterized Constructors

```
Class integer
{
    int m,n;
public:
    integer(int x,int y);
    //parameterized
    Constructor
    ..............
    ..............
};
Integer::integer(int
    x,int y)
{
    m=x; n=y;
}
```

```
void main()
{
    integer c1, c2;
    c1(10,20);
    c2(100,200)
    cout<<"\nc1="<< c1.m<<c1.n;
    cout<<"\nc2="<<c2.m<<c2.n;
}
```

```
c1=10 20
c2=100 200
```

Parameterized Constructors

```
class box
{
double width,height,depth;
public:
//Initialization list concept
box(double w,double h,double d)
{
width = w;
height = h;
depth = d;

double volume()
{
return width*height*depth;
}
};
```

```
void main()
{
box mybox1(10,20,15);
box mybox2(3,6,9);
cout<<mybox1.volume();
cout<<mybox2.volume();
}
```

```
3000.0
162.0
```

Multiple Constructors in a class

```
Class integer
{
    int m,n;
public:
    integer() { m=0; n=0;} //Default constructor
    integer(int x,int y)    // Parametrized constructor
    { m=x; n=y;}
    integer(integer &c1) // copy constructor
    { m=c1.m; n=c1.n; }

};
```

Destructor

- When an object is no longer needed it can be destroyed
- A class can have another special member function called the destructor, which is invoked when an object is destroyed
- A destructor is a special member function whose main operation is to de-allocate the required resources such as memory of a class
- A destructor is also a member function has the same name as its class and is invoked automatically.
- It cannot be overloaded, no return type and must be declared in the public portion of a class

Syntax

```
class classname {

.....

.....
public:
~classname(){ definition inside}

.....

......
};

classname::~classname() {definition outside }
```

Simple Example

```
class test {
public:
test();
~test();
 };
test::test()  {
cout<<“constructor of test
    class called\n”;
}
test::~test()  {
cout<<“destructor of test
    class called\n”;
}
```

```
void main()
{
test x;
cout<<“Terminating main()\n”;
}
```

```
constructor of test class called
Terminating main
destructor of test class called
```

Constructor Overloading

- An interesting feature of constructor overloading is that a class can have multiple constructors
- This is called constructor overloading
- All the constructors have the same name as the corresponding class, and they differ only in terms of their signature

Constructor Overloading

```
class AccClass {
int accno, float balance;
public:
AccClass() {
cin>>accno>>balance; }

AccClass( int accin) {
accno=accin; balance=0.0; }

AccClass(int accin,float bal) {
accno=accin; balance=bal; }

void display() {
cout<<accno<<balance; }

void MoneyTransfer(AccClass &
acc, float amount) {
balance=balance-amount;
 acc.balance +=amount;}
};
```

```
void main()
{
AccClass acc1;
AccClass acc2(10);
AccClass acc3(20,750);
acc1.display();
acc2.display();
acc3.display();
float credit;
  cout<<"enter the amount to be
  transferred from acc3 to acc1";
cin>>credit;
acc3.MoneyTransfer(acc1,credit);
            acc1.display();
             acc2.display();
            acc3.display();
}
```

Copy Constructor

```
class code
{
int id;
public:
code() {}
code(int a)
{
    id=a;
}
code(code &x)
{
    id=x.id;
}
Void display(void)
{
Cout<<id;
}
};
```

```
void main()
{
code A(100);//A is created and initialized
code B(A);// Copy Constructor called
code C = A; // Copy constructor called again
Code D;//D is created not initialized
D = A;

cout<<"\n id of A:"; A.display();
cout<<"\n id of B:"; B.display();
cout<<"\n id of C:"; C.display();
cout<<"\n id of D:"; D.display();
}
```

Hybrid Constructor

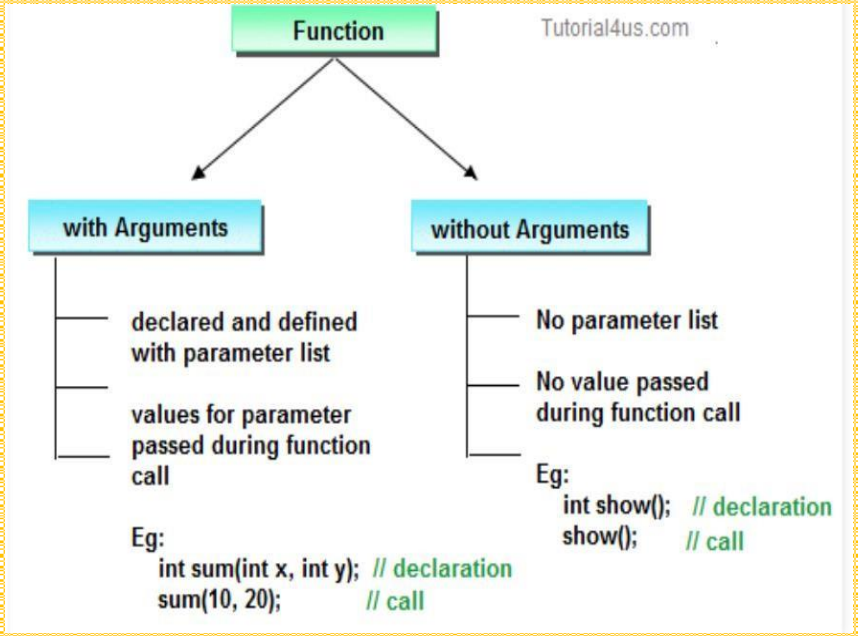
```
class test
{
int a,b,c,d;
public:
test() {}
test(int k1,int k2,int k3,int k4)
{
a=k1;b=k2;c=k3;d=k4;
}
test(test &t1,test &t2)
{
a=t1.a;b=t1.b;
c=t2.c;d=t2.d;
}
test(test &t2,int k1,int k2)
{
a=t2.a;b=t2.b;
c=k1;d=k2;
}
};
```

```
void test::showdata()
{
cout<<" a is "<<a<<endl;
cout<<" b is "<<b<<endl;
cout<<" c is "<<c<<endl;
cout<<" d is "<<d<<endl;
}
void main()
{
test t1(1,2,3,4); test t2(5,6,7,8);
test t3(t1,t2);     test t4(t2,0,0);
cout<<"t1 object"<<endl;
t1.showdata();
cout<<"t2 object"<<endl;
t2.showdata();
cout<<"t3 object"<<endl;
t3.showdata();
cout<<"t4 object"<<endl;
t4.showdata();
}
```

OUTPUT

```
t1 object
 a is 1
 b is 2
 c is 3
 d is 4
t2 object
 a is 5
 b is 6
 c is 7
 d is 8
t3 object
 a is 1
 b is 2
 c is 7
 d is 8
t4 object
 a is 5
 b is 6
 c is 0
 d is 0
```

Object as function arguments



- If a function takes **any arguments,** it must declare variables that accept the values as an argument.
- These variables are called the **formal parameters of the function**.
- There are **two ways** to pass value or data to function in C++ language which are given below;
  1. call by value
  2. call by reference

Object as function arguments

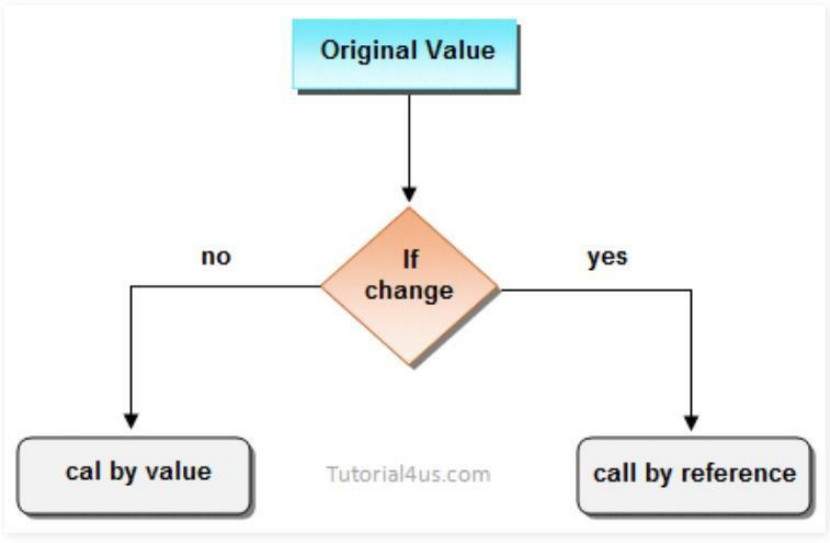
1. Pass by value /call by value

- A copy of the entire object is passed to the function.
- So, the changes made to the object inside the function do not affect the actual object
- Ie: In call by value, the original value can not be changed or modified.
- In call by value, when you passed value to the function it is locally stored by the function parameter in a stack memory location.
- If you change the value of the function parameter, it is changed for the current function only but it did not change the value of a variable inside the **caller function such as main().**

Object as function arguments

2. Pass by reference /Call by reference

- Only the address of the object is passed to its function.
- So the changes made to the object inside the function will reflect in the actual object.
- Ie: In call by reference, the original value is changed or modified because we pass a reference (address).
- Here, the address of the value is passed in the function, so actual and formal arguments share the **same address space**.
- Hence, any value **changed inside the function**, is ***reflected*** **inside as well as outside the function**.



Difference between call by value and call by reference.

	call by value	call by reference
1	This method copy original value into function as a arguments.	This method copy address of arguments into function as a arguments.
2	Changes made to the parameter inside the function have no effect on the argument.	Changes made to the parameter affect the argument. Because address is used to access the actual argument.
3	Actual and formal arguments will be created in different memory location	Actual and formal arguments will be created in same memory location

Note: By default, C++ uses call by value to pass arguments.

Call by Value and Call by Reference in C++ - Example

Call by value

```
#include<iostream.h>
#include<conio.h>

void swap(int a, int b)
{
 int temp;
 temp=a;
 a=b;
 b=temp;
}

void main()
{
 int a=100, b=200;
 clrscr();
 swap(a, b);   // passing value to function
 cout<<"Value of a"<<a;
 cout<<"Value of b"<<b;
 getch();
}
```

Example Call by reference

```
#include<iostream.h>
#include<conio.h>

void swap(int *a, int *b)
{
 int temp;
 temp=*a;
 *a=*b;
 *b=temp;
}

void main()
{
 int a=100, b=200;
 clrscr();
 swap(&a, &b);   // passing value to function
 cout<<"Value of a"<<a;
 cout<<"Value of b"<<b;
 getch();
}
```

Output

```
Value of a: 200
Value of b: 100
```