C++ Notes _ PART 2

Jaydeep Shah (radhey04ec@gmail.com)

Constructor:

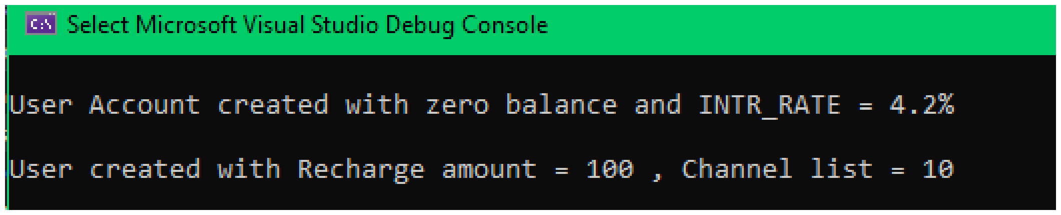
Constructor is member function of class, main purpose of this function is to initialization of objects value when it will be created.

It's name is same as class name. **Constructor is invokes when every object of associated class is created**.

- Constructor must be defined in public section
- Invoke when every object created.
- Constructor's name is same as class name.
- Constructor without any argument known as default constructor. It will invoke automatically when object is created.
- Constructor with argument, known as parameterized constructor, need to pass argument during create object.
- **Constructor has no return type, return type or void with constructor name will generate compile time error**.

Example Code:

```
//Tutorial 16: Constructor & Destructor in C++
//Member function of class an declared in public section, name of constructor always same as class name.
//Created by : Jaydeep Shah (radhey04ec@gmail.com)

#include <iostream>
using namespace std;


//Create class
class bank
{
public:
    int   BANK_BALANCE;
    float INTR_RATE;

    //Constructor -Default constructor without any argument - Constructor has no return type
    bank()
    {
        cout << endl << "User Account created with zero balance and INTR_RATE = 4.2%" << endl;
        BANK_BALANCE = 0;
        INTR_RATE = 4.2;
    }

};

//Create other class
class RECHARGE_PLAN
{
public:
    int RECHARGE_AMOUNT;
    int CHANNEL_SUBSCRIPTION;

    //Create Paarmeterized constructor
    RECHARGE_PLAN(int REC_AMOUNT, int CHANNEL)
    {
        RECHARGE_AMOUNT = REC_AMOUNT;
        CHANNEL_SUBSCRIPTION = CHANNEL;
        cout << endl << "User created with Recharge amount = " << RECHARGE_AMOUNT << " , Channel list = " << CHANNEL_SUBSCRIPTION <<
endl;

    }


};

int main()
{

    //Create object
    bank USER_1;                              //Object -- Default constructor type

    RECHARGE_PLAN USER_A(100, 10);    //Object with constructor values

    return 0;
}
```



Select Microsoft Visual Studio Debug Console

```
User Account created with zero balance and INTR_RATE = 4.2%

User created with Recharge amount = 100 , Channel list = 10
```

CONSTRUCTOR OVERLOADING:

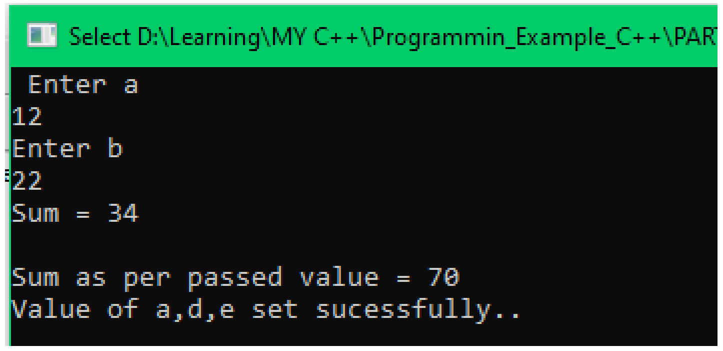
Constructor with multiple definitions inside class known as constructor overloading. It provides more flexibility while creating object.

Example:

```
//Tutorial 17 : Constructor overload
//Multiple constructor definition within class, provides more flexiblity for user
//Created by : Jaydeep Shah (radhey04ec@gmail.com)

#include <iostream>
using namespace std;

//Create Class
class sum
{
private:
      int a;
      int b;
      int c;
      float d;
      double e;

public:
      //Constructor - Default type - No Argument
      sum()
      {
            //Take the value from user
            cout << " Enter a " << endl;
            cin >> a;
            cout << "Enter b " << endl;
            cin >> b;
            cout << "Sum = " << (a + b) << endl;
      }
```

```cpp
	//Constructor when user pass value of a and b
	sum(int a, int b);

	//Constructor when user will pass value of a and d and c
	sum(int a, float d, double c);

};


//----------------------------------------------------
//Constructor overloading
//Constructor function definition
sum::sum(int x, int y)
{
	a = x;
	b = y;
	cout << endl << "Sum as per passed value = " << (a + b) << endl;

}

//Constructor function definition
sum::sum(int p, float q, double r)
{
	a = p;
	d = q;
	e = r;
	cout << "Value of a,d,e set sucessfully.." << endl;
}
//----------------------------------------------------

int main()
{

	//Create Obj
	sum First_Obj;                                           //Default constructor call

	sum Second_Obj(20, 50);                            //Constructor overload - Implicit call

	sum Third_Obj = sum(3, 3.2, 4.55);        //Other way of calling constructor - explicit call

	return 0;
}
```

OUTPUT:



Select D:\Learning\MY C++\Programmin_Example_C++\PAR

```
 Enter a
12
Enter b
22
Sum = 34

Sum as per passed value = 70
Value of a,d,e set sucessfully..
```

Operator overloading implementation:

As we know C++ supports **Polymorphism**, and operator overloading is part of it.

It is idea of giving special meaning of existing operator in C++ without changing it’s original meaning.

Can we overload all operators?

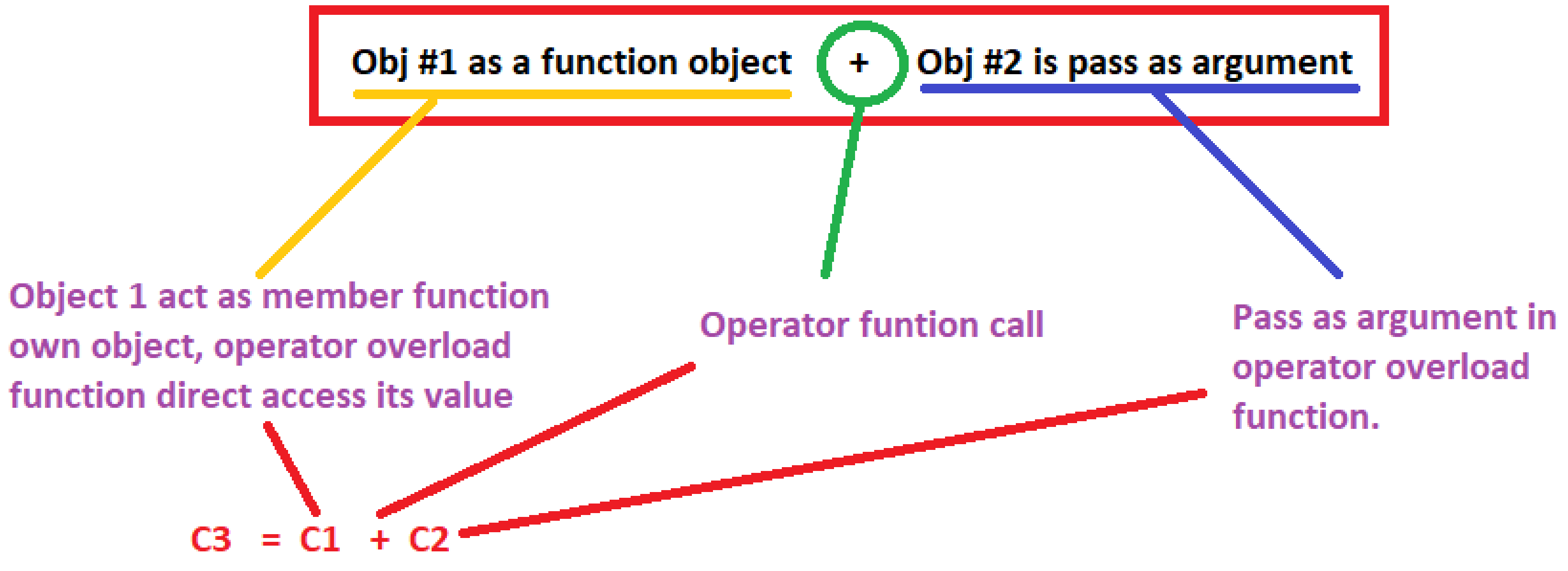
Almost yes, but there are few exceptions. Following operators we cannot overload.

- **sizeof(), typeid, scope resolution(::), Ternary condition (? :), Class member access operator (dot “.”)** .

How to use?

Syntax and use of operator overloading is same like function, just need to use keyword “operator” during declaration of operator.

It has Argument and return type like function.


Obj #1 as a function object
+
Obj #2 is pass as argument
Object 1 act as member function own object, operator overload function direct access its value
Operator funtion call
Pass as argument in operator overload function.
C3 = C1 + C2

Example:

int operator+(string &s)

{

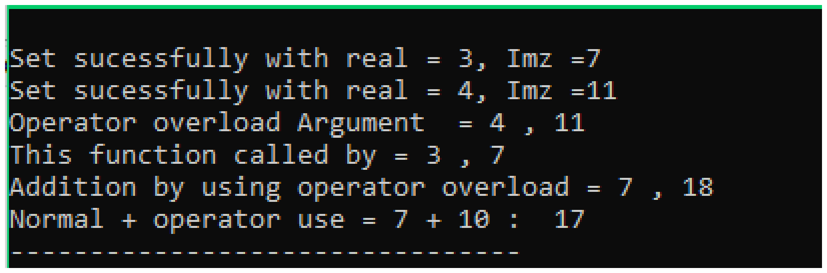
}

int is return type, + use as operator overload function call.

Code:

```
//Tutorial 18 : Operator overloading. (Polymorphism)
//Operator overloading is same like function declaration an definition. It has return type and argument with operator keyword.
//Created by : Jaydeep Shah (radhey04ec@gmail.com)

#include <iostream>
using namespace std;

//Create complex class
class complex
{
public:
	double real; //Variable related with object
	double imz;


	//Constructor
	complex()
	{
		real = 0;
		imz = 0;
	}

	//Member function
	void set(double a, double b)
	{
		real = a;
		imz = b;
		cout << endl << "Set sucessfully with real = " << real << ", Imz =" << imz;

	}

	//Operator overloading
	complex operator+(complex const& obj)   //"+" Operator overloading
	{
		complex data;

		//For knowing what is happening during call of operator
		cout << endl << "Operator overload Argument  = " << obj.real << " , " << obj.imz;
		cout << endl << "This function called by = " << real << " , " << imz;

		// "+" use of as a operator overloading
		data.real = real + obj.real;
		data.imz = imz + obj.imz;


		return(data);
	}


};



int main()
{
	//Create first object and set value
	complex C1;
	C1.set(3, 7);

	//Create second object and set value
	complex C2;
	C2.set(4, 11);

	//Operator overload call
	complex C3 = C1 + C2;   //"+" called when Both object is complex type
	//Print value
	cout << endl << "Addition by using operator overload = " << C3.real << " , " << C3.imz;

	//Normal "+" operator
	cout << endl << "Normal + operator use = 7 + 10 :  " << (7 + 10);

	return 0;
}
```

Output:



```
Set sucessfully with real = 3, Imz =7
Set sucessfully with real = 4, Imz =11
Operator overload Argument  = 4 , 11
This function called by = 3 , 7
Addition by using operator overload = 7 , 18
Normal + operator use = 7 + 10 :  17
--------------------------------
```

Dynamic memory allocation / Stack and Heap / use of new() and delete():

When we creates local variable or calling the function, required size of memory allocated for that variables inside memory area known as "**Stack**". This memory management handles during runtime by system itself. So during every function call, their local variables will be stored inside stack. And here is dangerous situation when recursive function comes into picture. If recursive functions not break/stop after meet specific condition, it leads toward stack overflow problem.

Stack work on "LIFO" principal, Last in variable will be deleted or Out First.

Stack and Heap always grow in opposite direction. Sometimes user need to allocate memory during runtime but as per requirement. This is known as Dynamic memory allocation, this dynamic memory allocation happens inside memory area known as **"Heap".** But this dynamic allocated memory need to be delete after usage otherwise Heap area continue increase and merge with stack/ or **creating memory leak** problem.

In C, there is calloc(), or malloc() and free() function for this purpose, in C++ here new() and delete().

Syntax:

new(data-type) , it will return memory pointer of related data type.

delete(memory pointer).

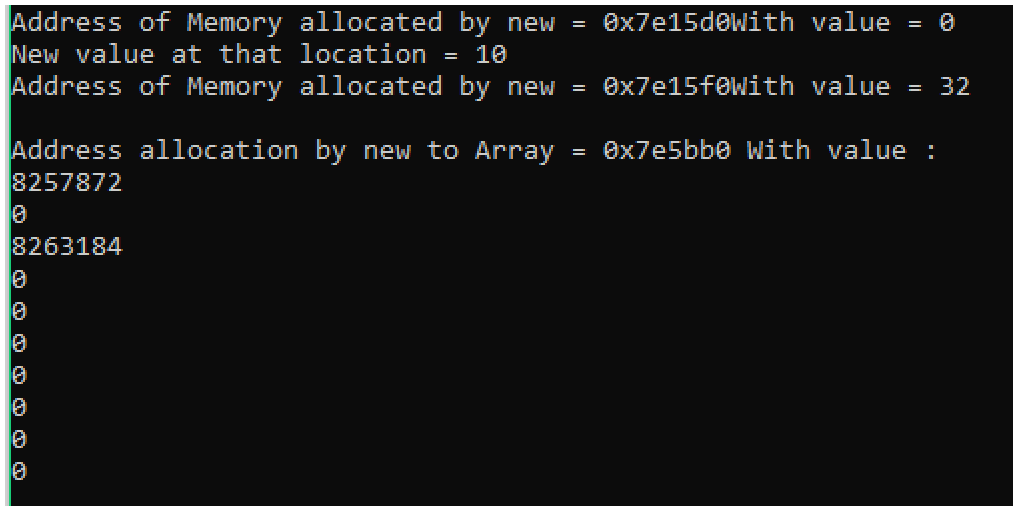
Note: We can pass initial value during call of new(), example int *p = new int(32). This weill allocate memory on heap with value of 32. This facility only available in new version of C++ after C19.

Dynamic memory allocation always return pointer.

Example:

```
//Tutorial 19: Dynamic memory allocation in C++
// new() for allocate memory, delete() for de allocate memory from Heap
//Created by : Jaydeep Shah (radhey04ec@gmail.com)

#include <iostream>
using namespace std;


int main()
{
    //new() operator
    int* point = new int();
    cout << "Address of Memory allocated by new = " << point << "With value = " << *point;
    *point = 10;
    cout << endl << "New value at that location = " << *point;

    //Pass initial value
    int* point1 = new int(32);
    cout << endl << "Address of Memory allocated by new = " << point1 << "With value = " << *point1 << endl;


    //Array memory allocation on Heap
    int* point_of_array = new int[10];  //10 bytes allocation on heap
    cout << endl << "Address allocation by new to Array = " << point_of_array << " With value : " << endl;
    for (int i = 0; i <= 9; i++)
    {
        cout << point_of_array[i] << endl;
    }

    //To avoid dangling need to delete this allocated memory
    delete(point);
    delete[] point_of_array;
    point = NULL;
    point_of_array = NULL;

    return 0;
}
```

Output:



```
Address of Memory allocated by new = 0x7e15d0With value = 0
New value at that location = 10
Address of Memory allocated by new = 0x7e15f0With value = 32

Address allocation by new to Array = 0x7e5bb0 With value :
8257872
0
8263184
0
0
0
0
0
0
0
```