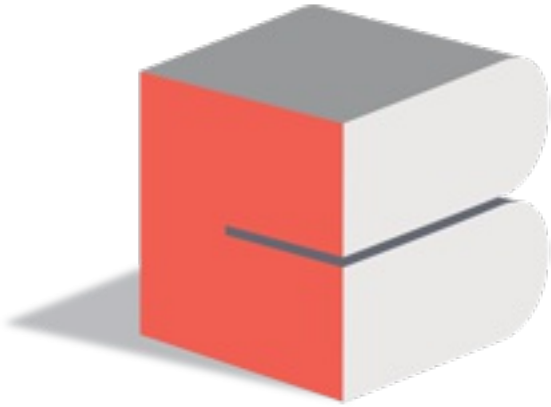

Object Oriented Programming - Constructors

- Constructor has same name as the class itself
- Constructors don’t have return type
- A constructor is automatically called when an object is created.
- If we do not specify a constructor, C++ compiler generates a default constructor for us (expects no parameters and has an empty body).

Default Constructor

Default constructor is the constructor which doesn’t take any argument. It has no parameters.

```
// 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;
}
```

OTUPUT

```
a: 10
b: 20
```

Note Even if we do not define any constructor explicitly, the compiler will automatically provide a default constructor implicitly. The default value of variables is 0 in case of automatic initialization.

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.

```
// 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

- It is used to initialize the various data elements of different objects with different values when they are created.
- It is used to overload constructors.

Copy Constructor

A copy constructor is a member function which initializes an object using another object of the same class.

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

class Point
{
private:
    int x, y;
public:
```

```
    Point(int x1, int y1) { x = x1; y = y1; }

    // Copy constructor
    Point(const Point &p2) {x = p2.x; y = p2.y; }

    int getX()            {  return x; }
    int getY()            {  return y; }
};

int main()
{
    Point p1(10, 15); // Normal constructor is called here
    Point p2 = p1; // Copy constructor is called here

    // Let us access values assigned by constructors
    cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY(
);
    cout << "\np2.x = " << p2.getX() << ", p2.y = " << p2.get
Y();

    return 0;
}
```

OUTPUT

```
p1.x = 10, p1.y = 15
p2.x = 10, p2.y = 15
```

Example showing use of default, parameterised and copy constructor

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

class Car{
private:
    int price;
public:
    int model_no;
    char name[20];

    //Constructor
    Car(){
        //Override the default Constructor
        cout<<"Making a car.."<<endl;
    }
    // Constructor with parameters - Parametrised Constructor

    Car(int p,int mn,char *n){
        price = p;
        model_no = mn;
        strcpy(name,n);

    }
```

```
    Car(Car &X){
        cout<<"Making a Copy of Car";
        price = X.price;
        model_no = X.model_no;
        strcpy(name,X.name);
    }


    void start(){
        cout<<"Grrrr...starting the car "<<name<<endl;
    }
    void setPrice(int p){
        if(p>1000){
            price  = p;
        }
        else{
            price = 1000;
        }
    }
    int getPrice(){
        return price;
    }
    void print(){
        cout<<name<<endl;
        cout<<model_no<<endl;
        cout<<price<<endl;
        cout<<endl;
    }
```

```
};

int main() {

    Car C;
    //Initialisation
    //C.price =-500;
    C.setPrice(1500);
    C.model_no = 1001;
    C.name[0] = 'B';
    C.name[1] = 'M';
    C.name[2] = 'W';
    C.name[3] = '\0';
    C.start();

   // cout<<C.price<<endl;
    cout<<C.name<<endl;
    cout<<C.getPrice()<<endl;

    Car D;
    D.setPrice(2000);
    cout<<D.getPrice()<<endl;

    Car E(100,2001,"Ferrari");
    //E.print();

    //Copy Constructor is to create a copy of given object of
```

```
 the same type
    Car F(E);
    F.setPrice(2000);
    F.name[0] = 'G';
    F.print();
    E.print();

    Car G = F;
    G.print();
    F.print();




    return 0;

}
```

OUTPUT

```
Making a car..
Grrrr...starting the car BMW
BMW
1500
Making a car..
2000
```

```
Making a Copy of CarGerrari
2001
2000

Ferrari
2001
100

Making a Copy of CarGerrari
2001
2000

Gerrari
2001
2000
```

Shallow and Deep Copy Constructor

A shallow copy of an object copies all of the member field values. This works well if the fields are values, but may not be what you want for fields that point to dynamically allocated memory. The pointer will be copied. but the memory it points to will not be copied – the field 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.

A deep copy copies all fields, and makes copies of dynamically allocated

memory pointed to by the fields. To make a deep copy, you must write a copy constructor and overload the assignment operator, otherwise the copy will point to the original, with disasterous consequences.

Example

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

class Car{
private:
    int price;
public:
    int model_no;
    char *name;

    //Constructor
    Car(){
        //Override the default Constructor
        name = NULL;
        cout<<"Making a car.."<<endl;
    }
    // Constructor with parameters - Parametrised Constructor

    Car(int p,int mn,char *n){
        price = p;
        model_no = mn;
```

```
        int l  = strlen(n);
        name = new char[l+1];
        strcpy(name,n);

    }

   //Deep Copy Constructor
   Car(Car &X){
       // cout<<"Making a Copy of Car";
        price = X.price;
        model_no = X.model_no;
        int l = strlen(X.name);
        name = new char[l+1];
        strcpy(name,X.name);
    }

    void setName(char *n){
        if(name==NULL){
            name = new char[strlen(n)+1];
            strcpy(name,n);
        }
        else{
            //Later...
            //Delete the oldname array and allocate a new one
.

        }
    }
```

```
    void start(){
        cout<<"Grrrr...starting the car "<<name<<endl;
    }
    void setPrice(int p){
        if(p>1000){
            price  = p;
        }
        else{
            price = 1000;
        }
    }
    int getPrice(){
        return price;
    }
    void print(){
        cout<<name<<endl;
        cout<<model_no<<endl;
        cout<<price<<endl;
        cout<<endl;
    }

};

int main() {

    Car C;
    //Initialisation
```

```
    //C.price =-500;
    C.setPrice(1500);
    C.setName("Nano");
    C.model_no = 1001;
    //C.start();
    C.print();

    Car D(100,200,"BMW");

    Car E(D); //Default Copy Constructor
    E.name[0] ='G';

    D.print();
    E.print();









    return 0;

}
```

Copy Assignment Operator

EXAMPLE

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

class Car{
private:
    int price;
public:
    int model_no;
    char *name;

    //Constructor
    Car(){
        //Override the default Constructor
        name = NULL;
        cout<<"Making a car.."<<endl;
    }
    // Constructor with parameters - Parametrised Constructor

    Car(int p,int mn,char *n){
        price = p;
        model_no = mn;
        int l  = strlen(n);
        name = new char[l+1];
        strcpy(name,n);
```

```
    }

    //Deep Copy Constructor
    Car(Car &X){
        // cout<<"Making a Copy of Car";
        price = X.price;
        model_no = X.model_no;
        int l = strlen(X.name);
        name = new char[l+1];
        strcpy(name,X.name);
    }

    void operator = (Car &X){
        cout<<"In Copy Assignment Operator"<<endl;
        price = X.price;
        model_no = X.model_no;
        int l = strlen(X.name);
        name = new char[l+1];
        strcpy(name,X.name);

    }

    void setName(char *n){
        if(name==NULL){
            name = new char[strlen(n)+1];
            strcpy(name,n);
        }
```

```
        else{
            //Later...
            //Delete the oldname array and allocate a new one
.

        }
    }

    void start(){
        cout<<"Grrrr...starting the car "<<name<<endl;
    }
    void setPrice(int p){
        if(p>1000){
            price  = p;
        }
        else{
            price = 1000;
        }
    }
    int getPrice(){
        return price;
    }
    void print(){
        cout<<name<<endl;
        cout<<model_no<<endl;
        cout<<price<<endl;
        cout<<endl;
    }
```

```
};

int main() {

    Car C;
    //Initialisation
    //C.price =-500;
    C.setPrice(1500);
    C.setName("Nano");
    C.model_no = 1001;
    //C.start();
    C.print();

    Car D(100,200,"BMW");

    Car E(200,400,"Audi") ;//Default Copy Constructor

   // E.name[0] ='G';
     D = E; //Copy Assigment Operator ->Shallow Copy
     D.name[0] = 'O';
     D.print();
     E.print();
```

```
    return 0;

}
```

Destructors

- Destructor is a member function which destructs or deletes an object.
- Destructors have same name as the class preceded by a tilde (~)
- Destructors don’t take any argument and don’t return anything

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

If we do not write our own destructor in class, compiler creates a default destructor for us. The default destructor works fine unless we have dynamically allocated memory or pointer in class. When a class contains a pointer to memory allocated in class, we should write a destructor to release memory before the class instance is destroyed. This must be done to avoid memory leak.

Example

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

class Car{
private:
    int price;
public:
    int model_no;
    char *name;

    //Constructor
    Car(){
        //Override the default Constructor
        name = NULL;
        cout<<"Making a car.."<<endl;
    }
    // Constructor with parameters - Parametrised Constructor

    Car(int p,int mn,char *n){
        price = p;
        model_no = mn;
        int l  = strlen(n);
        name = new char[l+1];
        strcpy(name,n);
```

```
    }

    //Deep Copy Constructor
    Car(Car &X){
        // cout<<"Making a Copy of Car";
        price = X.price;
        model_no = X.model_no;
        int l = strlen(X.name);
        name = new char[l+1];
        strcpy(name,X.name);
    }

    void operator = (Car &X){
        cout<<"In Copy Assignment Operator"<<endl;
        price = X.price;
        model_no = X.model_no;
        int l = strlen(X.name);
        name = new char[l+1];
        strcpy(name,X.name);

    }

    void setName(char *n){
        if(name==NULL){
            name = new char[strlen(n)+1];
            strcpy(name,n);
        }
```

```
        else{
            //Later...
            //Delete the oldname array and allocate a new one
.

        }
    }

    void start(){
        cout<<"Grrrr...starting the car "<<name<<endl;
    }
    void setPrice(int p){
        if(p>1000){
            price  = p;
        }
        else{
            price = 1000;
        }
    }
    int getPrice(){
        return price;
    }
    void print(){
        cout<<name<<endl;
        cout<<model_no<<endl;
        cout<<price<<endl;
        cout<<endl;
    }
```

```
    ~Car(){
        cout<<"Destroying the Car "<<name<<endl;
        //Write code to delete all dynamic data member
        if(name!=NULL){
            delete [] name;
        }
    }

};

int main() {

    Car C;
    //Initialisation
    //C.price =-500;
    C.setPrice(1500);
    C.setName("Nano");
    C.model_no = 1001;
    //C.start();
    C.print();

    Car D(100,200,"BMW");

    Car E(200,400,"Audi") ;//Default Copy Constructor

   // E.name[0] ='G';
     D = E; //Copy Assigment Operator ->Shallow Copy
     D.name[0] = 'O';
```

```
    D.print();
    E.print();


    //Suppose we create a dynamic object
    Car *DC = new Car(100,200,"Dynamic Tesla Car");

    delete DC;


    return 0;

}
```

Const Data Members

- Const keyword is used to make any element of a program constant.
- If we declare a variable as const, we cannot change its value. A const variable must be assigned a value at the time of its declaration.
- Once initialized, if we try to change its value, then we will get compilation error.

Const member functions in C++

- A function becomes const when const keyword is used in function’s

declaration. The idea of const functions is not allow them to modify the object on which they are called.

- When a function is declared as const, it can be called on any type of object. Non-const functions can only be called by non-const objects.

Example : Const Data Members, Initialisation List and Const Member functions

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

class Car{
private:
    int price;
public:
    int model_no;
    char *name;
    const int tyres;

    //Constructor
    Car():name(NULL),tyres(4){
    }
    // Constructor with parameters - Parametrised Constructor

    Car(int p,int mn,char *n,int t=4):price(p),model_no(mn),t
```

```
yres(t){
        int l  = strlen(n);
        name = new char[l+1];
        strcpy(name,n);

    }

   //Deep Copy Constructor
   Car(Car &X):tyres(X.tyres){
       // cout<<"Making a Copy of Car";
        price = X.price;
        model_no = X.model_no;
        int l = strlen(X.name);
        name = new char[l+1];
        strcpy(name,X.name);
    }

    void operator = (const Car &X){
        cout<<"In Copy Assignment Operator"<<endl;
        price = X.price;
        model_no = X.model_no;
        int l = strlen(X.name);
        name = new char[l+1];
        strcpy(name,X.name);

    }

    void setName(const char *n){
```

```
        if(name==NULL){
            name = new char[strlen(n)+1];
            strcpy(name,n);
        }
        else{
            //Later...
            //Delete the oldname array and allocate a new one
.

        }
    }

    void start() const{
        cout<<"Grrrr...starting the car "<<name<<endl;
    }
    void setPrice(const int p){
        if(p>1000){
            price  = p;
        }
        else{
            price = 1000;
        }
    }
    int getPrice() const{
        return price;
    }
    void print() const{
        cout<<name<<endl;
```

```
        cout<<model_no<<endl;
        cout<<price<<endl;
        cout<<endl;
    }
    ~Car(){
        cout<<"Destroying the Car "<<name<<endl;
        //Write code to delete all dynamic data member
        if(name!=NULL){
            delete [] name;
        }
    }

};

int main() {

    Car C;
    //Initialisation
    //C.price =-500;
    C.setPrice(1500);
    C.setName("Nano");
    C.model_no = 1001;
    //C.start();
    C.print();

    Car D(100,200,"BMW");

    Car E(200,400,"Audi") ;//Default Copy Constructor
```

```
    // E.name[0] ='G';
      D = E; //Copy Assigment Operator ->Shallow Copy
      D.name[0] = 'O';
      D.print();
      E.print();

      cout<<E.tyres<<endl;


     //Suppose we create a dynamic object
     Car *DC = new Car(100,200,"Dynamic Tesla Car");

     delete DC;


     return 0;

}
```