

CONSTRUCTOR INITIALIZATION LISTS

➤ Most constructors do nothing more than initialize the object's member data.

➤ Initialization lists are an alternative technique for initializing an object's data members in a constructor. For instance, the constructor

```
date::date(int m, int d, int y)
{
    month= m;
    day  = d;
    year = y;
}
```

can instead be written using an initialization list:

```
date::date(int m, int d, int y) : month(m), day(d), year(y)
{
}
```

- The assignment statements in the function's body that assigned m to month, d to day, and y to year are removed.
- Note that the list begins with a colon and precedes the function body which is now empty.

THE CLASS DESTRUCTOR

- When an object is created, a constructor is called automatically to manage its birth.
- Similarly, when an object comes to the end of its life, another special member function is called automatically to manage its death. *This function is called a destructor.*
- Each class has exactly one destructor.

-
- **Destructors are special member functions.**
 - **Release dynamic allocated memory.**
 - **Destructors takes the same name of class name.**
 - **The general Syntax of Destructors is**

~ classname();

- **Destructors defined in the public.**
- **No return type is specified.**

Consider the following program:

```
#include <iostream>
using namespace std;
class Ratio
{
public:
Ratio() { cout << "OBJECT IS BORN.\n"; }
~Ratio() { cout << "OBJECT DIES.\n"; }
private:
int num, den;
};
int main()
{
    { Ratio x; // beginning of scope for x
      cout << "Now x is alive.\n";
    } // end of scope for x
    cout << "Now between blocks.\n";
    { Ratio y;
      cout << "Now y is alive.\n";
    }
}
```

The output of this program will be as follows:

```
OBJECT IS BORN.  
Now x is alive.  
OBJECT DIES.  
Now between blocks.  
OBJECT IS BORN.  
Now y is alive.  
OBJECT DIES.
```

The output here shows when the constructor and the destructor are called.

When is destructor useful?

Executed when object destroyed

Can do anything, but interesting when deallocate memory

Want to delete items created using new to free up memory

The most common use of destructors is to deallocate memory that was allocated for the object by the constructor.

Objects as Function Arguments

- To pass an object as an argument we write the *object name as the argument* while *calling* the function the same way we do it for other variables.
- *The following program illustrates how you can use objects as function arguments*


```
#include <iostream>
using namespace std;
class Student {
public:
    double marks;

    // constructor to initialize marks
    Student(double m) {
        marks = m;
    }
};

// function that has objects as parameters
void calculateAverage(Student s1, Student s2) {

    // calculate the average of marks of s1 and s2
    double average = (s1.marks + s2.marks) / 2;

    cout << "Average Marks = " << average << endl;

}
```

Continued:

```
int main() {  
    Student student1(88.0), student2(56.0);  
  
    // pass the objects as arguments  
    calculateAverage(student1, student2);  
  
    return 0;  
}
```

Average Marks = 72


➤ *Here, we have passed two Student objects student1 and student2 as arguments to the calculateAverage() function.*

```
#include<iostream>

class Student {...};

void calculateAverage(Student s1, Student s2) {
    // code
}

int main() {
    ... ..
    calculateAverage(student1, student2);
    ... ..
}
```



The diagram illustrates the passing of arguments from the `main` function to the `calculateAverage` function. Two teal arrows originate from the arguments `student1` and `student2` in the `calculateAverage(student1, student2);` call within the `main` function. These arrows point upwards and to the right, terminating at the parameters `s1` and `s2` of the `calculateAverage` function signature, respectively. This visualizes the flow of data from the caller to the callee.

Ex. 2 Return Object from a Function

```
#include <iostream>
using namespace std;
class Student {
public:
    double marks1, marks2;
};
// function that returns object of Student
Student createStudent() {
    Student student;
    // Initialize member variables of Student
    student.marks1 = 96.5;
    student.marks2 = 75.0;
    // print member variables of Student
    cout << "Marks 1 = " << student.marks1 << endl;
    cout << "Marks 2 = " << student.marks2 << endl;
    return student;
}
```

Continue:

```
int main() {  
    Student student1;  
  
    // Call function  
    student1 = createStudent();  
  
    return 0;  
}
```



```
Marks 1 = 96.5  
Marks 2 = 75
```

```
#include<iostream>

class Student {...};

Student createStudent() {
    Student student;
    ... ..
    return student;
}

int main() {
    ... ..
    student1 = createStudent();
    ... ..
}
```

function
call

Ex. Consider the following program

```
#include <iostream>
using namespace std;
class Distance //English Distance class
{
private:
int feet;
float inches;
public: //constructor (no args)
Distance() : feet(0), inches(0.0)
{ } //constructor (two args)
Distance(int ft, float in) : feet(ft), inches(in)
{ }
void getdist() //get length from user
{
cout << "\nEnter feet : "; cin >> feet;
cout << "Enter inches : "; cin >> inches;
}
```

Continued:

```
void showdist() //display distance
{
    cout << feet << "' - " << inches << "'";
}
Distance add_dist(Distance); //add
};
//add this distance to d2, return the sum
Distance Distance::add_dist(Distance d2)
{
    Distance temp; //temporary variable
    temp.inches = inches + d2.inches; //add the inches
    if (temp.inches >= 12.0) //if total exceeds 12.0,
    { //then decrease inches
        temp.inches -= 12.0; //by 12.0 and
        temp.feet = 1; //increase feet
    } //by 1
    temp.feet += feet + d2.feet; //add the feet
    return temp;
}
```


Continued

```
int main()
{
    Distance dist1, dist3; //define two lengths
    Distance dist2(11, 6.25); //define, initialize dist2
    dist1.getdist(); //get dist1 from user
    dist3 = dist1.add_dist(dist2); //dist3 = dist1 + dist2
    //display all lengths
    cout << "\ndist1 = "; dist1.showdist();
    cout << "\ndist2 = "; dist2.showdist();
    cout << "\ndist3 = "; dist3.showdist();
    cout << endl;
    return 0;
}
```