CS-1004 Object Oriented programming
Week 3

Instructor:
ATIYA
LECTURER
(COMPUTER SCIENCE DEPARTMENT)
NATIONAL UNIVERSITY- FAST (KHI CAMPUS)
EMAIL: ATIYA.JOKHIO@NU.EDU.PK
ROOM: (LECTURER ROOM#19, CS DEPT BASEMENT-2)

Constructor

- **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.
- It constructs the values i.e. provides data for the object which is why it is known as constructor.
- 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.

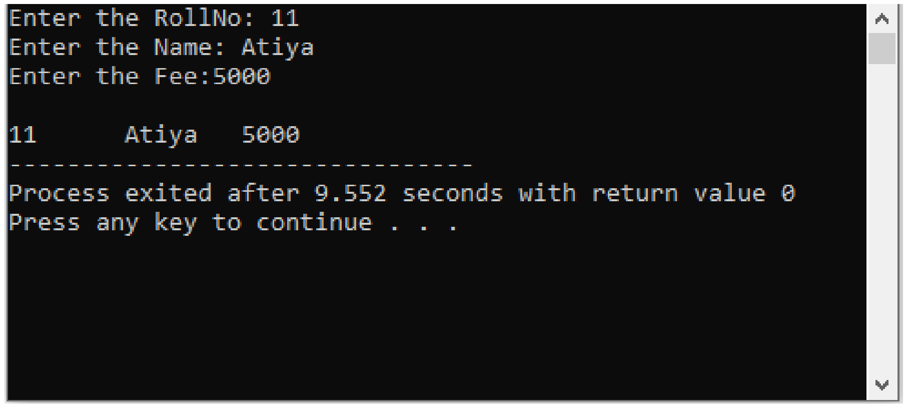
Constructor

```
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
        }
```

Constructor: Example

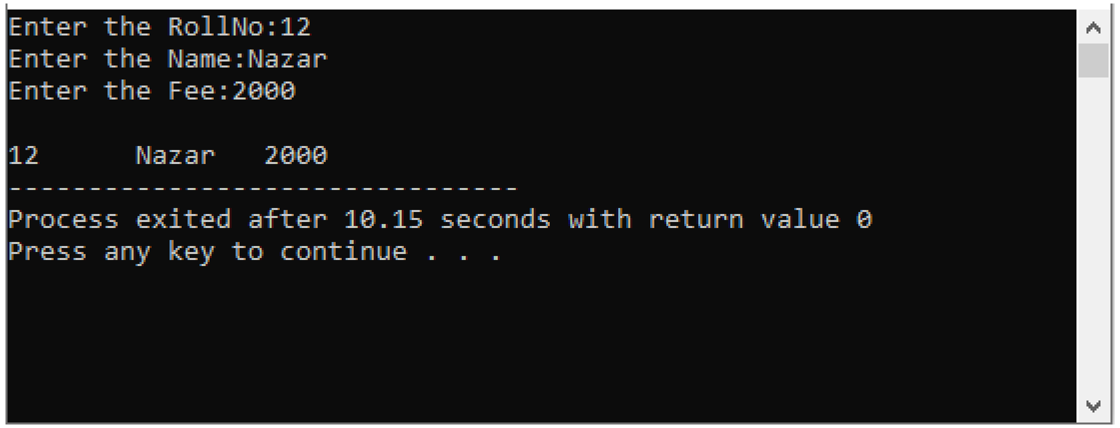
```cpp
// Example: defining the constructor within the class

#include<iostream>
using namespace std;
class student
{
    int rno;
    char name[50];
    double fee;
    public:
    student()
    {
        cout<<"Enter the RollNo:";
        cin>>rno;
        cout<<"Enter the Name:";
        cin>>name;
        cout<<"Enter the Fee:";
        cin>>fee;
    }



    void display()
    {
        cout<<endl<<rno<<"\t"<<name<<"\t"<<fee;
    }
};

int main()
{
    student s;  //constructor gets called automatically when we create the object of the class
    s.display();
    return 0;

}
```



```
Enter the RollNo: 11
Enter the Name: Atiya
Enter the Fee:5000

11      Atiya   5000
--------------------------------
Process exited after 9.552 seconds with return value 0
Press any key to continue . . .
```

Constructor: Example

```
// Example: defining the constructor outside the class

#include<iostream>
using namespace std;
class student
{
    int rno;
    char name[50];
    double fee;
    public:
    student();
    void display();

};

    student::student()
    {
        cout<<"Enter the RollNo:";
        cin>>rno;
        cout<<"Enter the Name:";
        cin>>name;
        cout<<"Enter the Fee:";
        cin>>fee;
    }

   void student::display()
    {
        cout<<endl<<rno<<"\t"<<name<<"\t"<<fee;
    }

int main()
{
    student s;
    s.display();
    return 0;
}
```



```
Enter the RollNo:12
Enter the Name:Nazar
Enter the Fee:2000

12      Nazar   2000
--------------------------------
Process exited after 10.15 seconds with return value 0
Press any key to continue . . .
```

Constructor: characteristics

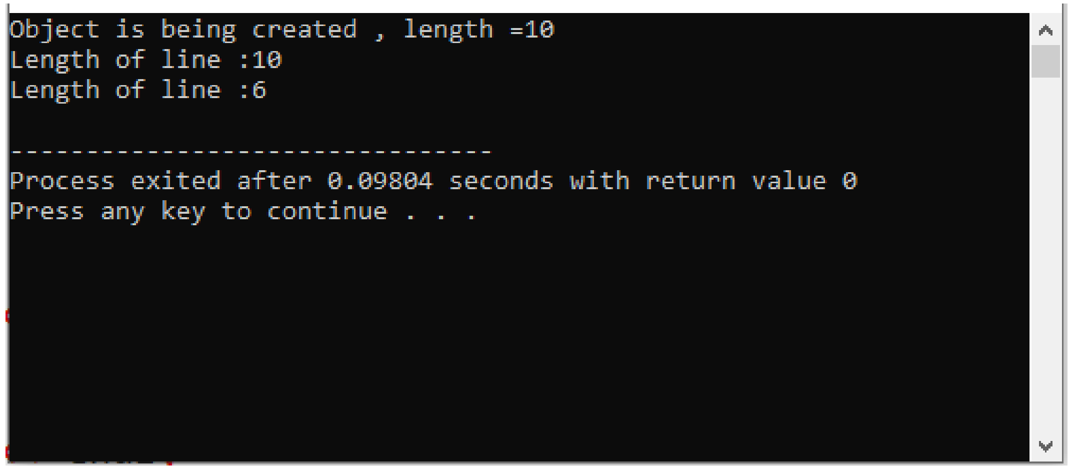
- The name of the constructor is same as its class name.
- Constructors are mostly declared in the public section of the class though it can be declared in the private section of the class.
- Constructors do not return values; hence they do not have a return type.
- A constructor gets called automatically when we create the object of the class.
- Constructors can be overloaded.
- Constructor can not be declared virtual.

How constructors are different from a normal member function?

- 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).

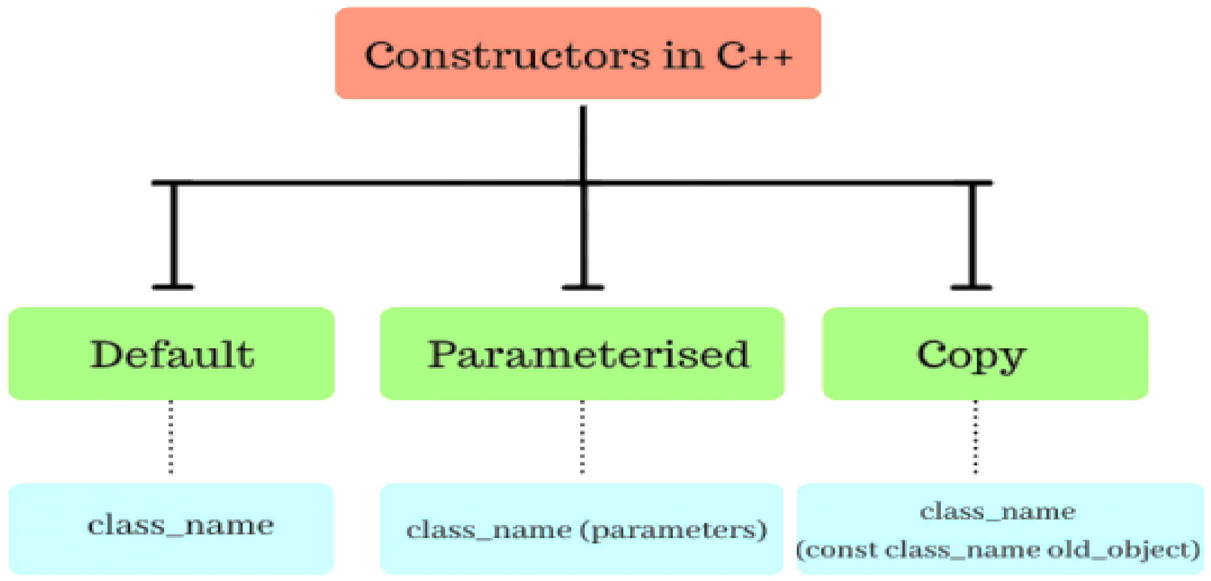
How constructors are different from a normal member function?

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

class Line {
  public:
     void setLength( double len );
     double getLength( void );
     Line( double len ); //This is the constructor
  private:
     double length;
};
//Member function definition including constructor
Line::Line( double len ) {
  cout<<"Object is being created , length ="<< len <<endl;
  length = len;
}
void Line::setLength( double len ) {
  length = len;
}
double Line::getLength( void ) {
  return length;
}
//Main function for the program
int main() {
  Line line(10.0);
  //get initially set length
  cout<<"Length of line :" << line.getLength() << endl;
  //set line length again
  line.setLength(6.0);
  cout<<"Length of line :" << line.getLength() << endl;

  return 0;
}
```



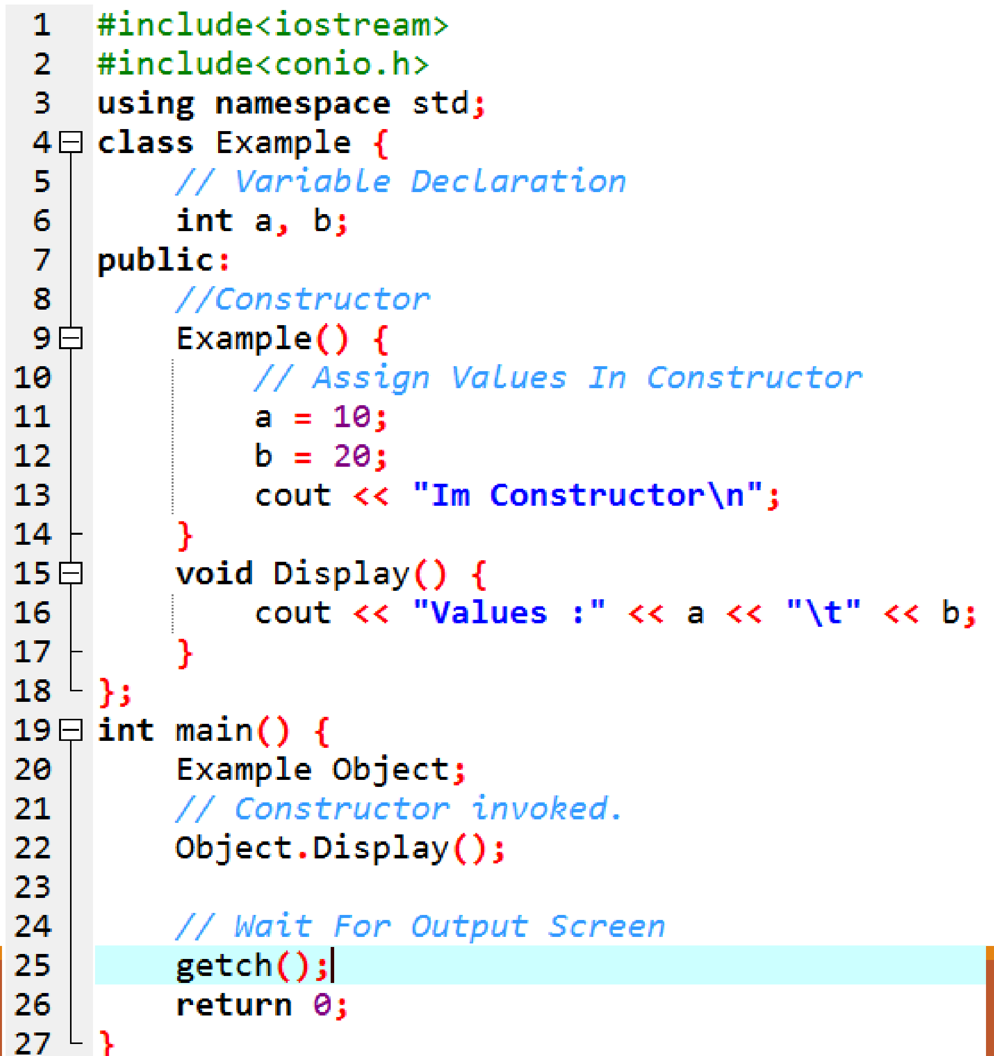
```
Object is being created , length =10
Length of line :10
Length of line :6

--------------------------------
Process exited after 0.09804 seconds with return value 0
Press any key to continue . . .
```

Types of constructors



Default Constructor

- Default constructor is the constructor which doesn’t take any argument.
- It has no parameters.
- Called as zero-argument constructor
- Even if we do not define any constructor explicitly, the compiler will automatically provide a default constructor implicitly.

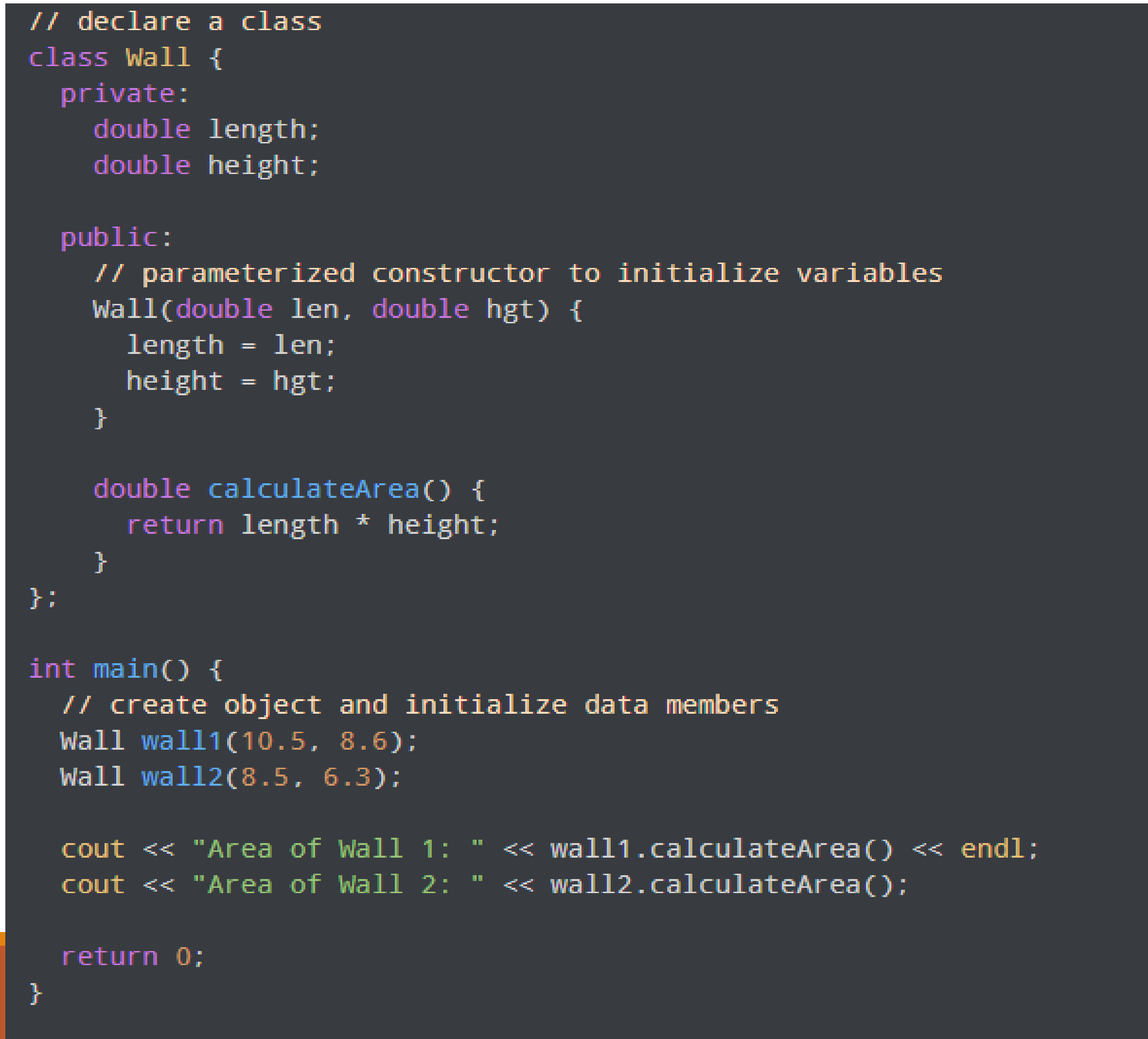


```cpp
#include<iostream>
#include<conio.h>
using namespace std;
class Example {
    // Variable Declaration
    int a, b;
public:
    //Constructor
    Example() {
        // Assign Values In Constructor
        a = 10;
        b = 20;
        cout << "Im Constructor\n";
    }
    void Display() {
        cout << "Values :" << a << "\t" << b;
    }
};
int main() {
    Example Object;
    // Constructor invoked.
    Object.Display();

    // Wait For Output Screen
    getch();
    return 0;
}
```

Parameterized Constructors

- Arguments can be passed to parameterised constructor.
- 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.
- **we can also have more than one constructor in a class and that concept is called constructor overloading.**

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



```
// declare a class
class Wall {
  private:
    double length;
    double height;

  public:
    // parameterized constructor to initialize variables
    Wall(double len, double hgt) {
      length = len;
      height = hgt;
    }

    double calculateArea() {
      return length * height;
    }
};

int main() {
  // create object and initialize data members
  Wall wall1(10.5, 8.6);
  Wall wall2(8.5, 6.3);

  cout << "Area of Wall 1: " << wall1.calculateArea() << endl;
  cout << "Area of Wall 2: " << wall2.calculateArea();

  return 0;
}
```

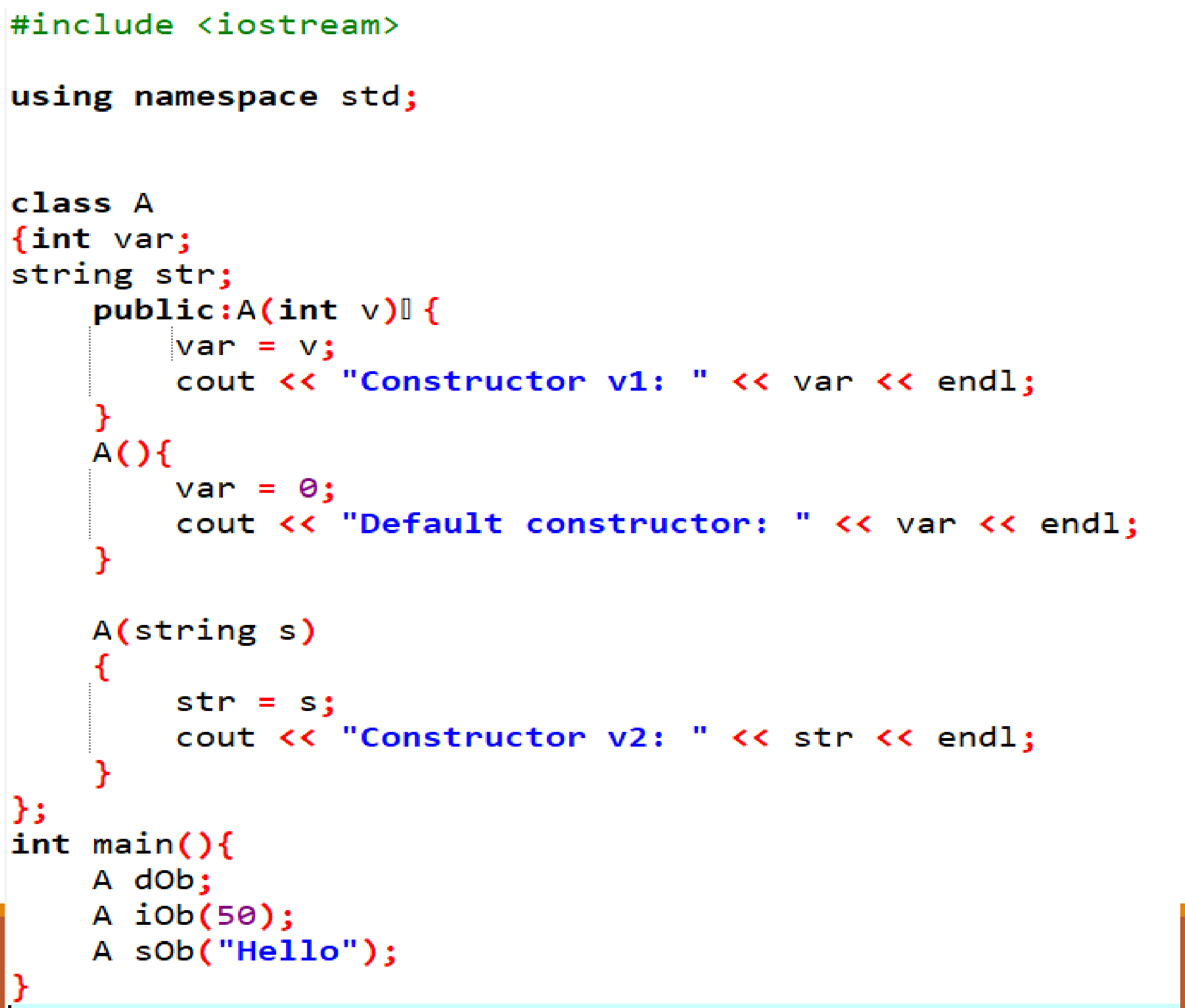
Constructor

- Can we have more than one parameterized constructors of a class?

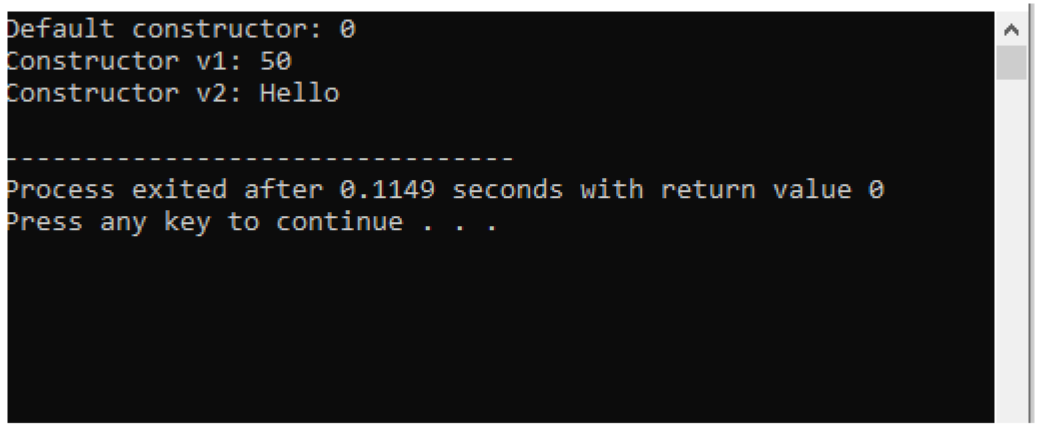
  Yes, we can. But not having the same parameter signature

- If you define any parameterized constructor(s) in the class, C++ will not implicitly create a default constructor for you

Constructor

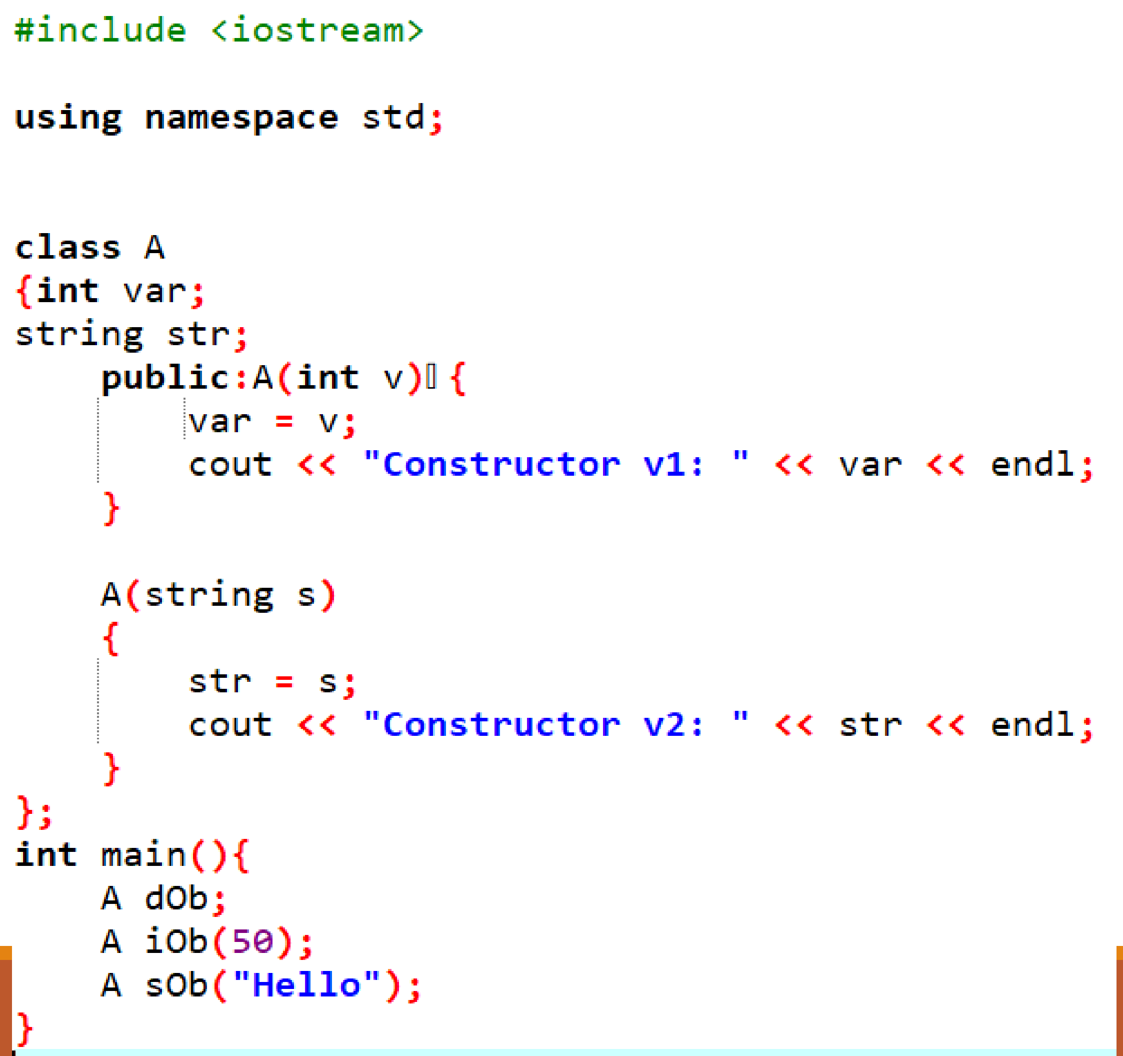


```
#include <iostream>

using namespace std;


class A
{int var;
string str;
    public:A(int v) {
        var = v;
        cout << "Constructor v1: " << var << endl;
    }
    A(){
        var = 0;
        cout << "Default constructor: " << var << endl;
    }

    A(string s)
    {
        str = s;
        cout << "Constructor v2: " << str << endl;
    }
};
int main(){
    A dOb;
    A iOb(50);
    A sOb("Hello");
}
```



```
Default constructor: 0
Constructor v1: 50
Constructor v2: Hello

--------------------------------
Process exited after 0.1149 seconds with return value 0
Press any key to continue . . .
```

Constructor



```
#include <iostream>

using namespace std;


class A
{int var;
string str;
    public:A(int v) {
        var = v;
        cout << "Constructor v1: " << var << endl;
    }

    A(string s)
    {
        str = s;
        cout << "Constructor v2: " << str << endl;
    }
};
int main(){
    A dOb;
    A iOb(50);
    A sOb("Hello");
}
```

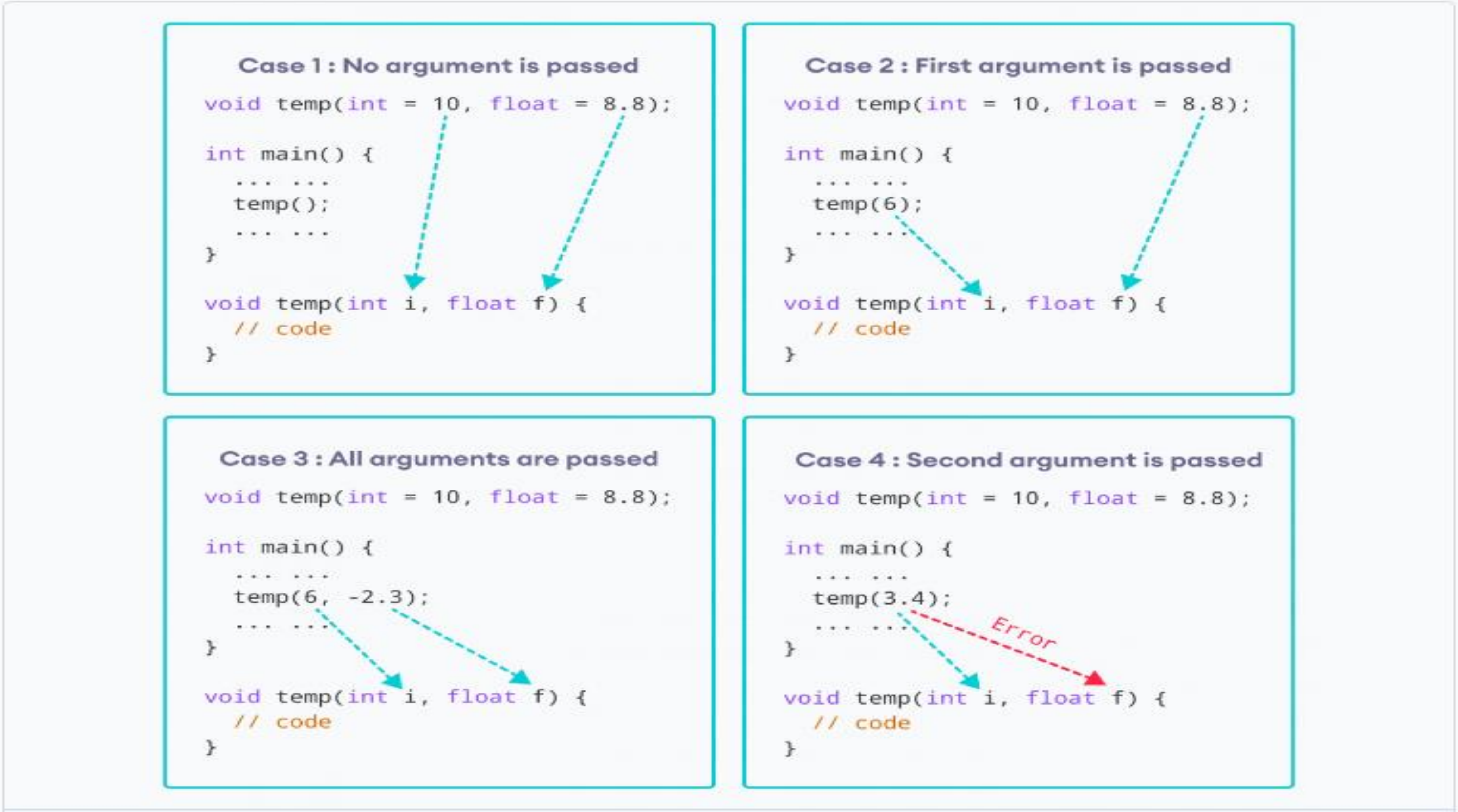
What will be an output?

Default Arguments in C++

A default argument is a value provided in a function declaration that is automatically assigned by the compiler if the caller of the function doesn’t provide a value for the argument with a default value.

Default arguments are overwritten when calling function provides values for them.

Working of default arguments



Case 1 : No argument is passed

```
void temp(int = 10, float = 8.8);

int main() {
  ... ...
  temp();
  ... ...
}

void temp(int i, float f) {
  // code
}
```

Case 2 : First argument is passed

```
void temp(int = 10, float = 8.8);

int main() {
  ... ...
  temp(6);
  ... ...
}

void temp(int i, float f) {
  // code
}
```

Case 3 : All arguments are passed

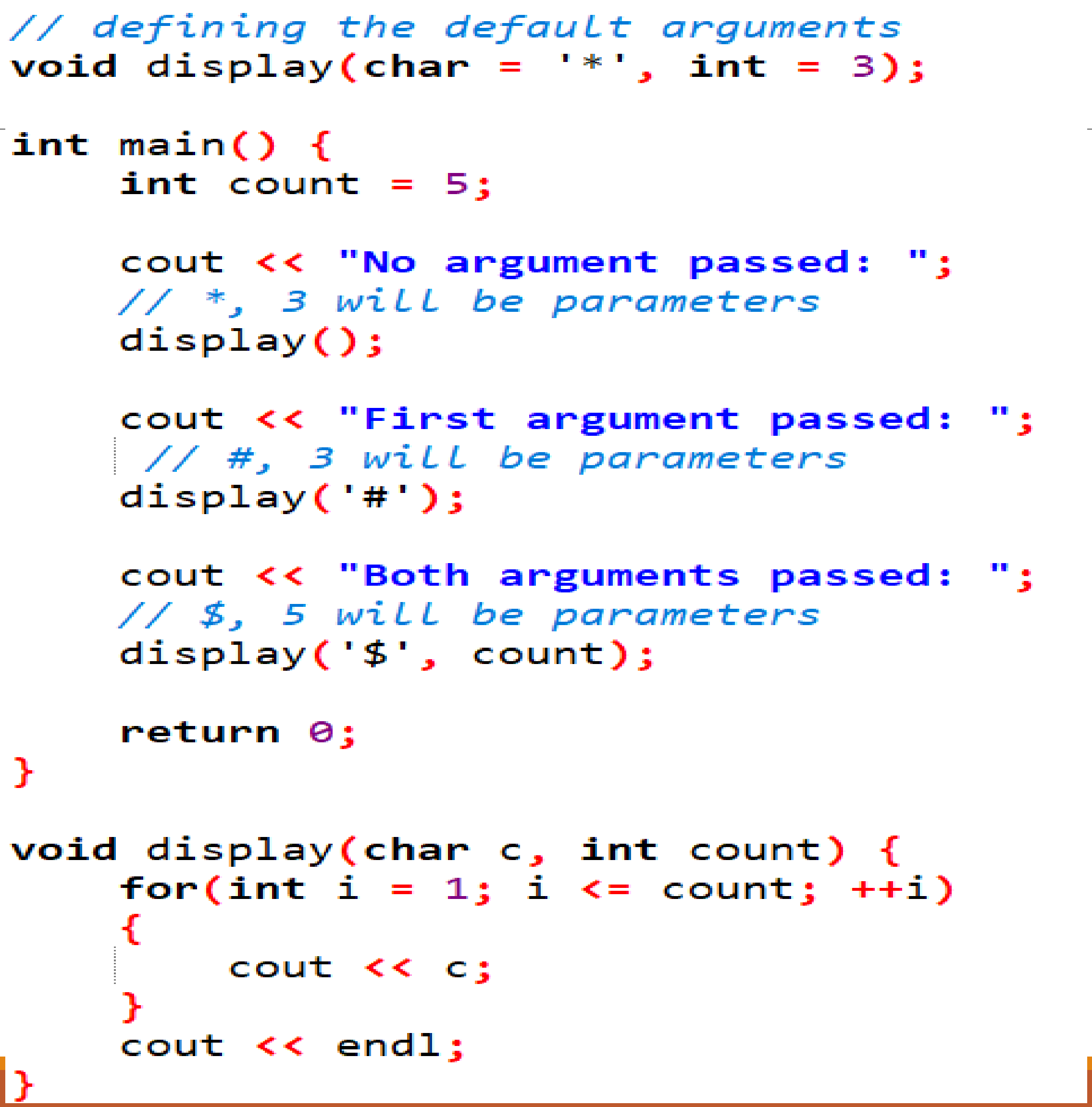
```
void temp(int = 10, float = 8.8);

int main() {
  ... ...
  temp(6, -2.3);
  ... ...
}

void temp(int i, float f) {
  // code
}
```

Case 4 : Second argument is passed

```
void temp(int = 10, float = 8.8);

int main() {
  ... ...
  temp(3.4);
  ... ...
}

void temp(int i, float f) {
  // code
}
```

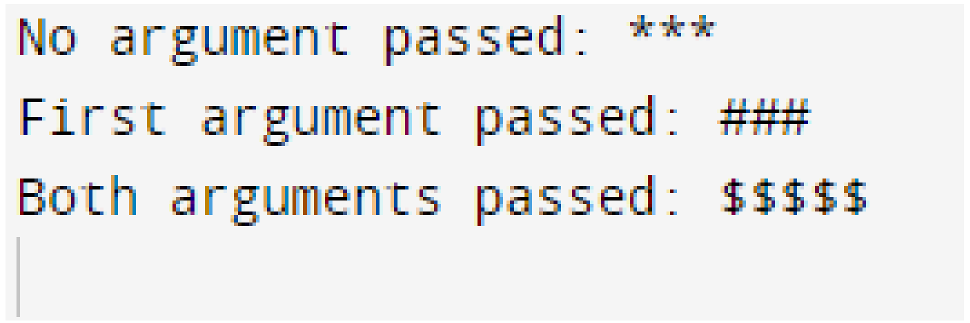
How default arguments work in C++

Example:



```cpp
// defining the default arguments
void display(char = '*', int = 3);

int main() {
    int count = 5;

    cout << "No argument passed: ";
    // *, 3 will be parameters
    display();

    cout << "First argument passed: ";
     // #, 3 will be parameters
    display('#');

    cout << "Both arguments passed: ";
    // $, 5 will be parameters
    display('$', count);

    return 0;
}

void display(char c, int count) {
    for(int i = 1; i <= count; ++i)
    {
        cout << c;
    }
    cout << endl;
}
```

Example: Output



```
No argument passed: ***
First argument passed: ###
Both arguments passed: $$$$$
```

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

- 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.

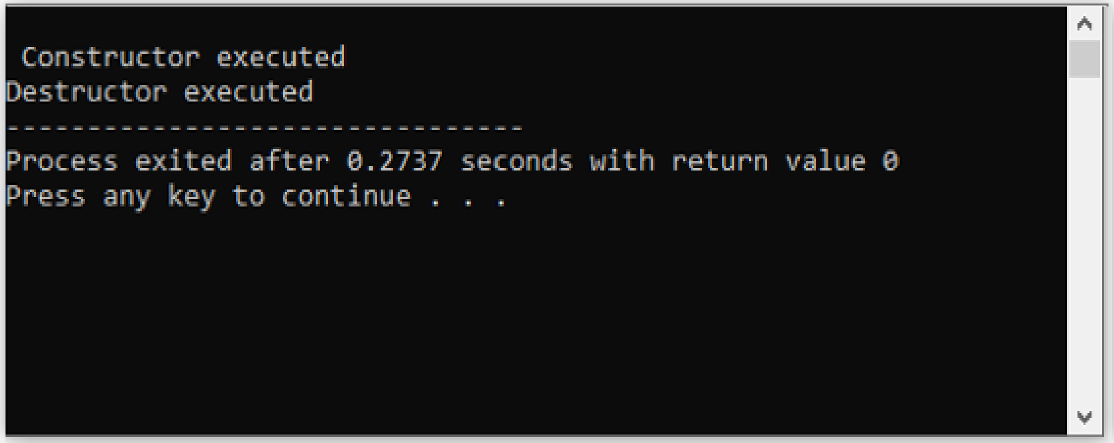
✓ The thing is to be noted here if the object is created by using new or the constructor uses new to allocate memory that resides in the heap memory or the free store, the destructor should use delete to free the memory.

Order of calling Destructors

```
class MyClass
{
   int objectID;

   MyClass(int objectID1)
   {
            objectID = objectID1;
   }

  ~MyClass()
  {
            cout << objectID << " deleted";
  }
}
```

Destructor: Example

```
// C++ program to demonstrate the execution of constructor
// and destructor

#include <iostream>
using namespace std;

class Test {
public:
    // User-Defined Constructor
    Test() { cout << "\n Constructor executed"; }

    // User-Defined Destructor
    ~Test() { cout << "\nDestructor executed"; }
};
main()
{
    Test t;

    return 0;
}
```



```
 Constructor executed
Destructor executed
--------------------------------
Process exited after 0.2737 seconds with return value 0
Press any key to continue . . .
```

Order of calling Destructors

Destructors for nonstatic member objects are called in the reverse **order** in which they appear in the class declaration

When objects store in the stack order will be LIFO

Order of calling Destructors

```
MyClass ob1 (1);


void func()

{

  MyClass ob3 (3);
  MyClass ob4 (4);

}


int main()

{

  MyClass ob2 (2);
  func();
  MyClass ob5 (5);

}
```

Order of calling Destructors

```
MyClass ob1 (1);          \\ destroyed fifth


void func()

{

  MyClass ob3 (3);        \\ destroyed second
  MyClass ob4 (4);        \\ destroyed first

}


int main()

{

  MyClass ob2 (2);        \\ destroyed fourth
  func();
  MyClass ob5 (5);        \\ destroyed third

}
```

What’s the order that local objects are destructed?

In reverse order of construction: First constructed, last destructed.(LIFO)

In the following example, b’s destructor will be executed first, then a’s destructor:

```
void userCode()

{

  Fred a;

  Fred b;

  // ...

}
```

What’s the order that the objects in an array are destructed?

In reverse order of construction: First constructed, last destructed.

In the following example, the order for destructors will be a[9], a[8], …, a[1], a[0]:

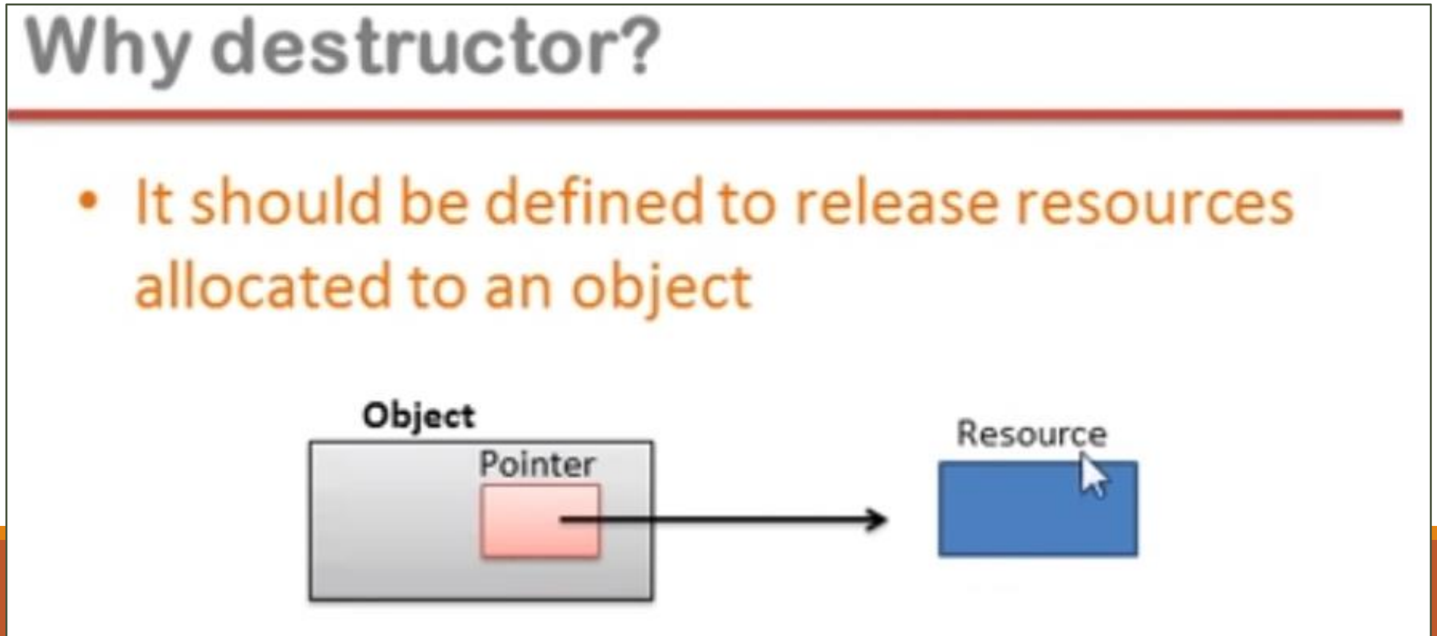
```
void userCode()
{
  Fred a[10];
  // ...
}
```

Why are Destructors useful?

Useful for garbage collection

Garbage-collected languages like JAVA do not have a destructor

- There is no guarantee of when an object will be destroyed



Why destructor?

- It should be defined to release resources allocated to an object

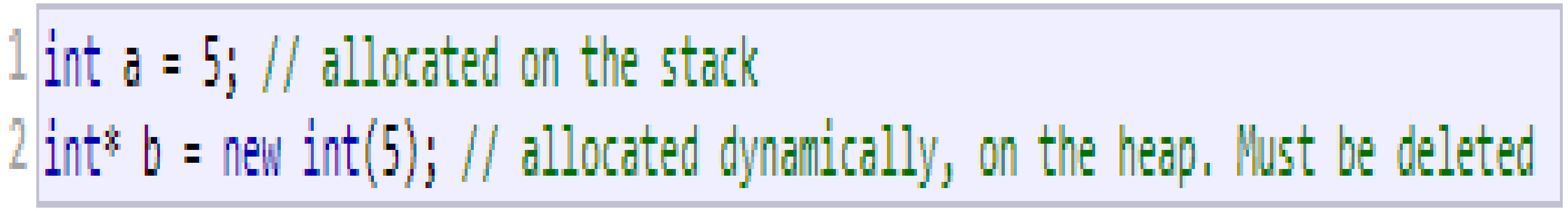
When do we need to write a user-defined destructor?

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.

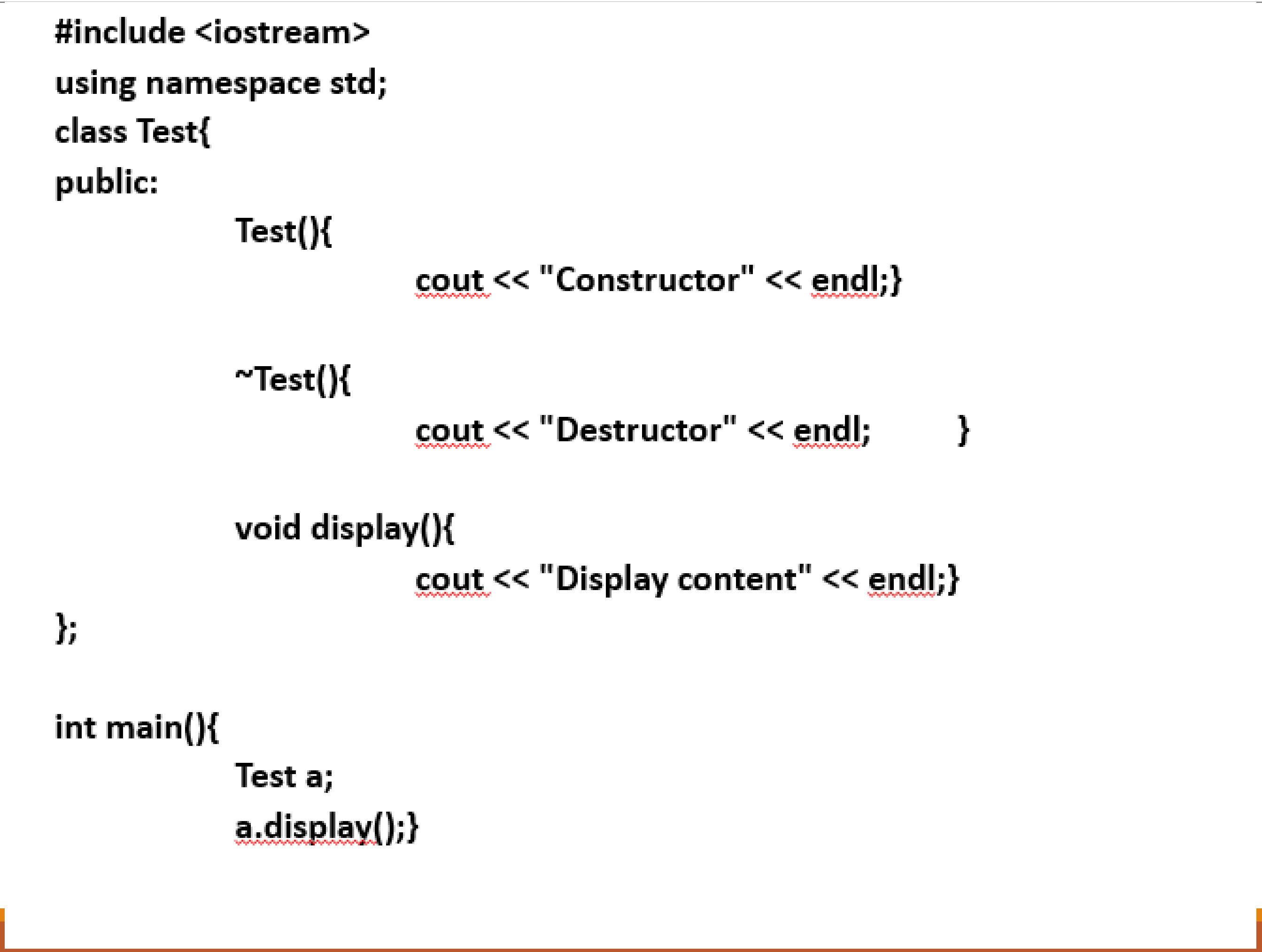
When do we need to write a user-defined destructor?

Only variables allocated on the stack are deallocated automatically when they go out of scope.

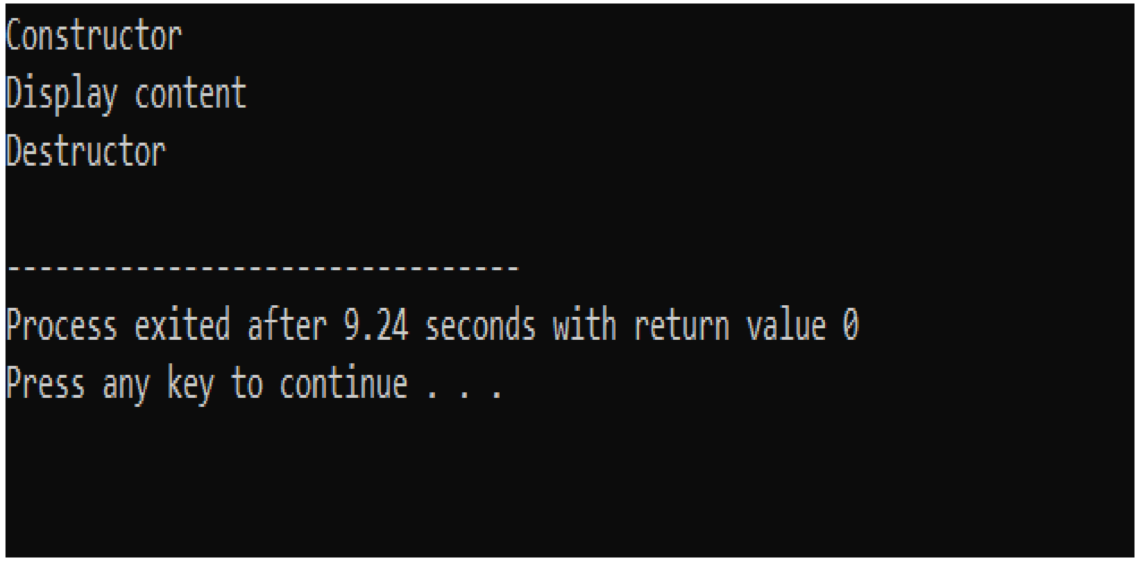
We need to call the destructor if have allocated memory in heap



```
int a = 5; // allocated on the stack
int* b = new int(5); // allocated dynamically, on the heap. Must be deleted
```



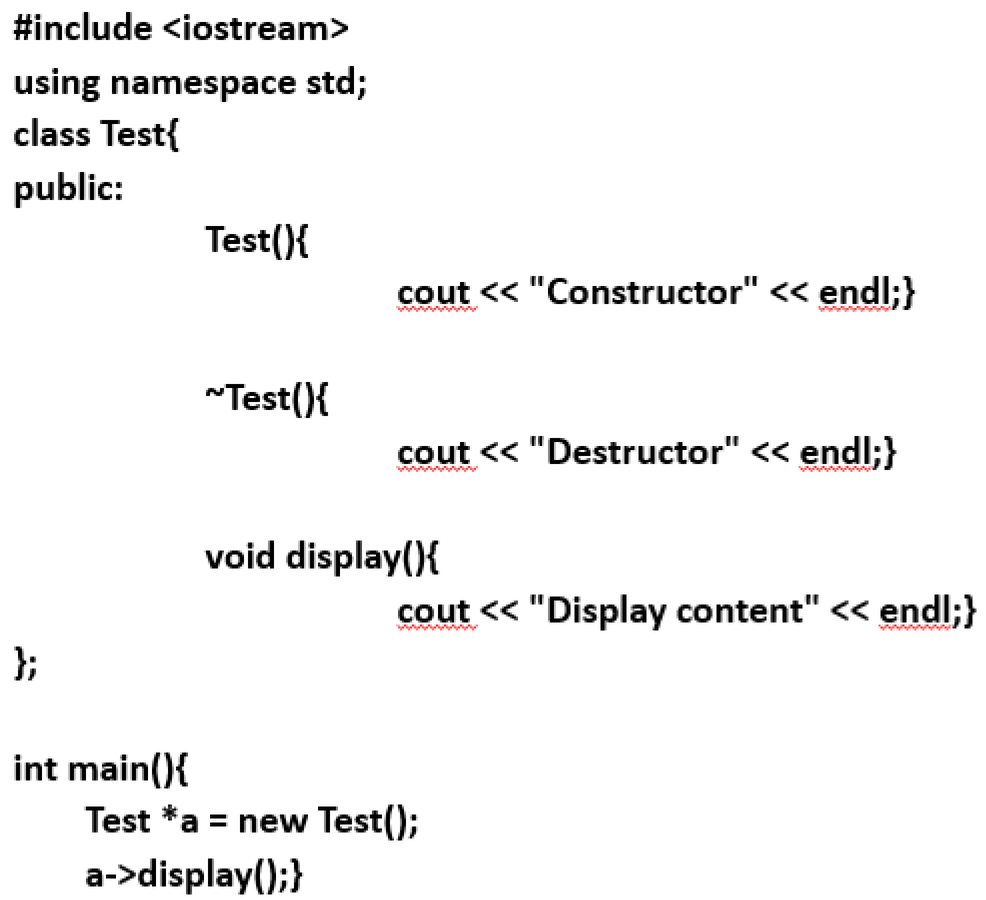
```
#include <iostream>
using namespace std;
class Test{
public:
        Test(){
                cout << "Constructor" << endl;}

        ~Test(){
                cout << "Destructor" << endl;       }

        void display(){
                cout << "Display content" << endl;}
};

int main(){
        Test a;
        a.display();}
```



```
Constructor
Display content
Destructor


--------------------------------
Process exited after 9.24 seconds with return value 0
Press any key to continue . . .
```

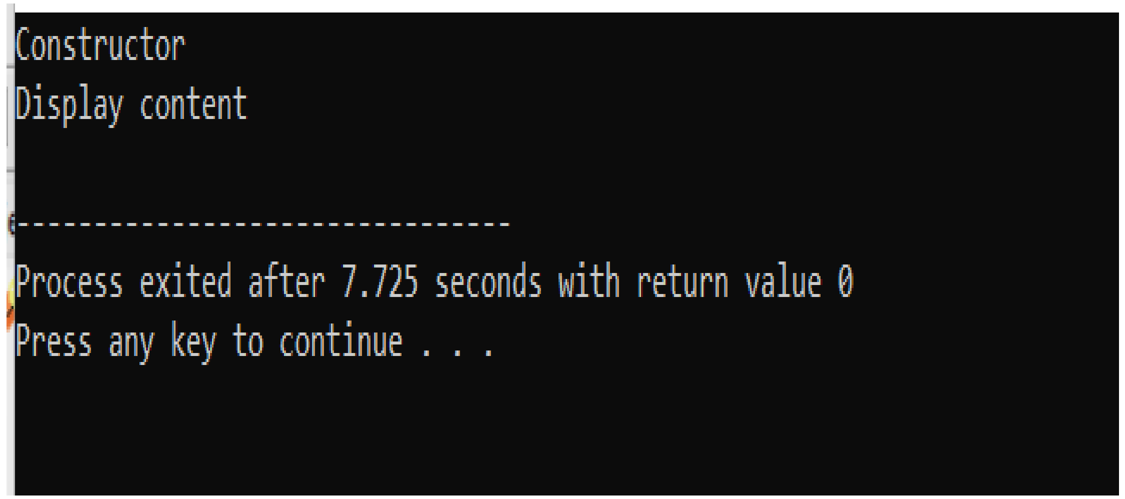
Here we do not want to call the destructor explicitly because we have created the object in the stack, not in heap. So, after object goes out of scope destructor will call automatically.

What if we create the object in heap instead of stack?



```
#include <iostream>
using namespace std;
class Test{
public:
		Test(){
				cout << "Constructor" << endl;}

		~Test(){
				cout << "Destructor" << endl;}

		void display(){
				cout << "Display content" << endl;}
};

int main(){
	Test *a = new Test();
	a->display();}
```

What if we create the object in heap instead of stack?


```
Constructor
Display content


--------------------------------
Process exited after 7.725 seconds with return value 0
Press any key to continue . . .
```

But in this scenario, you can see that it will not call the destructor automatically. So, we have to call it explicitly. Unless memory will not be released at the end of the scope of the object. We can call delete in order to invoke the destructor manually.

What is a memory leak?

- if we continuously allocate memory(dynamically) for a particular object without releasing that memory after completing its task, that memory can not be used in future. Because, although object has completed its task, it is holding that memory because we have not released it properly.
- So, others cannot grab that memory for their executions. This wasted memory can be started in few bytes but eventually, end up in megabytes or even more. This might slow down your application in execution when it finds hard to allocate memory or may be crashed as it failed to allocate enough memory for its execution. So, destructor comes in handy to deallocate memory that it has allocated when the constructor was called.

Exercise

Write a program to print the names of students by creating a Student class. If no name is passed while creating an object of the Student class, then the name should be "Unknown", otherwise the name should be equal to the String value passed while creating the object of the Student class.

(use constructor and destructor)