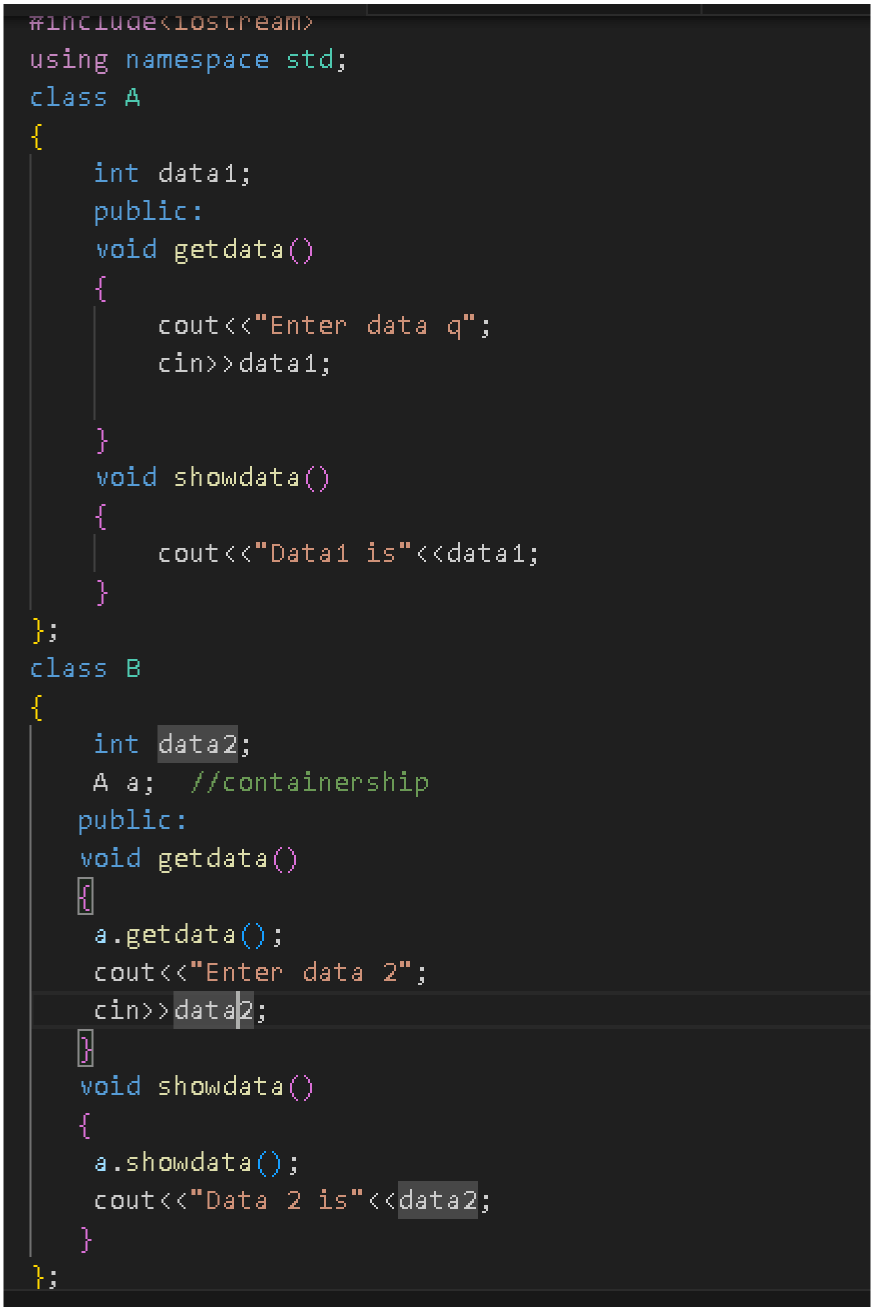
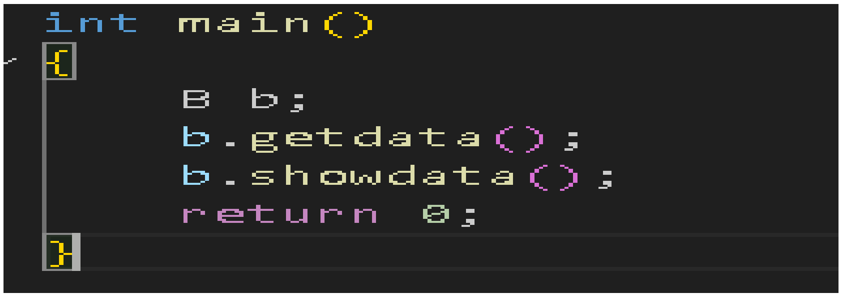
41). What is containership? Show example.

Containership refers to the concept of embedding one object within another. It's a way to create complex data structures by combining simpler ones. This is often referred to as a "has-a" relationship, as one object has another object as part of its composition.



```
#include<iostream>
using namespace std;
class A
{
    int data1;
    public:
    void getdata()
    {
        cout<<"Enter data q";
        cin>>data1;

    }
    void showdata()
    {
        cout<<"Data1 is"<<data1;
    }
};
class B
{
    int data2;
    A a;  //containership
   public:
   void getdata()
   {
    a.getdata();
    cout<<"Enter data 2";
    cin>>data2;
   }
   void showdata()
   {
    a.showdata();
    cout<<"Data 2 is"<<data2;
   }
};
```



```
int main()
{
        B b;
        b.getdata();
        b.showdata();
        return 0;
}
```

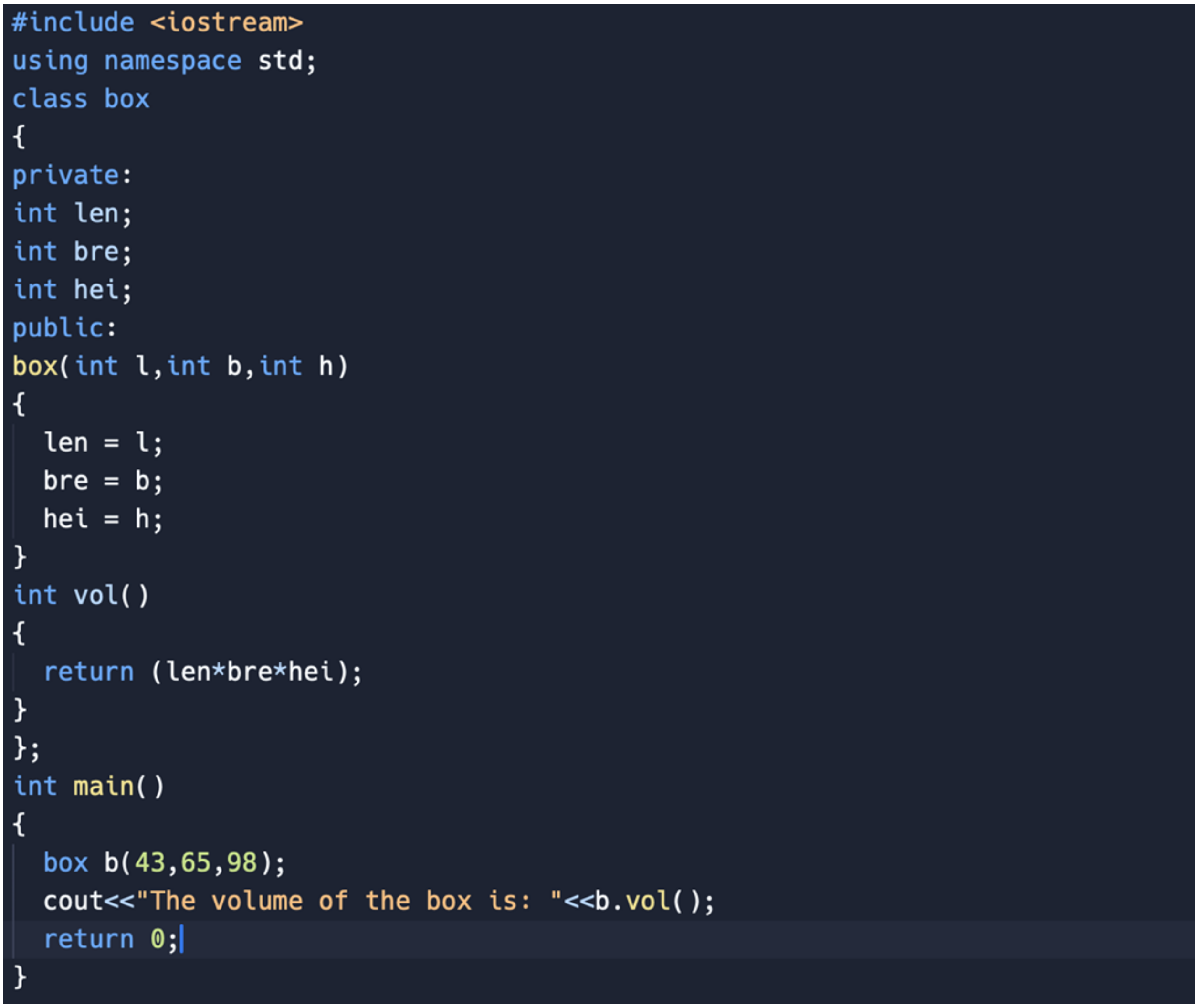
42). What is a constructor? List out its properties. WAP to calculate the volume of the box (Use constructor to initialize the values of l,b,h).

A constructor is a special type of function that is automatically called when an object of a class is created. Its primary purpose is to initialize the object's member variables with appropriate values.

Key characteristics of a constructor:

- It has the same name as the class.
- It doesn't have a return type.
- It's automatically invoked when an object is created.
- It can be overloaded and can’t be virtual.
- It can be parameterized.

Program to calculate the volume of the box (Use constructor to initialize the values of l,b,h).



```
#include <iostream>
using namespace std;
class box
{
private:
int len;
int bre;
int hei;
public:
box(int l,int b,int h)
{
  len = l;
  bre = b;
  hei = h;
}
int vol()
{
  return (len*bre*hei);
}
};
int main()
{
  box b(43,65,98);
  cout<<"The volume of the box is: "<<b.vol();
  return 0;
}
```

43). Why do we need a constructor? Explain its types with examples.

Constructor is needed for the following reasons:

1. Initializing Object State

- **Mandatory initialization:** Ensures that all object members are assigned appropriate initial values before the object is used. This prevents unexpected behavior due to uninitialized data.
- **Default values:** Provides a way to set default values for object properties, making object creation more convenient.
- **Complex initialization:** Handles complex initialization logic that requires multiple steps or calculations.

2. Enforcing Object Invariants

- **Precondition checks:** Verifies that the object is created in a valid state by performing necessary checks in the constructor.
- **Resource allocation:** Acquires required resources (e.g., database connections, file

handles) during object creation.

3. Dependency Injection

- **Passing dependencies:** Allows injecting dependencies into the object through constructor parameters, promoting loose coupling.

4. Builder Pattern

- **Complex object creation:** Facilitates creating complex objects with multiple optional parameters using the builder pattern.

The different types of constructors in C++ are as follows:

1. Default Constructor

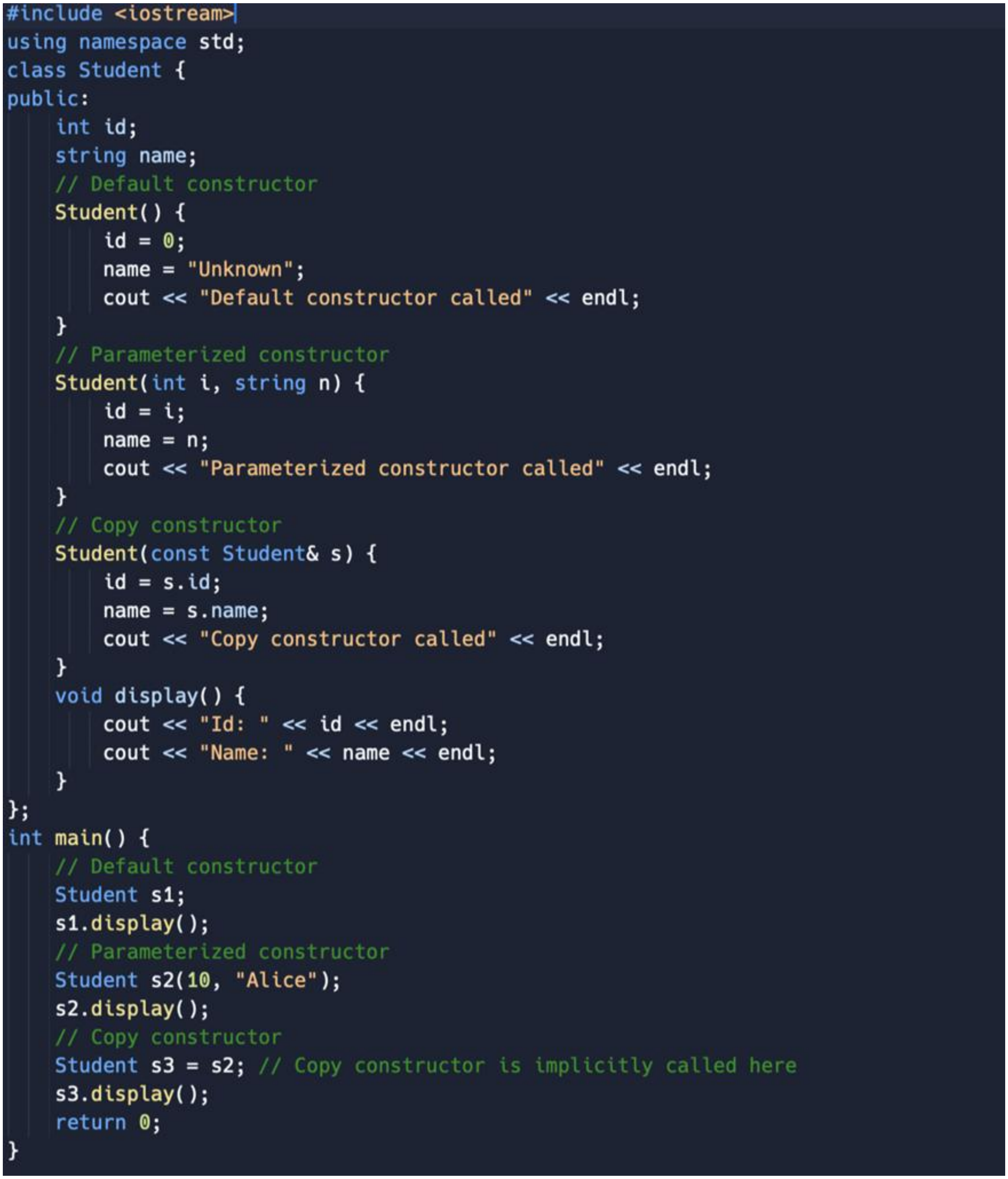
- A constructor with no parameters.
- It is used to initialize object members with default values.
- The compiler automatically generates a default constructor if no other constructor is defined.

2. Parameterized Constructor

- A constructor with parameters.
- It is used to initialize object members with specific values provided as arguments.
- Multiple parameterized constructors can be defined with different parameter lists (constructor overloading).

3. Copy Constructor

- A constructor that takes a reference to another object of the same class as a parameter.
- It is used to create a copy of an existing object.
- Important for deep copying to avoid shallow copy issues.



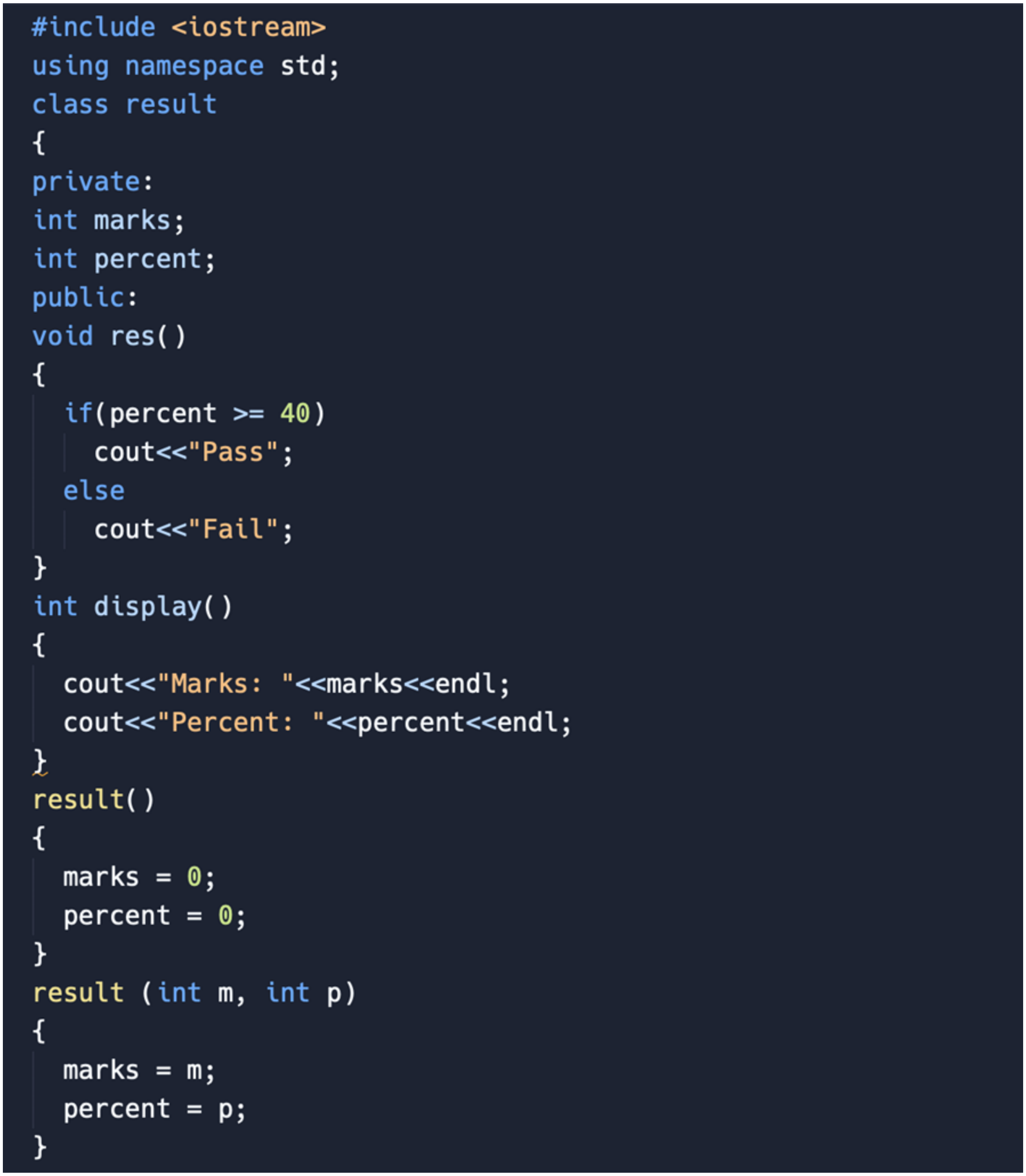
```
#include <iostream>
using namespace std;
class Student {
public:
    int id;
    string name;
    // Default constructor
    Student() {
        id = 0;
        name = "Unknown";
        cout << "Default constructor called" << endl;
    }
    // Parameterized constructor
    Student(int i, string n) {
        id = i;
        name = n;
        cout << "Parameterized constructor called" << endl;
    }
    // Copy constructor
    Student(const Student& s) {
        id = s.id;
        name = s.name;
        cout << "Copy constructor called" << endl;
    }
    void display() {
        cout << "Id: " << id << endl;
        cout << "Name: " << name << endl;
    }
};
int main() {
    // Default constructor
    Student s1;
    s1.display();
    // Parameterized constructor
    Student s2(10, "Alice");
    s2.display();
    // Copy constructor
    Student s3 = s2; // Copy constructor is implicitly called here
    s3.display();
    return 0;
}
```

44). What is a copy constructor? WAP to show use of copy constructor.

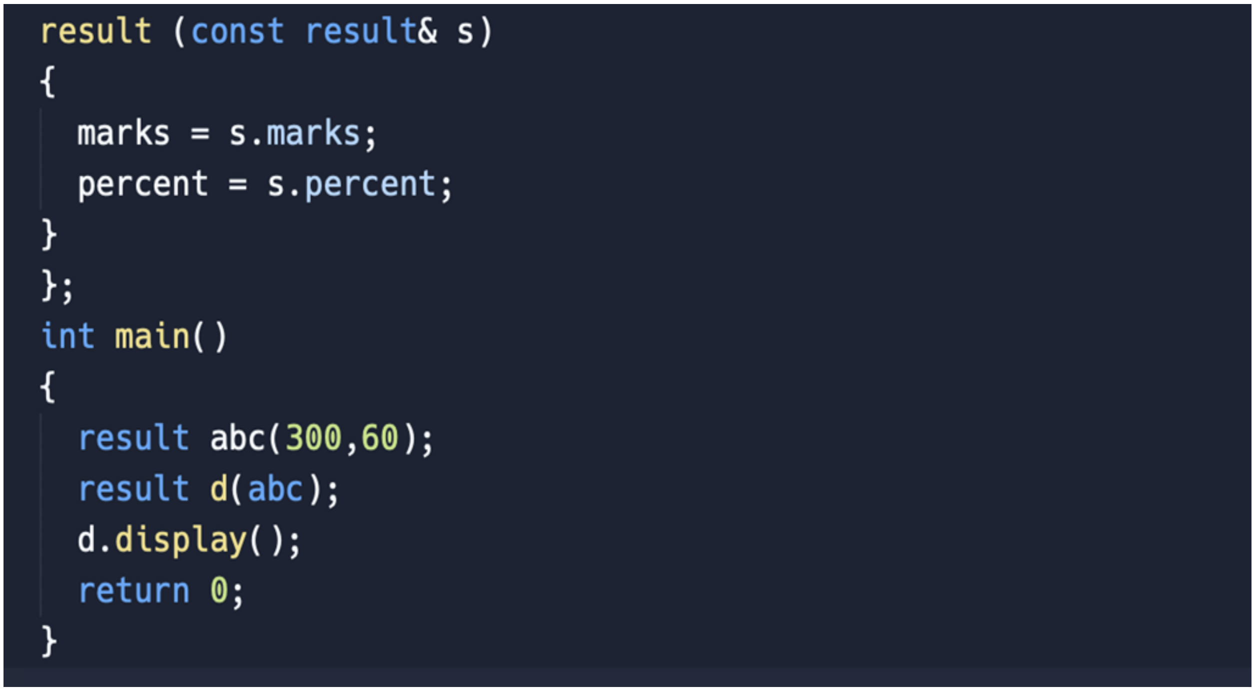
A copy constructor is a special member function that creates a new object as a copy of an existing object of the same class. It's invoked when:

- An object is passed by value to a function.

- An object is returned by value from a function.
- An object is initialized with another object of the same type.

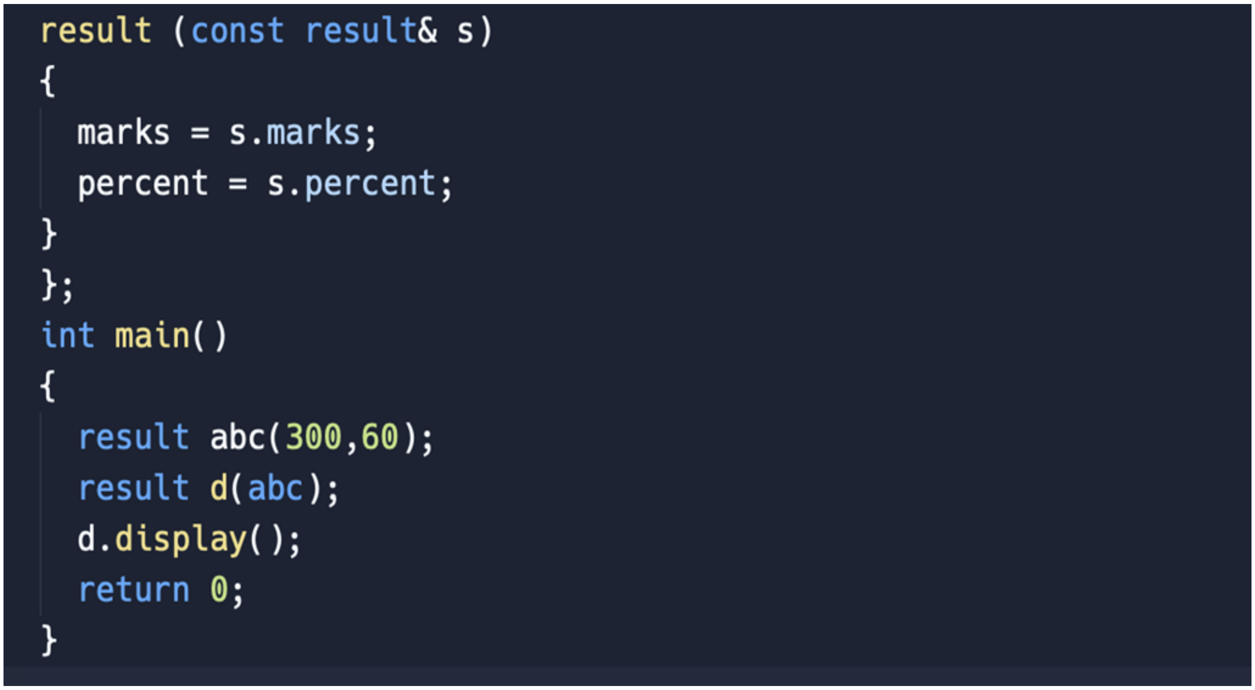


```
#include <iostream>
using namespace std;
class result
{
private:
int marks;
int percent;
public:
void res()
{
  if(percent >= 40)
    cout<<"Pass";
  else
    cout<<"Fail";
}
int display()
{
  cout<<"Marks: "<<marks<<endl;
  cout<<"Percent: "<<percent<<endl;
}
result()
{
  marks = 0;
  percent = 0;
}
result (int m, int p)
{
  marks = m;
  percent = p;
}
```



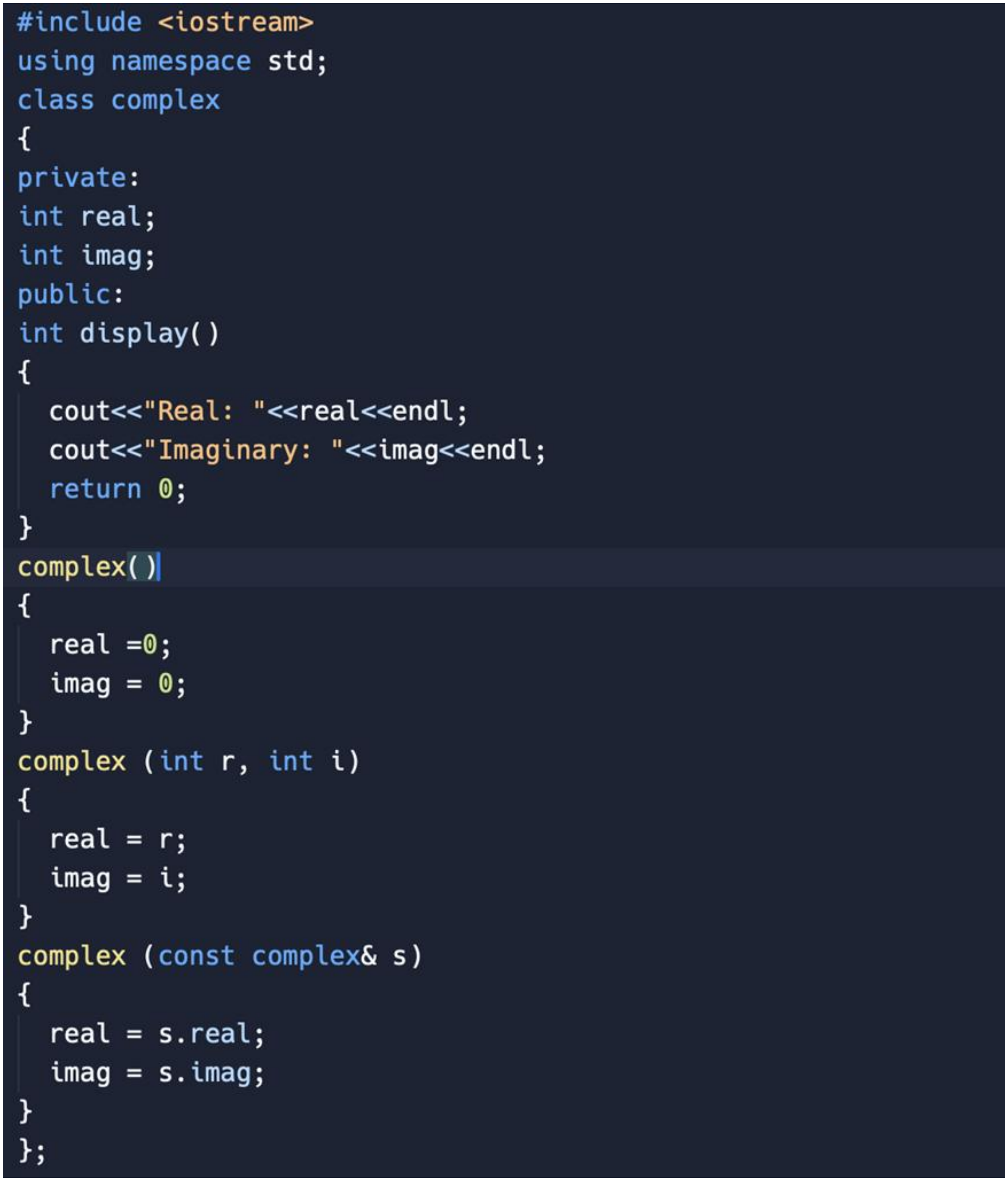
```
result (const result& s)
{
  marks = s.marks;
  percent = s.percent;
}
};
int main()
{
  result abc(300,60);
  result d(abc);
  d.display();
  return 0;
}
```

45). WAP to initialize an object of class with parameterized constructor and copy this object into another object into another object using copy constructor.

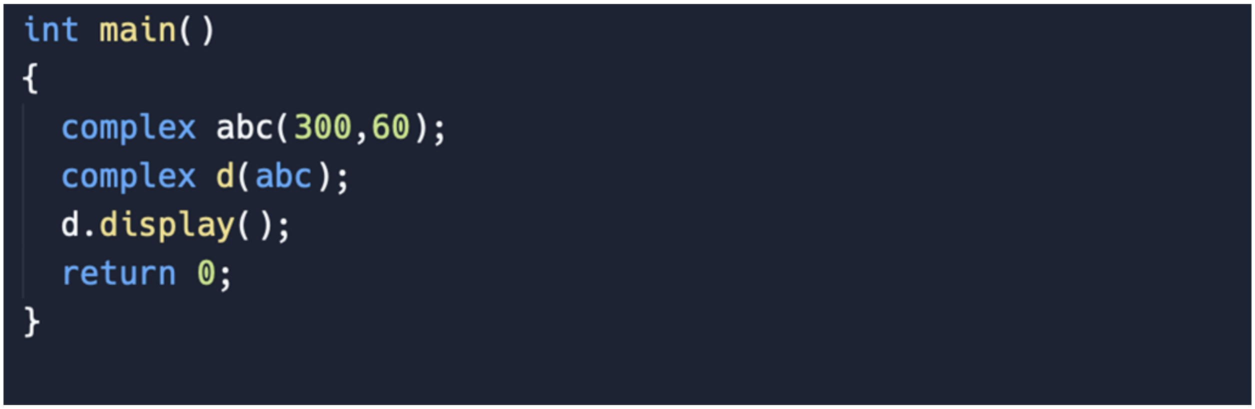


```
result (const result& s)
{
  marks = s.marks;
  percent = s.percent;
}
};
int main()
{
  result abc(300,60);
  result d(abc);
  d.display();
  return 0;
}
```

45). WAP to initialize an object of class with parameterized constructor and copy this object into another object into another object using copy constructor.



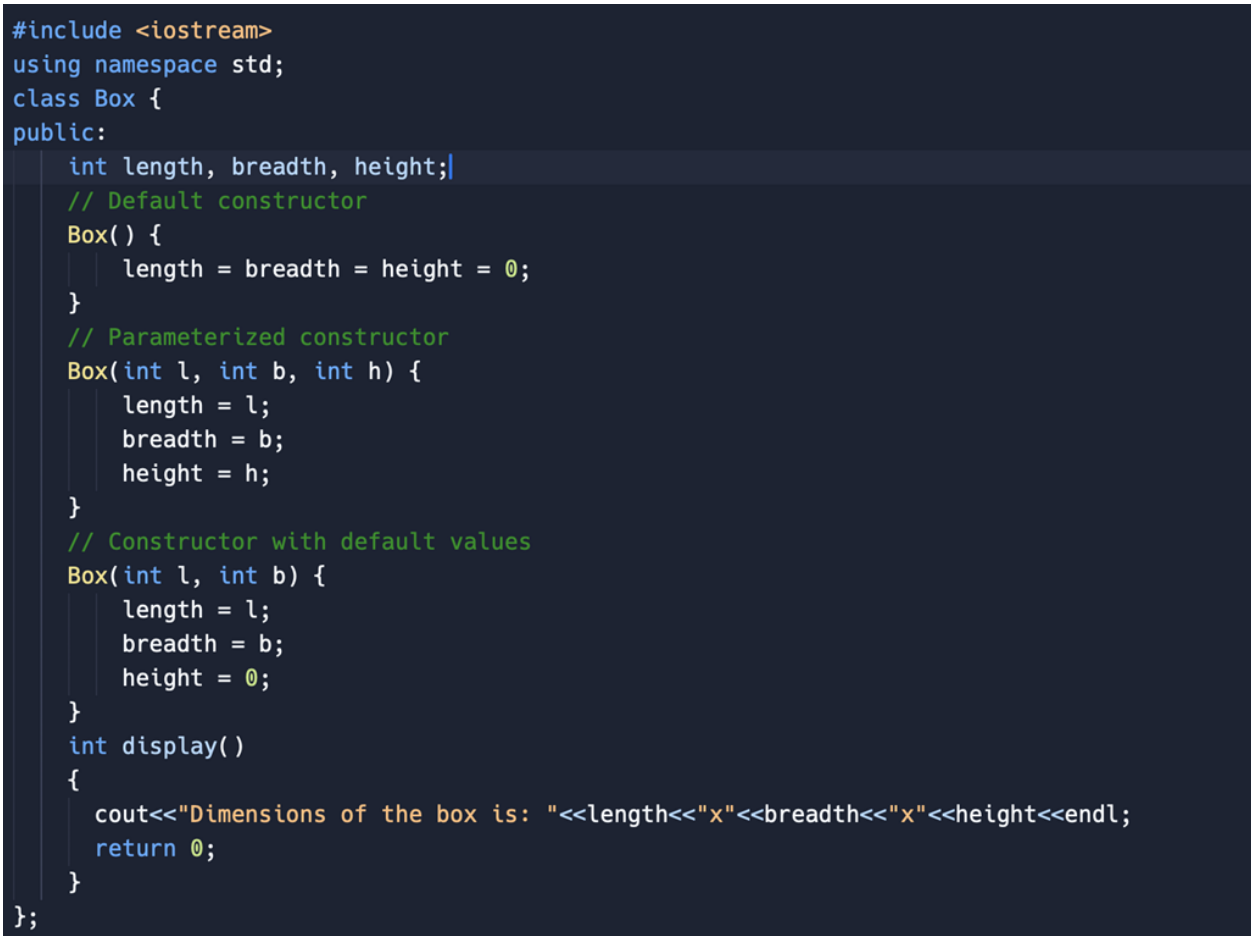
```cpp
#include <iostream>
using namespace std;
class complex
{
private:
int real;
int imag;
public:
int display()
{
  cout<<"Real: "<<real<<endl;
  cout<<"Imaginary: "<<imag<<endl;
  return 0;
}
complex()
{
  real =0;
  imag = 0;
}
complex (int r, int i)
{
  real = r;
  imag = i;
}
complex (const complex& s)
{
  real = s.real;
  imag = s.imag;
}
};
```



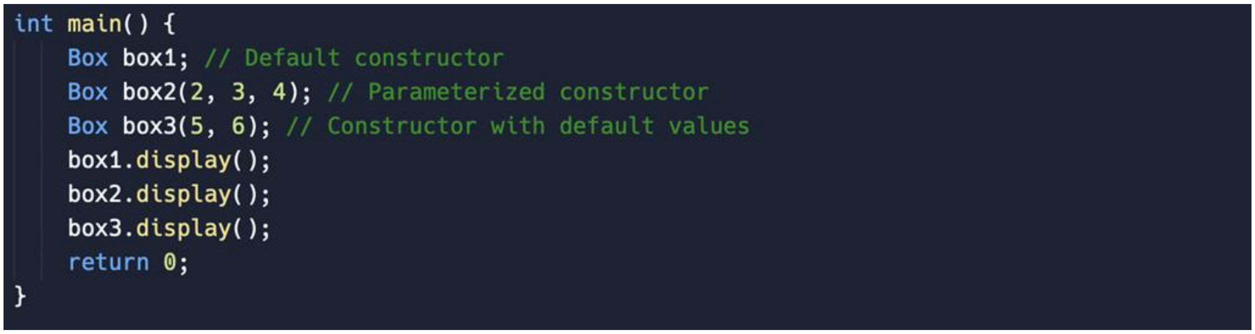
```
int main()
{
   complex abc(300,60);
   complex d(abc);
   d.display();
   return 0;
}
```

46). What is constructor overloading? Show What is constructor overloading? Show with an example.

Constructor overloading is a feature in object-oriented programming that allows a class to have multiple constructors with different parameter lists. This enables you to create objects of the same class with different initializations.



```
#include <iostream>
using namespace std;
class Box {
public:
    int length, breadth, height;
    // Default constructor
    Box() {
        length = breadth = height = 0;
    }
    // Parameterized constructor
    Box(int l, int b, int h) {
        length = l;
        breadth = b;
        height = h;
    }
    // Constructor with default values
    Box(int l, int b) {
        length = l;
        breadth = b;
        height = 0;
    }
    int display()
    {
      cout<<"Dimensions of the box is: "<<length<<"x"<<breadth<<"x"<<height<<endl;
      return 0;
    }
};
```



```
int main() {
    Box box1; // Default constructor
    Box box2(2, 3, 4); // Parameterized constructor
    Box box3(5, 6); // Constructor with default values
    box1.display();
    box2.display();
    box3.display();
    return 0;
}
```

47). What is a destructor? Why do we need it? List its characteristics. WAP to show destructors.

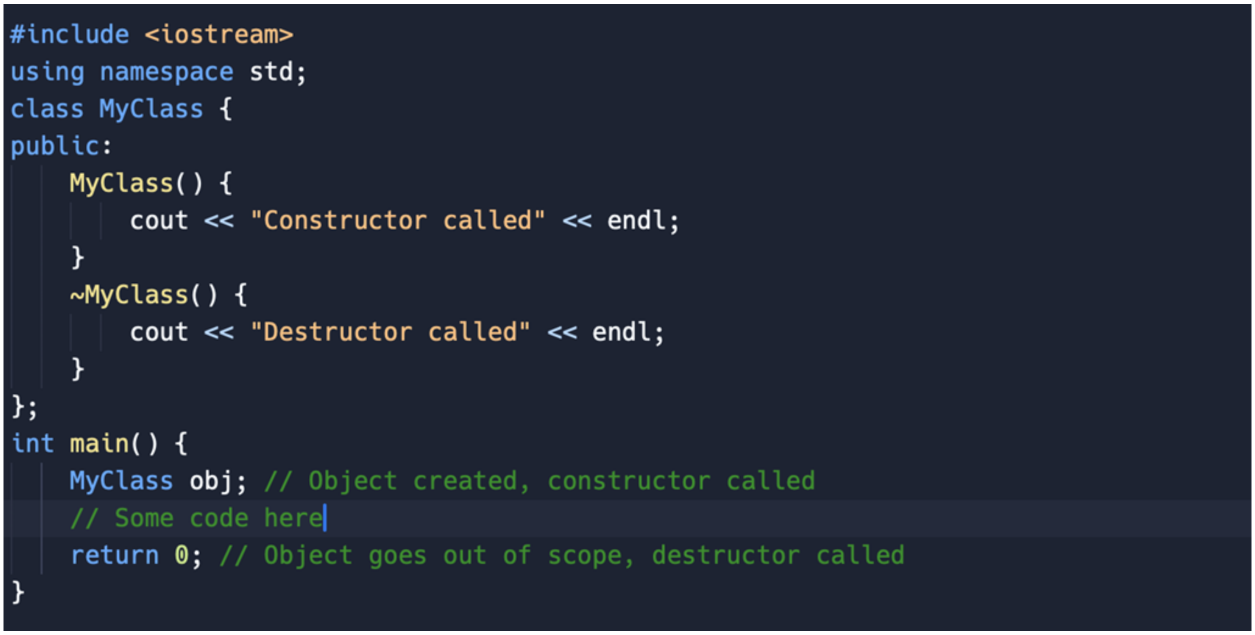
A destructor is a special member function of a class that is automatically invoked when an object of that class goes out of scope or is explicitly deleted. It's used to perform cleanup tasks before the object is destroyed.

We need destructors for the following reasons:

- To deallocate dynamically allocated memory.
- To close files or network connections.
- To perform other cleanup tasks before an object is destroyed.

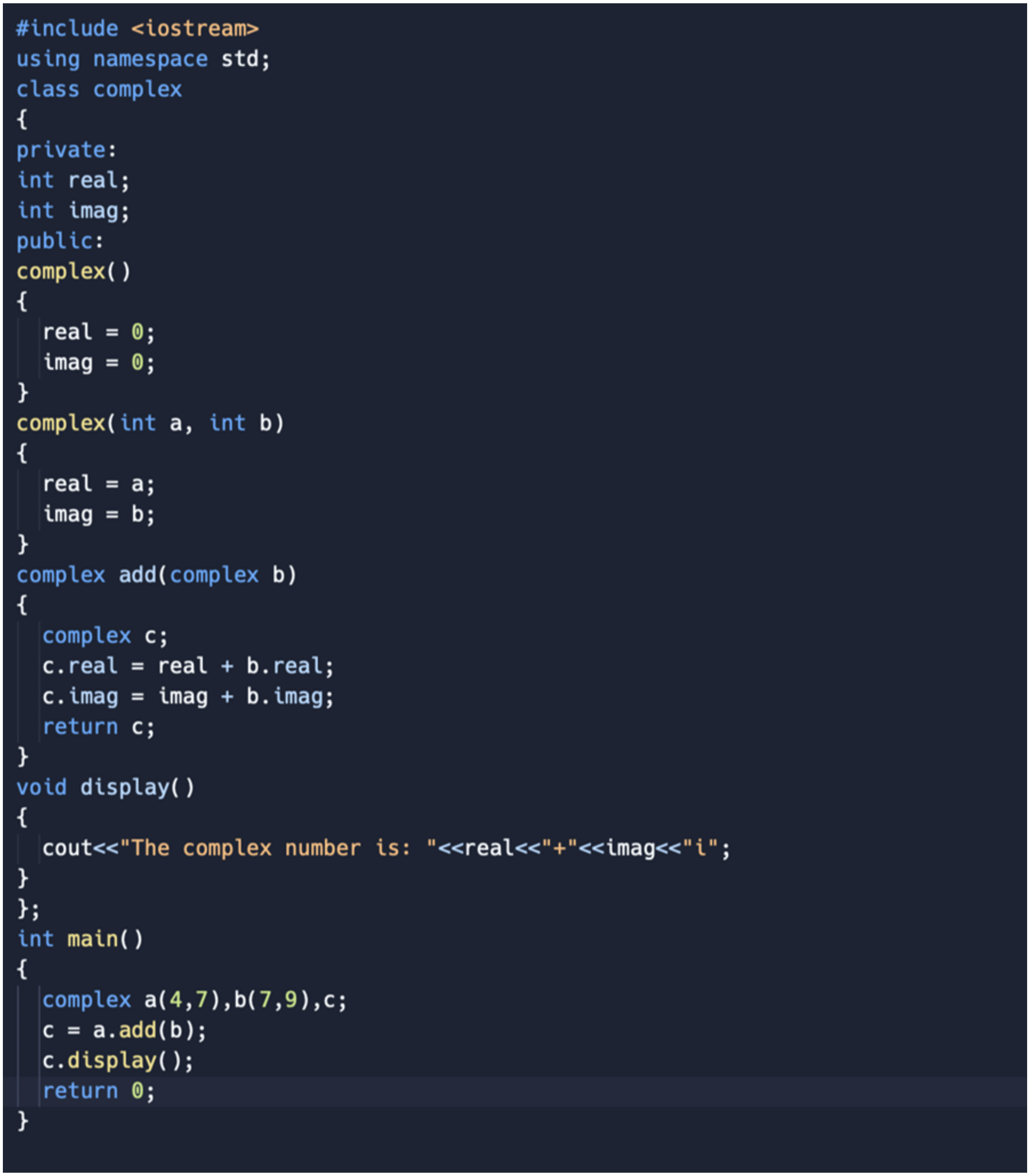
Key points:

- A destructor has the same name as the class, but it's preceded by a tilde (~).
- It doesn't take any parameters and has no return type.
- The compiler automatically generates a default destructor if you don't define one. However, this might not be sufficient for classes with dynamic memory allocation.



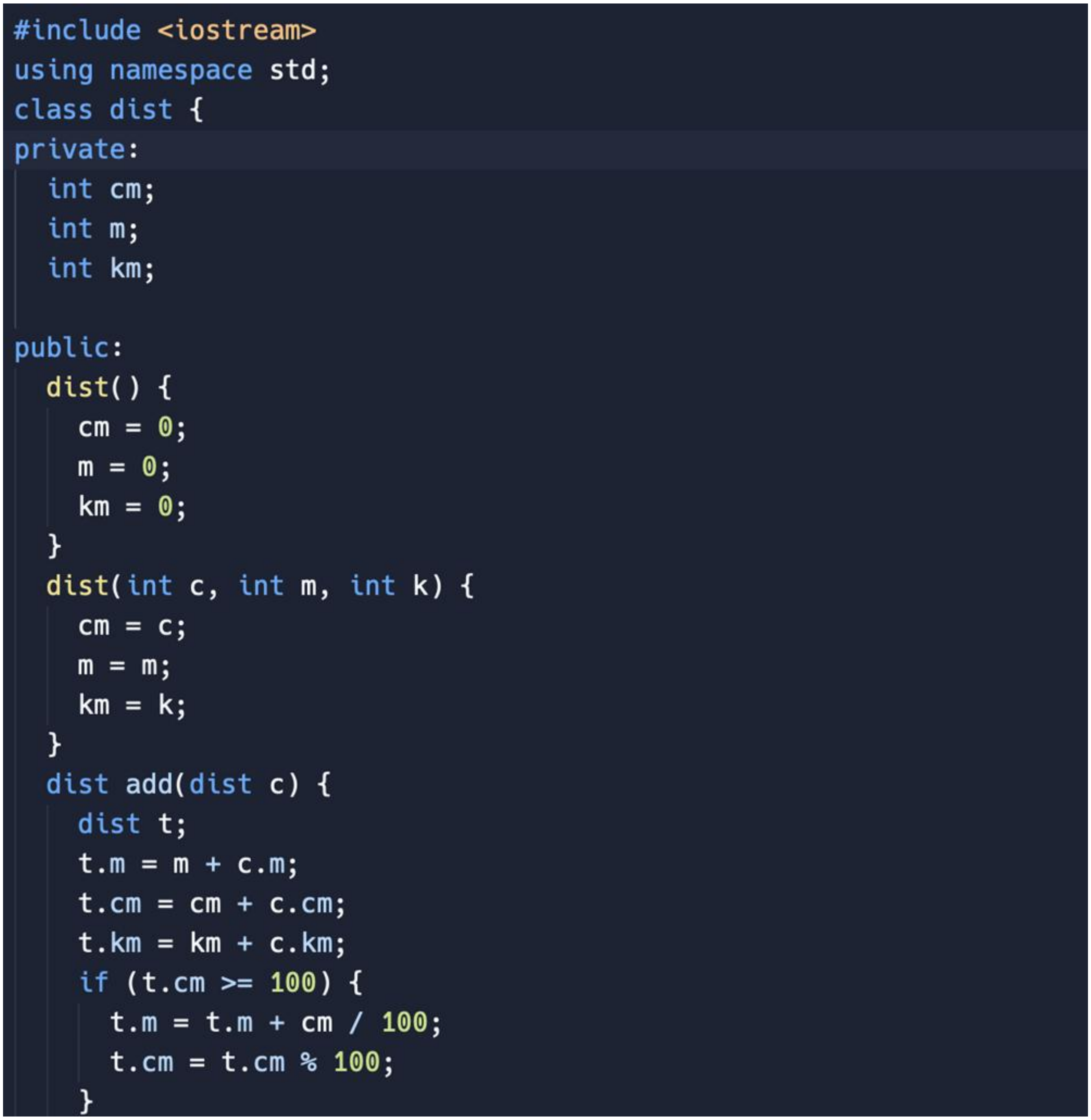
```
#include <iostream>
using namespace std;
class MyClass {
public:
    MyClass() {
        cout << "Constructor called" << endl;
    }
    ~MyClass() {
        cout << "Destructor called" << endl;
    }
};
int main() {
    MyClass obj; // Object created, constructor called
    // Some code here
    return 0; // Object goes out of scope, destructor called
}
```

48). Create a class called complex with data members real and imaginary. Initialize all the data members using the constructor and use necessary member functions to add two complex numbers.



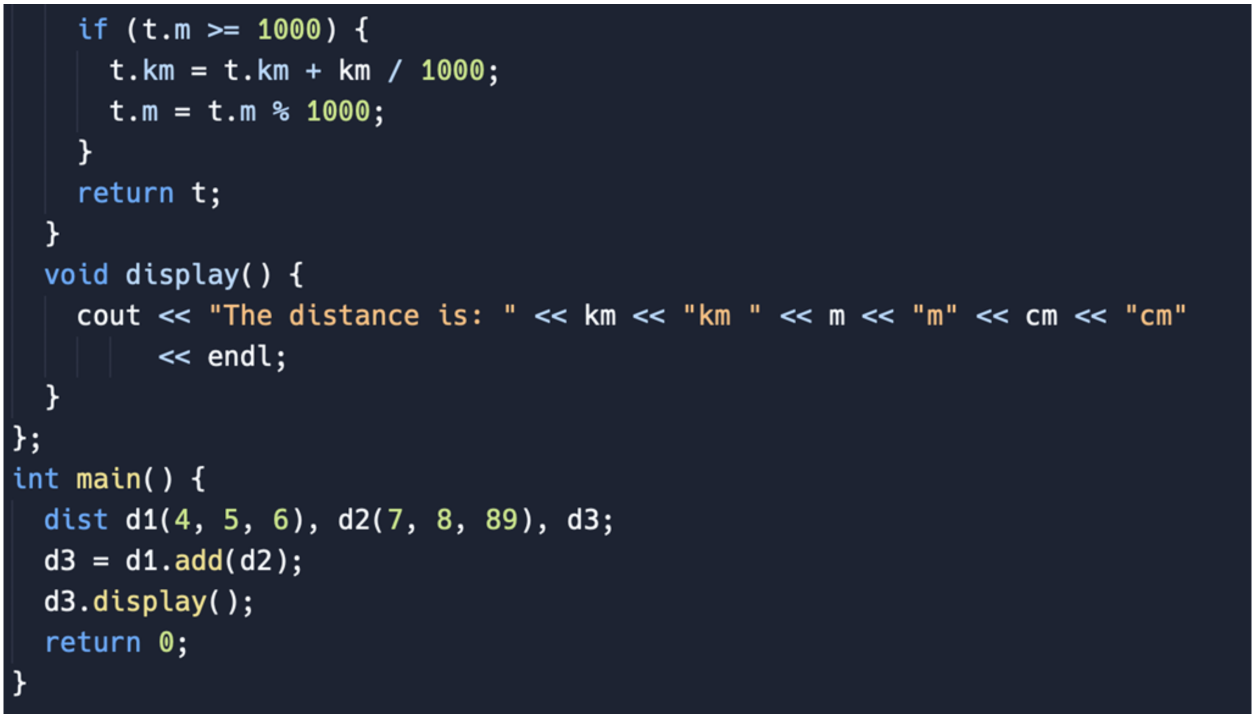
```
#include <iostream>
using namespace std;
class complex
{
private:
int real;
int imag;
public:
complex()
{
  real = 0;
  imag = 0;
}
complex(int a, int b)
{
  real = a;
  imag = b;
}
complex add(complex b)
{
  complex c;
  c.real = real + b.real;
  c.imag = imag + b.imag;
  return c;
}
void display()
{
  cout<<"The complex number is: "<<real<<"+"<<imag<<"i";
}
};
int main()
{
  complex a(4,7),b(7,9),c;
  c = a.add(b);
  c.display();
  return 0;
}
```

49). Create a class called distance with data members cm, m and km. Initialize all the data members using constructor and use necessary member functions to add two distances and display results.



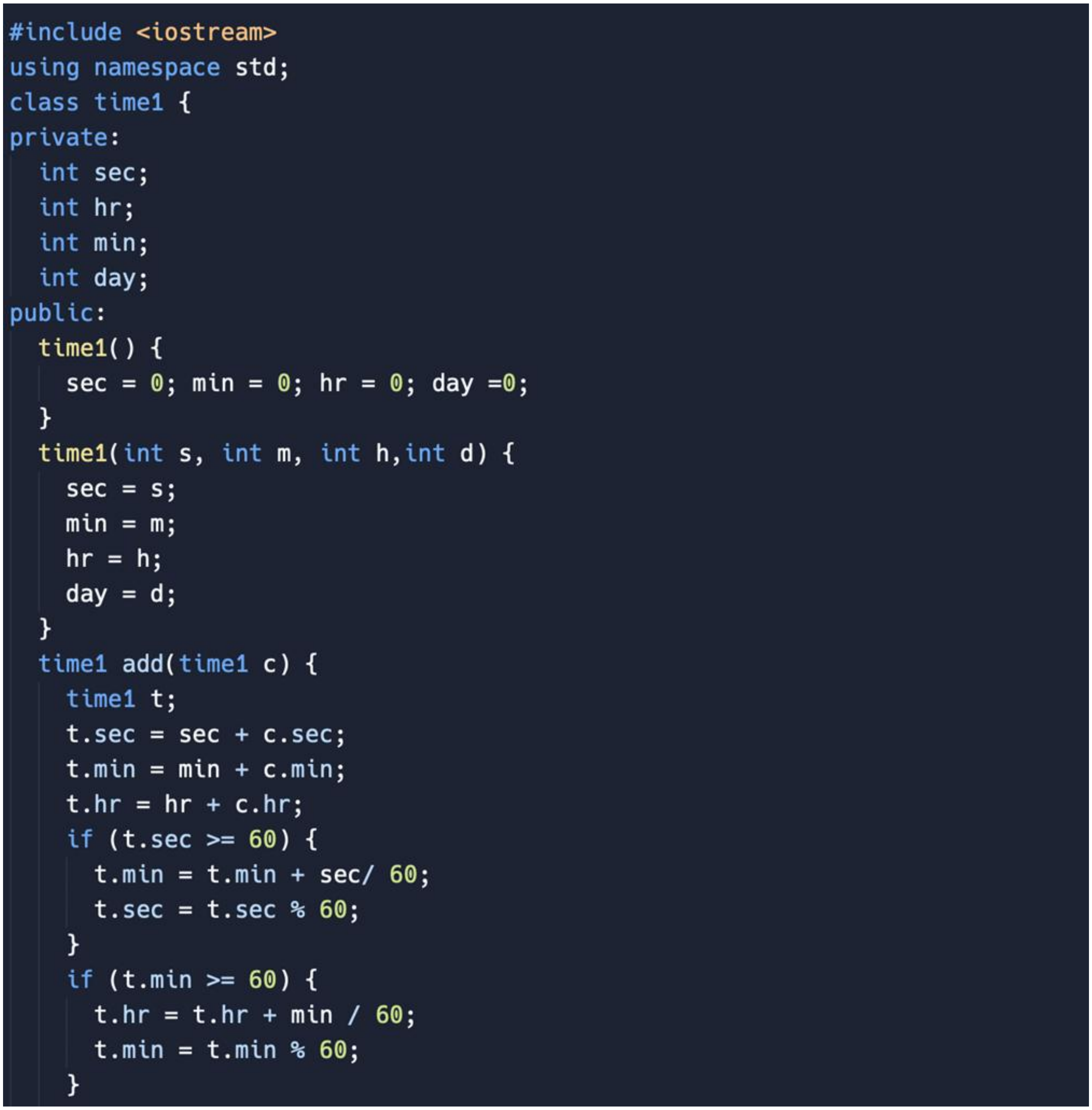
```cpp
#include <iostream>
using namespace std;
class dist {
private:
  int cm;
  int m;
  int km;

public:
  dist() {
    cm = 0;
    m = 0;
    km = 0;
  }
  dist(int c, int m, int k) {
    cm = c;
    m = m;
    km = k;
  }
  dist add(dist c) {
    dist t;
    t.m = m + c.m;
    t.cm = cm + c.cm;
    t.km = km + c.km;
    if (t.cm >= 100) {
      t.m = t.m + cm / 100;
      t.cm = t.cm % 100;
    }
```

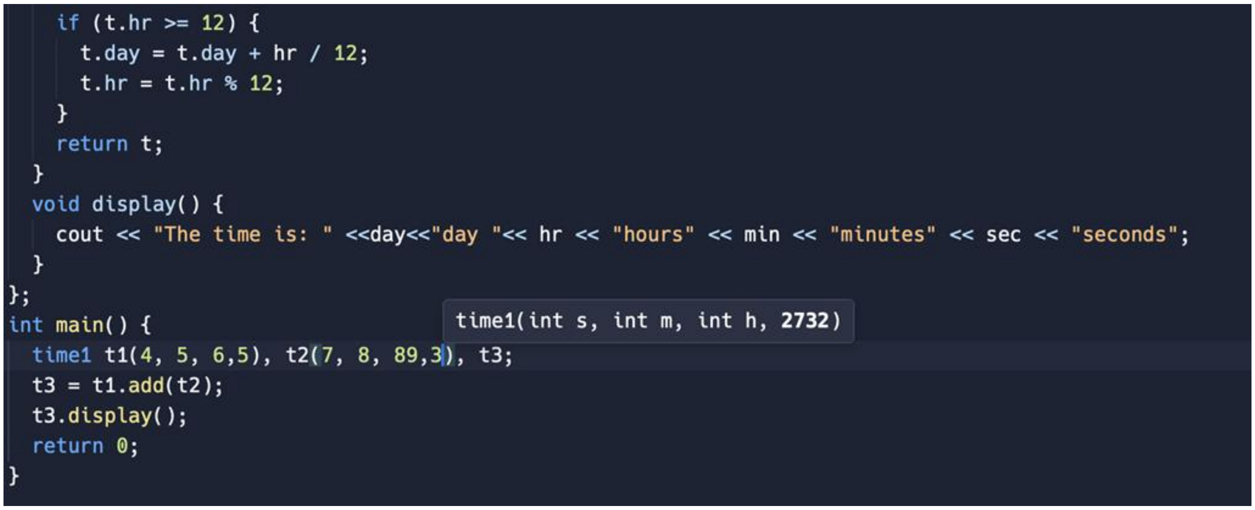


```
    if (t.m >= 1000) {
      t.km = t.km + km / 1000;
      t.m = t.m % 1000;
    }
    return t;
  }
  void display() {
    cout << "The distance is: " << km << "km " << m << "m" << cm << "cm"
         << endl;
  }
};
int main() {
  dist d1(4, 5, 6), d2(7, 8, 89), d3;
  d3 = d1.add(d2);
  d3.display();
  return 0;
}
```

50). Create a class called time with data members hour, minute, second and day. Use a constructor to initialize all the data members and use necessary member functions to add two times and display the result in the main function.



```
#include <iostream>
using namespace std;
class time1 {
private:
  int sec;
  int hr;
  int min;
  int day;
public:
  time1() {
    sec = 0; min = 0; hr = 0; day =0;
  }
  time1(int s, int m, int h,int d) {
    sec = s;
    min = m;
    hr = h;
    day = d;
  }
  time1 add(time1 c) {
    time1 t;
    t.sec = sec + c.sec;
    t.min = min + c.min;
    t.hr = hr + c.hr;
    if (t.sec >= 60) {
      t.min = t.min + sec/ 60;
      t.sec = t.sec % 60;
    }
    if (t.min >= 60) {
      t.hr = t.hr + min / 60;
      t.min = t.min % 60;
    }
```



```
    if (t.hr >= 12) {
      t.day = t.day + hr / 12;
      t.hr = t.hr % 12;
    }
    return t;
  }
  void display() {
    cout << "The time is: " <<day<<"day "<< hr << "hours" << min << "minutes" << sec << "seconds";
  }
};
int main() {
  time1 t1(4, 5, 6,5), t2(7, 8, 89,3), t3;
  t3 = t1.add(t2);
  t3.display();
  return 0;
}
```