# CONSTRUCTORS, DESTRUCTORS AND INHERITANCE IN CPP

There are two major questions that arise relative to constructors and destructors when inheritance is involved. First, when are base-class and derived-class constructors and destructors called? Second, how can parameters be passed to base-class constructors? This section examines these two important topics.

**When Constructors and Destructors Are Executed**

It is possible for a base class, a derived class, or both to contain constructors and/or destructors. It is important to understand the order in which these functions are executed when an object of a derived class comes into existence and when it goes out of existence.

To begin, examine this short program:

```
#include <iostream>
using namespace std;
class base {
public:
base() { cout << "Constructing base\n"; }
~base() { cout << "Destructing base\n"; }
};
class derived: public base {
public:
derived() { cout << "Constructing derived\n"; }
~derived() { cout << "Destructing derived\n"; }
};
int main()
{
derived ob;
// do nothing but construct and destruct ob
return 0;
}
```

As the comment in **main( )** indicates, this program simply constructs and then destroys an object called **ob** that is of class **derived**.

When executed, this program displays Constructing base

Constructing derived

Destructing derived

Destructing base

As you can see, first **base**'s constructor is executed followed by **derived**'s. Next (because **ob** is immediately destroyed in this program), **derived**'s destructor is called, followed by **base**'s. The results of the foregoing experiment can be generalized. When an object of a derived class is created, the base class' constructor will be called first, followed by the derived class' constructor. When a derived object is destroyed, its destructor is called first, followed by the base class' destructor. Put differently, constructors are executed in their order of derivation. Destructors are executed in reverse order of derivation. If you think about it, it makes sense that constructors are executed in order of derivation. Because a base class has no knowledge of any derived class, any initialization it needs to perform is separate from and possibly prerequisite to any initialization performed by the derived class. Therefore, it must be executed first.

Likewise, it is quite sensible that destructors be executed in reverse order of derivation. Because the base class underlies the derived class, the destruction of the base object implies the destruction of the derived object. Therefore, the derived destructor must be called before the object is fully destroyed. In cases of multiple inheritance (that is, where a derived class becomes the base class for another derived class), the general rule applies: Constructors are called in order of derivation, destructors in reverse order.

For example, this program

```
#include <iostream>
using namespace std;
class base {
public:
base() { cout << "Constructing base\n"; }
~base() { cout << "Destructing base\n"; }
};
class derived1 : public base {
public:
derived1() { cout << "Constructing derived1\n"; }
```

```
~derived1() { cout << "Destructing derived1\n"; }
};
class derived2: public derived1 {
public:
derived2() { cout << "Constructing derived2\n"; }
~derived2() { cout << "Destructing derived2\n"; }
};
int main()
{
derived2 ob;
// construct and destruct ob
return 0;
```

displays this output:

```
Constructing base
Constructing derived1
Constructing derived2
Destructing derived2
Destructing derived1
Destructing base
```

The same general rule applies in situations involving multiple base classes.

For example, this program

```
#include   <iostream>
using  namespace  std;
class base1 {
public:
base1() { cout << "Constructing base1\n"; }
~base1() { cout << "Destructing base1\n"; }
};
class base2 {
public:
base2() { cout << "Constructing base2\n"; }
```

```
~base2() { cout << "Destructing base2\n"; }
};
class derived: public base1, public base2 {
public:
derived() { cout << "Constructing derived\n"; }
~derived() { cout << "Destructing derived\n"; }
};
int main()
{
derived ob;
// construct and destruct ob
return 0;
}
```

produces this output:

```
Constructing base1
Constructing base2
Constructing derived
Destructing derived
Destructing base2
Destructing base1
```

As you can see, constructors are called in order of derivation, left to right, as specified in **derived**'s inheritance list. Destructors are called in reverse order, right to left. This means that had **base2** been specified before **base1** in **derived**'s list, as shown here:

```
class derived: public base2, public base1 {
```

then the output of this program would have looked like this:

```
Constructing base2
Constructing base1
Constructing derived
Destructing derived
Destructing base1
Destructing base2
```