# Derived Classes, Virtual Functions and Dynamic Memory

Dr. Matthew Smith
Swinburne University of Technology

## Compiling the Code

Use any compiler; the GNU compiler works fine. From a linux command prompt:

```
g++ demo.cpp -o test.run
```

To run the code:

```
./test.run
```

## Introduction and Background

An interesting feature of the C++ language is derived classes. Before we delve too deeply into this, we should know what a class is, as one might say it is the core of the object oriented features of the C++ language. A Class is a user-defined data type which holds its own data and functions - both of which are either publicly available to outside operations (public) or are for exclusive use internally (private). While 99% of functions a user adds to a class are custom, there are two special types of classes which can be used with functions - constructors and destructors:

- A constructor function is called when an object based on that class is created. This can either happen statically or dynamically, using new for example.
- A destructor function is called when said object is deleted using the delete function.

*Note:*
*In cases where objects are created statically, a constructor is called when they are declared. However, for these statically declared objects, the destructor is not called.*

## Derived Classes

A derived class is a new class which is based on an existing (base) class. For example, you might consider creating two classes for passenger vehicles and trucks. Writing the code for these two seperate classes would result in a lot of duplicated coding - as these both have a lot in common. Another strategy might be to create a base class describing the most basic details of a vehicle, and then to recycle these features by deriving new classes based on this base function.

A derived class will inherit the functions and data contained within the base class when created. While very convenient, this can also result in problems - the purpose of this article is to demonstrate some of these problems, and find solutions to them.

**Example**

Consider the class shown below - our Base class. It has 3 public member functions.

```
class Base {
    // The base class from which another class will be derived
    public:
        Base() {    cout << "Base:Base() Base constructor called\n";        }
        ~Base() {   cout << "Base:~Base() Base destructor called\n";       }
        void Print() {
            cout << "Base:Print() Base Print called\n";
        }
};
```

Each of these are:

- Base(): The constructor, which will be called when an object based on this class is created, either statically or dynamically.
- ~Base(): The destructor. This is called when an object dynamically allocated (using new) is deleted using delete. (Not called for static allocations, this is automatic)
- Print(): A simple function to let us print a message to screen.

Note I've taken care to make sure the user (you) is able to see that each of these functions originates from the base class - this will be useful later. Now, let's look at a derived class:

```
class Derived : public Base {
    public:
        Derived() {
            cout << "Derived:Derived() Derived constructor called\n";
        }
        ~Derived() {
            cout << "Derived:~Derived() Derived destructor called\n";
        }
        void Print() {
            cout << "Derived:Print() Derived Print called\n";
        }
};
```

This class, called Derived, is based on the Base() class. It has a constructor, destructor and Print function much like the class it was derived from.

**Jumping In**

Consider the main() function below - this is identical to the one contained within demo.cpp

```
int main() {
    // Create pointers to base and derived classes.
    Base *p;
    Derived *q;
    cout << "MSG:\t Creating new base p...\n";
    p = new Base;  // Create a new Base called p
```

```
cout << "\n";
cout << "MSG:\t Creating new derived q...\n";
q = new Derived;
cout << "\n";
```

When executed and run, this code will report the following messages to the stdout (screen):

```
MSG:    Creating new base p...
Base:Base() Base constructor called

MSG:    Creating new derived q...
Base:Base() Base constructor called
Derived:Derived() Derived constructor called
```

When we dynamically create p using new (p = new Base;), we call the base class's constructor function. When we create a derived class in memory using new, we are actually setting up two spaces - one for the base class in memory, and the other for the derived class. This means we will need to call both constructor functions - and indeed, this is the case here. Everything in this code is (so far) working as expected.

Let's look at the new section of code:

```
// P was declared as a Base pointer. So all will be fine.
cout << "MSG:\t Deleting p...\n";
delete p;
cout << "\n";

// q was declared as a pointer to a derived type. All will be fine.
cout << "MSG:\t Deleting q...\n";
delete q;
cout << "\n";
```

We are now deleting the objects p and q using delete, and so we expect the destructors to be called. In the case of the derived class, both the destructor from the base class and its own class is called:

```
MSG:        Deleting p...
Base:~Base() Base destructor called

MSG:        Deleting q...
Derived:~Derived() Derived destructor called
Base:~Base() Base destructor called
```

So far so good. The key is that, for our derived class q, both destructor functions were called. In the case where only one is called, we might have a memory leak. In this case, there have been no troubles so far - let's introduce Virtual functions.

**Virtual Functions**

Virtual functions are used in C/C++ to help us achieve runtime polymorphism. In the case where we have derived classes containing functions with identical names to those used in

the class from which they were derived, their general role is to ensure the correct function is used by the object regardless of its type. They are particularly helpful when the role of an object changes - for instance, if we have declared a pointer to a base class which is later referred to as a derived class. This is an example of polymorphism during run-time, and provides us an example of a memory leak which can be averted through the use of virtual functions. Consider the following code:

```
Base *p;
cout << "MSG:\t Casting p as derived...\n";
p = new Derived;
cout << "MSG:\t Deleting p (cast as derived)...\n";
delete p;
cout << "\n";
```

When this is compiled and run with the base and derived classes defined as they were above, this is the result:

```
MSG:	Casting p as derived...
Base:Base() Base constructor called
Derived:Derived() Derived constructor called
MSG:	Deleting p (cast as derived)...
Base:~Base() Base destructor called
```

You'll notice only the base destructor was called - even though we are deleting p which was cast as a derived class. The destructor for p is not defined here, and we have the potential for a memory leak. We can modify the base class to make sure the destructor for the derived class works correctly by making it virtual. Note the following modifications to the code:

```
class Base {
	public:
		Base() {
			cout << "Base:Base() Base constructor called\n";
		}
		virtual ~Base() {
			cout << "Base:~Base() Base destructor called\n";
		}
		void Print() {
			cout << "Base:Print() Base Print called\n";
		}
};
```

The destructor class has been made virtual. The result of this is shown below - when the code is compiled and run, we see that we are now

```
MSG:	Casting p as derived...
Base:Base() Base constructor called
Derived:Derived() Derived constructor called
MSG:	Deleting p (cast as derived)...
Derived:~Derived() Derived destructor called
Base:~Base() Base destructor called
```

Success - both the derived and base class destructor functions were called. No memory leaks here. We can use the virtual type with the print function as well:

```
class Base {
      public:
           Base() {
                 cout << "Base:Base() Base constructor called\n";
            }
            virtual ~Base() {
                 cout << "Base:~Base() Base destructor called\n";
            }
            virtual void Print() {
                 cout << "Base:Print() Base Print called\n";
            }
};
```

When used in conjunction with this set of code:

```
cout << "MSG:\t Casting p as derived...\n";
p = new Derived;
cout << "MSG:\t p is now printing...\n";
p->Print();
cout << "MSG:\t Deleting p (cast as derived)...\n";
delete p;
cout << "\n";
```

..should result in p printing from the derived class, as - even though p started as a base pointer, has been cast as a derived type. When compiled and run, we see these results:

```
MSG:      Casting p as derived...
Base:Base() Base constructor called
Derived:Derived() Derived constructor called
MSG:      p is now printing...
Derived:Print() Derived Print called
MSG:      Deleting p (cast as derived)...
Derived:~Derived() Derived destructor called
Base:~Base() Base destructor called
```

If print was not virtual in the base class, p would print from the base class:

```
MSG:      Casting p as derived...
Base:Base() Base constructor called
Derived:Derived() Derived constructor called
MSG:      p is now printing...
Base:Print() Base Print called
MSG:      Deleting p (cast as derived)...
Derived:~Derived() Derived destructor called
Base:~Base() Base destructor called
```

To conclude - in general, we use virtual functions in situations where we know objects will be subject to reassignments of classes to ensure that the correct function is used. This is important for destructors, but also important for any other functions - with the exception of the constructor, as these cannot be virtual.