**Inheritance - Additional Features**

**Constructors/Destructors**

When you instantiate a derived class object, a constructor for its base class is called first, followed by the derived class constructor. You can either:

Explicitly choose which base class constructor runs

***or***

The default base class constructor runs automatically (implicitly)

base class constructor is explicitly chosen using following syntax (on method header):

Square::Square(int side) : Rectangle(side, side) //Assumes Rectangle Constructor exists

{

…..

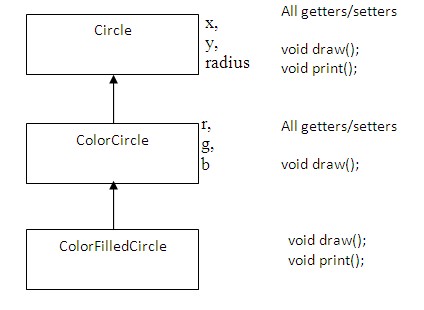
}

When an object of a derived class is destroyed (i.e., goes out of scope), **its** destructor is called first, then that of the base class

As an example, consider our hierarchy:

ColorFilledCircle c;

Constructors are chained together (i.e, invoked) as shown.



3rd

2nd

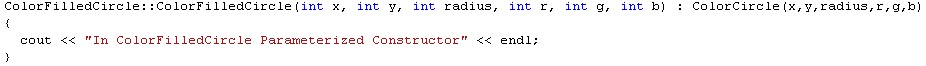
1st

Circle default constructor runs first, followed by ColorCircle’s default constructor, followed by ColorFilledCircle’s constructor

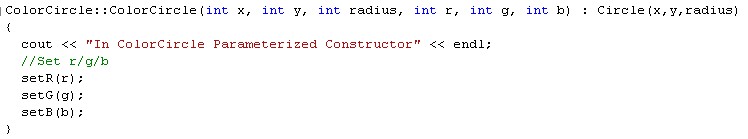
Suppose we declare ColorFilledCircle using a parameterized constructor as:

ColorFilledCircle c3(100,300,10,0,255,0);

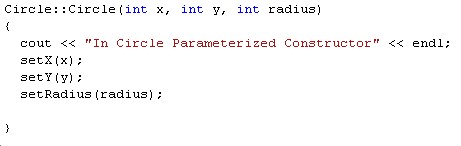
We can explicitly choose which ColorCircle constructor runs as follow:



We can also explicitly choose which Circle constructor runs as follow:



The parameterized constructor for Circle is as follows:



Once again, the constructor for Circle completes first, then the constructor for ColorCircle, and then the constructor for ColorFilledCircle.

Suppose we delete a dynamically allocated ColorFilledCircle object as shown below:

ColorFilledCircle\* cfc\_ptr;

//Create object dynamically

cfc\_ptr = new ColorFilledCircle();

//Delete ptr

delete cfc\_ptr;

In this case, the destructor for ColorFilledCircle runs first, followed by the destructor for ColorCircle, followed by the destructor for Circle - reverse of how the constructors run.

**Run Time Polymorphism and Virtual Methods**

Run Time Polymorphism essentially means that

*A Sub Class Pointer can be assigned to a Base Class Pointer*

Only pointers can be used - not objects. As an example though, consider the following:

Base base;

SubClass sub;

Where SubClass is a child object of Base. Will the following statement compile?

base = sub;

The above line does compile. However, I can never assign a base class object to a sub class object:

sub = base;

The main reason for assigning a SubClass to a Base Class is:

The Base class becomes a ***generic*** object - can represent any derived class!

**Methods declared/implemented in a Subclass object can be invoked by the generic Base class object.**

This is a very powerful feature in C++ - allows generic objects, functions, and generic data structures.

Consider the following examples:

vfunc1.cpp

vfunc2.cpp

vfunc3.cpp

The generic function is given below:

void func1(Base& base);

This function can accept either a Base object, Derived1 object or Derived2 object as an input parameter.

The results when running vfunc1.cpp are summarized below:

* **Even though we assigned a SubClass for the BaseClass (lines 1 and 3), the resulting object still behaves like the BaseClass (Undesirable)**
* **vfunc() of the BaseClass is invoked at lines 2 and 4 instead of vfunc() of the SubClass**
* **line 5 is not possible because you cannot assign a base class to a subclass (can only assign up the hierarchical inheritance tree, not down)**
* **line 6 is not possible - can only invoke those methods that are found (overriden) in the BaseClass**

In order to get the desired result, we need to modify vfunc as follows:

Use the virtual keyword before the desired method in the *BaseClass* to achieve this desired response

***virtual*** void vfunc() { ….};

**Runtime Polymorphism can only be invoked thru a *pointer* or *reference* to the base class:**

//Assign sub class to base class ptrs here

base\_ptr = &derived1;

//What is displayed here (SubClass vfunc invoked)

base\_ptr->vfunc();

Instead of invoking a virtual function using a base class pointer, a base class reference can be used.

Consider a function that passes a base class parameter by reference:

void func1(Base& base)

{

//What is displayed here

base.vfunc();

}

**Pure Virtural Methods**

* A pure virtual function is a virtual method that has no implementation (definition) with the base class
  + virtual data\_type func-name(parameter-list) = 0;
* When a virtual method is made pure virtual, any derived class **must provide its own definition for this method**. In other words, any class which inherits from a Base class with a pure virtual method **must override** that function.
* Example of pure virtual method in Base Class
  + virtual void func() = 0;
* A class that contains at least one pure virtual function is said to be ***abstract***.
* *Abstract* classes **cannot** be instantiated (i.e., objects cannot be created for *abstract* classes)
* Although you cannot create objects of an *abstract* class, **you can create pointers and references** to an *abstract* class

**Example:**

class Base

{

public:

virtual void vfunc() = 0;

};

**Why abstract classes?**

*Abstract classes allow us to implement “one interface, multiple methods” principle.*

One interface means - *all derived classes implement methods with the same signature/name*.

Multiple methods means - *all derived classes have their* ***own*** *implementation of these methods.*