

More on Inheritance

Object construction. Consider the following simple class.

```
class Animal {  
  
    // Constructor  
    Animal(const string & name);  
  
    // Retrieve name  
    const string & getName() const;  
  
    // An animal needs to eat  
    void eat();  
  
    // An animal needs to sleep  
    void sleep();  
  
    // An animal can have a typical day  
    void haveTypicalDay();  
  
private:  
  
    bool        _shouldEat;  
    bool        _shouldSleep;  
    string      _name;  
  
};
```

* Inheritance lets you extend this class.

A typical use of the class `Animal` would be something like this:

```
Animal genericBeast("John Q. Animal");
genericBeast.haveTypicalDay();
```

* The program prints out:

```
Animal John Q. Animal now eating
Animal John Q. Animal now falling asleep.
```

* We extend this class to `whale`. A whale can do everything an `Animal` does, except that it can also `spout`.

```
#ifndef _Whale_hh_
#define _Whale_hh_
#include "Animal.hh"

class Whale: public Animal {

public:

    // Constructor
    Whale(const string & whaleName);

    // A whale can spout
    void spout();

};

#endif
```

* A whale being a kind of animal, when a whale is constructed the constructor for the base class must be called first.

* The compiler will call this automatically...if a default constructor exists. In this case no default constructor exists (why?), so the constructor to whale needs to be told how to construct the base class:

```
Whale::Whale (const string & whaleName) :  
Animal(whaleName)  
{  
}  
}
```

* Before a whale is constructed, an Animal is constructed.

* After an whale is destroyed, an animal is destroyed.

* Construct Animal -> Construct Whale -> Destroy Whale -> Destroy Animal

* Client code can do this:

```
Whale greatWhale("Stuart Whale");  
greatWhale.haveTypicalDay();
```

* And it can do this:

```
greatWhale.spout();
```

* The output will look like:

```
Animal Stuart Whale now eating  
Animal Stuart Whale now falling asleep.  
Whale Stuart Whale now spouting
```

* The first two lines are from "Animal" and the third is from "Whale"

* It may occur to you that:

-Whales have their own way of eating that differs from that of generic animals.

-A whales typical day involves eating, sleeping and spouting.

* So, we're going to add a method for eating and a method for having a typical day that is specific for whales.

(First I'm going to show you the wrong way, but bear with me).

* Change the interface of whale to include these new methods:

```
class Whale: public Animal {  
  
public:  
  
    // Constructor  
    Whale(const string & whaleName);  
  
    // A whale can spout  
    void spout();  
  
    // A whale has its own way of eating:  
    void eat();  
  
    // A whale's typical day is different  
    // from that of a generic animal:  
    void haveTypicalDay();  
  
};
```

* Now we don't have to tell the whale to spout because that is already part of the Whale's typical day:

* Our program now looks like this:

```
Whale greatWhale("Stuart Whale");
greatWhale.haveTypicalDay();
```

* The program prints:

```
Whale Stuart Whale now eating KRILL
Whale Stuart Whale now spouting
Animal Stuart Whale now falling asleep.
```

* So far so good, but now consider the following.

* Since Whale inherits publically from Animal, it is always usable as an Animal.

* These two constructions are possible:

```
Animal *myCoolPet=new Whale("Stuart Whale");
myCoolPet->haveTypicalDay();
```

and

```
// A subroutine to make an Animal have a typical week:
void haveTypicalWeek(const Animal & myAnimal) {
    for (int day=0;day<7;day++) myAnimal.haveTypicalDay();
}

// could be called like this:
Whale greatWhale("Stuart Whale");
haveTypicalWeek(greatWhale);
```

* I will show you a problem now, and it is the same problem with both of these features. I will demonstrate it with the first construction

Here it is again, at the top of this page for reference:

```
Animal *myCoolPet=new Whale("Stuart Whale");  
myCoolPet->haveTypicalDay();
```

* What output does this produce? Here it is:

```
Animal Stuart Whale now eating  
Animal Stuart Whale now falling asleep.
```

* Since we created a whale, we expect that its typical day consists of eating KRILL and falling asleep! But this is apparently not happening! Why?

* The object myCoolPet has two data types, Whale and Animal.

* (*myCoolPet) is an object of class Whale.

* myCoolPet has type pointer-to-Animal

* Animal is said to be the "static type"

* Whale is said to be the "dynamic type".

* You are "handling the whale through the Animal interface"

* You can do anything to the whale that the Animal interface permits.

* But you should expect that

the request to eat is handled the way an Whale would handle it.

the request for a typical day is handled the way a Whale would handle it.

* To get that behaviour, you have to specify that a function be dynamically bound, e.g, that it be resolved by the class of the actual object, not by the pointer type.

Virtual Functions

- * That kind of function is called a virtual function.
- * The problem described above goes away entirely if the methods that are overridden are declared virtual.
- * Overriding a common function (i.e., a nonvirtual function) is a **mistake**, and the kind of behaviour that I have been describing is the **consequence**.

Here is how we fix the problem:

- * In the header files for **both** Whale and Animal, we modify the function declaration to include the keyword `virtual`:

```
virtual void eat();  
virtual void haveTypicalDay();
```

- * We don't need to change anything in the implementation.
- * While we're at it, we might as well override the sleep method, too. (A whale takes quick snoozes between spouts).

```
Whale Stuart Whale now eating KRILL  
Whale Stuart Whale now spouting  
Whale Stuart Whale now falling asleep.
```

- * This is correct behaviour.

Polymorphism.

The function:

```
// A subroutine to make an Animal have a typical week:
void haveTypicalWeek(const Animal & myAnimal) {
    for (int day=0;day<7;day++) myAnimal.haveTypicalDay();
}
```

can take as arguments an object of class Animal or any of its subclasses:

```
haveTypicalWeek(myOrgangutang);
haveTypicalWeek(myElephant);
...
```

* This ability to take different data types is called polymorphism.

* It doesn't happen accidentally, it only happens when a careful design effort has been made.

* The actual data type is determined at run time.

* This determines which member function is actually called.

* The process is known as **dynamic binding**, **run-time binding**, or **late binding**.

* It incurs a small performance overhead (which is often exaggerated)

Polymorphism implies a kind of message-dispatching system. A message is sent to a object whose type is unknown to the programmer. The compiler takes care that the message is handled by the proper function.

* Suppose you have created a class library or toolkit consisting of C++ classes.

* Or suppose that you have contributed a class or two to a project,

* Or even that you have carried out a small-scale program using some classes that you've made.

* There are two kinds of clients that you should expect.

#1 You should expect people to create instances of your class. (I'm sure you expected this).

#2 You should expect people to derive **subclasses**, too.

Example. The Open Inventor class library is a library that contains a few 3D shapes, plus lights, cameras, animation engines... In a visualization project for particle physics, we have created subclasses to represent geometrical shapes and trajectories common in particle physics.

* When you declare a **common** member function, you tell developers that **subclasses must inherit both the interface and the implementation.**

* When you declare a **virtual** member function, you tell developers that subclasses **must inherit the interface and may inherit the implementation** ...(which is also to say that they may override the implementation.)

* If you write a class you should think about these issues.

* A very common mistake is to **not use virtual functions because you haven't really bothered to learn about them.**

* And if you subclass, you should **NEVER** override a nonvirtual function.

* There is one other possibility:

* "A subclass must inherit the interface but cannot inherit the implementation"... Because there is no implementation.

* A function like this is called a **pure virtual function**.

* Why would you do this?

* Well, all animals can have typical days, but there is no such thing as a typical typical day. We can therefore assert, in Animal, that all Animals must somehow have a typical day, and concrete subclasses of Animals (Whales, Orangutang, Peguins, Trouts...) **must describe their own typical days** because an Animal just ain't got one, Jack.

In Animal.hh:

```
// An animal needs to eat (but I don't know how)
virtual void eat() = 0;
```

```
// An animal needs to sleep (but I don't know how)
virtual void sleep() = 0;
```

```
// An animal can have a typical day (but I don't know how)
virtual void haveTypicalDay() = 0;
```

* A perfectly reasonable consequence is that the Animal class can't be instantiated, e.g, you can have a whale, you can have an orangutang, you can have a great northern pike, but you can't have an object with class Animal because :

- that class has unimplemented functions.
- and anyway the compiler won't let you.

*So the only use of a class like this is as a base class.

*Such a class is called an Abstract Base Class.

We've shown that pure virtual functions are logical, but we haven't shown that they are useful. So, why would we want one?

Consider this piece of code:

```
// A subroutine to make an Animal have a typical week:
void haveTypicalWeek(const Animal & myAnimal) {
    for (int day=0;day<7;day++) myAnimal.haveTypicalDay();
}
```

This piece of code works on all Animals, even the ones that haven't been invented yet.

Now take away the pure virtual function. You can leave the `haveTypicalDay` method inside Whale, Orangutang, Sturgeon, & cetera.

You now have to make all the following functions:

```
void haveTypicalWeek(const Sturgeon & mySturgeon)
void haveTypicalWeek(const Orangutang & myOrangutang)
.
.
.
```

And the second there's a new species, you have a new method to add.

Whereas the Abstract base class permits the same piece of code work with all animals.

The Open/Closed principle says software may be made open to extension but closed to modification.

Some lessons:

Abstract Base Classes are sometimes called **pure interfaces** because they provide no implementations (at least not for some functions.)

It is useful to create Abstract Base Classes because they're the most extensible data types around. Since they have no implementation, there is no implementation to carry around in the subclass.

A concrete base class, or Implementation class, can be replaced with an alternate base class if needed, and without breaking existing code.

Lesson #1 Program to the highest level of generality possible, (remember `istack::print()`)

Program to a pure interface if you can.

Lesson #2 If you're designing a class library, try to find generality.

Separate the interface from the implementation. (E.G. if you only need a whale, then create the animal class anyway.)

These concepts are difficult but important, acquired over time through practice and study. And this is about all of the C++ language features that I will dump on you in the first week.

Now, we'll take a break from highbrow theory.