OOP Concepts by Example

by Randy Charles Morin

Of late, I have been writing very narrow focused articles that explain how to accomplish this or that task. Many of you have changed your questions from the narrow focus of how-to questions to broader theorectical questions. One question I got lately that intrigue me was to explain the concepts of OOPs showing C++ examples. Let's start by laying down some ground work. I assume that you are familar with the following OOP concepts; classes, objects, attributes, methods, types. If not, then this article might not be in your realm. I'd suggest starting with the basic concepts of C++ before you attempt to understand the more indepth concepts that I'll be discussing in this article. When we speak of OOP concepts, the conversation usually revolves around encapsulation, inheritance and polymorphism. This is what I will attempt to describe in this article.

Inheritance

Let us start by defining inheritnace. A very good website for finding computer science definitions is http://www.whatis.com. The definitions in this article are stolen from that website.

Definition: Inheritance

> Inheritance is the concept that when a class of object is defined, any subclass that is defined can inherit the definitions of one or more general classes. This means for the programmer that an object in a subclass need not carry its own definition of data and methods that are generic to the class (or classes) of which it is a part. This not only speeds up program development; it also ensures an inherent validity to the defined subclass object (what works and is consistent about the class will also work for the subclass).

The simple example in C++ is having a class that inherits a data member from its parent class.

```
class A
{
public:
integer d;
};

class B : public A
{
public:
};
```

The class B in the example does not have any direct data member does it? Yes, it does. It inherits the data member d from class A. When one class inherits from another, it acquires all of its methods and data. We can then instantiate an object of class B and call into that data member.

```
void func()
{
        B b;
        b.d = 10;
};
```

Polymorphism

Inheritance is a very easy concept to understand. Polymorphism on the other hand is much harder. Polymorphism is about an objects ability to provide context when methods or operators are called on the object.

Definition: Polymorphism

> In object-oriented programming, polymorphism (from the Greek meaning "having multiple forms") is the characteristic of being able to assign a different meaning to a particular symbol or "operator" in different contexts.

The simple example is two classes that inherit from a common parent and implement the same virtual method.

```
class A
{
public:
virtual void f()=0;
};

class B
{
public:
virtual void f()
        {std::cout << "Hello from B" << std::endl;};
};

class C
{
public:
virtual void f()
        {std::cout << "Hello from C" << std::endl;};
};
```

If I have an object A, then calling the method f() will produce different results depending on the context, the real type of the object A.

```
func(A & a)
{
        A.f();
};
```

Encapsulation

The least understood of the three concepts is encapsulation. Sometimes, encapsulation is also called protection or information hiding. In fact, encapsulation, protection and information hiding are three overlapping concepts.

Definition: Encapsulation

> Encapsulation is the inclusion within a program object of all the resources need for the object to function - basically, the method and the data. The object is said to "publish its interfaces." Other objects adhere to these interfaces to use the object without having to be concerned with how the object accomplishes it. The idea is "don't tell me how you do it; just do it." An object can be thought of as a self-contained atom. The object interface consists of public methods and instantiate data.

Protection and information hiding are techniques used to accomplish encapsulation of an object. Protection is when you limit the use of class data or methods. Information hiding

is when you remove data, methods or code from a class's public interface in order to refine the scope of an object. So how are these three concepts implemented in C++? You'll remember that C++ classes have a public, protected and private interface. Moving methods or data from public to protected or to private, you are hiding the information from the public or protected interface. If you have a class A with one public integer data member d, then the C++ definition would be...

```
class A
{
public:
integer d;
};
```

If you moved that data member from the public scope of the private scope, then you would be hiding the member. Better said, you are hiding the member from the public interface.

```
class A
{
private:
integer d;
};
```

It is important to note that information hiding are not the same as encapsulation. Just because you protect or hide methods or data, does not mean you are encapsulating an object. But the ability to protect or hide methods or data, provide the ability to encapsulate an object. You might say that encapsulating is the proper use of protection and information hiding. As an example, if I used information hiding to hide members that should clearly be in the public interface, then I am using information hiding techniques, but I am not encapsulating the class. In fact, I am doing the exact opposite (un-encapsulating the class). Do not get the idea that encapsulation is only information hiding. Encapsulation is a lot more. Protection is another way of encapsulating a class. Protection is about adding methods and data to a class. When you add methods or data to a class, then you are protecting the methods or data from use without first having an object of the class. In the previous example, the data member d cannot be used except as a data member of an object of class A. It is being protected from use outside of this scenario. I have also heard many computer scientist use information hiding and protection interchangeably. In this case, the scientist takes the meaning of protection and assign it to information hiding. This is quite acceptable. Although I'm no historian, I believe the definition of information hiding has taken some turns over the years. But I do believe it is stabilizing on the definition I presented here.

Abstraction

Another OOP concept related to encapsulation that is less widely used but gaining ground is abstration.

Definition: Abstraction

> Through the process of abstraction, a programmer hides all but the relevant data about an object in order to reduce complexity and increase efficiency. In the same way that abstraction sometimes works in art, the object that remains is a representation of the original, with unwanted detail omitted. The resulting object itself can be referred to as

an abstraction, meaning a named entity made up of selected attributes and behavior specific to a particular usage of the originating entity.

The example presented is quite simple. Human's are a type of land animal and all land animals have a number of legs. The C++ definition of this concept would be...

```
class LandAnimal
{
public:
virtual int NumberOfLegs()=0;
};

class Human : public LandAnimal
{
public:
virtual int NumberOfLegs()
        {return 2;};
};
```

The method NumberOfLegs in LangAnimal is said to be a pure virtual function. An abstract class is said to be any class with at least one pure virtual function. Here I have created a class LandAnimal that is abstract. It can be said that the LandAnimal class was abstracted from the commonality between all types of land animals, or at least those that I care about. Other land animals can derive there implementation from the same class.

```
class Elephant : public LandAnimal
{
public:
virtual int NumberOfLegs()
        {return 4;};
};
```

Although I cannot create an instance of the class LandAnimal, I can pass derived instances of the class to a common function without having to implement this function for each type of LandAnimal.

```
bool HasTwoLegs(LandAnimal & x)
{
        return (x.NumberOfLegs()==2);
};
```

There is also a less rigid definition of abstraction that would include classes that without pure virtual functions, but that should not be directly instantiated. A more rigid definition of abstraction is called purely abstract classes. A C++ class is said to be purely abstract, if the class only contains pure virtual functions. The LandAnimal class was such a class. Purely abstract classes are often called interfaces, protocol classes and abstract base classes.

More Concepts

Another growing concept in OOP is dynamic and static binding. Most languages provide one or the other. C++ provides both. A method that is not virtual is said to be statically bound, whereas virtual methods are said to be dynamically bound. Non-virtual methods are statically bound, because the binding of the method is performed at compile and link time and cannot be changed. Virtual methods are dynamically bound, because the binding of the method is actually performed at run-time. When you call a virtual method, a small lookup is performed in the object virtual table (a.k.a. vtable) to find the address of the method being called. By manipulating an objects vtable at run-time, the target address

can be altered. Four other growing OOP concepts are persistance, concurrency, reflection and object composition. I will not discuss these here, but maybe in a later article. Hope this article proves informative and thank you for your time.