What I know about...

OOP with C++

Charles Clayton

Last updated: February 27, 2017

I. Introduction

I like to consider myself a pretty decent programmer, but mostly I'm self-taught. Sure, I've taken formal courses requiring the use of C, C++ , R, MATLAB, 8051 Assembly, MIPS, VHDL, and SystemVerilog, but these never go through the language comprehensively. It's not a course about the details of a language, or the terminology of its features. Usually we'll get a "Hello, World!" template and a project to build something from it. The process here is to hack together some code that accomplishes the goal by looking up questions on StackOverflow; finding online tutorials; reading example code; gradually picking up good habits; refactoring old code; and experimenting with different styles. From these base knowledge I've branched out into at times using JavaScript, Python, C#, VBA, bash, whatever tool I think will suit the task at hand, learning them in exactly the same manner. Yeah, I think I've used OOP. If I'm keeping track of multiple copies of the same jumble of information, I'll just create a class for that so I don't have to write twice as much. Hey, if I need to do an operation on that jumble, why not write a function within the class using the internal information so that I don't have to pass all the arguments to it? Hey, this class is basically a sub class of that class, why not use this inheritance thing? I get the idea, but I've never gone through it formally. But recently I was asked about "polymorphism"... what? "Virtual functions", "static classes", all of it. I realized I don't know the right terminology, and I don't know what I don't know. So I'm going to read this book *Absolute C++* by Walter Savitch start to finish, and I'm going to summarize what I find interesting here for future reference.

II. Some Catchup

If you're coming in straight from C, let's quickly cover a couple C++ isms .

A. Modifiers and Qualifiers

You're probably familiar with normal types, but a modifier is where you can add more detail (or modify) the type of the variable or class you're declaring. These include *short*, *long*, *unsigned*, *signed*. Qualifiers can also provide additional detail to the definition, such as *const*, *volatile*, *restrict*.

```
int i = 0;                // normal int
const unsigned i = 0;     // modified int
```

Use the qualifier *volatile* when you don't want the compiler to optimize your variable. For instance, perhaps you're modifying the variable elsewhere in the code that your compiler doesn't know about. You probably won't ever need to bother with *restrict*, but it means that when you define a pointer to a memory location, that memory location can only be accessed through that specific pointer, and no other pointers.

B. Casting

If you're like me, you learned to cast in C like so:

```
int i = 2;
double j = (double)i/3;
```

That's supported in C++ , but we should leave that behind in C. Here we have four different kinds of casting, the main two you will understand right now are:

```
static_cast<double>(i); // returns with specified type
const_cast<double>(i);  // returns with specified type,
                           ignoring constantness
```

The other two won't really make sense until we talk more about object-oriented programming. So we'll revisit these later:

```
dynamic_cast<double>(i);
reinterpret_cast<double>(i);
```

//todo

C++ will also do type coercion for us, whereby the cast is done automatically for you:

```
double i = 25; // 25.0
int j = 'm';   // 109
```

C. Evaluation

C++ uses short-circuit evaluation not complete evaluation. This means if you ask it to evaluate a condition, it will evaluate it left-to-right, and won't evaluate the entire condition if the leftmost condition is enough information to stop. For instance:

```
(A || B)  // if A is true, B will not be checked
(A && B)  // if A is false, B will not be checked
```

D. Enumeration

Enumeration allows you to define a type which really is just an integer behind the scenes, as well as create a list of constant ints. For instance in the following code, constant ints for the days of whe week are created and automatically assigned values 0-6. Then we can use the type Day and assign it and compare it to values from those constants, this helps us avoid using integers as labels.

```
enum Day {MON, TUE, WED, THU, FRI, SAT, SUN};
Day today = SUN;   // really int today = 6;

if(today == MON) ...
```

If you don't want the identifiers MON, TUE, and on defined as global constants, you can use *enum class* instead of *enum*. If you use this, then these are only defined within the scope of the enum type, and accessed with "`::`", the scope resolution operator.

```
enum class Color BLUE = 3, RED = 2, GREEN = 1 ;
Color fav = Color::GREEN;

if (fav == Color::BLUE) ...
```

Additionally, note that the identifiers may be assigned arbitrary integer values. If no assignments are made, they are automatically sequential starting with 0.

E. Namespace

A namespace is a collection of class and variable definitions. When you use a namespace, it sets your block of code to the scope of the class you've called your namespace. Let's say you set your namespace with `using namespace std;` This means that you can use the functions and variables of the `std` class directly (such as `cout` or `endl`), without having to use the scope resolution operator like `std::cout`.

You can define your namespace globally, or on the fly in code to set the namespace of a specific scope, such as a function, class, or conditional block.

To group functions and variables within your own defined namespace, you can do so like:

```
namespace Foo {

  void buzz() {
    std::cout << "Foo, buzz" << std::endl;
  }

  void bar() {
    using namespace std; // just for this block
    cout << "Foo, bar" << endl;
  }

}

...

Foo::bar();
using namespace Foo; // using Foo::bar; for only bar
bar();
```

You're also free to use multiple namespaces.

F. Function Overloading

In C++ we can give multiple definitions to the same function name, and the function definition that will be used will be based on the parameters provided to it, this is called overloading the function. The following is valid code:

```
double pythagoras(double x, double y){
  return sqrt(x*x + y*y);
}

double pythagoras(double x, double y, double z){
  return sqrt(x*x + y*y + z*z);
}
```

This means that C++ will allow you to call the same function as both:

```
double i = pythagoras(3, 2);
double j = pythagoras(3, 2, 10);
```

Note that this means C++ will allow you to make multiple definitions of the same function name. If you're not careful, you may accidentally reuse the same function name and your code will behave unpredictably.

G. Default Arguments

With C++ you can also call functions without passing all the required arguments to them if you specify default arguments like so:

```
void say_hello(char* name = "friend") {
  cout << "Hello, " << name << "!" << endl;
}
```

Which can be called with or without a name argument:

```
say_hello();      // Hello, friend!
say_hello("Charlie");   // Hello, Charlie!
```

Note that the default argument must be the rightmost parameter. If you omit arguments, they must be those defined in the function header last.

III. Proto-OPP

A. Methods vs. Functions

A class is a defined type that can have both data and functions associated with it. These functions are often methods, and they are called *on* the class, not as a stand-alone function with arguments passed to it. C++ actually calls methods "member functions" as opposed to "free functions", but we'll stick with "methods" because it's commonly used.

B. Structures

Structs are seen in C as well, these are classes that don't have methods associated with them. They *can* have methods, but that's not the way to go. Rather, they're just a good way to create a class to bundle information together.

```
struct Coordinate {
  double x;
  double y;
  double z;
};

...

Coordinate current_position;
current_position.x = 10;
current_position.y = 0.4;
current_position.z = -5.5;
```

Here, Coordinate is the structure *tag*, and x, y, z are the *member names* and each have a *member value*.

If you're using a pointer to a struct (or a class), C++ has a separate operator for that to simplify the notation of accessing its member values. `(*p).value` is equivalent to `p->value`. The differences from the above example bolded below:

```
Coordinate *current_position = new Coordinate;
current_position->x = 10;
current_position->y = 0.4;
current_position->z = -5.5;
```

Let's move right along to classes though.

IV. Classes

Classes are like structures except we can have *member functions* as well as *member variables*. These are pretty much the whole point of OOP. So let's convert the above struct into a class.

A. Member Variables

```
class Coordinate {
public:
  double x;
  double y;
  double z;
};

...

Coordinate current_position;
current_position.x = 10;
current_position.y = 0.4;
current_position.z = -5.5;
```

So the first thing you'll note is that in order to continue accessing the member variables x, y, z, I had to label them as public, because by default member variables are private. This means they can be accessed internally to the class, but not directly from outside of the class.

B. Member Functions (Methods)

Now let's also define a member function that simply prints the coordinates the coordinates.

```
class Coordinate {
public:
  double x;
  double y;
  double z;
  void show_location() {
    cout << "At:" << x << "," << y << "," << z << endl;
  }
};

...

Coordinate currentPosition;
current_position.x = 10;
current_position.y = 0.4;
current_position.z = -5.5;
current_position.show_location(); // At: 10,0.4,-5.5
```

You could also define `show_location` outside the class definition using the scope resolution operator again like so:

```
class Coordinate {
public:
  double x;
  double y;
  double z;
  void show_location();
}

Coordinate::show_location() {
    cout << "At:" << x << "," << y << "," << z << endl;
};

...

Coordinate currentPosition;
current_position.x = 10;
current_position.y = 0.4;
current_position.z = -5.5;
current_position.show_location(); // At: 10,0.4,-5.5
```

Note that you use the scope resolution operator when you are accessing or defining a property of a class type, you use the dot operator when you are accessing an object of that class.

C. Constructors

Now since it's sort of tedious to set the x, y, z of a coordinate the above way, we'll create a *constructor*, so that when we create an instance of the class, we have to set x, y, z automatically. The constructor is simple a member function that has the same name as the class. When the class instance is created, you pass the arguments to the constructor. Here the constructor only initializes values to the member variables.

```
class Coordinate {
public:
  Coordinate(double x, double y, double z) {
    _x = x; _y = y; _z = z;
  }
  void show_location();
private:
  double _x;
  double _y;
  double _z;
};

void Coordinate::show_location() {
  cout << "At:" << _x << "," << _y << "," << _z << endl;
}

...

Coordinate pos(10, 0.4, -5.5);
pos.show_location(); // At: 10,0.4,-5.5
```

Constructors can also have optional/default parameters, which is very helpful.

Note that I have also changed x, y, z from being public to being private, so now we can only set and read them through the class' member functions. Because x, y, and z are private, by convention to indicate they are private I have preceded them with an underscore.

This leads us the principle of encapsulation.

D. Encapsulation

Encapsulation is all about bundling together the data and the methods that interact with the data, by doing this we hide this data from use outside of the class to prevent it from being improperly interfered with. This is most easily done by making certain member variables and member functions private as we did previously.

Instead, we allow the program outside of the class to interact with the class' data through *accessor functions* and *mutator functions*. Accessor functions are methods that return values within the class, for example:

```
class Coordinate {
public:
  Coordinate(double x, double y, double z) {
    _x = x; _y = y; _z = z;
  }
  double get_x() {
    return _x;
  }
  double set_x(double x) {
    if(-10 < x && x < 10) _x = x;
  }
  void show_location();
private:
  double _x;
  double _y;
  double _z;
};
```

Here, `_x` is not public, so we can't set or read it directly using `pos.x`, but we can now read it using `pos.get_x()`.

Similarly, mutator functions allow us to modify the data inside the class, but only through the functions defined within the class.

You might think that this defeats the purpose of making these variables private, but you could protect the member variables of the class by only allowing permissible values. For instance, a mutator function to assign a value to `_x` could disallow negative numbers, or numbers beyond a threshold, or so on.

So the first aspect of encapsulation is that we set the member variables of a class to be private, then create an *interface* to the object by showing the member function headers and adding explanatory comments. To add another step of encapsulation, we will define the member functions (*implementation*) in a separate place from where the class is defined then include it via a header file.

E. Static Members

You can also define variables that are shared by all objects of a class. These are called static variables, and they're defined with respect to the class type, not an object of that class. These can give the benefits of a global variables without getting ugly. Private static members can only be accessed by objects of that class.

For instance, continuing from our previous example, let's create a static variable called `_num` to keep track of the number of coordinates we initialize. You could imagine something like this being useful for games.

```
class Coordinate {
public:
  Coordinate(double x, double y, double z) {
    _x = x; _y = y; _z = z;
    _num++;
  }
  double get_x() {
    return _x;
  }
  double set_x(double x) {
    if(-10 < x && x < 10) _x = x;
  }
  static void print_num_coordinates() {
    cout << "Num coords: " << _num << endl;
  }
  void show_location();
private:
  static int _num;
  double _x;
  double _y;
  double _z;
};
```

And in order to initially set `_num` to 0, we need to use the following syntax *outside* of the main function and outside the class definition.

```
int Coordinate::_num = 0;
```

This is a little weird, and doesn't seem very private, but since it can only be initialized the once outside of the main function, it can't be set again. Then in main, after initializing two coordinates, we can call `print_num_coordinates()` like so:

```
Coordinate pos(10, 0.4, -5.5);
pos.show_location();
Coordinate pos2(0, 0, 0);
Coordinate::print_num_coordinates(); // Num coords: 2
```

So again note that static variables and functions are identified using "`::`" and non-static (dynamic) variables and functions are identified using "`.`"

F. Nested Classes

It should be easy to see how you may be able to develop nested classes, that is classes that have other classes as one of their properties. They have the predictable syntax:

```
class Coordinate {
public:
  Coordinate(double x = 0, double y = 0, double z = 0) {
    _x = x; _y = y; _z = z;
    _num++;
  }
  void set(double x, double y, double z) {
    _x = x; _y = y; _z = z;
  }
  void show_location() {
    cout << "At:" << _x << "," << _y << "," << _z << endl;
  }
  static void print_num_coordinates() {
    cout << "Created coordinates: " << _num << endl;
  }
private:
  static int _num;
  double _x;
  double _y;
  double _z;
};

class Plane {
public:
  // note default arguments
  Plane(int num, int x = 0, int y = 0, int z = 0) {
    num_passengers = num;
    location.set(x, y, z);
  }
  Coordinate location;
  int num_passengers;
  ...
};
```

And now we can use the coordinates class with our new class Plane.

```
Plane p(1000);
p.location.show_location();  // At: 0,0,0
p.location.set(1, 2, 3);
p.location.show_location();  // At: 1,2,3
```

G. this

When you're defining member functions of a class, you can use the `this` keyword as a pointer to the object that is calling the method and use the arrow operator to access its members. For instance:

```
class Cash {
public:
  double dollars;
  double cents;
  double sum() {
    return this->dollars + this->cents/100;
  }
};

...

Cash str;
str.dollars = 2;
str.cents = 3;
```

```
cout << str.sum() << endl; // 2.03
```

However, you should be able to see that here you can usually just define the member function as `return dollars + cents/100;` without needing the `this` keyword, but be aware of what it is.

H. Inheritance

The main benefit of inheritance is that you can reuse the same code across different classes. With this, we can first create generalized classes (called the *base/parent* class) and from there create more specific classes (called the *derived/child* classes) that can inherit the properties of the base class.

Say we have a class `Animal`, we could give our animal a property `_happy`, then check and set their mood with the methods `give_pets()`, `is_happy()`.

```
class Animal {
public:
  void give_pets(bool i) { // pets makes animal happy
    _happy = i;
  }
  bool is_happy() {
    return _happy;
  }
private:
  bool _happy;
};
```

Now let's create child classes for the specific types of animals `Dog` and `Cat`. Petting them makes them both happy, but different animals show they are happy in different ways, either by purring or wagging their tail. So let's create two more functions:

```
class Dog : public Animal {
public:
  bool is_tail_wagging() {
    return is_happy(); // can use Animal's is_happy()
  }
};

class Cat : public Animal {
public:
  bool is_purring() {
    return is_happy();
  }
};

...

Dog spot;
spot.give_pets(true); // can use Animal's give_pets()

Cat checkers;
checkers.give_pets(false);

cout << "Dog: " << spot.is_tail_wagging(); // 1
cout << "Cat: " << checkers.is_purring();  // 0
```

1) Private vs. Protected: In the definition of the Dog and Cat classes, even though we're children we still can't use the variable `_happy` because that's a `private` variable.

Private members of a class are still only accessible from within an object of that class. However *protected* members on the otherhand are accessible by both objects of a class, and objects of classes inherited from it.

So if we change `private` to `protected`, then we can access the `_happy` variable directly. You can also have protected vs. private inheritance as well as public, but don't bother with that.

2) Polymorphism/Virtual Functions: Polymorphism is all about using the same function name, but have it mean different things in different contexts (for different types/classes). For instance, let's add the method `make_noise()` to Animal. We can define it for our `Animal` class to output something generic like *animal noises*. If we left this as it were, both Dog and Cat objects, would output *animal noises*. But we can redefine the function for Dog and Cat to output *woof* and *meow* respectively.

We can redefine `make_noise()` in Dog and Cat without anything different from what we've already been doing, but there are some interesting cases that might catch you out[1]. So just to make sure that everything behaves as we expect, add the qualifier `virtual` to the function definition in the parent class. This tells the compiler that it'll need to figure it out what to use at runtime. Note that the virtual-ness of the function is inherited by the child.

```
class Animal {
public:
  void give_pets(bool i) { _happy = i; }
  virtual void make_noise() {
    cout << "*Animal noises*";
  }
private:
  bool is_happy() { return _happy; }
protected:
  bool _happy;
};

class Dog : public Animal {
public:
  bool is_tail_wagging()return _happy;
  void make_noise() { cout << "Woof!"; }
};

class Cat : public Animal {
public:
  bool is_purring() { return _happy;  }
  void make_noise() { cout << "Meow"; }
};

...

Dog spot;
Cat checkers;
Animal the_thing;

the_thing.make_noise();  // *Animal noises*
spot.make_noise();       // Woof!
checkers.make_noise();   // Meow
```

If you don't call the function virtual, changing it in children classes the function is called *redefining*. If you do call the function virtual, changing it in a child class is called *overriding*. That doesn't matter much to you since we've agreed to just always call these functions virtual, but it makes a difference to the compiler.

To force a programmer to create specific member functions of a class that's inherited from a base class, even if you don't want to define it in the base class, you can use an *abstract class*.

```
class Animal {
public:
  //so any child must have make_noise() defined
  virtual void make_noise() = 0;
};
```

[1]Why do we need Virtual Functions in C++? : http://stackoverflow.com/q/2391679/2374028

V. Operator Overloading

Operators like + and == are just functions that we use in a different way. We could easily have functions like `plus(a, b)` and `equals(a, b)` which would be more consistent to the language, but people don't really like that. So in the same way we can overload functions to work with different types, we can similarly overload operators.

For instance, continuing with our Coordinates class – which I'm going to simplify back to a basic form – We can define an operation for + to perform when called open two objects of the coordinates class.

```
class Coordinates {
public:
  Coordinates() {};
  Coordinates(double x, double y, double z)
    : _x(x), _y(y), _z(z) {}
  double _x;
  double _y;
  double _z;
  void show() {
    cout << _x << "," << _y << "," << _z << endl;
  }
};
```

Note here that I am overloading the constructor function. There is the first definition of Coordinates() which does nothing. This is the constructor that will happen if no arguments are passed upon initialization. For the second constructor, if three arguments are passed they are assign to x, y, and z respectively. Note that I am using a different syntax here than I did before, but these do the same thing.

Now we can define our operator, which simply returns a new coordinate initialized with the addition of each axis value respectively.

```
Coordinates operator + (Coordinates a, Coordinates b) {
  return Coordinates(a._x+b._x, a._y+b._y, a._z+b._z);
}
```

Now, by default C++ passes an entire copy of the object when an object is passed as a parameter. Logically, this is consistent with what you would expect and is nice from a theoretical computer science perspective, however, it can be very slow.

When we pass coordinates a and b as operators, we only do so to read their x, y, and z values and add them. We don't need an entire copy of the objects to do that, and if these objects are large and full of data it can be expensive in terms of memory and clock cycles, so let's pass these parameters by reference.

Similarly we'll call the parameters `const`, so that C++ knows we are not going to modify the arguments so that the operation can be even more efficient.

```
Coordinates operator +(const Coordinates &a,
                       const Coordinates &b){
  return Coordinates(a._x+b._x, a._y+b._y, a._z+b._z);
}
```

Now putting our new operator to use:

```
  Coordinates a(1, 2, 3);
  a.show(); // 1,2,3

  Coordinates b(10, 11, 12);
  b.show(); // 10,11,12

  Coordinates c;
  c = a + b;
  c.show(); // 11,13,15
```

Similarly, you can overload the ==, ++, %, and so on.

VI. Vectors

Vectors are arrays that can grow and shrink during runtime. Just like an array, you define them with a base type.

//todo

VII. Templates

Templates let us define functions and classes that have arguments of different types.

//todo