

Lecture 29 – Object-Oriented Programming

J. Zarnett

jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

August 27, 2016

Acknowledgments: W.D. Bishop

Up to this point we have organized our program using procedural programming: the use of functions to give us program structure.

We can break down a larger problem into smaller problems, each of which we can accomplish in a re-usable sequence of steps (function).

The painful reality is that most software projects fail.
They are often late, unreliable, and/or expensive.

This is because software is extremely complex.
We need to manage this complexity if a project is to succeed.
Procedural programming is not adequate to the task.

Idea: if an electrical engineer needs a transistor, she does not need to build one from scratch; just go to the bin and use an existing one.

Now take the idea of re-usable components and apply it to software.
We've already had re-usable functions, so why not components?

What we'd like is to package up data and functions that manipulate that data into single, self-contained, re-usable unit.

Objects are the way of packaging & managing our data and functions.
They are the building blocks of an object-oriented program.

An object models a tangible item or some abstract concept:
What properties does an object have? What can the object do?

The world is full of “things” (objects). Imagine a dog.
A dog has characteristics like colour, weight, age...
A dog also has behaviours like bark, move, sleep...

To represent a dog in software, create an object that possesses both characteristics (**data variables**) and behaviours (**functions**) of a dog.

Objects interact with other objects through their functions, and not by manipulating another object's data directly.

Modelling the system as objects and their interactions is known as **Object Oriented Programming**.

C++ supports and encourages this programming paradigm as an approach to designing useful, modular, reusable software systems.

Appropriate use of object oriented programming as a design technique will allow us to create and maintain complex software.

Example: Variable Voltage

Suppose we want to model a variable voltage source: an electric component that provides several different voltage output levels.

Let's say it has a selector to choose one of the following levels:
Off, 6V, 12V, 18V, and 24V.

What variables will we need to model this?
Some constants to hold the possible settings.
A variable to represent the current setting.

We also have some functions to represent voltage source behaviour:
Increase the voltage by one level.
Decrease the voltage by one level.
Shut it off.

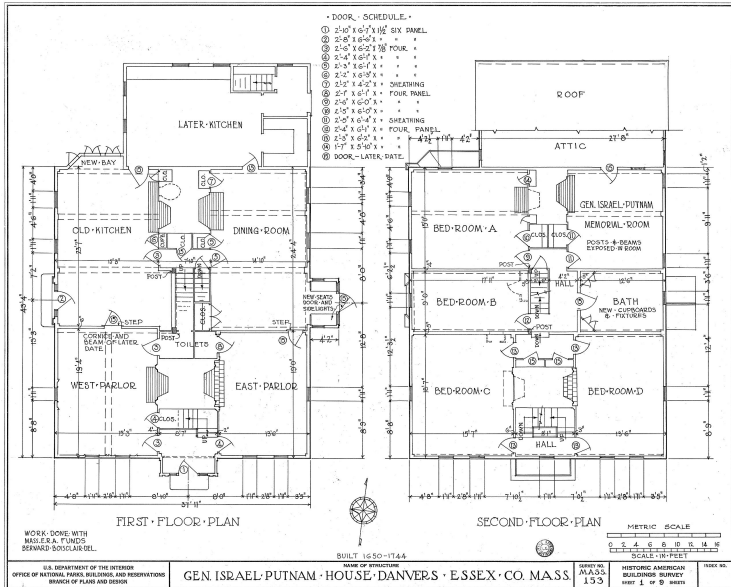
We have an idea now about how to model the variable voltage source.
Now, to represent that in the program.

Analogy: building “blueprints” (architectural plan).

Before a building is built, architects and civil engineers draft a document that describes in detail the structure of the building.

The construction team uses the document when building a structure.
It tells them what to do (though not generally how to do it).

Building Blueprints Example



To tell the system how to build an object, we need to provide it the blueprints for that object.

The blueprints for an object is a **class** definition.

This code tells the system how to create an object of that type.

The `class` definition specifies the variables and methods.

Classes are therefore programmer-defined types.

We've already seen programmer-defined types: the `struct`.

The `struct` can contain different variables of different types.

The `class` is an expanded version of the `struct`: it contains functions in addition to variables.

The data and functions of an object are referred to as **members**.

Member functions are called **methods**.

Member variables are called **fields**.

Let's draw up the blueprints for the voltage source:

```
class voltage_source
{
    public:
        int current_voltage;
        int[] voltageLevels = {0, 6, 12, 18, 24};

        void increase_voltage();
        void decrease_voltage();
        void turn_off();
};
```

The previous slide gave the structure for the voltage source we wanted to model: it showed the member fields and methods.

Given this definition, the system now knows how to make a `voltage_source` object.

The implementations of the functions are not shown for space reasons, but in defining the class, we would normally fill them in.

Of course, defining the structure of the object is the first step, but if we want to use a `voltage_source`, we will need to create one.

A specific realization of a class is called an **instance**.

A program may create many distinct instances of the same class.

Classes are programmer-defined **types**, so this is no surprise.

We can have many distinct `doubles` in the program.

The process of creating an instance of a class is called **instantiation**.

Classes are like structures, so we can use them as types

The declaration `voltage_source source;`
creates an instance of `voltage_source`.

We can then access fields of this instance with the dot operator.

```
source.current_voltage = 0;
```

Let's look at another object example: a car.

Data: make, model, year,...

Functions: `honk_horn()`, `flash_lights()`...

Given a class `Car`, you can make several instances:

Toyota Corolla, Chevrolet Corvette, BMW 335i, Volkswagen Jetta...

Here's a basic version of the Car class, that looks much like a struct:

```
class Car
{
    public:
        string make;
        string model;
        int year;

        void honk_horn();
        void flash_lights();
        // Other methods not shown for space reasons
};
```

As we learn more about OOP, we will improve this definition to be more useful and consistent with good programming practices.

Just like the struct, variables of an instance are accessed using the Dot Operator (.).

```
Car prius;
```

Declares and instantiates a Car object.

After creation with new, we may access the variables of the instance:

```
prius.make = "Toyota";  
prius.model = "Prius";  
prius.year = 2010;
```

Making a Copy of an Instance

If we want a copy, just using the assignment operator will not do what we want.

We must allocate and create another instance.

```
Car prius2;
```

To make a copy, create a new instance, and copy the data from the original object to the new instance, one variable at a time.

(In the next lecture, we'll learn a better way to do this.)

Let's build on the example of the Car class and look at a dealership.

A car dealership has zero or more cars on its parking lot.

A good way to implement this might be an array of Car objects.

Just like the struct we can have an array of objects.

When a sales associate wants to find a car to show a customer, they make the lights flash and the horn honk to find the car in the lot.

Suppose the parking lot has 42 spaces.

```
class Dealership
{
    public:
        int capacity = 42;
        Car[] available_cars[capacity];

        void park( Car new_car, int position );
        void find_car( string make, string model, int year);
        // Other methods and variables not shown
};
```

What happens if someone tries to park a car in a position that is already occupied?

Now let's write a method to find a specific car in the lot when we know the make, model, and year.

```
void Dealership::find_car( string make, string model, int year )
{
    for ( int i = 0; i < capacity; i++ )
    {
        if ( available_cars[i].make == make &&
            available_cars[i].model == model &&
            available_cars[i].year == year)
        {
            available_cars[i].honk_horn();
            available_cars[i].flash_lights();
        }
    }
}
```

This function behaves badly if the array of cars is not initialized!

Also, a new piece of syntax: :: – we will return to this soon.

A common mistake when working with classes is a method call like:

```
Car.honk_horn();
```

The compiler identifies this as an error.

To call the `honk_horn()` method, an instance of the class is needed.

Which instance of `Car` do we mean? The Prius? The Jetta?

Correct usage: `prius.honk_horn();`

Why? These are **instance methods**: they belong to an instance.

What does the static keyword mean? The member applies to the type, not an instance.

The keyword static may apply to a field or method.

Consider this very simple class definition:

```
class ExampleClass
{
    public:
        int value;
        static int count;
};
```

In this class, `value` is an instance member: a new `int` variable is created whenever a new instance of `ExampleClass` is created.

But `count` is a static member: there is only one `int` variable created, regardless of how many instances of `ExampleClass` there are.

This variable is shared between all instances.

It can even be accessed without an instance.

Invalid: `ExampleClass.value = 1;`

Valid: `ExampleClass.count = 1;`

Consider this code:

```
ExampleClass* e1 = new ExampleClass( );  
ExampleClass* e2 = new ExampleClass( );  
  
e1->value = 7;  
e2->value = -8;  
  
e1->count = 2;  
  
cout << "e1 value: " << e1->value << endl;  
cout << "e2 value: " << e1->value << endl;  
cout << "e1 count: " << e1->count << endl;  
cout << "e2 count: " << e2->count << endl;
```

In the previous slide, we used an instance, `e1`, to assign the variable `count`, even though `count` is static.

This practice is discouraged; if a variable is static, it should be accessed using the class name, not an instance reference.

```
ExampleClass.count = 2;
```

The compiler may identify using an instance to assign a static variable as a warning.

A static method can make use of static variables and call other static methods, but cannot use instance variables/methods.

Why? Inside the static method, there is no instance to reference.

An instance method can make use of both instance and static variables and can call both static and instance methods.

Why? Static methods and variables are always available, as they belong to the type itself.

In the `voltage_source` class is an array of different voltage levels:

```
int[] voltageLevels = {0, 6, 12, 18, 24};
```

This is a good candidate to be declared `static` since the voltage levels will be the same in all of the instances of the class.

```
static int[] voltageLevels = {0, 6, 12, 18, 24};
```

In general, any constant is likely to be a good candidate to be declared `static`.