Lecture 10 — Enumerated Types and Structures

J. Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

August 27, 2016

Acknowledgments: W.D. Bishop

Part I

Enumerated Types

Data Representation: Motivation

How might we represent the days of the week in our program?

We might use an integer and define Monday = 1, Tuesday = 2...

Problem 1: Most programmers start counting at zero:
Monday = 0, Tuesday = 1...

Problem 2: some countries consider Sunday the first day of the week.
Sunday = 0, Monday = 1... or is it Sunday = 1, Monday = 2... ?

Problem 3: if later on you see that a day variable contains the value "4", what day of the week does it represent?

Enumerated Types

The concept of Enumerated Types builds upon the simple variable types we have already seen.

In an enumerated type, we define our own “type” in which we specify all the possible values.

The options are a list of constants, and are human-readable.

Instead of “4”, in code we might see “THURSDAY”.

Enumerated Types: Set

An enumerated type is very much like a mathematical set.

A set is a well defined collection of items; duplicates are not allowed.

A mathematical set example: { 1, 3, 5, 7, 9 }

A set does not have to be of numbers; it could be a set of books:
{ “Les Misérables”, “War and Peace”, “A Tale of Two Cities”}

So the set of days of the week is:
{ Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday }

Enumerated Types: Set

Let's turn the set of days of the week into some code.
The format is `enum` *name* { *values* };

```
enum Days { Monday, Tuesday, Wednesday, Thursday,
Friday, Saturday, Sunday };
```

The keyword `enum` indicates this will be an enumerated type.

`Days` is the name of the `enum`.

The seven values inside the { } braces are the members of the set;
the values of the enumeration.

Using an Enumeration

Declaring and initializing an enumeration looks like this:

```
Days today = Days.Monday;
```

The type of the variable in this assignment expression is `Days`, the `enum` type that we defined.

The right side of the assignment operator has to be a value of type `Days`, i.e., one of the items we defined.

So that the compiler knows we mean the "Monday" defined in `Days`, we precede `Monday` with `Days` and a dot (`.`) separates them.

Enumeration and the Switch-Case Statement

The `switch` statement is often used on an `enum`.

```
switch ( today )
{
    case Monday:
    case Tuesday:
    case Wednesday:
    case Thursday:
    case Friday:
        cout << "Time to go to work." << endl;
        break;
    case Saturday:
    case Sunday:
        cout <<  "Day off!" << endl;
        break;
}
```

Enumeration Behind-the-Scenes

Computers really like numbers and they aren't fans of strings.

The computer associates each entry in the set with an integer (and behind the scenes, that's how it thinks about it).

So to the computer, `Monday` is still `0`.

By default, the next item in the list is the previous integer plus 1.

Thus, an `enum` is different from a proper mathematical set in that the order of the elements in the `enum` matters.

Setting enum Values

If you want an enum to start at 1, you can do that:

```
enum Days { Monday = 1, Tuesday, Wednesday, Thursday,
Friday, Saturday, Sunday };
```

If appropriate, you can set some or all of the integer values for the items of an enumeration manually.

```
enum CompassDirection { NORTH = 0, EAST = 90, SOUTH =
180, WEST = 270 };
```

Further Examples of Enumerations

There are lots of situations where enumerated types make sense.

```
enum SalesTax = { NO_TAX, GST, HST };
```

```
enum WeightUnit = { KILOGRAM, POUND, STONE };
```

In some coding conventions, the names of the options are in all-caps, to indicate that they are constants (read-only).

Using Enumerated Types

Use enumerated types when there is a specific list of valid values.

You can always add more options later if it makes sense.

They can make code much easier to understand for human readers.

Part II

Structures

Defining Our Own Types

Looking at enumerated types gave an introduction to the idea of defining our own types.

Consider complex numbers. Recall that a complex number has a real part a and an imaginary part b in the form $a + bj$.

(Notation: in Engineering, the symbol i is used for electric current, so the symbol j is used to denote the imaginary part).

Some complex numbers: 4 + 2j, -3.5 -8j, 9 - 2j.

Without Our Own Types

Solution 1: Have a variable for each.

```
double point1_real = 4;
double point1_imaginary = 2;
double point2_real = -3.5;
double point2_imaginary = -8;
double point3_real = 9;
double point3_imaginary = -2;
```

It has a lot of potential for error and there's no real association between the real and imaginary parts of each number.

Wouldn't it be nice if we could write our own type that associates the real and imaginary parts?

Enter the Structure

The concept for this is the structure, which allows us to have a collection of multiple different types and treat this as a single item.

Like the `enum`, the `struct` definition should be placed outside the `main()` block of statements.

```
struct Complex
{
    double real;
    double imaginary;
};
```

(The complex number example has two `double` variables, but they could be of different types, and we can have as many as needed.)

Structure Terminology

The keyword `struct` indicates we are defining a structure.
Typically, a `struct` name begins with an uppercase letter.

The variables declared inside the { } braces are called the members.

Once a structure is defined, you can use it just like any other type.

The Dot Operator

The dot operator is used to access a member.

In fact, the dot operator is used in several contexts in C++.
As we proceed through the course we'll see them all.

Using the Complex Number Structure

Now let's fill in the declaration and initialization of the points.

```
Complex point1; // Declare Point 1
Complex point2; // Declare Point 2
Complex point3; // Declare Point 3

point1.real = 4;
point1.imaginary = 2;
point2.real = -3.5;
point2.imaginary = -8;
point3.real = 9;
point3.imaginary = -2;
```

This is actually more lines than it was before, but there are some advantages to using a structure.

Structure Assignment

Here's an example of how a structure can save us some work:

```
point2_real = point1_real;
point2_imaginary = point1_imaginary;
```

... compared to...

```
point2 = point1;
```

And this has the effect of setting `point2`'s real and imaginary parts to the values that were in `point1`.

For two variables, this is not a big savings, but imagine a much more complicated `struct` and you can see how this is useful.

Structures within Structures

Are structures within structures allowed? Yes!

```
struct Date
{
    int day;
    int month;
    int year;
};

struct Invoice
{
    int number;
    Date date;
    bool paid;
};
```

Structures within Structures

If we declare `Invoice invoice1;`
then we can access the invoice year like so: `invoice1.date.year`

The dot operator is used twice in that expression. It can be used as many times as needed in the hierarchy of structures.

There's no practical limit to the depth of the hierarchy (although eventually things will get silly).

Re-Use of the Structure

One of the great things about a `struct` is that it can be re-used over and over again in the program.

Using the same `Date struct` we saw on an earlier slide, we can also define the following:

```
struct Cheque
{
    string payee;
    double amount;
    int number;
    Date date;
};
```

This allows us to re-use common elements, like dates.

Re-Use of the Structure

This, however, is an error:

```
struct Invoice
{
    int number;
    Date date;
    bool paid;
    Invoice previousInvoice;
};
```

The compiler can't figure out what to do with this; it would get stuck in an infinite loop trying to compile this.