Types and Structures
ECE2036

Lecture 8
The **Built-in Types**

We have discussed in detail, and used the so-called *built-in* types. These are variable types that are pre-defined by the compiler. We have used:

1. `int`
2. `unsigned`
3. `double`
4. `char`
5. `string` (*string is not technically a built-in type*)
6. `bool`  (*Simple True/False Value*)
7. **Pointers** to each of the above.
Defining New Types

1. We can either re-define existing types, or create completely new types using the `typedef` keyword.
   
   ```
   typedef int Coord_t;
   typedef unsigned Size_t;
   typedef double Meters_t;
   typedef char* CharPtr_t;
   ```

2. Note that the 
   
   The name can be any legal C/C++ identifier.

3. Each of the above defines a new type that can be used anywhere a built-in type name is expected:
   
   ```
   Coord_t xPosition;
   Size_t arraySize;
   Meters_t distance;
   CharPtr_t myAddress = "266 Ferst Dr";
   ```

4. Simply re-defining the built-in types as shown above is useful for writing readable code, it is not strictly necessary, and in fact many C/C++ programmers do not do this.
The *Enumeration* type

1. It is often convenient to define a series of names that are synonyms for numeric values. This is done by defining an `enum` type.

   ```c
   typedef enum { NONE, SUSPENDED, RESUMED } TimerState_t
   ```

2. This defines a new type `TimerState_t`, and each of the symbols:
   ```c
   NONE = 0
   SUSPENDED = 1
   RESUMED = 2
   ```

3. We can then use new type to define a variable:
   ```c
   TimerState_t backoff = NONE;
   ```

4. We can assign values to the variable:
   ```c
   backoff = SUSPENDED;
   ```

5. We can check the value:
   ```c
   if (backoff == SUSPENDED)
   ```

6. We can compare the value:
   ```c
   if (backoff < SUSPENDED)
   ```
The *Enumeration* type continued

1. We can also assign specific values to each of the identifiers in an enumeration type:

   ```c
   typedef enum { SEND_PKT = 100, START_GEN, STOP_GEN, CHECK_CONN } AppBlueEvent_t;
   ```

2. This defines a new type `AppBlueEvent_t`, and each of the symbols:

   ```c
   SEND_PKT = 100
   START_GEN = 101
   STOP_GEN = 102
   CHECK_CONN = 103
   ```

3. `typedef enum { FIN = 0x01, SYN = 0x02, RST = 0x04, PSH = 0x08, ACK = 0x10, URG = 0x20 } TCPFlags;`

   Note the use of the *hexadecimal* constants above, using the `0x` prefix.
Defining *Structures*

1. Often we have a set of variables that are logically related in some way, and would naturally always be accessed or modified as a group.

2. For example:

   ```
   string name;
   int streetNumber;
   string streetName;
   string city;
   string state;
   int zipCode;
   ```

3. Another example:

   ```
   Coord_t x;
   Coord_t y;
   ```
Defining Structures, continued

In these cases it is convenient to define a **structure** which encapsulates all of the related variables into a *single* variable. This is done with `typedef struct`.

```c
typedef struct
{
    Coord_t x;  // Current x position
    Coord_t y;  // Current y position
} Point_t;

typedef struct
{
    string name;
    int    streetNumber;
    string streetName;
    string city;
    string state;
    int    zipCode;
} EmployeeInfo_t;

The above defines *new types* called **Point_t** and **EmployeeInfo_t**.

This *does NOT* declare any variables.
Once we have defined a new structure type, we can then create variables of that type.

Point_t currentScreenLocation;

The above defines a variable of type Point_t.

The currentScreenLocation variable contains two sub-variables, x and y.

Clearly, when a structure variable has been created, we need to individually access the sub-variables in the structure. This is done with the period symbol as follows:

Point_t currentScreenLocation;
currentScreenLocation.x = 100;
currentScreenLocation.y = 200;

You can pass the structure variable as an argument to a subroutine. The entire structure is passed, and the subroutine can access all of the sub-variables.

void Sub1(Point_t currentPoint)
{
    PrintLn("Current Point x is ", currentPoint.x);
    PrintLn("Current Point y is ", currentPoint.y);
}

// Elsewhere in the program;
Point_t currentScreenLocation;
currentScreenLocation.x = 100;
currentScreenLocation.y = 200;
Sub1(currentScreenLocation);
Pointers to Structures

1. We can declare variables that are *pointers to structures*, just like pointers to any other types. This is often done in passing structure arguments to subroutines, as it is more efficient to pass a pointer than the entire structure if the structure is large.

2. Sub1(Point_t* pCurrentPoint)

   {
   // De-reference pointer and get x coordinate
   PrintLn("x is ", (*pCurrentPoint).x);
   // A short cut way to do this
   PrintLn("x is ", pCurrentPoint->x);
   }

   // Elsewhere
   Point_t currentPoint;
   currentPoint.x = 10;
   currentPoint.y = 20;
   Sub1(&currentPoint); // Note address-of operator