Enumeration Types

Eventually we will talk about creating our "own" types, but initially, we'll talk about how to create new atomic/primitive type that is a subset of a given atomic type in C. (Some atomic types in C are int, float, double, char, etc.)

Typically, enumerated types are used to enhance a program's readability. Consider the following declaration example:

```
typedef enum { North, East, South, West }
directionT;
```

Once you do this, you can declare a variable of the type directionT:

```
directionT housedir;
```

Internally, any enumerated type declared this way has corresponding integer values stored, starting at 0. (Thus in our example, North is 0, East is 1, South is 2, and West is 3.) However, we can specify the internal integer values of enumerated types as follows:

```
typedef enum {
  penny = 1,
  nickel = 5,
  dime = 10,
  quarter = 25,
  halfdollar = 50
} coinT;
```

Why use enumerated types?
1) Each code/value does not need to be specified explicitly.
2) Code readability, which is also useful in debugging.

Data and Memory

bit - a single zero or one. The word comes from the contraction of "binary digit."

byte - eight consecutive bits. Can store a char.

word - the size of this depends on the computer, but they are almost always either 2 or 4 bytes. Usually stores an int.

You can imagine memory as a long array of numbered cells. For example, if we had 4 megabytes, then we would have memory locations (each storing a byte) numbered from 0 to $4x2^{20}$-1. If we execute the line

```
ch = 'A';
```

and the variable ch happened to be stored in memory location 1000, our picture would look like this:

An integer would be stored over 4 consecutive bytes, and the address of the variable storing the integer would be the first memory address the variable was stored in.

In certain situations, it's useful to know the size of a particular variable or type/struct. The following expression returns the desired information:

```
sizeof(int)
```
or

```
sizeof x
```

Pointers

The designers of the C programming language WANTED the programmer to have the ability to directly manipulate memory. Pointers allow this. In other languages (such as Java), pointers are hidden from the user. In C, this feature is a mixed blessing. The programmer has more control, but also has to take responsibility for that control.

Here are some of the reasons to use pointers in C:

1) Allow you to refer to a large data structure in a compact way.

2) Allow you to share data efficiently among functions.

3) Allow you to dynamically reserve memory.

4) Allow you to record relationships between data.

l-value: An internal memory location capable of storing data.

Here are some rules that l-values adhere to:

1) Every l-value is stored somewhere in memory and has an address.

2) Once it's been declared the address of an l-value never changes.

3) Different l-values require different amounts of memory.

4) The address of an l-value is a pointer value.

Basic Pointer Syntax

We can declare a pointer as follows:

```
int *p;
```

There are two key pointer operators:

***p: Dereferences a pointer, returns the value stored at the memory location p is pointing to.**
&x: Returns the address of where the variable x is stored.

We can observe the key difference between these two operators by tracing through the following segment of code:

```
int x, y;
int *p1, *p2;
x = -42;
y=163;
p1 = &x;
p2 = &y;
*p1 = 17;
p1 = p2;
p2 = &x;
*p1 = *p2;
```

When we want to specify that a pointer is NOT pointing to any variable, we can set it to NULL as follows:

```
p = NULL;
```

It's important not to dereference a null pointer. (This means saying p->something, when p isn't pointing to anything.) Doing so will lead to a run-time error.
Of course, one of the key uses of pointers is passing parameters by reference.

Consider the functions:

```
void swap1(int x, int y) {
  int temp;
  temp = x;
  x = y;
  y = temp;
}

void swap2(int *p1, int *p2) {
  int temp;
  temp = *p1;
  *p1 = *p2;
  *p2 = temp;
}
```

When called properly, one of these performs its task while the other one does not. Why?

Arrays

An array is a collection of variables that have two key things in common:

1) It is ordered.
2) All the variables in an array are of the same type.

We define an array as follows:

```
int scores[10];
```

Note that only constant expressions (that signify the size of the array) can appear in between the brackets. To index a particular variable/element of the array do as follows:

```
scores[4] = 120;
```

The value in the brackets is the allocated size of the array. However, sometimes, even if the user allocates a certain amount of space, sometimes they end up using less. The amount of space actually used is known as the effective size of the array. One would need to declare an extra integer variable to keep track of this.

Since the name of an array is equivalent to a pointer to the beginning of an array, arrays are always passed by reference.

Here is one way to initialize an array upon declaration:

```
int scores[] = {87, 99, 75, 88, 93, 56, 77, 84,
89, 79};
```

Multidimensional Arrays

You can declare an array with more than one dimension. Often times, it's easier to visualize data stored in tabular form, so conceptually, a two-dimensional array often times makes code easier to read. Regardless, technically speaking, a 2D array is still stored in one dimension in memory. Consider the following example:

```
int grid[4][3];
```

You index into a 2D array as you might expect:

```
grid[0][1] = 5;
```

One key item to note is that all elements on a single "row" of an array appear in consecutive memory locations, but not all elements in a single "column" do.

Pointers and Arrays

Pointer arithmetic allows you to navigate through an array using a pointer. Consider the following example:

```
double values[10];
double *p;

p = &values[9];
while (p >= &values[0]) {
  scanf("%lf", p);
  p = p-1;
}
```

Technically speaking, even though each double is separated by 8 bytes in memory, when we increment or decrement a pointer, it AUTOMATICALLY increments or decrements by the size of the type it is pointing to.

One key item to note is that the array name itself IS a pointer. So the two expressions:

```
values      AND     &values[0]
```

are identical. This helps explain exactly how arrays are passed in C.

So what is the difference between a pointer and an array?

While `double values[10];` allocates 10 locations to store doubles, `double *p;` does not.

Instead this only allocates the memory to store a single location in memory. However, we CAN use a pointer to dynamically allocate memory!

Structs

When the data you want to store doesn't fit neatly into a predefined C type, you can create your own. Here is an example:

```
struct employee {
  char[20] name;
  double salary;
  int empID;
};
```

Now, to declare a variable of this declared type, we write:

```
struct employee officeworker;
```

To access the fields of a struct employee variable do as follows:

```
officeworker.empID = 1;
```

Often times, for efficiency purposes, it's easier to declare pointers to structs and use those to manipulate the structures formed. Consider the following:

```
struct employee *temp;
temp = &officeworker;
(*temp).salary = 50000;
```

Since expressions similar to the last one are used so often, there is a shorthand to access a field of a record through a pointer to that record. The last statement can also be written as:

```
temp->salary = 50000
```

Another reason to use a pointer to a struct is to dynamically allocate the memory for the struct. This allows the memory to be allocated beyond the "life"/"scope" of the function within which it was declared. This will be covered more specifically in the lecture notes titled, "Dynamic Memory Allocation."