Week 6 - Wednesday

COMP 2400

Last time

- What did we talk about last time?
- Exam 1!
- Before that:
  - Strings

Questions?

Project 3

Quotes

For a long time it puzzled me how something so expensive, so leading edge, could be so useless. And then it occurred to me that a computer is a stupid machine with the ability to do incredibly smart things, while computer programmers are smart people with the ability to do incredibly stupid things. They are, in short, a perfect match.

Bill Bryson

Pointers

Pointers

- A **pointer** is a variable that holds an address
- Often this address is to another variable
- Sometimes it's to a piece of memory that is mapped to file I/O or something else
- Important operations:
  - Reference (**&**) gets the address of something
  - Dereference (*****) gets the contents of a pointer

Declaration of a pointer

- We typically want a pointer that points to a certain kind of thing
- To declare a pointer to a particular type

```
type * name;
```

- Example of a pointer with type **`int`**:

```
int * pointer;
```

Whitespace doesn't matter!

- Students sometimes get worried about where the asterisk goes
- Some (like me) prefer to put it up against the type:

```
char* reference;
```

- Some like to put it against the variable:

```
char *reference;
```

- It is possible to have it hanging in the middle:

```
char * reference;
```

- Remember, whitespace doesn't matter in C

Reference operator

- A fundamental operation is to find the address of a variable
- This is done with the reference operator (**&**)

```
int value = 5;
int* pointer;
pointer = &value; // pointer has value's address
```

- We usually can't predict what the address of something will be

Dereference operator

- The reference operator doesn't let you do much
- You can get an address, but so what?
- Using the dereference operator, you can read and write the contents of the address

```c
int value = 5;
int* pointer;
pointer = &value;
printf("%d", *pointer); // prints 5
*pointer = 900; // value just changed!
```

Aliasing

- Java doesn't have pointers
  - But it does have references
  - Which are basically pointers that you can't do arithmetic on
- Like Java, pointers allow us to do aliasing
  - Multiple names for the same thing

```
int wombat = 10;
int* pointer1;
int* pointer2;
pointer1 = &wombat;
pointer2 = pointer1;
*pointer1 = 7;
printf("%d %d %d", wombat, *pointer1, *pointer2);
```

Pointer arithmetic

- One of the most powerful (and most dangerous) qualities of pointers in C is that you can take arbitrary offsets in memory
- When you add to (or subtract from) a pointer, it jumps the number of bytes in memory of the size of the type it points to

```c
int a = 10;
int b = 20;
int c = 30;
int* value = &b;
value++;
printf("%d", *value); // What does it print?
```

Arrays are pointers too

- An array **is** a pointer
  - It is pre-allocated a fixed amount of memory to point to
  - You can't make it point at something else
- For this reason, you can assign an array directly to a pointer

```
int numbers[] = {3, 5, 7, 11, 13};
int* value;

value = numbers;
value = &numbers[0]; // Exactly equivalent

value = &numbers; // What about this?
```

Surprisingly, pointers are arrays too

- Well, no, they aren't
- But you can still use array subscript notation (`[]`) to read and write the contents of offsets from an initial pointer

```
int numbers[] = {3, 5, 7, 11, 13};
int* value = numbers;

printf("%d", value[3] );       // Prints 11
printf("%d", *(value + 3) ); // Prints 11
value[4] = 19; // Changes 13 to 19
```

Don't try this at home

- We can use a pointer to scan through a string

```
char s[] = "Hello World!"; // 13 chars

char* t = s;
do
{
  printf("(%p): %c %3d 0x%X\n", t, *t,
         (int)*t, (int)*t);
} while (*t++); // Why does this work?
```

Or what if we pretend…

- That it's an **`int`** pointer

```
char s[] = "Hello World!"; // 13 chars
int* bad = (int*)s; // Unwise...

do
{
  printf("(%p): %12d 0x%08X\n", bad, *bad, *bad);
} while (*bad++);
```

`void` pointers

- What if you don't know what you're going to point at?
- You can use a **`void*`**, which is an address to … something!
- You have to cast it to another kind of pointer to use it
- You can't do pointer arithmetic on it
- It's not useful very often

```
char s[] = "Hello World!";
void* address = s;
int* thingy = (int*)address;
printf("%d\n", *thingy);
```

Why do we care about pointers?

- There are some tricks you can do by accessing memory with pointers
- You can pass pointers to functions allowing you to change variables from outside the function
- Next week we're going to start allocating memory dynamically
  - Arrays of arbitrary size
  - Structs (sort of like classes without methods)
- We need pointers to point to this allocated memory

Functions that can change arguments

- In general, data is passed **by value**
- This means that a variable cannot be changed for the function that calls it
- Usually, that's good, since we don't have to worry about functions screwing up our data
- It's annoying if we need a function to return more than one thing, though
- Passing a pointer is equivalent to passing the original data **by reference**

Example

- Let's imagine a function that can change the values of its arguments

```
void swapIfOutOfOrder(int* a, int* b)
{
  if (*a > *b)
  {
     int temp = *a;
     *a = *b;
     *b = temp;
  }
}
```

How do you call such a function?

- You have to pass the addresses (pointers) of the variables directly

```
int x = 5;
int y = 3;
swapIfOutOfOrder (&x, &y); // Will swap x and y
```

- With normal parameters, you can pass a variable or a literal
- However, you **cannot** pass a reference to a literal

```
swapIfOutOfOrder (&5, &3); // Not allowed
```

Command Line Arguments

Strings

- Before we get into command line arguments, remember the definition of a string
  - An array of **`char`** values
  - Terminated with the null character
- Since we usually don't know how much memory is allocated for a string (and since they are easier to manipulate than an array), a string is often referred to as a **`char*`**
- Remember, the only real difference between a **`char*`** and a **`char`** array is that you can't change where the **`char`** array is pointing

Command line arguments

- Did you ever wonder how you might write a program that takes command line arguments?
- Consider the following, which all have command line arguments:

```
ls -al
chmod a+x thing.exe
diff file1.txt file2.txt
gcc program.c -o output
```

Getting command line arguments

- Command line arguments do **not** come from **`stdin`**
- You can't read them with **`getchar()`** or other input functions
- They are passed directly into your program
- But how?!

You have to change **`main()`**

- To get the command line values, use the following definition for **`main()`**

```
int main(int argc, char** argv)
{

   return 0;
}
```

- Is that even allowed?
  - Yes.
- You can name the parameters whatever you want, but **`argc`** and **`argv`** are traditional
  - **`argc`** is the number of arguments (argument count)
  - **`argv`** are the actual arguments (argument values) as strings

Example

- The following code prints out all the command line arguments in order on separate lines

```
int main(int argc, char* argv[])
{
  for(int i = 0; i < argc; i++ )
    printf("%s\n", argv[i]);

  return 0;
}
```

- Since **`argv`** is a **`char**`**, dereferencing once (using array brackets), gives a **`char*`**, otherwise known as a string

Command line example

- Let's write a program that
  - Expects exactly one command line flag
  - If the flag is:
    - **`-y`** Print "**`yak`**"
    - **`-c`** Print "**`cormorant`**"
    - **`-t`** Print "**`Tasmanian devil`**"
  - For any other argument, we should print "**`Unknown animal`**"
  - If there is not exactly one command line argument (after the program name), print:

```
"Usage: program [-y | -c | -t ]"
```

Upcoming

Next time…

- More on pointers

Reminders

- Work on Project 3
- Keep reading K&R Chapter 5