CSE378 - Lecture 4

- Announcements
  - HW1 out

- Today:
  - Finish-up control-flow
  - IFs
  - loops
  - case/switch
  - Array indexing vs. Pointers
  - In particular pointer arithmetic
  - String representation

---

Control-flow Example

Let's write a program to count how many bits are set in a 32-bit word.

```c
switch (two_bits) {
  case 0:
    break;
  case 1:
    // Fall-through
    count++;
    break;
  case 2:
    count = 2;
    break;
  case 3:
    count = 3;
    break;
}
```

- We could just translate the code to if, thens, and elses:

```c
if (two_bits == 0) {
  count = 0;
} else if (two_bits == 1) {
  count = 1;
} else if (two_bits == 2) {
  count = 2;
} else if (two_bits == 3) {
  count = 3;
}
```

- This isn't very efficient if there are many, many cases.

---

Translating an if-then-else statement

If there is an else clause, it is the target of the conditional branch.
- And the then clause needs a jump over the else clause

```c
// Increase the magnitude of V0 by one
if (two_bits == 0) {
  // Do nothing
} else if (two_bits == 1) {
  // Count
  count++;
} else if (two_bits == 2) {
  // Move V0, V0
} else if (two_bits == 3) {
  // Move V0, V0
}
```

- Drawing the control-flow graph can help you out.

---

Case/Switch Statement

- Many high-level languages support multi-way branches, e.g.

  ```c
  switch (two_bits) {
    case 0:
      break;
    case 1:
      // Fall-through
      count++;
      break;
    case 2:
      count = 2;
      break;
    case 3:
      count = 3;
      break;
  }
  ```

- Alternatively, we can:
  1. Create an array of jump targets
  2. Load the entry indexed by the variable two_bits
  3. Jump to that address using the jump register, or `jr` instruction

---

Representing strings

- A C-style string is represented by an array of bytes.
  - Elements are one-byte ASCII codes for each character.
  - A 0 value marks the end of the array

```c
\"\" 0x0
\"\" 0x1
\"\" 0x2
\"\" 0x3
```

---

1
Null-terminated Strings

- For example, "Harry Potter" can be stored as a 13-byte array:

\[
\begin{array}{cccccccccccc}
\text{72} & \text{97} & \text{116} & \text{114} & \text{116} & \text{32} & \text{99} & \text{116} & \text{114} & \text{114} & \text{0} \\
H & a & r & y & p & t & t & o & r & r & n & t
\end{array}
\]

- Since strings can vary in length, we put a 0, or null, at the end of the string. This is called a null-terminated string.

- Computing string length:
  - We'll look at two ways.

Array Indexing Implementation of strlen

```c
int strlen(char *string) {
    int len = 0;
    while (string[len] != 0) {
        len++;
    }
    return len;
}
```

```
string[10] = x (strlen = 8)
```

What does this C code do?

```c
int foo(char *s) {
    int L = 0;
    while (s++) {
        L++;
    }
    return L;
}
```

```
0 \times 1 = 0
0 \times 2 = 0
0 \times 3 = 0
```

Pointers & Pointer Arithmetic

- Many programmers have a vague understanding of pointers.
  - Looking at assembly code is useful for their comprehension.
    - (But if you have an aggressive optimizing compiler, you may see the same assembly code for both versions!)

```c
int strlen(char *string) {
    int len = 0;
    while (string[len] != 0) {
        len++;
    }
    return len;
}
```

```
string[10] = x (strlen = 8)
```

What Is a Pointer?

- A pointer is an address.
- Two pointers that point to the same thing hold the same address.
- Dereferencing a pointer means loading from the pointer's address.
- In C, a pointer has a type, the type tells us what kind of data to do
  - Use load byte (b) for char`
  - Use load half double (h) for short`
  - Use load word (w) for int`
  - Use load single precision floating point (f) for float`
- Pointer arithmetic is often used with pointers to arrays.
  - Incrementing a pointer (i.e., ++) makes it point to the next element.
  - The amount added to the pointer depends on the type of pointer:
    - Pointer = pointer + sizeof(pointer's type)
      - 1 for char', 4 for int', 4 for float', 8 for double'

```c
char *p;  
int *q;

p = (char *) malloc(10);  
q = (int *) malloc(5);
```

What is really going on here...

```c
int strlen(char *string) {
    int len = 0;
    while (*string != 0) {
        string++;
        len++;
    }
    return len;
}
```
Pointers Summary

- Pointers are just addresses!
  - "Pointers" are locations in memory
- Pointer arithmetic updates the address held by the pointer
  - "string ++" points to the next element in an array
  - Pointers are typed so address is incremented by sizeof(pointee)