

More C

CS 350, Lecture 2

A. Why?

- C is a high-level language that maps nicely to low-level data and operations

B. Outcomes

At the end of today, you should:

- Understand some of the basic differences between C and more contemporary languages like C++ and Java.

C. More C

Conditional Compilation

- At compile time, decide whether to insert some code or not.
- In Java, you use if (constant true or false)
 - if (DEBUG) print whatever...
 - Compiler optimizes away if (false) code
- In C, use conditional compilation
- #if, #ifndef, and #ifdef ... #endif (if true, if not defined, if defined)

```
#ifdef DEBUG
    printf("DEBUG message\n");
#endif
```

- Often used in header files to avoid repeating included code

```
#ifndef THING
#define THING
.... code for THING ...
#endif
```

C Compiler and Link Editor

- Compiling a C program yields an object file (Unix *.o file); includes symbol table of “external” variables defined in other files.
- To get complete program, must combine object files and reconcile external references.
- Done by link editor (“linker”).
- On Unix, produces executable file `a.out` by default.

Main program

- Main program returns an integer 0 on successful execution, non-0 for errors.
- Standard header takes two arguments to indicate command-line options.

- We'll ignore this for a while and just use `void` as the parameter list (indicates a function with no parameters).

Standard Input/Output

- C standard library includes I/O functions (names & types declared in `stdio.h`).
- Printing done with `printf` function (takes variable number of arguments).
- First argument is a formatting string; remaining arguments are values to print.

```
#include <stdio.h>
int main(void) {
    int sum = 43+59;
    printf("%d plus %d = %d (decimal), %x (hex), %c (as a char)\n",
        43, 59, sum, sum, sum);
    return 0;
}
```

- The `%d`, `%x`, `%c` are formatting directions.
- Reading is done with `scanf` function.

```
#include <stdio.h>
int main(void) {
    int myIntVar;
    float myFloatVar;
    printf("Enter an int and a float:");
    scanf("%d %f", &myIntVar, &myFloatVar);
    return 0;
}
```

- Read a decimal number into `myIntVar`, one or more whitespace characters, and a floating point number into `myFloatVar`.
- The `&` indicates “address of” — we’ll study this later with C pointers.

Some General Properties of C

- We'll go over these more as we need
- Datatypes include `int`, `short`, `long`, `float`, `double`, `char` (typically 32, 32, 32, 64, and 8 bits) but is implementation-dependent.
 - There are signed and unsigned versions of these (the default is signed: `int` means `signed int`).
- For booleans, use integers (0 for false, non-0 for true). Comparison operators return 0 or 1. E.g., `int flag = (x > 0)`; either sets `flag` to 0 or 1.
 - Note `if (x == y == z)` is legal but doesn't do what you'd like.

- Equality testing is left-associative, so we compare x and y for equality (get 0 or 1) and compare 0 or 1 with z .
- In C99, there's a `bool` type and symbolic constants `true` and `false`, but these are just names for integers 1 and 0.
- Strings are represented as character arrays with an extra `'\0'` character to terminate the string.
- Arrays don't know their size (there's no runtime length operation as in Java).
- There aren't classes, constructors, or objects in C; there's no public/private encapsulation, no inheritance, no interfaces.
- There are structures (similar to classes where everything's public)
- There's no built-in memory allocation via `new` or garbage collection — it's up to the programmer to allocate space dynamically and free it (return it back to the operating system).

Binary Integers

D. Unsigned Binary Integers

- Our hardware represents data using bits; we use bits to represent binary numbers.
- Given n bits, there are 2^n possible bit patterns.
 - We can use them to name 2^n possible items.
 - For unsigned binary integers, we read the n -bit string as a base 2 number that's ≥ 0 .
 - E.g. for 3 bits: 000, 001, 010, 011, 100, 101, 110, 111 are 0 through 7.
 - The bit positions (left to right) are $2^{n-1}, \dots, 2^2, 2^1, 2^0$.
 - So the largest unsigned n -bit number (n 1's), represents $2^{n-1} + \dots + 2^1 + 2^0 = 2^n - 1$.
- Unsigned binary addition is like decimal addition: Add right to left, add carry to left if necessary. Unsigned binary subtraction is like decimal subtraction (borrow from left).

E. Signed Binary Integers

- With signed integers, the leftmost bit used as the sign bit (0 for nonnegative numbers, 1 for negative numbers).
- To get symmetry, we'd like to represent (for some k), the $2k+1$ numbers $-k, -k+1, -k+2, \dots, -1, 0, 1, 2, \dots, k-1, k$.
 - This is an odd number, and n bits gives us 2^n bit patterns.
- Any scheme for representing negative numbers is either going to have
 - Unequal numbers of positive and negative integers

- Multiple representations of the same number. (Typically two versions of zero: “positive” zero and “negative” zero.)
- We’ll see three methods for representing negative numbers and for taking the negative of a number. (All three methods represent positive numbers the same way.)

F. Sign-Magnitude Negative Numbers

- To take the negative of a number, flip its sign bit. (Easy!)
 - Example: 111 is 3-bit unsigned 7, so 0111 represents 7; In sign-magnitude, 1111 represents -7 .
- In sign-magnitude, adding a positive and negative number is different from adding two unsigned numbers.
- A problem with sign-magnitude is that it has a “negative” zero: 1000 (in addition to “positive” zero: 0000).

G. One’s complement

- In one’s complement, to take the negative of a number, flip all its bits.
 - E.g. the negative of 7 is the bit-flip of 0111, which is 1000. Also, the negative of (-7) is the bit-flip of 1000 is 0111.
- Another way to think of taking the negative (in one’s complement) involves subtracting each bit from 1.
 - I.e., to calculate $-k$, let N be n one bits, and do an unsigned subtraction of $N-k$.
 - E.g., to calculate -0111 , do the unsigned subtraction $1111-0111 = 1000$. To calculate -1000 , do the unsigned subtraction $1111-1000 = 0111$.
 - -0101 is $1111-0101 = 1010$, and -1010 is $1111-1010 = 0101$.
 - In decimal, “Nine’s complement” corresponds to binary one’s complement.
 - E.g., the negative of 1234 = $9999-1234 = 8765$ [unsigned decimal]. The negative of 8765 is $9999 - 8765 = 1234$.
- In one’s complement, adding a positive and negative number is different from adding two unsigned numbers.
- A problem with one’s complement is that it has a “negative” zero: 1111 (in addition to “positive” zero: 0000).

H. Two’s complement

- In two’s complement, to take the negative of a number, take the one’s complement and add 1 (as unsigned addition).
 - E.g., the negative of 7 is $(\text{bit-flip of } 0111)+1 = 1000+1 = 1001$. The negative of -7 is $(\text{bit-flip of } 1001)+1 = 0110+1 = 0111$.
- There’s also a shortcut for taking the negative in two’s complement:

- Take our bitstring; going right to left, skip over zeros until you reach a 1, then flip all bits to the left of that 1.
- E.g., for the negative of 0110, we keep the rightmost 10 and flip the 01 to its left, which gives us 1010. For the negative of 1010, we keep the rightmost 10 and flip the 10 to its left, which gives us 0110.
- Note for negative 0111, we skip over no zeros and flip to get 1001. For negative 1001, we skip over no zeros and flip to get 0111.
- Another way to think of taking the negative in two's complement is that $-k$ is represented as $N-k+1$ (where N is n one bits and the subtraction and addition are done as unsigned operations).
 - Note: As an unsigned number, n one bits represents 2^n-1 , so $N-k+1$ is $(2^n-1)-k+1$ (all using unsigned operations), which equals 2^n-k (using unsigned subtraction).
 - The power 2^n is why this is called 2's complement.
 - In decimal, "ten's complement" corresponds to binary two's complement.
 - E.g., the negative of 1234 = $9999 - 1234 + 1 = 8765 + 1 = 8766$. The negative of 8766 is $9999 - 8766 + 1 = 1233 + 1 = 1234$.
- Two's complement only has one kind of zero.
 - The negative of 0000 is $1111 - 0000 + 1 = 1111 + 1 = 0000$ (we throw away the carry out of the leftmost position).
- The problem with two's complement is that it has one more negative number than positive number, and the bitstring 100...0 is its own negative.
 - E.g., negative 1000 is $0111 + 1 = 1000$.
 - Two's complement 1000 represents decimal -8 ; 1001 represents -7 (and 0111 represents 7).
- **The really nice feature** of 2's complement (and the reason it's used in hardware) is that two's complement subtraction is the same as unsigned addition of the negative. I.e., in two's complement $x - y$ can be implemented as $x + (-y)$, where the $+$ is unsigned.
 - E.g., $6 - 4$ is $0110 + (-0100) = 0110 + (1011 + 1) = 0110 + 1100 = 0010$ (there's a carry in and carry out of the leftmost position). In the other direction, $4 - 6 = 0100 + (-0110) = 0100 + (1001 + 1) = 0100 + 1010 = 1110$, which represents -2 .

I. Overflow

- Overflow: Result of operation on n bits doesn't fit into n bits.
- In general, if x and y have different signs, then $x + y$ won't overflow.
 - Overflow occurs when you try to go too far from zero.
- When can overflow occur in two's complement?
 - Taking the negative of the most negative number.
 - Symptom: k is negative and so is $-k$.

- Adding together two positive or two negative numbers and getting a result of the other sign.
 - Symptom: positive + positive = negative result or negative + negative = positive result.
 - More generally, (if you add two numbers of the same sign), if the carry in to the leftmost position doesn't equal the carry out of the leftmost position, then overflow has occurred.
 - If you carry in a 1 and carry out a 0, the sign has changed from + to -.
 - If you carry in a 0 and carry out a 1, the sign has changed from - to +.