CMSC 216
Introduction to Computer Systems
Lecture 2
Introduction to C

Administrivia

- Read Chapters 2 and 3 of Reek
- Class examples available in grace at:

  /afs/glue.umd.edu/class/summer12011/cmsc/216/0101/public/LectureExamples
- TA will illustrate how to get to this folder in lab

Chapter 2, Reek

Basic Concepts

Overview

- Let's first go over the concepts discuss in this lecture via a set of examples
- Compilation
  - gcc –Wall file.c
  - Executable a.out
- Fundamentals
  - **Example:** evenValues.c
  - **Example:** computeGrade.c

Compilation stages

- Preprocessor
  - used to make sure the program parts see declarations they need (and other purposes too, e.g., macros)
  - directives begin with a **#** (pound sign)
  - do not end in a ; like C statements do
- Translation
  - makes sure individual parts are consistent within themselves
  - creates an object (.o) file
- Linking
  - joins one or more compiled object files together
    - includes matching function call to callee across separate files
    - and matching global variable use to where it's defined
  - makes sure caller/callee consistent with each other
  - creates an executable file (the convention in Unix is no filename extension is used for executable files, although sometimes .x or .exe are used). Default: a.out

Basic C syntax is similar to Java

- Variable declarations
  - format: *typename variable1, variable2;*
  - each variable can optionally have an initializer
  - examples:

```
int a;
float x, y;
int m = 8;
```

  - if appearing in a function, must appear before executable code
  - may be local to a function, or global (remainder of the same file)
- Legal variable names similar to Java
- Looping and conditionals
  - **`if`**, **`switch`**
  - **`while`**, **`do`**/**`while`**, **`for`**
- Function calls
  - format: *functionname* **`(`***argument1, argument2, …*`)`
- Compound statement
  - can go anywhere a single statement goes
  - surrounded by **`{ }`**
  - can have block local variables declared at beginning

Output – the **`printf()`** function

- Definition
  - its declaration is included with **`#include <stdio.h>`**
  - then just use it because its definition is in the C standard I/O library that the compiler always links for you
- Syntax:
  - **`printf("`***formatstring*`",`** *expression1*, *expression2*, *expression3*, ...**`);`**
  - a function with a variable number of arguments!
- Within the format string, normal characters print as themselves, and:
  - to print values of expressions use format specifiers, for example:

    `%d` print an integer in decimal

    `%f` print a floating point value

    `%c` print a character

    `%s` print a character string
  - there are also special *escape sequences*, similar to Java:

    `\n` print a newline

    `\t` print a tab
  - each format specifier in the *formatstring* is substituted by the corresponding expression, and the *formatstring* is printed

Input – the **`scanf()`** function

- Definition
  - its declaration is also included with **`#include <stdio.h>`** (it's also in the C standard I/O library)
- Syntax:
  - **`scanf("`***formatstring*`", &`***variable1*`, &`***variable2*`, &`***variable3*`, ...);`**
  - the format string has the same basic structure as in **`printf()`**
  - to match data entered in the input, use the same kinds of formatting expressions and format specifiers in the format string as for **`printf()`**:

    normal characters match themselves

    `%d` match a decimal integer

    `%f` match a floating point value

    `%c` match a character
  - arguments (variable names) indicate where converted input is to be stored, and their types should match the format specifiers
  - be careful that the variable names are all *preceded by* **`&`** (we'll discuss why)
- The format string matches against characters in the input stream until the last specifier is matched, or until matching fails
- Returns number of specifiers that are matched **and** assigns to variables

Chapter 3, Reek

DATA

Data types in C

- C has four basic types: integers, floating point numbers, pointers, and aggregate types (e.g., arrays and structures)
  - The first three are considered scalar types
- Integers include characters as well as the numbers you commonly think of
- Integer types have signed and unsigned versions
- Floating point types have (mostly) the same names as in Java: **`float`** and **`double`**

Data sizes in C

Type name	Minimum size	Size on `linux.grace`
`char`	1	1
`short`	2	2
`int`	2	4
`long`	4	8
`float`	4	4
`double`	8	8
`long double`	10	12

All sizes are in bytes.

Note that variables of type **`char`** are guaranteed to always be one byte.

There is no maximum size for a type, but the following relationships must hold:

```
sizeof(short) <= sizeof(int) <= sizeof(long)
sizeof(float) <= sizeof(double) <= sizeof(long double)
```

Numeric literals

- Suffixes allow you to specify a number is of a given type:
  - **`30000`** is of type **`int`**, but **`30000L`** is of type **`long`**
  - **`2.5`** is of type **`double`**, but **`2.5f`** is of type **`float`**
- Can also give numbers in different bases:
  - Precede with a zero for octal: **`017 == 15`**
  - Precede with **`0x`** for hexadecimal: **`0x1f == 31`**

Enumerated types

- Can use these to represent things that only take on certain values
- Values equal to 0, 1, 2... based on order of definition
- Values can be set by programmer to things other than 0, 1, 2...
- Based on integers, but don't mix enums with integers unless absolutely necessary

```
#include <stdio.h>
int main() {
  enum Suit {
    SPADES, HEARTS,
    DIAMONDS = 42, CLUBS
  };
  enum Suit suit1, suit2;

  suit1 = SPADES;
  suit2 = CLUBS;

  if (suit1 < suit2)
    printf("Spades are first.\n");
  else
    printf("Clubs are first.\n");
  printf("Spades = %d, Clubs = %d\n",
          suit1, suit2);
  return 0;
}
```

Variable declaration, initialization

- Declaring a variable provides space for it in memory
  - "**`int x;`**" will cause 4 bytes on the stack to be reserved for x
- No initialization takes place! And unlike Java nothing will prevent you from using that random value.
- Consider this code:

```
int x;
printf("X: %d\n", x);
```

- It might print **`X: 0`**
- It might also print **`X: -2349235`**, or **`X: 82373`**
- It's up to you to initialize your variables properly
- Initialization can be done at the same time as variable declaration:

```
int x = 42;
```

Identifier scope

- C has two main types of scope
  - Block scope: a variable declared inside a block is visible only within the block (includes nested blocks inside that block)
  - File scope: an identifier declared outside of any block is visible everywhere in the file **after** the declaration
    - applies both to global variables and function names
    - **Example:** global.c

Identifier linkage

- What happens if we encounter two instances of the same identifier across different files?
- A function named **`foo()`** in **`file1.c`** can cause problems if there's a function named **`foo()`** in **`file2.c`**
- We can resolve these types of conflicts by changing the linkage of the functions
- Linkage: a property of an identifier that determines if multiple declarations of that identifier refer to the same object
- **Example:** linkageExample folder

Identifier linkage, cont.

- Three types of linkage
  - none: all declarations of an identifier refer to different entities (i.e., one copy per declaration)
  - internal: all declarations of an identifier inside a given file refer to the same entity, but declarations across files refer to different entities (i.e., one copy per file)
  - external: all declarations of an identifier refer to the same entity (i.e., one copy per program)
- Default linkage is different for different types of identifiers
  - all functions and file scope variables default to external linkage
  - variables with block scope default to no linkage
- Use **`extern`** and **`static`** to modify linkage

Storage

- How long is memory allocated for an object?
- After this function returns, is there any guarantee that the value 5 will stay at the spot in memory at which it was stored?

```
int example(int i) {
   int j = 5;
   return i + j;
}
```

- There are two types of storage: static and automatic; the variable `j` above has automatic storage, meaning it is no longer maintained after its function returns
- **Example:** static.c

Storage, cont.

- Static storage means that the variable exists throughout the entire life of the program
  - global variables have this kind of storage
- Automatic storage is the default for block-scoped variables, but this can be changed with the **`static`** keyword
- Initializations to static variables occur only once