Basic C Programming

CS 350: Computer Organization & Assembler Language Programming
Lab 0 (Not for turning in)

A. Why?

- You'll be writing your programs for CS 350 in C and you'll definitely be writing programs for CS 351 in C.
- One of our later topics will be seeing how high-level programs in C are implemented as lower-level programs in machine code (the instructions that the hardware understands).

B. Outcomes

After this lab, you should be able to:

- Log into the `alpha.cs.iit.edu` machine and compile and run a simple C program.

C. Discussion

- C is a "lower-level" language than Java: its constructs more easily map to the data and operations found on typical hardware.
- You should have accounts on `alpha.cs.iit.edu`; if not, let me know (Piazza would be useful here).
- As part of the zip file that makes up this lab, you should find `Lab00_sample.c`. Open it with a text editor and read through it. You'll find much of C is similar to Java, but there are some fairly large differences too.
- `#include <stdio.h>` tells the C compiler to read in the "header" file for the standard I/O library. (Program files generally have the extension `.c`. Header files contain prototypes and type definitions and have the extension `.h`.
- `double sqrt(double);` is the prototype for the square root function. In C, functions have to be defined (with their body) or declared (with just their prototype) before they can be used.

- `int main() { … return 0; }` is the declaration of the main program, which is what will be run when you execute the compiled version of this program. The integer returned is an error code passed back to the operating system: `0` means no error.
- `char string1[6] = {'h', 'e', 'l', 'l', 'o', '\0'};` is a declaration with initialization for an array of characters called `string1`. The `'\0'` character has all zeros as its bits; it's used to delimit (announce the end of) the string. A string of length *n* requires $n+1$ characters when stored.
- `char string2[6] = "hello";` uses a different syntax for the character array initializer; it's much more convenient than the notation used for `string1` but gives `string2` the same string value.
- `char empty1[1] = {'\0'};` defines `empty1` to be an *empty string* (it has length zero). When we print `empty1` later, we'll print nothing.
- The three print statements

  ```
  printf("%s\n", string1);
  printf("%s\n", string2);
  printf("empty1 ->%s<-empty1\n", empty1);
  ```

  cause the three strings `string1`, `string2`, and `empty1` to be printed. The first argument to `printf` ("print, formatted") is a format string; the value(s) following the format string are used to fill in any format specifications in the format string. Format specifications can be recognized because they start with `%`. In these cases, we use `%s` to indicate that a string should be printed.
- `char ch = '!';` is a declaration with initialization for a variable `ch` of type character.
- `char *string3 = "This is a string";` is a declaration with initialization for a third string variable. The star (asterisk) means that `string3` is of type *pointer-to-*`char`: It stands for the memory address of a character (namely, the initial character in a sequence of characters). [If you're thinking that this sounds similar to an array of characters, you're right; we'll see this later in the semester.]

- `char *empty2 = "";` is a declaration with initialization for a string variable that has the empty string as its value. There are some technical differences between this and `empty1` above but they both behave like empty strings.
- The two print statements

  ```
  printf("%s\n", string3);
  printf("empty2 ->%s<-empty2\n", empty1);
  ```

  use printf with `%s` to print `string3` and `empty2` as strings.
- The lines

  ```
  int x = 17;
  double d = sqrt(x);
  printf("The square root of %d = %f\n", x, d);
  ```

  declare, initialize, and print the values of an integer variable `x` and a double-precision floating-point variable `d`. The `sqrt` routine is part of the standard math library. The `%f` format is used for floating point and double-precision values.
- The lines

  ```
  int y;
  printf("Enter an integer >= 0 (and then return): ");
  scanf("%d", &y);
  ```

  Declare, prompt for, and read in a value for an integer variable `y`. The `scanf` statement ("scan formatted") reads from the standard input (the keyboard). The format `%d` indicates we expect the user to enter an integer (followed by a carriage return). White space (including carriage returns) before the integer is ignored. If there are nonblank characters after the integer and before the next carriage return, then those characters will be read in during the next `scanf` (if there is one). The ampersand in front of `y` says we pass `scanf` the memory address (i.e., location) that `y` is stored at; this way `scanf` can actually change the value of `y`. (C uses call-by-value, like Java, so just passing an integer variable doesn't give us a way to change the value of the variable.)
- The lines

  ```
  double sqrt_y = sqrt(y);
  printf("The square root of %05d = %.3f = %10.4f\n",
  y, sqrt_y, sqrt_y);
  ```

declare and initialize `sqrt_y` and then print out `y` and `sqrt_y` (twice). Note the format strings `%05d` and `%10.4f` -- the `5` and `10` indicate the *field width* (we want to use 5 and 10 positions respectively when printing out the values). The `0` in `%05d` says we should print out leading zeros. The `.4` in `%10.4f` says we want exactly 4 digits after the decimal.

- The next collection of lines shows what might happen if we read in a value that's too large for the type of variable we're reading into.
  - `printf("Now enter an integer >= 2147483648: ");` prompts for a value too large to be represented as a 32-bit integer (the size we're likely to have with our current hardware).
  - `scanf("%d", &y);` tries to read that too-large integer into `y`. (Since the value is too big, `y` will have the wrong value.)
  - `printf("The square root of %d = %f\n", y, sqrt(y));` calculates and prints out the square root of `y`
    - The actual bits stored in `y` won't stand for the value that was typed in, so we'll get some sort of wrong result.
    - The most likely wrong result happens if what got stored into `y` looks like a negative number: The `sqrt` routine will return a special value, *Not-A-Number*, which gets printed out as `nan`.
    - `if (y < 0) { printf("\"nan\" …" }` checks for this possibility and prints out an explanatory message.
- The remaining lines asks us to repeat the operation on the same value, but this time we read in the value as a `long` (probably 64-bit) integer.
  - `long int z;` declares `z` to be a variable of type long integer.
  - `printf("Try the same value (we'll read it as a long integer): ");` is a prompt.
  - `scanf("%ld", &z);` reads a long integer value into `z`. Note the format code `%ld`, which means long integer (compare to `%d` for regular integers).
  - `printf("The square root of %ld = %.10f\n", z, sqrt(z));` prints out `z` and its square root (to 10 decimal places for the square root). Note again the `%ld` format for `z`.

D. Logging Into alpha and Compiling

- The `alpha` machine runs Linux; if you don't already know how to use Linux, it'll be good for you to do so. The `linux-account.pdf` file that's part of this lab will show you the basics of linux; the version attached refers to the old computer; substitute `alpha.cs.iit.edu` everywhere you see `dijkstra.cs.iit.edu`.
- Your Lab TAs will be able to show you how to log into `alpha.cs.iit.edu` from the lab machines.
- They'll also discuss possible ways to do this from your own personal machines. Basically, you'll need a way to run a secure shell session (ssh) in some sort of command-level terminal environment.
- For this lab, practice logging into the `alpha` machine and compiling and running the `Lab00_sample.c` program. Once you have a copy of the program in your current directory, the Linux command to compile the program is

  ```
  gcc Lab00_sample.c -lm
  ```

  "`gcc`" means "GNU [pronounced Guh-Noo] C compiler," the standard compiler for Linux environments[1]. The `-lm` says to include the math library (so you can use `sqrt`). Depending on your setup, you may not need the `-lm`; if you get a complaint about a missing `sqrt` routine when you compile your program, then you need the `-lm`. It may also be possible to put the -lm before the *filename*`.c`: `gcc -lm Lab00_sample.c`
- If the compile succeeds, it produces an executable file named `a.out`. To run your program, execute that file with the command `./a.out`

[1] "GNU" stands for "GNU's Not Unix", a reference to GNU being different from the versions of Unix that existed when the GNU project was started