Week 3 Lab

Beginning C

Today, you'll be doing some introductory exercises on C. You'll also be learning some new `bash` techniques that will help you manage your system and develop and test your programs.

The programming in this lab won't require much problem-solving ability, but unless you're a fairly experienced C developer you will have a lot of reading to do – rather than tell you which functions to use, we will usually expect you to discover them yourself. The C standard library is very large, far too large to cover in detail in one semester, so in order to become a good C programmer you'll need to learn how to find information on what's in it. It's a good idea to share this work with a classmate.

Note: this lab will not require you to have root access to your system unless you wish to install new software.

Task 1. Compiling C programs: warnings and errors

Using the text editor of your choice[1], type in the "Hello, World" program that was presented in lectures and save it as `hello.c`. Compile this program in the terminal, naming the output `hello`, and run it.

Note that by default, `bash` does not look for executable files in the current directory. This is a security precaution: imagine what could happen if an unscrupulous person were to put a file into your home directory called `cd` or `ls`. In order to run `hello`, you'll need to tell `bash` to look in the current directory for it: `./hello`

Here's the source code:

```
#include <stdio.h>

int main()
{
        printf("Hello,␣world!\n");
        return 0;
}
```

A list of helpful `gcc` command line parameters is appended to this lab sheet. Use them to help you find answers to the following questions.

Exercises

a) Research: in `gcc`, what is the difference between an *error* and a *warning*?

b) Find the command line parameter that causes `gcc` to display all warnings. Recompile – if you've done this correctly, you shouldn't get any warnings.

c) Delete the `#include<stdio.h>` directive. Recompile. Does the code still compile? Do you get any warnings?

d) Replace the `#include<stdio.h>` directive. Delete or comment out the line `return 0;` and recompile. Do you get any errors or warnings? Does it make a difference whether all warnings are turned on?

e) Without reinstating the `return` statement, edit the declaration of `main()` so that it returns `void` rather than `int`. Recompile. Do you get any warnings? Do you get any errors?

[1] Linux supports many different text editors. If you don't like the default, and have sufficient privilege on the computer you're using, feel free to explore the package manager to find and install one that you like better.

f) Leave `main()` declared as returning `void` and replace the `return 0;` line in its original position. Recompile. Does this generate any warnings or errors?

Task 2. I/O redirection

I/O redirection is a capability built into Unix-style command shells such as `bash`. When you run a program from the command line, part of the data that `bash` passes into `crt0` includes three I/O streams: an *input stream* that becomes `stdin` and two *output streams* that become `stdout` and `stderr`. As far as C programs are concerned, there's no difference between a console I/O stream and a file I/O stream – they're all just instances of `FILE*`.

Command line I/O redirection tells your shell that, instead of passing in a pointer to the keyboard buffer or shell output stream, it should replace one or more of `stdin`, `stdout`, or `stderr` with a file that has been freshly opened for reading or writing. Because this does not alter the type of the stream that is passed into your program, your program will not be able to tell the difference. This means that any Unix command-line program that uses console-based input and output can also work on files, without needing to touch its source code or recompile.

The command line redirection operators available to you in `bash` are:

- `program > filename` – redirect standard output to a file. This will *overwrite* the given file if it already exists. Example: `ls -l > outfile.txt`.
- `program >> filename` – redirect standard output to a file in append mode. This will *append to* the given file if it already exists. Example: `ls -l >> outfile.txt`.
- `program < filename` – redirect input from a file. This will give an error if the file doesn't exist. Example: `cat < infile.txt`.
- `program 2> filename` – redirect `stderr` to a file. This will *overwrite* the given file if it already exists. Example: `buggy_program 2> error.log`).
- `program 2>> filename` – redirect `stderr` to a file. This will *append to* the given file if it already exists. Example: `buggy_program 2>> error.log`).
- `program1 | program2` – directs `program1`'s standard output to `program2`'s standard input. This has the effect of chaining the programs together. This operator is called a *pipe*, and you can use multiple pipes on one line.

In this Task, we'll use some standard Linux commands to practise I/O redirection. This is a powerful technique that will help you test your C programs. We'll also be seeing it later, when we look at `bash` scripting.

Exercises

a) Skim the manual pages for the programs `cat`, `less`, and `rev` to get an idea of what these programs can do.

b) List all the files in `/etc` and store the result into a file called `etc-list.txt` in your home directory.

c) Display the file `etc-list.txt` in reverse, a page at a time.

d) See if you can list all the files in `/etc` in reverse, a page at a time, without using an intermediate file (i.e. by typing a single command line).

e) Make `ls` produce an error message by trying to list a nonexistent file. Redirect that error message to a file called `ls-err.txt`. You'll know it's worked if the error message appears in the file but not on the screen.

f) Repeat the previous exercise, but this time use the operator that appends to `ls-err.txt`. Verify that the error message appears more than once in the file.

Task 3. Console I/O with `stdin` and `stdout`

Write a C program called `char-counter`. This program must:

- read characters one at a time from `stdin` (*standard input*, i.e. the console)
- count how many characters are seen
- stop counting when end of file is reached
- display the number

Hints:

- You will need to find a C function that can read a character from `stdin`, which is an *input stream*. You will then need to read the manual page for this function so that you can find out which header you'll need to include, and what the function will return when it reaches end of file.
- In order to display the number of characters you've typed, you'll need to use the `printf()` function with a format character that tells it to output a decimal integer. A brief `printf()` cheat sheet is attached to this lab sheet.
- You can test this program by running it and then typing on your keyboard. Note that if you choose to test this way, you'll need to press Ctrl-D to send the end of file marker to the keyboard buffer. You can also test it using command line redirection (see Task 3).

Compile and run your program – test it to make sure that it works.

Exercises

a) The Ctrl-D key combination sends the end of file marker to your keyboard buffer. What happens if you press Ctrl-D at a `bash` prompt? Try it and see.

b) Modify your program so that it doesn't count spaces and newlines. Verify that your modifications have worked.

Task 4. FizzBuzz: iteration, conditionals, preprocessor directives

In this task, you must develop an algorithm to solve the `fizzbuzz` problem. Your algorithm must do the following:

- count from 0 up to 100
- if the counter is a multiple of 3 (but not 5), print "Fizz"
- if the counter is a multiple of 5 (but not 3), print "Buzz"
- if the counter is a multiple of both 3 and 5, print "FizzBuzz"
- if the counter is not a multiple of 3 or 5, print the value of the counter

This is a fairly common style of problem in interviews for programming positions, but anecdotal evidence suggests that many candidates can't get it to work[2].

Exercises

a) Write a C implementation of your `fizzbuzz` algorithm. Compile and test it to verify that it works.

b) Replace the literal strings "Fizz" and "Buzz" with *preprocessor macros*, so that their values can be set at compile time. Use `gcc` command line options to set these strings to "Ping" and "Pong" at compile time, and test the resulting executable to verify that it works.

[2](see `http://blog.codinghorror.com/why-cant-programmers-program/`).

Appendix I: `printf` format specifiers

Usage example:

```
printf("%d %c %s\n", 24, 'A', "aString");
```

Format Specifier	Meaning
%c	Single character
%d or %i	Signed decimal integer
%e %E	Floating-point number, scientific format (e or E)
%f	Floating-point number, decimal notation
%d	Double precision floating-point number, decimal notation
%g %G	Causes %f or %e/%E to be used, whichever is shorter
%o	Unsigned octal integer
%p	Pointer
%s	Character string
%u	Unsigned decimal integer
%x %X	Unsigned hex integer using a-f/A-F
%%	Print the percent sign

Appendix II: `gcc` options

Option	Meaning
-c	compile source but don't link
-S	stop after compilation, do not assemble
-E	run the pre-processor only
-g	include debugging information
-O*level*	optimise the code to *level* (e.g. -O3)
-W*warn*	turn on/off particular warnings (-Wall is good)
-I*dir*	specify a directory to look for include files
-L*dir*	specify a directory to look for library files
-D*macro[=defn]*	define a macro (#define *macro defn)*
-U*macro*	undefine a macro
-o *outfile*	place output in *outfile*