SIENA*college*
Computer Science

Computer Science 330
Operating Systems
Siena College
Fall 2020

Lab 2: C
Due: 11:00 AM, Monday, August 31, 2020

In this lab, we will begin to develop your skills as a C programmer.

You must work individually on this lab.

Learning goals:

1. To learn how to compile and run programs at the Unix command line.
2. To learn about the search path in Unix shells.
3. To learn the basic structure of a C program, including console input/output, and apply that understanding to implement some simple programs in C.

Getting Set Up

You will receive an email with the link to follow to set up your GitHub repository, which will be named `c-lab-yourgitname`, for this lab.

The C Programming Language

C is a widely-used, general purpose language, well-suited to low-level systems programming and scientific computation. Few languages have maintained popularity for as long as C has.

We will initially study it assuming you have Java experience, focusing on the features that make C significantly different from Java. Fortunately, Java borrowed much of its syntax from C, so it is not difficult for a Java programmer to read most C programs.

C++ is a superset of C (that is, any valid C program is also a valid C++ program, just one that doesn't take advantage of the additional features of C++). C++ adds object-oriented feautures. In this course, we will look only at C, not C++.

We saw in the first lab how to compile and run a "Hello, World" program. You used the `gcc` command to produce an executable file `a.out`. Your repository for this lab includes a similar program, `hello.c`. Even in this simple program, there are several things worth noting as a beginning C programmer.

The command

```
gcc hello.c
```

is essentially just another program that can can run at the command prompt. We run a program named `gcc`, which is a free C compiler, part of the GNU Compiler Collection.

This example uses the `gcc` command in its simplest form, where it is used to compile a complete C program that is contained in a single file. In this case, we're asking `gcc` to compile a C program (the *source code*) found in the file `hello.c`. Since we didn't specify what to call the *executable* program produced, `gcc` produces a file `a.out`. The name is `a.out` for historical reasons, and stands for "assembler output".

This is analogous to a Java program consisting of one class (let's say it's the `public class Hello` in `Hello.java`) that has nothing but a `main` method. There is an important difference, however. In Java, when you compile, either by pressing a button in your IDE or at the command line with

```
javac Hello.java
```

the file produced is `Hello.class`, which needs to be run inside a Java Virtual Machine (JVM):

```
java Hello
```

It cannot run directly on the computer's hardware. The program `java`, the implementation of the JVM, runs (more) directly on the hardware, but that program runs the Java program on our behalf.

Executables and Search Paths

But... when we compile the `hello.c` program, the `a.out` file produced is an actual executable program that runs on the hardware.

To understand how we run the program and why it's done that way, we need to understand how Unix shells run any program. Basically, to run a program we type its name. But the names it recognizes are only those programs that exist in a set of directories on the system called the *search path*.

The search path is simply a list of directory names, which are searched in the order they're specified for an executable program with the name that was typed at the shell prompt.

The search path is specified using an *environment variable*. Environment variables are used in Unix to provide information to a variety of programs. We can see the set of environment variables assigned to our shell with the `env` command. Run the command and redirect its output to a file `env.out`.

Output Capture:
`env.out` for 1 point(s)

In the file `env.out`, find the line that specifies the `PATH` environment variable. This is the list of directories where your shell will look for programs when you type a name at the prompt.

Using `ls`, look at the contents of some of the directories in your path. Can you find some of the commands you learned in the earlier lab?

So, if we want to figure out which actual executable file will run when we type a name, we can (as the shell would do), search each directory in our search path. The first one we encounter is the one that will execute. That's a lot of work. If we want to know which program will execute if we issue a particular command, we can use the `which` command to find out.

? Question 1:
Which executable file is run when you issue a `gcc` command on noreaster? (2 points)

So when we run one of our own programs, such as the `a.out` we generated from `hello.c`, we type its name. But if you do that on noreaster, you will likely get an error message, even those `a.out` is in your working directory:

```
[jcool@noreaster ~]$ a.out
bash: a.out: command not found
```

The problem is that your working directory is not part of your search path! That's why when we ran the program above, we ran it with a slightly different command:

```
[jcool@noreaster hello]$ ./a.out
Hello, C World!
```

The "`./`" before the name tells our shell that we want to run the program in "`.`", which is the Unix shortcut for specifying our home directory. We could just as well give an entire absolute path to our program:

```
[jcool@noreaster hello]$ /home/jcool/opsys/c-lab-jcool/a.out
Hello, C World!
```

We could have programs in our current directory execute without the "`./`" or absolute path, but having "`.`" in a search path is generally considered a bad idea.

We'll be writing lots of C programs, and we probably don't want all of our executables to be named `a.out`. We could certainly rename the ones we want to keep using the `mv` command. But let's just have `gcc` produce an executable with the name we want right way:

```
gcc -o hello hello.c
```

Here, the executable file produced is called `hello` because the `-o` command-line parameter is specified, which tells `gcc` that the next command-line parameter following the `-o` should be used as the output file name.

Details of our Simple Program

Finally, we examine the source code for our `hello.c` program.

As Java programmers learning about C, you will notice a lot of familiar syntax, but also some striking differences. The similarities are not surprising, as Java borrowed much of its syntax from C. The differences are also not surprising, as C has been around for a few decades longer than Java, which allows Java to include more modern features.

At the top of the file, we have a big comment (the equivalent of the class comment in Java) describing what the program does, who wrote it, and when. Your programs should have something similar in each C file.

As with Java, we need to tell C if there are libraries or other code that we will be using within this file. In Java, this is done with `import` statements, but nothing needs to be imported to use parts of some of Java's core API that fall under the `java.lang` package, like `System` and `Math`. In C, we need to inform the compiler for even things like basic input/output. In this case, our program uses a C library function called `printf` to print a message to the screen. For C library functions, the needed information is provided in *header files*, which usually end in `.h`. In this case, we need to include `stdio.h`. How do we know? Well, in this case, it's a header file included by nearly every C program, so you'll just get to know it. But in general, we can check the Unix manual with "`man 3 printf`" and see which header files are listed. We'll learn more about using the Unix manual to find out about C library functions and think more about the actual mechanism employed here later this semester.

Every C program starts its execution by calling the function `main`. The line

```
int main(int argc, char *argv[])
```

is the *function header* for `main`. It corresponds very nicely to the typical `main` method header in a Java application

```
  public static void main(String args[])
```

and plays the same role. The keyword `public` is not needed in C, as it has no notion of data protection like Java or C++. The `static` is not needed because all functions in C are essentially like `static` methods: they have a global scope and anyone can call them. C's `main` has an `int` return instead of `void`, since C uses the return value of the `main` function as a return code that the whole program provides to the operating system. The two command-line parameters are provided to `main`, traditionally declared as `argc`, the number of command-line parameters (including the name of the program itself), and `argv`, an array of pointers to character strings, each of which represents one of the command-line parameters. In this case, we don't use them, but they are often listed anyway (though they may be omitted if not used). These provide the same information as Java's array of `Strings`. As we will see soon, C arrays do not come equipped with a length attribute, so `argc` is needed to tell how many entries exist in the array `argv`, and string data is represented by a pointer to an array of `char`, hence the `char *`.

`printf` plays the role of Java's `System.out.print` and results in the string passed as a parameter to be printed to the screen. The `\n` results in a new line. We will see soon that the mechanism for constructing strings to print is quite different from that in Java.

A value of 0 returned from `main` generally indicates a successful execution, while a non-zero return indicates an error condition. So we return a 0. Many C compilers will also allow `main` to have a return type of `void` and no `return` statement, but the `int` return type is normally used.

Among the familiar for Java programmers learning C: `;`-terminated statements and code blocks enclosed in `{}` pairs, most of the arithmetic, boolean, and logical operators, and the names and syntax of control structures (loops and conditionals), and more.

The biggest difference that is evident in this simple program is that there are no classes and methods, just *functions*, which can be called at any time. Any information a function needs to do its job must be provided by its parameters or exist in *global variables* – variable declared outside of every function and which are accessible from all functions.

The C `for` loop is much like Java's `for` loop, except that the loop index variable needs to be declared before the loop. That is, a Java loop that looks like this:

```
for (int i=0; i<10; i++) {
  ...
}
```

would need to have the declaration of `i` outside of the loop:

```
int i;

// any other code that happens before the loop

for (i=0; i<10; i++) {
  ...
}
```

Practice Program:
Write your own C program named `helloloop.c`, much like the "Hello, World" example, but which prints some other message and prints it 10 times inside of a `for` loop. (5 points)

More C Basics

Question 2:
Consider any C program that uses the `printf` function. What happens if you leave out the `#include <stdio.h>` line? Explain briefly. (2 points)

There are many C programming references and tutorials online and you are welcome to look at them. We will refer to some pages on `http://www.cprogramming.com/` and elsewhere to help get you up to speed on some C topics.

The `printf` Function

C's `printf` function is the primary mechanism for printing to the standard output (terminal). While you are most likely familiar with Java's `print` and `println` methods, it also contains a `printf` method that is very similar to C's. Check out Wikipedia's printf article for some information about C's `printf`.

Practice Program:
Write a C program `temps.c` that prints a nicely-formatted table of Fahrenheit to Celsius temperature conversions. See the required output format below. (10 points)

```
-100F = -73.333C
 -99F = -72.778C
 -98F = -72.222C
 ...
 -10F = -23.333C
  -9F = -22.778C
  -8F = -22.222C
 ...
  -1F = -18.333C
   0F = -17.778C
   1F = -17.222C
 ...
  31F =  -0.556C
  32F =   0.000C
  33F =   0.556C
 ...
 998F = 536.667C
 999F = 537.222C
1000F = 537.778C
```

Command-line Parameters

You have likely seen Java applications that take command-line parameters (the `String args[]` parameter to the `main` method of a class). A C program that wishes to make use of command-line parameters must declare two parameters to the `main` function, traditionally named `argc` and `argv`.

The parameter `argc` to the `main` function is a count of how many command-line strings are included in `argv`, which is an array of strings.

These are demonstrated in the `printargs.c` program included in your repository.

Note: argv[0] is not the first parameter, it is the program name itself, and this array entry for the program name is included in the value of argc.

Even when we enter numbers for command-line parameters, the operating system will provide them to your program as strings. So we need to be able to convert strings to a numeric equivalent.

This is demonstrated in the repeat.c program.

Note that the string to integer conversion uses the the overly complicated strtol function, which we use, then check error conditions. There's a lot here we have not yet seen.

- The man page for strtol tells us we need to include two additional header files, stdlib.h and limits.h.
- It also tells us about the parameters to strtol, which are the string which we would like to convert to a number, a pointer into the string at the point beyond which we matched a number (which we don't care about, so we pass in NULL), and the base to use for the conversion. We also see that the number is the return value.
- Error checking for strtol is messy – we need to check the variable errno, defined in errno.h, to see if an error condition was encountered. If so, errno will be a non-zero value and we print an error message and exit.
- We use fprintf instead of printf when printing the error message. This is because we want to give this output special significance. Rather than sending it to the *standard output*, which is what printf would do, we send it to *standard error*, by using fprintf and specifying stderr as the first parameter. Java supports the same idea: use System.err instead of System.out.
- Other than that, it works just like printf. We give it a format string. In this case, it includes one specifier, a %s, which means to expect an additional parameter which is a character string (well, really a pointer to a null-terminated array of char). Here, the string is argv[0], the first command-line parameter, which is always the name of the program. This labels the error message with the program name.
- Once we have detected the error, we don't want to continue, so we call the exit function with an error code of 1 to terminate execution. We could also use the call return 1;.
- Note that the error check here has two %s's, so we have two additional parameters to fprintf, both pointers to strings.

Practice Program:
Write a C program that takes an arbitrary number of command-line parameters, each of which should represent an integer value. Print out the sum of the values provided. Call your C program argadder.c (10 points)

Formatted Keyboard Input

We have seen how to use the `getchar` function to get input from the keyboard or redirected from a file, one character at a time. But often, we'd like to read input as words or numbers.

C's standard mechanism for this is the `scanf` function, as shown in the `scanf-example.c` program.

- `scanf` is a very strange thing. It will make a bit more sense once you have more experience with the `printf` function, but for now we can summarize what we see there as "read in an integer value (represented by the `%d` in the *format string*), and put it into the place pointed at by the address of `x`, then return the number of values that matched the input with the correct format." Similarly for the `double` value using a `%lf` in the format string.
- The `scanf` call forces us to think a bit about *pointers*, which are the key to understanding so much of how C works. `scanf`'s parameters after the format string are always a list of pointers to a place in memory where there is room to put the values being read in. In this case, we want the `int` value to end up in the local variable `x`, so we have to take the address of the variable with the `&` operator. Don't worry, it will make better sense when you see more examples.
- The `scanf` to read in a string is a special case and does not require the `&` operator. This is because the name of a C string already is a pointer to the first element in the array. Again, much more on this when we study C pointers in more detail.
- Next, we check to make sure that the input to `scanf` did, in fact, represent an `int` value. If not, we print an error message and exit. Otherwise, we continue.

Practice Program:
Write a program `inputadder.c` that takes its input from the keyboard rather than from the command line. Your program should read in integer values one by one, accumulating a sum as you go, until you encounter an invalid (non-integer) or the end of the input (someone types `Ctrl-d`). At that point, print out the sum and exit. (10 points)

Wrapup

Question 3:
Give a one-sentence definition of each of the following terms below, which you encountered in this lab. (10 points)

- source code
- executable
- search path
- environment variable

- header file
- function header
- global variables
- standard output
- standard error
- format string

Submission

Commit and push!

Grading

This assignment will be graded out of 50 points.

Feature	Value	Score
`env.out`	1	
Question 1	2	
`helloloop.c`	5	
Question 2	2	
`temps.c`	10	
`argadder.c`	10	
`inputadder.c`	10	
Question 3	10	
Total	50	