Lab 1.1

Out: February 2, 2005

What you'll learn.

In this lab, you will write and compile your first C program, all from scratch. We'll go through step by step what is required to write a complete program in C, and then you will write a few small programs that do different things:

- print out the string "Hello, World!"
- read in a string from the terminal, and then print it back out
- sort characters in a string and print out their memory addresses

How you'll do it.

In the first task, we'll walk you through how to write a complete C program that prints out "Hello, World!". This will require you to create a ".c" file. You may want to take what you learn in the first task and make a stencil .c file that you can use for all of your C programs. Also, please note that your C program files must end in .c, and that it must be a lowercase "c". In the first task we will also go through how and why to create header files, which end in ".h".

Task 1: How to write, compile, execute, and `printf`

Program Files

A C program requires only one thing to compile: a function called `main`. If `main` is missing, you would get an error saying something about an "undefined reference to `main` in function `_start`." This error alone gives you a bit of insight into what is happening in `gcc`, the C compiler. More on `gcc` later.

The `main` function in a C program must have a return type of `int`. This is because the `_start` symbol mentioned before expects `main` to have a certain signature. Since `main` has a return value, your program can return an error code. Most of the time, the error codes aren't checked, so in this class you can just return 0 from `main`.

Any functions called in `main` must be declared above `main` in the file, or included from a header file. Within `main`, as within any function, any locally declared variables that will be used in the function should be declared at the beginning of the function (or at the top of any block of code).

Actually, defining functions is a little more sophisticated than that. The C language makes a distinction between the declaration and definition of a function. It is legal to declare a function without defining it (filling in the body of the function) at the same time. This distinction does not exist in Java. Here is an example of how one might use a function declaration:

```
#include <stdio.h>

void foo(int a, int b);

int main() {
  foo(1, 2);
  return 0;
}

void foo(int a, int b) {
  printf("%d\n", a + b);
}
```

In the above program, the function `foo(int, int)` is declared above `main()` and defined below it. Without the first declaration (sometimes called a prototype), the compiler would complain in `main()` that `foo(int, int)` was being used without first having been declared.

One might ask why we didn't simply define the `foo` function above `main` and avoid the problem altogether. In this case, we could have easily done that. However, there are other cases where a function prototype is necessary. For example, the following example requires a function prototype:

```
void foo();

void bar() {
  foo();
}

void foo() {
  bar();
}
```

The function prototype for `foo` cannot be avoided because `foo` and `bar` have a cyclic dependency. Although this example is clearly contrived, cyclic dependencies will arise frequently in larger C programs. Therefore, as a general practice, you should declare function prototypes for every function you write. In the next section you will see that there is a nice place to do this, called a header file.

Header Files

For your first program, we want you to print the string "Hello, World!" to the command line using the function `printf`. In C, printing and reading functions are declared in the C header file `stdio.h` (for "standard input/output"), so you'll need to include this like so:

```
#include <stdio.h>
```

`stdio` is a standard library defined by the C language[1]. This line tells the C compiler (actually the pre-processor) to look for the file called `stdio.h` and then include all of the declarations it finds there. In fact, the entire contents of the file `stdio.h` will be copied and pasted directly into your C program. This automatic copy-and-paste will be extremely useful as our programs grow through the semester: not everything will have to be in the same file. Any files you want to include in a C program should always go at the very top of the file.

Using `printf`

You'll print the string "Hello, World!" with the function `printf`, as found in `stdio.h`. You can refer to the man pages for `printf`[2] for a detailed explanation. However, this is a complicated function, so we'll show you a few of the most common arguments here.

`printf` takes a string (the "format string") as its first argument, followed by some number of variables. Wherever it encounters a % in the format string, it converts the next argument to string form and prints it out. `%d` expects an integer, `%c` a character, `%s` a string, and `%p` a pointer. In addition, there are some sequences, starting with a `'\'`, to print special characters—`'\n'` prints a newline and `'\a'` makes a beep, for example. So the line

```
printf("%s is %d inches tall\n", name, height);
```

will insert a regular, null-terminated string called "name" and an integer called "height" into that message and print it to the terminal.

Compiling

After you've written your program, the next step is to compile it. We'll be using `gcc`, the GNU project C and C++ compiler. `gcc` can be run from the command line. You can check

[1]To see all the I/O methods that `stdio.h` provides, a good place to refer to is:
http://www.cplusplus.com/ref/
This website has concise explanations and examples of the most common C libraries, including most of the ones you'll be using in this class.

[2]manual page. If you type `man foo` in a shell, you'll get an explanation of foo or how to use foo. There are man pages for programs, system calls, library functions like `printf`, and more. You can even try `man man` to learn more about man pages. You may notice that man pages are divided into sections. Thus, for example, `man 3 printf` describes the C function, while `man printf` tells you about some shell utility. To find out which sections have content about `printf` you can try `man -k printf`.

out the man page of `gcc` to learn more about the many possible arguments that `gcc` can take. However, we'll tell you right now how we recommend you use it:

```
% gcc -Wall -g -o myExecutable myProgram.c
```

where myProgram.c is the name of your C program file and myExecutable is the name that you want to give to the executable `gcc` will produce for you (you get to pick whatever name you want). When you run the executable, the program you wrote will go do stuff.

When choosing a name for your executable, be careful not to name it something that is already taken, like `man` or `acroread`. Those examples may seem silly, but beware of the fact that `test`, for example, is already taken. One option is to not include the `-o myExecutable` part of the `gcc` command. If you don't, your executable will be created and named `a.out` by default.

The `-Wall` option turns on compiler warnings. C is a very permissive language, so many errors you make will still technically compile. `gcc` will only warn you if you ask it to, so `-Wall` is highly recommended. The `-g` option allows `gdb`, the debugger, to understand your code, which will come in handy later.

Once you've compiled, you can run your program by simply typing the name of your executable into your command line, presuming you are still in the same directory as it. If you would like to see the return value of your `main` function after you've run your executable, type `echo $status`[3] in the command line.

> **Task:** Write, compile, and execute a C program that prints the string "Hello, World!" to the terminal using the print method `printf`. For this first program, you don't need to write your own header file.

Task 2: Reading from input

Now that you know how to print a string that you've hard-coded into your program, let's move on to writing a program that can print *any* string, given a maximum size. In C, a string is actually an array of characters ending in the null character, represented as `'\0'`. When you initialize a character array using `char myString[] = "whatever"`, a null character is automatically added on the end.

For this task we want you to read in up to 100 characters from the command line, using `getchar`. Remember, your character array should end in the null character when you are done getting the string. Furthermore, your program should be able to handle the possibility that more than 100 characters will be entered. You should ignore any excess characters, not crash or otherwise have an error.

[3] Note that this command depends on which kind of shell you are using. The default CS account shell is called `tcsh`, and the remarks in this lab are all true for that shell.

When you type characters into the terminal, they are not instantly sent to your program. Rather, they are put in a buffer. Only when you hit the enter key, i.e. type in a newline character, is the buffer sent to your program. Thus, you must keep both the newline character and your size limit in mind when using reading functions like `getchar`. For details, `man getchar` or check the website reference.

> **Task:** Write, compile, and execute a C program that reads in up to 100 characters, using `getchar`, from the terminal and prints them out to the terminal again without crashing or reporting an error, regardless of the actual length of the input. Do this in a new file—don't delete or replace the program file you made in the first task.

Task 3: selectsort and alphabet index

In this task, we want you to read in lowercase letters from the terminal, convert them into their corresponding alphabetical index[4], sort them, and print them out again. For sorting, we provide `sort.c` in `/course/cs034/pub/lab1.1`, which implements the algorithm discussed in class. It's up to you to write the header file, `sort.h`, which goes with it.

You'll also need to write a separate function that can convert your input characters from letters into numbers. It's handy to know that in C you can do subtraction with characters—you can get the alphabetical index of any character letter by subtracting `'a'` from it. You are also allowed to use `isalpha` and `islower`, but make sure you `man` them to be sure you know how to use them.

This conversion function provides an excellent opportunity to practice using `assert`, found in `assert.h`. `assert` is a great way to check conditions that must be true in your program. If the assertion fails, the program will exit. But adding this check won't slow down your code, because the `assert`s can be optimized out once you're done debugging. For example, if your conversion function works correctly, it should return either a number between 0 and 25, inclusive, or an error code that is some number not in [0, 25] (commonly, -1). This is a good thing to check with `assert`.

In addition to practicing with arrays, we're going to take this opportunity to conduct some experiments with pointers. We've talked about pointers, but now that we're in the lab we can actually take a look at a few. We would like you to print out the following memory addresses while your sort is running (print one per line, and label them in some coherent way):

- `main` (that's right, `main` is just a memory address!)
- the address of the first item in the array
- the address of the last item in the array

[4] Meaning, a = 0, b = 1, c = 2, etc

- the address of the item at position `min` right before each call to `swap`
- the address of the item at position `i` right before each call to `swap`

When you've got this task up and running, you'll have evidence before your eyes that there's nothing mystical about pointers. They're just memory addresses, which are just numbers (we print them in hex), stored in a variable of a certain type, and you can print them out, look at them, and compare them.

You should be able to see from the addresses you're printing that all of the selectsort-addresses are in between the first and last address of the list. You should see clearly how the selectsort algorithm works by looking at the addresses of the `i` items and the `min` items for each iteration.

More about header files

We can also use header files for our own function prototypes. Instead of putting our function prototypes at the top of our .c file, we will place them all in a header file and use `#include` to make the compiler recognize them.

If the same header file is included more than once, the compiler will complain that the functions in that file have already been declared. So any header file you create should look like this:

```
#ifndef __NAME_OF_HEADER__
#define __NAME_OF_HEADER__

...
declarations and prototypes
...

#endif /* __NAME_OF_HEADER__ */
```

This makes sure that the header file is only read in once. The __NAME_OF_HEADER__ is often the filename of the header in all caps, but it can be any alphanumeric sequence as long as it is unique, different from any other header name or constant in your program. Using underscores around the header name helps to keep these names unique.

You can use the header file with the rest of your C program by typing:
`#include "name_of_header.h"`
(Note the quotes, instead of the angle brackets. Quotes are for files in your own directory.)

Here's an example of how all this is used in practice:

In the file called foo.h:

```
#ifndef FOO_HEADER
```

```
#define FOO_HEADER

void foo(int a, int b);

#endif
```

In the file called main.c:

```
#include "foo.h"
#include <stdio.h>

int main() {
  foo(1, 2);
  return 0;
}
```

In the file called foo.c:

```
#include "foo.h"

void foo(int a, int b) {
  printf("%d\n", a + b);
}
```

The `foo.h` file is `#included` in both `main.c` and `foo.c` to ensure that they are consistent.

Then we need to compile all the files together, like this:

```
% gcc -Wall -o myExecutable main.c foo.c
```

Task: Write, compile, and execute a C program that reads in up to 100 characters, converts them to their corresponding integer values (using a separate function), uses an assertion to check the work of the conversion function, sorts them using the sorting algorithm in `sort.c`, and prints them out. Before the selectsort is over, your program should print out a decently formatted display of the following memory addresses: `main`, the first item in the list, the last item in the list, and the addresses of the items at positions `i` and `min` for each iteration. Remember, your program should not crash or error no matter what the input, though it is only required to recognize lowercase English letters.

To use the `sort.c` file, copy it into your directory, and write a new file called `sort.h` that declares the functions `sort.c` defines. Then `#include sort.h` in your main program file.