Department of Electrical and Computer Engineering
ECSE 202 – Introduction to Software Development
Assignment 5
Introduction to C Programming

Problem Description

This assignment introduces the concept of a command line program using the "C" programming language. In contrast to Java, the "C" programming environent is somewhat primitive (despite the fact that you can still develop code in Eclipse). The code that you will write will read in two arguments from the command line, a decimal number and base in which to express the decimal number.

Example: convert 1278 to Base-2

```
% dec2base 1278 2
The Base-2 form of 1278 is: 10011111110

Now Base-5

% dec2base 1278 5
The Base-5 form of 1278 is: 20103
```

Notice that the program takes 2 arguments, the number to be converted and the target base, both expressed as decimal numbers. In fact, we can write the program to convert to a default base, e.g. 2:

```
% dec2base 255
The Base-2 form of 255 is: 11111111
```

Command Line Programs

In the second to last Java lecture, we encountered the "standard" program entry point,

```
public class test {
      public static void main(String[] args) {
            // do something
      }
}
```

which wasn't discussed much other than to point out that this program was run from a command line interpreter:

```
C:\Users\ferrie\> java test This is a test.
```

Before control is passed to program test, the command line interpreter (CLI) reads the text on the command line following `java test` and places each white space delimited text into the args string array, i.e., args[0] → This, args[1] → is, args[2] → a, args[3] → test.

In this way the program can process these arguments after the fact. "C" is structured much in the same way, except that since arrays don't have `.length` instance variable, length must be explicitly returned through a second int variable, `argc`, which stands for "argument count".

Let's start off by considering how to run a "C" program. In this course we will assume that you are working with a computer running the Windows, OS X (Mac), or Linux operating systems. Most software written for PC's interacts with the User using a Graphical User Interface, GUI for short. Simply double clicking on an icon activates the program and starts the dialog with the user using a graphical display, mouse and keyboard (welcome to Assignment 4). A much simpler alternative is to use a Command Line Interface (CLI) as shown in the example above. When the CLI starts up, it generally opens up a text window and prints a command prompt, the % character in the example. To run a program you simply type its name and any arguments that go with it. This causes the operating system of the computer to load the program into memory and execute it.

Let's consider starting the CLI in each of the 3 operating systems.

1. Windows:
Click the Windows icon at the lower left hand corner of the screen. This brings up a panel with a search window at the bottom. Type CMD to launch the Windows CLI.

```
Microsoft Windows [Version 10.0.10586]
(c) 2015 Microsoft Corporation. All rights reserved.

C:\Windows\system32>
```

We'll assume that you wrote a version of the Hello World program using the Eclipse IDE (Integrated Development Environment). If you are new to programming, this is probably your safest bet. Otherwise you can use whatever tools suit your needs. In any case, to run your program from the CLI, you first need to know *where* your program resides in your computer's file hierarchy (there is a way around this, but for now let's assume not). If you used eclipse, then each project (all the files comprising your computer program) is usually located in C:\Users\<your user name>\workspace\<project name>. Notice that when you start the CLI on Windows, it defaults to C:\Windows\system32. In order to access our program we will have to change directories using the `cd` (change directory) command as follows:

```
C:\Windows\system32> cd c:\Users\ferrie\workspace

C:\Users\ferrie\workspace>dir
 Volume in drive C has no label.
 Volume Serial Number is 4841-8823

 Directory of c:\Users\ferrie\workspace

2017-06-28  10:19 AM    <DIR>          .
2017-06-28  10:19 AM    <DIR>          ..
2016-07-07  10:40 AM    <DIR>          .metadata
2016-09-12  04:59 PM    <DIR>          argDemo
```

```
2016-09-12  11:44 AM    <DIR>          classDemo
2016-07-07  10:41 AM    <DIR>          Hello-C
2016-07-07  11:07 AM    <DIR>          HelloWorld
2017-02-28  11:16 PM    <DIR>          JCalcGUI
2017-06-28  10:19 AM    <DIR>          myHello
               0 File(s)              0 bytes
               9 Dir(s)  112,799,371,264 bytes free

C:\Users\ferrie\workspace>
```

The default behavior of the windows CLI is to echo the current location before the command prompt. To list the contents of the workspace directory, I followed with a `dir` (directory) command. This lists all active projects in my directory. The "C" program I want to run is in project myHello, so I have to change directories again.

```
C:\Users\ferrie\workspace>cd myHello

C:\Users\ferrie\workspace\myHello>dir
 Volume in drive C has no label.
 Volume Serial Number is 4841-8823

 Directory of c:\Users\ferrie\workspace\myHello

2017-06-28  10:19 AM    <DIR>          .
2017-06-28  10:19 AM    <DIR>          ..
2017-06-28  10:19 AM            11,468 .cproject
2017-06-28  10:19 AM               785 .project
2017-06-28  10:19 AM    <DIR>          .settings
2017-06-28  10:19 AM    <DIR>          Debug
2017-06-28  10:19 AM    <DIR>          src
               2 File(s)         12,253 bytes
               5 Dir(s)  112,799,248,384 bytes free

C:\Users\ferrie\workspace\myHello>
```

Again, I used the `dir` command to list the contents of the `myHello` project folder. The actual program (executable) lives in the Debug directory, so we need to change directories one more time.

```
C:\Users\ferrie\workspace\myHello>cd Debug

C:\Users\ferrie\workspace\myHello\Debug>dir
 Volume in drive C has no label.
 Volume Serial Number is 4841-8823

 Directory of c:\Users\ferrie\workspace\myHello\Debug
```

```
2017-06-28  10:19 AM    <DIR>          .
2017-06-28  10:19 AM    <DIR>          ..
2017-06-28  10:19 AM            82,300 myHello.exe
2017-06-28  10:19 AM    <DIR>          src
               1 File(s)         82,300 bytes
               3 Dir(s)  112,799,326,208 bytes free
```

The file containing the program is the one with the `.exe` extension, i.e., `myHello.exe`.

To run the program, simply type the name (no need to include the extension):

```
C:\Users\ferrie\workspace\myHello\Debug>myHello
!!!Hello World!!!

C:\Users\ferrie\workspace\myHello\Debug>
```

Of course we could have simply cut to the chase and run the program directly by specifying the complete path:

```
C:\Windows\system32>c:\Users\ferrie\workspace\myHello\Debug\myHe
llo
!!!Hello World!!!

C:\Windows\system32>
```

The procedure is similar for Mac or Linux users. In the OS X environment, run the Terminal or XQuartz applications. For Linux, XTerm is the default application. In both cases, the CLI (or shell in Linux parlance) will start off in your home directory, so all you need to do is use the `cd` command to change to the appropriate workspace location, e.g.,

```
ferrie@lizard{myHello}: cd ~/Documents/workspace/myHello/Debug
ferrie@lizard{Debug}: ls
makefile        myHello         objects.mk      sources.mk
src
ferrie@lizard{Debug}:
```

Note a couple of subtle changes. First, the workspace directory is usually placed under the Documents folder in the Mac environment and under the home directory in the Linux environment. Instead of using the `dir` command, the `ls` (list) command is used.

Admittedly there are a lot of details here that can be somewhat overwhelming. The trick is to take things one step at a time and make use of online resources (Google is your friend) when you hit a wall. The first tutorial will help you to set things up on your own computer, so that your introduction to software development can be as painless as possible.

The Structure of a “C” Program (revisited)

The “C” programming language was written originally for a UNIX (predecessor of Linux) environment where interactive programs are run by a shell (CLI) program such as `sh`. A consequence of this is that a “C” program has the form of a *function*. Notice the similarity to Java.

```
int main (int argc, char *argv[])
{
  printf(“Hello World!\n”);
}
```

You can literally take this code, build the code (it will generate warnings), and run it. Let’s look at things in more detail. This function, called `main`, takes 2 arguments. The first, `argc`, is an integer variable, and the second, `argv` is an array of character strings. We will go into more detail as to why this is so in future lectures. These variables are filled in by the CLI when the program is run from the command line. The function itself can return a value (it doesn’t return anything here), which is enables programs to be combined together in scripts (another topic). For this assignment we need to figure out how to pass arguments from the command line to the program – time for another example.

```
1     #include <stdio.h>
2
3     int main(int argc, char *argv[]) {
4       int a, b;
5       if (argc != 3) {
6         printf(“wrong number of arguments\n”);
7         return(0);
8       }
9       sscanf(argv[1],”%d”,&a);
10      sscanf(argv[2],”%d”,&b);
11      printf(“%d + %d = %d\n”, a, b, a+b);
12    }
```

Let’s assume that I created a file called plus.c containing the above code (in a project called plus within Eclipse). What will it do? Let’s evaluate this bit of code line by line:

1. Any time you use a function in a “C” program, you have to provide a definition that defines its parameters. These are usually contained in an include file. The notation used above references a system include file for the functions used in this program, i.e., `sscanf` and `printf.`
2. Blank lines can increase the readability of a program.
3. The CLI views each command line as a set of character strings separated by whitespace, i.e., characters such as spaces and tabs. For example if you entered

   ```
   ferrie@FastCat{ferrie}: plus 3 5
   ```

the command line arguments would consist of 3 character strings, `plus, 3 and 5` respectively. In this case variable `argc` would have a value of 3. The structure of the `argv` variable is a bit more complicated. As we will see later in these lectures, character strings are represented by what is called a *pointer*, a variable that holds the memory address (i.e. points to) of the first character in the string. This is not unlike Java where argv would be a reference to a block of memory on the heap. So `argv` corresponds to, in fact, an array of pointers, one for each character string in the command line. In this example, `argv[0]` corresponds to the character string `plus`, `argv[1]` to the character string for 3 and `argv[2]` to `5`. The details of these representations will be discussed in more detail in the subsequent lectures. For now all you need to know is how to gain access to the command line arguments.

4. Every piece of information that is explicitly represented in a computer program must have a corresponding variable with a data type that matches its usage. Since this program will compute the sum of two integers, they are explicitly represented by variables `a` and `b`.
5. Any program that interacts with user input needs to do at least some rudimentary checking to make sure that sufficient data has been received for the program to produce the expected output. In this case, since we're adding two integer values, we need to make sure that 2 arguments have been supplied. In this case `argc=3`; recall that `argc` returns the total number of tokens on the command line, including the program name.
6. If the number of arguments is wrong, we send a message to the user using the `printf` function. Without going into detail, `printf` displays anything between the double quotes literally, except for tokens indicated with the `%` character. In the example above, the first instance of `%d` means convert the value of the first matching variable, i.e. `a`, into a string of numbers representing a decimal number, and substitute into the expression. The next instance does the same for variable `b`, and the final one for the value of `a+b`.
7. This causes the program to terminate and return to the CLI.
8. Termination of the `if` clause.
9. Recall that `argv[1]` corresponds to the character string 3 (or whatever the user entered, e.g., 52764), that corresponds to a decimal number. Here the `sscanf` function is used to convert a string of characters, `argv[1]`, that represent a decimal number, `%d`, into the computer's internal binary representation, `b`. The ampersand character, &, passes the location of `b` in memory to `sscanf` so that the converted value can be returned directly.
10. Do the same for the second argument and return the value to variable `b`.
11. At this point we succeeded in reading two values from the command line and storing them as binary variables. Here we compute the sum, convert the result to a character string, and print out the result using the `printf` function as in Line 6. Notice that the third instance of `%d`, which corresponds to the sum, operates on an expression (a+b) instead of a single variable.
12. Close of program.

Given the shared syntax of Java and "C", it should not be too difficult to implement this program.

Instructions

1. Write a "C" command line program, dec2base, as shown at the beginning of these assignment notes. It should handle each of the cases shown (wrong number of arguments, one argument, two arguments) and render the result in the target base. It must also check that the input arguments are within bounds.

2. Your program must implement the following functions:

   ```
   void dec2base(int input,int base,char *str);

   void revStr(char *str, int length);
   ```

   Note that neither of these functions use the function return mechanism. Both functions use the pass by reference mechanism to write their results directly to the corresponding variables in the caller program. Function dec2base performs decimal to target base conversion as per Assignment 1 and writes the result to a string in *reverse order*. This makes it particular easy (simple) to implement with very few lines of code. Although the <string.h> library has a string reverse function, you are to implement one yourself to gain familiarity in how "C" implements character strings. Note that since you determine the length of the output string in dec2base, you can use this to simplify the writing of revStr. Also take note that unlike Java, strings are not immutable.

   You may not use any other functions in writing dec2base and revStr.

3. Run the following examples, making sure that your results are correct. Save these results to a file called dec2base.txt.

   ```
   ferrie@Lizard{src}: dec2base 65535
   The Base-2 form of 65535 is: 1111111111111111
   ferrie@Lizard{src}: dec2base 65535 16
   The Base-16 form of 65535 is: FFFF
   ferrie@Lizard{src}: dec2base 3333333 3
   The Base-3 form of 3333333 is: 20021100110210
   ferrie@Lizard{src}: dec2base 100000 36
   The Base-36 form of 100000 is: 255S
   ferrie@Lizard{src}: dec2base 0
   The Base-2 form of 0 is: 0
   ferrie@Lizard{src}: dec2base 104976 18
   The Base-18 form of 104976 is: 10000
   ferrie@Lizard{src}: dec2base 104975 18
   The Base-18 form of 104975 is: HHHH
   ferrie@Lizard{src}: dec2base 2147483647 16
   The Base-16 form of 2147483647 is: 7FFFFFFF
   ferrie@Lizard{src}: dec2base 2147483647 32
   The Base-32 form of 2147483647 is: 1VVVVVV
   ferrie@Lizard{src}: dec2base 2147483648
   Error: number must be in the range of [0,2147483647]
   ferrie@Lizard{src}: dec2base -100 4
   Error: number must be in the range of [0,2147483647]
   ferrie@Lizard{src}: dec2base 100 -4
   Error: base must be in the range [2,36]
   ferrie@Lizard{src}: dec2base 436567890 12
   The Base-12 form of 436567890 is: 102257556
   ```

4. Make sure that your "C" source file is properly documented and that it contains your name and student ID in the comments. Save this file as dec2base.c

5. Upload your files to myCourses as indicated.

About Coding Assignments

We encourage students to work together and exchange ideas. However, when it comes to finally sitting down to write your code, this must be done *independently*. Detecting software plagiarism is pretty much automated these days with systems such as MOSS.

https://www.quora.com/How-does-MOSS-Measure-Of-Software-Similarity-Stanford-detect-plagiarism

Please make sure your work is your own. If you are having trouble, the Faculty provides a free tutoring service to help you along. You can also contact the course instructor or the tutor during office hours. There are also numerous online resources – Google is your friend. The point isn't simply to get the assignment out of the way, but to actually learn something in doing.

fpf/Nov 5, 2018