Department of Electrical and Computer Engineering
ECSE 202 – Introduction to Software Development
Assignment 1
Introduction to C Programming

Problem Description

The objective of this assignment is to learn how to write a simple "C" program that uses a command line interface, a very basic form of person-machine interaction. Admittedly, this is rather like teaching someone how to swim by throwing them into a pool, but it is an effective way to introduce the problem of developing software. Specifically, this program, which we'll call `dec2base` (*decimal to base*) will convert an input number in Base-10 to an integer value in Base-X where X represents the target base.

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
```

Where Do I Begin?

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. 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

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*:

```
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 (in fact `argv` is an example of the infamous pointer variables). 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 the course, 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. 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.

The program you need to write follows this example closely, with the exception that we require an *algorithm* to convert a decimal value into a corresponding string of digits.

Decimal to Binary Conversion

The ulterior motive behind this assignment (besides figuring out how to do basic coding in "C") is to explore the internal representation in most standard digital computers, i.e., binary. When you use a decimal constant in your program, the compiler translates it into its corresponding binary representation. Equivalently, when you read in a decimal number from the command line interface with the %d conversion specified, a function is called which converts from decimal to binary representation.

Let's begin by considering binary numbers:

$124_{10} = 01111100_2$

Note the convention of using subscripts to denote base of representation. If no base is specified, the default is to assume Base-10. In more detail,

$$124_{10} = 01111100_2 = 0\times2^7 + 1\times2^6 + 1\times2^5 + 1\times2^4 + 1\times2^3 + 1\times2^2 + 0\times2^1 + 0\times2^0 \ .$$

The general form of this representation is

$$\begin{aligned} I_{10} &= b_{n-1}\times 2^{n-1} + b_{n-2}\times 2^{n-2} + \cdots + b_1\times 2^1 + b_0\times 2^0, \\ &= \sum_{i=0}^{i=n-1} 2^i \times b_i. \end{aligned} \quad . \tag{1}$$

So the problem of decimal to binary conversion reduces to finding the coefficients of the expansion for I_{10} above, where n represents the number of bits in the binary representation. We start off by writing I as an even number plus 0 or 1 as appropriate,

$$I = Q^{(0)}\times 2 + R^{(0)} \quad . \tag{2}$$

Similarly we can write

$$Q^{(0)} = Q^{(1)}\times 2 + R^{(1)}, \tag{3}$$

or

$$I = (Q^{(1)}\times 2 + R^{(1)})\times 2 + R^{(0)} \quad . \tag{4}$$

We can continue expanding I recursively,

$$\begin{aligned} I &= (Q^{(1)}\times 2 + R^{(1)})\times 2 + R^{(0)} \\ &= ((Q^{(2)}\times 2 + R^{(2)})\times 2 + R^{(1)})\times 2 + R^{(0)} \\ &= (((Q^{(3)}\times 2 + R^{(3)})\times 2 + R^{(2)})\times 2 + R^{(1)})\times 2 + R^{(0)} \end{aligned} \quad . \tag{5}$$

Expanding and collecting terms we obtain

$$\begin{aligned} I &= 2\times 2\times 2\times 2\times Q^{(3)} + 2\times 2\times 2\times R^{(3)} + 2\times 2\times R^{(2)} + 2\times R^{(1)} + R^{(0)} \\ &= 2^4\times Q^{(3)} + 2^{(3)}\times R^{(3)} + 2^{(2)}\times R^{(2)} + 2^{(1)}\times R^{(1)} + R^{(0)} \\ &= \sum_{i=0}^{n-1} 2^i \times R^{(i)} \end{aligned} \quad , \tag{6}$$

which is exactly the same form as Equation (1) at the top of the page for a binary number. In other words, the remainder terms, $R^{(i)}$, are the binary coefficients, b_i.

Decimal to Binary Conversion Algorithm

The key to a decimal to binary conversion algorithm can be seen in Equation (5). The superscripts attached to *Q* and *R* can be thought of as steps of an iteration, starting at 0 and progressing to *n-1*, where *n* is the number of bits in the binary number. Let's try this out starting with 0, and let 13 be the "input" to our algorithm:

$13 = Q^{(0)} \times 2 + R^{(0)} = 6 \times 2 + 1$ $\quad Q^{(0)} = 6,\ R^{(0)} = 1$
$6 = 3 \times 2 + 0$ $\quad Q^{(1)} = 3,\ R^{(1)} = 0$
$3 = 1 \times 2 + 1$ $\quad Q^{(2)} = 1,\ R^{(2)} = 1$
$1 = 0 \times 2 + 1$ $\quad Q^{(3)} = 0,\ R^{(3)} = 1$

Since $Q^{(3)} = 0$, the algorithm can go no further. The result is 1101which is the binary representation of decimal value 13.

Here is the decimal to binary conversion algorithm in psuedocode:

```
assign lastQ = number to be converted
assign i = 0

while (lastQ > 0) {
     Q = integer (lastQ/2)
     R[i] = remainder (lastQ/2)
     lastQ = Q;
     i  = i+1;
}
```

Let's "try" this algorithm with 124 as input.

```
lastQ = 124,   I = 0

Iteration 0:   Q = 124/2 = 62,     R[0] = 0
Iteration 1:   Q = 62/2 = 31,      R[1] = 0
Iteration 2:   Q = 31/2 = 15,      R[2] = 1
Iteration 3:   Q = 15/2 = 7,       R[3] = 1
Iteration 4:   Q = 7/2 = 3,        R[4] = 1
Iteration 5:   Q = 3/2 = 1,        R[5] = 1
Iteration 6:   Q = 1/2 = 0,        R[6] = 1
```

Result = 1111100

The code in the `while` clause of the psuedocode executes repeatedly as long as the control variable, lastQ, is greater than 0. Each pass through the code is referred to as an iteration, so we can see that it takes 7 iterations to convert 124 to binary. Let's take a closer look at the code in the `while` clause.

The first line computes the current value of Q based on its last value. If Q was assigned as an integer datatype in "C", this statement could be written simply as `Q = lastQ/2,` since the assignment operator, `=`, will store the integer part of the right hand side by definition. In "C", the modulus operator, `%`, returns the remainder after performing integer division. That statement would be written as `R[i]=lastQ % 2`. Since we want to keep track of each digit of the answer, we need to create an array, `R[sizeof(int)*8]`, to hold each bit of the result. We will cover "C" constructs in more detail during the lectures, but in "C" the `sizeof` operator returns the size of the argument in parentheses in bytes. Unlike Java, datatypes in "C" are machine dependent, so we use the `sizeof` operator here to figure out the size of an integer in bytes, then multiply by 8 to get the size in bits. Such details can be very confusing when starting out, especially if you have never done coding before. For now, concentrate on the general ideas and use the code templates provided to help you complete this programming assignment.

By the way, Equations (1) to (6) can easily be amended to handle *any* base. Instead of dividing by 2, you divide by the target base. Consider converting from decimal to hexadecimal notation (Base-16).

```
LastQ = 124,    I = 0

Iteration 0:    Q = 124/16 = 7,          R[0] = 12
Iteration 1:    Q = 7/16 = 0,            R[1] = 7
```

Result = 7C

A couple of observations are in order. First, if you replace each hexadecimal digit by its binary equivalent, one ends up with 7C = 01111100. Second, in printing out the result, as long as the target base is in the range of 2-10, the remainder will be in the range of [0,9], so each digit can be printed out directly. However when the base is 11 and above, the convention is to substitute letters of the alphabet starting at A. For the case of Base-16, the equivalents are as follows:

10	A
11	B
12	C
13	D
14	E
15	F

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.

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

```
ferrie@FastCat{Assignment 1}: dec2base 255
The Base-2 form of 255 is: 11111111

ferrie@FastCat{Assignment 1}: dec2base 65535 16
The Base-16 form of 65535 is: FFFF

ferrie@FastCat{Debug}: dec2base 2147483647
The Base-2 form of 2147483647 is: 11111111111111111111111111111111

ferrie@FastCat{Debug}: dec2base 33975 5
The Base-5 form of 33975 is: 2041400

ferrie@FastCat{Debug}: dec2base 760976 9
The Base-9 form of 760976 is: 1378768

ferrie@FastCat{Debug}: dec2base 67878903 10
The Base-10 form of 67878903 is: 67878903

ferrie@FastCat{Debug}: dec2base 86 33
The Base-33 form of 86 is: 2K

ferrie@FastCat{Debug}: dec2base 100100100 3
The Base-3 form of 100100100 is: 20222100121112010

ferrie@FastCat{Debug}: dec2base 2147483647 35
The Base-35 form of 2147483647 is: 15V22UM

ferrie@FastCat{Debug}: dec2base 17 8
The Base-8 form of 17 is: 21
```

3. 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

4. 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/July 4, 2017