CMPS 12M
Introduction to Data Structures Lab
Winter 2009

Lab Assignment 3

Due Wednesday February 4, 10:00 pm

The purpose of this lab assignment is to introduce the C programming language, including standard input-output functions, command line arguments, File IO, and compilation with makefiles.

Introduction to C

If you are not already familiar with C (or even if you are) it is recommended that you purchase a good C reference such as *C for Java Programmers: a Primer* by Charlie McDowell (Lulu.com 2006). The C programming language is in a certain sense the grandparent of java (C++ being its parent). Java is known as an Object Oriented Programming (OOP) language, which means that data structures and the procedures which operate on them are grouped together into one language construct, namely the *class*. Common behavior amongst classes is specified explicitly through the mechanism of inheritance. The C programming language on the other hand does not directly support OOP (although OOP can be implemented with some effort). C is known as a procedural programming language, which means that data structures and procedures are separate language constructs. There are no classes, no objects, and no inheritance. New data types in C are created using the `typedef` and `struct` constructs, which will be illustrated in future lab assignments. There is however much common syntax between java and C. Many control structures such as loops (while, do-while, for), and branching (if, if-else, switch) are virtually identical in the two languages. One major difference is in the way program input and output is handled, both to and from standard IO devices (keyboard and screen), and to and from files. The following is an example of a "Hello World!" program in C.

Example

```
/*
*  hello.c
*  Prints "Hello World!" to stdout
*/
#include <stdio.h>

int main(void){
   printf("Hello World!\n");
   return 0;
}
```

Comments in C are specified by bracketing them between the strings `/*` and `*/`, and may span several lines. For instance `/* comment */` or

```
/* comment
   comment */
```

or

```
/*
* comment
* comment
*/
```

are all acceptable. Any line beginning with # is known as a *preprocessor directive*. The preprocessor performs the first phase of compilation during which these directives, which are literal text substitutions, are performed, making the program ready for later stages of compilation. The line `#include <stdio.h>` inserts the standard library header file `stdio.h` which specifies functions for performing standard input-output operations. One can also specify constant macros using the `#define` preprocessor directive as follows.

```
/*
*  hello.c
*  Prints "Hello World!" to stdout
*/
#include <stdio.h>
#define HELLO_STRING "Hello World!\n"

int main(void){
   printf(HELLO_STRING);
   return 0;
}
```

Notice that preprocessor commands in C do not end in a semicolon. Although you can call most C programs anything you want, each C program must contain exactly one function called *main*. Function main is the control module, i.e. the point at which program execution begins. Typically main will call other functions, which may in turn call other functions. Function definitions in C look very much like they do in java.

```
return-type function-name(data-type variable-name, data-type variable-name,. . .){

   /* declarations of local variables */

   /* executable statements */

   /* return statement (provided return-type is not void) */
}
```

One difference however is that local variables must be declared at the beginning of the function body before any executable statements. Variable declarations in C may not be interspersed with executable statements as they can in java.

The function `printf()` prints formatted text to `stdout`. It's first argument is known as a format string and consists of two types of items. The first type is made up of characters which will be printed to the screen. The second type contains format commands that define the way the remaining arguments are displayed. A format command begins with a percent sign and is followed by the format code. There must be exactly the same number of format commands as there are remaining arguments, and the format commands are matched with the remaining arguments in order. For example

```
printf("There are %d days in %s\n", 30, "April");
```

prints the text "There are 30 days in April". Some common format commands are:

```
%c    character
%d    signed decimal integer
%f    decimal floating point
%s    string of characters
%e    scientific notation
%%    prints a percent sign
```

See a good reference on C for other format commands. (Note java contains a method `System.out.printf` which behaves exactly like `printf` in C)

Observe that the above main function has return type int. A return value of 0 indicates to the caller (i.e. the operating system) that execution was nominal and without errors. Actually the proper exit values for main are system dependent. For the sake of portability one should use the exit codes EXIT_SUCCESS and EXIT_FAILURE which are predefined constants found in the library header file stdlib.h. The exit code allows for the status of the main program to be tested at the level of the operating system. (In Unix the exit code will be stored in the shell variable `$status`.)

```
/*
*  hello.c
*  Prints "Hello World!" to stdout
*/
#include <stdio.h>
#include <stdlib.h>
#define HELLO_STRING "Hello World!\n"

int main(void){
   printf(HELLO_STRING);
   return EXIT_SUCCESS;
}
```

Compiling a C program

A C program may be comprised of any number of source files. Each source file name must end with the extension `.c`. There are four components to the compilation process:

$$\text{Preprocessor} \rightarrow \text{Compiler} \rightarrow \text{Assembler} \rightarrow \text{Linker}$$

The preprocessor expands symbolic constants in macro definitions and inserts library header files. The compiler creates assembly language code corresponding to the instructions in the source file. The assembler translates the assembly code into native machine readable object code. One object file is created for each source file. Each object file has the same name as the corresponding source file with the `.c` extension replaced by `.o`. The linker searches specified libraries for functions which the program uses (such as `printf()` above) and combines pre-compiled object code for those functions with the program's object code. The finished product is a complete executable binary file. Most Unix systems provide several C compilers. We will use the `gcc` compiler exclusively in this course. To create the object file `hello.o` do

```
%gcc –c –ansi –Wall hello.c
```

(Remember `%` stands for the unix prompt.) The `–c` option means compile only (i.e. do not link), `-ansi` means adhere to the ANSI C language standard, and `–Wall` means print all recommended warnings. To link `hello.o` to the standard library functions in `stdio.h` and `stdlib.h` do

```
%gcc –o hello hello.o
```

The `–o` option specifies `hello` as the name of the executable binary file to be created. If this option is left out the executable will be named `a.out`. To run the program type its name at the command prompt.

```
%hello
Hello World!
```

The whole four step process can be automated by doing

```
%gcc –ansi –Wall –o hello hello.c
```

which automatically deletes the object file `hello.o` after linking. If the program resides in just one source file, this may be the simplest way to compile. If the program is spread throughout several source files (as will often be the case in this course) a makefile should be used to automate compilation. Here is a simple makefile for `hello.c`.

```
# makefile for hello.c

hello : hello.o
      gcc -o hello hello.o

hello.o : hello.c
      gcc -c -ansi -Wall hello.c

clean :
      rm -f hello hello.o
```

Here is an equivalent makefile which uses macros.

```
# makefile for hello.c with Macros

FLAGS = -ansi -Wall
SOURCES = hello.c
OBJECTS = hello.o
EXEBIN = hello

all: $(EXEBIN)

$(EXEBIN) : $(OBJECTS)
      gcc -o $(EXEBIN) $(OBJECTS)

$(OBJECTS) : $(SOURCES)
      gcc -c $(FLAGS) $(SOURCES)

clean :
      rm -f $(EXEBIN) $(OBJECTS))
```

See the examples page for a fancy hello world program in C which prints out operating system environment variables.

Command Line Arguments

The main function in a C program may have two possible prototypes:

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

The second prototype is used for C programs which use command line arguments. The first argument `argc` specifies the number of string arguments on the command line (including the program name.) The second argument is an array containing the string arguments themselves. (Note that in C a string is just an array of `chars` which is specified by the type `char*`.) The parameter `argc` is necessary since arrays in C do not know their own lengths, unlike java. The following program behaves similarly to it's namesake in lab2.

```
/*
*  CommandLineArguments.c
*/
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[]){
   int i;
   printf("argc = %d\n", argc);
   for(i = 0; i<argc; i++ ){
      printf("%s\n", argv[i]);
   }
   return EXIT_SUCCESS;
}
```

Compile this program and run it with several different sets of arguments. Notice that there is one important difference between this program and the java version from lab2. If the executable is called `CLA` for brevity, then the java version gives

```
%CLA x y z
args.length = 3
x
y
z
```

whereas the C version above gives

```
%CLA x y z
argc = 4
CLA
x
y
z
```

We see that in C, `argv[0]` is the name of the command which invoked the program, i.e. `CLA`. In java the array of strings which holds the command line arguments does not include the name of the executable.

Console Input and Output

The function `printf()` discussed above sends its output to *stdout*, which is the data stream normally associated with the computer screen. The library `stdio.h` also includes a function called `scanf()`, which is a general purpose input routine that reads the stream *stdin* (normally associated with the keyboard) and stores the information in the variables pointed to by its argument list. It can read all the built-in data types and automatically convert them into the proper internal format. The first argument to `scanf()` is a format string consisting of three types of characters: format specifiers, whitespace characters, and non-whitespace characters. The format specifiers are preceded by a `%` sign and tell `scanf()` what type of data is to be read. For example `%s` reads a string while `%d` reads an integer. Some common format codes are:

```
%c    character
%d    signed decimal integer
%f    decimal floating point
%s    string of characters
```

See a good C reference for other `scanf()` codes, which are similar but not identical to `printf()`'s codes. The format string is read left to right and the format codes are matched, in order, with the remaining arguments that comprise the argument list of `scanf()`. A whitespace character (i.e. space, newline, or tab) in the format string causes `scanf()` to skip over one or more whitespace characters in the input stream. In other words, one whitespace character in the format string will cause `scanf()` to read, but not to store, any number (including zero) of whitespace characters up to the first non-whitespace character. A non-whitespace character in the format string causes `scanf()` to read and discard a single matching character in the input stream. For example, the control string `" %d, %s"` causes `scanf()` to first skip any number of leading whitespace characters, then read one integer, then read and discard one comma, then skip any number of whitespace characters, then read one string. If the specified character is not found, `scanf()` will terminate.

All the variables receiving values through `scanf()` must be passed by reference, i.e. the address of each variable receiving a value must be placed in the argument list. We use the "*address-of*" operator `&` to specify the address of a variable. For instance the following program asks the user for three integers, then echoes them back to the screen.

```
#include<stdio.h>
#include<stdlib.h>
int main(void){
   int x, y, z;
   printf("Enter three integers separated by commas, then press return: ");
   scanf(" %d, %d, %d", &x, &y, &z);
   printf("The integers entered were %d, %d, %d\n", x, y, z);
   return EXIT_SUCCESS;
}
```

A sample run of this program looks like

```
% a.out
Enter three integers separated by commas, then press return: 12,-7, 13
The integers entered were 12,-7, 13
```

Running again, but this time leaving out the separating commas in the input line gives

```
% a.out
Enter three integers separated by commas, then press return: 12 -7 13
The integers entered were 12, 4, -4261060
```

Since the comma separating the first and second integers was left out `scanf()` read the first integer, then expected to read and discard a comma but failed to do so, then just returned without reading anything else. The values printed for variables `y` and `z` are random and may be different when you run the program. Thus we see that `scanf()` is intended for reading *formatted* input. (This is what the "`f`" it its name stands for). The `scanf()` function returns a number equal to the number of fields which were successfully assigned values. That number can be tested and used within the program, as the following example illustrates.

```
#include<stdio.h>
#include<stdlib.h>
int main(void){
   int n, i; int x[3];
   printf("Enter three integers separated by spaces, then press return: ");
   n = scanf(" %d %d %d", &x[0], &x[1], &x[2]);
   printf("%d numbers were successfully read: ", n);
   for(i=0; i<n; i++) printf("%d ", x[i]);
   printf("\n");
   return EXIT_SUCCESS;
}
```

Some sample runs of this program follow:

```
% a.out
Enter three integers separated by spaces, then press return: 1 2 G
2 numbers were successfully read: 1 2
% a.out
Enter three integers separated by spaces, then press return: monkey at the keyboard!
0 numbers were successfully read:
```

If an error occurs before the first field is assigned, then `scanf()` returns the pre-defined constant `EOF`. Reading the end-of-file character is considered such an error. You can place an end-of-file character into the input stream by typing `control-D`. The value of `EOF` is always a negative integer and is defined in the header file `stdlib.h`. Evidently the value of `EOF` on my system is `-1`, as the following test run illustrates.

```
% a.out
Enter three integers separated by spaces, then press return: ^D
-1 numbers were successfully read:
%
```

File Input and Output

The library header file `stdio.h` defines many functions for doing various types of input and output. See any good C reference for a listing of these functions (e.g. `getc()`, `getchar()`, `gets()`, `putc()`, `putchar()`, and `puts()` are commonly used by C programmers). Most of these functions have both a console version and a file version. The functions `fprintf()` and `fscanf()` are the file equivalents of `printf()` and `scanf()` respectively. They operate exactly as do `printf()` and `scanf()` except that they have an additional (first) argument which specifies a file to write to or read from. This file "handle" as it is sometimes called, is a variable of type `FILE*`, which is defined in `stdio.h`. Files are opened and closed using the `fopen()` and `fclose()` functions. The following program illustrates the use of functions `fopen()`, `fclose()`, `fscanf()`, and `fprintf()`.

```
/*
*  FileIO.c
*  Reads input file and prints each word on a separate line of
*  the output file.
*/
#include<stdio.h>
#include<stdlib.h>

int main(int argc, char* argv[]){
   FILE* in;  /* file handle for input */
   FILE* out; /* file handle for output */
   char word[256]; /* char array to store words from input file */

   /* check command line for correct number of arguments */
   if( argc != 3 ){
      printf("Usage: %s input-file output-file\n", argv[0]);
      exit(EXIT_FAILURE);
   }

   /* open input file for reading */
   in = fopen(argv[1], "r");
   if( in==NULL ){
      printf("Unable to read from file %s\n", argv[1]);
      exit(EXIT_FAILURE);
   }
```

```
   /* open output file for writing */
   out = fopen(argv[2], "w");
   if( out==NULL ){
      printf("Unable to write to file %s\n", argv[2]);
      exit(EXIT_FAILURE);
   }

   /* read each word in input file, print words on separate lines to output file */
   while( fscanf(in, " %s", word) != EOF ){
      fprintf(out, "%s\n", word);
   }

   /* close input and output files */
   fclose(in);
   fclose(out);

   return(EXIT_SUCCESS);
}
```

Notice that by inserting the leading space in the format string for `fscanf()`, we skip all intervening whitespace characters and thus parse the tokens in the file. (Recall that token is here defined to mean a substring which is maximal with respect to the property of containing no whitespace characters.)

What to turn in

Write a C program called `FileReverse.c` which behaves exactly like the program `FileReverse.java` which you wrote in lab 2. Thus FileReverse.c will take two command line arguments which are the input and output files respectively (following the `FileIO.c` example above.) Your program will read each word in the input file, then print it backwards on a line by itself. For example given a file called `in` containing the lines:

```
abc def ghij
klm nopq rstuv
w xyz
```

the command `%FileReverse in out` will create a file called `out` containing the lines:

```
cba
fed
jihg
mlk
qpon
vutsr
w
zyx
```

Your program will contain a function called `stringReverse()` with the heading

```
void stringReverse(char* s)
```

which reverses its string argument. Place the definition of this function after all preprocessor directives but before the function `main()`. Your main function will be almost identical to `FileIO.c` above, except that the while loop will contain a call to `stringReverse(word)`. Although it is possible to write `stringReverse()` as a recursive function, it is not recommended that you do so unless you are very familiar with C strings and the string handling functions in the standard library `string.h`. Instead it is recommended that you implement a simple iterative algorithm to reverse the string `s` (see below).

There is one function from `string.h` that you *will* need however, and that is `strlen()`, which returns the length of its string argument. For example the following program prints out the length of a string entered on the command line.

```
/*
*  charCount.c
*  prints the number of characters in a string on the command line
*/
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

int main(int argc, char* argv[]){
   if(argc<2){
      printf("Usage: %s some-string\n", argv[0]);
      exit(EXIT_FAILURE);
   }
   printf("%s contains %d characters\n", argv[1], strlen(argv[1]) );
   return EXIT_SUCCESS;
}
```

Remember that a string in C is a `char` array, not a separate data type as it is in java. Recall also that arrays in C cannot be queried as to their length. How then does `strlen()` work? Actually a string in C is a little bit more than a `char` array. By convention C strings are terminated by the null character `'\0'`. This character acts as a sentinel, telling the functions in `string.h` where the end of the string is. Function `strlen()` returns the number of characters in its `char*` (`char` array) argument up to (but not including) the null character.

Here is an algorithm in high level pseudo-code which will perform the string reversal:

1. Set two indices `i` and `j` where `i=0` and `j=strlen(s)-1`
2. Swap the characters `s[i]` and `s[j]`
3. Increment `i` and decrement `j`
4. Stop when `i>=j`

One more question should be answered here. Are arrays in C passed by value or by reference? Obviously by reference, for otherwise functions like `stringReverse()` would have no effect on their array arguments outside the function call. In fact an array name in C is literally the address of (i.e. a pointer to) the first element in the array. Arrays, strings, and pointer variables in C will be discussed in subsequent lab assignments.

Write a makefile which creates an executable binary file called `FileReverse`, and includes a `clean` utility. Submit the files: `README`, `makefile`, and `FileReverse.c` to the assignment name lab3. Start early and ask for help if you are at all confused.