CSE 15
Introduction to Data Structures
Lab Assignment 1

The purpose of this assignment is threefold: (1) get a basic introduction to the Andrew File System which is used by the ITS Unix timshare, (2) learn how to compile a C program on the timeshare, and (3) learn to automate compilation and other tasks using Makefiles.

The Andrew File System (AFS)
Logon to your ITS unix timeshare account at `unix.ucsc.edu`. If you don't know how to do this, ask for help at a lab session, or see Lab Assignment 1 from my CMPS 12A/L Spring 2018 webpage:

https://classes.soe.ucsc.edu/cmps012a/Spring18/lab1.pdf

Create a subdirectory within your home directory called `cse15` in which you will keep all your work for CSE 15/CSE 15L. Create a subdirectory within `cse15` called `lab1`. From within `lab1` create a subdirectory called `private`, then set access permissions on the new directory so that other users cannot view its contents. Do all this (starting in your home directory) by typing the lines below. The Unix prompt is depicted here as `$` (although it may look different in your login session). The lines without the Unix prompt are the output of your typed commands.

```
$ mkdir cse15
$ cd cse15
$ mkdir lab1
$ cd lab1
$ mkdir private
$ fs setacl private system:authuser none
$ fs listacl private
Access list for private is
Normal rights:
  foobar rlidwka
```

Here `foobar` will be replaced by your own cruzid. The last line of output says that your access rights to directory `private` are `rlidwka` which means: read, list, insert, delete, write, lock, and administer. In other words you have all rights in this directory, while other users have none. If you are unfamiliar with any Unix command, you can view its manual page by typing: `man <command name>`. (Do not type the angle brackets <>.) For instance `man mkdir` brings up the man pages for `mkdir`. Man pages can be very cryptic, impenetrable even, especially for beginners, but it is best to get used to reading them as soon as possible. Under AFS, `fs` denotes a file system command, `setacl` sets the access control list (ACL) for a specific user or group of users, and `listacl` displays the access lists for a given directory. The command

```
$ fs setacl <some directory> <some user> <some subset of rlidwka or all or none>
```

sets the access rights for a directory and a user. Note that `setacl` can be abbreviated as `sa` and `listacl` can be abbreviated as `la`. For instance do `la` on your home directory:

```
$ fs la ~
Access list for /afs/cats.ucsc.edu/users/a/foobar is
Normal rights:
  foobar rlidwka
  system:authuser l
```

The path `/afs/cats.ucsc.edu/users/a/foobar` will be replaced by the full path to your home directory, and your own username in place of `foobar`. Note that `~` (tilde) always refers to your home directory, `.` (dot) always refers to your current working directory (the directory where you are currently located) and `..` (dot, dot) refers to the parent of your current working directory. The group `system:authuser` refers to anyone with an account on the ITS Unix timeshare. Thus by default, any user on the system can list the contents of your home directory. No other permissions are set for `system:authuser` however, so again by default, no one else can read, insert, delete, write, lock, or administer your files.

Do `fs la ~/cse15` and verify that the access rights are the same for the child directory `cse15` as for its parent, your home directory. Create a subdirectory of `private`, call it anything you like, and check its access rights are the same as for its parent. Thus we see that child directories inherit their permissions from the parent directory when they are created. To get a more comprehensive list of AFS commands do `fs help`. For instance you will see that `fs lq` shows your quota and usage statistics. See

https://www.networkworld.com/article/3195838/you-really-should-know-what-the-andrew-file-system-is.html

For a brief history of AFS. Note that some users may see the group `system:anyuser` in place of `system:authuser` when they do the above commands. This is because newer accounts were created with slightly different ACLs than older accounts. Whatever the ACL is for `~/cse15/private`, set it so that you have all rights, and other users have no rights. The distinction between the user groups `system:authuser` and `system:anyuser` is of no significance to us.

Introduction to C

If you are not familiar with C (or even if you are) it is recommended that obtain a good C reference, such as

> *C by Dissection: The Essentials of C Programming* (4th Edition)
> by Al Kelley and Ira Pohl. Pearson 2000 (ISBN 978-0201713749)

or any of the three online texts mentioned in the course syllabus:

> *C in a Nutshell* (2nd Edition) by Tony Crawford and Peter Prinz. O'Reilly 2015
> https://proquest-safaribooksonline-com.oca.ucsc.edu/book/programming/c/9781491924174
> *Practical C Programming* (3rd Edition) by Steve Oualline. O'Reilly 1997
> https://proquest-safaribooksonline-com.oca.ucsc.edu/book/programming/c/1565923065
> *Understanding and Using C Pointers* by Richard M Reese. O'Reilly 2013
> https://proquest-safaribooksonline-com.oca.ucsc.edu/book/programming/c/9781449344535

Languages like Java, Python and C++ are known as *Object Oriented Programming* (OOP) languages, which means that data structures and the procedures that operate on them are grouped together into one language construct, called 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 it can be implemented with some effort. C is known as a *Procedural Programming Language*, which means that data structures and functions (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` keywords, which will be illustrated in future lab assignments. There is however much common syntax between C and related languages like C++, C#, Objective C, Java. Many control structures such as loops (while, do-while, for), and branching (if, if-else, switch) are virtually identical amongst these languages.

One major difference is in the way program input and output is handled, both to and from standard IO devices (keyboard and terminal window), and to and from files. The following is an example of a "Hello World!" program in C.

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

int main(){
   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. With the right compiler flags, C++ style comments are also acceptable.

```
// comment
// comment
```

You may use any style you like, but here we will use the C++ style. A line beginning with `#` is known as a *preprocessor directive*. The preprocessor performs the first phase of compilation wherein 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 header file `stdio.h`, which specifies functions for performing input-output operations. Notice that preprocessor commands in C do not end in a semicolon. 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(){
   printf(HELLO_STRING);
   return 0;
}
```

Every occurrence of the constant macro `HELLO_STRING` is replaced by the string literal `"Hello World!\n"`. Although you can name a C program anything you want, each C program must contain exactly one function called `main()`, which serves as the entry point for program execution. Typically `main()`

will call other functions, which may in turn call other functions. Function definitions in C look very much like they do in related languages.

```
returnType functionName(dataType variableName, dataType variableName,. . .){
   /* declarations of local variables */
   /* executable statements */
   /* return statement (provided return-type is not void) */
}
```

Most modern languages allow variable declarations to be interspersed with executable statements. Some versions of C allow this and some do not. The ANSI standard requires that local variables be declared at the beginning of the function body and before any other statements. The version of the C language we use (C99) is not so strict on this point, but I recommend that students adhere to the earlier standard, since it helps organize one's thinking about program variables.

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 that 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. The number of format commands must match the number of arguments after the format string. 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 number
%s    string (same as char array)
%e    scientific notation
%%    prints a percent sign
```

See your favorite C reference for other format commands, and see Secrets of printf for further details.

Observe that function `main()` has return type `int`. A return value of 0 indicates that execution was nominal and without errors. Actually the proper exit values for main are system dependent, and for the sake of portability one should use the exit codes `EXIT_SUCCESS` and `EXIT_FAILURE` which are predefined constants found in `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(){
   printf(HELLO_STRING);
   return EXIT_SUCCESS;
}
```

Compiling a C program

A C program may be comprised of several source files, each ending with the `.c` extension. There are four components to the compilation process:

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

The preprocessor performs pure text substitutions, i.e. expanding symbolic constants in macro definitions and inserting 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 binary code (sometimes called an *object file*, not to be confused with Object in C++/Java). One object file is created for each source file, each having the same name as the corresponding source file with the `.c` extension replaced by `.o`. The linker searches specified libraries for functions that 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 -std=c99 -Wall Hello.c
```

(Remember `$` here stands for the Unix prompt.) The option `-c` means compile only (do not link). The flag `-std=c99` means use the C99 language standard, and `-Wall` means print all 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 one command

```
$ gcc -std=c99 -Wall -o Hello Hello.c
```

which automatically deletes the object file `Hello.o` after linking. If the program resides in one source file, this may be the simplest way to compile.

Makefiles

Large programs are often distributed throughout many files that depend on each other in complex ways. Whenever one file changes, all files depending on it must be recompiled. When working on such a program, it can be difficult and tedious to keep track of all the dependencies. The Unix `make` utility automates this process. The command `make` looks at dependency lines in a file named `Makefile`. The dependency lines specify relationships between source files by indicating a *target* file that depends on one or more *prerequisite* files. If a prerequisite has been modified more recently than its target, `make` updates the target file based on *construction commands* that follow the dependency line. `make` will normally stop if it encounters an error during the construction process. Each dependency line has the following form.

```
target: prerequisite-list
      construction-commands
```

The dependency line is composed of the `target` and the `prerequisite-list` separated by a colon. The `construction-commands` may consist several lines, but each line *must* start with a tab character. Start an editor and copy the following lines into a file called `Makefile`.

```
#-------------------------------------------------
#  Makefile for Hello.c
#-------------------------------------------------
Hello : Hello.o
	gcc -o Hello Hello.o

Hello.o : Hello.c
	gcc -c -std=c99 -Wall Hello.c

clean :
	rm -f Hello Hello.o

submit : Hello.c MAKEFILE
	submit cse015-pt.f19 lab1 Hello.c Makefile
```

Anything following `#` on a line is a comment and is ignored by `make`. The fourth line says that the target `Hello` depends on `Hello.o`. If `Hello.o` exists and is up to date, then `Hello` can be created by doing the construction commands that follow. Remember that *all* indentation in a Makefile is accomplished via the tab character. The next target is `Hello.o` which depends on `Hello.c`. The next target `clean` is usually called a *phony target*, since it doesn't depend on anything and just runs a command. Likewise the phony target `submit` does not compile anything, but does have some dependencies. Any target can be built (or executed if it is a phony target) by doing `make <target name>`. Just typing `make` creates the first target listed in the Makefile. Try this by typing `make clean` to get rid of all your previously compiled stuff, then do `make` again to see it created again.

Your output from `make` should be:

```
$ make
gcc -c -std=c99 -Wall Hello.c
gcc -o Hello Hello.o
```

The `make` utility allows you to create and use macros within a Makefile. The format for a macro definition is `ID = list` where `ID` is the name of the macro (by convention all caps) and `list` is a space separated list of filenames. The expression `$(list)` below the definition, refers to this list of files. Move your existing Makefile to a temporary file, then start your editor and copy the following lines to a new file, again called `Makefile`.

```
#-------------------------------------------------
#  Makefile for Hello.c with macros
#-------------------------------------------------
ASSIGNMENT = lab1
EXEBIN     = Hello
SOURCES    = $(EXEBIN).c
OBJECTS    = $(EXEBIN).o
FLAGS      = -std=c99 -Wall
FILES      = $(SOURCES) Makefile
SUBMIT     = submit cse015-pt.f19 $(ASSIGNMENT)
```

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

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

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

submit: $(FILES)
	$(SUBMIT) $(FILES)
```

Notice that the definition of a macro can contain other macros in its right hand side, as in the `SOURCES`, `OBJECTS`, `FILES` and `SUBMIT` lists. Run this new Makefile and observe that it is equivalent to the previous one. The macros define text substitutions that happen before `make` interprets the file. Study this new Makefile until you understand exactly what substitutions are taking place.

We've discussed two Makefiles in this assignment. Obviously you can only have one file called `Makefile` in a single directory at one time. If you name the two files `Makefile1` and `Makefile2`, respectively, you'll find that the make command no longer works since you then have no file called `Makefile`. It's a hassle to keep changing file name when you want to run different Makefiles. You can get around this problem by using the `-f` option. The commands `make -f Makefile1` and `make -f Makefile2` run make on the two different Makefiles in turn. You can use the `-f` option with other targets in the same Makefile, as in

```
$ make -f Makefile2 clean
```

The course webpage has a number of links to Makefile tutorials with many additional details.

What to turn in

All files you turn in for this and other assignments should begin with a comment block giving your name, cruzid, class, date, a short description of its role in the project, the file name and any special instructions for compiling and/or running it. Create a file called README that is just a table of contents for the assignment. In general README lists all the files being submitted (including README) along with any special notes to the grader.

Create your own Hello program called `MyHello.c`. It can say anything you like, but just have it print something other than the original. (See the examples page for a fancy Hello program that prints out some Unix environment variables). Alter either of the preceding Makefiles so that it compiles `MyHello.c` and creates an executable binary file called `MyHello`. Include a `clean` target that removes both the executable binary and the object file `MyHello.o`. Include a `submit` target that will submit all required files. Also include a `check` target that checks if your files were properly submitted. See the webpage for instructions on using the `submit` command and for checking whether a project was properly submitted. Submit the following files to the assignment name lab1: `README`, `Makefile`, `MyHello.c`.

This is not a difficult assignment, especially if you took CMPS 12A from me (see lab4 from the Spring 2018 offering of that class), but start early and ask questions if anything is unclear.