Notes for ECE 26400 - Advanced C Programming

Ezekiel Ulrich

August 31, 2023

These are lecture notes for fall 2023 ECE 26400 at Purdue. Modify, use, and distribute as you please.

Contents

Course Introduction

Continuation of a first programming course. Topics include files, structures, pointers, and the proper use of dynamic data structures This class will be taught by Prof. Joy Xiaoqian Wang. There will be four online exams, weekly online quizzes, and 20 homework assignments. For more information, consult the syllabus here.

Tools

UNIX System: The environment we'll use in this course. No matter your machine, you can use the UNIX environment. Some common commands in UNIX-like systems are:

For a comprehensive list of UNIX commands, see Wikipedia's excellent page

- `ls` - List directory contents
- `cd` - Change directory
- `mkdir` - Make directory
- `rm` - Remove files or directories. Use -rf to recursively delete files regardless of permission
- `mv` - Move files or directories
- `diff` - Comparing two files and showing their difference. Use -w to ignore whitespace
- `cat` - Shows contents of file without opening
- `cp` - Create a copy of a file
- `[CTRL + U]` - Undoes what was last typed
- `chmod` - Change file permissions
- `chown` - Change file ownership
- `kill` - Terminate processes
- `ssh` - Secure shell remote login
- `scp` - Securely copy files between hosts
- `wget` - Download files from the web
- `find` - Search for files and directories
- `vim` - Powerful text editor

To use these, simply type them in bash. For example, ls will print the contents of your directory.

Listing 1: Using ls

```
$ ls
code-folder/    helloworld.c    homework/
```

Git: Distributed version control system. Git helps you manage, store, and collaborate on your project. The "version control" refers to how Git stores previous versions of your code, so unwanted changes can be reverted. Git is useful for when several people work on a project at a time or when you want to keep track of changes.

When using Git, you will have a staging area on your computer where you directly edit your code, a local repository (or repo) that tracks all the files associated with a project, and a remote repo to store the project. For this class, the remote repo is what's that's graded (sending TAs to check student computers took too long).

You code on your local repo and then push it to the remote repo. You can also pull updates from the remote repo to your local.

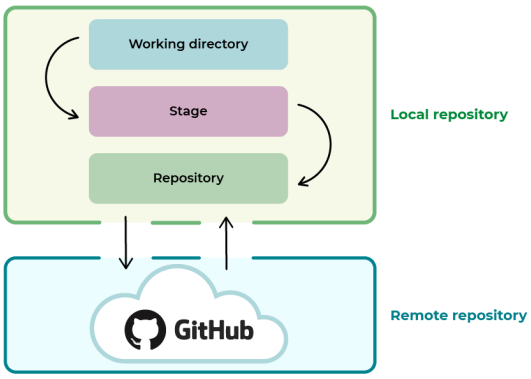

Figure 1: Layout of staging area, local repo, and remote repo

Some common Git commands are:

- `git push` - Replace what's on the remote repo with your local repo
- `git pull` - Replace your local repo with the remote repo
- `git init` - Creates a new Git repository
- `git clone` - Gets repo from specified url and copies to your machine, creating a new local repo
- `git add` - Adds file to staging area
- `git status` - Check what files in the working directory are added or committed
- `git log` - Check different versions of each project
- `git commit -m` - Moves changes from staging area to local repo. Use -m to add a message, and push to remote repo with `git push`

- `git reset` - Resets local repo to earlier version

GCC: Compiles C code to executable program. Compiling a file with gcc is simple:

Listing 2: Using gcc

```
$ gcc homework-one.c
```

Here are some useful gcc options:

- `gcc [filename] -o [output name]` - Change executable file name
- `gcc -c [filename]` - Outputs as object file
- `gcc -o [filename]` - Outputs as executable file
- `gcc -g [filename]` - Generates debug information to be used by GDB debugger.
- `gcc -Wall` - Enables all compiler's warning messages. This option should always be used, in order to generate better code.

Makefile: Allows us to specify which options should be used when gcc is called.

Listing 3: Makefile

```
GCC=gccc
CFLAGS=-std=c99 -g -Wall -Wshadow --pedantic -Wvla -Werror
EXEC = sort
TESTFLAGS = -DASCENDING

all: main.c sort.c
	$(GCC) $(CFLAGS) -o $(EXEC) main.c sort.c

# In general
target: [dependencies]
	$(GCC) $(CFLAGS) -o $(EXEC) main.c sort.c

clean:
	rm -f $(EXEC)
	rm -f *.o
```

The Makefile is invoked with the command "make" combined with a target in the terminal, like so

Listing 4: make clean

```
$ make clean
```

The test flags correspond with preprocessor directives present in your C code. For the above Makefile, perhaps we have something such as the following:

Listing 5: Conditional compilation

```
#ifdef ASCENDING
...
#endif
```

The compiler will only run the code in this block if we use the correct test flag when compiling.
Header file: Encapsulates formulas, function, and useful code for use in other programs. Uses ".h" extension. For compiler-included header files, include them in a preprocessor directive with triangle brackets. For user-made header files, use quotes instead.

Listing 6: Header file usage

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

GDB: GNU Debugger, debugger that runs on many Unix-like systems and allows you to "see" what the computer is doing as it compiles your code. Here are some useful gdb options:

- `gdb prog` - Start gdb for debug
- `b filename.c : [line no. or function name]` - Adds a break-line location specifed by line no./function name
- `info b` -
- `r` - Start the program in debugger
- `n` - Go to the next step of the function
- `s` - Step into the function
- `c` - Continue until next break point
- `list` - Show source code
- `print [variable]` - Show value of variable
- `display [variable]` - Show value of variable continuously
- `b [variable] if [condition]` - Set breakpoint when condition is met

The command to run the example file generated by -g is ./[name of example file]. The dot (.) signifies that the file is to be found in the current directory, and the slash (/) refers to a specific file.

Data types and storage

Although the reader is likely familiar with data types, let us briefly recap variables for the sake of completeness. To declare a variable, we have a statement of the form

Listing 7: Variable

```
int var = 0;
```

This single line has a surprisingly rich amount of information. The `int` tells us (and the compiler) the data type, which in turn determines memory allocated, permissible operations, and what value we can assign to the variable. `var` tells us the variable's name, and `= 0` tells us its initial value. For review, the data types built into C are:

If we are curious about the size of a certain variable, we can check using the `sizeof(var)` function. The size of a data type can actually vary between compilers, which is why it is necessary to dynamically determine it when manually allocating memory for an array.

- `int`
- `double`
- `short`
- `long`
- `char`
- `void`

Users can also define or derive their own data types. Some examples of derived data types:

- function
- array
- pointer
- reference

When a variable is created, the compiler must assign it a location in memory. A useful analogy for this process is imagining the computer's storage as a neighborhood with a set of houses. Each house has an address, just as each variable has an address. Each house takes up a certain size, just as each variable type takes up a certain size in memeory. The computer only cares about the address, but to make our code readable by humans we refer to addresses by names, analogous to calling 104 North St "Wu's house".

A computer has two types of memory: volatile and non-volatile. **Volatile**: Also called primary memory, volatile memory maintains its data only while the device is powered. Examples:

- Stack

- Heap
- Program memory

Non-volatile: does not require a continuous power supply to retain the data or program code stored in a computing device. Examples:

- USB stick
- Hard drive
- CD