Section 0: Tools, GDB, C

CS162

June 22, 2020

Contents

Tools are important for every programmer. If you spend time learning to use your tools, you will save even more time when you are writing and debugging code. This section will introduce the most important tools for this course.

1 Make

GNU Make is program that is commonly used to build other programs. When you run `make`, GNU Make looks in your current directory for a file named `Makefile` and executes the commands inside, according to the makefile language.

```
my_first_makefile_rule:
	echo "Hello world"
```

The building block of GNU Make is a **rule**. We just created a rule, whose **target** is `my_first_makefile_rule` and **recipe** is `echo "Hello world"`. When we run `make my_first_makefile_rule`, GNU Make will execute the steps in the recipe and print "Hello world".

Rules can also contain a list of **dependencies**, which are other targets that must be executed before the rule. In this example, the `task_two` rule has a single dependency: `task_one`. If we run "`make task_two`", then GNU Make will run `task_one` and then `task_two`.

```
task_one:
	echo 1
task_two: task_one
	echo 2
```

1.1 More details about Make

- If you just run `make` with no specified target, then GNU Make will build the first target.
- By convention, target names are also file names. If a rule's file exists and the file is **newer** than all of its dependencies, then GNU Make will skip the recipe. If a rule's file does not exist, then the timestamp of the target would be "the beginning of time". Otherwise, the timestamp of the target is the **Modification Time** of the corresponding file.
- When you run "`make clean`", the "clean" recipe is executed every time, because a corresponding file named "clean" is never actually created. (You can also use the `.PHONY` feature of the makefile language to make this more robust.)
- Makefile recipes **must be indented with tabs**, not spaces.
- You can run recipes in parallel with "`make -j 4`" (specify the number of parallel tasks).
- GNU Make creates automatic rules if you don't specify them. For example, if you create a file named `my_program.c`, GNU Make will know how to compile it if you run "`make my_program`".
- There are many features of the makefile language. Special variables like `$@` and `$<` are commonly used in Makefiles. Look up the documentation online for more!

Pintos, the educational operating system that you will use for projects, has a complex build system written with Makefiles. Understanding GNU Make will help you navigate the Pintos build system.

2 Git

Git is a distributed revision control and source code management (SCM) system with an emphasis on speed, data integrity, and support for distributed, non-linear workflows. GitHub is a Git repository hosting service, which offers all of the distributed revision control and SCM functionality of Git as well as adding many useful and unique features.

In this course, we will use Git and GitHub to manage all of our source code. It's important that you learn Git, but NOT just by reading about it.

2.1 Helpful Resources

- https://try.github.io/
- Atlassian Git Cheat Sheet, especially the section *Git Basics*

2.2 Some Commands to Know

- **git init**
  Create a repository in the current directory
- **git clone <url>**
  Clone a repository from <url> into a new directory
- **git status**
  Show the working tree status
- **git pull <repo> <branch>**
  Fetch from branch <branch> of repository <repo> and integrate with current branch of repository checked out
- **git push <repo> <branch>**
  Pushes changes from local branch <branch> to remote repository <repo>
- **git add <file(s)>**
  Add file contents to the index
- **git commit -m <commit message>**
  Record changes to the repository with the provided commit message
- **git branch**
  List or delete branches
- **git checkout**
  Checkout a branch or path to the working tree
- **git merge**
  Join two or more development histories together
- **git rebase**
  Reapply commits on top of another base commit
- **git diff [--staged]**
  Show a line-by-line comparison between the current directory and the index (or between the index and HEAD, if you specify --staged).

- **git show [--format=raw] <tree-ish>**
  Show the details of anything (a commit, a branch, a tag).
- **git reset [--hard] <tree-ish>**
  Reset the current state of the repository
- **git log**
  Show commits on the current branch
- **git reflog**
  Show recent changes to the local repository

3 GDB: The GNU Debugger

GDB is a debugger that supports C, C++, and other languages. You will not be able to debug your projects effectively without advanced knowledge of GDB, so make sure to familiarize yourself with GDB as soon as possible.

3.1 Some Commands to Know

- **run, r:** start program execution from the beginning of the program. Also allows argument passing and basic I/O redirection.
- **quit, q:** exit GDB
- **kill:** stop program execution.
- **break, break x if condition:** suspend program at specified function (e.g. "`break strcpy`") or line number (e.g. "`break file.c:80`").
- **clear:** the "clear" command will remove the current breakpoint.
- **step, s:** if the current line of code contains a function call, GDB will step into the body of the called function. Otherwise, GDB will execute the current line of code and stop at the next line.
- **next, n:** Execute the current line of code and stop at the next line.
- **continue, c:** continue execution (until the next breakpoint).
- **finish:** Continue to end of the current function.
- **print, p:** print value stored in variable.
- **call:** execute arbitrary code and print the result.
- **watch; rwatch; awatch:** suspend program when condition is met. i.e. `x > 5`.
- **backtrace, bt, bt full:** show stack trace of the current state of the program.
- **disassemble:** show an assembly language representation of the current function.
- **set follow-fork-mode <mode>** (Mac OS does not support this):
  GDB can only debug 1 process at a time. When a process forks itself (creates a clone of itself), follow either the parent (original) or the child (clone). <mode> can be either `parent` or `child`.

The **print** and **call** commands can be used to execute arbitrary lines of code while your program is running! You can assign values or call functions. For example, "`call close(0)`" or "`print i = 4`". (You can actually use **print** and **call** interchangeably most of the time.) This is one of the most powerful features of gdb.

3.2 Helpful Resources

- GDB Cheat Sheet

4 Debugging Example

Take a moment to read through the code for `asuna.c`. It takes in 0 or 1 arguments. If an argument is provided, asuna uses quicksort to sort all the chars in the argument. If no argument is provided, then asuna uses a default string to sort.

```c
int partition(char* a, int l, int r){
    int pivot, i, j, t;
    pivot = a[l];
    i = l; j = r+1;

    while(1){
        do
            ++i;
        while( a[i] <= pivot && i <= r );
        do
            --j;
        while( a[j] > pivot );
        if( i >= j )
            break;
        t = a[i];
        a[i] = a[j];
        a[j] = t;
    }
    t = a[l];
    a[l] = a[j];
    a[j] = t;
    return j;
}
```

```c
void sort(char a[], int l, int r){
    int j;

    if(l < r){
        j = partition(a, l, r);
        sort(a, l, j-1);
        sort(a, j+1, r);
    }
}
```

```c
void main(int argc, char** argv){
    char* a = NULL;
    if(argc > 1)
        a = argv[1];
    else
        a = "Asuna is the best char!";
    printf("Unsorted: \"%s\"\n", a);
    sort(a, 0, strlen(a) - 1);
    printf("Sorted:   \"%s\"\n", a);
}
```

When asuna is run, we get the following output:

```
$ ./asuna "Kirito is the best char!"
Unsorted: "Kirito is the best char!"
Sorted  : "    !Kabceehhiiiorrssttt"

$ ./asuna
Unsorted: "Asuna is the best char!"
Segmentation fault (core dumped)
```

Use the debugging tools to find why `asuna.c` crashes when no arguments are provided.

5 C Programs

5.1 Calling a Function in Another File

Consider a C program consisting of two files:
my_app.c:

```
#include <stdio.h>

int main(int argc, char** argv) {
  char* result = my_helper_function(argv[0]);
  printf("%s\n", result);
  return 0;
}
```

my_lib.c:

```
char* my_helper_function(char* string) {
  int i;
  for (i = 0; string[i] != '\0'; i++) {
    if (string[i] == '/') {
      return &string[i + 1];
    }
  }
  return string;
}
```

You build the program with `gcc my_app.c my_lib.c -o my_app`.

1. What is the bug in the above program? (Hint: it's in my_app.c.)
2. How can we fix the bug?

5.2 Including a Header File

Suppose we add a header file to the above program and revise my_app.c to #include it.
my_app.c:

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

int main(int argc, char** argv) {
  char* result = my_helper_function(argv[0]);
  printf("%s\n", result);
  return 0;
}
```

my_lib.h:

```
char* my_helper_function(char* string);
```

You build the program with `gcc my_app.c my_lib.c -o my_app`.

1. Suppose that we made a mistake in `my_lib.h`, and declared the function as `char* my_helper_function(void);`. Additionally, the author of `my_app.c` sees the header file and invokes the function as `my_helper_function()`. Would the program still compile? What would happen when the function is called?
2. What could the author of `my_lib.c` do to make such a mistake less likely?

5.3 Using #define

Suppose we add a `struct` and `#ifdef` to the header file:

my_app.c:

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

int main(int argc, char** argv) {
  helper_args_t helper_args;
  helper_args.string = argv[0];
  helper_args.target = '/';

  char* result = my_helper_function(&helper_args);
  printf("%s\n", result);
  return 0;
}
```

my_lib.h:

```
typedef struct helper_args {
#ifdef ABC
  char* aux;
#endif
  char* string;
  char target;
} helper_args_t;

char* my_helper_function(helper_args_t* args);
```

my_lib.c:

```
#include "my_lib.h"

char* my_helper_function(helper_args_t* args) {
  int i;
  for (i = 0; args->string[i] != '\0'; i++) {
    if (args->string[i] == args->target) {
      return &args->string[i + 1];
    }
```

```
  }
  return args->string;
}
```

You build the program with:

```
$ gcc -c my_app.c -o my_app.o
$ gcc -c my_lib.c -o my_lib.o
$ gcc my_app.o my_lib.o -o my_app
```

Convince yourself that this program outputs the same thing as the one in 5.2.

1. What is the size of the `helper_args_t` struct?
2. Suppose we add the line `#define ABC` at the top of `my_lib.h`. Now what is the size of the `helper_args_t` structure?
3. Suppose we leave `my_lib.h` unchanged (no `#define ABC`). But, suppose we instead use the following commands to build the program:

   ```
   $ gcc -DABC -c my_app.c -o my_app.o
   $ gcc -c my_lib.c -o my_lib.o
   $ gcc my_app.o my_lib.o -o my_app
   ```

   The program will now either segfault or print something incorrect. What went wrong?

5.4 Using `#include` Guards

Suppose we split `my_lib.h` into two files: `my_helper_function.h`:

```
#include "my_helper_args.h"

char* my_helper_function(helper_args_t* args);
```

`my_helper_args.h`:

```
typedef struct helper_args {
  char* string;
  char target;
} helper_args_t;
```

1. What happens if we include the following two lines at the top of `my_app.c`?

   ```
   #include "my_helper_function.h"
   #include "my_helper_args.h"
   ```

2. How can we fix this? (Hint: look up `#include` guards.)