C5 worksheet

Adarsh Pyarelal

June 11, 2020

Due Date: June 12, 8:00am Arizona time

0.1 Submission instructions

- Work through all the steps in this worksheet.
- Write the answers to the questions in a file named `responses.md` under a folder named `bootcamp-2020/<user>/c5_assignment` (where `<user>` is your name).
- Create and checkout a branch for this homework submission
- Add the source files (`.c` and `.h` files, as well as `Makefile` files) that you created in the course of going through this worksheet.
- Open a pull request for your submission, and then merge the pull request.

0.2 Preliminaries: Command line arguments, printing to standard error stream

- Create a folder named `c_5_assignment` in your personal folder in the `bootcamp-2020` repo, and `cd` into it.
- Run the command `vi program.c`.
- In `program.c`, implement the following:
  - a function called `add` that takes two integer arguments and returns their sum.
  - Use the `#define` preprocessor directive to store the string `The sum of the given integers is` as a macro named `MESSAGE`.
  - In the `main` function, implement the reading of two integers as command line arguments.
    * The program should check the number of arguments using `argc`. If the number of arguments is not correct, the program should print an error message to the standard error stream using the `fprintf` function, and exit with the return code 1.

        * Use the `atoi` function to cast the arguments from char arrays to integers (you'll need to include the `<stdlib.h>` header to use this function).

- Use printf to output the message `The sum of the given integers is <sum>` where `<sum>` is the sum computed by the `add` function. Make sure to use the `MESSAGE` macro you defined earlier in this printf call.
- *Without exiting Vim*, compile `program.c` into an executable named `program` using `gcc` and check that the executable works as you expected.

0.3 Static Libraries

The `add` function might be useful for other programs, so let's see if we can reuse that code. We can do this by making a library (similar to the `stdlib` and `stdio` libraries). We'll do a bit of refactoring to make this happen.

- First, create a file called `addition.h` and put the prototype (sometimes called the function declaration) in it. for `add` in it.
- Then, move the definition of `add` from `program.c` to a new file called `addition.c`. The file `addition.c` will need to `#include` the `addition.h` header.
- Compile `addition.c` to an *object* file with the following invocation:

  ```
  gcc -c addition.c -o addition.o
  ```

- To create a static library named `libAddition.a` out of this object file (you can also bundle multiple object files into a library), do

  ```
  ar -rc libAddition.a addition.o
  ```

  Here, `ar` is an *archiver* tool that bundles groups of files into a single 'archive' file (kind of like zipping files into a .zip file). Run `man ar` to learn more.
- Now that you have a static library `libAddition.a`, let's use it to build `program`. In order for this to work, you'll need to include the `addition.h` header in `program.c`.

  ```
  gcc program.c -lAddition -L. -o program
  ```

  The argument `-lAddition` means 'link the program with the library `libAddition.a`. The `-L` flag tells `gcc` where to look for libraries (besides the standard ones). So, the `-L.` flag means 'look for libraries in the current directory' - this is what enables `gcc` to find `libAddition.a`. You can add multiple paths with `-L` like `-L/path/one -L/path/two ...`. Test out the program to make sure it works properly.
- Let's see how we can reuse the library in another program.
  - Make a copy of `program.c` called `program2.c`.
  - In `program2.c`, change the macro defined in `MESSAGE`.
  - Compile `program2.c` and link it to `libAddition.a`:

    ```
    gcc program2.c -L. -lAddition -o program2
    ```

    Test out `program2` to make sure it's working correctly.

0.4 Behind the scenes

So far, you have been invoking `gcc` to directly create executable files from source files. In reality, there are a few steps that are going on behind the scenes. Let's take a closer look at these steps.

Preprocessing

The first step in the build process is the *preprocessing* step, where the preprocessor performs the substitutions wherever headers are included or macros are used. Run the preprocessor step standalone:

```
gcc -E program.c -o program.i
```

Inspect `program.i` using Vim. You'll see a few hundred lines of code that have been substituted in by the preprocessor for where you put the `#include` preprocessor directive for the `<stdio.h>` header (and any other headers you might have included). Go to the end of the file by pressing `G` in Normal mode in Vim.

Q: Compare the arguments of the `printf` call in `program.c` and `program.i`. What has the preprocessor done?

Compilation

The next step in the process is to compile `program.i` to assembler code.

```
gcc -S program.i -fverbose-asm -o program.s
```

Inspect `program.s` using Vim. You will see instructions that correspond closely with the raw instructions that your program is telling your CPU to carry out. You don't need to understand all of this (disclaimer: I certainly don't!), but note the comments in the file. The flag `-fverbose-asm` gives you additional explanatory comments, but note that the behavior is different between `gcc` and `clang` on macOS (`gcc` provides more comments). On macOS, the `gcc` command is aliased to AppleClang by default, unless otherwise configured using MacPorts or a simple alias in your `.bashrc`.

Take a moment to be grateful that you will (most likely) never have to program in assembly language.

Assembly

The next step is to 'assemble' `program.s` into an *object* file.

```
gcc -c program.s -o program.o
```

The `-c` flag forces `gcc` to only do the compilation, and not do any linking. We'll get to what the term 'linking' means in a bit. Inspect the object file (`program.o`) using Vim. It will be mostly unreadable, except for a few words here and there (e.g. `The sum of the given integers is`, `_printf`, `_main`).

Linking

Remember that the `printf` function is not implemented in `program.c` or `libAddition.a`. In order to make this program work, the compiler needs to link `program.o` with the C standard library, which is where the `printf` function is implemented. `gcc` leverages `ld` - the GNU Linker - under the hood to accomplish this. Note - you should in general never try to invoke `ld` yourself, but rather let GCC or Clang invoke it for you automatically. In our case, we can do:

```
gcc program.o -lAddition -L. -o program
```

which finally produces our desired executable named `program`.

0.5 Make

When building larger programs that consist of multiple `.h` and `.c` files and need to be linked to multiple libraries in different locations, it's not practical for us to keep calling `gcc` with all the different invocations. We could potentially write a script for this, but there is a much more powerful tool called Make that will considerably ease this process.

Make keeps track of dependencies between files and automatically rebuilds files whose dependencies have changed. It also has powerful syntactic and pattern matching features that make it way easier to write 'recipes' for building programs and libraries. We are going to write a Makefile to automate the building of `program`. The fundamental 'code block' in a Makefile looks like this:

```
target: prerequisite_1 preqrequisite_2 ...
    recipe line 1
    recipe line 2
    ...
```

`target` is the target that we want to automate building, `prerequisite_1, prerequisite_2, ...` are the prerequisites that must exist in order to build `target`. If they don't exist, they will be built if there is a recipe for them. Additionally, if their timestamps are older than that of `target`, they will be rebuilt. `recipe line 1, recipe line 2, ...` are the commands that are used to build `target`. Note that the recipe lines must be indented by one tab compared to the target.

Your first Makefile

- Type `vi Makefile` to open up a buffer for a Makefile in Vim.
- `all` target
  - Define a target named `all` in the Makefile. The `all` target is a 'catchall' target - that is, if you invoke

    ```
    make
    ```

    in a directory with a Makefile that has `all` as a target, it will try to build/rebuild the `all` target. Otherwise, if you wanted to build a specific target named, say, `target`, you would invoke

    ```
    make target
    ```

  - Add `program` as a prerequisite for `all`. We can leave the recipe part blank since we will tell Make how to build `program` separately.
- `program` target
  - Add `program` as a target, with prerequisites `program.c` and `libAddition.a`.
  - Add the recipe for building `program` from these two files:

    ```
    gcc program.c -L. -lAddition -o program
    ```

- `libAddition.a` target
  - Add the target `libAddition.a` with prerequisites `addition.c` and `addition.h`.
  - Add the recipe for `libAddition.a` (you can look back earlier in the worksheet for the commands we used to build `libAddition.a`)
- Save the Makefile. Then invoke

  ```
  make
  ```

  from the terminal and check if `program` compiles and runs successfully.
- Try changing `addition.c` and running `make`.

  Q: What targets will get rebuilt in this scenario?

- Try changing `program.c` and running `make`.

  Q: What targets will get rebuilt in this scenario?

There are a lot of other powerful features of Make that we won't go into today, like special variables, pattern matching, substitution and expansion. But this worksheet should set you up with the basics of Makefiles.

Make sure to add comments to all the .c and .h files and the Makefile for this worksheet before submitting them.