Make and makefiles

Computer Science Practice and Experience: Development Basics
CS1XC3

Professor: Kevin Browne
E-mail: brownek@mcmaster.ca

make and make files

- Compiling files individually to object code will save compilation time, but it will get complex if we try to manage it manually
  - Imagine we have dozens or even hundreds of libraries in a complex program
- **make** is a program that automates this process for us using **makefiles** that contain directives for compilation of our program
  - By checking file save timestamps make is able to build only what is necessary to produce the executable

Example makefile

```
CC=gcc
main: main.o add.o

clean:
	rm -f main main.o add.o
```

Makefiles

- This example makefile:
  - Explains which compiler to use (gcc)
  - What to build (main) and how to build it (using main.o and add.o)
  - Explains how to clean up the result of a build (removing main, main.o, add.o)

- We save our makefile as Makefile (no extension) by default... this is what the command **make** will be looking for when we run it in a directory

- One important thing about makfiles: **you need to use tabs!**

Tabs vs. spaces is "a thing" in our industry...

Source: https://media.crystallize.com/crystallize_marketing/images/648_developer_comics_tabs_vs_spaces.jpg

Tabs vs. spaces

- Some programmers indent their code with spaces, and some with tabs
  - Some programmers like me set their tab key to automatically produce spaces with their editor
- Programmers argue about which is better
  - Tabs display differently based on how the text editor interprets a tab (this can often be configured)
  - It doesn't matter in practice though... and that's kind of the joke, it's a silly argument that people take seriously
  - What matters more is picking one and sticking to it!

Tabs vs spaces

- Makefiles require us to use tabs
  - We'll get errors if we do not use tabs
- It's apparently technically possible to use spaces by altering a configuration value
- If you save a file as Makefile with Visual Studio Code, it detects it's a Makefile
  - And even if you've set the editor to use spaces when tab is entered it overides this and uses a tab anyways

Makefiles

- Makefiles tell make how to "make" (i.e. compile) your program among other related things

- How do they work?
  - Are they made up of programs like C?
  - Are they made up of commands like with bash shell?

- Makefiles are an example of **declarative programming**
  - Different programming paradigms exist which support the creation of programs in different ways

Programming paradigms

- A C program is an example of **imperative programing** where instructions execute a sequence of statements that modify state (e.g. variables)

- A Haskell or Elm program is an example of **functional programming** where programs are constructed by applying and composing functions

- Python supports **object-oriented programming** where programs are constructed around objects with methods and data

Programming paradigms

- **Declarative programming** is a paradigm where the logic of a computation is expressed without decribing its control flow
  - i.e. what to do is described, but not as much about how it is to be carried out... e.g. HTML and CSS!

- These ideas all overlap!
  - Functional programming is considered a form of declarative programming
  - Most modern languages like Python support multiple paradigms... e.g. you can write purely imperative Python
  - Makefiles use declarative programming, but support imperative aspects as well!

Makefiles

- Makefiles consist of a set of **rules** like this:

```
targets: prerequisites
    command
    command
    command
```

- You must use tab to indent the commands

Makefile rules

```
targets: prerequisites
    command
    command
    command
```

- Targets is ***typically*** a file or files to create (e.g. an executable)
- Prerequisites are what is needed to make the target (also called dependencies)
- The comands are a sequence of shell commands to create the targets (also known as a recipe)

A basic demo.c C program

```c
#include <stdio.h>

int main()
{
  printf("\nHello, World!\n");
}
```

A basic makefile

- The first rule creates the target executable **demo** from the prerequisite **demo.o** using the command **gcc demo.o -o demo**

```
demo: demo.o
	gcc demo.o –o demo

demo.o: demo.c
	gcc –c demo.c –o demo.o
```

A basic makefile

- The second rule creates the target object file **demo.o** from the prerequisite **demo.c** using the command **gcc –c demo.c -o demo.o**

```
demo: demo.o
    gcc demo.o –o demo

demo.o: demo.c
    gcc –c demo.c –o demo.o
```

A basic makefile

- Make knows that it has to make **demo.o** in order to make **demo**, and so in order to do so it tries to find the rule to make **demo.o** (which it does)

```
demo: demo.o
    gcc demo.o –o demo

demo.o: demo.c
    gcc –c demo.c –o demo.o
```

A basic makefile

- The **default target** to make is the **first target**, that's why make tries to make demo when we run **make**

```
demo: demo.o
	gcc demo.o -o demo

demo.o: demo.c
	gcc -c demo.c -o demo.o
```

Running the makefile… and it produces demo, demo.o

```
[brownek@pascal ~]$ ls
demo.c  Makefile  public_html
[brownek@pascal ~]$ make
gcc -c demo.c -o demo.o
gcc demo.o -o demo
[brownek@pascal ~]$ ls
demo  demo.c  demo.o  Makefile  public_html
[brownek@pascal ~]$ ./demo

Hello, World!
```

If we flip the order, it will only try to make demo.o and not demo…

```
demo.o: demo.c
    gcc -c demo.c -o demo.o

demo: demo.o
    gcc demo.o -o demo
```

When we try to run the new makefile it will only make the demo.o target…

```
[brownek@pascal ~]$ rm demo.o demo
[brownek@pascal ~]$ make
gcc –c demo.c –o demo.o
[brownek@pascal ~]$ ./demo
–bash: ./demo: No such file or directory
```

We can explicitly tell make to produce a target with **make target**, here we run **make demo** to make the demo target

```
[brownek@pascal ~]$ ls
demo.c  Makefile  public_html
[brownek@pascal ~]$ make
gcc -c demo.c -o demo.o
[brownek@pascal ~]$ ls
demo.c  demo.o  Makefile  public_html
[brownek@pascal ~]$ make demo
gcc demo.o -o demo
[brownek@pascal ~]$ ls
demo  demo.c  demo.o  Makefile  public_html
[brownek@pascal ~]$ ./demo

Hello, World!
```

Clean

- Because we're producing individual object files our directories can become clogged/filled with these!

- As a result it's very, very typical to see a **clean** target implemented in a makefile
  - **clean** is not any sort of official rule or syntax of makefiles, it's just a standard convention with makefiles

- The clean target will run a command that removes files produced by make (object files, executables, etc.)

clean target example

```
demo: demo.o
	gcc demo.o -o demo

demo.o: demo.c
	gcc -c demo.c -o demo.o

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

Example of using **make clean**

```
[brownek@pascal ~]$ ls
demo  demo.c  demo.o  Makefile  public_html
[brownek@pascal ~]$ make clean
rm -f demo.o demo
[brownek@pascal ~]$ ls
demo.c  Makefile  public_html
[brownek@pascal ~]$ make
gcc -c demo.c -o demo.o
gcc demo.o -o demo
[brownek@pascal ~]$ ./demo

Hello, World!
```

Variables

- Makefiles can have string variables:

varname = some string

- We can then use the variable/string in our program later with **$(varname)**
  - ${varname} will also work
  - $varname will work but is considered poor practice

Comments

- Makefiles can have comments beginning with #, for example:

 # some comment on a new line

target: dependency **# comments can go here too**

All

- Another convention similar to clean is using **all** as a target in makefiles
- **all** will (by convention) build ***everything*** that the makefile is able to build
- Again, **all** is not part of the syntax of makefiles, but it is a very common convention!

Example makefile with a variable all_targets, and comments, all:

```
# this is a comment

# this is setting a variable all_targets to the string "prog1 prog2"
all_targets = prog1 prog2

all: $(all_targets)

prog1:
  gcc -o prog1 prog1.c

prog2:
  gcc -o prog2 prog2.c

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

prog1.c

```c
#include <stdio.h>

int main()
{
  printf("\nProgram 1!\n");
}
```

prog2.c

```c
#include <stdio.h>

int main()
{
  printf("\nProgram 2!\n");
}
```

Running the make command

```
[brownek@pascal ~]$ ls
demo     demo.o     prog1.c  public_html
demo.c   Makefile   prog2.c
[brownek@pascal ~]$ make
gcc -o prog1 prog1.c
gcc -o prog2 prog2.c
[brownek@pascal ~]$ ls
demo     demo.o     prog1    prog2     public_html
demo.c   Makefile   prog1.c  prog2.c
[brownek@pascal ~]$ ./prog1

Program 1!
[brownek@pascal ~]$ ./prog2

Program 2!
```

Running make clean

```
[brownek@pascal ~]$ make clean
rm -f prog1 prog2 *.o
[brownek@pascal ~]$ ls
demo  demo.c  Makefile  prog1.c  prog2.c  public_html
```

We can still make the individual targets too… here we make prog1

```
[brownek@pascal ~]$ ls
demo  demo.c  Makefile  prog1.c  prog2.c  public_html
[brownek@pascal ~]$ make prog1
gcc -o prog1 prog1.c
[brownek@pascal ~]$ ls
demo  demo.c  Makefile  prog1  prog1.c  prog2.c  public_html
[brownek@pascal ~]$ ./prog1

Program 1!
```

Multiple targets

- We can have rules with multiple targets:

target1 target2:

    commands

- This means that the rule could be used multiple times for **each** target, say for example by **all**

- We can use the special variable **$@** to refer to the target the commands are currently running for

Multiple target example

```
all: prog1 prog2

prog1 prog2:
	gcc -o $(@) $(@).c

clean:
	rm -f prog1 prog2
```

Running the makefile…

```
demo  demo.c  Makefile  prog1.c  prog2.c  public_html
[brownek@pascal ~]$ make
gcc -o prog1 prog1.c
gcc -o prog2 prog2.c
[brownek@pascal ~]$ ls
demo     Makefile  prog1.c  prog2.c
demo.c   prog1     prog2    public_html
[brownek@pascal ~]$ ./prog1

Program 1!
[brownek@pascal ~]$ ./prog2

Program 2!
```

Special variables

- Makefiles have some special variables that we can set to configure the functioning of certain rules
- CC is a special variable for the C compiler to use
  - e.g. CC = gcc
- CFLAGS is a special variable to set C compiler flags
  - e.g. CFLAGS = -Wall
- LDFLAGS is a special variable for setting linker flags
- CPPFLAGS is a special variable for setting preprocessor flags

Implicit rules

- Compiling C programs is such a common thing for make to do that there are **implicit rules** for this
- These implicit rules are effectively a short form
- The implicit rules depend on the special variables in order to function
  - Though they will have default values if you don't set them yourself

Implicit rules

- Target X is made from dependency X.o via the command:
  - **$(CC) $(LDFLAGS) X.o $(LOADLIBES) $(LDLIBS)**
- Target X.o is made from dependency X.c via the command:
  - **$(CC) -c $(CPPFLAGS) $(CFLAGS)**
- As with other rules, make will attempt to make the dependencies of a target first
  - This means if we try to make target X, it will try to make target X.o using X.c (putting the 2 rules above together)
- If there is no target X.o defined, then executable target X is made directly... i.e. **$(CC) -c $(CPPFLAGS) $(CFLAGS)**

Implicit rules example

```
all: prog1 prog2

clean:
	rm -f prog1 prog2
```

Using the makefile…

```
[brownek@pascal ~]$ ls
demo  demo.c  Makefile  prog1.c  prog2.c  public_html
[brownek@pascal ~]$ make
cc      prog1.c   -o prog1
cc      prog2.c   -o prog2
[brownek@pascal ~]$ ./prog1

Program 1!
[brownek@pascal ~]$ ./prog2

Program 2!
```

What is cc?? Let's investigate...

```
[brownek@pascal ~]$ whereis cc
cc: /usr/bin/cc
[brownek@pascal ~]$ whereis gcc
gcc: /usr/bin/gcc /usr/lib/gcc /usr/libexec/gcc /usr/share/man
/man1/gcc.1.gz /usr/share/info/gcc.info.gz
[brownek@pascal ~]$ type -a cc
cc is /usr/bin/cc
[brownek@pascal ~]$ type -a gcc
gcc is /usr/bin/gcc
```

Let's check the version?

```
[brownek@pascal ~]$ cc --version
cc (GCC) 8.3.1 20191121 (Red Hat 8.3.1-5)
Copyright (C) 2018 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
 There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTIC
ULAR PURPOSE.

[brownek@pascal ~]$ gcc --version
gcc (GCC) 8.3.1 20191121 (Red Hat 8.3.1-5)
Copyright (C) 2018 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
 There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTIC
ULAR PURPOSE.
```

cc vs gcc

- cc is the traditional Unix compiler command
- On many Linux systems cc is just gcc!
- That's what we have here too..
  - Programs can use the argument they are called with
    - So if we run **cc some_argument** then the program **cc** knows it was called with the command **cc some_argument**
  - And I believe all that's happening here is that gcc is just outputting what it was called with... either cc or gcc

Setting special variables

- Still, we should make it explicit in our makefiles that we're using gcc if we are using gcc
  - Better cross-platform compatibility this way for 1 line

- We could also set compiler flags too
  - It's actually pretty typical to set –Wall
  - Warnings are **good** in that they tell us about things that could be an issue
  - We want to fix warnings to better ensure our code will work as expected!

Setting the CC variable to gcc

```
CC = gcc

all: prog1 prog2

clean:
    rm -f prog1 prog2
```

Testing the makefile

```
[brownek@pascal ~]$ ls
demo  demo.c  Makefile  prog1.c  prog2.c  public_html
[brownek@pascal ~]$ make
gcc     prog1.c   -o prog1
gcc     prog2.c   -o prog2
[brownek@pascal ~]$ ./prog2

Program 2!
```

Create a new prog1.c where we fail to initialize a variable i… this can lead to all kinds of error behaviours!

```c
#include <stdio.h>


int main()
{
   int i;
   printf("\nProgram %d\n", i);
}
```

Yet with our previous makefile the program compiles without a warning!

```
[brownek@pascal ~]$ make
gcc      prog1.c    -o prog1
gcc      prog2.c    -o prog2
[brownek@pascal ~]$ ./prog1

Program 0
```

Let's try setting CLFAGS to -Wall

```
CC = gcc
CFLAGS = -Wall

all: prog1 prog2

clean:
	rm -f prog1 prog2
```

And now we will get a warning!

```
[brownek@pascal ~]$ make
gcc -Wall    prog1.c   -o prog1
prog1.c: In function ‘main’:
prog1.c:6:3: warning: ‘i’ is used uninitialized in this functi
on [-Wuninitialized]
   printf("\nProgram %d\n", i);
   ^~~~~~~~~~~~~~~~~~~~~~~~~~
gcc -Wall    prog2.c   -o prog2
```

Makefiles

- You will be learning more about makefiles in tutorial next week

- I'll be posting some resources to help you
  - The official guide is **very** lenghy as it references everything in formal language
  - But there are some great tutorials online too

- Your next assignment will involve using makefiles, will be posted before Monday