Make Tutorial

DR. YIHSIANG LIOW (FEBRUARY 4, 2024)

Contents

1 What is make?

In software development, make is a utility for automatically building executable programs and libraries from source code. Files called makefiles specify how to derive the target program from each of its dependencies. Make can decide where to start through topological sorting. Though Integrated Development Environments and language-specific compiler features can also be used to manage the build process in modern systems, make remains widely used, especially in Unix-based platforms.

- from wikipedia.org

For this tutorial, we will introduce the make utility for compiling C/C++ programs. Therefore you should first read the tutorial on gcc/g++. You can use the make utility for program compilation activity for any programming language. For the examples in this tutorial, I will compile using the simplest options. For real work you might have to use options such as -Wall and -g.

You may use either UNIX or Linux or Cygwin. You must have gcc/g++ and make. (Typing gcc and make at your shell prompt will tell you if you have gcc and make.) For this tutorial, I will be using Linux. Executables will be generated. You should create a temporary directory for these experiments. I will assume throughout that the working directory is this temporary directory. We'll be writing some C/C++ source files and some header files. Let me remind your that header files should end with (at least) one blank line or you might get some confusing error(s).

[Cygwin users: Remember that if you're using Cygwin, executables produced will end with exe by default. You will have to modify what follows on your own.]

2 Compiling a C Program Using `make`

The contents of a `makefile` looks something like a series of

```
[target]: [dependencies]
        [command]
```

WARNING: Each `[command]` must begin on a new line and is preceded by a tab!!! (In other words, in the above, the 8 spaces before the `[command]` is actually a tab. If you're using emacs/xemacs, to force a tab character, do `C-Q` before you press the tab key.)

First create a C program and call it `hw.c`:

```
#include <stdio.h>

int main()
{
    printf("hello world\n");
    return 0;
}
```

Next create a file called `makefile` with the following content:

```
# This is my first makefile
hw.exe: hw.c
        gcc hw.c -o hw.exe
```

Now type `make` at your shell prompt. Read the output. Do an `ls -la`. What do you think happened?

Exercise 2.1. Write a helloworld C++ program (using `std::cout` for printing of course) and call the source file `hw.cpp`. Modify the `makefile` to execute `g++` instead of `gcc`. Execute `make`. □

3 Executing More Than one Command

By the way, you can have more than one command for each target:

```
[target]: [dependencies]
        [command-1]
        [command-2]
```

Let's try that out. First clean up by doing `rm hw`. Next modify your `makefile`:

```
# This is my first makefile
hw.exe: hw.c
        gcc hw.c -o hw.exe
        ls -la
```

Type `make` again.

Exercise 3.1. Redo the above exercise for C++ (instead of C). □

Now revert the `makefile` to what it was (i.e. remove the `ls -la`).

Exercise 3.2. Modify the `makefile` so that after building the executable, you run the executable. After you're done with this experiment, revert your `makefile` to the original version. □

4 Recompiling a C Program

For this section, the first thing I want you to do is to remove the compiled executable by doing `rm hw.exe`.

Next, execute `make`. Do an `ls -la` and look at the timestamp of your executable `hw.exe`.

Wait a minute and then type `make`. Read the output. Look at the timestamp of your executable `hw.exe` again. Let's have some intelligent interpretation ...

Next let's say we modify our program:

```
#include <stdio.h>

int main()
{
    printf("hello world ... 0\n");
    return 0;
}
```

Save the program. Now type `make` again. Do `ls -la` and look at the timestamp of your executable `hw.exe`.

Punchline: Whether `make` will execute a command or not depends on the timestamp of the target, `hw.exe`, i.e., output file of the command of the target, and the file(s) from the dependency (or dependencies), i.e., `hw.c`. So using the above example, if `hw.c` was modified after the program `hw.exe` was generated, `make` will execute the command to recompile the program. However, if `hw.c` was not modified after `hw.exe` was built, then `make` will not rebuild `hw.exe`. And what if the target `hw.exe` is not found? Of course the command for that target is executed.

Now clean up by doing `rm hw.exe`.

Exercise 4.1. Do the same experiment as above with the following `makefile`:

```
# This is my first makefile
hw.exe:
        gcc hw.c -o hw.exe
```

(See the difference?) Remove `hw.exe` and run `make`. Now make some minor modification to `hw.c` and run `hw.exe` again. Get it? When you're done with this experiment, revert your `makefile` to the original. □

Exercise 4.2. Do the same experiment as above with the following `makefile`:

```
# This is my first makefile
abc:
        gcc hw.c -o hw.exe
```

(See the difference?) Remove `hw.exe` and run `make`. Here's the question: What will happen with you execute `make` a second time? Now test your conjecture by running `make`. Get it? When you're done, revert your `makefile` to the original. □

5 Makefile with Two Targets

Now create another C program, say `hw1.c`.

```
#include <stdio.h>

int main()
{
    printf("hello world ... 1\n");
    return 0;
}
```

Modify your `makefile`:

```
# This is my first makefile
hw.exe: hw.c
        gcc hw.c -o hw.exe

hw1.exe: hw1.c
        gcc hw1.c -o hw1.exe
```

First execute `make hw1.exe`. What happened?

This shows you that you can specify the target when you execute `make`. Clean up: do `rm hw1.exe`.

Now do `make`. What happened? This shows you that if you do not specify the target, the first target of the `makefile` is used. Clean up: do `rm hw` and `rm hw1`; revert your `makefile` to what it was:

```
# This is my first makefile
hw.exe: hw.c
        gcc hw.c -o hw.exe
```

6 Building Object Files

Now add this `printhw.h`:

```
#ifndef PRINTHW_H
#define PRINTHW_H

void printhw(int);

#endif

```

(remember to include that blank line at the end!) and `printhw.c`:

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

void printhw(int i)
{
    printf("hello world ... %d\n", i);
}
```

(Well, in this case the second `#include` is actually redundant for this example. But anyway ...)

Modify `hw.c`:

```
#include "printhw.h"

int main()
{
    printhw(0);
    return 0;
}
```

Now modify your `makefile` as follows:

```
# This is my first makefile
hw.exe: printhw.o hw.o
        gcc printhw.o hw.o -o hw.exe

printhw.o: printhw.h printhw.c
        gcc -c printhw.c

hw.o: printhw.h hw.c
        gcc -c hw.c
```

Type `make`. What happens? Read the output message of `make`. Do an `ls la`.

The default target executed is the `hw.exe` target. However in this case, before the command for target `hw` is execute, `make` will enter the targets from the dependencies, `printhw.o`

and `hw.o` and execute their commands. Get it?

Now suppose you make a change in `hw.c`:

```
#include "printhw.h"

int main()
{
    printhw(0);
    printhw(1);
    return 0;
}
```

If you run `make`, how many targets will have their commands executed? (Verify!)

The important point: Recall (example: from CISS240/245) that building an executable is made up of several steps, not one single step. The last step is combining (linking) object files into a single executable. Of course the step before that is to build some object files. Object files can be created from C/C++ file. If you have 1000 C/C++ files and you only modify one of them, then you really only need to build the object file for that modified file and then link the 999 object files with this new object file to create the executable. This will save you a lot of time. This is particularly the case if your executable involves many C++ classes or C structures and you build many intermediate object files and keep track of what needs to be rebuilt using a `makefile`. (There are other ways to build an executable.)

Exercise 6.1. What will happen if you have this `makefile` instead

```
# This is my first makefile
printhw.o:printhw.h printhw.c
        gcc -c printhw.c

hw.o:   printhw.h hw.c
        gcc -c hw.c

hw.exe: printhw.o hw.o
        gcc printhw.o hw.o -o hw.exe
```

and you execute `make`.

WARNING: It's extremely important to get the dependencies correct! If written incorrectly, you could end up with a `makefile` that does *not* recompile a particular file even though it's been modified!

7 Personal Header File Dependency

If your source code uses header files which does not change, then you do not have to include the header files. For instance you do not need to have `iostream.h` in the list of dependencies since `iostream.h` won't change (you don't intend to change it, right? I hope not.)

But what if you're writing your own header files? Again if these header files are not changed, then they do not have to be included. But what if you're still working on them?

In general, if `x.cpp` has the following includes

```
#include "a.h"
#include "b.h"
#include "c.h"

...
```

where `a.h`, `b.h`, `c.h` are your own header files and they might change and they are in the same directory where you have the `makefile`, then the section in your `makefile` for building `x.o` should look like this:

```
x.o:    a.h b.h c.h x.c
        gcc -c x.c
```

Of course if you're done working on `c.h`, then you can leave it out:

```
x.o:    a.h b.h x.c
        gcc -c x.c
```

But there's no harm in including `c.h` anyway.

Here's an example. Write the following files:

Now when we execute `make` and when we execute `main.exe` we get:

```
a() ... 1
```

Now modify `a.h` like this:

```
#ifndef A_H
#define A_H
#include <iostream>

void a()
{
    std::cout << "a() ... 2" << std::endl;
}

#endif
```

When you execute `make`, you will see the `main.exe` is not rebuilt: `make` will tell you that

`main.exe` is up to date.

However if your `makefile` is changed to this:

```
main.exe: main.cpp a.h
    g++ main.cpp -o main.exe
```

you will see that on executing `make`, you *will* get a new `main.exe`. You can change your `a.h` to this:

```
#ifndef A_H
#define A_H
#include <iostream>

void a()
{
    std::cout << "a() ... 3" << std::endl;
}

#endif
```

and again, `make` *will* rebuild your `main.exe`.

In the above example, the header file is in the same directory as the `makefile`. The case where the header file is somewhere else, i.e., in a different directory, can also be easily handled. Create a subdirectory `include/` and move `a.h` into `include/`. Change your `makefile`:

```
# File: makefile
main.exe: main.cpp include/a.h
    g++ main.cpp -o main.exe -I include/
```

Test your `makefile` including modification to `include/a.h`.

That's it.

8 Cleaning up

Sometimes you want to get rid of the output files of `make`. In the case of our C/C++ programs, that would be the object files and perhaps the executable. Here's the standard way of doing it using the `makefile`:

```
# This is my first makefile
hw.exe: printhw.o hw.o
        gcc printhw.o hw.o -o hw.exe

printhw.o: printhw.h printhw.c
        gcc -c printhw.c

hw.o:   printhw.h hw.c
        gcc -c hw.c

clean:
        rm -f printhw.o hw.o hw.exe
```

Type `make clean` at the shell prompt. Read the output. Do an `ls -la` to see what files are removed.

Exercise 8.1. Take any 10 problems from your CISS240 class (or any 5 problems from your C++ textbook) and write the programs and one makefile for each program. You want to have each program be in a separate folder. □

9 Macros/Variables

Some of the terms used in the `makefile` occurs several times. It's a common practice to create macros for them. It makes the `makefile` easier to read and maintain.

```
# This is my first makefile
CC        = gcc                 # C compiler
CCFLAGS   = -g                  # compiler options: debugging flag on
LINK      = gcc                 # Linker
LINKFLAGS =                     # Linker options
OBJS      = hw.o printhw.o      # Object files
EXE       = hw.exe              # Executable file

$(EXE): $(OBJS)
        $(LINK) $(LINKFLAGS) $(OBJS) -o $(EXE)

printhw.o: printhw.h printhw.c
        $(CC) $(CCFLAGS) -c printhw.c

hw.o:   printhw.h hw.c
        $(CC) $(CCFLAGS) -c hw.c

clean:
        rm -f $(OBJS) $(EXE)
```

`CC`, `CCFLAGS`, `LINKFLAGS`, `OBJS` and `EXE` are macros. For instance `CC` is a macro for `gcc`. When you run `make` and it sees `$(CC)`, it will replace `$(CC)` with `gcc`.

If you need to use a different C compiler all you need to do is to change the macros. The macros are pretty standard. Don't comment them! Or someone will think you're a clueless noob. I'm putting them there just to help you read the `makefile`. There are however some minor variations in the names of these macros. For instance instead of `CCFLAGS` you might see `CCOPTS`. Another thing is that the `LINK` macro has the same value as the `CC` macro. So usually there is no `LINK` macro. When you read some `makefile`s, you will see `LD` or `LINKER` instead of `LINK`.

Exercise 9.1. Write a C++ project. Choose one that has several cpp files. Use a `makefile`. The usual name for the C++ macro is `CXX`. Your compiler and linker value should be `g++`. □

10 Miscellaneous

There are many other features in `make` that I won't go into. You will find lots of additional information on `make` on the web. I'll leave you with three final things ...

I also include a macro to run the executable and a macro `c` for clean.

```
main.exe: main.cpp
        g++ *.cpp -o main.exe

clean:
        rm -f main.exe
c:
        rm -f main.exe

run:
        ./main.exe
r:
        ./main.exe
```

If a line (dependency or command) is too long, you can break the line in the middle with a backslash \ like this: For instance:

```
clean:
         rm firstobjectfile, secondobjectfile, thirdobjectfile, \
fourthobjectfile
```

It's probably neater to indent:

```
clean:
         rm firstobjectfile, secondobjectfile, thirdobjectfile, \
         fourthobjectfile
```

There's a built-in macro that refers to the target. For instance suppose this is a piece of your `makefile`:

```
hw.o:   hw.cpp
        g++ hw.cpp -c -o hw.o
```

then you can do this instead:

```
hw.o:   hw.cpp
        g++ hw.cpp -c -o $@
```

11 Summary

For your `makefile` for C++ programs, you should probably start with this:

```
main.exe: *.cpp *.h
        g++ *.cpp -o main.exe -Wall -g

clean:
        rm -f main.exe
c:
        rm -f main.exe

run:
        ./main.exe
r:
        ./main.exe
```

12 Sample makefiles

Go to https://github.com/yliow/makefiles for some sample makefiles.

13 FAQ

Q: What do I do with this error when I run `make` after modifying my `main.cpp`?

```
make: Warning: File `main.cpp' has modification time 0.032 s in the future
```

A: There's a time issue. Just get rid of the executable:

```
make clean
```

and do `make` again.

Q: No matter how much I modify my program, `make` keeps telling me my executable is up to date.
A: The file you are modifying is probably is not on the dependencies of your executable target. Make sure you are modifying a source file that is really used by `make` that you are executing. Frequently I find people working on a file in directory `x` and executing `make` in directory `y`. This happens especially if you have a duplicate copy of your project.

TODO:

1. Object/library file dependency. See `python-c-cpp/`.