How to make a SIMPLE C++ Makefile

Asked 14 years, 1 month ago Modified 4 years, 2 months ago Viewed 594k times

346

We are required to use a Makefile to pull everything together for our project, but our professor never showed us how to.

I only have *one* file, `a3driver.cpp` . The driver imports a class from a location, `"/user/cse232/Examples/example32.sequence.cpp"` .

That's it. Everything else is contained with the `.cpp` .

How would I go about making a simple Makefile that creates an executable called `a3a.exe` ?

c++ makefile

Share Edit Follow

edited Oct 22, 2019 at 5:31
Peter Mortensen
31.2k 22 109 132

asked Mar 20, 2010 at 0:02

Befall
6,660 9 25 29

14 .EXE so its definitely Windows. On second thought... the path is Unix-style. Probably using Mingw-32. – Nathan Osman Mar 20, 2010 at 0:07

3 Sigh. I suppose you have to learn the basic of every trade, even if you will never use them. Just have to understand how stuff works. Chances are good, though, that you will always develop in an IDE, like Eclipse. You will get an answer here for your simple one-line case and there are plenty of web tutorials, but if you want in-dpth knowledge, you can't beat the O'reilly book (same for most s/w topics). amazon.com/Managing-Projects-Make-Nutshell-Handbooks/dp/... Pick a 2nd hand copy from amazon, half.com, betterworldbooks eBay – Mawg Mar 20, 2010 at 1:08

3 The link posted by @Dennis is now dead, but the same material can be found in this archive.org page. – Guilherme Salomé Feb 9, 2018 at 20:51

I prefer this person's ideas. (hiltmon.com/blog/2013/07/03/...) The project structure can be easily modified to suit. And I also agree that developer time should be spent on other things than automake/autoconf. These tools have their place, but perhaps not for internal projects. I am building a script that will produce such a project structure. – Daisuke Aramaki Nov 21, 2018 at 14:12

@GuilhermeSalomé Thanks, I believe this is the best simple and complete tutorial. – Hareen Laks Apr 16, 2020 at 5:46

7 Answers

Sorted by: Highest score (default)

641

Since this is for Unix, the executables don't have any extensions.

One thing to note is that `root-config` is a utility which provides the right compilation and linking flags; and the right libraries for building applications against root. That's just a detail related to the original audience for this document.

Make Me Baby

or You Never Forget The First Time You Got Made

An introductory discussion of make, and how to write a simple makefile

What is Make? And Why Should I Care?

The tool called Make is a build dependency manager. That is, it takes care of knowing what commands need to be executed in what order to take your software project from a collection of source files, object files, libraries, headers, etc., etc.---some of which may have changed recently---and turning them into a correct up-to-date version of the program.

Actually, you can use Make for other things too, but I'm not going to talk about that.

A Trivial Makefile

Suppose that you have a directory containing: `tool` `tool.cc` `tool.o` `support.cc` `support.hh`, and `support.o` which depend on `root` and are supposed to be compiled into a program called `tool`, and suppose that you've been hacking on the source files (which means the existing `tool` is now out of date) and want to compile the program.

To do this yourself you could

1. Check if either `support.cc` or `support.hh` is newer than `support.o`, and if so run a command like

   ```
   g++ -g -c -pthread -I/sw/include/root support.cc
   ```

2. Check if either `support.hh` or `tool.cc` are newer than `tool.o`, and if so run a command like

   ```
   g++ -g  -c -pthread -I/sw/include/root tool.cc
   ```

3. Check if `tool.o` is newer than `tool`, and if so run a command like

   ```
   g++ -g tool.o support.o -L/sw/lib/root -lCore -lCint -lRIO -lNet -lHist -lGraf -
   lGraf3d -lGpad -lTree -lRint \
   -lPostscript -lMatrix -lPhysics -lMathCore -lThread -lz -L/sw/lib -lfreetype -lz
   -Wl,-framework,CoreServices \
   -Wl,-framework,ApplicationServices -pthread -Wl,-rpath,/sw/lib/root -lm -ldl
   ```

Phew! What a hassle! There is a lot to remember and several chances to make mistakes. (BTW-- the particulars of the command lines exhibited here depend on our software environment. These ones work on my computer.)

Of course, you could just run all three commands every time. That would work, but it doesn't scale well to a substantial piece of software (like DOGS which takes more than 15 minutes to compile from the ground up on my MacBook).

Instead you could write a file called `makefile` like this:

```
tool: tool.o support.o
    g++ -g -o tool tool.o support.o -L/sw/lib/root -lCore -lCint -lRIO -lNet -
lHist -lGraf -lGraf3d -lGpad -lTree -lRint \
        -lPostscript -lMatrix -lPhysics -lMathCore -lThread -lz -L/sw/lib -
lfreetype -lz -Wl,-framework,CoreServices \
        -Wl,-framework,ApplicationServices -pthread -Wl,-rpath,/sw/lib/root -lm -
ldl

tool.o: tool.cc support.hh
    g++ -g  -c -pthread -I/sw/include/root tool.cc

support.o: support.hh support.cc
    g++ -g -c -pthread -I/sw/include/root support.cc
```

and just type `make` at the command line. Which will perform the three steps shown above automatically.

The unindented lines here have the form "*target: dependencies*" and tell Make that the associated commands (indented lines) should be run if any of the dependencies are newer than the target. That is, the dependency lines describe the logic of what needs to be rebuilt to accommodate changes in various files. If `support.cc` changes that means that `support.o` must be rebuilt, but `tool.o` can be left alone. When `support.o` changes `tool` must be rebuilt.

The commands associated with each dependency line are set off with a tab (see below) should modify the target (or at least touch it to update the modification time).

Variables, Built In Rules, and Other Goodies

At this point, our makefile is simply remembering the work that needs doing, but we still had to figure out and type each and every needed command in its entirety. It does not have to be that way: Make is a powerful language with variables, text manipulation functions, and a whole slew of built-in rules which can make this much easier for us.

Make Variables

The syntax for accessing a make variable is `$(VAR)`.

The syntax for assigning to a Make variable is: `VAR = A text value of some kind` (or `VAR := A different text value but ignore this for the moment`).

You can use variables in rules like this improved version of our makefile:

```
CPPFLAGS=-g -pthread -I/sw/include/root
LDFLAGS=-g
LDLIBS=-L/sw/lib/root -lCore -lCint -lRIO -lNet -lHist -lGraf -lGraf3d -lGpad -
lTree -lRint \
       -lPostscript -lMatrix -lPhysics -lMathCore -lThread -lz -L/sw/lib -
lfreetype -lz \
       -Wl,-framework,CoreServices -Wl,-framework,ApplicationServices -pthread -
Wl,-rpath,/sw/lib/root \
       -lm -ldl
```

```
tool: tool.o support.o
    g++ $(LDFLAGS) -o tool tool.o support.o $(LDLIBS)

tool.o: tool.cc support.hh
    g++ $(CPPFLAGS) -c tool.cc

support.o: support.hh support.cc
    g++ $(CPPFLAGS) -c support.cc
```

which is a little more readable, but still requires a lot of typing

Make Functions

GNU make supports a variety of functions for accessing information from the filesystem or other commands on the system. In this case we are interested in `$(shell ...)` which expands to the output of the argument(s), and `$(subst opat,npat,text)` which replaces all instances of `opat` with `npat` in text.

Taking advantage of this gives us:

```
CPPFLAGS=-g $(shell root-config --cflags)
LDFLAGS=-g $(shell root-config --ldflags)
LDLIBS=$(shell root-config --libs)

SRCS=tool.cc support.cc
OBJS=$(subst .cc,.o,$(SRCS))

tool: $(OBJS)
    g++ $(LDFLAGS) -o tool $(OBJS) $(LDLIBS)

tool.o: tool.cc support.hh
    g++ $(CPPFLAGS) -c tool.cc

support.o: support.hh support.cc
    g++ $(CPPFLAGS) -c support.cc
```

which is easier to type and much more readable.

Notice that

1. We are still stating explicitly the dependencies for each object file and the final executable
2. We've had to explicitly type the compilation rule for both source files

Implicit and Pattern Rules

We would generally expect that all C++ source files should be treated the same way, and Make provides three ways to state this:

1. suffix rules (considered obsolete in GNU make, but kept for backwards compatibility)
2. implicit rules
3. pattern rules

Implicit rules are built in, and a few will be discussed below. Pattern rules are specified in a form like

```
%.o: %.c
    $(CC) $(CFLAGS) $(CPPFLAGS) -c $<
```

which means that object files are generated from C source files by running the command shown, where the "automatic" variable `$<` expands to the name of the first dependency.

Built-in Rules

Make has a whole host of built-in rules that mean that very often, a project can be compile by a very simple makefile, indeed.

The GNU make built in rule for C source files is the one exhibited above. Similarly we create object files from C++ source files with a rule like `$(CXX) -c $(CPPFLAGS) $(CFLAGS)`.

Single object files are linked using `$(LD) $(LDFLAGS) n.o $(LOADLIBES) $(LDLIBS)`, but this won't work in our case, because we want to link multiple object files.

Variables Used By Built-in Rules

The built-in rules use a set of standard variables that allow you to specify local environment information (like where to find the ROOT include files) without re-writing all the rules. The ones most likely to be interesting to us are:

- `CC` -- the C compiler to use
- `CXX` -- the C++ compiler to use
- `LD` -- the linker to use
- `CFLAGS` -- compilation flag for C source files
- `CXXFLAGS` -- compilation flags for C++ source files
- `CPPFLAGS` -- flags for the c-preprocessor (typically include file paths and symbols defined on the command line), used by C and C++
- `LDFLAGS` -- linker flags
- `LDLIBS` -- libraries to link

A Basic Makefile

By taking advantage of the built-in rules we can simplify our makefile to:

```
CC=gcc
CXX=g++
RM=rm -f
CPPFLAGS=-g $(shell root-config --cflags)
LDFLAGS=-g $(shell root-config --ldflags)
LDLIBS=$(shell root-config --libs)

SRCS=tool.cc support.cc
```

```
OBJS=$(subst .cc,.o,$(SRCS))

all: tool

tool: $(OBJS)
    $(CXX) $(LDFLAGS) -o tool $(OBJS) $(LDLIBS)

tool.o: tool.cc support.hh

support.o: support.hh support.cc

clean:
    $(RM) $(OBJS)

distclean: clean
    $(RM) tool
```

We have also added several standard targets that perform special actions (like cleaning up the source directory).

Note that when make is invoked without an argument, it uses the first target found in the file (in this case all), but you can also name the target to get which is what makes `make clean` remove the object files in this case.

We still have all the dependencies hard-coded.

Some Mysterious Improvements

```
CC=gcc
CXX=g++
RM=rm -f
CPPFLAGS=-g $(shell root-config --cflags)
LDFLAGS=-g $(shell root-config --ldflags)
LDLIBS=$(shell root-config --libs)

SRCS=tool.cc support.cc
OBJS=$(subst .cc,.o,$(SRCS))

all: tool

tool: $(OBJS)
    $(CXX) $(LDFLAGS) -o tool $(OBJS) $(LDLIBS)

depend: .depend

.depend: $(SRCS)
    $(RM) ./.depend
    $(CXX) $(CPPFLAGS) -MM $^>>./.depend;

clean:
    $(RM) $(OBJS)

distclean: clean
    $(RM) *~ .depend

include .depend
```

Notice that

1. There are no longer any dependency lines for the source files!?!

2. There is some strange magic related to .depend and depend
3. If you do `make` then `ls -A` you see a file named `.depend` which contains things that look like make dependency lines

Other Reading

- GNU make manual
- Recursive Make Considered Harmful on a common way of writing makefiles that is less than optimal, and how to avoid it.

Know Bugs and Historical Notes

The input language for Make is whitespace sensitive. In particular, *the action lines following dependencies must start with a tab*. But a series of spaces can look the same (and indeed there are editors that will silently convert tabs to spaces or vice versa), which results in a Make file that looks right and still doesn't work. This was identified as a bug early on, but (the story goes) it was not fixed, because there were already 10 users.

(This was copied from a wiki post I wrote for physics graduate students.)

Share Edit Follow edited Mar 9, 2020 at 22:25 community wiki
23 revs, 11 users 84%
dmckee --- ex-moderator kitten

11 This method to generate dependencies is obsolete and actually harmful. See Advanced Auto-Dependency Generation. – Maxim Egorushkin Sep 30, 2011 at 16:46

5 `-pthread` flag causes `gcc` to define the necessary macros, `-D_REENTRANT` is unnecessary. – Maxim Egorushkin Sep 30, 2011 at 16:47

8 @jcoe It does an unnecessary extra preprocessor pass to generate dependencies. Doing unnecessary work it just dissipates heat melting the ice poles and, on a bigger scale, is nearing the heat death of our universe. – Maxim Egorushkin Aug 11, 2014 at 8:47

2 @jcoe In addition to Maxim's high minded costs there is the very immediate cost of making your build take longer. Once projects get larger than a few developers and a few score files compilation time gets to be an issue, and with unwise use of `make` features one can generate ridiculously slow and still incorrect builds. If you've got some time read "Recursive make considered harmful". I do it the wrong way here because (1) cargo-cult and (2) I know how to explain it when someone asks. – dmckee --- ex-moderator kitten Aug 12, 2014 at 1:05

5 Really, the accepted answer should not depend on a very specific piece of software (`root-config`). A more general alternative with the same capability should be proposed if any or it should be just left out. I didn't downvote because of the listing and explanation of the most frequently used make macros. – green diod Dec 6, 2016 at 13:59

68

I've always thought this was easier to learn with a detailed example, so here's how I think of makefiles. For each section you have one line that's not indented and it shows the name of the section followed by dependencies. The dependencies can be either other sections (which

will be run before the current section) or files (which if updated will cause the current section to be run again next time you run `make`).

Here's a quick example (keep in mind that I'm using 4 spaces where I should be using a tab, Stack Overflow won't let me use tabs):

```
a3driver: a3driver.o
    g++ -o a3driver a3driver.o

a3driver.o: a3driver.cpp
    g++ -c a3driver.cpp
```

When you type `make`, it will choose the first section (a3driver). a3driver depends on a3driver.o, so it will go to that section. a3driver.o depends on a3driver.cpp, so it will only run if a3driver.cpp has changed since it was last run. Assuming it has (or has never been run), it will compile a3driver.cpp to a .o file, then go back to a3driver and compile the final executable.

Since there's only one file, it could even be reduced to:

```
a3driver: a3driver.cpp
    g++ -o a3driver a3driver.cpp
```

The reason I showed the first example is that it shows the power of makefiles. If you need to compile another file, you can just add another section. Here's an example with a secondFile.cpp (which loads in a header named secondFile.h):

```
a3driver: a3driver.o secondFile.o
    g++ -o a3driver a3driver.o secondFile.o

a3driver.o: a3driver.cpp
    g++ -c a3driver.cpp

secondFile.o: secondFile.cpp secondFile.h
    g++ -c secondFile.cpp
```

This way if you change something in secondFile.cpp or secondFile.h and recompile, it will only recompile secondFile.cpp (not a3driver.cpp). Or alternately, if you change something in a3driver.cpp, it won't recompile secondFile.cpp.

Let me know if you have any questions about it.

It's also traditional to include a section named "all" and a section named "clean". "all" will usually build all of the executables, and "clean" will remove "build artifacts" like .o files and the executables:

```
all: a3driver ;

clean:
    # -f so this will succeed even if the files don't exist
    rm -f a3driver a3driver.o
```

EDIT: I didn't notice you're on Windows. I think the only difference is changing the `-o a3driver` to `-o a3driver.exe`.

Share Edit Follow

edited Oct 23, 2019 at 14:18

answered Mar 20, 2010 at 0:15

Brendan Long
53.7k 21 150 193

The absolute code I'm trying to use is: p4a.exe: p4driver.cpp g++ -o p4a p4driver.cpp BUT, it tells me "missing separator". I'm using TAB, but it still tells me that. Any idea? – Befall Mar 20, 2010 at 0:25

---

2 As far as I can tell, that error message only comes up if you have spaces. Make sure that you don't have any lines starting with spaces (space + tab will give that error). That's the only thing I can think of.. – Brendan Long Mar 20, 2010 at 1:13

Note to future editors: StackOverflow can't render tabs even if you edit them into the answer, so please don't try to "fix" my note about that. – Brendan Long Oct 23, 2019 at 14:20

40

Why does everyone like to list out source files? A simple find command can take care of that easily.

Here's an example of a dirt simple C++ Makefile. Just drop it in a directory containing `.c` files and then type `make` ...

```
appname := myapp

CXX := clang++
CXXFLAGS := -std=c++11

srcfiles := $(shell find . -name "*.C")
objects  := $(patsubst %.C, %.o, $(srcfiles))

all: $(appname)

$(appname): $(objects)
    $(CXX) $(CXXFLAGS) $(LDFLAGS) -o $(appname) $(objects) $(LDLIBS)

depend: .depend

.depend: $(srcfiles)
    rm -f ./.depend
    $(CXX) $(CXXFLAGS) -MM $^>>./.depend;

clean:
    rm -f $(objects)

dist-clean: clean
    rm -f *~ .depend

include .depend
```

Share Edit Follow

edited Oct 22, 2019 at 6:22

Peter Mortensen
31.2k 22 109 132

answered Feb 22, 2015 at 22:17

friedmud
673 6 7

5 A reason not to auto-find source files is that one can have different build targets needing different files. – hmijail Aug 27, 2015 at 18:41

Agreed @hmijail, as well as submodules that contain a ton of sources/headers you don't want compiled / linked... and undoubtedly many other circumstances where exhaustive search / use is unsuitable. – Engineer Oct 15, 2015 at 10:47

Why use "shell find" and not "wildcard" instead? – Nolan Aug 28, 2017 at 22:39

1 @Nolan to find source files in a source directory tree – AlejandroVD Feb 13, 2018 at 22:15

15

You had two options.

Option 1: simplest makefile = NO MAKEFILE.

Rename "a3driver.cpp" to "a3a.cpp", and then on the command line write:

```
nmake a3a.exe
```

And that's it. If you're using GNU Make, use "make" or "gmake" or whatever.

Option 2: a 2-line makefile.

```
a3a.exe: a3driver.obj
    link /out:a3a.exe a3driver.obj
```

Share Edit Follow

edited Oct 22, 2019 at 6:13 — Peter Mortensen — 31.2k 22 109 132

answered Apr 16, 2014 at 15:40 — No one — 151 1 2

3 This would be an excellent answer if it didn't presuppose so many things about details of the OP's environment. Yes, they are on Windows, but that doesn't mean they are using `nmake` . The `link` command line also looks very specific to a particular compiler, and should at the very least document which one. – tripleee May 19, 2019 at 11:20

12

I used friedmud's answer. I looked into this for a while, and it seems to be a good way to get started. This solution also has a well defined method of adding compiler flags. I answered again, because I made changes to make it work in my environment, Ubuntu and g++. More working examples are the best teacher, sometimes.

```
appname := myapp

CXX := g++
CXXFLAGS := -Wall -g

srcfiles := $(shell find . -maxdepth 1 -name "*.cpp")
objects  := $(patsubst %.cpp, %.o, $(srcfiles))

all: $(appname)
```

```
$(appname): $(objects)
    $(CXX) $(CXXFLAGS) $(LDFLAGS) -o $(appname) $(objects) $(LDLIBS)

depend: .depend

.depend: $(srcfiles)
    rm -f ./.depend
    $(CXX) $(CXXFLAGS) -MM $^>>./.depend;

clean:
    rm -f $(objects)

dist-clean: clean
    rm -f *~ .depend

include .depend
```

Makefiles seem to be very complex. I was using one, but it was generating an error related to not linking in g++ libraries. This configuration solved that problem.

Share Edit Follow

edited Oct 22, 2019 at 6:19 — Peter Mortensen — 31.2k — 22 — 109 — 132

answered Aug 22, 2015 at 20:43 — VectorVortec — 709 — 7 — 10

8

I suggest (note that the indent is a TAB):

```
tool: tool.o file1.o file2.o
    $(CXX) $(LDFLAGS) $^ $(LDLIBS) -o $@
```

or

```
LINK.o = $(CXX) $(LDFLAGS) $(TARGET_ARCH)
tool: tool.o file1.o file2.o
```

The latter suggestion is slightly better since it reuses GNU Make implicit rules. However, in order to work, a source file must have the same name as the final executable (i.e.: `tool.c` and `tool`).

Notice, it is not necessary to declare sources. Intermediate object files are generated using implicit rule. Consequently, this `Makefile` work for C and C++ (and also for Fortran, etc...).

Also notice, by default, Makefile use `$(CC)` as the linker. `$(CC)` does not work for linking C++ object files. We modify `LINK.o` only because of that. If you want to compile C code, you don't have to force the `LINK.o` value.

Sure, you can also add your compilation flags with variable `CFLAGS` and add your libraries in `LDLIBS`. For example:

```
CFLAGS = -Wall
LDLIBS = -lm
```

One side note: if you have to use external libraries, I suggest to use pkg-config in order to correctly set `CFLAGS` and `LDLIBS`:

```
CFLAGS += $(shell pkg-config --cflags libssl)
LDLIBS += $(shell pkg-config --libs libssl)
```

The attentive reader will notice that this `Makefile` does not rebuild properly if one header is changed. Add these lines to fix the problem:

```
override CPPFLAGS += -MMD
include $(wildcard *.d)
```

`-MMD` allows to build .d files that contains Makefile fragments about headers dependencies. The second line just uses them.

For sure, a well written Makefile should also include `clean` and `distclean` rules:

```
clean:
    $(RM) *.o *.d

distclean: clean
    $(RM) tool
```

Notice, `$(RM)` is the equivalent of `rm -f`, but it is a good practice to not call `rm` directly.

The `all` rule is also appreciated. In order to work, it should be the first rule of your file:

```
all: tool
```

You may also add an `install` rule:

```
PREFIX = /usr/local
install:
    install -m 755 tool $(DESTDIR)$(PREFIX)/bin
```

`DESTDIR` is empty by default. The user can set it to install your program at an alternative system (mandatory for cross-compilation process). Package maintainers for multiple distribution may also change `PREFIX` in order to install your package in `/usr`.

One final word: Do not place source files in sub-directories. If you really want to do that, keep this `Makefile` in the root directory and use full paths to identify your files (i.e. `subdir/file.o`).

So to summarise, your full Makefile should look like:

```
LINK.o = $(CXX) $(LDFLAGS) $(TARGET_ARCH)
PREFIX = /usr/local
override CPPFLAGS += -MMD
include $(wildcard *.d)
```

```
all: tool
tool: tool.o file1.o file2.o
clean:
    $(RM) *.o *.d
distclean: clean
    $(RM) tool
install:
    install -m 755 tool $(DESTDIR)$(PREFIX)/bin
```

Share Edit Follow

edited Oct 22, 2019 at 8:20

answered Apr 20, 2016 at 8:41

Jérôme Pouiller
9,687 5 42 48

Near the end: Shouldn't there be empty lines between the rules? John Knoeller's answer claimed that.
– Peter Mortensen Oct 22, 2019 at 6:36

None of implementation of `make` that I know (GNU Make and BSD Make) need empty lines between rules. However, it exists tons of `make` implementations with theirs own bugs^Wspecificities.
– Jérôme Pouiller Oct 22, 2019 at 8:32

8

Your Make file will have one or two dependency rules depending on whether you compile and link with a single command, or with one command for the compile and one for the link.

Dependency are a tree of rules that look like this (note that the indent *must* be a TAB):

```
main_target : source1 source2 etc
    command to build main_target from sources

source1 : dependents for source1
    command to build source1
```

There *must* be a blank line after the commands for a target, and there must *not* be a blank line before the commands. The first target in the makefile is the overall goal, and other targets are built only if the first target depends on them.

So your makefile will look something like this.

```
a3a.exe : a3driver.obj
    link /out:a3a.exe a3driver.obj

a3driver.obj : a3driver.cpp
    cc a3driver.cpp
```

Share Edit Follow

edited Oct 22, 2019 at 6:11

Peter Mortensen
31.2k 22 109 132

answered Mar 20, 2010 at 0:19

John Knoeller
33.9k 4 64 92

Highly active question. Earn 10 reputation (not counting the association bonus) in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.