make

make is a utility for building applications. This tutorial will teach you how to use this utility with Makefiles. This tutorial is primarily focused on GNU make.

> You may think of Make as merely a tool for building large binaries or libraries (and it is, almost to a fault), but it's much more than that. **Makefiles are machine-readable documentation that make your workflow reproducible.**
>
> *– Mike Bostock , "Why Use Make"*[1]

Hand compilation

Before we start talking about "make", let's first describe the build process used long ago: "hand compilation", aka "manual build". That processes isn't too bad if you only have one or two source files. A simple compilation process would be by typing the following code by hand:

```
g++ -o file file.cpp
```

For beginners, this might take a while to get used to. As an alternative, using make the command would be:

```
make file
```

Both commands will do the same thing - take the file.cpp source file, compile it and then link it into an executable.

But where the problem comes in is where the user has to use more than one source file, or is dependent on libraries. Making an SDL application might consist of something like this:

```
g++ `sdl-config --cflags` -o file file.cpp
```

This is getting harder to remember, isn't it?

Let's add some optimisation:

```
g++ -O3 -fomit-frame-pointer `sdl-config --cflags` -o file file.cpp
```

Do you really want to type that whole command line every time you make the tiniest edit to your source files? This is where make comes in - it will save your time!

Basics

A make file works as a simple dependency tree - it compiles the stuff that is outdated and then links your software together. You will have to specify the compilation options, but they will not be as tough on you anymore. A simple makefile for compiling your application:

Makefile

```
default: myapp
myapp:
   g++ -o file file.cpp
```

To compile your application, you may do:

```
make
```

or:

```
make myapp
```

Well, you may now ask, how much of use is this? I can make a shell script for this myself! So lets now work on multiple files and object files - your application is big enough to be linked now: it contains five files.

The Test Case

Globally, there are three functions present:

```
int main(); (file1.cpp)
void file2function(std::string from); (file2.cpp)
void file3function(std::string from); (file3.cpp)
```

File2 and file3 have their header files, so they could be inter-linked. The actual application will look like this:

file1.cpp

```
#include "file2.h"
#include "file3.h"
int main()
{
   file2function("main");
   file3function("main");
}
```

file2.h

```
#include <string>
#include <iostream>
#include "file3.h"
void file2function(std::string source);
```

file2.cpp

```
#include "file2.h"
void file2function(std::string from)
{
   std::cout << "File 2 function was called from " << from << std::endl;
   file3function("file2");
}
```

file3.h

```
#include <string>
#include <iostream>
#include "file2.h"
void file3function(std::string source);
```

file3.cpp

```
#include "file3.h"
void file3function(std::string from)
{
   std::cout << "File 3 function was called from " << from << std::endl;
}
```

So, how would I link this all up? Well, the most basic way is to do it all by hand:

```
g++ -c -o file2.o file2.cpp
g++ -c -o file3.o file3.cpp
g++ -o file1 file1.cpp file2.o file3.o
```

However, make would offer a simpler way, **Makefile**:

Makefile

```
default: testcase
testcase: file2.o file3.o
    g++ -o file1 file1.cpp file2.o file3.o
```

Pretty different, eh? There's a different approach to do this, which is what makes *make* so good:

Makefile

```
# Create the executable "file1" from all the source files.
COMPILER=g++
OBJS=file1.o file2.o file3.o
default: testcase
testcase: $(OBJS)
    $(COMPILER) -o file1 $(OBJS)
```

How easy is it to expand now? Just add a file to the OBJS list and you're done!

This is a complete makefile that handles the complete process of converting several source files (and creating a bunch of ".o" files as an intermediate step) into the final "file1" executable.

Good programmers put human-readable comments (lines starting with the "#" character) in the makefile to describe why the makefile was written and what *make* is intended to do with this makefile (which, alas, is not always what actually happens).

Advanced make

Long Lines

Some statements in a makefile can be very long. To make them easier to read, some programmers break up long statements into several shorter, easier-to-read physical lines. Many consoles displayed 80 characters across, which is the threshold some programmers use for breaking up lines and comments.

In short, a programmer may decide it looks nicer to writes several short physical lines that together compose one statement, like this:

```
some really long line \
with, like, \
a bajillion parameters and \
several file names \
and stuff
```

Rather than the same statement, crammed into a single physical line:

```
some really long line with, like, a bajillion parameters and several file names and stuff
```

Make Clean

When you type "make clean" from the command prompt, "make" deletes all the files that you specified were easily-replaceable files -- i.e., the intermediate and output files generated from your irreplaceable hand-written source files.

To tell "make" that some file is a easily-replaceable file, add it to the "rm" line of the "clean" section of your makefile, and that file will be deleted *every* time you type "make clean" at the command prompt. (If you don't already have a "clean" section in your makefile, add a section to the end of your makefile that looks like the following):

```
.PHONY: clean
clean:
    rm *.o
    rm file
```

In this example, we've told "make" that all intermediate object files ("*.o") and the final executable ("file") are safe-to-delete, easily-regenerated files. Typically people (or automake scripts) set up the clean step to delete every target in the makefile.[2]

Typically a programmer periodically runs "make clean", then archives *every* file in the directory, and then runs "make" to regenerate every file. Then he tests the program executable, confident that the program he is testing can be easily regenerated entirely from the files in the archive.

Variables

"make" accepts three ways of setting variables.

- *Recursively expanded variables* are defined by using `variable = value`. This means if one variable 'x' contains references to another variable 'y', and 'y' changes after the 'x' being defined, 'y' will include the change made to 'x'.

```
x = abc
y = $(x)ghi
x = abcdef
# x will be abcdef, while y will be abcdefghi
```

- *Simply expanded variables* are defined by using `variable := value`. This means if one variable 'x' contains references to another variable 'y', the 'y' variable will be scanned for once and for all.

```
x := abc
y := $(x)ghi
# x is 'abc' at this time
x := abcdef
# x will be abcdef, while y will be abcghi
```

- If you just want to make sure a variable is set, you can use `variable ?= value`. If the variable is already set, then the definition will not be run; if not, then 'variable' will be set as 'value'.[3]

File Names With Special Characters

Many programmers recommend not creating files with spaces or dollar signs in them, in order to avoid the 'spaces in file names' error in "make" and many other utilities.[4]

If you must deal with a file that has spaces or dollar signs in the file name (such as the file "my $0.02 program.c"), sometimes you can escape the literal name of the file with double slashes and double dollar signs —in the Makefile, that filename can be represented as "my\\ $$0.02\\ program.c".[5]

Alas, the double-slash escape doesn't work when "make" works with lists of files (which are internally represented as space-separated filenames). One work-around is to refer to that file name using a space-free representation like "my+$$0.02+program.c", then later in the make file use the $(subst) function to convert that space-free representation back to the actual on-disk file name.[4] Alas, this work-around can't handle filenames that include both spaces and plus-signs.

Perhaps it's simplest to avoid filenames with spaces in them as inputs or outputs of "make".

Name of the makefile

When creating a new makefile, give it the literal name "`Makefile`" (8 characters, no extension).

In principle, you could create a makefile with some other name, such as `makefile` or `GNUmakefile`, which (like `Makefile`) are automatically found by the GNU version of the `make` utility, or some other name that you pass to the *make* utility with the `--file` option.[6]

Example makefiles

- Examples

References

1. Mike Bostock "Why Use Make" (http://bost.ocks.org/mike/make/). 2013.
2. "Makefiles and RMarkdown" (http://www.r-bloggers.com/makefiles-and-rmarkdown/). 2015.
3. "GNU Make Manual" (http://www.gnu.org/software/make/manual/) section "6.2 The Two Flavors of Variables".
4. John Graham-Cumming. "GNU Make meets file names with spaces in them" (http://www.cmcrossroads.com/ask-mr-make/7859-gnu-make-meets-file-names-with-spaces-in-them). CM Crossroads 2007.
5. John Graham-Cumming. "GNU Make escaping: a walk on the wild side" (http://www.cmcrossroads.com/ask-mr-make/8442-gnu-make-escaping-a-walk-on-the-wild-side) CM Crossroads 2007.
6. "GNU Make Manual" (http://www.gnu.org/software/make/manual/) section "3.2 What Name to Give Your Makefile".

Further reading

Books and manuals

- GNU Make documentation (http://www.gnu.org/software/make/)
- FreeBSD make manual page (http://www.freebsd.org/cgi/man.cgi?query=make&sektion=1)
- Microsoft NMAKE reference (http://msdn.microsoft.com/en-us/library/dd9y37ha.aspx)
- GNU make Standard Library (http://gmsl.sourceforge.net/).
- Managing Projects with GNU make (http://notendur.hi.is/jonasson/software/make-book/).

Tutorials

- The GNU C Programming Tutorial - Writing a makefile (http://crasseux.com/books/ctutorial/Writing-a-makefile.html#Writing%20a%20makefile)
- Makefile Tutorial: How To Write A Makefile (https://sites.google.com/site/michaelsafyan/software-engineering/how-to-write-a-makefile)
- OPUS Makefile Tutorial (http://www.opussoftware.com/tutorial/TutMakefile.htm)
- Makefile Tutorial for C, gcc, & gmake (http://www.thomasstover.com/make.html)
- Using NMake (http://www.c2.com/cgi/wiki?UsingNmake)

Articles

- "Ask Mr. Make" series of article about GNU Make (http://www.cmcrossroads.com/cm-articles/columns/ask-mr-make)
- What is wrong with make? (http://freshmeat.net/articles/view/1702/)
- What's Wrong With GNU make? (http://www.conifersystems.com/whitepapers/gnu-make/)
- Peter Miller. "Recursive Make Considered Harmful" (https://github.com/sam-falvo/nrmf/blob/master/doc/auug97.pdf). *(was Recursive Make Considered Harmful (http://miller.emu.id.au/pmiller/books/rmch/))*
- Advanced Auto-Dependency Generation (http://make.paulandlesley.org/autodep.html).
- "Kleknev: a coarse-grained profiler for build systems" (http://ansuz.sooke.bc.ca/entry/260). Sometimes "make" is used to build extremely large, complicated systems; the Kleknev system helps people figure out what items are eating the most time during that build process.