How to make a Makefile

John Romein, SAM

Vrije Universiteit
Department of Mathematics and Computer Science
Amsterdam, The Netherlands

john@cs.vu.nl, sam@cs.vu.nl

Version 1.0se

June 20, 1996

Abstract

Make is a program that is commonly used to build and maintain large programs. Although very useful, it is not user-friendly. This tutorial tries to explain how *make* works, and how to write a `Makefile`. We start with some simple examples in C and C++, which should be sufficient for most programmers to maintain their programs.

1 Introduction

Large programs usually consist of several components, structured in a modular way. A module has an interface part and an implementation part. Modern languages like C, C++ and Modula-2 provide mechanisms to specify the interface of a module in one file and its implementation in another file. If module B is built upon module A, i.e. uses functions exported by module A, module B merely needs to import module A's interface, but not its implementation.

Compiling a large program costs a lot of time. Fortunately, each module can be compiled separately. The advantage is that we merely have to recompile module A if we change something to the *implementation* of module A, other modules need not be recompiled. This can save a lot of time. If we change something to the *interface* of module A, we have to recompile all modules that use the interface of module A.

Each module's implementation is compiled to a so called *object file*. An object file contains a.o. binary machine instructions for the functions that are exported by the module. All object files that form a program must be linked together to create an *executable*. The executable can be loaded into the computer's memory and executed.

Make helps us to keep the object files and executables up to date. We describe in a `Makefile` how the modules are built upon each other, and how we can generate the object files from the source files, and the executable from the object files. Each time we type "`make`", *make* checks whether the executable and object files exist, and if they do exist, it checks the timestamps of the files to see if they are still up to date with respect to the sources. If this is not the case, *make* concludes that the sources have been changed since the last time the programmer typed "`make`", and it takes the appropriate actions to recompile the necessary modules, and link the objects to a new executable.

Another very important advantage of using a `Makefile` is that it eases the distribution of your program. Somebody who wants to recompile your program on his system merely needs to type "`make`", and an executable is generated that can be run on his or her system.

2 What you need to know about the compiler

file`.c`	C source file
file`.cc` or *file*`.C`	C++ source file
file`.h`	C or C++ header file
file`.o`	object file
file`.a`	library file

Figure 1: File extensions.

In this tutorial, we assume that files have file extensions as shown in Figure 1.

2.1 The stages from sourcefiles to an executable

2.1.1 Making objectfiles from source: `.c` → `.o`

The compiler compiles a C source file to an object file. This is done in three consecutive phases. During the first phase, the C compiler calls the preprocessor `cpp` to `#include` all files and substitute all `#define`s. Then the front-end compiler compiles the preprocessed file to *assembly code*. Next, the assembler (`as`) assembles the assebly code and generates an *object file*.

2.1.2 Making an executable from objectfiles and libraries: `.o`,`.a` → `a.out`

The linker (`ld`) links all object and library files together to an *executable*.

2.2 Compiler options

Various options can be given when the compiler is invoked:

`-c`	Compile each source file to an object file, but do not link.
`-o` *file*	Place the output in file *file*.
`-g`	Place debug information in the generated code, suitable for use with debuggers like `gdb`. It makes objects and executables usually 2–8 times as big, so use with care. The debug information can be removed afterwards with `strip`.
`-I`*path*	Search *path* for include files.
`-L`*path*	Search *path* for library files.
`-l`*library*	Link `lib`*library*`.a` with the object files[1].
`-Wall`	Warn about all questionable constructs (`gcc` and `g++` only).
`-ansi -pedantic`	Warn about all non ANSI-C or ANSI-C++ constructs (`gcc` and `g++` only).

When used in the various stages of the compiling, typical commands would be:

`.c` → `.o`:
  `$ gcc -Wall -ansi -pedantic -g -I/home/john/include -o file1.o -c file1.c`
The object file `file1.o` is created (with debug-information) from the sourcefile `file1.c`.

`.o`,`.a` → `a.out`:
  `$ gcc -o myprog file1.o file2.o file3.o main.o -L/home/john/lib -lmylib -lm`
From all objectfiles (`file1.o`, `file2.o`, `file3.o` and `main.o`) and some libraries (`libmylib` in `/home/john/lib` and `libm`) the executable named `myprog` is created.

3 A simple Makefile

Using a simple example, we are going to develop a `Makefile` that helps us to keep our program up to date.

3.1 Doing it all by hand

Suppose we want to write a module *my_math* that exports the functions `fac()` and `fib()` to compute the faculty and the fibonacci number. We write an interface for *my_math* in the file `my_math.h`, as shown in Figure 2.

```
#ifndef my_math_h
#define my_math_h

extern unsigned fac(unsigned n);
extern unsigned fib(unsigned n);

#endif
```

Figure 2: `my_math.h`: interface of the module *my_math*.

It is a good habit to protect all the declarations in a `.h` file by `#ifndef` constructs. Now this file can be included as often as a user wants, without having the compiler complain about illegal redeclarations.

After we defined an interface for the module, we implement the functions in `my_math.c` (see Figure 3).

[1] Assuming there are no shared libraries.

```
#include "my_math.h"

unsigned fac(unsigned n)
{
    return n == 0 ? 1 : n * fac(n - 1);
}

unsigned fib(unsigned n)
{
    return n < 2 ? n : fib(n - 1) + fib(n - 2);
}
```

Figure 3: `my_math.c`: implementation of the module *my_math*.

Now we want to write a program that reads unsigned integers, and prints its faculty and fibonacci number, until we read something other than a number. We write a separate implementation for this in `main.c`. `main.c` includes `my_math.h`, the interface of the module *my_math*.

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

int main(void)
{
    unsigned n;

    while (scanf("%u", &n) == 1)
        printf("fac(%u) = %u, fib(%u) = %u\n", n, fac(n), n, fib(n));

    return 0;
}
```

Figure 4: `main.c`: main routine.

To compile the program, we need an ANSI-C compiler, e.g. `gcc`. We can compile the program with the following command line:

```
$ gcc -O2 -ansi -pedantic -Wall main.c my_math.c -o facfib
$
```

This produces a file `facfib` which can be executed directly. The default name of an executable is `a.out`, but it can be overridden by specifing `-o` *file*.

We can also compile the modules separately. The `-c` flag tells the compiler to generate an object file, and then stop.

```
$ gcc -O2 -ansi -pedantic -Wall -c main.c
$ gcc -O2 -ansi -pedantic -Wall -c my_math.c
$ gcc main.o my_math.o -o facfib
$
```

The first command tells `gcc` to create `main.o`, the second command creates `my_math.o` and the last command links the two files to create the executable `facfib`.

3.2 Finding the dependencies

Now assume that we want to change the implementation of the faculty and fibonacci functions so that the computations are done iteratively instead of recursively.

```
#include "my_math.h"

unsigned fac(unsigned n)
{
    unsigned p = 1;

    while (n > 1)
        p *= n --;

    return p;
}

unsigned fib(unsigned n)
{
    unsigned f0 = 0, f1 = 1, f2 = n;

    while (n -- > 1)
        f2 = f0 + f1, f0 = f1, f1 = f2;

    return f2;
}
```

Figure 5: `my_math.c`: iterative implementation of the module *my_math*.

Now we want to compile as few source files as possible. We therefore draw a dependency graph as shown in Figure 6.

The dependency graph shows which file should be regenerated if one or more header or implementation files change. It matches the `#include` structure of our program very closely. Only the `#include <stdio.h>` is omitted, as is the dependency `a.out` → `libc.a`, while we assume that the standard library and include files seldom change.

From the graph we derive that changing `my_math.c` affects `my_math.o` and `facfib`. We therefore have to recompile `my_math.c` and relink the object files. It is not necessary to recompile `main.c`, because `main.o` does not depend on `my_math.c`. We can do this manually by typing:

```
$ gcc -O2 -ansi -pedantic -Wall -c my_math.c
$ gcc main.o my_math.o -o facfib
$
```

3.3 Let *make* do the work

We can automate this process by writing a `Makefile` for our program. A sample `Makefile` is shown in Figure 7.

This `Makefile` is very simple. Three **rules** are shown for the files that must be generated. Each file name at the left hand side of a colon is called a **target**; after the colon the target's **dependencies**

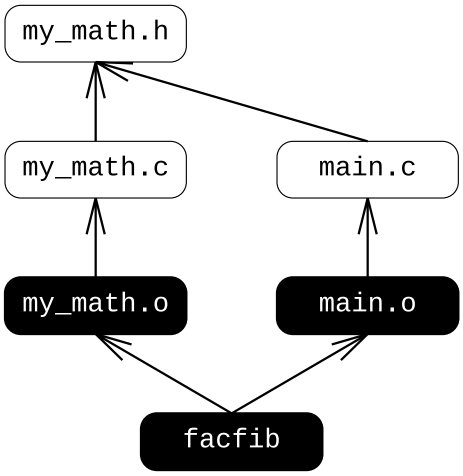

Figure 6: Dependencies in our example. White boxes are source files, black boxes are generated files.

```
facfib:         my_math.o main.o
                gcc my_math.o main.o -o facfib

my_math.o:      my_math.c my_math.h
                gcc -O2 -ansi -pedantic -Wall -c my_math.c

main.o:         main.c my_math.h
                gcc -O2 -ansi -pedantic -Wall -c main.c
```

Figure 7: An example `Makefile`.

are specified. *Make* rebuilds a target whenever it does not exist or when it is out-of-date with respect to one of its dependencies. For example, after we rewrote our `my_math.c` implementation, `my_math.o` and `facfib` are out-of-date with respect to `my_math.c`.

Rebuilding is done according to the **commands** that are specified on the subsequent lines. Note that a command must start with a TAB character, and not with a SPACE. The order in which entries appear is unimportant, except that *make* builds the first target (in this case `facfib`) if no target is specified on the command line. So always write the rule that builds the executable of your program as first rule. Multiple commands may span consecutive lines; the command line is terminated by a blank line. A line that is terminated by a backslash ('
') character is continued on the next line.

Dependencies are checked *recursively.* A dependency might be target of another entry. So is `my_math.o` both dependency in the first rule and target in the second rule. Before *make* decides whether or not to update `facfib`, it checks if it is necessary to update `my_math.o` and `main.o`.

After entering the `Makefile`, we can run *make* to generate an executable named `facfib`. Unless we tell *make* to be silent, *make* tells what it is doing. Let us see what happens if we simply type "`make`" (after deleting `my_math.o`, `main.o` and `facfib`).

```
$ make
gcc -c -O2 -ansi -pedantic -Wall my_math.c
gcc -c -O2 -ansi -pedantic -Wall main.c
```

```
gcc my_math.o main.o
$ make
make: `facfib' is up to date.
$ touch my_math.c
$ make
gcc -c -O2 -ansi -pedantic -Wall my_math.c
gcc my_math.o main.o
$
```

The first time we run *make*, it builds the targets as specified by the rules. The second time, *make* responds that `facfib` is up to date. If we touch (i.e. modify) `my_math.c` and rerun *make*, it decides to rebuild the targets `my_math.o` and `facfib`.

3.4 Using macro's

Although this `Makefile` works, it is not very flexible. A `Makefile` may contain **macros** that contain some common definitions.

```
SOURCES =       my_math.c main.c
OBJECTS =       my_math.o main.o

CC =            gcc
CFLAGS =        -O2 -ansi -pedantic -Wall

facfib:         $(OBJECTS)
                $(CC) $(OBJECTS) -o facfib

my_math.o:      my_math.c my_math.h
                $(CC) $(CFLAGS) -c my_math.c

main.o:         main.c my_math.h
                $(CC) $(CFLAGS) -c main.c
```

Figure 8: A `Makefile` with macros.

Figure 8 shows a `Makefile` that starts with some **macro definitions**. A macro can be **expanded** by surrounding its name with parenthesis or curly brackets[2] and prepending a dollar sign. Note that `$$` expands to a literal dollar sign.

`CC` is commonly used to denote the name of the C compiler. If its definition is omitted, the default `cc` is used instead. `CFLAGS` is used to express the compiler options; it defaults to an empty string.

3.5 Using patterns

The Makefile in figure 9 shows how we can use pattern substitution to replace patterns in a list of identifiers. We changed the definition of `OBJECTS` to replace all trailing `.c` in `$(SOURCES)` by `.o`. Note also the use of comments, which start with a hash and extend to the end of the line.

A pattern can also be substituted in targets and dependencies. This is very useful to specify how a source file can be compiled to an object file. All source files are compiled in a similar way, so we write a **pattern rule** for generating a `.o` file from a `.c` file.

[2] Not necessary for single character macros.

```
SOURCES =       my_math.c main.c
OBJECTS =       $(SOURCES:%.c=%.o)      # replace trailing .c by .o

CC =            gcc
CFLAGS =        -O2 -ansi -pedantic -Wall

facfib:         $(OBJECTS)
                $(CC) $(OBJECTS) -o facfib

my_math.o:      my_math.c my_math.h
                $(CC) $(CFLAGS) -c my_math.c

main.o:         main.c my_math.h
                $(CC) $(CFLAGS) -c main.c
```

Figure 9: Pattern substitution in macro expansions.

```
SOURCES =       my_math.c main.c
OBJECTS =       $(SOURCES:%.c=%.o)

CC =            gcc
CFLAGS =        -O2 -ansi -pedantic -Wall

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

facfib:         $(OBJECTS)
                $(CC) $(OBJECTS) -o facfib

main.o:         main.c my_math.h
my_math.o:      my_math.c my_math.h
```

Figure 10: The use of *pattern rules*.

Figure 10 shows how we can use such a pattern rule. This rule states how we can make a `.o` file from its corresponding `.c` file. The macro `$<` is used to denote the current dependency. Analogous, `$@` denotes the current target. Note that the dependencies at the end of the `Makefile` remain. *Make* still needs to know *when* it must update a target.

In this example we explicitely gave the pattern rule for `%.o: %.c`, but we do not need to do so. Generation of an object file from a source file is so common, that *make* knows how to do it. If we omit the whole rule, *make* uses a slightly different but functionally equivalent built-in default (suffix) rule. However, defining the rule explicitly never hurts.

3.6 Removing targets

```
clean:
                $(RM) $(OBJECTS) fibfac core
```

Figure 11: An alternate target `clean`.

The most useful property of *make* is that we can automatically create object files and executables. Another useful property is that we can easily tell *make* how to clean them up. In order to do this, we need to specify an alternate target `clean`, as shown in Figure 11. We need not give a definition of `RM`, because *make* expands it to `rm -f` by default. The alternate rule may be placed anywhere, as long as it is after the rule for `facfib`. If we place it before `facfib` and we type "`make`", *make* picks the first target, which will be `clean` instead of `facfib`.

`clean` has no dependencies, so if we type "`make clean`", *make* executes the command at the next line[3]. It is good practice to provide a target `clean` to clean up all generated files.

3.7 Using libraries

Sometimes the user wants to link other libraries with his programs. A very common example is the use of the math library, which is not linked by default. Consider the following program:

```
#include <stdio.h>
#include <math.h>

int main(void)
{
    printf("sqrt(17.0) = %f\n", sqrt(17.0));
    return 0;
}
```

A `Makefile` for this program would look something like:

```
SOURCES =       example.c
OBJECTS =       $(SOURCES:%.c=%.o)

LIBRARIES =     -lm

CC =            gcc
CFLAGS =        -O2 -ansi -pedantic -Wall

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

example:        $(OBJECTS)
                $(CC) $(OBJECTS) $(LIBRARIES)
```

3.8 C++

A `Makefile` for C++ programs needs some minor modifications. Unfortunately, for C++ there are few standards: C++ implementation files can have a `.c`, `.C` or `.cc` extension, and a compiler can recognize any combination of these extensions as source files. Different *make* implementations can have different default macros for compiler names too. *GNU Make* defines `CXX` as `g++` by default and uses `CXXFLAGS` to pass flags to the C++ compiler. *Solaris Make* defines `CCC` as `CC` by default and uses `CCCFLAGS` to pass flags to the C++ compiler.

[3]Except if a file named clean exists! This can be worked around by using a double colon instead of a single one.

We should therefore never rely on default implicit rules provided by a certain flavor of *make*. Always write a `%.o: %.C` or `%.o: %.cc` pattern rule, depending on which extension you use for your implementation files.

3.9 Compilation failures

It is very hard to write a program at once that is larger than 10 lines and syntactically correct. After *make* starts the compiler, the compiler will emit some error messages and terminate without generating an object file. *Make* can detect the compilation failure by looking at the exit status of the compiler. Whenever a program returns a non-zero exit code, *make* assumes that it failed and will not fork other compilation phases[4].

4 Generating dependencies

The last few lines in figure 10 were used to declare the dependencies. For programs with less than five source files, this is manageable, but for larger programs it is not. Each time the `#include` structure of a program changes, the dependencies change and the `Makefile` has to be changed by hand, which is a nuisance for the programmer and extremely error-prone.

Basically, there are a number of ways to automate the process of keeping dependencies up to date. The one most easy to use (although not very flexible) will be explained in this section. There is a tool `makedepend` that automatically updates the dependencies in our `Makefile`.

```
SOURCES =       my_math.c main.c
OBJECTS =       $(SOURCES:%.c=%.o)

CC =            gcc
CFLAGS =        -O2 -ansi -pedantic -Wall

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

facfib:         $(OBJECTS)
                $(CC) $(OBJECTS)

clean:
                $(RM) $(OBJECTS) facfib core Makefile.bak

depend:
                makedepend $(CFLAGS) $(SOURCES)
```

Figure 12: A modified `Makefile` that uses `makedepend`.

In Figure 12 we specify another target `depend` and omit the hand written dependencies at the bottom lines. If the user types `make depend`, the program `makedepend` runs the preprocessor to find out which dependencies exists for each object file. It keeps a copy of our original `Makefile` in `Makefile.bak` and appends the dependencies to our `Makefile`. Each time we change our `#include` structure, we should run "`make depend`" prior to doing a plain "`make`". An example of what is appended is shown in Figure 13.

[4] Unless `-k` was given as option to `make`, the rule started with a minus sign, or the special target `.IGNORE:` is given in the `Makefile`.

```
# DO NOT DELETE THIS LINE -- make depend depends on it.

my_math.o: my_math.h
main.o: /usr/include/stdio.h /usr/include/sys/feature_tests.h my_math.h
```

Figure 13: A modified `Makefile` that uses `makedepend`.

5 Useful to know ...

5.1 Commonly used *make*-options

`-n` No execution mode.
Make only prints commands, but does not execute them[5].

`-k` Continue building targets as long as possible.
Normally *make* stops if one of the executed commands returns a non-zero exit status, usually because of a compile or link error. With the `-k` option, *make* continues to update other targets that do not depend on the target that failed.

`-s` Silent mode.
Make does not print what it executes.

5.2 Other documentation

- Manual pages: See `man gcc`, `man make` and `man makedepend`.
- GNU Info package: A lot of information about GNU *make* can be found in the GNU Info package about *make*. One version can be found in `/home/se/doc/makeinfo` (type `info -f make.inf`).
- `/usr/local/doc/gmake.dvi`

[5] Only lines with `$(MAKE)` in it are executed to allow recursion into other subdirectories.