Glen Cowan
Royal Holloway, University of London
31 May, 1999

Compiling and linking with gmake

In large software projects, compiling and linking programs can consume a significant amount of time, and the utility program `gmake`, available on Unix systems, can be used to speed things up. `gmake` has been chosen by several major particle physics experiments as the tool for building executable programs, and so you may have to become familiar with it even if you don't care about speed. `gmake` is the GNU version of the Unix program `make`. The complete manual (148 pages!) can be obtained from the GNU web site

`http://www.gnu.org/manual/manual.html`

We will take a look at the most important features of `gmake` with an example, which is a simplified version of the one from Chapter 2 of the GNU Make manual. To use `gmake`, the user must supply a *makefile*, which is a text file containing instructions. We will show how this is done, and then we will refine the makefile in several steps so that the final product can serve as a template for further projects.

A simple example

Suppose we've written a simple C++ program,

```
#include <iostream.h>
#include "goodbye.h"          // contains prototype of function goodbye
void main(){
cout << "Hello, world!" << endl;
goodbye();
}
```

which is in the file `hello.cc`. The function `goodbye` is in the file `goodbye.cc`,

```
#include <iostream.h>
void goodbye(){
cout << "Good-bye, world!" << endl;
}
```

and its prototype,

```
// prototype of function goodbye
void goodbye();
```

is contained in the file `goodbye.h`. First, consider what we would do without `gmake` to build the executable program. To compile the routines individually, we type

```
g++ -c hello.cc
g++ -c goodbye.cc
```

which produces the object files `hello.o` and `goodbye.o`. These are then linked to build an executable program with the command

```
g++ -o hello hello.o goodbye.o
```

where the `-o hello` causes the output file to be named `hello`. Usually we compile and link with the single command

```
g++ -o hello hello.cc goodbye.cc
```

Keep in mind, however, that this is really a short cut for two distinct steps: compiling and linking.

Now suppose we modify `goodbye.cc`. To rebuild the executable program, we really only need to recompile `goodbye.cc`. If, however, we use the single command that combines compiling and linking, then both `hello.cc` and `goodbye.cc` will be recompiled. For programs built out of a large number of source files, it can take a lot of time to recompile everything if only a small number of changes are made. Even if we compile the routines separately, things can get difficult if, for example, we change a header file. We would then have to recompile all of the files in which the modified header file is included, and it is not in general obvious which those might be.

An example with `gmake`

We'll now see how to use `gmake` with the simple example above. The user supplies a special file called a makefile which tells `gmake` which routines depend on which other files. `gmake` figures out which routines need to be recompiled by looking at the modification dates of the files. The user simply types `gmake` (with an optional argument), and `gmake` looks in the current directory for the makefile and figures out what to do.

The first step is to create the makefile. `gmake` looks for the makefile under the names `GNUmakefile`, `makefile`, and `Makefile`, in that order. The authors of `gmake` recommend the name `Makefile` because it will be listed near the top of the directory.

The makefile can contain several different types of statements, the most important of which is called a *rule*. The general format of a rule is:

```
target : dependencies ...
         command
             .
             .
             .
```

The *target* is usually the name of a file we want to produce and the *dependencies* are the other files on which the target depends. On the next line there is a *command*, which must always be preceded by a tab character. A rule can contain more than one command, in which case each should start on a new line and be preceded by a tab. The commands tell `gmake` what to do to produce the target.

In our example above, the makefile will contain the following three rules:

```
hello : hello.o goodbye.o
        g++ -o hello hello.o goodbye.o
hello.o : hello.cc goodbye.h
        g++ -c hello.cc
goodbye.o : goodbye.cc
        g++ -c goodbye.cc
```

where the `g++` commands are preceded by tab characters. To build a certain target, type `gmake` followed by the name of the target, e.g.

```
gmake hello
```

If we type `gmake` without an argument, then the first target listed in the makefile is taken as the default. Now, for example, if we modify the file `goodbye.h` but none of the others, then `gmake` will figure out that `hello.cc` needs to be recompiled, but that `goodbye.cc` does not. We could also type

```
gmake goodbye.o
```

if we just wanted to build the target `goodbye.o` in order to test whether `goodbye.cc` will compile without error.

Refinements

In addition to specifying rules, the statements in makefiles can define variables. This can be useful if certain lists of files are repeated at several places in the makefile. For example, rather than repeating `hello.o goodbye.o` in both the dependencies and the command for the target `hello`, we can define a variable called `objects` to stand for the object files. The makefile would then read

```
objects = hello.o goodbye.o

hello : $(objects)
        g++ -o hello $(objects)
hello.o : hello.cc goodbye.h
        g++ -c hello.cc
goodbye.o : goodbye.cc
        g++ -c goodbye.cc
```

When `gmake` encounters `$(objects)`, it translates the variable inside the parentheses. Other variables could be used to stand for lists of libraries, compiler options, etc.

Another short-cut is to let `gmake` figure out what command should be executed, rather than writing it explicitly. For the target `hello.o`, for example, we want to build a file with the suffix `.o`, which depends on a source file with the suffix `.cc` and a header file ending in `.h`. As long as these conventions are followed, `gmake` will assume that `hello.o` is an object file and that it should compile with the `g++ -c` command. So our improved makefile now reads

```
objects = hello.o goodbye.o

hello : $(objects)
        g++ -o hello $(objects)
hello.o : hello.cc goodbye.h
goodbye.o : goodbye.cc
```

The main use of `gmake` is to compile and link, but in fact the rules can contain any shell commands. A common feature is to include a rule containing commands to clean up the directory by removing the executable and object files. We can call the rule's target `clean`, and since it is not itself a file, it doesn't have to depend on any other files. The makefile now reads

```
objects = hello.o goodbye.o

hello : $(objects)
        g++ -o hello $(objects)
hello.o : hello.cc goodbye.h
goodbye.o : goodbye.cc
clean :
        rm hello $(objects)
```

With this makefile we could have a problem if there existed a file named `clean`, and so the `gmake` authors recommend that we specify that `clean` is a so-called phony target, i.e. it is not a file to be built, but rather a name that simply tells `gmake` to execute a certain set of commands. This is done by including the built-in target `.PHONY` and by making it depend on `clean`, as shown below. In addition, it's recommended to place a hyphen before the `rm` command, which tells `gmake` to keep going even if the files you want to delete don't exist. More details on these sorts of advanced features can be found in the `gmake` manual.

Other features can greatly improve the readability of makefiles. If a line gets too long it can be continued with a backslash. This will happen, for example, if a target depends on many other files. You can always leave blank lines, e.g. to separate logically distinct parts of the file. Comments can be added with the `#` character. Shown below is our final makefile, which can serve as a template for more complicated projects:

```
# A simple makefile

objects = hello.o goodbye.o

hello : $(objects)
        g++ -o hello $(objects)
hello.o : hello.cc goodbye.h
goodbye.o : goodbye.cc

.PHONY : clean
clean :
        -rm hello $(objects)
```