Make - a tutorial

Author: Ben Yoshino (ben@wiliki.eng.hawaii.edu)

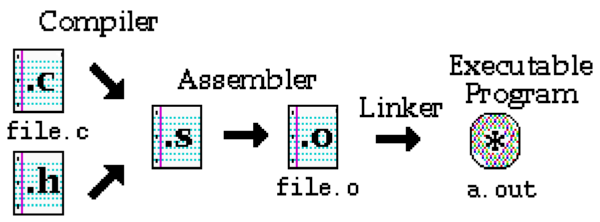
Table of Contents

1 - The Make command

The **make** command allows you to manage large programs or groups of programs. As you begin to write larger programs, you will notice that re-compiling larger programs takes much longer than re-compiling short programs. Moreover, you notice that you usually only work on a small section of the program (such as a single function that you are debugging), and much of the rest of the program remains unchanged.

The **make** program aids you in developing your large programs by keeping track of which portions of the entire program have been changed, compiling only those parts of the program which have changed since the last compile.

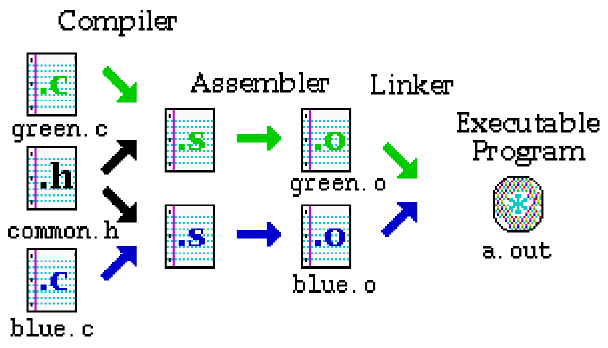
1.1 - A simple compilation



Compiling a small C program requires at least a single *.c* file, with *.h* files as appropriate. Although the command to perform this task is simply `cc file.c`, there are 3 steps to obtain the final executable program, as shown:

1. **Compiler** stage: All C language code in the *.c* file is converted into a lower-level language called Assembly language; making *.s* files.
2. **Assembler** stage: The assembly language code made by the previous stage is then converted into *object code* which are fragments of code which the computer understands directly. An object code file ends with *.o*.
3. **Linker** stage: The final stage in compiling a program involves linking the object code to code libraries which contain certain "built-in" functions, such as `printf`. This stage produces an executable program, which is named *a.out* by default.

1.2 - Compiling with several files

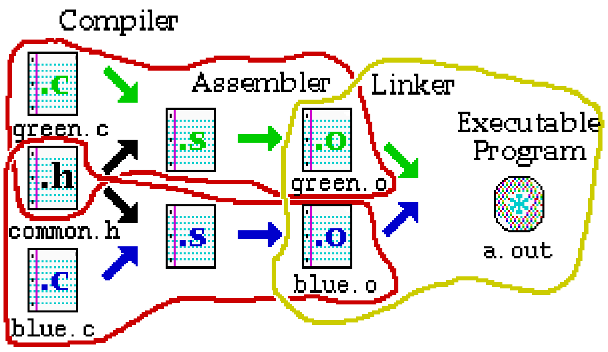


When your program becomes very large, it makes sense to divide your source code into separate easily-manageable *.c* files. The figure above demonstrates the compiling of a program made up of two *.c* files and a single *common.h* file. The command is as follows:

```
cc green.c blue.c
```

where both *.c* files are given to the compiler. Note that the first two steps taken in compiling the files are identical to the previous procedure for a single *.c* file, but the last step has an interesting twist: The two *.o* files are linked together at the **Linker** stage to create one executable program, *a.out*.

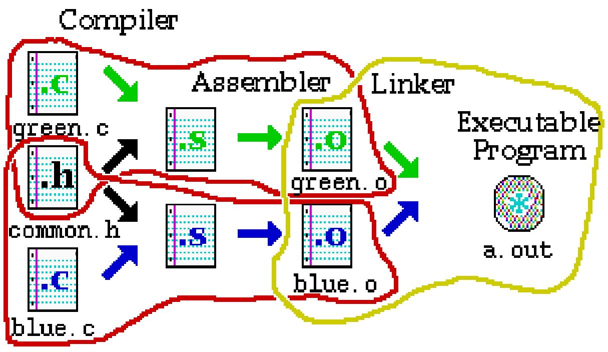
1.3 - Separate compilation



The steps taken in creating the executable program can be divided up in to two compiler/assembler steps circled in red, and one final linker step circled in yellow. The two *.o* files may be created separately, but both are required at the last step to create the executable program.

You can use the `-c` option with **cc** to create the corresponding object (*.o*) file from a *.c* file. For example, typing the command: `cc -c green.c` will not produce an *a.out* file, but the compiler will stop after the assembler stage, leaving you with a *green.o* file.

1.4 - Separate compilation steps



The three different tasks required to produce the executable program are as follows:

- Compile *green.o*: `cc -c green.c`
- Compile *blue.o*: `cc -c blue.c`
- Link the parts together: `cc green.o blue.o`

For example, it is important to note that in order to create the file, *green.o*, the two files, *green.c* and the header file *common.h* are required. Similarly, in order to create the executable program, *a.out*, the object files *green.o* and *blue.o* are required.

1.5 - Splitting your C program

When you separate your C program into many files, keep these points in mind:

- Be sure no two files have functions with the same name in it. The compiler will get confused.
- Similarly, if you use global variables in your program, be sure no two files define the same global variables.
- If you use global variables, be sure only *one* of the files defines them, and declare them in your *.h* as follows: `extern int globalvar;`
- To use functions from another file, make a *.h* file with the function prototypes, and use `#include` to include those *.h* files within your *.c* files.
- At least one of the files **must** have a `main()` function.

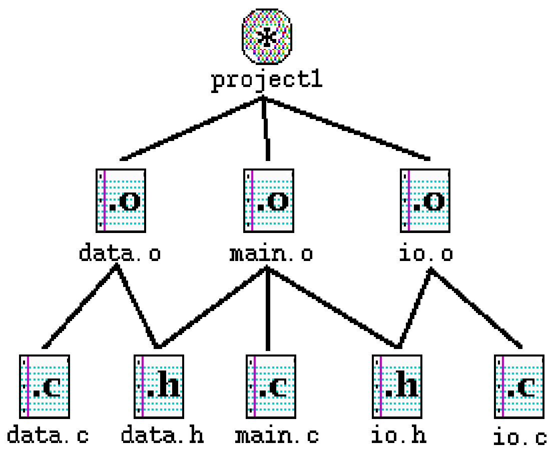
Note: When you **define** a variable, it looks like this: `int globalvar;`. When you **declare** a variable, it looks like this: `extern int globalvar;`. The main difference is that a variable definition creates the variable, while a declaration indicates that the variable is defined elsewhere. A definition implies a declaration.

2 -Dependencies

The principle by which **make** operates was described to you in the last section. It creates programs according to the file dependencies. For example, we now know that in order to create an object file, *program.o*, we require at least the file *program.c*. (There may be other dependencies, such as a *.h* file.)

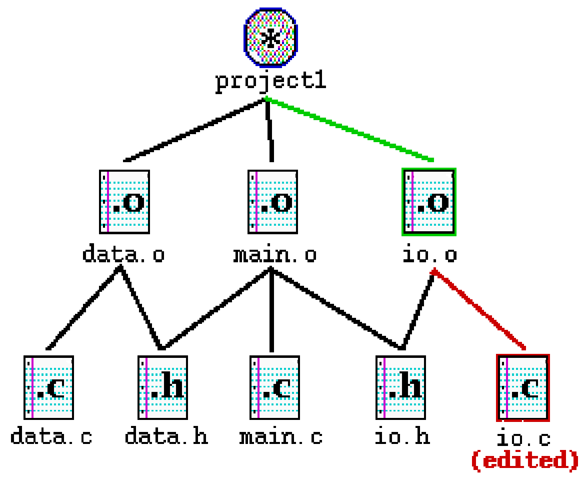
This section involves drawing what are called "dependency graphs", which are very similar to the diagrams given in the previous section. As you become proficient using **make**, you probably will not need to draw these diagrams, but it is important to get a feel for what you are doing.

2.1 - Dependency graphs



This graph shown in the figure is a program which is made up of 5 source files, called *data.c*, *data.h*, *io.c*, *io.h*, and *main.c*. At the top is the final result, a program called *project1*. The lines which radiate downwards from a file are the other files which it depends on. For example, to create *main.o*, the three files *data.h*, *io.h*, and *main.c* are needed.

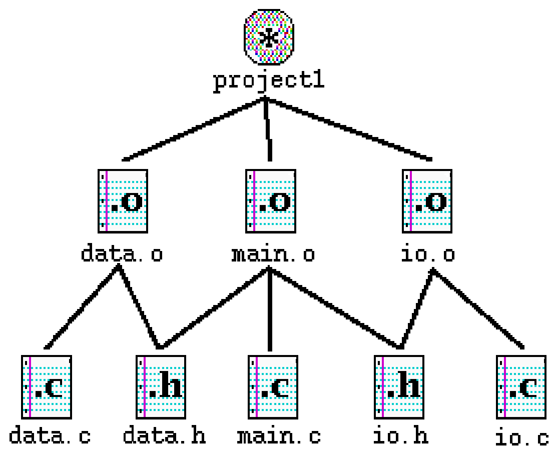
2.2 - How dependency works



Suppose that you have gone through the process of compiling the program, and while you are testing the program, you realize that one function in *io.c* has a bug in it. You edit *io.c* to fix the bug.

The figure above shows *io.c* outlined in red. By going up the graph, you notice that *io.o* needs to be updated because *io.c* has changed. Similarly, because *io.o* has changed, *project1* needs to be updated as well.

2.3 - How does make do it?



```
project1: data.o main.o io.o
    cc data.o main.o io.o -o project1
data.o: data.c data.h
    cc -c data.c
main.o: data.h io.h main.c
    cc -c main.c
io.o: io.h io.c
    cc -c io.c
```

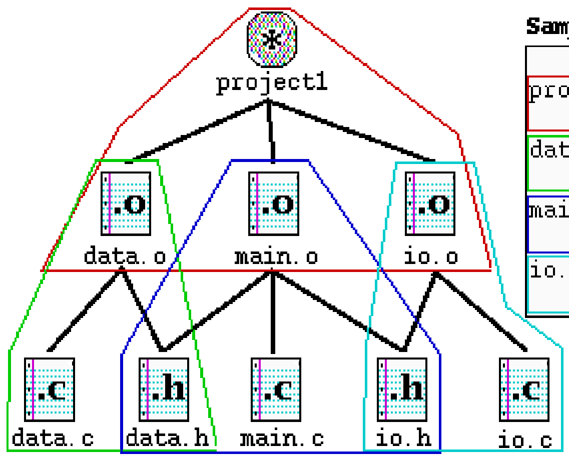
The **make** program gets its dependency "graph" from a text file called *makefile* or *Makefile* which resides in the same directory as the source files. **Make** checks the modification times of the files, and whenever a file becomes "newer" than something that depends on it, (in other words, modified) it runs the compiler accordingly.

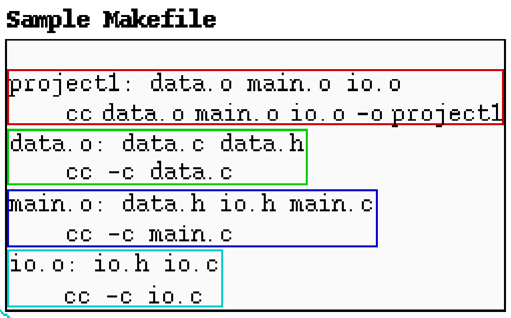
For example, the previous page explained *io.c* was changed. If you edit *io.c*, it becomes "newer" than *io.o*, meaning that **make** must run `cc -c io.c` to create a new *io.o*, then run `cc data.o main.o io.o -o project1` for *project1*.

3 - The Makefile

The previous section described dependencies between files. This section describes the **make** program in more detail by describing the file it uses, called *makefile* or *Makefile*. This file determines the relationships between the source, object and executable files.

3.1 - Translating the dependency graph





Sample Makefile

```
project1: data.o main.o io.o
    cc data.o main.o io.o -o project1
data.o: data.c data.h
    cc -c data.c
main.o: data.h io.h main.c
    cc -c main.c
io.o: io.h io.c
    cc -c io.c
```

Each dependency shown in the graph is circled with a corresponding color in the *Makefile*, and each uses the following format:

target : **source file(s)**
        command *(must be preceded by a tab)*
A target given in the *Makefile* is a file which will be created or updated when any of its source files are modified. The command(s) given in the subsequent line(s) (which **must** be preceded by a *tab* character) are executed in order to create the target file.

3.2 - Listing dependencies

```
project1: data.o main.o io.o
        cc data.o main.o io.o -o project1
data.o: data.c data.h
        cc -c data.c
main.o: data.h io.h main.c
        cc -c main.c
io.o: io.h io.c
        cc -c io.c
```

Note that in the *Makefile* shown above, the *.h* files are listed, but there are no references in their corresponding commands. This is because the *.h* files are referred within the corresponding *.c* files through the `#include "file.h"`. If you do not explicitly include these in your *Makefile*, your program will not be updated if you make a change to your header (*.h*) files.

Note: Comments can be placed in a *Makefile* by placing a pound s ign (#) in front of it.

3.3 - Using the Makefile with make

Once you have created your *Makefile* and your corresponding source files, you are ready to use **make**. If you have named your *Makefile* either *Makefile* or *makefile*, **make** will recognize it. If you do not wish to call your *Makefile* one of these names, you can use `make -f mymakefile`. The order in which dependencies are listed is important. If you simply type `make` and then return, **make** will attempt to create or update the first dependency listed.

You can also specify one of the other targets listed in the *Makefile*, and only that target (and its corresponding source files) would be made. For example, if we typed `make`, the output of **make** would look as follows:

```
% make
        cc -c data.c
        cc -c main.c
        cc -c io.c
        cc data.o main.o io.o -o project1
%
```

When making its targets, **make** first checks the source files and attempts to create or update the source files. That is why *data.o*, *main.o* and *io.o* were created before attempting to create the target: *project1*.

4 - Shortcuts for make

The **make** program has many other features which have not been discussed in previous sections. Most important of these features is the **macro** feature. Macros in **make** work similarly to macros used in C programming. **Make** also has its own pre-defined rules which you can take advantage of to make your *Makefile* smaller.

4.1 - Macros in make

The **make** program allows you to use macros, which are similar to variables, to store names of files. The format is as follows:

```
OBJECTS = data.o io.o main.o
```

Whenever you want to have **make** expand these macros out when it runs, type the following corresponding string `$(OBJECTS)`.

Here is our sample *Makefile* again, using a macro.

```
OBJECTS = data.o main.o io.o
project1: $(OBJECTS)
        cc $(OBJECTS) -o project1
data.o: data.c data.h
        cc -c data.c
main.o: data.h io.h main.c
        cc -c main.c
io.o: io.h io.c
        cc -c io.c
```

You can also specify a macro's value when running **make**, as follows:
`make 'OBJECTS=data.o newio.o main.o' project1`
This overrides the value of `OBJECTS` in THE *Makefile*

4.2 - Special macros

In addition to those macros which you can create yourself, there are a few macros which are used internally by the **make** program. Here are some of those, listed below:

`CC`
    Contains the current C compiler. Defaults to `cc`.
`CFLAGS`
    Special options which are added to the built-in C rule. (See next page.)
`$@`
    Full name of the current target.
`$?`
    A list of files for current dependency which are out-of-date.
`$<`
    The source file of the current (single) dependency.

You can also manipulate the way these macros are evaluated, as follows, assuming that `OBJS = data.o io.o main.o`, using `$(OBJS:.o=.c)` within the *Makefile* substitutes .o at the end with .c, giving you the following result: `data.c io.c main.c`

4.3 - Predefined rules

By itself, **make** knows already that in order to create a *.o* file, it must use `cc -c` on the corresponding *.c* file. These rules are built into **make**, and you can take advantage of this to shorten your *Makefile*. If you just indicate just the *.h* files in the dependency line of the *Makefile* that the current target is dependent on, **make** will know that the corresponding *.c* file is already required. You don't even need to include the command for the compiler.

This reduces our *Makefile* further, as shown:

```
OBJECTS = data.o main.o io.o
project1: $(OBJECTS)
        cc $(OBJECTS) -o project1
data.o: data.h
main.o: data.h io.h
io.o: io.h
```

One thing to consider, however, is that when you are compiling programs on **Wiliki**, you may wish to add a `CFLAGS` macro at the top of your *Makefile* to enable the compiler to use ANSI standard C compilation. The macro looks like this:
`CFLAGS=-Aa -D_HPUX_SOURCE`
This will allow **make** to use ANSI C with the predefined rules.

4.4 - Miscellaneous shortcuts

Although the examples we have shown do not explicitly say so, you can put more than one file in the **target** section of the dependency rules. If a file appears as a target more than once in a dependency, all of its source files are included as sources for that target.

Here is our sample *Makefile* again:

```
CFLAGS = -Aa -D_HPUX_SOURCE
OBJECTS = data.o main.o io.o
project1: $(OBJECTS)
        cc $(OBJECTS) -o project1
data.o main.o: data.h
io.o main.o: io.h
```

This *Makefile* shows *main.o* appearing in two places. **Make** knows by looking at all the dependencies that *main.o* depends on both *data.h* and *io.h*.

5 - Advanced features

Make has some advanced features which we will discuss very briefly in this section. This includes specialized dependency rules, as well as writing *Makefile*s to work with your own suffixes, and writing new "built-in" rules.

5.1 - Special dependencies

Usually, **make** uses the same command to create or update a target, regardless of which file changes. Some other files, such as **libraries** allow users to replace a portion of its code. For this kind of different behavior, **make** allows a special form of the dependency, where the action specified can differ, depending on which file has changed. Here is an example for this rule:

target :: **source1**
        command1
target :: **source2**
        command2

As we have described, if **source1** changes, **target** is created or updated using **command1**; **command2** is used if **source2** is modified instead.

5.2 - Custom suffixes and rules

Make uses a special target, named `.SUFFIXES` to allow you to define your own suffixes. For example, the dependency line:

```
.SUFFIXES: .foo .bar
```

tells **make** that you will be using these special suffixes to make your own rules.

Similar to how **make** already knows how to make a *.o* file from a *.c* file, you can define rules in the following manner:

```
.foo.bar:
        tr '[A-Z][a-z]' '[N-Z][A-M][n-z][a-m]' < $< > $@
```

```
.c.o:
        $(CC) $(CFLAGS) -c $<
```
The first rule allows you to create a *.bar* file from a *.foo* file. (Don't worry about what it does, it basically scrambles the file.) The second rule is the default rule used by **make** to create a *.o* file from a *.c* file.

Make - bibliography

1. Stuart Feldman. Make - A Program for Maintaining Computer Programs. *Software - Practice and Experience*, 9:255-265, 1979.
2. Hewlett-Packard. On-line manual pages for **make**. 1992.