Maemo Diablo GNU Make and makefiles
Training Material

February 9, 2009

Contents

1 GNU Make and makefiles **2**

1.1 What is GNU Make? . . . 2
1.2 How does make work? . . . 3
1.3 The simplest real example . . . 3
1.4 Anatomy of a makefile . . . 6
1.5 The default goal . . . 7
1.6 On names of makefiles . . . 8
1.7 Questions . . . 8
1.8 Adding make goals . . . 8
1.9 Making a target at a time . . . 10
1.10 PHONY-keyword . . . 10
1.11 Specifying the default goal . . . 11
1.12 Other common phony goals . . . 11
1.13 Variables in makefiles . . . 12
1.14 Variable flavors . . . 12
1.15 Recursive variables . . . 12
1.16 Simple variables . . . 13
1.17 Automatic variables . . . 15
1.18 Integrating with pkg-config . . . 16

Chapter 1

GNU Make and makefiles

1.1 What is GNU Make?

Now for a small diversion from our graphical endeavours. We'll introduce a tool that we'll be using from now on to automate our software builds. You'll find it most useful once your project consists of more than one file.

If you already feel comfortable using GNU Make and writing your own Makefiles, feel free to skip this chapter.

The *make* program from the GNU project is a powerful tool to aid implementing automation in your software building process. Beside this, it can be used to automate any task which uses files and in which these files are transformed into some other form. make by itself does not know what the files contain or what they represent, but using simple syntax we can teach make how to handle them.

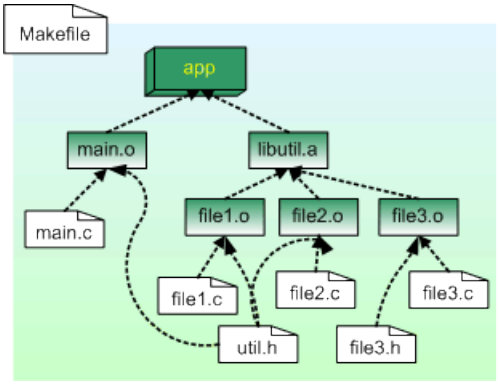
When developing software with gcc (and other tools), you will often invoke gcc repeatedly with the same parameters and flags. After changing one source file, you'll notice that you have to rebuild other object files and the whole application if some interface changed between the functions. This might happen whenever declarations change, new parameters are added to function prototypes and so on.

These tasks could of course be always done manually, but after a while you'll find yourself wondering whether there is some nicer way of doing things.

GNU make is a software building automation tool that will execute repetitive tasks for you. It is controlled via a **Makefile** that contains lists of dependencies between different source files and object files. It also contains lists of commands that should be executed to satisfy these dependencies. make uses the **timestamps** of files and the information of the files' existence to automate rebuilding of applications (targets in make) as well as the rules that we specify in the **Makefile**.

make can be used for other purposes as well. You can easily create a target for installing the built software on destination computer, a target for generating documentation by using some automatic documentation generation tool and so forth. Some people use make to keep multiple Linux systems up to date with the newest software and various system configuration changes. In short, make is flexible enough to be generally useful.

1.2 How does make work?

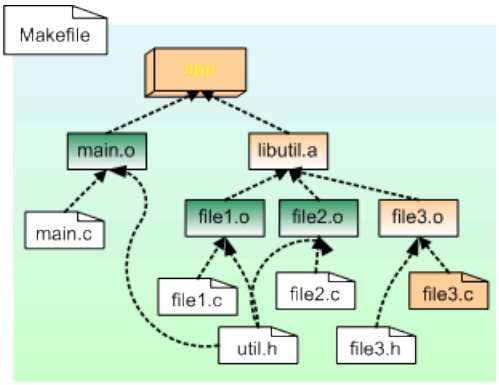


The dependencies between different files making up a software project.

The aim of make is to satisfy the target. Each target has its own dependencies. A user will generally select a target for make to satisfy by supplying the target name on the command line. make will start by checking whether all of the dependencies exist and have an older timestamp than the target. If so, make will do nothing as nothing has changed. However, since a header file (that an application is not dependent on directly) might change, make will propagate the changes to the 'root' of the target as shown in the above picture.

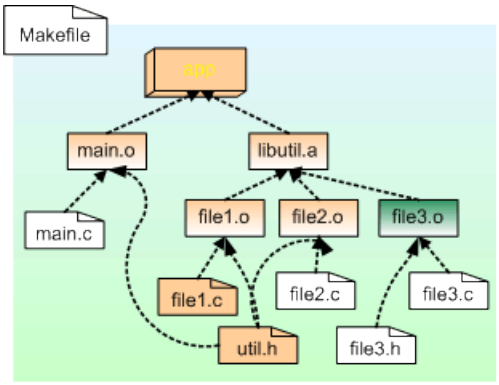
make will rebuild all of the necessary dependencies and targets on the way towards the target. In this way, make will only rebuild those dependencies that actually affect something, and thus, will save time and effort. In big projects the amount of time saved is significant.

To illustrate this, suppose that we modify **file3.c** in the above picture. We then run *make*, and it will automatically rebuild the necessary targets (**file3.o**, **libutil.a** and **app**):



Source file file3.c has been modified. make rebuilds the depending objects and target (**app**)

Now suppose that we add another function to **file1.c**. We would then also need to modify **util.h** accordingly. From the picture you see that quite many objects and targets depend on this header file, so we end up rebuilding a sizable number of objects (but not all):



Header file util.h has been modified

Note that in the example pictures we have a project with a custom static library which will be linked against the test application. Developing custom libraries is beyond the scope of this material but the library scenario will continue to serve as an example.

1.3 The simplest real example

Before delving too deeply into the syntax of makefiles, it is instructive to first see make in action. For this, we'll use a simple project that is written in the C language.

In C, it is customary to write "header" files (conventional suffix for them is **.h**) and regular source files (**.c**). The header files describe calling conventions, APIs and structures that we want to make usable for the outside world. The **.c**-files contain the implementation of these interfaces. This is nothing new to you hopefully.

We'll start with the following rule: if we change something in the interface file then the binary file containing the code implementation (and other files that use the same interface) should be regenerated. Regeneration in this case means invoking *gcc* to create the binary file out of the C-language source file.

We need to tell make two things at this point:

- If a file's content changes, which other files will that affect? Since a single interface will likely affect multiple other files, make uses a reversed ordering here. For each resulting file, we need to list the files on which this one file depends on.
- What are the commands to regenerate the resulting files when the need arises?

We'll do this as simply as possible. We have a project that consists of two source files and one header file. The contents of these files are listed below:

```c
/**
 * The old faithful hello world.
 *
 * This maemo code example is licensed under a MIT-style license,
 * that can be found in the file called "License" in the same
 * directory as this file.
 * Copyright (c) 2007-2008 Nokia Corporation. All rights reserved.
 */

#include <stdlib.h>    /* EXIT_* */
#include "hello_api.h" /* sayhello */

int main(int argc, char **argv) {
  sayhello();

  return EXIT_SUCCESS;
}
```

Listing 1.1: Contents of simple-make-files/hello.c

```c
/**
 * Implementation of sayhello.
 *
 * This maemo code example is licensed under a MIT-style license,
 * that can be found in the file called "License" in the same
 * directory as this file.
 * Copyright (c) 2007-2008 Nokia Corporation. All rights reserved.
 */

#include <stdio.h>     /* printf */
#include "hello_api.h" /* sayhello declaration */

void sayhello(void) {
  printf("Hello world!\n");
}
```

Listing 1.2: Contents of simple-make-files/hello_func.c

```c
/**
 * Interface description for the hello_func module.
 *
 * This maemo code example is licensed under a MIT-style license,
 * that can be found in the file called "License" in the same
 * directory as this file.
 * Copyright (c) 2007-2008 Nokia Corporation. All rights reserved.
 */

#ifndef INCLUDE_HELLO_API_H
#define INCLUDE_HELLO_API_H
/* The above is protection against circular header inclusion. */

/* Function to print out "Hello world!\n". */
extern void sayhello(void);

#endif
/* ifndef INCLUDE_HELLO_API_H */
```

Listing 1.3: Contents of simple-make-files/hello_api.h

So, in effect, we have the main application in**hello.c** which uses a function which is implemented in **hello_func.c** and declared in **hello_api.h**. To build an application out of these files we could do it manually like this:

```
gcc -Wall hello.c hello_func.c -o hello
```

Or, we could do it in three stages:

```
gcc -Wall -c hello.c -o hello.o
gcc -Wall -c hello_func.c -o hello_func.o
gcc -Wall hello.o hello_func.o -o hello
```

In both cases, the resulting application (**hello**) is the same.

In the first case, we instruct gcc to process all source files in one go, link the resulting object code and store that code (program in this case) into **hello**.

In the second case, we instruct gcc to create a binary object file for each of the source files. After that, we instruct gcc to link these object files (**hello.o** and **hello_func.o**) together, and store the linked code into **hello**.

Notice that when gcc reads through the C source files, it will also read in the header files, since the C code uses the `#include` -preprocessor directive. This is because gcc will internally will run all files ending with **.c** through cpp (the preprocessor) first.

The file describing this situation to make is as follows:

```
# define default target (first target = default)
# it depends on 'hello.o' (which will be created if necessary)
# and hello_func.o (same as hello.o)
hello: hello.o hello_func.o
  gcc -Wall hello.o hello_func.o -o hello

# define hello.o target
# it depends on hello.c (and is created from it)
# also depends on hello_api.h which would mean that
# changing the hello.h api would force make to rebuild
# this target (hello.o).
# gcc -c: compile only, do not link
hello.o: hello.c hello_api.h
  gcc -Wall -c hello.c -o hello.o

# define hello_func.o target
# it depends on hello_func.c (and is created from)
# and hello_api.h (since that's its declaration)
hello_func.o: hello_func.c hello_api.h
  gcc -Wall -c hello_func.c -o hello_func.o
```

Listing 1.4: Contents of simple-make-files/Makefile

This file is in the simplest form without using any variables or relying on built-in magic in GNU make. Later we'll see that we could write the same rules in a much shorter way.

You can test this makefile out by running make in the directory which contains the makefile and the source files:

```
user@system:~$ make
gcc -Wall -c hello.c -o hello.o
gcc -Wall -c hello_func.c -o hello_func.o
gcc -Wall hello.o hello_func.o -o hello
```

Running make.

```
user@system:~$ ls -la
total 58
drwxr-xr-x 3  user user 456  Aug 11 15:04 .
drwxr-xr-x 13 user user 576  Aug 11 14:56 ..
-rw-r--r-- 1  user user 699  Jun 1  14:48 Makefile
-rwxr-xr-x 1  user user 4822 Aug 11 15:04 hello
-rw-r--r-- 1  user user 150  Jun 1  14:48 hello.c
-rw-r--r-- 1  user user 824  Aug 11 15:04 hello.o
-rw-r--r-- 1  user user 220  Jun 1  14:48 hello_api.h
-rw-r--r-- 1  user user 130  Jun 1  14:48 hello_func.c
-rw-r--r-- 1  user user 912  Aug 11 15:04 hello_func.o
```

The resulting files after running make.

```
user@system:~$ ./hello
Hello world!
```

Executing the binary.

1.4 Anatomy of a makefile

From the simple example above, we can deduce some of make's syntax.

Here are the rules that we can learn:

- Comments are lines that start with the #-character. To be more precise, when make reads the makefile, it will ignore the #-character and any characters after it up to the end of the line. This means that we could also put comments at the end of lines and make would ignore them, but this is considered bad practise as it would lead to subtle problems later on.
- The backslash character(\ ) can be used to escape the special meaning of next character. The most important special character in makefiles is the dollar-character($), which is used to access the contents of variables. There are also other special characters. To continue a line that is too long to your liking, you may escape the newline character on that line. Place the backslash at the end of your line. make will then ignore the newline when reading input.
- Empty lines by themselves are ignored.
- A line that starts at column 0 (start of the line) and contains a colon character(`:`) is considered a rule. The name on the left side of the colon is created by the commands. This name is called a target. Any filenames specified after the colon are the files that the target depends on. They are called prerequisites (i.e. they're required to exist before make decides to create the target.
- Lines starting with the tabulation character(`tab`) are commands that make will run to achieve the target.

In the printed material the tabs are represented with whitespace, so be careful when reading the example makefiles. Note also that in reality, the command lines are considered as part of the rule.

Using these rules, we can now deduce that:

- make will regenerate **hello** when either or both of its prerequisites change. So, if either **hello.o** or **hello_func.o** change, make will regenerate **hello** by using the command: **gcc -Wall hello.o hello_func.o -o hello**
- If either **hello.c** or **hello_api.h** change, make will regenerate **hello.o**.
- If either **hello_func.c** or **hello_api.h** change, make will regenerate **hello_func.o**.

1.5 The default goal

Note that nowhere do we explicitly tell make what it should do by default when run without any command line parameters (as we did above). How does it know that creating **hello** is the target for the run?

The first target given in a makefile is the default target. A target that achieves some higher-level goal (like linking all the components into the final application) is sometimes called a goal in GNU make documentation.

So, the first target in the makefile is the default goal when make is run without command line parameters.

Note that in a magic-like way, make will automatically learn the dependencies between the various components and deduce that in order to create **hello**, it will also need to create **hello.o** and **hello_func.o**. make will regenerate the missing prerequisites when necessary. In fact, this is a quality that causes make do its magic.

Since the prerequisites for **hello** (**hello.o** and **hello_func.o**) don't exist and **hello** is the default goal, make will first create the files that the **hello**-target needs. You can evidence this from the screen capture of make running without command line parameters. In it, you see the execution of each command in the order that make decides is necessary to satisfy the default goal's prerequisites and finally it will create the **hello**.

```
user@system:~$ make
gcc -Wall -c hello.c -o hello.o
gcc -Wall -c hello_func.c -o hello_func.o
gcc -Wall hello.o hello_func.o -o hello
```

1.6 On names of makefiles

The recommended name for your makefile is **Makefile**. This is not the first filename that GNU make will try to open, but it is the most portable one. In fact, the order of filenames that make attempts to open is: **GNUMakefile**, **Makefile** and finally **makefile**.

Unless you're sure that your makefile won't work on other make systems (not GNU), refrain from using **GNUMakefile**. We will be using **Makefile** for the most of this material. The idea behind using **Makefile** instead of **makefile** is related to how shells and commands sort filenames when contents of directories are requested. Since in ASCII the capital letters come before the small case letters, the important files are listed first. Makefiles are considered important since they're the basis for building the project. (The collation order might be different in your locale and your environment.)

You may tell make which file to read explicitly by using the `-f` command line option. This option will be used in the next example.

1.7 Questions

Based on common sense, what should make do when we run it after:

- Deleting the **hello** file?
- Deleting **hello.o** file?
- Modifying **hello_func.c**?
- Modifying **hello.c**?
- Modifying **hello_api.h**?
- Deleting the **Makefile**?

Think about what would you do if you'd be writing the commands manually.

1.8 Adding make goals

Sometimes it's useful to add targets whose execution doesn't result in a file but instead cause some commands to be run. This is commonly used in makefiles of software projects to get rid of the binary files, so that building can be attempted from a clean state. This kind of targets can also be justifiably called goals. GNU documentation uses the name "phony target" for these kinds of targets since they don't result in creating a file like normal targets do.

We'll next create a copy of the original makefile and add a goal that will remove the binary object files and the application. Note that other parts of the makefile do not change.

```
# add a cleaning target (to get rid of the binaries)

# define default target (first target = default)
# it depends on 'hello.o' (which must exist)
# and hello_func.o
hello: hello.o hello_func.o
  gcc -Wall hello.o hello_func.o -o hello

# define hello.o target
# it depends on hello.c (and is created from)
# also depends on hello_api.h which would mean that
# changing the hello.h api would force make to rebuild
# this target (hello.o).
# gcc -c: compile only, do not link
hello.o: hello.c hello_api.h
  gcc -Wall -c hello.c -o hello.o

# define hello_func.o target
# it depends on hello_func.c (and is created from)
# and hello_api.h (since that's it's declaration)
hello_func.o: hello_func.c hello_api.h
  gcc -Wall -c hello_func.c -o hello_func.o

# This is the definition of the target 'clean'
# Here we'll remove all the built binaries and
# all the object files that we might have generated
# Notice the -f flag for rm, it means "force" which
```

```
# in turn means that rm will try to remove the given
# files, and if there are none, then that's ok. Also
# it means that we have no writing permission to that
# file and have writing permission to the directory
# holding the file, rm will not then ask for permission
# interactivelly.
clean:
  rm -f hello hello.o hello_func.o
```

Listing 1.5: Contents of simple-make-files/Makefile.2

In order for make to use this file instead of the default Makefile, we need to use the `-f` command line parameter as follows:

```
user@system:~$ make -f Makefile.2
gcc -Wall -c hello.c -o hello.o
gcc -Wall -c hello_func.c -o hello_func.o
gcc -Wall hello.o hello_func.o -o hello
```

We tell make to use Makefile.2 instead of the default one.

To control which target make will pursuit (instead of the default goal), we need to give the target name on the command line like this:

```
user@system:~$ make -f Makefile.2 clean
rm -f hello hello.o hello_func.o
```

Test the `clean` target.

```
user@system:~$ ls -la
total 42
drwxr-xr-x 3  user user 376  Aug 11 15:08 .
drwxr-xr-x 13 user user 576  Aug 11 14:56 ..
-rw-r--r-- 1  user user 699  Jun 1  14:48 Makefile
-rw-r--r-- 1  user user 1279 Jun 1  14:48 Makefile.2
-rw-r--r-- 1  user user 150  Jun 1  14:48 hello.c
-rw-r--r-- 1  user user 220  Jun 1  14:48 hello_api.h
-rw-r--r-- 1  user user 130  Jun 1  14:48 hello_func.c
```

No binary files remain in the directory.

1.9 Making a target at a time

Sometimes it's useful to ask make to process only one target. Similar to the `clean`-case, we just need to give the target name on the command line:

```
user@system:~$ make -f Makefile.2 hello.o
gcc -Wall -c hello.c -o hello.o
```

Making a single target.

We can also supply multiple target names as individual command line parameters:

```
user@system:~$ make -f Makefile.2 hello_func.o hello
gcc -Wall -c hello_func.c -o hello_func.o
gcc -Wall hello.o hello_func.o -o hello
```

Making two targets at a time.

You can do this with any number of targets, even phony ones. make will try to complete all of them in the order that they're on the command line. Should any of the commands within the targets fail, make will abort at that point, and will not pursue the rest of the targets.

1.10 PHONY-keyword

Suppose that for some reason or another, a file called **clean** appears in the directory in which we run make with **clean** as the target. In this case, make would probably decide that since **clean** already exists, there's no need to run the commands leading to the target, and in our case make would not run the **rm** command at all. Clearly this is something we want to avoid.

For these cases, GNU make provides a special target called **.PHONY** (note the leading dot). We list the real phony targets (clean) as a dependency to **.PHONY**. This will signal to make that it should never consider a file called **clean** to be the result of the phony target.

In fact, this is what most open source projects that use make do.

This leads in the following makefile (comments omitted for brevity):

```
hello: hello.o hello_func.o
  gcc -Wall hello.o hello_func.o -o hello

hello.o: hello.c hello_api.h
  gcc -Wall -c hello.c -o hello.o

hello_func.o: hello_func.c hello_api.h
  gcc -Wall -c hello_func.c -o hello_func.o

.PHONY: clean
clean:
  rm -f hello hello.o hello_func.o
```

Listing 1.6: Using .PHONY (simple-make-files/Makefile.3)

1.11 Specifying the default goal

We know that make will use the first target in a makefile as its default goal. What if we explicitly want to set the default goal instead? Since it is not possible to actually change the "first target is the default goal"-setting in make, we will have to take this into account. So, we'll just add a new target and make sure that it will be processed as the first target in a makefile.

In order to achieve this, we create a new phony target and list the wanted targets as the phony target's prerequisites. You can use whatever name you want for the target but `all` is a very common name for this use. The only thing you need to remember is that this target needs to be the first one in the makefile.

```
.PHONY: all
all: hello

hello: hello.o hello_func.o
  gcc -Wall hello.o hello_func.o -o hello
```

```
hello.o: hello.c hello_api.h
  gcc -Wall -c hello.c -o hello.o

hello_func.o: hello_func.c hello_api.h
  gcc -Wall -c hello_func.c -o hello_func.o

.PHONY: clean
clean:
  rm -f hello hello.o hello_func.o
```

Listing 1.7: Explicit default goal (simple-make-files/Makefile.4)

You will notice something peculiar in the listing above. Since the first target is the default goal for make, won't `.PHONY` now be the default target? Since `.PHONY` is a special target in GNU make, we're safe. Also, because of compatibility reasons, GNU make will ignore any targets that start with a leading dot (.) when looking for the default goal.

1.12 Other common phony goals

You will encounter many other phony goals in makefiles that projects use.

Some of the more common ones include:

- `install`: Prerequisites for `install` is the software application binary (or binaries). The commands (normally *install* is used on Linux) will specify which binary file to place under which directory on the system (`/usr/bin`, `/usr/local/bin`, etc).
- `distclean`: Similar to `clean`, remove object files and such, but remove other files as well (sounds scary). Normally this is used to implement the removal of **Makefile** in the case that it was created by some automatic tool (*configure* for example).
- `package`: prerequisites are as in `install`, but instead of installing, create an archive file (*tar*) with the files in question. This archive can then be used to distribute the software.

You can find other common phony targets in the GNU make manual (not the man-page!).

1.13 Variables in makefiles

So far our files have been listing filenames explicitly and hopefully you've noticed that writing makefiles in this way can get rather tedious if not error prone.

This is why all make programs (not just the GNU variant) provide variables. Some other make programs call them macros.

Variables work almost as they do inside regular UNIX command shells. They are a simple mechanism by which we can associate a piece of text with a name that can be referenced later on in multiple places in our makefiles. make variables are based on text-replacement just like shell variables are.

1.14 Variable flavors

The variables that we can use in makefiles come in two basic styles or flavors.

The default flavor is where referencing a variable will cause make to expand the variable contents at the point of reference. This means that if the value of the variable is some text in which we reference other variables, their contents will also be replaced automatically. This flavor is called `recursive`.

The other flavor is `simple`, meaning that the content of a variable is evaluated only once, when we define the variable.

Deciding on which flavor to use might be important when the evaluation of variable contents needs to be repeatable and fast. In these cases simple variables are often the correct solution.

1.15 Recursive variables

Here are the rules for creating recursive variables:

- Names must contain only ASCII alphanumerics and underscores (to preserve portability).
- You should use only lower case letters in the names of your variables. make is case sensitive and variable names written in capital letters are used when we want to communicate the variables outside make or their origin is from outside (environment variables). This is however a convention only, and you will also see variables that are local to the makefile, but still written in all capital letters.
- Values may contain any text. The text will be evaluated by make when the variable is used. Not when it's defined.
- Long lines may be broken up with a backslash character ( ) and newline combination. This same mechanism can be used to continue other long lines as well (not just variables). Do not put any whitespace after the backslash.
- Do not reference the variable that you're defining in its text portion. This would result in an endless loop whenever this variable would be used. GNU make will stop on error in this case.

We'll re-use the same makefile as before, but introduce some variables. You will also see the syntax on how to define the variables and how to use them (reference them):

```
# define the command that we use for compilation
CC = gcc -Wall

# which targets do we want to build by default?
# note that 'targets' is just a variable, its name
# does not have any special meaning to make
targets = hello

# first target defined will be the default target
# we use the variable to hold the dependencies
```

```
.PHONY: all
all: $(targets)

hello: hello.o hello_func.o
  $(CC) hello.o hello_func.o -o hello

hello.o: hello.c hello_api.h
  $(CC) -c hello.c -o hello.o

hello_func.o: hello_func.c hello_api.h
  $(CC) -c hello_func.c -o hello_func.o

# we'll make our cleaning target more powerful
# we remove the targets that we build by default and also
# remove all object files
.PHONY: clean
clean:
  rm -f $(targets) *.o
```

Listing 1.8: The makefile with some variables (simple-make-files/Makefile.5)

The `CC` variable is the standard variable to hold the name of the C-compiler executable. GNU make comes pre-loaded with a list of tools which can be accessed in similar way (we'll see `$(RM)` shortly, but there are others). Most UNIX-tools related to building software already have similar pre-defined variables in GNU make. Here we override one of them for no other reason than to demonstrate how it's done. Overriding variables like this is not recommended since the user running make later might want to use some other compiler, and would have to edit the makefile to do so.

1.16 Simple variables

Suppose you have a makefile like this:

```
CC = gcc -Wall

# we want to add something to the end of the variable
CC = $(CC) -g

hello.o: hello.c hello_api.h
  $(CC) -c hello.c -o hello.o
```

Listing 1.9: The makefile with some variables (simple-make-files/Makefile.5)

What might seem quite logical to a human reader, won't seem very logical to make.

Since the contents of recursive variables are evaluated when they're referenced, you will notice that the above fragment will lead to an infinite loop.

How do we correct this? make actually provides two mechanisms for this. This is the solution with simple variables:

```
CC := gcc -Wall

# we want to add something to the end of the variable
CC := $(CC) -g

hello.o: hello.c hello_api.h
```

```
  $(CC) -c hello.c -o hello.o
```

Listing 1.10: The makefile with some variables (simple-make-files/Makefile.5)

Notice that the equals character(=) has been changed into `:=` .

This is how we create simple variables. Whenever we reference a simple variable, make will just replace the text that is contained in the variable without evaluating it. It will do the evaluation only when we define the variable. In the above example, `CC` is created with the content of `gcc -Wall` (which is evaluated, but is just plain text) and when we next time define the `CC`-variable, make will evaluate `$(CC) -g` which will be replaced with `gcc -Wall -g` as one might expect.

So, the only two differences between the variable flavors are:

- We use `:=` when defining simple variables.
- make evaluates the contents when the simple variable is defined and not when it's referenced later.

Most of the time you'll want to use the recursive variable flavor since it does what you want.

There was a mention about two ways of appending text to an existing variable. The other mechanism is the += operation as follows:

```
CC = gcc -Wall

# we want to add something to the end of the variable
CC += -g

hello.o: hello.c hello_api.h
  $(CC) -c hello.c -o hello.o
```

Listing 1.11: The makefile with some variables (simple-make-files/Makefile.5)

You can use the prepending operation with simple variables too. make will not change the type of variable on the left side of the += operator.

1.17 Automatic variables

There is a predefined set of variables inside make that can be used to avoid repetitive typing when writing out the commands in a rule.

This is a list of the most useful ones:

- `$<` : replaced by first prerequisite of the rule.
- `$^` : replaced by the list of all prerequisites of the rule.
- `$@` : replaced by the target name.
- `$?` : list of prerequisites that are newer than the target is (if target doesn't exist, they are all considered newer).

By rewriting our makefile using automatic variables we get:

```
# add the warning flag to CFLAGS-variable
CFLAGS += -Wall

targets = hello

.PHONY: all
all: $(targets)

hello: hello.o hello_func.o
  $(CC) $^ -o $@

hello.o: hello.c hello_api.h
  $(CC) $(CFLAGS) -c $< -o $@

hello_func.o: hello_func.c hello_api.h
  $(CC) $(CFLAGS) -c $< -o $@

.PHONY: clean
clean:
  $(RM) $(targets) *.o
```

Listing 1.12: Contents of simple-make-files/Makefile.9

Note the cases when we use `$^` instead of `$<` in the above snippet. We don't want to pass the header files for compilation to the compiler since the source file already includes the header files. For this reason we use `$<`. On the other hand, when we're linking programs and have multiple files to put into the executable, we'd normally use `$^`.

We're relying on a couple of things here:

- `$(RM)` and `$(CC)` will be replaced with paths to the system compiler and removal commands.
- `$(CFLAGS)` is a variable that contains a list of options to pass whenever make will invoke the C compiler.

You might also notice that all these variable names are in capital letters. This signifies that they have been communicated from the system environment to make. In-fact, if you create an environment variable called `CFLAGS` make will create it internally for any makefile that it will process. This is the mechanism to communicate variables into makefiles from outside.

Writing variables in all capital letters is a convention signalling external variables or environmental variables, so this is the reason why you should use lower case letters in your own private variables within a **Makefile**.

1.18 Integrating with pkg-config

You now have the knowledge to write a simple **Makefile** for the Hello World example. In order to get the the result of *pkg-config*, we will use the GNU make `$(shell command params)`-function. Its function is similar to the backtick operation of the shell.

```
# define a list of pkg-config packages we want to use
pkg_packages := gtk+-2.0 hildon-1
```

```
# get the necessary flags for compiling
PKG_CFLAGS := $(shell pkg-config --cflags $(pkg_packages))
# get the necessary flags for linking
PKG_LDFLAGS := $(shell pkg-config --libs $(pkg_packages))

# additional flags
# -Wall: warnings
# -g: debugging
ADD_CFLAGS := -Wall -g

# combine the flags (so that CFLAGS/LDFLAGS from the command line
# still work).
CFLAGS  := $(PKG_CFLAGS) $(ADD_CFLAGS) $(CFLAGS)
LDFLAGS := $(PKG_LDFLAGS) $(LDFLAGS)

targets = hildon_helloworld-1

.PHONY: all
all: $(targets)

hildon_helloworld-1: hildon_helloworld-1.o
  $(CC) $^ -o $@ $(LDFLAGS)

hildon_helloworld-1.o: hildon_helloworld-1.c
  $(CC) $(CFLAGS) -c $< -o $@

.PHONY: clean
clean:
  $(RM) $(targets) *.o
```

Listing 1.13: Contents of simple-make-files/Makefile.10

You might be wondering where the `CC` and `RM` variables come from. We certainly didn't declare them anywhere in the Makefile. GNU make comes with a list of predefined variables for many programs and their arguments. GNU make also contains a preset list of pattern rules (which we won't be going in here), but basically these are predefined rules that are used when (for example) one needs an `.o` file from an `.c` file. The rules that we specified manually above are actually the same rules that GNU make already contains, so we can make our **Makefile** even more compact by only specifying the dependencies between files.

This leads to the following, slightly simpler version:

```
# define a list of pkg-config packages we want to use
pkg_packages := gtk+-2.0 hildon-1

PKG_CFLAGS := $(shell pkg-config --cflags $(pkg_packages))
PKG_LDFLAGS := $(shell pkg-config --libs $(pkg_packages))

ADD_CFLAGS := -Wall -g

# combine the flags
CFLAGS  := $(PKG_CFLAGS) $(ADD_CFLAGS) $(CFLAGS)
LDFLAGS := $(PKG_LDFLAGS) $(LDFLAGS)

targets = hildon_helloworld-1

.PHONY: all
all: $(targets)
```

```
# we can omit the rules and just specify the dependencies
# for our simple project this doesn't seem like a big win
# but for larger projects where you have multiple object
# files, this will save considerable time.
hildon_helloworld-1: hildon_helloworld-1.o
hildon_helloworld-1.o: hildon_helloworld-1.c

.PHONY: clean
clean:
  $(RM) $(targets) *.o
```

Listing 1.14: Contents of simple-make-files/Makefile.11

An up-to-date manual for GNU make can be found in gnu.org