The Linux Programming Environment

Computer Science Department
Texas State University

TUTORIAL REQUIREMENTS 1

INTRODUCTION 1

COMPILATION 1

COMMAND LINE COMPILATION 1
Basic Compilation 2
The Process: Intermediate Files 2
Compile Flags 3
Dependency 4
COMPILATION USING MAKE 5
Basic Make and Makefiles 5
Common Errors 7
Using Variables 8

DEBUG AND TESTING 9

COMPILING YOUR PROGRAM FOR GDB: 10
Important Commands for Running GDB: 10
GCOV 11
Compiling your program for gcov: 11
Running gcov on your program: 11
gcov output interpretation: 11
GPROF 12
Compiling your program for gprof: 12
Running gprof: 12
DOS2UNIX AND UNIX2DOS 12
Running dos2unix or unix2dos: 12

REFERENCES: 14

APPENDIX: 15

A MORE COMPLICATED MAKEFILE 15
ADDITIONAL INFORMATION ON MAKEFILES 15
Variable Flavors 15
Advanced Variable Use 16

Tutorial Requirements

This tutorial assumes that participants have the following basic requirements.

- Computer Science Linux account
- Familiarity with the basic Linux command line environment
- Some experience programming with the gcc and g++ compilers

Introduction

Computer Science classes that teach programming focus on learning to construct well written and well designed programs. The languages that are taught in these classes are part of the tools that are used to implement the programming principles that students are taught. This tutorial is not intended to provide instruction about how to write programs but to help introduce students to tools available to help them implement working programs.

In this tutorial we will cover the major aspects of software development in the Linux environment that students are likely to need under the Computer Science program at Texas State University.

This includes:

- Compilation
  - Command line compilation for C/C++
  - Compilation using *make*
- Debug and Testing
  - The gnu debugger (*gdb*)
  - *gcov*
  - *gprof*
  - *dos2unix* and *unix2dos*

Compilation

Compiling C/C++ programs in the Linux environment will be broken down into two major sections

- simple compilation from the command line using the gcc or g++ compiler
- compiling more complicated projects using the Unix "make" utility.

We will use the following notational conventions:

- Files names are *italicized*.
- Code snippets are in `Courier font` and indented
- Linux terminal commands are in `Courier font` and are preceded by a dollar sign, `$`, representing the shell prompt.

Command Line Compilation

Part of a programmer's job is to turn one or more source files into an executable program file, often referred to as a "binary" or an "executable". The process must change commands written in human readable form into a form that can be understood and executed by the computer system. Compilers are one of the software tools used to perform this translation of source file to binary file. A source file is a plain text file (no formatting codes such as are found when using word processors) which contains codes written in a particular language. For the purposes of this tutorial we will utilize the C/C++ programming language. Files containing C code normally have a file extension of *.c* and files containing C++ code normally have a *.cpp* extension, while header files always have a *.h* extension. Below are some examples of file names and types.

utils.h, calc.h - C/C++ header files

foo.c, calc.c, main.c - C source files

foo.cpp, calc.cpp, main.cpp - C++ source files

It is important to remember that source files should always be plain text, which means no rtf (.rft), html (.htm), or god forbid word documents (.doc). The compiler is expecting text with no extra embedded formatting characters.

Basic Compilation

We will start with the simplest case in which there is one source file with no header files that we wish to compile into an executable program. If you have not already done so, please log in to your account and follow along. If you are using the graphical interface you will need to open a terminal window.

Make a directory in which to work using the following command:

```
$ mkdir comp_tutorial
```

Now `cd` into this directory and use your favorite Linux editor to enter the following:

```
#include <iostream>
using namespace std;
int main()
{
    cout << "hello world" << endl ;
return 0;
}
```

Save this file as *hello.cpp*. Now open another terminal window and `cd` to your comp_tutorial directory. You will find it more convenient to keep the editor open while you are working.

Enter the following command:

```
$ g++ hello.cpp
```

Now type `ls` … You should see a file named `a.out` in the comp_tutorial directory. This is the executable program file. It can be run with the following command:

```
$ ./a.out
```

If you do not supply a filename for your executable GCC (gcc or g++) will always name your file `a.out`. So lets supply a name using the `-o` flag.

```
$ g++ hello.cpp -o hello
```

Now we find an additional executable file in our directory named `hello`. It is always advisable to place this flag at the end of your command rather than the beginning since it is possible to accidentally leave out the name and thus name the executable the same as your first source file, thus permanently overwriting your source file with the executable.

What if your program contains more than one source file? In that case, simply include all the source files in the command:

The following is an example for reference only [do not type in]:

```
g++ hello.cpp util.cpp util2.cpp -o hello
```

Generally speaking, header files (*.h*) are not included in the list of source files but may be necessary for proper compilation.

This seems pretty simple so far but we have not covered the whole story. Let's take a closer look at what the compiler is actually doing.

The Process: Intermediate Files

In order for the C/C++ compiler to turn our source files into an executable program file it must go through four major stages:

1. Pre-process -- strip out comments, expand #define macros, #include lines, etc.
2. Compile -- parse the source code and build assembly output.
3. Assemble -- take the assembly code and build an object file out of it.
4. Link -- build an executable file by linking together object files.

So what we commonly refer to as "compile" is actually a series of steps. Likewise, what we commonly refer to as a compiler is a suite of programs all working together.

The compiler can be stopped at any stage using the appropriate flag:

-E -- stops after the preprocessing stage. It outputs source code after preprocessing to standard out (the terminal).

-S -- stops after the compile stage. It outputs the assembly for each source file to a file of the same name but with a .s extension.

-c -- stops after the assemble stage. It outputs an object file for each source file with the same name but with an ".o" extension.

Stopping after the assemble stage using a -c flag is by far the most common as we shall soon see. In fact compilation is commonly broken into two phases compile and link.

Example:

Let's explore this by stopping at each stage for our simple *hello.cpp* program. Change to the directory containing *hello.cpp* and type the following:

```
$ g++ -E hello.cpp
```

The output you see is our hello program with the `#include<iostream>` statement replaced by the source code that makes up the iostream library. As you can see the library is quite large.

Next type in the following:

```
$ g++ -S hello.cpp
```

[instructor note: use up arrow] …

This should create a file named *hello.s*. Let's take a look at this file using `less`.

```
$ less hello.s
```

We are looking at the assembly code created by the compiler. Use `q` to quit `less`. Now we'll create an object file.

```
$ g++ -c hello.cpp
```

This should create a file named *hello.o*. This object file is machine code (binary) and cannot be viewed using `less`. But we can view it using `hexdump` which converts the binary numbers into hexadecimal numbers that can be viewed by humans.

```
$ hexdump hello.o
```

Now link the object file into an executable program file. This of course is a trivial example since we only have one object file, but here goes.

```
$ g++  hello.o -o hello
```

This should produce an executable named *hello*. We can view it using `hexdump` to see that the content is different from *hello.o* and we can run *hello* to verify that it is an executable.

```
$ hexdump hello
$ ./hello
```

Compile Flags

There are also many other useful compiler flags that can be supplied. We will cover some of the more important ones here but others can be found using `$ man g++`.

-g - includes debug symbols

This flag must be specified to debug programs using gdb.

-Wall - show all compiler warnings.

This flag is useful for finding problems that are not necessarily compile errors. It tells the compiler to print warnings issued during compile which are normally hidden.

-O or **–O1** - optimizes programs.

This tells the compiler to reduce the code size and execution time of the program it produces. This may cause unexpected behavior when run in debug, since GCC is taking every opportunity to eliminate temporary variables and increase efficiency. The process generally takes much more time and memory for compilation, but the resulting executable may be drastically faster.

-O2 – optimize even more.

This performs almost all supported optimizations except loop unrolling, function inlining and register renaming.

-O3 – all optimizations

-pg – generates extra code to write profile information suitable for the analysis program "gprof".

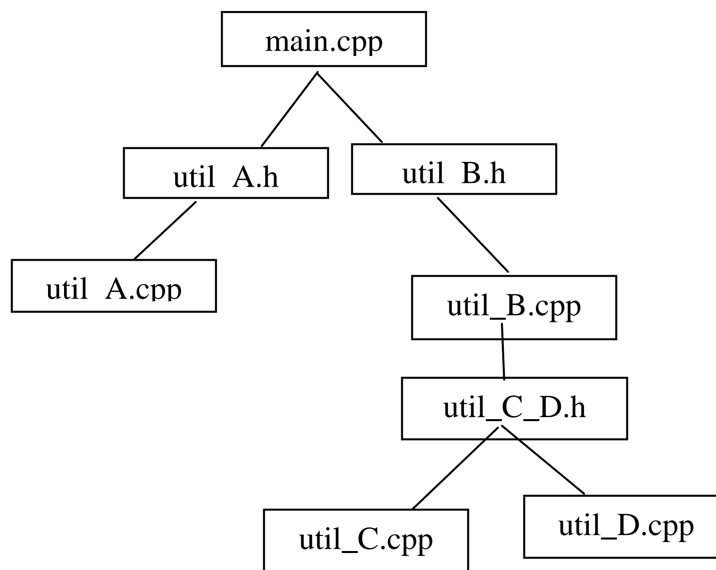
-fprofile-arcs – used to generate coverage data or for profile-directed block ordering.

During execution the program records how many times each branch is executed and how many times a branch is taken. When the compiled program exits it saves this data to a file called *sourcename.da* for each source file.

-ftest-coverage - create data files for the “gcov” code-coverage utility.

Dependency

Very often, a C/C++ project of even a moderate size will have a complex interdependency between source files and libraries. An interdependency occurs when source code in one source file references one or more objects declared or defined in another source file. For this reason the order in which we compile and link our files can be very important. Let’s say that we have five source files *main.cpp*, which contains our main function and four other files *util_A.cpp*, *util_B.cpp*, *util_C.cpp* and *util_D.cpp*. These files have the dependencies illustrated in the following graph.



Linux and Unix systems have a built-in in tool available to manage this complexity called make.

Compilation Using Make

Basic Make and Makefiles

The make utility is designed to manage large groups of source files. It automatically determines which pieces of a large program need to be recompiled, and issues the commands to recompile them. The make utility uses a file usually named makefile or Makefile to represent the dependencies between source files. These files consist of a list of rules which execute the given command when called. They are formatted as follows:

rule_name: [list of rule dependencies] [list of file dependencies]

[tab] [command]

rule name	This can be anything but is generally a source file name with a ".o" extension
list of rule dependencies	A list of the other rules on which this rule is dependent.
list of file dependencies	A list of files which must be present to execute this rule.
Tab	The tab must be present for the command to execute
Command	The command to execute.

We execute these rules using the `make` utility. **`make`** called with no arguments executes the first rule in the Makefile. `make` followed by a rule name executes that rule.

A seemingly trivial but important feature of makefiles is that comments can be added which allowing you to document various facts about compiling the project. Comments are denoted by the pound sign, `#`.

Let's try a trivial example using the *hello.cpp* file we created earlier. Using your favorite Linux editor create a file named *makefile* containing the following:

```
# our first makefile
hello: hello.cpp
      g++ hello.cpp –o hello
```

Now remove the old *hello* and `make` *hello* using the following commands:

```
$ rm hello
$ make
```

`make` with no arguments executes the first rule in *makefile*. If we list the contents of our directory using `ls` we see that it contains our newly compiled executable. So far this is not too useful. It is only a little better than typing it in every time, however **`make`** can do a lot more than this. For instance **make** knows if your code has been modified since the last compile. Let's type in **make** again.

```
$ make
```

This time we get a message telling us that hello is up to date.

```
make: `hello' is up to date.
```

`make` accomplishes this by comparing the modification date of the output file *hello* to the modification date of the source file, *hello.cpp*. So if we `touch` this file to change its modification date then we can trick **make** into recompiling.

```
$ touch hello.cpp
$ make
```

Here we see that **make** recompiles *hello*.

In general **make** compares the modification date for every file specified in the dependency list against the modification date of the output file made by the rule. If any of the dependency files are newer it re-executes the command specified by the rule.

Next we will illustrate how to handle multiple files with **make**. Create a new file named *util.cpp* containing the following code:

```
#include <iostream>
using namespace std;

void emphasize()
{
    cout << “!!!” << endl ;
}
```

Then create another file called *util.h* containing the following:

```
void emphasize();
```

Next open *hello.cpp* in an editor and modify it so that it contains the following:

```
#include "util.h"
#include <iostream>
using namespace std;

int main()
{
  cout << "Hello World";
  emphasize();
 return 0;
}
```

Now our list of source files consists of three files: *hello.cpp*, *util.cpp* and *util.h*. Now that we have our source files we will add them to our *makefile*. So open *makefile* in an editor and modify it as follows:

```
# our first makefile - version 2


hello: hello.o util.o
      g++ hello.o util.o –o hello
hello.o: hello.cpp
      g++ -c hello.cpp
util.o: util.h util.cpp
      g++ -c util.cpp


clean:
      rm *.o hello
```

This makefile more clearly illustrates how dependencies can be used. Let's walk through what happens when we enter the **make** command:

```
$ make
```

1. **make** executes the first rule it finds, namely hello.
2. hello has dependencies *hello.o* and *util.o*. If these files are not present **make** looks for a rule that tells it how to construct them. Assume both files are not present.
3. First **make** looks for a rule to build *hello.o*, which is found below.
4. This rule depends on *hello.cpp* which is present, so the command executes and an object file *hello.o* is created. (Note the –c flag.)
5. Then **make** returns to the hello rule and finds that *util.o* is still not present, so it looks for a rule to build this file.
6. In the util.o rule we see that it depends on *util.h* and *util.cpp*. These are present so the command is executed and *util.o* is created.
7. Finally both *hello.o* and *util.o* are present so the hello rule executes its command and links the object files into an executable named *hello*.

The final addition to our makefile is the `clean` rule. This rule removes object files and the executable. A clean rule is often very useful during the development process.

Now we have something a little more useful. We can build our whole project using **make**, or we can compile each individual piece as we work on it, and we can clean up after ourselves. (Note: Often during the development process we may want the ability to compile only the source file on which we are currently working.) Let's try to rebuild our project piece by piece. Enter the following commands:

```
$ make clean
$ make util.o
$ make hello.o
$ make hello
$ ls
```

Our makefile is pretty useful now, but we can add even more features using variables.

Common Errors

By far the most common error encountered is leaving out the tab when you begin using `make` is not using a tab before the command

for a given rule. This produces the following error message:

```
makefile:2: *** missing separator.  Stop.
```

This is tells us that at line two in *makefile* we are missing a separator and should have used a tab. Let's cause this error in our *makefile* right now by replacing the **tab** in our rule for **hello** with spaces. The highlighted area in the following shows where to remove the tab and add spaces.

```
# our first makefile - version 2


hello: hello.o util.o
      g++ hello.o util.o –o hello
hello.o: hello.cpp
      g++ -c hello.cpp
util.o: util.h util.cpp
      g++ -c util.cpp


clean:
      rm *.o hello
```

Now run make again.

```
$ make
```

You should see a similar error message as before.

```
makefile:2: *** missing separator.  Stop.
```

Before we continue, make sure you correct the error we just caused.

The next most common error is when the make file cannot find one of the files specified in the list of file dependencies. This is usually caused by a typo in the file name. For the sake of brevity we will not do an example of this. But here is what the outlook would look if we had misspelled *hello.cpp* as *hellow.cpp*.

```
make: *** No rule to make target `hellow.cpp', needed by `hello.o'.  Stop.
```

Using Variables

Variables can be used inside a makefile in a way similar to that in an ordinary programming language. Variables can act as a substitute for any item.

Variable Declaration

Variable declaration is very simple:

<var_name> = < values >

It is the variable name followed by an equals sign and then one or more values such as:

```
cflags = -g -Wall
```

White space after the equals sign is ignored. Variables can also be assigned to other variables.

```
objects1 = hello.o
objects2 = ${object1} util.o
```

The above assignment is the same as this one:

```
objects2 = hello.o util.o
```

Variable References

We reference variables (i.e. expand them) by putting them inside curly brackets or parenthesis with a dollar sign in front, such as:

```
objects2 = hello.o util.o
program: ${objects2}
      g++ ${objects2} -o program
```

Uses

In many instances variables serve to simplify our make files and make them easier to use. For instance, we can use a variable named cflags to maintain a list of compiler flags we would like to use. Then if we wish to add or remove flags in the future we need only modify this variable. We can do the same thing with the names of our object files by using a variable named objects.

Let's do this right now for our makefile. Modify your makefile as follows:

```
# our first makefile - version 3
#-------------------------------
cflags = -g -Wall
objects = hello.o util.o

hello: $(objects)
      g++ $(cflags) $(objects) -o hello

hello.o: hello.cpp
      g++ -c $(cflags) hello.cpp

util.o: util.h util.cpp
      g++ -c $(cflags) util.cpp

clean:
      rm *.o hello
```

Let's compile again using our new makefile. Enter the following commands:

```
$make clean
$make
```

Debug and Testing

A debugger is software that allows you to see what is going on inside a program while it executes or what a program was doing at the moment it crashed. We will use **`gdb`** to debug programs written in C and C++.

gdb can do four kinds of things (plus other things in support of these) to help you catch bugs in the act:

- Start your program, specifying anything that might affect it behavior.
- Make your program stop on specified conditions.
- Examine what has happened, when your program has stopped.
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

Once started, **gdb** reads commands from the terminal until you tell it to exit with the command *"quit"*. You can get online help from gdb itself by using the command *"help"*.

You can run **gdb** with no arguments or options; but the most usual way to start **gdb** is with one argument:

gdb program

For more information on **gdb** type the following at your bash shell prompt:

man gdb

Compiling your program for GDB:

For GDB to work with your program you will have to use the *"-g"* option with the gcc or g++ compiler. This option produces debugging information in the operating system's native format.

"gcc sort.c -o sort -g"

Important Commands for Running GDB:

(gdb) set width 70
Sets number of characters gdb thinks are in a lines.

(gdb) run
Starts debugged program You may specify arguments to give it as if in a bash shell.

(gdb)break random_gen
Sets breakpoint a specified line or function.

(gdb)n
Step program, proceeding through subroutine calls.

(gdb)s
*S*tep program until it reaches a different source line.

(gdb)bt
Print backtrace of all stack frames, or innermost COUNT frames.

(gdb)p array
Print value of expression EXP. Variables accessible are those of the lexical environment of the selected stack frame, plus all those whose scope is global or an entire file.

(gdb)l
List specified function or line. With no argument, lists ten more lines after or around previous listing.

(gdb)p variable = new_variable
Use assignment expressions to give values to convenience variables.

(gdb)c
Continue program being debugged, after signal or breakpoint.

(gdb)quit

Exits gdb.

Gcov

gcov is a test coverage program. Use it in concert with gcc to analyze your programs to create more efficient, faster running code. You can use gcov as a profiling tool to help discover where optimization efforts will best affect your code. You can also use gcov along with the other profiling tool, gprof, to assess which parts of your code use the greatest amount of computing time.

Profiling tools help analyze your code's performance. Using a profiler such as gcov or gprof, you can find out some basic performance statistics, such as:

- how often each line of code executes
- what lines of code are actually executed
- how much computing time each section of code uses

Once you know these things about how your code works when compiled, you can look at each module to see which modules should be optimized. Gcov helps you determine where to work on optimization.

For more information on gcov type at your bash shell prompt:

man gcov

Compiling your program for gcov:

In order to use gcov on your program you must use the *"-fprofile-arcs"* and *"-ftest-coverage"* options with gcc or g++. This tells the compiler to generate additional information needed by gcov (basically a flow graph of the program) and also includes addition code in the object files for generating the extra profiling information needed by gcov.

gcc sort.c -o sort -fprofile-arcs -ftest-coverage

Running gcov on your program:

gcov -fb sort.c

This will generate a *sort.c.gcov* file that contains the output of gcov. The *-b* option outputs branch probabilities which allows you to see who often each branch in your program was taken. The *-f* option outputs function summaries for each function.

gcov output interpretation:

For each basic block, a line is printed after the last line of the basic block describing the branch or call that ends the basic block. There can be multiple branches and calls listed for a single line if there are multiple basic blocks that end on that line. For a branch, if it was executed at least once, then a percentage indicating the number of times the branch was executed will be printed. Otherwise, the message "never executed" is printed. For a call, if it was executed at lease once, then a percentage indicating the number of times that call returned divided by the number of times the call was executed will be printed.

The execution counts are cumulative. If a program is executed again without removing the .da file, the count for the number of times each line in the source was executed would be added to the results of the previous runs.

If you want to prove that every single line in your program was executed, you should not compile with the optimization at the same time. On some machines the optimizer can eliminate some simples code lines by combining them with other lines.

Gprof

Gprof produces an execution profile of C programs. The effect of called routines is incorporated in the profile of each caller. The profile data is taken from the call graph profile file (gmon.out default) which is created by programs that are compiled with the *"-pg"* option of gcc. The *"-pg"* option also links in versions of library routines that are compiled for profiling. Gprof reads the given object file (the default is "a.out") and established the relation between its symbol table and the call graph profile form gmon.out. If more than one profile file is specified, the gprof output shows the sum of the profile information in the given profile files.

Gprof calculates the amount of time in each routine. Next, these times are propagated along the edges of a call graph. Cycles are discovered, and calls into a cycle are made to share the time of the cycle.

Several forms of output are available for the analysis.

The flat profile shows how much time your program spent in each function, and how many times that function was called. If you simply want to know which function burns most of the cycles, it is stated concisely here.

The call graph shows, for each function, which functions called it, which other functions it called, and how many times. There is also an estimate of how much time was spent in the subroutines of each function. This can suggest places where you might try to eliminate function calls that use a lot of time.

The annotated source listing is a copy of the program's source code, labeled with the number of times each line of the program was executed.

For more information on gprof type at your bash shell prompt:

man gprof

Compiling your program for gprof:

You must use the *"-pg"* option for gcc or g++ for gprof to function properly. This option generates extra code to write profile information suitable for gprof. You must use this option when compiling the source files and linking the object files.

gcc sort.c -o sort -pg

Running gprof:

gprof sort > sort.out

This will generate a sort.out file that contains the detailed output of gprof.

Dos2unix and Unix2dos

These programs convert plain text files in DOS/MAC format to UNIX format and vice versa.

For more information on dos2unix or unix2dos type at your bash shell prompt:

man dos2unix *or* *man unix2dos*

Running dos2unix or unix2dos:

dos2unix sort.c

This converts sort.c from DOS/MAC format and writes sort.c in UNIX format.

dos2unix -n sort.c sort_unix.c

This converts sort.c from DOS/MAC format and write sort_unix.c in UNIX format.

unix2dos sort.c
This converts sort.c from UNIX format and writes sort.c in DOS/MAC format.

unix2dos -n sort.c sort_dos.c
This converts sort.c from UNIX format and write sort_dos.c in UNIX format.

References:

1. Online GNU Make Manual.
   http://mirrors.mix5.com/gnu/Manuals/make-3.80/html_node/make.html#SEC_Top

2. The Linunx man pages
   $ man make

3. *GNU Make: A Program for Directed Compilation*, R. Stallman and R McGrath, Free Software Foundation, 2002.

Appendix:

A More Complicated Makefile

```
#
# Maintain the following definitions:
#
#       HDR     all header files (*.h) that you create
#       SRC     all C source files (*.cpp) that you create
#       OBJ     all object files (*.o) required to load your program
#       EXE     name of the executable
#       DOC     all the documentation files
#       CFLAGS  all the compiler options
#
# Use the following make targets:
#
#       all     (or nothing) to build your program (into EXE's value)
#       clean   to remove the executable and .o files

SRC =  main.cpp airplane.cpp queue.cpp
OBJ =  main.o airplane.o queue.o
EXE =  airplane
CFLAGS =  -Wall


all: ${OBJ}
        g++   ${OBJ} -o ${EXE} ${CFLAGS}

main.o: main.cpp
        g++ -c -g  ${CFLAGS}$  main.cpp

airplane.o: airplane.cpp
        g++ -c -g ${CFLAGS}$  airplane.cpp

queue.o: queue.cpp
        g++ -c -g ${CFLAGS}$  queue.cpp

clean:
        rm -f ${OBJ} ${EXE} core
```

Additional Information on Makefiles

Variable Flavors

There are two flavors of variables but we will only discuss *recursively expanded variables* here. The value you specify is installed verbatim; if it contains references to other variables, these references are expanded whenever this variable is substituted (in the course of expanding some other string). When this happens, it is called *recursive expansion*.[1] This means

that we can define variables out of order.

```
objects2 = ${object1} util.o
objects1 = hello.o
```

This code definition works just as well as our earlier one, since objects1 is not expanded until objects2 is referenced. This also leads to unusual behavior such as the following:

```
Cflags = $(Cflags) -g
```

When Cflags is expanded later on in a makefile it results in an infinite loop, since Cflags is recursively expanded. We can fix this by using the += operator such as:

```
Cflags += -g
```

This appends "-g" to the end of Cflags.

Advanced Variable Use

Rules can also modify variables. We can use this fact to build a makefile that allows us to produce a debug version of our *hello* program.

First we will add another variable to our make file and then use a rule to change the value of this variable.

Add cflags to makefile as follows:

```
# our first makefile - version 4a
#-------------------------------
cflags = -O
objects = hello.o util.o
hello: $(objects)
      g++ $(cflags) $(objects) –o hello

hello.o: hello.cpp
      g++ -c $(cflags) hello.cpp

util.o: util.h util.cpp
      g++ -c $(cflags) util.cpp

clean:
      rm $(objects) hello
```

We have included the optimization flag, so that our executable is optimized. Now lets add a rule to change our clfags variable

to flags that are useful during debugging.

Change your make file as follows:

```
# our first makefile - version 4b
#-------------------------------
cflags = -O
objects = hello.o util.o

#compile with optimization
hello: $(objects)
      g++ $(cflags) $(objects) -o hello

# compile for debug
debug: cflags = -g -Wall
debug: hello

hello.o: hello.cpp
      g++ -c $(cflags) hello.cpp

util.o: util.h util.cpp
      g++ -c $(cflags) util.cpp

clean:
      rm $(objects) hello
```

With the debug rule we have set cflags to a new value so that it no longer includes the optimazion flag, instead it includes the debug symbols flag, -g, and the show all warning flag, -Wall. These are very useful when debugging code.

This has been a very rudimentary introduction to make. Make is capable of much, much more. For more information take a look at the sources listed in the References section below.