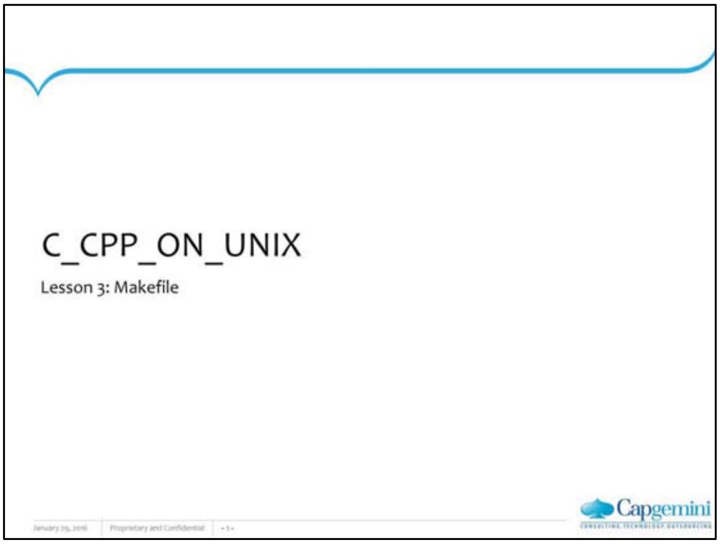


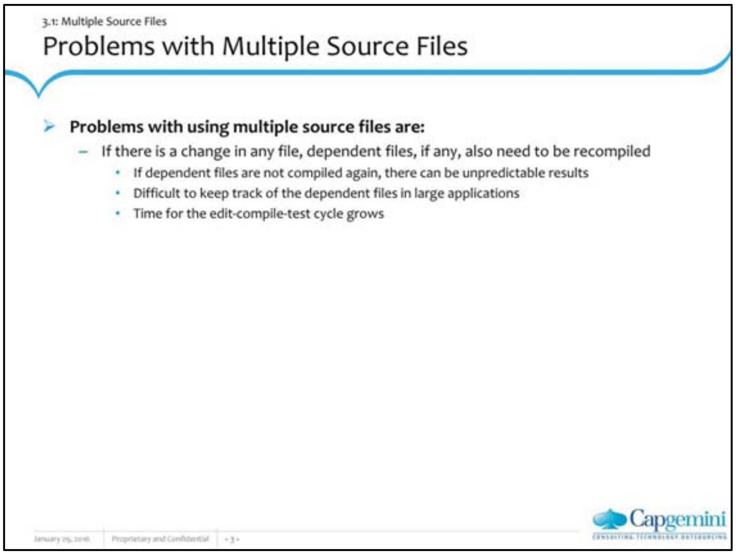
C_CPP_ON_UNIX

Lesson 3: Makefile

Lesson Objectives

- To understand the following topics:
  - The make command and makefiles

Lesson Objectives:
This lesson introduces to the fundamentals of the Java programming language.
Lesson 3: This lesson introduces you to *makefile*, a UNIX utility used to build your application.



3.1: Multiple Source Files

Problems with Multiple Source Files

- **Problems with using multiple source files are:**
  - If there is a change in any file, dependent files, if any, also need to be recompiled
    - If dependent files are not compiled again, there can be unpredictable results
    - Difficult to keep track of the dependent files in large applications
    - Time for the edit-compile-test cycle grows

January 29, 2016 | Proprietary and Confidential | - 3 -

When writing small programs, many people simply rebuild their application after edits by recompiling all the files. However, with larger programs, some problems with this simple approach become apparent.

Time for the edit–compile–test cycle grows. Even the most patient programmer prefers to avoid recompiling all files when only one is changed.

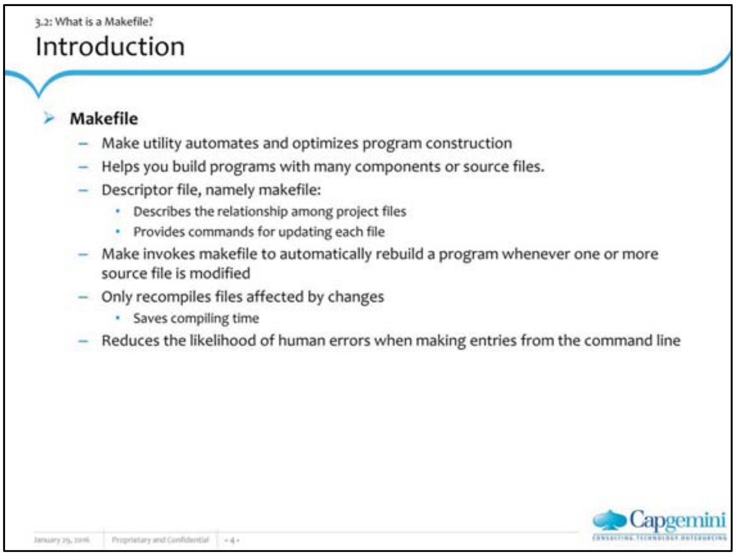
A potentially much more difficult problem arises when multiple header files are created and included in different source files. Suppose you have header files a.h, b.h and c.h, and C source files main.c, 2.c and 3.c. (We hope that you choose better names than these for real projects!) you could have the situation follows:

```
/* main.c */
#include "a.h"
...
/* 2.c */
#include "a.h"
#include "b.h"
...
/* 3.c */
#include "b.h"
#include "c.h"
…
```

If the programmer changes c.h, the files main.c and 2.c do not need to be recompiled, since they do not depend on this header file.

The file 3.c does depend on c.h and should therefore be recompiled if c.h is changed.

However, if b.h was changed and the programmer forgot to recompile 2.c, then the resulting program might no longer function correctly.

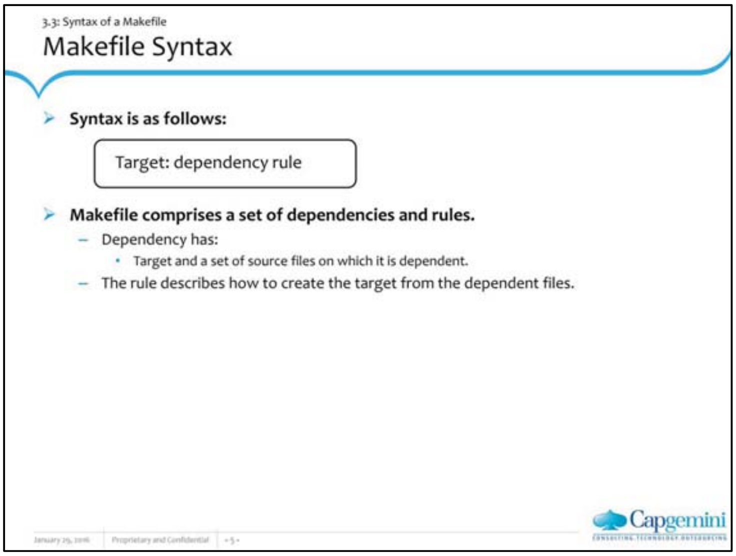


3.2: What is a Makefile?

Introduction

- **Makefile**
  - Make utility automates and optimizes program construction
  - Helps you build programs with many components or source files.
  - Descriptor file, namely makefile:
    - Describes the relationship among project files
    - Provides commands for updating each file
  - Make invokes makefile to automatically rebuild a program whenever one or more source file is modified
  - Only recompiles files affected by changes
    - Saves compiling time
  - Reduces the likelihood of human errors when making entries from the command line

January 29, 2016 | Proprietary and Confidential | - 4 -



3.3: Syntax of a Makefile

Makefile Syntax

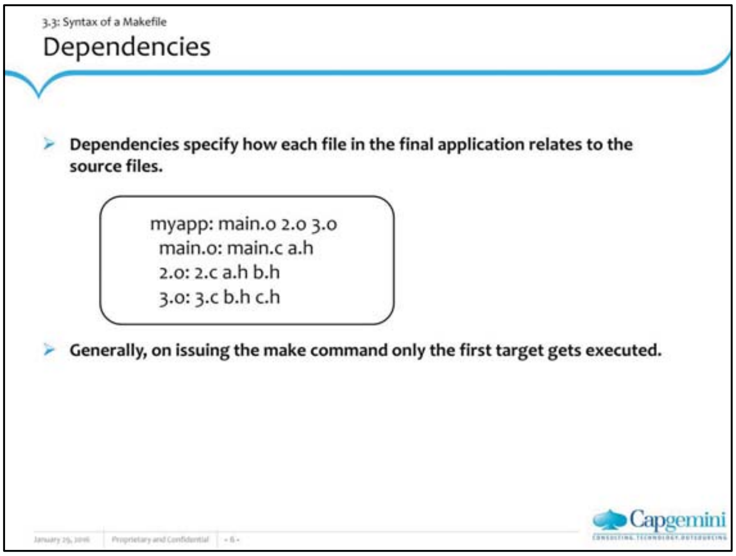
- **Syntax is as follows:**

```
Target: dependency rule
```

- **Makefile comprises a set of dependencies and rules.**
  - Dependency has:
    - Target and a set of source files on which it is dependent.
  - The rule describes how to create the target from the dependent files.

January 29, 2016 | Proprietary and Confidential | - 5 -

A makefile consists of a set of dependencies and rules. A dependency has a target (a file to be created) and a set of source files upon which it is dependent. The rules describe how to create the target from the dependent files. Commonly, the target is a single executable file.
The makefile is read by the make command, which determines the target file or files that are to be made and then compares the dates and times of the source files to decide which rules need to be invoked to construct the target. Often, other intermediate targets have to be created before the final target can be made. The make command uses the makefile to determine the order in which the targets have to be made and the correct sequence of rules to invoke.


3.3: Syntax of a Makefile

Dependencies

- **Dependencies specify how each file in the final application relates to the source files.**

```
myapp: main.o 2.0 3.0
 main.o: main.c a.h
 2.0: 2.c a.h b.h
 3.0: 3.c b.h c.h
```

- **Generally, on issuing the make command only the first target gets executed.**

January 25, 2016 | Proprietary and Confidential | - 6 -

Dependencies specify how each file in the final application relates to the source files. In your example above, you might specify dependencies that say your final application requires (depends on) main.o, 2.0 and 3.0; and likewise for main.o (main.c and a.h); 2.0 (2.c, a.h and b.h); and 3.0 (3.c, b.h and c.h). Thus main.o is affected by changes to main.c and a.h, and it needs to be recreated, by recompiling main.c, if either of these two files changes.

In a makefile, you write these rules by writing the name of the target, a colon, spaces or tabs and then a space or tab–separated list of files that are used to create the target file. The dependency list for your example is:

```
myapp: main.o 2.0 3.0
main.o: main.c a.h
2.0: 2.c a.h b.h
3.0: 3.c b.h c.h
```

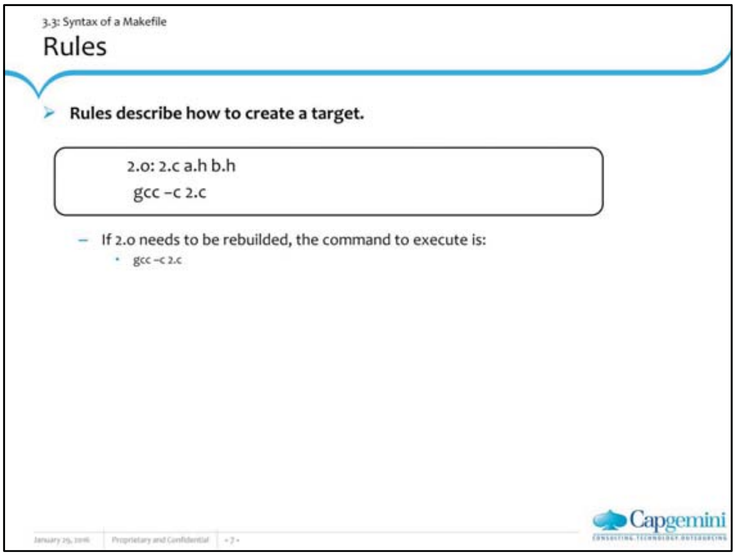
This says that myapp depends on main.o, 2.0 and 3.0, main.o depends on main.c and a.h, and so on. This set of dependencies gives a hierarchy showing how the source files relate to one other.

You can see quite easily that, if b.h changes, then you need to revise both 2.0 and 3.0 and, since 2.0 and 3.0 will have changed.

You also need to rebuild myapp. If you wish to make several files, then use the phony target *all*. Suppose your application consisted of both the binary file myapp and a manual page, myapp.1. Specify this with the line:

```
all: myapp myapp.1
```

If you do not specify an *all* target, make simply creates the first target it finds in the makefile.



3.3: Syntax of a Makefile

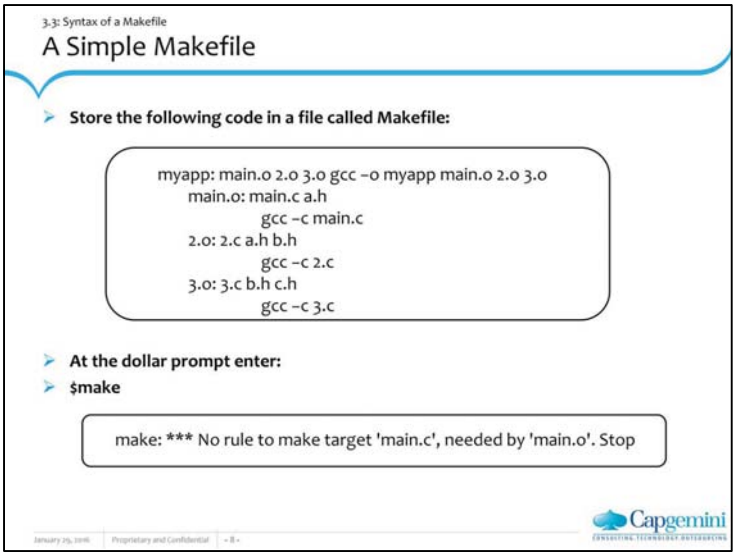
Rules

- Rules describe how to create a target.

```
2.o: 2.c a.h b.h
 gcc –c 2.c
```

- If 2.o needs to be rebuilded, the command to execute is:
  - gcc –c 2.c

All rules must be on lines that start with a tab; a space will not do. Also, a space at the end of a line in the makefile may cause a make command to fail.



3.3: Syntax of a Makefile

A Simple Makefile

- Store the following code in a file called Makefile:

```
myapp: main.o 2.o 3.o gcc –o myapp main.o 2.o 3.o
     main.o: main.c a.h
             gcc –c main.c
     2.o: 2.c a.h b.h
             gcc –c 2.c
     3.o: 3.c b.h c.h
             gcc –c 3.c
```

- At the dollar prompt enter:
- $make

```
make: *** No rule to make target 'main.c', needed by 'main.o'. Stop
```

January 29, 2016 | Proprietary and Confidential | - 8 -

The make command has assumed that the first target in the makefile, myapp, is the file that you wish to create. It has then looked at the other dependencies and, in particular, has determined that a file called main.c is needed. Since you haven't created this file yet and the makefile does not say how it might be created, make has reported an error. Let's create the source files and try again. Since we're not interested in the result, these files can be very simple. The header files are actually empty, so you can create them with touch.

```
$ touch a.h
$ touch b.h
$ touch c.h
```

main.c contains main, which calls function_two and function_three. The other two files define function_two and function_three. The source files have #include lines for the appropriate headers, so they appear to be dependent on the contents of the included headers. It is not much of an application, but here are the listings.

```
/* main.c */
#include "a.h"
extern void function_two();
extern void function_three();
int main(){
function_two();
function_three();
exit (EXIT_SUCCESS);  }
```

```
/* 2.c */
#include "a.h"
#include "b.h"
void function_two() {
}
/* 3.c */
#include "b.h"
#include "c.h"
void function_three() {
}
$ make
gcc –c main.c
gcc –c 2.c
gcc –c 3.c
gcc –o myapp main.o 2.o 3.o
$
```

This is a successful make.

How It Works

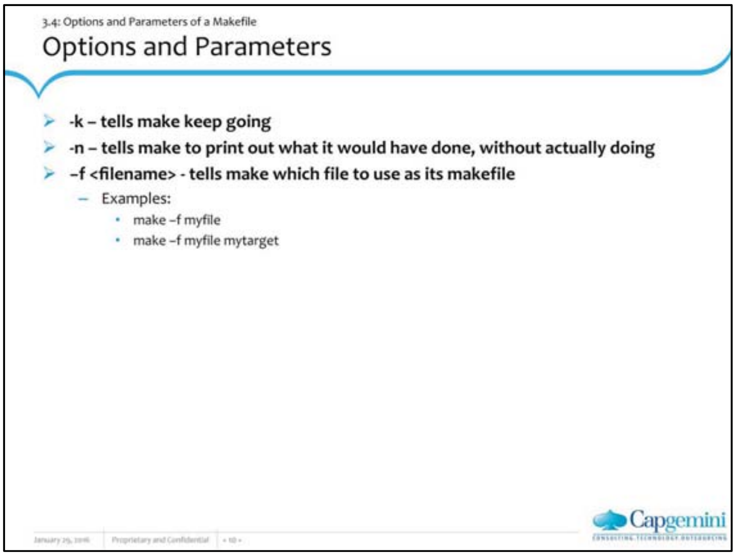
The *make* command has processed the dependencies section of the makefile and determined the files that need to be created and in which order. Even though you listed how to create myapp first, make has determined the correct order for creating the files. It has then invoked the appropriate commands you gave it in the rules section for creating those files. The make command displays the commands as it executes them. you can now test your makefile to see whether it handles changes to the file b.h correctly.

```
$ touch b.h
$ make
gcc –c 2.c
gcc –c 3.c
gcc –o myapp main.o 2.o 3.o
$
```

The make command has read your makefile, determined the minimum number of commands required to rebuild myapp and carried them out in the correct order. Let's see what happens if you delete an object file.

```
$ rm 2.o
$ make –f Makefile1
gcc –c 2.c
gcc –o myapp main.o 2.o 3.o
$
```

Again, make correctly determines the actions required.



3.4: Options and Parameters of a Makefile

Options and Parameters

- **-k – tells make keep going**
- **-n – tells make to print out what it would have done, without actually doing**
- **–f <filename> - tells make which file to use as its makefile**
  - Examples:
    - make –f myfile
    - make –f myfile mytarget

January 29, 2016 | Proprietary and Confidential | - 10 -

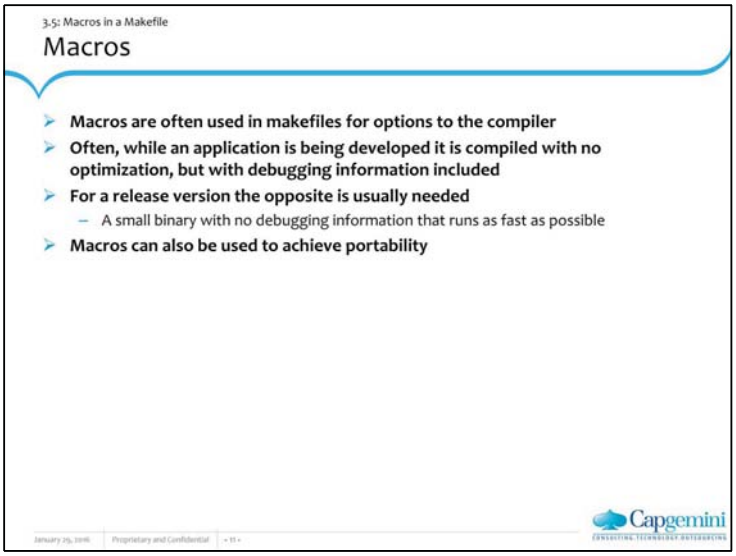
The make program has several options. The three most commonly used are:
–k, which tells make to 'keep going' when an error is found, rather than stopping as soon as the first problem is detected. You can use this, for example, to find out in one go which source files fail to compile.

–n, which tells make to print out what it would have done, without actually doing it.

–f <filename>, which allows you to tell make which file to use as its makefile. If you do not use this option, make looks first for a file called makefile in the current directory. If that doesn't exist, it looks for a file called Makefile. By convention, most UNIX programmers use Makefile.

To make a particular target, which is usually an executable file, you can pass its name to make as a parameter. If you do not, make will try to make the first target listed in the makefile.

Many programmers specify all as the first target in their makefile and then list the other targets as being dependent on all. This convention makes it clear which target the makefile should attempt to build by default when no target is specified. you suggest you stick to this convention.



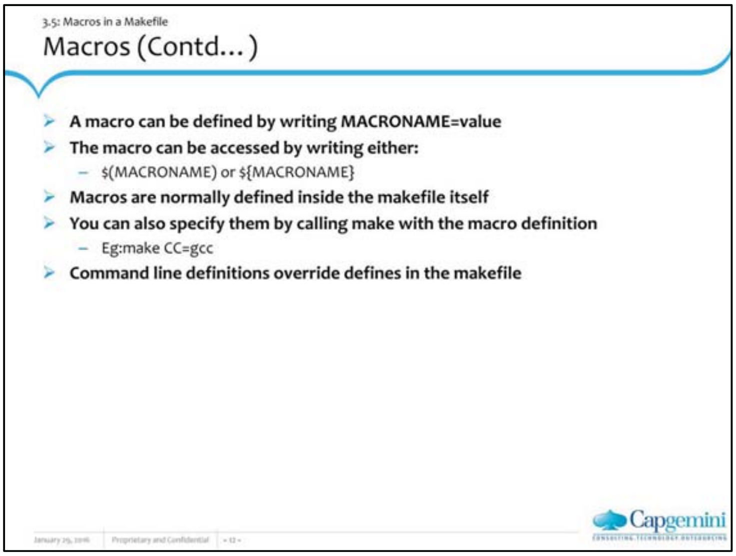
3.5: Macros in a Makefile

Macros

- Macros are often used in makefiles for options to the compiler
- Often, while an application is being developed it is compiled with no optimization, but with debugging information included
- For a release version the opposite is usually needed
  - A small binary with no debugging information that runs as fast as possible
- Macros can also be used to achieve portability

January 29, 2016 | Proprietary and Confidential | - 11 -

Let us look at an example where Makefile can be used. You have written a Makefile with the *gcc* Compiler. On other UNIX systems, you might be using cc or c89. If you ever wanted to take your makefile to a different version of UNIX, or even if you obtained a different compiler to use on your existing system, you would have to change several lines of your makefile to make it work. Macros are a good way of collecting together all these system dependent parts, making it easy to change them.

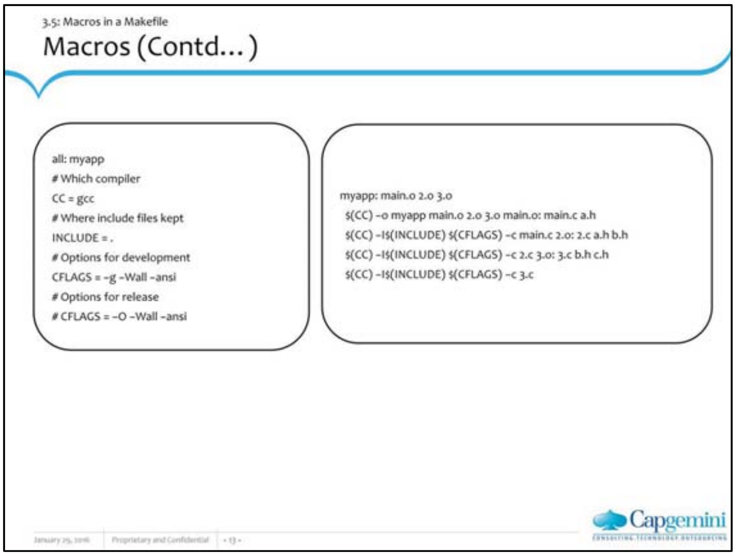


3.5: Macros in a Makefile

Macros (Contd…)

- **A macro can be defined by writing MACRONAME=value**
- **The macro can be accessed by writing either:**
  - $(MACRONAME) or ${MACRONAME}
- **Macros are normally defined inside the makefile itself**
- **You can also specify them by calling make with the macro definition**
  - Eg:make CC=gcc
- **Command line definitions override defines in the makefile**

January 29, 2016 | Proprietary and Confidential | - 12 -



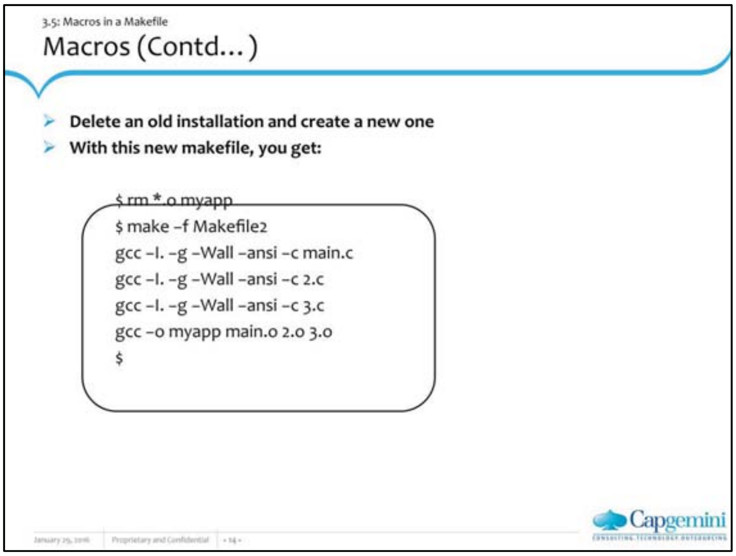
3.5: Macros in a Makefile

Macros (Contd…)

```
all: myapp
# Which compiler
CC = gcc
# Where include files kept
INCLUDE = .
# Options for development
CFLAGS = -g -Wall -ansi
# Options for release
# CFLAGS = -O -Wall -ansi
```

```
myapp: main.o 2.o 3.o
  $(CC) -o myapp main.o 2.o 3.o main.o: main.c a.h
  $(CC) -I$(INCLUDE) $(CFLAGS) -c main.c 2.o: 2.c a.h b.h
  $(CC) -I$(INCLUDE) $(CFLAGS) -c 2.c 3.o: 3.c b.h c.h
  $(CC) -I$(INCLUDE) $(CFLAGS) -c 3.c
```

January 29, 2016 | Proprietary and Confidential | - 13 -



3.5: Macros in a Makefile

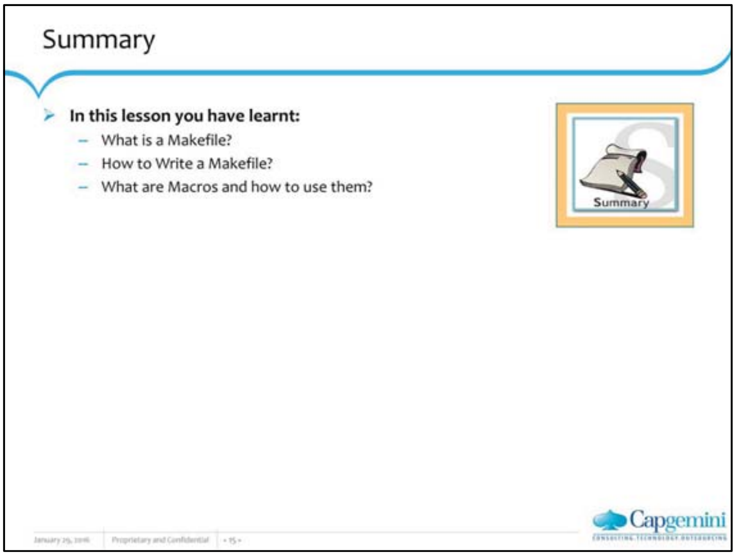
Macros (Contd...)

- Delete an old installation and create a new one
- With this new makefile, you get:

```
$ rm *.o myapp
$ make –f Makefile2
gcc –I. –g –Wall –ansi –c main.c
gcc –I. –g –Wall –ansi –c 2.c
gcc –I. –g –Wall –ansi –c 3.c
gcc –o myapp main.o 2.o 3.o
$
```

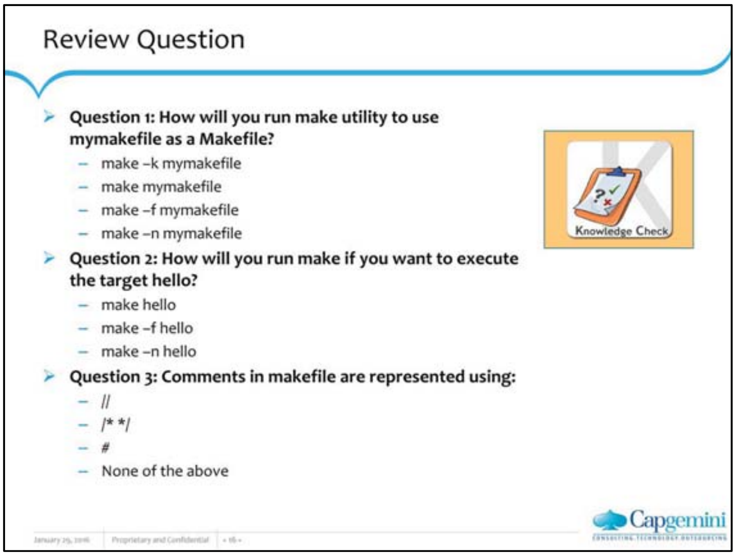
January 29, 2016 | Proprietary and Confidential | - 14 -

How It Works
The make command replaces the $(CC), $(CFLAGS) and $(INCLUDE) with the appropriate definition, rather like the C compiler does with #define. Now if you want to change the compile command, you only need to change a single line of the makefile.



Summary

- **In this lesson you have learnt:**
  - What is a Makefile?
  - How to Write a Makefile?
  - What are Macros and how to use them?



Review Question

- **Question 1: How will you run make utility to use mymakefile as a Makefile?**
  - make –k mymakefile
  - make mymakefile
  - make –f mymakefile
  - make –n mymakefile
- **Question 2: How will you run make if you want to execute the target hello?**
  - make hello
  - make –f hello
  - make –n hello
- **Question 3: Comments in makefile are represented using:**
  - //
  - /* */
  - #
  - None of the above

Knowledge Check