```
                 U puts

$ nm hello.exe | grep main
00000001004080cc I __imp___main
0000000100401120 T __main
00000001004010e0 T main
......
```

"nm" is commonly-used to check if a particular function is defined in an object file. A `'T'` in the second column indicates a function that is *defined*, while a `'U'` indicates a function which is *undefined* and should be resolved by the linker.

"ldd" Utility - List Dynamic-Link Libraries

The utility "ldd" examines an executable and displays a list of the shared libraries that it needs. For example,

```
> ldd hello.exe
ntdll.dll => /cygdrive/c/WINDOWS/SYSTEM32/ntdll.dll (0x7ff9ba3c0000)
KERNEL32.DLL => /cygdrive/c/WINDOWS/System32/KERNEL32.DLL (0x7ff9b9880000)
KERNELBASE.dll => /cygdrive/c/WINDOWS/System32/KERNELBASE.dll (0x7ff9b6a60000)
SYSFER.DLL => /cygdrive/c/WINDOWS/System32/SYSFER.DLL (0x6ec90000)
ADVAPI32.dll => /cygdrive/c/WINDOWS/System32/ADVAPI32.dll (0x7ff9b79a0000)
msvcrt.dll => /cygdrive/c/WINDOWS/System32/msvcrt.dll (0x7ff9b9100000)
sechost.dll => /cygdrive/c/WINDOWS/System32/sechost.dll (0x7ff9b9000000)
RPCRT4.dll => /cygdrive/c/WINDOWS/System32/RPCRT4.dll (0x7ff9b9700000)
cygwin1.dll => /usr/bin/cygwin1.dll (0x180040000)
```

2. GNU Make

The "make" utility automates the mundane aspects of building executable from source code. "make" uses a so-called `makefile`, which contains rules on how to build the executables.

You can issue "`make --help`" to list the command-line options; or "`man make`" to display the man pages.

2.1 First Makefile By Example

Let's begin with a simple example to build the Hello-world program (`hello.c`) into executable (`hello.exe`) via make utility.

```
1  // hello.c
2  #include <stdio.h>
3
4  int main() {
5      printf("Hello, world!\n");
6      return 0;
7  }
```

Create the following file named "makefile" (without any file extension), which contains rules to build the executable, and save in the same directory as the source file. Use "tab" to indent the command (NOT spaces).

```
all: hello.exe

hello.exe: hello.o
	gcc -o hello.exe hello.o
```

```
hello.o: hello.c
         gcc -c hello.c

clean:
         rm hello.o hello.exe
```

Run the "make" utility as follows:

```
> make
gcc -c hello.c
gcc -o hello.exe hello.o
```

Running make without argument starts the target "all" in the makefile. A makefile consists of a set of rules. A rule consists of 3 parts: a target, a list of pre-requisites and a command, as follows:

```
target: pre-req-1 pre-req-2 ...
        command
```

The *target* and *pre-requisites* are separated by a colon (:). The *command* must be preceded by a tab (NOT spaces).

When make is asked to evaluate a rule, it begins by finding the files in the prerequisites. If any of the prerequisites has an associated rule, make attempts to update those first.

In the above example, the rule "all" has a pre-requisite "hello.exe". make cannot find the file "hello.exe", so it looks for a rule to create it. The rule "hello.exe" has a pre-requisite "hello.o". Again, it does not exist, so make looks for a rule to create it. The rule "hello.o" has a pre-requisite "hello.c". make checks that "hello.c" exists and it is newer than the target (which does not exist). It runs the command "gcc -c hello.c". The rule "hello.exe" then run its command "gcc -o hello.exe hello.o". Finally, the rule "all" does nothing.

More importantly, if the pre-requisite is not newer than than target, the command will not be run. In other words, the command will be run only if the target is out-dated compared with its pre-requisite. For example, if we re-run the make command:

```
> make
make: Nothing to be done for `all'.
```

You can also specify the target to be made in the make command. For example, the target "clean" removes the "hello.o" and "hello.exe". You can then run the make without target, which is the same as "make all".

```
> make clean
rm hello.o hello.exe

> make
gcc -c hello.c
gcc -o hello.exe hello.o
```

Try modifying the "hello.c" and run make.

NOTES:

- If the *command* is not preceded by a tab, you get an error message "makefile:4: *** missing separator. Stop."
- If there is no makefile in the current directory, you get an error message "make: *** No targets specified and no makefile found. Stop."
- The makefile can be named "makefile", "Makefile" or "GNUMakefile", without file extension.

2.2 More on Makefile

Comment & Continuation

A comment begins with a # and lasts till the end of the line. Long line can be broken and continued in several lines via a back-slash (\).

Syntax of Rules

A general syntax for the rules is:

```
target1 [target2 ...]: [pre-req-1 pre-req-2 ...]
        [command1
         command2
         ......]
```

The rules are usually organized in such as way the more general rules come first. The overall rule is often name "`all`", which is the default target for `make`.

Phony Targets (or Artificial Targets)

A target that does not represent a file is called a phony target. For example, the "`clean`" in the above example, which is just a label for a command. If the target is a file, it will be checked against its pre-requisite for out-of-date-ness. Phony target is always out-of-date and its command will be run. The standard phony targets are: `all`, `clean`, `install`.

Variables

A variable begins with a `$` and is enclosed within parentheses `(...)` or braces `{...}`. Single character variables do not need the parentheses. For example, `$(CC)`, `$(CC_FLAGS)`, `$@`, `$^`.

Automatic Variables

Automatic variables are set by make after a rule is matched. There include:

- $@: the target filename.
- $*: the target filename without the file extension.
- $<: the first prerequisite filename.
- $^: the filenames of all the prerequisites, separated by spaces, discard duplicates.
- $+: similar to $^, but includes duplicates.
- $?: the names of all prerequisites that are newer than the target, separated by spaces.

For example, we can rewrite the earlier makefile as:

```
all: hello.exe

# $@ matches the target; $< matches the first dependent
hello.exe: hello.o
        gcc -o $@ $<

hello.o: hello.c
        gcc -c $<

clean:
        rm hello.o hello.exe
```

Virtual Path - VPATH & vpath

You can use VPATH (uppercase) to specify the directory to search for dependencies and target files. For example,

```
# Search for dependencies and targets from "src" and "include" directories
# The directories are separated by space
VPATH = src include
```

You can also use vpath (lowercase) to be more precise about the file type and its search directory. For example,

```
# Search for .c files in "src" directory; .h files in "include" directory
# The pattern matching character '%' matches filename without the extension
vpath %.c src
vpath %.h include
```

Pattern Rules

A pattern rule, which uses pattern matching character '%' as the filename, can be applied to create a target, if there is no explicit rule. For example,

```
# Applicable for create .o object file.
# '%' matches filename.
# $< is the first pre-requisite
# $(COMPILE.c) consists of compiler name and compiler options
# $(OUTPUT_OPTIONS) could be -o $@
%.o: %.c
        $(COMPILE.c) $(OUTPUT_OPTION) $<

# Applicable for create executable (without extension) from object .o object file
# $^ matches all the pre-requisites (no duplicates)
%: %.o
$(LINK.o) $^ $(LOADLIBES) $(LDLIBS) -o $@
```

Implicit Pattern Rules

Make comes with a huge set of implicit pattern rules. You can list all the rule via --print-data-base option.

2.3 A Sample Makefile

This sample makefile is extracted from Eclipse's "C/C++ Development Guide -Makefile".

```
# A sample Makefile
# This Makefile demonstrates and explains
# Make Macros, Macro Expansions,
# Rules, Targets, Dependencies, Commands, Goals
# Artificial Targets, Pattern Rule, Dependency Rule.

# Comments start with a # and go to the end of the line.

# Here is a simple Make Macro.
LINK_TARGET = test_me.exe

# Here is a Make Macro that uses the backslash to extend to multiple lines.
OBJS =  \
 Test1.o \
```

```
 Test2.o \
 Main.o

# Here is a Make Macro defined by two Macro Expansions.
# A Macro Expansion may be treated as a textual replacement of the Make Macro.
# Macro Expansions are introduced with $ and enclosed in (parentheses).
REBUILDABLES = $(OBJS) $(LINK_TARGET)

# Here is a simple Rule (used for "cleaning" your build environment).
# It has a Target named "clean" (left of the colon ":" on the first line),
# no Dependencies (right of the colon),
# and two Commands (indented by tabs on the lines that follow).
# The space before the colon is not required but added here for clarity.
clean :
  rm -f $(REBUILDABLES)
  echo Clean done

# There are two standard Targets your Makefile should probably have:
# "all" and "clean", because they are often command-line Goals.
# Also, these are both typically Artificial Targets, because they don't typically
# correspond to real files named "all" or "clean".

# The rule for "all" is used to incrementally build your system.
# It does this by expressing a dependency on the results of that system,
# which in turn have their own rules and dependencies.
all : $(LINK_TARGET)
  echo All done

# There is no required order to the list of rules as they appear in the Makefile.
# Make will build its own dependency tree and only execute each rule only once
# its dependencies' rules have been executed successfully.

# Here is a Rule that uses some built-in Make Macros in its command:
# $@ expands to the rule's target, in this case "test_me.exe".
# $^ expands to the rule's dependencies, in this case the three files
# main.o, test1.o, and  test2.o.
$(LINK_TARGET) : $(OBJS)
  g++ -g -o $@ $^

# Here is a Pattern Rule, often used for compile-line.
# It says how to create a file with a .o suffix, given a file with a .cpp suffix.
# The rule's command uses some built-in Make Macros:
# $@ for the pattern-matched target
# $< for the pattern-matched dependency
%.o : %.cpp
  g++ -g -o $@ -c $<

# These are Dependency Rules, which are rules without any command.
# Dependency Rules indicate that if any file to the right of the colon changes,
# the target to the left of the colon should be considered out-of-date.
# The commands for making an out-of-date target up-to-date may be found elsewhere
# (in this case, by the Pattern Rule above).
# Dependency Rules are often used to capture header file dependencies.
Main.o : Main.h Test1.h Test2.h
Test1.o : Test1.h Test2.h
Test2.o : Test2.h

# Alternatively to manually capturing dependencies, several automated
# dependency generators exist.  Here is one possibility (commented out)...
# %.dep : %.cpp
```

```
#   g++ -M $(FLAGS) $< > $@
# include $(OBJS:.o=.dep)
```

2.4 Brief Summary

I have presented the basic make features here so that you can read and understand simple makefiles for building C/C++ applications. Make is actually quite complex, and can be considered as a programming language by itself!!

REFERENCES & RESOURCES

1. GCC Manual "Using the GNU Compiler Collection (GCC)" @ http://gcc.gnu.org/onlinedocs.
2. GNU 'make' manual @ http://www.gnu.org/software/make/manual/make.html.
3. Robert Mecklenburg, "Managing Projects with GNU Make", 3rd Edition, 2004.

Latest version tested: CygWin GCC 6.4.0, MinGW-W64 GCC 6.4.0
Last modified: March, 2018

Feedback, comments, corrections, and errata can be sent to Chua Hock-Chuan (ehchua@ntu.edu.sg) | HOME