MAKE : A FIRST CONTACT

1. Short Description

SYNOPSIS: make [-f makefile] [option] ... target ...

The purpose of the make utility is to determine automatically which pieces of a large program need to be recompiled, and issue the commands to recompile them.

To use **make**, you need to write a file called the makefile which describes the relationships among files in your program. In a program, typically the executable file is updated from object files, which are in turn created by compiling source files.

make executes commands in the makefile to update one or more target names, where a name is typically a program. If no -f option is present, make will look for the makefiles GNUmakefile, makefile, and Makefile, in that order. Note that the name GNUmakefile is only recommended for makefile that is specific to GNU make, and will not be understood by other versions of make. An abstract example is illustrated by figures 1 and 2.

Useful options are:

-f file Use file as a makefile

-n Print the commands that would be executed, but do not execute them.

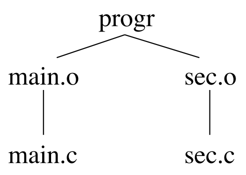
-W file Pretend that the target file has just been modified. When used with the **-n** flag, this shows you what would happen if you were to modify that file. Without **-n**, it is almost the same as running a touch command on the given file before running make, except that the modification time is changed only for make.

Example of a Makefile

```
(1)  progr : main.o sec.o
(2)    gcc -o progr main.o sec.o

(3)  main.o : main.c
(4)    gcc -c main.c

(5)  sec.o : sec.c
(6)    gcc -c sec.c
```



Common Usage

```
% make
% make <target>
```

Syntax of a Makefile Rule

```
<target> : [<target's prerequisites>]<newline>   # Dependency line
<tab><command line><newline>  # Show how to build the target out of their prerequisites
[<tab><command line><newline>]
```

Figure 1. A first abstract example of make and makefile.

In this exercise we consider that the three files 'hello.c', 'tellMe.c' and 'Makefile' contain the following content (for your convenience, these files are available online on Moodle -> Tutorials -> C01 -> extras) :

```
hello.c:
      void tellMe(void);
      main () {tellMe();}

tellMe.c:
      #include <stdio.h>
      void tellMe () {printf("hello\n");}

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

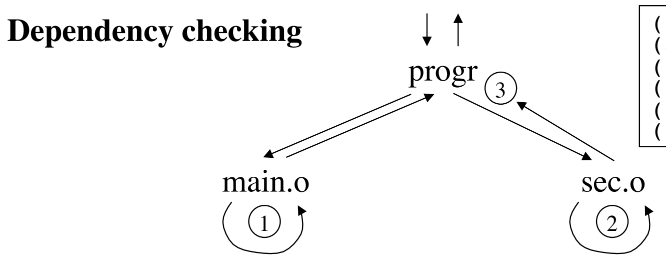
and we consider the following scenario:

```
(1)   % make
(2)   % touch tellMe.c  # or modify tellMe.c
(3)   % make tellMe.o
(4)   % make
(5)   % make            # sic!, redo a make
(6)   % make -W tellMe.c
```

<u>Question 1</u>. Draw the dependency tree of hello.

<u>Question 2</u>. What is the information displayed by the commands (1), (3)-(6)? Explain and verify your answer on machine.

<u>Question 3</u>. Draw the execution tree of (1), (3)-(5).



```
(1) progr : main.o sec.o
(2)   gcc -o progr main.o sec.o
(3) main.o : main.c
(4)   gcc -c main.c
(5) sec.o : sec.c
(6)   gcc -c sec.c
```

Execution Tree

```
①  if (main.o don't exist) then <execute cmd (4)>
    else if ( $t_{main.o} < t_{main.c}$ ) then < execute cmd (4)>  // main.o is out-of-date
    else <do nothing>

②  Similar to ①

③  if (progr don't exist) then <execute cmd (2)>
    else if (( $t_{progr} < t_{main.o}$ ) or ( $t_{progr} < t_{sec.o}$ )) then < execute cmd (2)>  // progr is out-of-date
    else <do nothing>
```

Remark 1. 'make' echoes each command to the standard output before executing it.
Remark 2. If <target's prerequisites> is absent the associated command lines are executed.

Figure 2. Make: Execution Model

2. Header Files: Hello World (revisited)

Consider that the files tellMe.h, hello.c and tellMe_X.c contain respectively (line numbers are not part of the files), cf. also Fig. 3 :

tellMe.h
```
(01)  void tellMe(char s[])
```

hello.c
```
(01)  #include "tellMe.h"
(02)  main() {tellMe("Hello World");}
```

tellMe_ok.c
```
(03)  #include <stdio.h>
(04)  #include "tellMe.h"
(05)  void tellMe(char s[]) {printf("tellMe> %s\n", s);}
```

tellMe_unwiseBug.c (#include "tellMe.h" is missing)
```
(06)  #include <stdio.h>  /* #include "tellMe.h" is missing */
(07)  void tellMe(char s[]) {printf("tellMe> %s\n", s);}
```

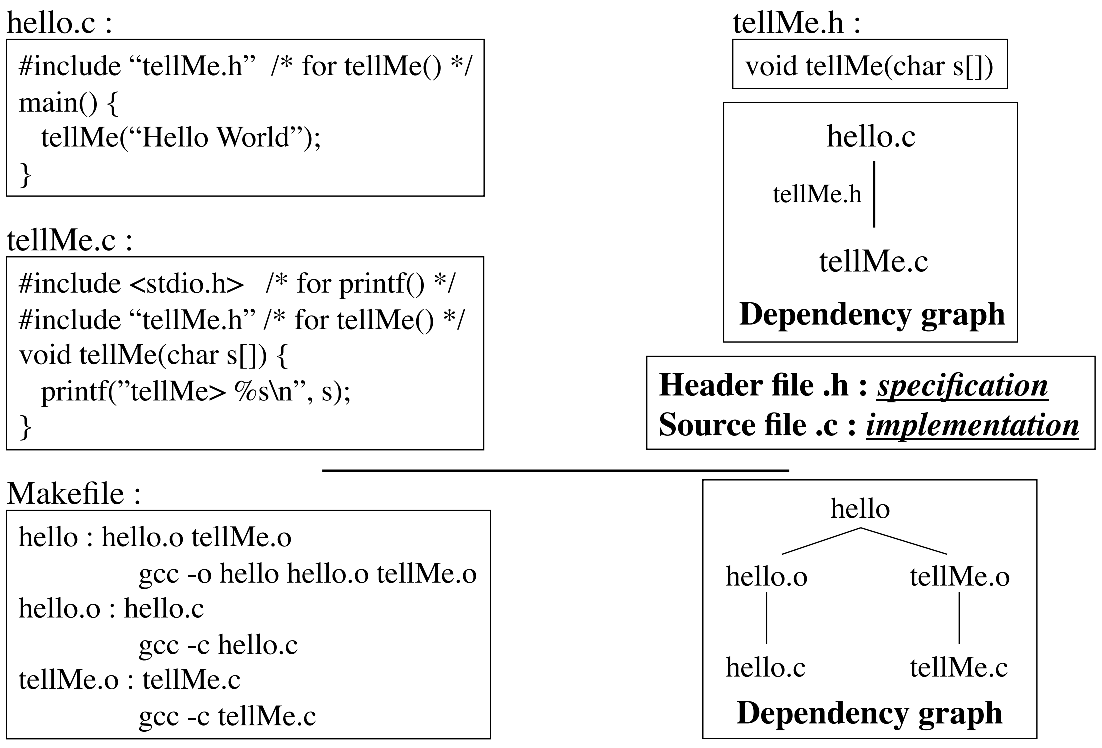
`tellMe_execBug.c` (#include "tellMe.h" is missing and function prototype is lightly different)
```
(08)  #include <stdio.h>
(09)  void tellMe(char c) {printf("tellMe> %d\n", c);}
```

tellMe_compBug (function prototype is lightly different)
```
(10)  #include <stdio.h>
(11)  #include "tellMe.h"
(12)  void tellMe(char c) {printf("tellMe> %d\n", c);}
```

and build the four programs 'hello_X' resp. with the files 'hello.c' and 'TellMe_X.c'.

<u>Question 4</u>. Explain what will happen resp. at compile and run time. Test your answer on machine.

Hint: The program 'hello_ok' is correct. But what about the others? For your convenience, these files and associated Makefile are available online on Moodle -> Tutorials -> C01 -> extras.



hello.c :
```
#include “tellMe.h”  /* for tellMe() */
main() {
  tellMe(“Hello World”);
}
```

tellMe.h :
```
void tellMe(char s[])
```

tellMe.c :
```
#include <stdio.h>   /* for printf() */
#include “tellMe.h” /* for tellMe() */
void tellMe(char s[]) {
  printf(”tellMe> %s\n”, s);
}
```

Header file .h : *specification*
Source file .c : *implementation*

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

Figure 3. Header Files : Hello World (revisited)

3. Make and Timestamp: about virtually and really retarding the clock

In this exercise we consider that the files 'hello.c' and 'Makefile' contain resp.:

```
#include <stdio.h>
main () {printf("hello\n");}

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

a) Now suppose you have just compiled the source file 'hello.c' and executed it with the commands :

```
(1)   % make
(2)   % ./hello
```

Then, in order to try other variants without loosing your actual program, you type the following commands:

```
(3)   % cp hello.c hello_old.c   # Look the timestamp of these two files
```

And finally you modify the program 'hello.c', i.e. by replacing the string "hello\n" by "hello world\n", recompile and execute it with :

```
(4)   % make
(5)   % ./hello
```

For some reason you are not satisfied and want to come back to your original version by typing :

```
(6)   % mv hello_old.c hello.c    # Look at hello.c's timestamp !
(7)   % make
(8)   % ./hello          # This is the modified program! What happened?
```

Retry with :

```
(9)   % touch hello.c
(10)  % make
(11)  % ./hello
```

Question 5. What is the information displayed by the commands (1) to (11) ? Explain and verify your answer on machine.

Hint : Note that in (6) the timestamp of 'hello.c' has been retarded (this is fully compatible with the semantics of mv). Another way to retard or advance a timestamp of a file can be done with the command "touch -t".

b) Suppose the following scenario with the given 'hello.c' and 'Makefile' files (and file 'hello' don't exists):

```
(20)  % make ; date
      Wed Apr 14 15:52:07 CEST 2004
(21)  % sudo date -s 1452  # retard the clock to 14:52, i.e. winter time
                           # under OSX 10/bash:  % sudo date 1452
(22)  % make
(23)  % touch hello.c  # Look the timestamp of hello.c and hello
(24)  % make
(25)  % make -W hello.c
```

Question 6 : Explain the behavior of the command make at lines (22), (24) and (25). Verify your answer on machine!

Hint : To execute a 'sudo' command you need administrator or super-user privileges!

Hint2 : Note that (22) and (23)-(24) are tricky and subject to unpredictable behaviors in conjunction with (21) !!

"MORALITY": NEVER RETARD THE CLOCK, neither virtually as in (a), nor really as in (b) and if you are not sure, use the command make with the option -W.