The “make” Utility in Unix

Recall the reverse Module

reverse.h

```
1 /* reverse.h */
2
3 void reverse (char before[], char after[]);
4      /* Declare but do not define this function */
```

Recall the reverse Module

reverse.c

```
1 /* reverse.c */
2 #include <stdio.h>
3 #include <string.h>
4 #include “reverse.h”
5
6 /*********************************************/
7
8 void reverse (char before[], char after[])
9
10 {
11   int i;
12   int j;
13   int len;
14
15   len = strlen (before);
16
17   for (j = len – 1, i=0; j>= 0; j--,i++) /*Reverse loop*/
18      after[i] = before[j];
19
20   after[len] = ‘\0’;  /* NULL terminate reversed string */
21  }
```

Recall the main program

- Here's a listing of a main program that uses reverse():

main1.c

```
1   /* main1.c */
2
3 #include <stdio.h>
4 #include "reverse.h" /*Contains the prototype of reverse)*/
5
6 /*****************************************/
7
8 void main ()
9
10 {
11 char str [100];
12
13 reverse ("cat", str); /* Invoke external function */
14 printf ("reverse (\"cat\") = %s\n", str);
15 reverse ("noon", str); /* Invoke external function */
16 printf ("reverse (\"noon\") = %s\n", str);
17 }
```

Compiling And Linking Modules Separately

- To compile each source code file separately, use the **-c** option of **gcc**. This creates a separate object module for each source code file, each with a ".o" suffix. The following commands are illustrative:

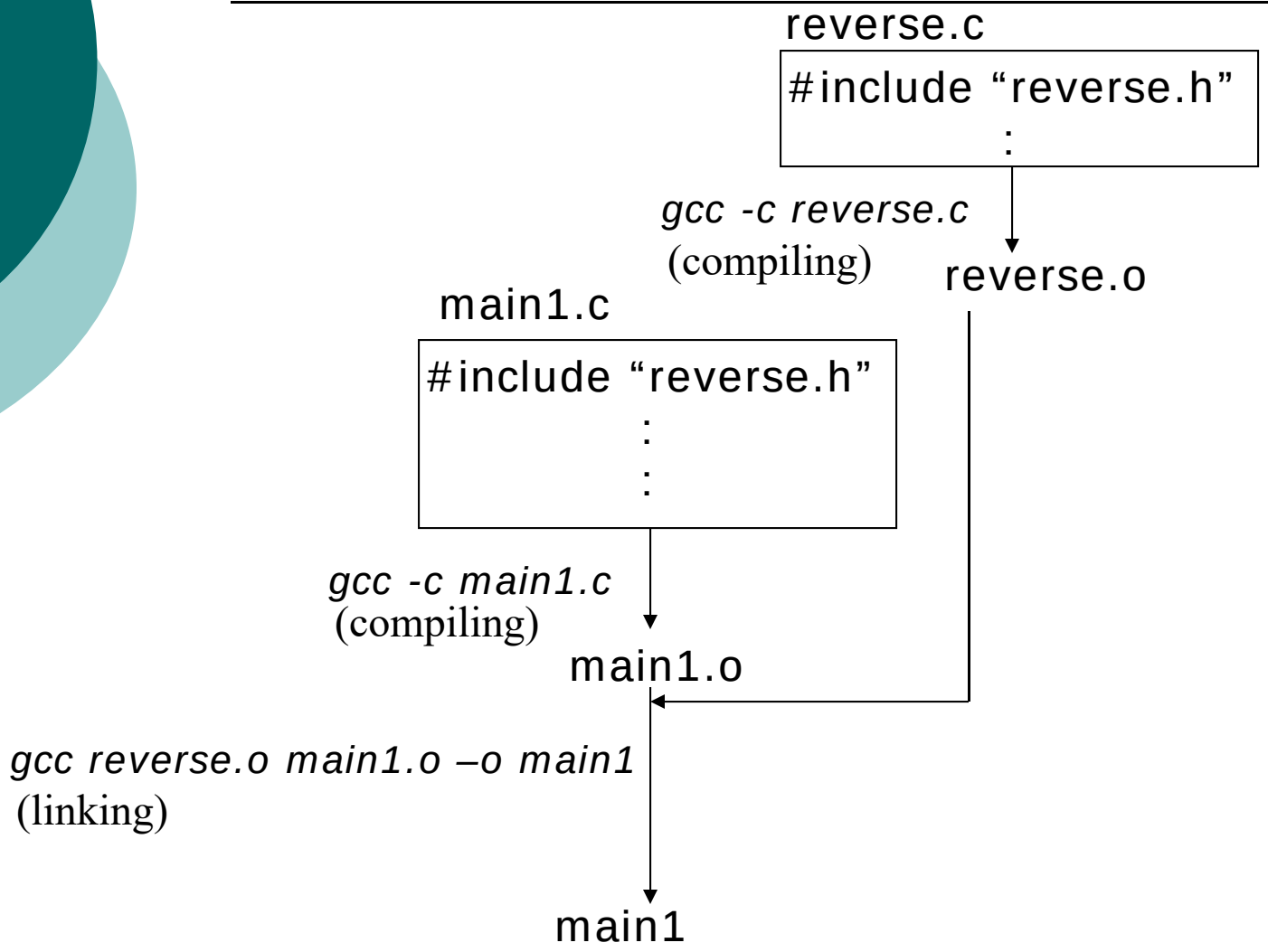
```
sepc92:> gcc -c reverse.c   ... compile reverse.c to reverse.o.
sepc92:> gcc -c main1.c     ... compile main1.c to main1.o.
sepc92:> ls -l reverse.o main1.o
-rw-r--r-- 1 glass   311 Jan 5 18:24 main1.o
-rw-r--r-- 1 glass   181 Jan 5 18:08 reverse.o
sepc92:> _
```

Compiling And Linking Modules Separately (con't)

To *link* them all together into an executable called "main1", list the names of all the object modules after the **gcc** command:

```
sepc92:> gcc reverse.o main1.o –o main1   …link object modules.
sepc92:> ls -l main1
-rwxr—xr-x 1 glass      24576 Jan 5 18:25 main1*
sepc92:> ./main1                          … run the executable.
reverse ("cat") = tac
reverse ("noon") = noon
sepc92:> _
```

Compiling And Linking Modules Separately – Facilitate Code Sharing



Reusing a Function

- Here's a listing of another program that uses reverse():

main8.c

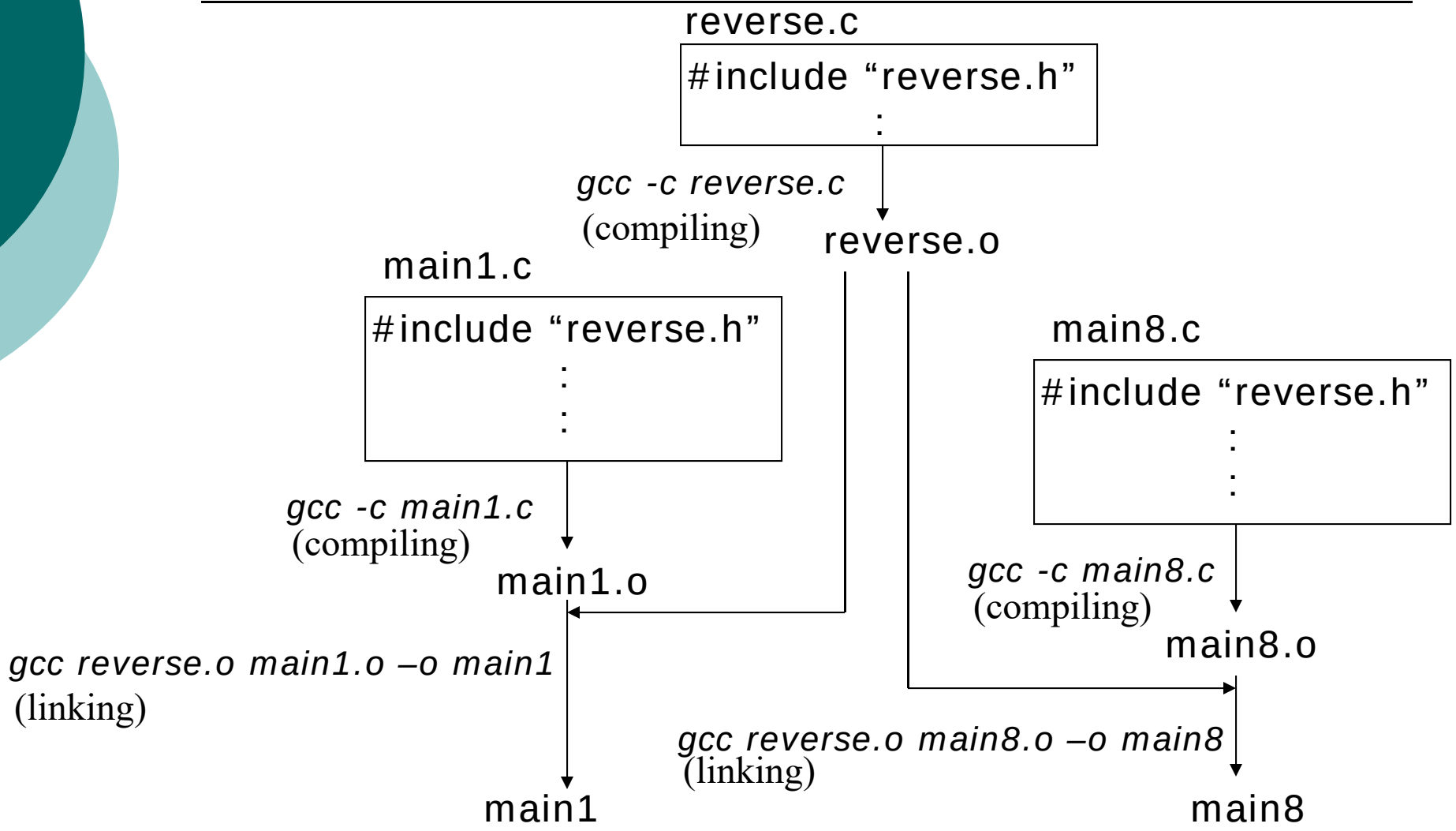
```
1   /* main8.c */
2
3 #include <stdio.h>
4 #include "reverse.h" /*Contains the prototype of reverse)*/
5
6 /*****************************************/
7
8 int main ()
9 {
10    char person[100];
11    reverse ("tom", person); /* Invoke external function */
12    strcat(person, " Berth");
13    printf ("person = (%s)\n", person);
14 }
```

Re-using a Function

- The reverse function (module) can also be used by other programs such as main8.c and the compilation is as follows.

sepc92:> *gcc -c main8.c* ... compile
sepc92:> *gcc reverse.o main8.o –o main8* ...link object modules

Compiling And Linking Modules Separately – Facilitate Code Sharing



Maintaining Multi-module Programs

- Several different issues must be considered in maintaining multi-module systems:
  1. What ensures that object modules and executable files are kept up to date?
  2. What stores the object modules?
  3. What tracks each version of source and header files?

- Fortunately, there are UNIX utilities that address each problem such as the **make** utility
- The **make** utility is the UNIX file dependency system

The Unix File Dependency System: make

- You've now seen how several independent object modules may be linked into a single executable file. You've also seen that the same object module may be linked to several different executable files.
- Although multi-module programs are efficient in terms of reusability and disk space, they must be carefully maintained.
- For example, let's assume that we change the source code of "reverse.c" so that it uses pointers instead of array subscripts. This would result in a faster reverse function.
- In order to update the two main program executable files, such as "main1" and "main8", manually, we'd have to perform the following steps, in order:

  1. Recompile "reverse.c".
  2. Link "reverse.o" and "main1.o" to produce a new version of main1".
  3. Link "reverse.o" and "main8.o" to produce a new version of "main8".

The Unix File Dependency System: make (con't)

- Similarly, imagine a situation where a **#define** statement in a header file is changed. All of the source code files that directly or indirectly include the file must be recompiled, and then all of the executable modules that refer to the changed object modules must be re-linked.
- Although this might not seem like a big deal, imagine a system with a thousand object modules and 50 executable programs. Remembering all of the relationships among the headers, source code files, object modules, and executable files would be a nightmare.
- One way to avoid this problem is to use the UNIX **make** utility, which allows you to create a makefile, with file name called *Makefile* that contains a list of all interdependencies for each executable file. Once such a file is created, re-creating the executable file is easy: You just use the **make** command

  cuse93:> *make*

The Unix File Dependency System: make (con't)

- *Utility:* **make** [-f *makefile*]
- **make** is a utility that updates a file on the basis of a series of dependency rules stored in a text file with a special format, "make file". The **-f** option allows you to specify your own **make** filename; if none is specified, the name "Makefile" or "makefile" is assumed.

Make Files

- To use the **make** utility to maintain an executable file, you must first create a text file known as a make file. Here the file name is “Makefile”.
- This file contains a list of all the interdependencies that exist between the files that are used to create the executable file.

Make Files

- In its simplest form, a make file contains *rules* of the form shown below

```
targetList:     dependencyList
                commandList
```

- where *targetList* is a list of target files and
- *dependencyList* is a list of files that the files in *targetList* depend on.
- *commandList* is a list of zero or more commands, separated by newlines, that reconstruct the target files from the dependency files.
- **Each line in *commandList* must start with a tab character.**
- *Rules* must be separated by at least one blank line.

Make Files (con't)

- For example, let's think about the file interdependencies related to the executable file "main1". This file is built out of two object modules: "main1.o" and "reverse.o". If either file is changed, then "main1" may be reconstructed by linking the files, using the gcc utility. Therefore, one rule in the text file "Makefile" would be:

```
main1: main1.o reverse.o
    gcc main1.o reverse.o –o main1
```

Make Files (con't)

- This line of reasoning must now be carried forward to the two object files.
- The file "main1.o" is built from two files: "main1.c" and "reverse.h". (Remember that any file which is either directly or indirectly #included in a source file is effectively part of that file.) If either file is changed, then "main1.o" may be reconstructed by compiling "main1.c".
- Here, therefore, are the remaining rules in "Makefile":

```
main1.o: main1.c reverse.h
	gcc -c main1.c

reverse.o: reverse.c reverse.h
	gcc -c reverse.c
```

Make Files (con't)

- As a result, the whole content of "Makefile" is:

```
main1: main1.o reverse.o
	gcc main1.o reverse.o -o main1

main1.o: main1.c reverse.h
	gcc -c main1.c

reverse.o: reverse.c reverse.h
	gcc -c reverse.c
```

The Order Of Make Rules

- The order of make rules is important. The make utility creates a “tree” of interdependencies by initially examining the first rule.
- Each target file in the first rule is a root node of a dependency tree, and each file in its dependency list is added as a leaf of each root node. In our example, the initial tree would look like Figure 12.4.

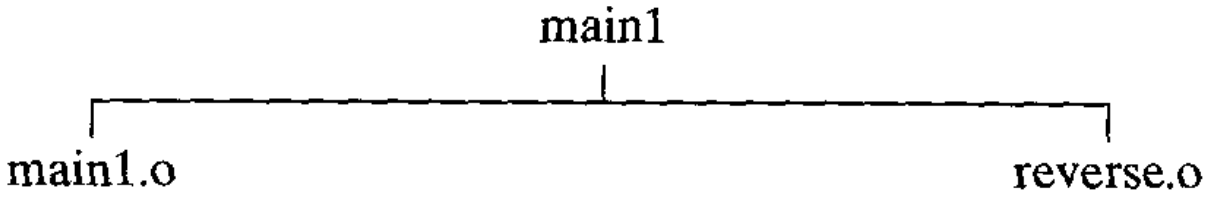


FIGURE 12.4

Initial **make** dependency tree.

The Order Of Make Rules (con't)

- The **make** utility then visits each rule associated with each file in the dependency list and performs the same actions. In our example, the final tree would, therefore, look like Figure *12.5*.

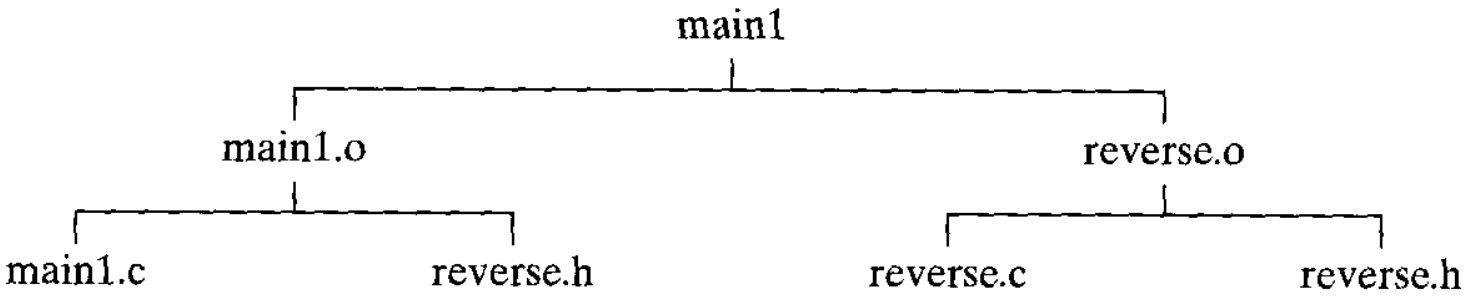
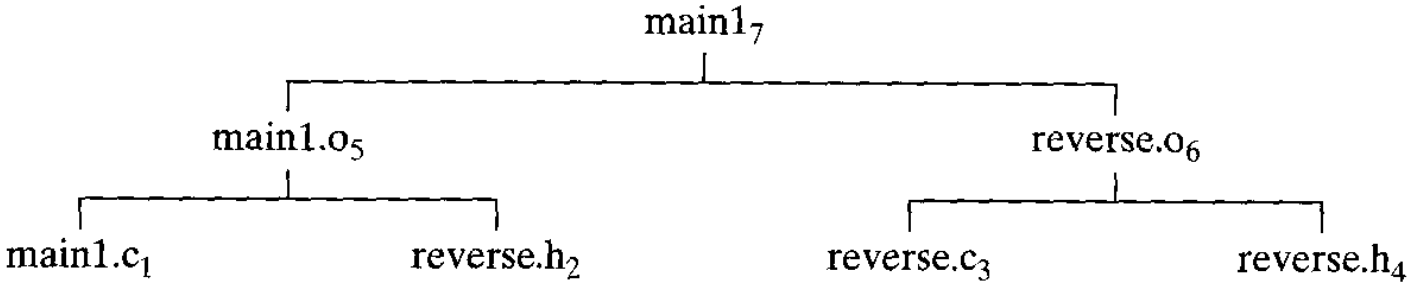


FIGURE 12.5

Final **make** dependency tree.

The Order Of Make Rules (con't)

- Finally, the **make** utility works up the tree from the bottom leaf nodes to the root node, looking to see if the last modification time of each child node is more recent than the last modification time of its immediate parent node.
- For every case where this is so, the associated parent's rule is executed.
- If a file is not present, its rule is executed regardless of the last modification times of its children.
- To illustrate the order in which the nodes would be examined, we show a number for each element as below:



Executing A Make

- Once a make file has been created, you're ready to run **make** to re-create the executable file whose dependency information is specified by the make file.
- To show you how this works, suppose we deleted all of the object modules and the executable file to force every command list to execute. When I then performed the make, here's what I saw:

```
sepc92:> make                    …make executable up-to-date
gcc -c main1.c
gcc -c reverse.c
gcc main1.o reverse.o -o main1
sepc92:> _
```

Reusing The Reverse Function for Building Another Re-usable Function

- Similar to previous example, the reverse module can be used to build the palindrome program. Here are the header and source code listing of the palindrome function:

palindrome.h

```
1 /* palindrome.h */
2
3 int palindrome (char str[]);
4       /* Declare but do not define */
```

Reusing The Reverse Function for Building Another Re-usable Function

palindrome.c

```
1 /* palindrome.c */
2
3 #include “palindrome.h”
4 #include “reverse.h”
5 #include <string.h>
6
7 /*********************************************/
8
9  int palindrome (char str[])
10
11 {
12     char reversedStr [100];
13     reverse (str, reversedStr); /* Reverse original */
14     return (strcmp (str, reversedStr) ==0);
15                         /* Compare the two */
16 }
```

Reusing The Reverse Function for Building Another Re-usable Function

- The program “main2.c” that tests the palindrome function

```
1 /* main2.c */
2
3 #include <stdio.h>
4 #include “palindrome.h”
5
6 /*************************************/
7
8 void main ()
9
10 {
11 printf (“palindrome (\“cat\”) = %d\n”, palindrome (“cat”));
12 printf (“palindrome (\“noon\”) = %d\n”, palindrome(“noon”));
13 }
```

Reusing The Reverse Function for Building Another Re-usable Function

```
sepc92:> gcc –c  palindrome.c      … compile palindrome.c to palindrome.o
sepc92:> gcc -c main2.c            … compile main2.c to main2.o
sepc92:> gcc reverse.o palindrome.o main2.o -o main2   … link them all.
sepc92:> ls  -l reverse.o palincdrome.o main2.o main2
-rwxr-xr-x 1 glass        24576 Jan 5 19:09 main2*
-rw-r--r--  1 glass          306 Jan 5 19:00 main2.o
-rw-r--r--  1 glass          189 Jan 5 18:59 palindrome.o
-rw-r--r--  1 glass          181 Jan 5 18:08 reverse.o
sepc92:> ./main2                          … run the program.
palindrome (“cat”) = 0
palindrome (“noon”) = 1
sepc92:> _
```

Another Make File

- Now we prepare a make file, with our own file name called “main2.make”. Here we will also utilize the make utility using this make file.

Executing A Make

- Notice that every make rule was executed. Since we created a second executable file when we made the palindrome program, we also fashioned a second make file, called “main2.make”. Here it is:

```
main2: main2.o reverse.o palindrome.o
	gcc main2.o reverse.o palindrome.o -o main2

main2.o: main2.c palindrome.h
	gcc -c main2.c

reverse.o: reverse.c reverse.h
	gcc -c reverse.c

palindrome.o:  palindrome.c palindrome.h reverse.h
	gcc -c palindrome.c
```

Executing A Make (con't)

- When we performed a make using this file, we saw the following output:

```
sepc92:> make -f main2.make      ...make executable up-to-date
gcc -c main2.c
gcc -c palindrome.c
gcc main2.o reverse.o palindrome.o -o main2
sepc92:> _
```

- Notice that "reverse.c" was not recompiled. This is because the previous make had already created an up-to-date object module,
- However, "reverse.c" will be compiled if it has been modified since the last compilation

Another Make for a Different Scenario

- The previous makefile assumes that reverse.c is available. A different scenario is that reverse.c is not available and only reverse.o is available.
- For the above scenario, we can prepare the makefile called “main2-team.make” as shown below.
- For the content of makefile, we can give comments by starting the line with #

- # This scenario assumes that only reverse.o is available

```
main2-team: main2-team.o reverse.o palindrome.o
        gcc main2.o reverse.o palindrome.o -o main2-team

main2-team.o: main2.c palindrome.h
        gcc -c main2.c –o main2-team.o

palindrome.o:  palindrome.c palindrome.h reverse.h
        gcc -c palindrome.c
```

Executing A Make (con't)

- The following output can be observed when we perform the make utility:

```
sepc92:> make -f main2-team.make
gcc -c main2.c –o main2-team.o
gcc -c palindrome.c
gcc main2-team.o reverse.o palindrome.o -o main2-team
sepc92:> _
```

Arguments to Specify the Goal

- Consider the makefile main2.make. We can also put the commands for compiling main1 (discussed at the beginning of this topic) into this makefile. The updated makefile is called main.make.
  - One advantage is that the step of the compilation of reverse.c can be shared.
- We can make use of the notion of *goals* in the makefile.

Arguments to Specify the Goal

- The updated makefile called “main.make” is:

```
main1: main1.o reverse.o
	gcc main1.o reverse.o -o main1

main1.o: main1.c reverse.h
	gcc -c main1.c

main-target: main2.o reverse.o palindrome.o
	gcc main2.o reverse.o palindrome.o -o main2

main2.o: main2.c palindrome.h
	gcc -c main2.c

reverse.o: reverse.c reverse.h
	gcc -c reverse.c

palindrome.o:  palindrome.c palindrome.h reverse.h
	gcc -c palindrome.c
```

Arguments to Specify the Goal

- In main.make, some examples of goals are: main1, main-target
- The goals are the targets that make should strive ultimately to achieve.
- The basic command for specifying the goal is:

make *goal*

sepc92:> *make main1 -f main.make*
gcc -c main1.c
gcc -c reverse.c
gcc main1.o reverse.o –o main1

- Now main1 is ready for execution

sepc92:> rm *.o * delete all object files for demonstrate purpose
sepc92:> *make main-target -f main.make*
gcc -c main2.c
gcc –c reverse.c
gcc -c palindrome.c
gcc main2.o reverse.o palindrome.o –o main2

Arguments to Specify the Goal

- Notice that other targets are processed as well if they appear as prerequisites of goals in a similar way to the processing of the order of make rules.
- In general, the name of the goal or target is not necessarily the same as the name of the generated file or output file of the Unix command.
  - However, in such cases, the “make” utility cannot check the modification time properly.
- By default, the goal is the first target in the makefile.

The touch Utility

- To confirm that the new version of the make file worked, we requested a **make** and obtained the following output:

```
sepc92:> make main2 -f main.make
make: 'main2' is up to date.
sepc92:> _
```

- Obviously, since we'd already performed a successful **make,** another one wasn't going to trigger any rules!
- To force a **make** for testing purposes, we can use a handy utility called **touch,** which makes the last modification time of all the named files equal to the current system time.

The touch Utility (con't)

- *Utility:* **touch -c** { *filename* }+

- **touch** updates the last modification and access times of the named files to equal the current time. By default, if a specified file doesn't exist, it is created with zero size. To prevent this, use the -c option.

- I **touch**ed the file "reverse.h", which subsequently caused the recompilation of several source files:

```
sepc92:> touch reverse.h                    … fool make
sepc92:> make -f main2.make
gcc -c -o reverse.c
gcc -c -o palindrome.c
gcc main2.o reverse.o palindrome.o -o main2
sepc92:> _
```