C Programming Tools

- Read Chapter 11
- Linux typically comes with compilers
  - GNU C (gcc)
    - (the old cc compiler is linked to gcc)
  - GNU C++ (g++)

C Programming Tools

- Utility: *gcc* -cv [-o fileName] [-pg] { fileName }*
  - *gcc* compiles C program code in one or more files and produces object modules or an executable file.
  - Files specified should have a ".c" extension.
  - -c option to produce object modules suitable for linking later
  - -o option to specify a filename other than the default "a.out"
  - -pg option to produce profiling data for the GNU profiler *gprof*.
  - -v option to produce verbose commentary during the compilation and/or linking process.

C Programming Tools

- Separately compiling and linking
  - using gcc with -c option allows us to compile files separately
  - the output are .o files

```
$ gcc -c reverse.c     ...compile reverse.c to reverse.o.
$ gcc -c main1.c       ...compile main1.c to main1.o.
$ ls -lG reverse.o main1.o
-rw-r--r--  1 ables      311 Jan  5 18:08 main1.o
-rw-r--r--  1 ables      181 Jan  5 18:08 reverse.o
$ _
```

Alternatively, you can place all of the source code files on one line:

```
$ gcc -c reverse.c main1.c      ...compile each .c file to .o file.
$ _
```


C Programming Tools

- Linking .o files into an executable
  - we can use gcc for that

```
$ gcc reverse.o main1.o -o main1   ...link object modules.
```

  - note: in unix environment one often uses the stand-alone linking loader (ld) to link separate modules.
  - gcc can do the same
    - use -v option to see how gcc works

Multimodule Programs

- Motivation
  - assume reverse program from book is to be used in other programs
    - e.g., use reverse to check for palindrome
  - could copy-and-paste
    - tedious if we wanted to change the function: would have to change every instant of the function, besides
      - copy-and-paste operation is tedious
      - waste of disk space

Multimodule Programs

- Reusable Functions
  - remove function of interest from program
  - compile separately
  - link resultant object code to programs that want to us it

Multimodule Programs

- Preparing a reusable function
  - Create a source code module that contains the source code of the function
  - Create header file that contains the function's prototype.
  - Then compile it into an object module by using the -c option of gcc.
    - An object module contains machine code together with symbol-table information that allows it to be combined with other object modules when an executable file is being created.
    - recall the -c option means *Compile or assemble the source files, but do not link*.

Multimodule Programs

- example reverse.h

```
1  /* REVERSE.H */
2
3  int reverse (); /* Declare but do not   */
                   /* define this function */
```

Multimodule Programs

- example reverse.c

```
1  /* REVERSE.C */
2
3  #include <stdio.h>
4  #include "reverse.h"
5
6  /*****************************************************************/
7
8  reverse (before, after)
9
10  char *before; /* A pointer to the original string */
11  char *after; /* A pointer to the reversed string */
12
13  {
14    int i;
15    int j;
16    int len;
17
18    len = strlen (before);
19
20    for (j = len - 1, i = 0; j >= 0; j--, i++) /* Reverse loop */
21      after[i] = before[j];
22
23    after[len] = 0; /* terminate reversed string */
24  }
```


Multimodule Programs

- example main1.c

```
1  /* MAIN1.C */
2
3  #include <stdio.h>
4  #include "reverse.h" /* Contains the prototype of reverse () */
5
6  /*************************************************************/
7
8  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  }
```

Multimodule Programs

- Now compile

```
$ gcc -c reverse.c main1.c
                      ...compile each .c file to .o file.
```

- link

```
$ gcc reverse.o main1.o -o main1
                       ...link object modules.
```

- and execute

```
$ ./main1           ...run the executable.
reverse ("cat") = tac
reverse ("noon") = noon
$ _
```


Multimodule Programs

- Example: check whether word is a palindrome

- example palindrome.h

```
1  /* PALINDROME.H */
2
3  int palindrome (); /* Declare but do not define */
```

Multimodule Programs

- example palindrome.c

```
1  /* PALINDROME.C */
2
3  #include "palindrome.h"
4  #include "reverse.h"
5  #include <string.h>
6
7  /*************************************************************/
8
9  int palindrome (str)
10
11  char *str;
12
13  {
14    char reversedStr [100];
15    reverse (str, reversedStr); /* Reverse original */
16    return (strcmp (str, reversedStr) == 0); /* Compare the two */
17  }
```


Multimodule Programs

- example main2.c

```
1  /* MAIN2.C */
2
3  #include <stdio.h>
4  #include "palindrome.h"
5
6  /*************************************************************/
7
8  main ()
9
10  {
11    printf ("palindrome (\"cat\") = %d\n", palindrome ("cat"));
12    printf ("palindrome (\"noon\") = %d\n", palindrome ("noon"));
13  }
```

Multimodule Programs

- run

```
$ gcc -c palindrome.c  ...compile palindrome.c to palindrome.o.
$ gcc -c main2.c       ...compile main2.c to main2.o.
$ gcc reverse.o palindrome.o main2.o -o main2    ...link them.


$ ./main2                          ...run the program.
palindrome ("cat") = 0
palindrome ("noon") = 1
$ _
```


Archiving Modules: ar

- Utility: ar key archiveName { fileName }*
  - ar allows you to create and manipulate archives. The archive file should end with a ".a" suffix. key may be:
    - d - deletes a file from an archive
    - q - appends file to archive, even if it's already present
    - r - adds a file to an archive if it isn't already there, or replaces the current version if it is
    - s - builds index (table of contents) of library for faster access
    - t - displays an archive's table of contents to standard output
    - x - copies a list of files from an archive into the current directory
    - v - generates verbose output

make

- The rules in a make file tells *make* how to execute commands to build a target file from source files.
- It also specifies a list of dependencies of the target file.
- Make files can contain comments.
  - Comments start with a # and are used to describe what is happening in the makefile or to hide definitions from

make

- Utility: make [-f makefile]
  - make is a utility that updates a file based on a series of dependency rules stored in a special format "make file".
  - The -f option allows you to specify your own make filename
    - if none is specified, make will look for the files "GNUmakefile," "makefile," and "Makefile," in that order.

make

- Figure 11-9. make dependency specification.

  targetList:dependencyList

  commandList

  - *targetList* is a list of target files
  - *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 reconstructs 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

- example from book: reverse
  - two object modules: *main1.o* and *reverse.o*
  - executable will be called *main1*
  - if either file is changed then *main1* may be reconstructed by linking the files using gcc. Thus, one rule in the make file would be:

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

make

- example from book: reverse
  - main1.o is built from two files: main1.c and reverse.h
  - here are the remaining rules of our make file

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

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


make

- summary: main1.make:

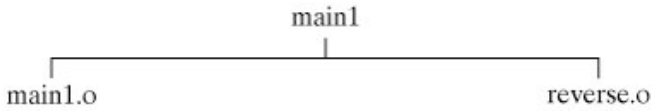
```
main1: main1.o reverse.o
            gcc main1.o reverse.o -o main1
main1.o:    main1.c reverse.h
            gcc -c main1.c

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

make

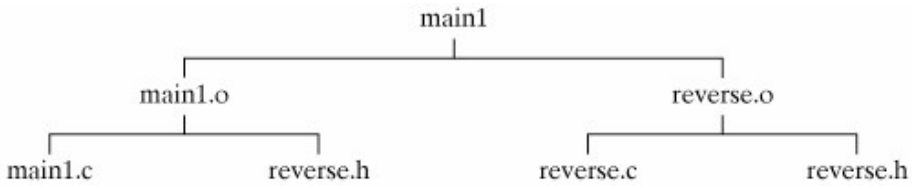
- The order of Make Rules
  - *make* creates tree of interdependencies by first examining the first rule
  - each target file in the first rule is root node of dependence tree
  - each file in its dependence list is added as a leaf of each root node

make

- The order of Make Rules
  - initial *make* dependence tree

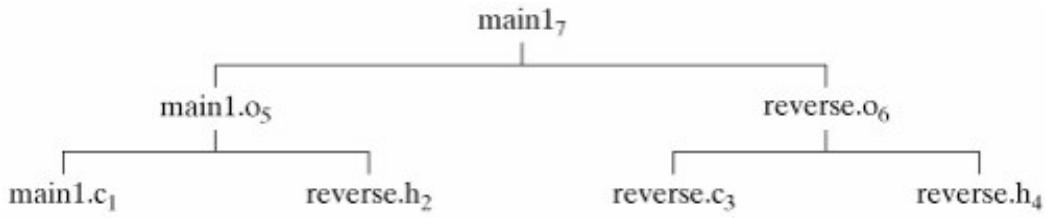


  - final *make* dependence tree



make

- The order of Make Rules
  - *make* ordering



make

- Running Make

```
$ make -f main1.make     ...make executable up-to-date.
gcc -c main1.c
gcc -c reverse.c
gcc main1.o reverse.o -o main1
$ _
```

make

- example palindrome: make2.make

```
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
```

- and “make”

```
$ 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
```

- note that reverse.c was not recompiled

make

- Make Rules
  - the previous rules were over-complicated of form:

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

  - the **make** utility contains a predefined rule similar to the following:

```
.c.o:
            gcc -c -O $<
```

  - this allows to leave off the C recompilation rule

make

- Make Rules
  - new file: main2.make

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

  - make also has inference rule that deduces file xxx.o is dependent of xxx.c

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


make

- Forcing Compilation
  - check if we have newest version

```
$ make -f main2.make
'main2' is up to date.
$ _
```

  - can use *touch* to update last modification time of file
  - Utility: *touch* -c { fileName }+
    - touch updates the last modification and access times of the named files to 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.

make

- Forcing Compilation
  - after *touch* of file reverse.h every file that depends on it will be recompiled

```
$ touch reverse.h        ...fool make.
$ make -f main2.make
gcc -c -O reverse.c
gcc -c -O palindrome.c
gcc main2.o reverse.o palindrome.o -o main2
$ _
```


make

- Macros
  - make supports simple macros of form:

    token = *replacementText*
  - If you specify such a line at the top of a make file, every occurrence of $(*token*) in the make file is replaced by *replacementText*.

make

- Macros
  - standard rules file contains default definitions of macros, e.g.
  - CFLAG, which is used by some built-in rules, e.g.,

```
.c.o:
        gcc -c $(CFLAGS) $<
```

    - the rule that tells the make utility how to update an object file from a C source file

make

- Macros
  - example

```
CFLAGS =        -O2
main2:          main2.o reverse.o palindrome.o
                gcc main2.o reverse.o palindrome.o -o main2
main2.o:        palindrome.h
reverse.o:      reverse.h
palindrome.o:   palindrome.h reverse.h
```

  - recompile the suit of programs

```
$ touch *.c       ...force make to recompile everything.
$ make -f main2.make
gcc -O2 -c main2.c
gcc -O2 -c palindrome.c
gcc -O2 -c reverse.c
gcc main2.o reverse.o palindrome.o -o main2
$ _
```