Makefiles

Multiple Source Files (1)

- Obviously, large programs are not going to be contained within single files.
- C provides several techniques to ensure that these multiple files are managed properly.
  - These are not enforced rules but every good C programmer know how to do this.
- A large program is divided into several modules, perhaps using abstract data types.
- The header (.h) file contains function prototypes of a module.
- The (.c) file contains the function definitions of a module.
- Each module is compiled separately and they are linked to generate the executable file.

Multiple Source Files (2)

- C programs are generally broken up into two types of files.
  - .c files:
    - contain source code (function definitions) and global variable declarations
    - these are compiled once and never included
  - .h files:
    - these are the “interface” files which “describe” the .c files
      - type and struct declarations
      - extern const and #define constant declarations
      - #includes of other header files that must be included
      - prototypes for functions

Example - Main Program sample.c

```
#include <stdio.h>
#include "my_stat.h"
int main()
{
    int a, b, c;
    puts("Input three numbers:");
    scanf("%d %d %d", &a, &b, &c);
    printf("The average of %d %d %d is %f.\n",
            a, b, c, average(a,b,c));
    return 0;
}
```

Example - Module my_stat

```
/* my_stat.h */
#define PI 3.1415926
float average(int x, int y, int z);
float sum( int x, int y, int z);
```

```
/* my_stat.c */
#include "my_stat.h"
float average(int x, int y, int z)
{
    return sum(x,y,z)/3;
}

float sum(int x, int y, int z)
{
    return x+y+z;
}
```

Example - Compile the Sample Program

- You need my_stat.c and my_stat.h to compile the my_stat module to object code

  cc –c my_stat.c

- You need my_stat.h and sample.c to compile sample.c to object code

  cc –c sample.c

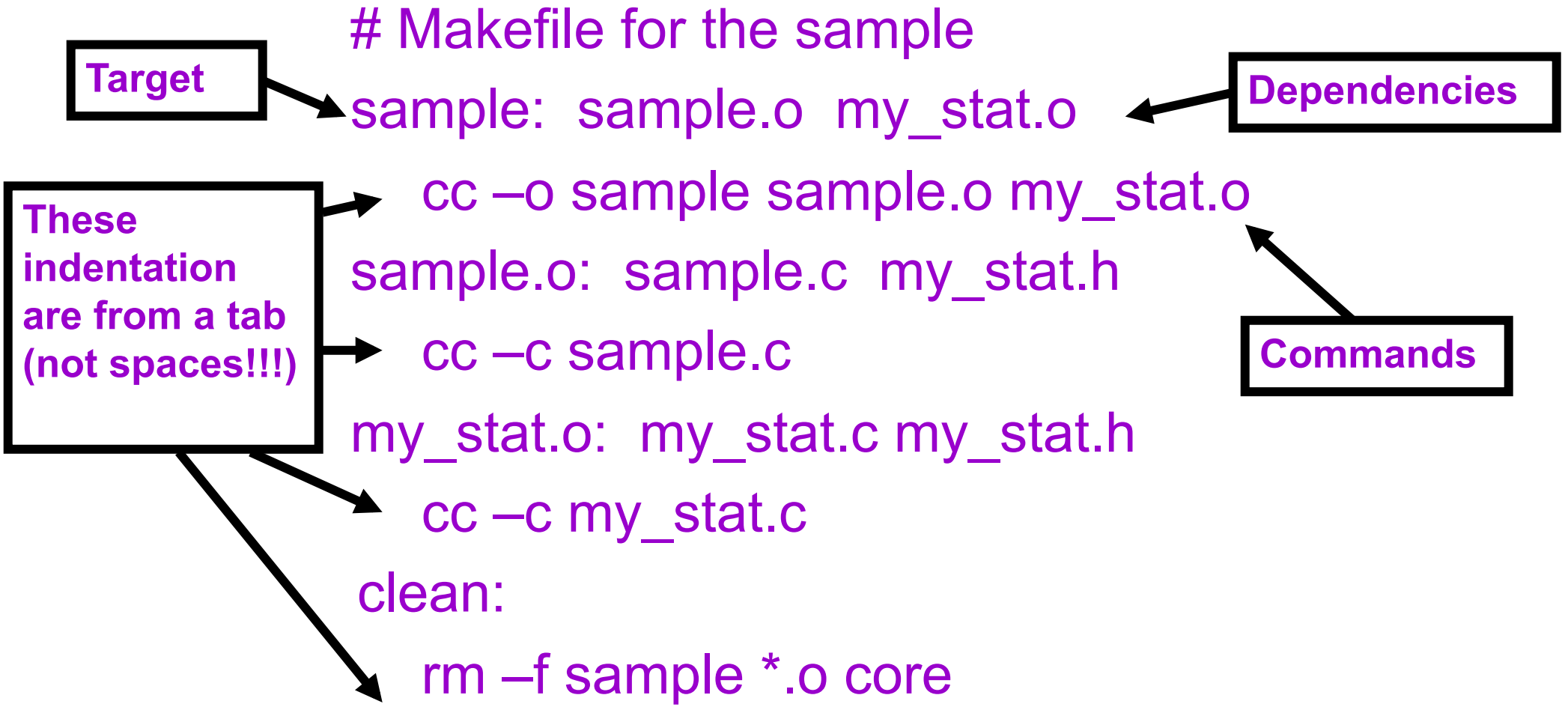
- You need my_stat.o and sample.o to generate an executable file

  cc –o sample sample.o my_stat.o

- Therefore, the module my_stat can be reused just with the my_stat.o and my_stat.h. In fact, this is how the standard libraries work. (Libraries are just collections of object code, with headers describing functions and types used in the libraries.)

The make Utility (1)

- Programs consisting of many modules are nearly impossible to maintain manually.
- This can be addressed by using the make utility.



The make Utility (2)

- Save the file with name "Makefile" (or "makefile") at the same directory.
- For every time you want to make your program, type

  make
- The make utility will
  - Find the Makefile
  - Check rules and dependencies to see if an update is necessary.
  - Re-generate the necessary files that need updating.
- For example:
  - If only sample.c is newer than sample, then only the following commands will be executed:
    - cc –c sample.c
    - cc –o sample sample.o my_stat.o

The make Utility (3)

- To clean all generated files:

  make clean

- To re-compile, you can
  - Remove all generated files and make again.
    - make clean; make
  - Or you can:
    - touch my_stat.h and then make again
    - This changes the time stamp of my_stat.h, so make thinks it necessary to make all the files.

Using make with Several Directories

- As the number of .c files for a program increases, it becomes more difficult to keep track of all the parts.
- Complex programs may be easier to control if we have one Makefile for each major module.
- A program will be stored in a directory that has one subdirectory for each module, and one directory to store *all* the .h files.
- The Makefile for the main program will direct the creation of the executable file.
- Makefiles for each module will direct the creation of the corresponding .o files.

A Makefile Example (1)

- Consider a C program that uses a Stack ADT, a Queue ADT and a main module.
- Suppose that the program is in seven files: StackTypes.h, StackInterface.h, QueueTypes.h, QueueInterface.h, StackImplementation.c, QueueImplementation.c, and Main.c
- We will build the program in a directory called Assn that has four subdirectories: Stack, Queue, Main, and Include.
- All four .h files will be stored in Include.

A Makefile Example (2)

- Stack contains StackImplementation.c and the following Makefile:

```
export: StackImplementation.o

StackImplementation.o: StackImplementation.c \
                       ../Include/StackTypes.h \
                       ../Include/StackInterface.h
        gcc -I../Include -c StackImplementation.c
# substitute a print command of your choice for lpr below
print:
        lpr StackImplementation.c
clean:
        rm -f *.o
```

A Makefile Example (3)

- Queue contains QueueImplementation.c and the following Makefile:

```
export: QueueImplementation.o

QueueImplementation.o: QueueImplementation.c \
                       ../Include/QueueTypes.h \
                       ../Include/QueueInterface.h
        gcc -I../Include -c QueueImplementation.c
# substitute a print command of your choice for lpr below
print:
        lpr QueueImplementation.c
clean:
        rm -f *.o
```

A Makefile Example (4)

- Note: The -I option (uppercase i) for cc and gcc specifies a path on which to look to find .h files that are mentioned in statements of the form #include "StackTypes.h" in .c files.
- It is possible to specify a list of directories separated by commas with -I.
- By using -I, we can avoid having to put copies of a .h file in the subdirectories for every .c file that depends on the .h file.

A Makefile Example (5)

- Main contains Main.c and the following Makefile:

```
export: Main

Main: Main.o StackDir QueueDir
        gcc -o Main Main.o ../Stack/StackImplementation.o \
                ../Queue/QueueImplementation.o
Main.o: Main.c ../Include/*.h
        gcc -I../Include -c Main.c
StackDir:
        (cd ../Stack; make export)
QueueDir:
        (cd ../Queue; make export)

#continued on next page...
```

A Makefile Example (6)

```
print:
        lpr Main.c
printall:
        lpr Main.c
        (cd ../Stack; make print)
        (cd ../Queue; make print)

clean:
        rm -f *.o  Main  core
cleanall:
        rm -f *.o  Main  core
        (cd ../Stack; make clean)
        (cd ../Queue; make clean)
```

A Makefile Example (7)

- Note: When a sequence of Unix commands is placed inside parentheses (), a new subprocess is created, and the commands are executed as part of that subprocess.
- For example, when (cd ../Stack; make export) is executed, the subprocess switches to the Stack directory and executes the make command; when the subprocess terminates, the parent process resumes in the original directory. No additional cd command is needed.

Using Macros in Makefiles

- Macros can be used in Makefiles to reduce file size by providing (shorter) names for long or repeated sequences of text.
- Example: The definition name = text string creates a macro called name whose value is text string.
- Subsequent references to $(name) or ${name} are replaced by text string when the Makefile is processed.
- Macros make it easier to change Makefiles without introducing inconsistencies.

Makefile Example Revisited (1)

- The Makefile for Stack can become:

```
CC = gcc
HDIR = ../Include
INCPATH = -I$(HDIR)
DEPH =  $(HDIR)/StackTypes.h  $(HDIR)/StackInterface.h
SOURCE = StackImplementation
export:  $(SOURCE).o

$(SOURCE).o:  $(SOURCE).c  $(DEPH)
        $(CC)  $(INCPATH)  -c  $(SOURCE).c
print:
        lpr  $(SOURCE).c
clean:
        rm  -f  *.o
```

Makefile Example Revisited (2)

- The Makefile for Queue can become:

```
CC = gcc
HDIR = ../Include
INCPATH = -I$(HDIR)
DEPH =  $(HDIR)/QueueTypes.h  $(HDIR)/QueueInterface.h
SOURCE = QueueImplementation

export:  $(SOURCE).o

$(SOURCE).o:  $(SOURCE).c  $(DEPH)
        $(CC)  $(INCPATH)  -c  $(SOURCE).c
print:
        lpr  $(SOURCE).c
clean:
        rm  -f  *.o
```

Makefile Example Revisited (3)

- The Makefile for Main.c can become:

```
CC = gcc
HDIR = ../Include
INCPATH = -I$(HDIR)
OBJ = ../Stack/StackImplementation.o   \
          ../Queue/QueueImplementation.o

export: Main

Main: Main.o  StackDir  QueueDir
          $(CC)  -o  Main  Main.o  $(OBJ)

#continued on next page...
```

Makefile Example Revisited (4)

```
Main.o: Main.c  $(HDIR)/*.h
        $(CC)  $(INCPATH)  -c  Main.c
StackDir:
        (cd ../Stack; make export)
QueueDir:
        (cd ../Queue; make export)
print:
        lpr  Main.c
printall:
        lpr  Main.c
        (cd ../Stack; make print)
        (cd ../Queue; make print)

# continued on next page...
```

Makefile Example Revisited (5)

```
clean:
        rm  -f  *.o  Main  core
cleanall:
        rm  -f  *.o  Main  core
        (cd ../Stack; make clean)
        (cd ../Queue; make clean)
```

A Makefile Exercise

- Rewrite the Makefiles for the previous example so that the command make debug will generate an executable Maingdb that can be run using the debugger gdb.

More Advanced Makefiles

- Many newer versions of make, including the one with Solaris and GNU make program gmake include other powerful features.
  - Control structures such as conditional statements and loops.
  - Implicit rules that act as defaults when more explicit rules are not given in the Makefile.
  - Simple function support for transforming text.
  - Automatic variables to refer to various elements of a Makefile, such as targets and dependencies.

- See the following web page for more details on gmake: http://www.gnu.org/software/make/manual/make.html

How Targets are Made: a recursive procedure

tagt:

commands

(base case:)

If tagt is not a file, execute commands

tagt: dep_1 dep_2 ⋯ dep_n

commands

(general case:)

for i from 1 to n

if dep_i is a target, make dep_i

If tagt is not a file, execute commands

else if at least one of dep_i is not a file, execute commands

else if at least one of dep_i is newer than tagt, execute commands