UNIX Make

CS2023 Winter 2004

Outcomes: Make

- *Managing Projects with Make*, Andrew Oram and Steve Talbott, on reserve in the library
- After the conclusion of this section you should be able to
  - Understand the purpose of the *make* utility
  - Write your own simple makefiles

Multi-File example

square.c:

```
#include "square.h"

double square(double x)
{
   return x * x;
}
```

square.h:

```
double
  square(double x);
```

square_test.c:

```
#include <stdio.h>
#include "square.h"

int main()
{

int i;
scanf("%d", &i);
printf("%g\n",
  square(i))

return 0;
}
```

Dependencies

```
gcc -c square.c
gcc -c square_test.c
gcc -o square square_test.o square.o
```

- Which commands must be re-executed if a change is made to **`square.c`**?
- We say that **`square.o`** depends on **`square.c`**
- Also, **`square`** depends on **`square_test.o`** and **`square.o`**
- Which commands must be re-executed if a change is made to **`square.h`**?

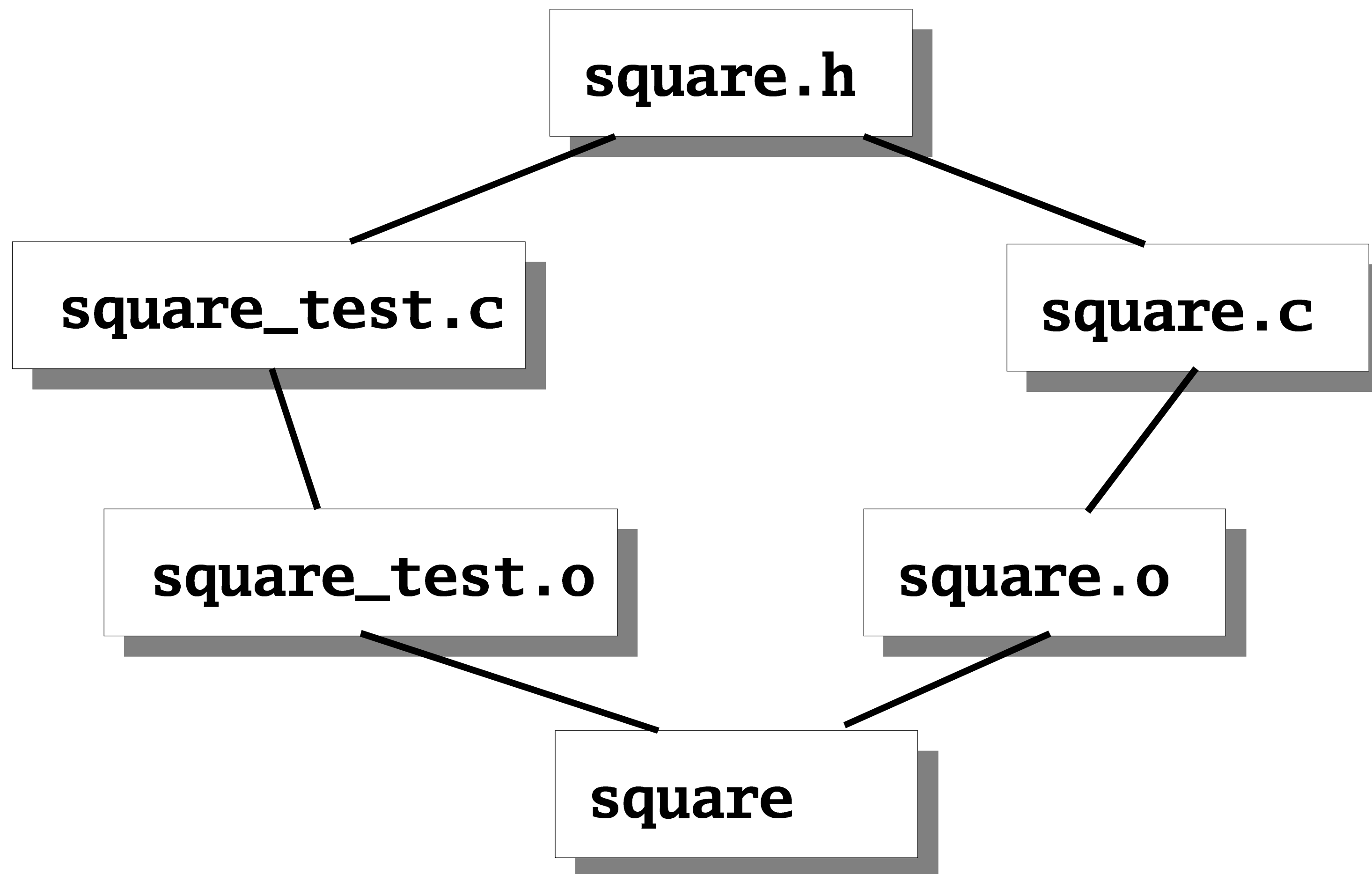
Source of Error

- What should be recompiled?
- Have you ever made a change to one or two of many files making up a program and forgot to recompile before linking? Bug is difficult to find because actual program does not correspond to the source files which you're looking at.

Possible Solutions

- Keep everything in one file
  - editing is slow
  - compiling is slow
- Keep things in separate files but always recompile everything
  - compiling is slow
- Automatically choose things to be recompiled
  - use *make* utility

Dependency Diagram



UNIX Software Development Environment

" The UNIX operating system earned its reputation above all by providing an unexcelled environment for software development. The *make* and *sccs* (*rcs*) utilities are widely regarded as the greatest contributors to the efficiency of this environment. Although the immense growth of the computer industry and the increasing scale of software projects reveal limitations in these tools, most of their potential successors are just extensions along the lines of principles established by *make* and *sccs*."

– *Managing Projects with make*, Andrew Oram and Steve Talbott

make

- From the *make* man page:
  - The purpose of the make utility is to determine automatically which pieces of programs need to be recompiled and issues the commands to recompile them.
- Using a description file (*makefile*) *make* creates a sequence of commands for execution by the UNIX shell

Description File

- Using an editor, create a file named:

  `makefile`

  or

  `Makefile`

- This file specifies two things:
  - the dependencies
  - the shell commands necessary to make a new version of a file

Using make

After editing any source file, just type:

```
make
```

and the appropriate commands will be executed

makefile

```
square: square_test.o square.o
→ gcc -o square square_test.o square.o
square_test.o: square_test.c square.h
→ gcc -c square_test.c
square.o: square.c square.h
→ gcc -c square.c
```

→ : represents a tab character

Steps for Efficient Software Development

- Create a directory for program development
- Create .h and .c files
- Create Makefile
- Type make

  errors → revise files

- Run program
- Done

make target

- You can specify several targets in a single makefile
- By default the first target is the only one examined (along with any rules for its dependencies)
- Can build any target in the description file:

  make *target*

- If no prerequisite files modified since last time *target* was created, *make* issues the message:

  '*target*' is up to date

make macros

- Real-life description files are much more succinct than our previous example, due to:
  - macros
  - suffix rules
- Simple macro

  token = replacement text
- To recall a macro

  $(token)

Example makefile with macro

```
OBJS = square_test.o square.o
CFLAGS = -Wall
square: $(OBJS)
	gcc -o square $(OBJS)
square_test.o: square_test.c square.h
	gcc -c square_test.c $(CFLAGS)
square.o: square.c square.h
	gcc -c square.c $(CFLAGS)
```

make macros

- Macro names are uppercase by convention
- You can use macros in macro definitions

```
ABC = XYZ
FILE = TEXT.$(ABC)
```

- Macro definitions that have no string after "=" are assigned null string
- You don't have to worry about the order in which you define macros

Internally defined macros

- **$(CC)**: recognized as C compiler

```
basic.o: basic.c
    $(CC) -c basic.c
```

same as:

```
basic.o: basic.c
    cc -c basic.c
```

- Note that **cc** typically linked to **gcc** command in Linux
- Can redefine internally defined macros:

```
CC = /home/fred/bin/cc
```

- C compiler and linker flags: **CFLAGS, LDFLAGS**

Internally defined macros

- `$@` macro evaluates to current target

```
plot_prompt: basic.o prompt.o
	cc -o $@ basic.o prompt.o
```

make suffix rules

- Can simplify description file even further
- C language source files always have **`.c`** suffix, Fortran source files have **`.f`** suffix.
- C and Fortran compilers automatically place object modules in **`.o`** files
- *make* uses these and other conventions

make example revisited

```
OBJS = square_test.o square.o
CFLAGS = -Wall
square: $(OBJS)
	$(CC) -o $@ $(OBJS) $(CFLAGS)
square_test.o: square.h
square.o: square.h
```

make and header files

- Keeping track of all the header files can be tricky
- Can be automated (see *Managing Projects with Make*, pp. 85-87
- **gcc -M** runs only the preprocessor and produces a dependency list suitable for make

```
% gcc -M square.c
square.o: square.c square.h
```

make all target

- We said *make* by default only processes the first target rule
- What if we want to compile several independent targets?
- We can create a special target rule as the first rule
- By convention this target is called "all"

```
all: $(PROGS)
```

make all target

- This will work fine as long as we define the macro **PROGS** to equal the names of all the executables we want to create

```
PROGS = myprog1 myprog2 myprog3 ...
```

- This means that we simply type make and the first target is processed, which in turn processes each of the targets **myprog1**, **myprog2**, **myprog3**, ...

touch command

`touch -c` ***`filename`***

- The UNIX command touch is used to update the modification timestamp of a file.
- It sets the time to the current system time
- If the file does not exist, a zero-byte file is created
  - If **`-c`** option is used, no file is created
- Very useful for testing makefiles and forcing complete recompiles