static
make

David Notkin ● Autumn 2009 ● CSE303 Lecture 21

static

```
// example.c
int passcode = 12345;                 // public
static int admin_passcode = 67890;    // private
```

- **`static`** used on global variables and functions
  - visible only to the current file/module (sort of like Java's **`private`**)
  - declare things **`static`** if you do not want them exposed
  - avoids potential conflicts with multiple modules that happen to declare global variables with the same names
  - **`passcode`** will be visible through the rest of **`example.c`**, but not to any other modules/files compiled with **`example.c`**

Function static data

- When used inside a function
  `static type name = value;`
  … declares a static local variable that will be remembered across calls

```
 int nextSquare() {
     static int n = 0;
     static int increment = 1;
     n += increment;
     increment += 2;
     return n;
 }
```

- **`nextSquare()`** returns `1`, then `4`, then `9`, then `16`, ...

The compilation process

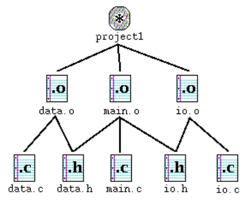
- What happens when you compile a Java program?
  `$ javac Example.java`
  - **`Example.java`** is compiled to create **`Example.class`**
- But...
  - what if you compile it again?
  - what if **`Example.java`** uses **`Point`** objects from **`Point.java`**?
  - what if **`Point.java`** is changed but not recompiled, and then we try to recompile **`Example.java`**?

Compiling large programs

- Compiling multi-file programs repeatedly is cumbersome
  `$ gcc -g -Wall -o myprogram file1.c file2.c file3.c`
- Retyping the above command is wasteful
  - for the developer (so much typing), and it's error-prone
  - for the compiler (may not need to recompile all; save them as .o)
- Improvements
  - use up-arrow or history to re-type compilation command for you
  - use an alias or shell script to recompile everything
  - use a system for compilation/build management, such as make

Dependencies

- Dependency : When a file relies on the contents of another – can be displayed as a dependency graph
  - to build **`main.o`**, we need **`data.h`**, **`main.c`**, and **`io.h`**
  - if any of those files is updated, we must rebuild **`main.o`**
  - if **`main.o`** is updated, we must update **`project1`** (which is probably an executable like **`a.out`**)



make

- **`make`** : a utility for automatically compiling ("building") executables and libraries from source code.
  - a very basic compilation manager
  - often used for C programs, but not language-specific
  - primitive, but still widely used due to familiarity, simplicity
  - similar programs: ant, maven, IDEs (Eclipse), ...
- **`makefile`** : A script file that defines rules for what must be compiled and how to compile it.
  - makefiles describe which files depend on which others, and how to create / compile / build / update each file in the system as needed.
  - The basic idea is to compare file modification dates and to rebuild any file A dependent on another file B that has changed more recently than A

Makefile rule syntax

```
target: source1 source2 ... sourceN
        command
        command
        ...
```

Example:

```
myprogram: file1.c file2.c file3.c
      gcc -o myprogram file1.c file2.c file3.c
```

- The command line must be indented by a single tab
  - not by spaces; NOT BY SPACES! SPACES WILL NOT WORK!

Running make

```
$ make target
```

- uses the file named **`Makefile`** in current directory
- finds rule in **`Makefile`** for building **`target`**
  - if the **`target`** file does not exist, or if it is older than any of its sources, its commands will be executed
- variations:
  - **`$ make`**
  - builds the first target in the **`Makefile`**
  - **`$ make -f makefilename`**
  - **`$ make -f makefilename target`**
  - uses a makefile other than **`Makefile`**

Rules with no sources

```
clean:
       rm file1.o file2.o file3.o myprog
```

- make assumes that a rule's command will build or create its target
  - but if your rule does not actually create its target, the target will still not exist the next time, so the rule will always execute (**`clean`** above)
  - **`make clean`** is a convention for removing all compiled files (but not source or header files!)

Rules with no commands

```
all: myprog myprog2

myprog: file1.o file2.o file3.o
      gcc -g -Wall -o myprog file1.o file2.o file3.o

myprog2: file4.c
      gcc -g -Wall -o myprog2 file4.c
...
```

- all rule has no commands, but depends on **`myprog`** and **`myprog2`**
  - **`make all`** ensures that **`myprog`**, **`myprog2`** are up to date
  - **`all`** rule often put first, so that typing **`make`** will build everything

Variables

```
NAME = value         (declare)
$(NAME)              (use)

OBJFILES = file1.o file2.o file3.o
PROGRAM = myprog

$(PROGRAM): $(OBJFILES)
        gcc -g -Wall -o $(PROGRAM) $(OBJFILES)

clean:
        rm $(OBJFILES) $(PROGRAM)
```

- variables make it easier to change one option throughout the file
  - also makes the makefile more reusable for another project

More variables

```
OBJFILES = file1.o file2.o file3.o
PROGRAM = myprog
ifdef WINDIR        # assume it's a Windows box
        PROGRAM = myprog.exe
endif
CC = gcc
CCFLAGS = -g -Wall

$(PROGRAM): $(OBJFILES)
        $(CC) $(CCFLAGS) -o $(PROGRAM) $(OBJFILES)
```

- variables can be conditional (ifdef above)
- many makefiles create variables for the compiler, flags, etc.
  - this can be overkill, but you will see it "out there"

Special variables include

`$@` the current target file
`$^` all sources listed for the current target
`$<` the first (left-most) source for the current target

```
myprog: file1.o file2.o file3.o
        gcc $(CCFLAGS) -o $@ $^

file1.o: file1.c file1.h file2.h
        gcc $(CCFLAGS) -c $<
```

Auto-conversions

- Rather than specifying individually how to convert every `.c` file into its corresponding `.o` file, you can set up an implicit target:

```
# conversion from .c to .o
.c.o:
        gcc $(CCFLAGS) -c $<
```

  - "To create filename.o from filename.c, run gcc -g -Wall -c filename.c"
- For making an executable (no extension), simply write .c :

```
.c:
        gcc $(CCFLAGS) -o $@ $<
```

- Related rule: .SUFFIXES (what extensions can be used)

Dependency generation

- You can make gcc figure out dependencies for you:

```
$ gcc -M filename
```

  - instead of compiling, outputs a list of dependencies for the given file

```
$ gcc -MM filename
```

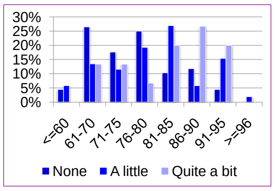
  - similar to -M, but omits any internal system libraries (preferred)
- Example:

```
$ gcc -MM linkedlist.c
linkedlist.o: linkedlist.c linkedlist.h util.h
```

- related command: makedepend

Midterm grades vs. initial experience

- Unix experience
  - 49% None
  - 37% A little
  - 11% Quite a bit
  - 3% Pays my tuition
- C/C++ experience
  - 52% None
  - 37% A little
  - 9% Quite a bit
  - 2% Pays my tuition



- This is a histogram of midterm scores
- The bars are the % of people who self-described as having “none”, “a little” or “quite a bit” of Unix/C/C++ experience
  - Each color should total 100%
  - “Pays my tuition” omitted because of the small numbers

	Mean/Median
None	76/75
A little	81/80
Quite a bit	85/83

Questions?