Today

- Structs
- Streams, Files
- C summary
- Make

Figuring out sizes: `sizeof()`

- sizeof() applied to an array returns the total size
- Be careful of implicit array/pointer conversions

```
#include <stdio.h>

int function(int x[]) {
    return (int) sizeof(x);
}

int main() {
   int a[20];
   printf("sizeof(int) = %d; sizeof(a) = %d\n",
sizeof(int), sizeof(a));
   printf("function returns %d\n", function(a));
}
```

what is passed to function() is a pointer, not the whole array

```
sizeof(int) = 4; sizeof(a) = 80
function returns 8
```


`sizeof` is an operator

- Not a function
  - a ha! That's why we couldn't replicate behavior of `sizeof` as a function
- When called from the same scope where an array was created, sizeof knows of array's size
- When you pass the array as a parameter, array's size information is lost.
  - Only a reference to the array position in memory is received by the function
  - Reference is treated as a pointer
  - `sizeof` returns the pointer's size

Figuring out sizes: `sizeof()`

- sizeof() applied to an array returns the total size
- Be careful of implicit array/pointer conversions

```
#include <stdio.h>

int function(int x[]) {
    return (int) sizeof(x);
}

int main() {
   int a[20];
   printf("sizeof(int) = %d; sizeof(a) = %d\n",
sizeof(int), sizeof(a));
   printf("function returns %d\n", function(a));
}
```

what is passed to function() is a pointer, not the whole array

```
sizeof(int) = 4; sizeof(a) = 80
function returns 8
```


STRUCTS

Structs

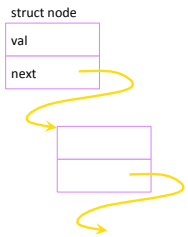
- A **struct** is
  - an *aggregate* data structure (i.e., a collection of data)
  - can contain components ("fields") of different types
    Whereas arrays contain elements of the same type
  - fields are accessed by name
    Whereas array elements are accessed by index position
- Unlike Java classes, a **struct** can only contain data, not code – like a class with only fields

Declaring structs

- A node for a linked list of integers:

```
struct node {
   int val;
   struct node *next;
}
```

optional "structure tag" refers to the structure



Accessing structure fields

- Given a struct **s** containing a field **f** , to access **f** we write
  s.f

Example:

```
struct foo {
    int count, bar[10];
} x, y;

x.count = y.bar[3];
```

declares x, y to be variables of type "struct foo"

- Given a *pointer* **p** to a struct **s** containing a field **f,** to access f we write
  p->f // eqvt. to: (***p**).**f**

Example:

```
struct foo {
    int count, bar[10];
} *p, *q;

p->count = q->bar[3];
```

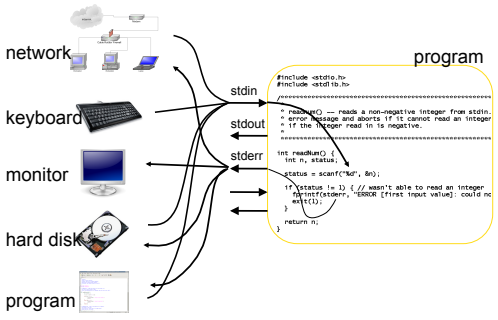

STREAMS

Input and output

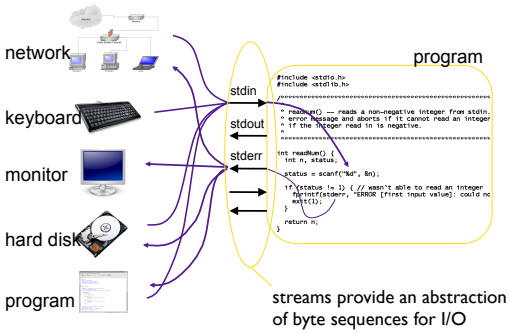
- Data is read from and written to I/O ***streams***
- 3 predefined streams:
  - **stdin**: "standard input" - usually, keyboard input
  - **stdout**: "standard output" - usually, the screen
  - **stderr**: "standard error" - for error messages (usually, the screen)

Other streams can be created using system calls (e.g., to read or write a specific file)

Streams



Streams



streams provide an abstraction of byte sequences for I/O

I/O Redirection

- Default input/output behavior for commands:
  - **stdin**: keyboard; **stdout**: screen; **stderr**: screen
- We can change this using I/O redirection:

cmd < *file*	redirect **cmd**'s stdin to read from *file*
cmd > *file*	redirect **cmd**'s stdout to *file*
cmd >> *file*	append **cmd**'s stdout to *file*
cmd >& *file*	redirect **cmd**'s stdout and stderr to *file*
cmd$_1$ | **cmd**$_2$	redirect **cmd**$_1$'s stdout to **cmd**$_2$'s stdin

Depends on the shell

Redirecting Output

- Save output from a program

```
> java OlympicScore > score.out
```

  - Redirected `stdout` to `score.out`
  - `stderr` would still go to terminal
- To redirect `stderr` to file as well

```
> java OlympicScore >& score.out
```


Review: Combining commands with pipes

- The output of one command can be fed to another command as input.
  - Syntax: *command*$_1$ | *command*$_2$

"pipe"

Example:

`ls`	lists the files in a directory
`more` *foo*	shows the file *foo* one screenful at a time
`ls | more`	lists the files in a directory one screenful at a time

How this works:
- ls writes its output to its **stdout**
- more's input stream defaults to its **stdin**
- the pipe connects ls's stdout to more's stdin
- the piped commands run "in parallel"

Pipeline Chaining

- Redirections & pipes can be combined for some nifty automated purposes:

```
./myscript < input1.txt | ./other.sh > out.txt
```

- Many handy UNIX commands can be piped together to quickly automate tasks:

```
> ls /usr/bin/ | grep "^wh" | sort -r
      whois
      whoami
      who
      which
      whereis
      whatis
      what
```

Easy way to automatically test your programs

(this example could be written as `ls -r /usr/bin/wh*`)

STREAMS IN C

File Streams in C

- A ***stream*** is any source of input or any destination for output
  - conceptually, just a sequence of bytes
  - accessed through a file pointer, type `FILE*`
  - not all streams are associated with files
    - 3 standard predefined streams: stdin, stdout, stderr

Typical structure of I/O operations

A program's I/O operations usually have the following structure:

1. Open a file — `fopen`
2. Perform I/O — `fprintf, fscanf, fread, fwrite, fgets`
3. Close the file — `fclose`

19

Opening a file

FILE* fopen(char * filename, char * mode)

name of file to open

file pointer for the stream, if fopen succeeds; NULL otherwise

"r"	read
"w"	write (file need not exist)
"a"	append (file need not exist)
"r+"	read and write, starting at the beginning
"w+"	read and write; truncate file if it exists
"a+"	read and write; append if file exists

20

Closing a file

int fclose(FILE *fp)

file pointer for stream to be closed

return value:
0 if the file was closed successfully;
EOF otherwise

21

Example Code Structure

```
FILE *fp;
…
fp = fopen(filename, "r");

if (fp == NULL) {
   … give error message and exit …
}
… read and process file …
int status = fclose(fp);
if (status == EOF) {
   … give error message…
}
```

22

Reading and writing

- **fprintf**, **fscanf**
  - similar to printf and scanf, with additional **FILE*** argument
- **fread**(*ptr*, *sz*, *num*, *fp*)
  - reads *num* elements, each of size *sz*, from stream *fp* and stores them at *ptr*
  - does not distinguish between end-of-file and error
    - use **feof**() and **ferror**()
- **fwrite**(*ptr*, *sz*, *num*, *fp*)
  - writes *num* elements, of size *sz*, from *ptr* into stream *fp*
- return values:
  - no. of items successfully read/written (not no. of bytes)

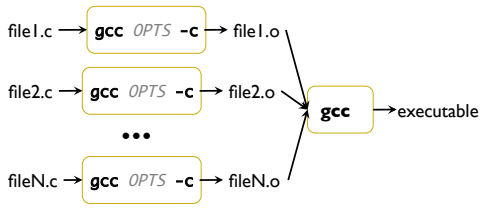
23

C, in Summary

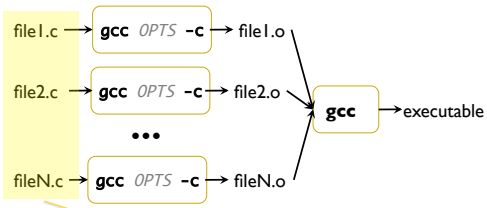
- **Compiled, statically typed**
- **Data types:** int, char, float, double (short, long, signed, unsigned)
  - What's missing?
- Pointer-related operations: *, &
  - Can do arithmetic on pointers
- Arrays are pointers
- Libraries, functions available

C PROGRAM ORGANIZATION & DEVELOPMENT USING MAKE

Compiling multi-file programs

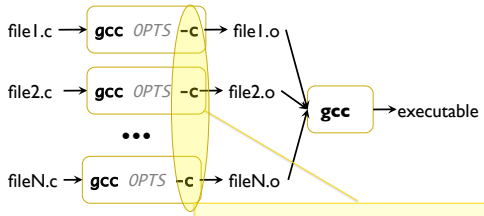


Compiling multi-file programs



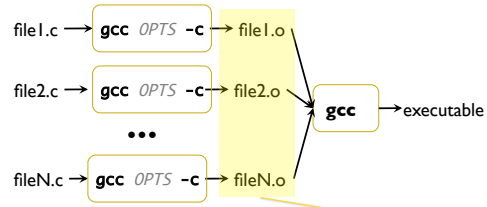
source files
Only one of these files contains main()

Compiling multi-file programs



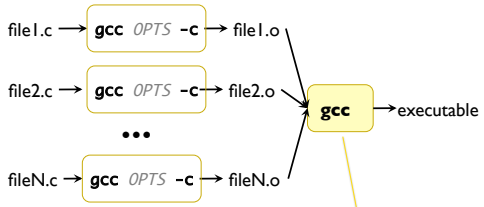
gcc -c
compile to a linkable object
& don't worry about main()

Compiling multi-file programs



object files
machine code, but not executable

Compiling multi-file programs



linker invocation
combines various ***.o** files together

Functions from special libraries

- Some library code is not linked in by default
  - Examples: **sqrt**, **ceil**, **sin**, **cos**, **tan**, **log**, … [math library]
  - requires specifying to the compiler/linker that the math library needs to be linked in
    - you do this by adding "**-lm**" at the end of the compiler invocation:

      gcc -Wall foo.c -lm

      linker command to add math library
- Libraries that need to be linked in explicitly like this are indicated in the man pages

Structuring large applications

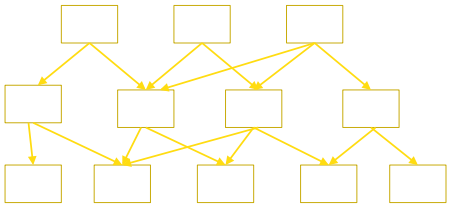
- So far, all of our programs have involved a single source file
  - impractical for large(r) programs
  - even where practical, may not be good from a design perspective
- If an application is broken up into multiple files, we need to manage the build process:
  - how do we (re)compile the various different files that make up the application?

Structuring large applications

- When one file is edited, other files may need to be recompiled
  - changes to typedefs or macros in header files
  - changes to types of shared variables
- Applications can contain a lot of files
  - E.g.: Linux kernel source code: ~ 4,900 files
- Recompiling all files whenever any file is changed can be very time-consuming.

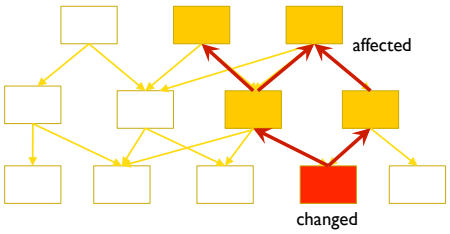
Structuring large applications

- Idea: only recompile those files that need to be recompiled – but which are those?



Structuring large applications

- Idea: only recompile those files that may be affected by a change.

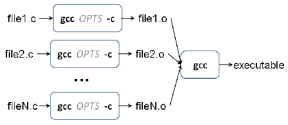


Structuring large applications

- "Smart recompilation" : issues
  - need to be able to express & keep track of dependencies between files
  - "dependency" ≈ which files affected by a change to another?
  - need to recompile all (and only) affected files
    - doing this manually is tedious and error-prone
    - want an automated solution
- make: a tool to automatically recompile based on user-specified dependencies ("make file")

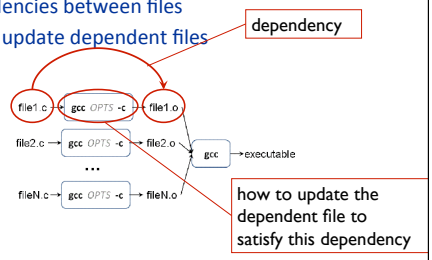
make files

- make files specify:
  - dependencies between files
  - how to update dependent files



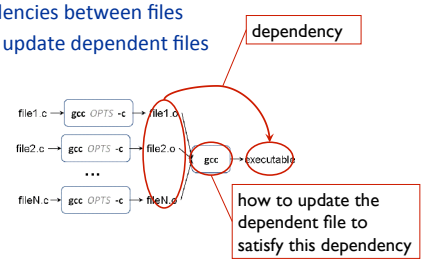
make files

- make files specify:
  - dependencies between files
  - how to update dependent files



make files

- make files specify:
  - dependencies between files
  - how to update dependent files



make files: structure

Structure of a make file:

Definitions (optional)

rule: *target* ... *prerequisites* ...

\t (tab) *command*

\t (tab) *command*

. . .

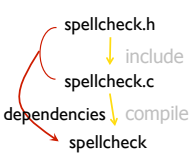
target: (usually) the name of a file that is created by a program

prerequisite: a file used as input to create the target

command: an action carried out by **make** to (re)construct *target*

make files: an elementary example

Dependency structure:



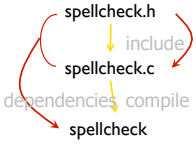
make file:

```
spellcheck: spellcheck.c  spellcheck.h
	gcc –Wall spellcheck.c
```

must be a tab!

make files: an elementary example

Dependency structure:



why is this not a dependency?

make files: another example

```
file1.o : file1.c  hdrfile1.h
        gcc  -Wall  -g  -c  file1.c
file2.o : file2.c  hdrfile1.h  hdrfile2.h
        gcc  -Wall  -g  -c  file2.c
execFile : file1.o  file2.o
        gcc  file1.o  file2.o  -o  execFile
```

Notice any similarities between the rules?

make files: Definitions

- Makes make files easier to write, modify
  - define: Name = replacement list
  - use: $(Name)
- Example:

```
CC = gcc
OPTLEV = –O2    # optimization level
CFLAGS = –Wall –g –D DEBUG $(OPTLEV) – c
. . .
file1.o  :  file1.c  hdrfile1.h
        $(CC)   $(CFLAGS)  file1.c
```


make files: Automatic Variables

- Automatic Variables make it easy to write default rules
  - %: indicates pattern rule in file name
  - $@: target file name
  - $<: first dependency
- Example:

```
CC = gcc
CFLAGS = –Wall –g –D DEBUG

%.o  :  %.c
        $(CC) -c $(CFLAGS) $< -o
$@
```


Using make

Invocation:

make [-f makeFileName] [target]

default:
make searches (in order) for:
makefile
Makefile

default:
builds the first target in the make file

How make works

- When invoked, begins processing the appropriate target
- For each target, considers the prerequisites it depends on:

  target : $file_1$ $file_2$ …
  - checks (recursively) whether each of $file_i$ (1) exists and (2) is more recent than the files that $file_i$ depends on;
    - if not, executes the associated command(s) to update $file_i$
  - checks whether *target* exists and is more recent that $file_i$
    - if not, executes the commands associated with *target*

Phony Targets

- A phony target is not the name of a file:

```
clean:
        rm  –f  *.o  a.out
```

phony target

- "make clean" will remove a.out and *.o files
- Can put any bash commands here

More on Make

- make has a lot of functionality, e.g.:
  - implicit rules
  - implicit variables
  - conditional parts of make files
  - recursively running make in subdirectories
- See online make tutorials for more information

TODO

- Assignment 0b
  - Word counter in C
  - 3 parts to the assignment
  - Due next Monday