Today

- Reviewing C programming
  - Pointers
    - command line arguments
  - Structs
  - Standard Streams
  - Dynamic Memory Allocation
- Make

Review

- Describe the C language
- How do we create executables?
- How do you represent strings in C?

My Goals for You and C

- Show you some good ways to do things
- Provide warnings about some easy ways to screw up

Review: C review – Data Types

```c
/*
 * A review of the basic data types in C.
 */

#include <stdio.h>

int main() {
        int x, y;
        char a;
        float f, e;
        double d;

        x = 4;
        y = 7;
        a = 'H';
        f = -3.4;
        d = 54.123456789;
        e = 54.123456789;

        printf("%d %c %f %lf\n", x, a, e, d);
        printf("%d %c %.9f %.3lf\n", x, a, e, d);
}
```

Output:

```
4 H 54.123455 54.123457
4 H 54.123455048 54.123
```

54.123456789 is too much precision for `float e` to handle

datatypes_and_print.c

Review: Strings, a.k.a. character arrays

- Example:

```
char a[6];
a[0] = 'H' ';
a[1] = 'i';
a[2] = '!';
a[3] = '\0';
```

Declared but not initialized

Initializing values

Label	Value
a[0]	'H'
a[1]	'i'
a[2]	'!'
a[3]	'\0'
a[4]	
a[5]	

Make null character explicit;
don't rely on the memory being 0'd out.
(Why isn't the memory necessarily zero'd out?)

- String processing methods will stop when the string delimiter, '\0', is reached

string.c

Char by Char String Processing

```
#include <stdio.h>
#include <string.h>

int main() {
    int i, j;
    char s[6];

    s[0] = 'a';
    s[1] = 'b';
    s[2] = 'a';
    s[3] = 'c';
    s[4] = '\0';
    printf("%s\n", s);
    i = 0;
    j = 0;
    while (s[i] != '\0') {
        if (s[i] != 'a') {
            s[j] = s[i];
            j++;
        }
        i++;
    }
    s[j] = '\0';
    printf("%s\n", s);
}
```

- What is the output of this program?
- What are more descriptive names for i and j?

charbychar.c

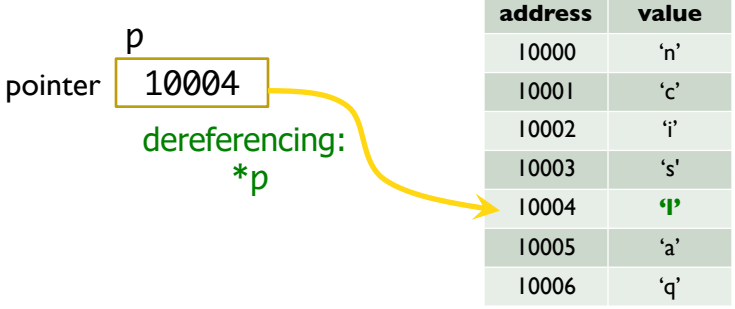
String Library Functions

- **`strlen`**: returns the number of characters in a string (before the delimiter)
- **`strcmp`**: returns whether two strings:
  - 0: are identical
  - -1: first string smaller
  - 1: second string smaller
- **`strcpy`**(dest, src): copies from src to dest
- **`strcat`**(orig, append): append a string to orig
- **`sprintf`**(a, "",?): store printf output to string a

POINTERS

Pointers

- A ***pointer*** in C holds a ***memory address***
  - the value of a pointer is an **address**
  - the value of the memory location pointed at can be obtained by "dereferencing the pointer" (retrieving the contents of that address)



address	value
10000	'n'
10001	'c'
10002	'i'
10003	's'
10004	**'l'**
10005	'a'
10006	'q'

Pointers

- Operators (unary, prefix):
  - **&** : "address of"
  - ***** : "dereference" or "value of"
- Example Declarations:

```
int *p;          // p: pointer to an int
char **w;        // w: pointer to a pointer to a char
```

- Spacing doesn't matter
  - I prefer to put the * next to the *type* during declarations, and next to the name when using as an operator

```
int* p;
int x = 5;
p = &x;
```

Using pointer-related operators

- If **x** is a variable, **&x** is the address of **x**
- If **p** is a pointer, ***p** is the value of whatever **p** points to
- ***(&p)** ≡ **p** always

Pointer Arithmetic

- Incrementing a pointer causes it to point to the next memory address, relative to the size of the type
  - for char* pointers, "+= 1" increments by 1
  - for int* pointers, "+= 1" increments by 4
    - if size of int is 4
- In general, "+= 1" will increment a pointer by the size in bytes of the type being pointed at
- Why? **Portability**:
  - We want to be able to step through an array of values without worrying about architecture-dependent issues like int size

address	value
10000	'w'
10001	'l'
10002	00
10003	00
10004	07
10005	E2
10006	00
10007	00
10008	07
10009	E3

(Representing ints 2018 and 2019 as 4B in hex)

Arrays are really Pointers

- To pass an array as a parameter, you pass the array name (i.e., a pointer)
- In C, arrays do not include size information
  - The called function doesn't know how big array is
- Solutions
  - pass the size of the array separately; or
  - terminate the array with a known value (e.g., 0)

Figuring out sizes: `sizeof()`

- `sizeof()` returns the total size of an array
  - (but not the # of non-null elements)
- Be careful of implicit array/pointer conversions

```
#include <stdio.h>

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

sizeof.c

Figuring out sizes: `sizeof()`

- `sizeof()` returns the total size of an array
  - (but not the # of non-null elements)
- Be careful of implicit array/pointer conversions

```
#include <stdio.h>

int function(int x[]) {
    return 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

Figuring out sizes: `sizeof()`

- `sizeof()` returns the total size of an array
  - (but not the # of non-null elements)
- Be careful of implicit array/pointer conversions

```
#include <stdio.h>

int function(int x[]) {
    return 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
-- size of address is 8 bytes

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

How do you read input into a C program?

scanf() and pointers

- **scanf**: the input equivalent to printf's output
- scanf takes 2 parameters:
  - a format string with conversion specifications (%d, %s, etc.) that says what kind of value is being read in;
  - a pointer to (i.e., the address of) a memory area where the value is to be placed
- Reading in an integer:

```
int x;
scanf("%d",  &x);    // &x = address of x
```

- Reading in a string:

```
char str[…];
scanf("%s", str);  // str = address of the
                   //    array str
```

Dereferencing & updating pointers

- A common C idiom is to use an expression that
  - gives the value of what a pointer is pointing at; **and**
  - updates the pointer to point to the next element:

    *p++

    Interpreted as: *p then p++
  - similarly: *p--

Walking a pointer through an array

```c
#include <stdio.h>

int main() {
    int iarray[100];
    int n, num, status, sum, i;
    int* iptr;

    iptr = iarray;
    n=0;

    while( n < 100 ) {
        status = scanf("%d", &num);
        if( status == 0 || num == 0 ) {
            break;
        }
        *iptr++ = num;
        n++;
    }

    for( iptr = iarray, sum=0; n > 0; n--) {
        sum += *iptr++;
    }

    printf("sum = %d\n", sum);
}
```

array_walk.c

Walking a pointer through an array

```c
#include <stdio.h>

int main() {
    int iarray[100];
    int n, num, status, sum, i;
    int* iptr;

    iptr = iarray;
    n=0;

    while( n < 100 ) {
        status = scanf("%d", &num);
        if( status == 0 || num == 0 ) {
            break;
        }
        *iptr++ = num;
        n++;
    }

    for( iptr = iarray, sum=0; n > 0; n--) {
        sum += *iptr++;
    }

    printf("sum = %d\n", sum);
}
```

Process the integer inputs and put them in the `iarray`

dereference the pointer to access memory, then increment the pointer

Add up those values in `iarray`

Command line arguments

```c
/* Print out the command line arguments
 * - they are an array of strings
 */

#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]) {
    int i, j;

    for (i = 0; i < argc; i++) {
        j = 0;
        while (argv[i][j] != '\0') {
            printf("%c", argv[i][j]);
            j++;
        }
        printf("\n");
    }

    for (i = 0; i < argc; i++)
        printf("%s\n", argv[i]);
}
```

Two common pointer problems

- Uninitialized pointers
  - the pointer is not initialized to point to a valid location
- Dangling pointers
  - the pointer points to a memory location that has been deallocated
    - Out of scope → seg fault

Using Pointers: Passing by Reference

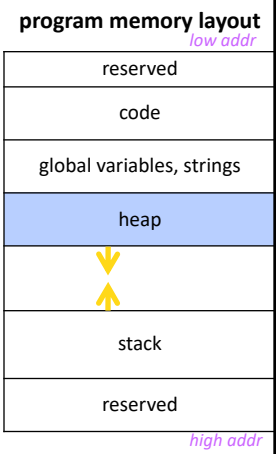
- What if you want to return multiple values from a function?
  - Java: encapsulate data in a class
  - C: encapsulate with a struct; ***OR***
  - Pass by reference (pointer)!
- Example:

```
int division(int numerator, int denominator,
                int* dividend, int* remainder) {
    if (denominator < 1)
        return 0;
    *dividend=numerator/denominator;
    *remainder=numerator%denominator;
}
```

```
int main() {
        int d,r;
        division(9,2,&d,&r);
}
```

Dynamic memory allocation

- We can't always anticipate how much memory to allocate at compile time
  - too little ⇒ program doesn't work
  - too much ⇒ wastes space
- Solution: allocate memory at runtime as necessary
  - malloc(), calloc()
    - allocates memory in the heap area
  - free()
    - deallocates previously allocated heap memory block



Dynamic memory allocation: usage

- `void *malloc(size_t size);`
- `void *calloc(size_t nmemb, size_t size);`
  - "clear" the memory

`void *` : "generic pointer"

- Example usage:
  - `int* iptr = malloc(sizeof(int));`
    - one int
  - `char* str = malloc(64);`
    - an array of 64 chars, sizeof(char) = 1
  - `int* iarr = calloc(40, sizeof(int));`
    - a 0-initialized array of 40 ints

Dynamic memory allocation example

```c
#include <stdio.h>
#include <stdlib.h>
void readVec(int size, int vec[]);
// computes the dot product of two integer vectors,
// each of size size
int dotprod(int *vec1, int *vec2, int size) {
    int i, dp;
    for(i=0, dp=0; i < size; i++ ) {
        dp += vec1[i] * vec2[i];
    }
    return dp;
}
int main() {
    int *vec1, *vec2, size;
    scanf("%d", &size);
    vec1 = malloc(size*sizeof(int));
    vec2 = malloc(size*sizeof(int));
    if( vec1 == NULL || vec2 == NULL ) { // error check
        fprintf(stderr, "Out of memory!\n");
        return 1;
    }
    readVec(size, vec1);    readVec(size, vec2);
    printf("dot product = %d\n", dotprod(vec1, vec2, size));
}
```

ALWAYS check the return value of any system call that may fail

STRUCTS

Structs

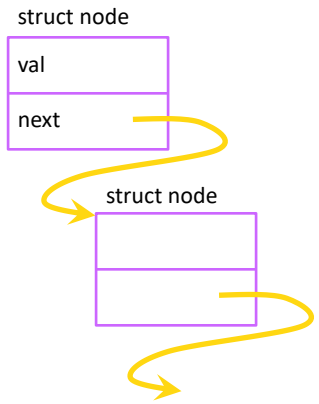
- A **struct** is
  - an *aggregate* data structure (i.e., a collection of data)
  - can contain components ("fields") of different types
    - Whereas arrays contain elements of the same type
  - fields are accessed by *name*
    - Whereas array elements are accessed by index position
- A **struct**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

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:

```
typedef struct {
    int count, bar[10];
} foo;
foo 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:

```
typedef struct {
  int count, bar[10];
} foo;
foo *p, *q;
p->count  =  q->bar[3];
```


STREAMS IN C

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

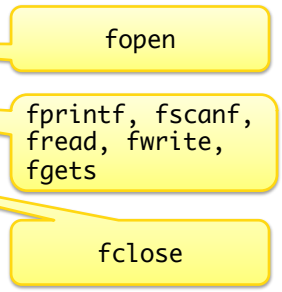
Standard Streams

- 3 predefined streams (`FILE*`) in C with stdio.h
  - **`stdin`**: "standard input" – usually, keyboard input
  - **`stdout`**: "standard output" – usually, the screen
  - **`stderr`**: "standard error" – for error messages (usually, the screen)
- These can be redirected to other sources

Typical structure of I/O operations

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

1. Open a file
2. Perform I/O
3. Close the file



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

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

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…
}
```

Buffering

- Sometimes program output isn't synchronized between multiple output streams (like stdout & stderr)
- Using a newline '\n' in printf helps, but does not always force
- Use **fflush**(stream) to force any remaining characters out of the buffer (even without a newline)

Reading and writing

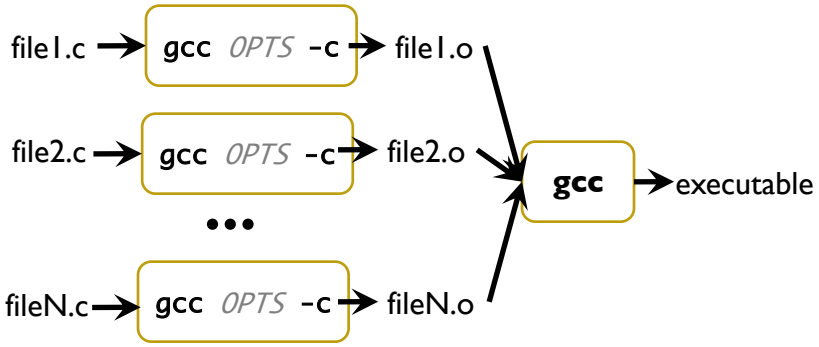
- **fprintf**, **fscanf**
  - similar to printf and scanf, with additional **FILE*** argument
- **fread**(*ptr*, *sz*, *num*, *fp*)
  - reads *num* elements, each of size *sz*, from stream *fp* and stores them at *ptr*
  - does not distinguish between end-of-file and error
    - use **feof**() and **ferror**()
- **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)

C, in Summary

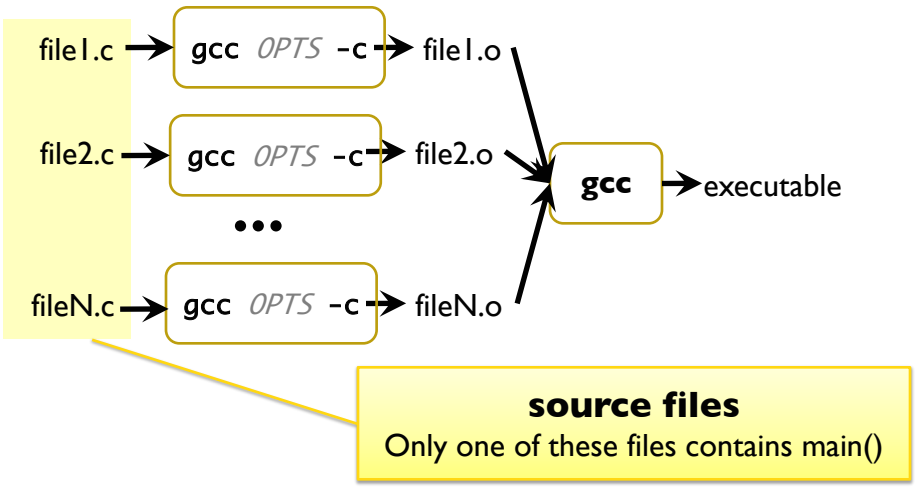
- **Compiled, statically typed**
- **Data types:** int, char, float, double (short, long, signed, unsigned)
  - What's missing?
- Pointer-related operations: *, &
  - Can do arithmetic on pointers
- Arrays are pointers
- Libraries, functions available

C PROGRAM ORGANIZATION & DEVELOPMENT USING MAKE

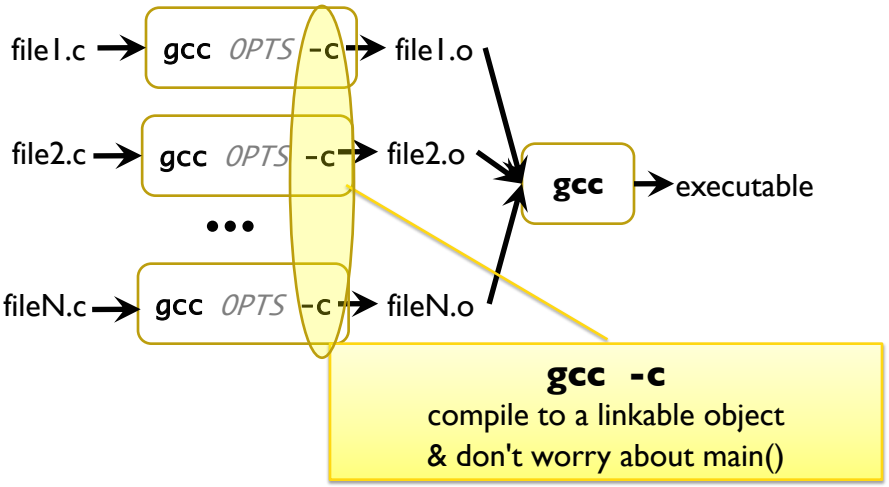
Compiling multi-file programs



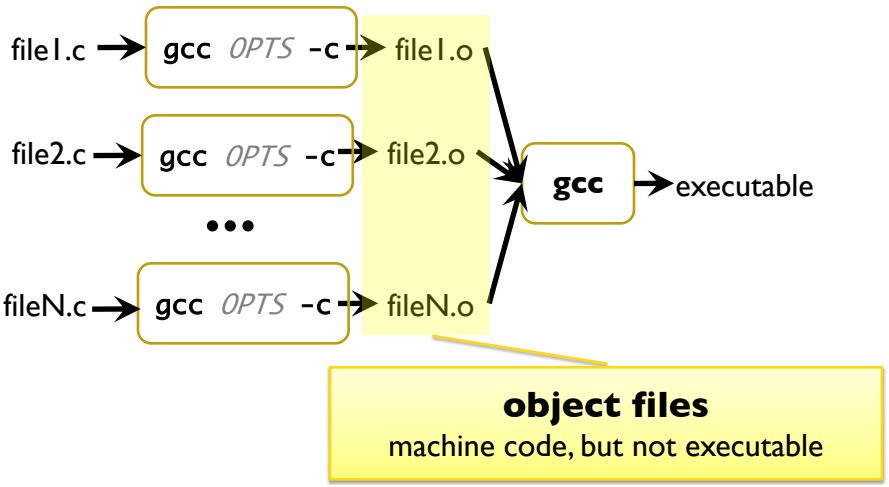
Compiling multi-file programs



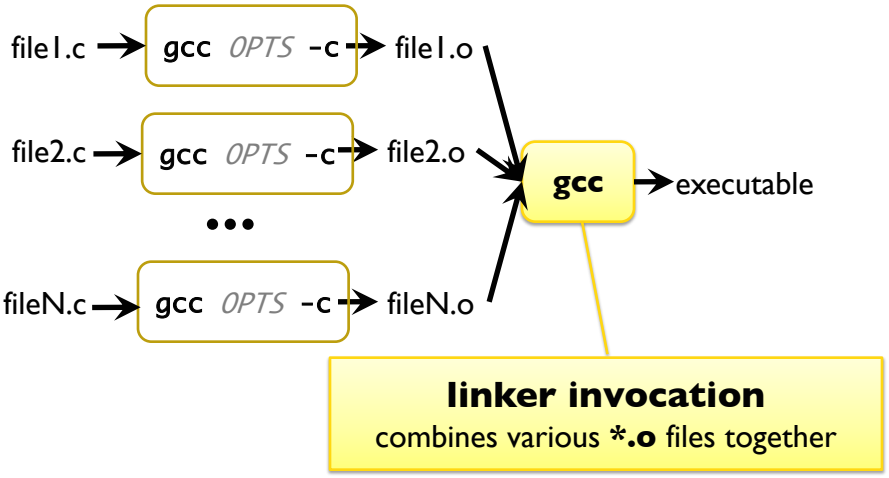
Compiling multi-file programs



Compiling multi-file programs



Compiling multi-file programs



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

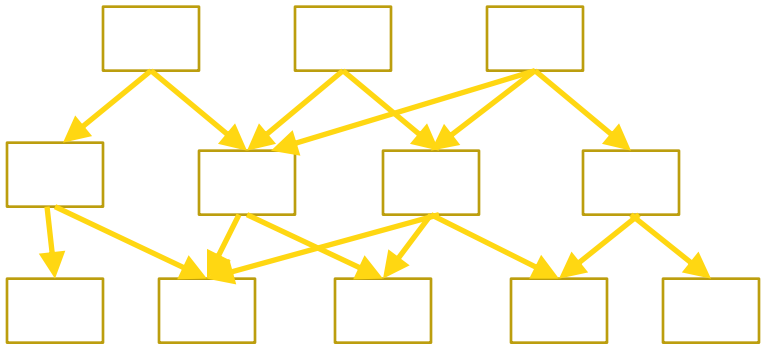
- So far, all of our programs have involved a single source file
  - impractical for large(r) programs
  - even where practical, may not be good from a design perspective
- If an application is broken up into multiple files, we need to manage the build process:
  - how do we (re)compile the various different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

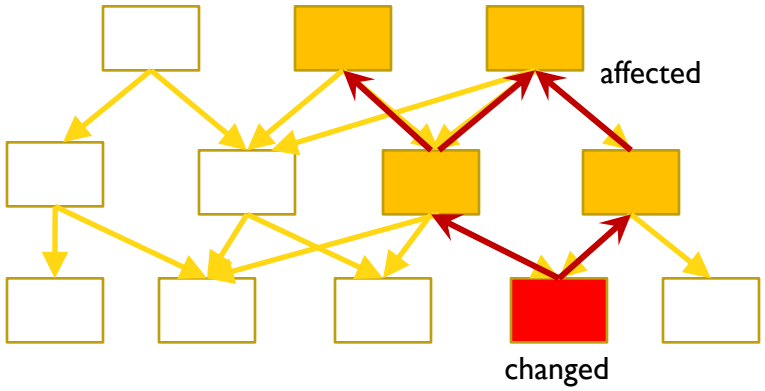
Structuring large applications

- 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.

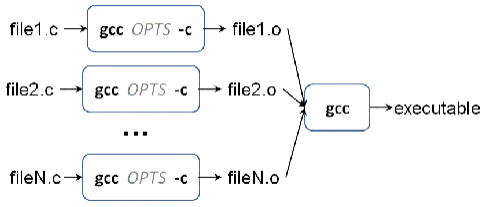


Structuring large applications

- "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

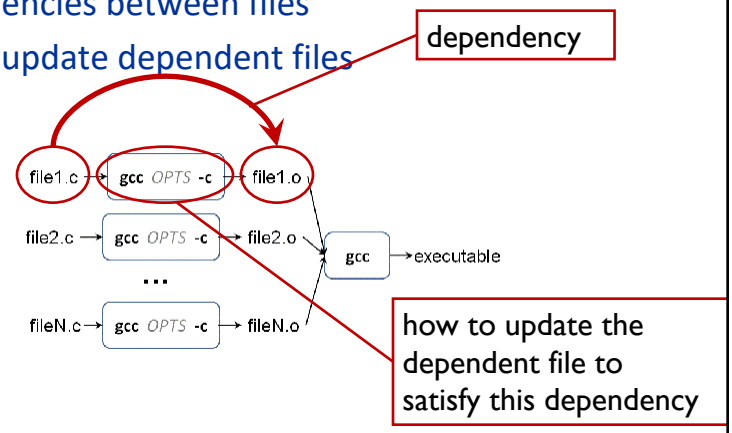
Makefiles

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



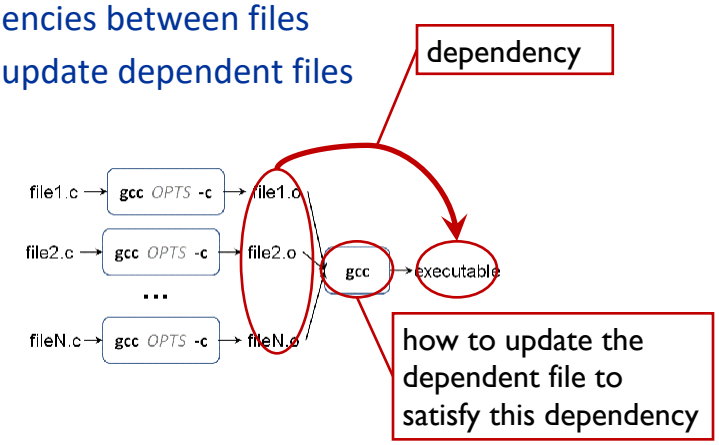
Makefiles

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



Makefiles

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



Makefiles: structure

Structure of a make file:

Definitions (optional)

rule

target ... : *prerequisites* ...

\t (tab) *command*

\t (tab) *command*

. . .

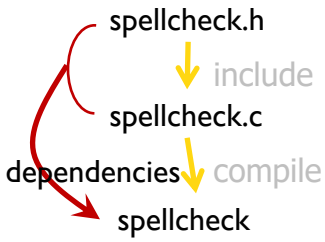
target: (usually) the name of a file that is created by a program

prerequisite: a file used as input to create the target

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

Makefiles: an elementary example

Dependency structure:



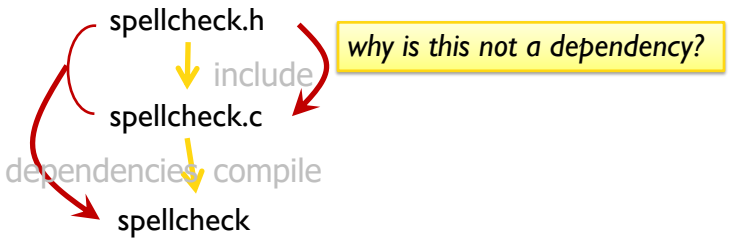
make file:

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

must be a tab!

Makefiles: dependencies

Dependency structure:



Makefiles: another example

```
execFile : file1.o  file2.o
     gcc  file1.o  file2.o  -o  execFile
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
```

Notice any similarities between the rules?

Makefiles: Definitions

- Definitions make make files easier to maintain
- 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
```


Makefiles: Automatic Variables

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

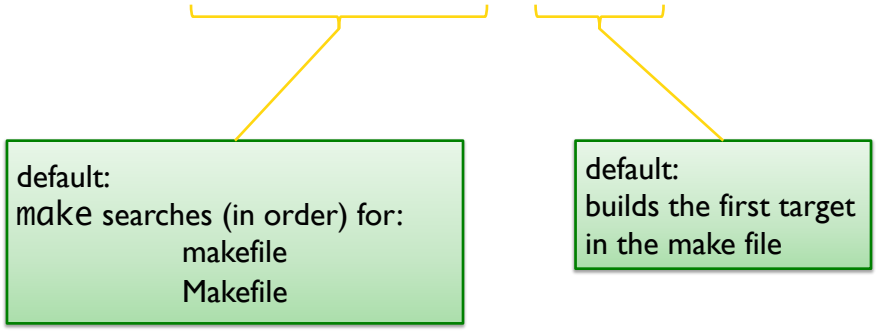
```
CC = gcc
CFLAGS = –Wall –g –D DEBUG

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

Using make

Invocation:

make [-f makeFileName] [target]



How make works

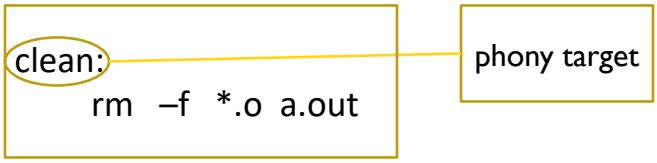
- When invoked, begins processing the appropriate target
- For each target, considers the prerequisites it depends on:

  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:



- `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

Looking Ahead

- C assignment due Wednesday
- Monday – more core OS discussion