CSE 374
Programming Concepts & Tools

Brandon Myers
Winter 2015
C: Linked list, casts, the rest of the preprocessor
(Thanks to Hal Perkins)

Linked lists, trees, and friends

- Very, very common data structures
- Building them in C
  - Use **`malloc`** to create nodes
  - Need to use casts for “generic” types
  - Memory management issues if shared nodes
  - Usually need to explicitly free entire thing when done
  - Shows tradeoffs between lists and arrays

- Look at the sample code and understand what it does/how it does it

Linked list

Generic data structures

Java:

```
class ListNode<V> {
		private V value;
		private ListNode<V> next;
}
```

No one best solution in C. Possibilities include

a) casts and void* pointer to data (style of malloc)

b) casts and fixed size data (i.e., not fully generic)

c) macros to substitute in types (generate type-specific code)

C++ uses (c) for generic programming but has a better/type-safe tool called templates…

Generic List, example

```
struct GenericListNode {
   void* data;
   GenericListNode* next;
}
```

```
struct List {
   int data_size;
   GenericListNode* head;
}
```

```
void insert_copy(List* li, void* value) {
    tail = …/* find tail */
    GenericListNode* newtail = (GenericListNode*)
                                                    malloc(sizeof(GenericListNode));
     newtail->data = malloc(li->data_size);
     memcpy(newtail->data, tail->data, li->data_size);
     newtail->next = NULL;
}
```

C types

- There are an infinite number of types in C, but only a few ways to make them:
  - char, int, double, etc. (many variations like unsigned int, long, short, …; mostly “implementation-defined”)
  - void (placeholder; a “type” no expression can have)
  - struct T where there is already a declaration for that struct type
  - Array types (basically only for stack arrays and struct fields, every use is automatically converted to a pointer type)
  - t* where t is a type
  - union T, enum E (later, maybe)
  - function-pointer types (later)
  - typedefs (just expand to their definition; type synonym)

Typedef

- Defines a synonym for a type – does ***not*** declare a new type
- Syntax

```
typedef  type  name;
```

After this declaration, writing `name` is the same as writing `type`

Caution: array typedef syntax is weirder

- Examples:

```
typedef int int32;            // use int32 for portability
typedef struct point {
   int32 x, y;
} Point2d;            // Point2d is synonym for struct Point
typedef Point2d * ptptr;      // pointer to Point2D

Point2d p;                    // var declaration
ptptr ptlist;                 // declares pointer
```

Casts, part 1

- Syntax: **`(t)e`** where t is a type and e is an expression (same as Java)
- Semantics: It depends
  - If e is a numeric type and t is a numeric type, this is a conversion
    - To wider type, get same value
    - To narrower type, may not (will get mod)
    - From floating-point to integral, will round (may overflow)
    - From integral to floating-point, may round (but int to double is exact on most machines)

    Note: Java is the same without the "most machines" part

    Note: Lots of implicit conversions such as in function calls

    Bottom Line: Conversions involve actual operations; **`(double)3`** is a very different bit pattern than **`(int)3`**

Casts, part 2

- If e has type t1*, then **`(t2*)e`** is a (pointer) cast.
  - You still have the ***same*** pointer (index into the address space).
  - Nothing "happens" at run-time.
  - You are just "getting around" the type system, making it easy to write any bits anywhere you want.
  - Old example: malloc has return type void*; we cast to required pointer type

```
void evil(int **p, int x) {
   int * q = (int*)p;
   *q = x;
}
void f(int **p) {
   evil(p,345);
   **p = 17;          // writes 17 to address 345 (HYCSBWK)
}
```

Note: The C standard is more picky than we suggest, but few people know that and little code obeys the official rules.

C pointer casts, continued

Questions worth answering:

- How does this compare to Java’s casts?
  - Unsafe, unchecked (no “type fields” in objects)
  - Otherwise more similar than it seems
- When should you use pointer casts in C?
  - For “generic” libraries (**`malloc`**, linked lists, operations on arbitrary (generic) pointers, etc.)
  - For “subtyping” (later)
- What about other casts?
  - Casts to/from **`struct`** types (*not* struct pointer casts) are compile-time errors.

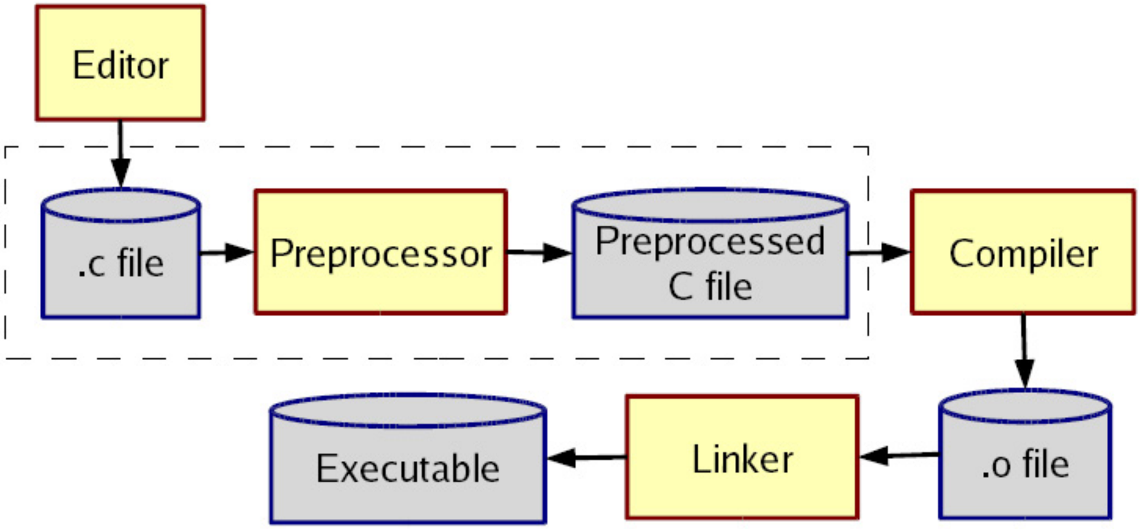
Course feedback

- We’re half way done
- Anonymous course feedback on website will be up later today. Optional. Take it by Sunday night
- How to improve the utility of lectures, homeworks, office hours
- Compliments and suggestions
- Complaints and suggestions

Preprocessor: The story so far…

- We've looked at the basics of the preprocessor
  - #include to access declarations in header files
  - #define for symbolic constants
- Now:
  - More details; where it fits
  - Multiple source and header files
  - A bit about macros (somewhat useful, somewhat a warning)

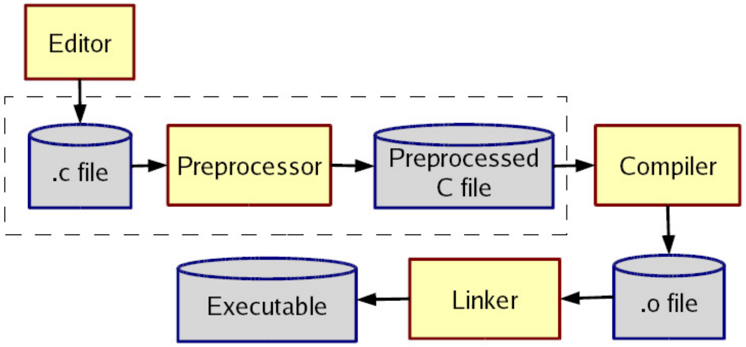
The compilation picture



gcc does all this for you (reminder)

- -E to only preprocess; result on stdout (rare)
- -c to stop with .o (common for individual files in larger program)

More about multiple files



Typical usage:

- Preprocessor #include to read file containing declarations describing code
- Linker handles your .o files *and* other code
  - By default, the "standard C library"
  - Other .o and .a files
  - Whole lecture on linking and libraries later…

The preprocessor

- Rewrites your .c file before the compiler gets at the code.
  - Lines starting with # tell it what to do
- Can do crazy things (please don't); uncrazy things are:
  1. Including contents of header files
  2. Defining constants and parameterized macros
     - Token-based, but basically *textual replacement*
     - Easy to mis-define and misuse
  3. Conditional compilation
     - Include/exclude part of a file
     - Example uses: code for debugging, code for particular computers (handling portability issues), "the trick" for including header files only once

File inclusion (review)

#include <hdr.h>

- Search for file hdr.h in “standard include directories” and include its contents in this place
  - Typically lots of nested includes, result not fit for human consumption
  - Idea is simple: declaration of standard library routines are in headers; allows correct use after declaration

#include “hdr.h”

  - Same, but first look in current directory
  - How to break your program into smaller files that can call routines in other files

- gcc -I option: look first in specified directories for headers (keep paths out of your code files) (not needed for 374)

Header file conventions

Conventions: always follow these when writing a header file

1. Give included files names ending in .h; only include these header files. ***Never*** #include a .c source file
2. Do not put functions definitions in a header file; only struct definitions, prototypes (declarations), and other includes
3. Do all your #includes at the beginning of a file
4. For header file foo.h start it with:

```
#ifndef FOO_H
#define FOO_H
```

and end it with:

```
#endif
```

(We will learn why very soon)

Simple macros (review)

Symbolic constants and other text

```
#define NOT_PI  22/7
#define VERSION 3.14
#define FEET_PER_MILE  5280
#define MAX_LINE_SIZE 5000
```

- Replaces all matching *tokens* in rest of file
  - Knows where “words” start and end (unlike sed)
  - Has no notion of scope (unlike C compiler)
  - (Rare: can shadow with another #define or use #undef to remove)

Some predefined macros

- e.g., __LINE__: source file line, __FILE__ source file name

e.g., log message that has source code information

```
printf("%s:%d %s\n", __FILE__, __LINE__, x)
```

Macros with parameters

```
#define TWICE_AWFUL(x) x*2
#define TWICE_BAD(x) ((x)+(x))
#define TWICE_OK(x) ((x)*2)
double twice(double x) { return x+x; }   // best (editorial opinion)
```

- Replace all matching “calls” with “body” but with text of arguments where the parameters are (*just* string substitution)
- Gotchas (understand why!):

```
        y=3;  z=4;  w=TWICE_AWFUL(y+z);
        y=7;  z=TWICE_BAD(++y);  z=TWICE_BAD(y++);
```

- Common misperception: Macros avoid performance overhead of a function call (maybe true in 1975, not now)
- Macros can be more flexible though (TWICE_OK works on ints and doubles without conversions (which could round))

Justifiable uses

Parameterized macros are generally to be avoided (use functions), but there are things functions cannot do:

- generating code
  - use type names (or other code) as arguments

```
#define NEW_T(t,howmany)  ((t*)malloc((howmany)*sizeof(t))
```

  - create new identifiers and write generic definitions

```
#define SCHEMA(t1, t2) \
typedef struct schema_##t1_##t2 { \
      t1 field1;  \
      t2 field2;  \
} schema_##t1_##t2;
```

Conditional compilation

```
#ifdef FOO       (matching #endif later in file)
#ifndef FOO      (matching #endif later in file)
#if FOO > 2      (matching #endif later in file)
```

(You can also have a #else inbetween somewhere.)

Simple use:

```
#ifdef DEBUG // do following only when debugging
    printf(...);
#endif
```

Fancier: (and another use of parameterized macros)

```
#ifdef DEBUG // use DBG_PRINT for debug-printing
#define DBG_PRINT(x) printf("%s",x)
#else
#define DBG_PRINT(x) // replace with nothing
#endif
```

- gcc -D FOO makes FOO “defined”

Back to header files

- Now we know what this means:

```
#ifndef SOME_HEADER_H_
#define SOME_HEADER_H_
... rest of some_header.h ...
#endif // SOME_HEADER_H_
```

- Assuming nobody else defines SOME_HEADER_H_ (convention), the first #include "some_header.h" will do the define and include the rest of the file, but the second and later will skip everything
  - More efficient than copying the prototypes over and over again
  - In presence of circular includes, necessary to avoid “creating” an infinitely large result of preprocessing
- So we always do this
- nicer alternative is to put the following at the top of the header:

```
#pragma once
```

(not in the language standard but is supported by most C compilers)

C preprocessor summary

- A few easy to abuse features and a bunch of conventions (for overcoming C’s limitations).
  - #include (the way you say what other definitions you need; cycles are fine with “the trick”)
  - #define (parameterized macros have a few justifiable uses; token-based text replacement)
  - #if... (for showing the compiler less code)