Poll Everywhere

pollev.com/naomila

About where are you on Homework 1?

A. Haven't started yet
B. Working on Part A (LinkedList)
C. Working on Part B (HashTable)
D. Finished or about finished
E. grl i dont owe u an answer

CSE333 Systems Programming

Preprocessor Tricks, Linking, File I/O

Instructors:

Naomi Alterman

Teaching Assistants:

Ann Baturytski	Barbora Buzkova	Mendel Carroll
Camden Harris	Hannah Hempstead	Rishabh Jain
Irene Lau	Jonathan Nister	Advay Patil
Aviel Wood	Angela Wu	Jennifer Xu

Administriviaaa

- Exercise 3 due Wednesday - be careful about how you submit!!!
  - We've been fixing incorrect submissions so far (tag issues, folder issues, filename issues)
  - But we're going to stop now!!

- Homework 1 due Thursday
  - ✓ Don't wait – Part B is longer and harder than Part A
  - ✓ OHs get crowded – come prepared to describe (1) your incorrect behavior, (2) what you think the issue is, and (3) what you've tried

Lecture Outline

- **C Preprocessor In-Depth**
- Visibility of Symbols
- File I/O with the C standard library

`#include` and the C Preprocessor

- The C preprocessor (`cpp`) is a *sequential* and *stateful* search-and-replace text-processor that transforms your source code before the compiler runs
  - The input is a C file (text) and the output is still a C file (text)
  - It processes the directives it finds in your code (*#directive*)
    - *e.g.* `#include "ll.h"` is replaced by the post-processed content of `ll.h`
    - *e.g.* `#define PI 3.1415` defines a symbol and replaces later occurrences
    - Several others that we'll see soon…
  - Run automatically on your behalf by `gcc` during compilation

C Preprocessor Example (1/10)

Arrow points to *next* line to process

- We can manually run the preprocessor:
  - cpp is the preprocessor (can also use `gcc -E`)
  - "–P" option suppresses some extra debugging annotations

```
#define BAR 2 + FOO

typedef long long int verylong;
```

cpp_example.h

```
#define FOO 1

#include "cpp_example.h"

int main(int argc, char** argv) {
  int x = FOO;    // a comment
  int y = BAR;
  verylong z = FOO + BAR;
  return 0;
}
```

cpp_example.c

```
$ cpp -P cpp_example.c out.c
$ cat out.c
```

C Preprocessor Example (2/10)

- We can manually run the preprocessor:
  - cpp is the preprocessor (can also use `gcc -E`)
  - "–P" option suppresses some extra debugging annotations

Pre-processor state

FOO	1

```
#define BAR 2 + FOO

typedef long long int verylong;
```

cpp_example.h

```
#define FOO 1

#include "cpp_example.h"

int main(int argc, char** argv) {
  int x = FOO;    // a comment
  int y = BAR;
  verylong z = FOO + BAR;
  return 0;
}
```

cpp_example.c

```
$ cpp -P cpp_example.c out.c
$ cat out.c
```

C Preprocessor Example (3/10)

- We can manually run the preprocessor:
  - cpp is the preprocessor (can also use `gcc -E`)
  - "–P" option suppresses some extra debugging annotations

Pre-processor state

FOO	1

```
#define BAR 2 + FOO

typedef long long int verylong;
```

cpp_example.h

```
#define FOO 1

#include "cpp_example.h"

int main(int argc, char** argv) {
  int x = FOO;    // a comment
  int y = BAR;
  verylong z = FOO + BAR;
  return 0;
}
```

cpp_example.c

```
$ cpp -P cpp_example.c out.c
$ cat out.c
```

C Preprocessor Example (4/10)

- We can manually run the preprocessor:
  - cpp is the preprocessor (can also use `gcc -E`)
  - "–P" option suppresses some extra debugging annotations

Pre-processor state

FOO	1
BAR	2 + 1

```
#define BAR 2 + FOO

typedef long long int verylong;
```

cpp_example.h

```
#define FOO 1

#include "cpp_example.h"

int main(int argc, char** argv) {
  int x = FOO;    // a comment
  int y = BAR;
  verylong z = FOO + BAR;
  return 0;
}
```

cpp_example.c

```
$ cpp -P cpp_example.c out.c
$ cat out.c
```

C Preprocessor Example (5/10)

- We can manually run the preprocessor:
  - cpp is the preprocessor (can also use `gcc -E`)
  - "–P" option suppresses some extra debugging annotations

Pre-processor state

FOO	1
BAR	2 + 1

```
#define BAR 2 + FOO

typedef long long int verylong;
```

cpp_example.h

```
#define FOO 1

#include "cpp_example.h"

int main(int argc, char** argv) {
  int x = FOO;    // a comment
  int y = BAR;
  verylong z = FOO + BAR;
  return 0;
}
```

cpp_example.c

```
$ cpp -P cpp_example.c out.c
$ cat out.c

typedef long long int verylong;
```

C Preprocessor Example (6/10)

- We can manually run the preprocessor:
  - cpp is the preprocessor (can also use `gcc -E`)
  - "–P" option suppresses some extra debugging annotations

Pre-processor state

FOO	1
BAR	2 + 1

```
#define BAR 2 + FOO

typedef long long int verylong;
```

cpp_example.h

```
#define FOO 1

#include "cpp_example.h"

int main(int argc, char** argv) {
  int x = FOO;    // a comment
  int y = BAR;
  verylong z = FOO + BAR;
  return 0;
}
```

cpp_example.c

```
$ cpp -P cpp_example.c out.c
$ cat out.c

typedef long long int verylong;
int main(int argc, char** argv) {
```

C Preprocessor Example (7/10)

- We can manually run the preprocessor:
  - cpp is the preprocessor (can also use `gcc -E`)
  - "–P" option suppresses some extra debugging annotations

Pre-processor state

FOO	1
BAR	2 + 1

```
#define BAR 2 + FOO

typedef long long int verylong;
```

cpp_example.h

```
#define FOO 1

#include "cpp_example.h"

int main(int argc, char** argv) {
  int x = FOO;   // a comment
  int y = BAR;
  verylong z = FOO + BAR;
  return 0;
}
```

cpp_example.c

```
$ cpp -P cpp_example.c out.c
$ cat out.c

typedef long long int verylong;
int main(int argc, char** argv) {
  int x = 1;
```

C Preprocessor Example (8/10)

- We can manually run the preprocessor:
  - cpp is the preprocessor (can also use `gcc -E`)
  - "–P" option suppresses some extra debugging annotations

Pre-processor state

FOO	1
BAR	2 + 1

```
#define BAR 2 + FOO

typedef long long int verylong;
```

cpp_example.h

```
#define FOO 1

#include "cpp_example.h"

int main(int argc, char** argv) {
  int x = FOO;   // a comment
  int y = BAR;
  verylong z = FOO + BAR;
  return 0;
}
```

cpp_example.c

```
$ cpp -P cpp_example.c out.c
$ cat out.c

typedef long long int verylong;
int main(int argc, char** argv) {
  int x = 1;
  int y = 2 + 1;
```

C Preprocessor Example (9/10)

- We can manually run the preprocessor:
  - cpp is the preprocessor (can also use `gcc -E`)
  - "–P" option suppresses some extra debugging annotations

Pre-processor state

FOO	1
BAR	2 + 1

```
#define BAR 2 + FOO

typedef long long int verylong;
```

cpp_example.h

```
#define FOO 1

#include "cpp_example.h"

int main(int argc, char** argv) {
  int x = FOO;   // a comment
  int y = BAR;
  verylong z = FOO + BAR;
  return 0;
}
```

cpp_example.c

```
$ cpp -P cpp_example.c out.c
$ cat out.c

typedef long long int verylong;
int main(int argc, char** argv) {
  int x = 1;
  int y = 2 + 1;
  verylong z = 1 + 2 + 1;
```

C Preprocessor Example (10/10)

- We can manually run the preprocessor:
  - cpp is the preprocessor (can also use `gcc -E`)
  - "–P" option suppresses some extra debugging annotations

Pre-processor state

FOO	1
BAR	2 + 1

```
#define BAR 2 + FOO

typedef long long int verylong;
```

cpp_example.h

```
#define FOO 1

#include "cpp_example.h"

int main(int argc, char** argv) {
  int x = FOO;   // a comment
  int y = BAR;
  verylong z = FOO + BAR;
  return 0;
}
```

cpp_example.c

```
$ cpp -P cpp_example.c out.c
$ cat out.c

typedef long long int verylong;
int main(int argc, char** argv) {
  int x = 1;
  int y = 2 + 1;
  verylong z = 1 + 2 + 1;
  return 0;
}
```

But There's a Problem with `#include`

- What happens when we compile `foo.c`?

```
struct Pair {
  int a, b;
};
```

pair.h

```
#include "pair.h"

// a useful function
struct Pair* MakePair(int a, int b);
```

util.h

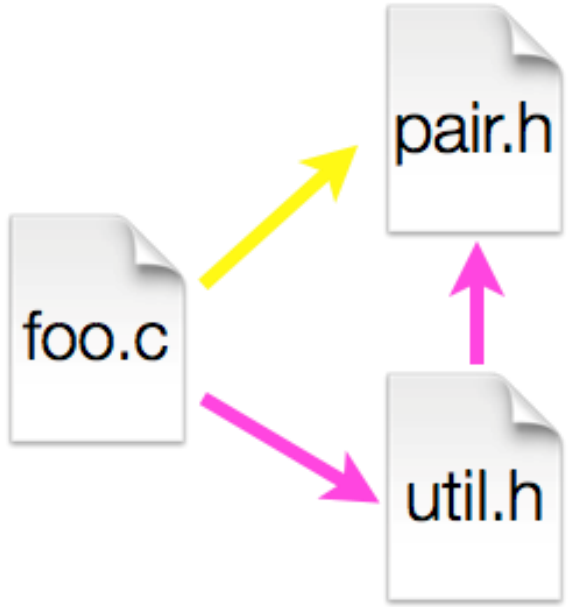
```
#include "pair.h"
#include "util.h"

int main(int argc, char** argv) {
  // do stuff here
  ...
  return 0;
}
```

foo.c

A Problem with `#include`

- What happens when we compile `foo.c`?

```
$ gcc -Wall -g -o foo foo.c
In file included from util.h:1,
                 from foo.c:2:
pair.h:1:8: error: redefinition of 'struct Pair'
    1 | struct Pair { int a, b; };
      |        ^~~~
In file included from foo.c:1:
pair.h:1:8: note: originally defined here
    1 | struct Pair { int a, b; };
      |        ^~~~
```



- `foo.c` includes `pair.h` twice!
  - Second time is indirectly via `util.h`
  - Struct definition shows up twice
    - Can see using `cpp`

Preprocessor Tricks: Conditional Compilation

- The preprocessor has conditional control flow just like C itself does
  - If there is no macro with the name stated in `#ifdef <macro name>`, the preprocessor will skip to the next `#else` or `#endif`.
  - Opposite is `#ifndef`
  - Nestable
- In this example, `#define TRACE` before `#ifdef` to include debug **`printf`**s in compiled code

```
#ifdef TRACE
#define ENTER(f) printf("Entering %s\n", f)
#define EXIT(f)  printf("Exiting  %s\n", f)
#else
#define ENTER(f)
#define EXIT(f)
#endif

// print n
void Pr(int n) {
  ENTER("Pr");
  printf("\n = %d\n", n);
  EXIT("Pr");
}
```

ifdef.c

Preprocessor Tricks: Header Guards

- A *classic* preprocessor idiom to prevent headers from getting copy-and-pasted multiple times into one C file

```
#ifndef PAIR_H_
#define PAIR_H_

struct Pair {
  int a, b;
};

#endif   // PAIR_H_
```

pair.h

```
#ifndef UTIL_H_
#define UTIL_H_

#include "pair.h"

struct Pair* MakePair(int a, int b);

#endif   // UTIL_H_
```

util.h

```
#include "pair.h"
#include "util.h"

int main(int argc, char** argv) {
  // do stuff here
```

foo.c

Preprocessor Tricks: Constants

- A way to deal with "magic constants"

```
int globalbuffer[1000];

void circalc(float rad,
             float* circumf,
             float* area) {
  *circumf = rad * 2.0 * 3.1415;
  *area = rad * 3.1415 * 3.1415;
}
```

Bad code
(littered with magic constants)

```
#define BUFSIZE 1000
#define PI 3.14159265359

int globalbuffer[BUFSIZE];

void circalc(float rad,
             float* circumf,
             float* area) {
  *circumf = rad * 2.0 * PI;
  *area = rad * PI * PI;
}
```

Better code

Preprocessor Tricks: Macros

- You can pass arguments to macros

```
#define ODD(x) ((x) % 2 != 0)

void Foo() {
  if ( ODD(5) )
    printf("5 is odd!\n");
}
```

cpp

```
void Foo() {
  if ( ((5) % 2 != 0) )
    printf("5 is odd!\n");
}
```

- Beware of operator precedence issues! Use parentheses!

```
#define ODD(x) ((x) % 2 != 0)
#define WEIRD(x) x % 2 != 0

ODD(5 + 1);

WEIRD(5 + 1);
```

cpp

```
((5 + 1) % 2 != 0);

5 + 1 % 2 != 0;
```

- Discouraged in favor of inline functions (Google style guide)

Macro Alternatives

- `const`: a type qualifier that indicates that the data is read only
  - Compile-time construct that will generate a compiler error or warning if violated
  - Much more heavily used in C++ and we'll return to the nuances here later on in the course (pointers are weird!)
  - Can replace constant macro with a `const` variable

- `inline`: keyword used in front of a function definition to suggest to the compiler to optimize the function call away
  - Mostly beyond the scope of this course
  - Can replace macro with arguments with (`static`) inline functions

Preprocessor Tricks: Defining Tokens

- Besides `#define`s in the code, preprocessor values can be set from the command line:

```
$ gcc -Wall -g -DTRACE -o ifdef ifdef.c
```

- assert can be controlled the same way – defining NDEBUG causes assert to expand to "empty"
  - It's a macro – see `assert.h`

```
$ gcc -Wall -g -DNDEBUG -o faster useassert.c
```

Poll Everywhere

pollev.com/naomila

What will happen when we try to compile and run?

```
$ gcc -Wall -DFOO -DBAR -o condcomp condcomp.c
$ ./condcomp
```

A. Output "333"

B. Output "334"

C. Compiler error about EVEN

D. Compiler error about BAZ

E. w...wut?

```
#ifdef FOO
#define EVEN(x) !(x%2)
#endif
#ifndef DBAR
#define BAZ 333
#endif

int main(int argc, char** argv) {
  int i = EVEN(42) + BAZ;
  printf("%d\n",i);
  return EXIT_SUCCESS;
}
```

Lecture Outline

- C Preprocessor In-Depth
- **Visibility of Symbols**
  - **`extern`, `static`**
- File I/O with the C standard library

Namespace Problem

- If we define a global variable named "counter" in one C file, is it visible in a different C file in the same program?

  - Yes, if you use *external linkage*
    - The name "counter" refers to the same variable in both files
    - The variable is *defined* in one file and *declared* in the other(s)
    - When the program is linked, the symbol resolves to one location

  - No, if you use *internal linkage*
    - The name "counter" refers to a different variable in each file
    - The variable must be *defined* in each file
    - When the program is linked, the symbols resolve to two locations

External Linkage

- `extern` makes a *declaration* of something externally-visible
  - Works slightly differently for variables and functions…

```c
#include <stdio.h>
#include <stdlib.h>

// A global variable, defined and
// initialized here in foo.c.
// It has external linkage by
// default.
int counter = 1;

int main(int argc, char** argv) {
  printf("%d\n", counter);
  Bar();
  printf("%d\n", counter);
  return EXIT_SUCCESS;
}
```

foo.c

```c
#include <stdio.h>

// "counter" is defined and
// initialized in foo.c.
// Here, we declare it, and
// specify external linkage
// by using the extern specifier.
extern int counter;

void Bar() {
  counter++;
  printf("(Bar): counter = %d\n",
         counter);
}
```

bar.c

Internal Linkage

- `static` (in the global context) restricts a definition to visibility within that file

```c
#include <stdio.h>
#include <stdlib.h>

// A global variable, defined and
// initialized here in foo.c.
// We force internal linkage by
// using the static specifier.
static int counter = 1;

int main(int argc, char** argv) {
  printf("%d\n", counter);
  Bar();
  printf("%d\n", counter);
  return EXIT_SUCCESS;
}
```

foo.c

```c
#include <stdio.h>

// A global variable, defined and
// initialized here in bar.c.
// We force internal linkage by
// using the static specifier.
static int counter = 100;

void Bar() {
  counter++;
  printf("(Bar): counter = %d\n",
         counter);
}
```

bar.c

Function Visibility

bar.c

```
// By using the static specifier, we are indicating
// that Foo() should have internal linkage. Other
// .c files cannot see or invoke Foo().
static int Foo(int x) {
  return x*3 + 1;
}

// Bar is "extern" by default. Thus, other .c files
// could declare our Bar() and invoke it.
int Bar(int x) {
  return 2*foo(x);
}
```

main.c

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

extern int Bar(int x); // "extern" is default, usually omit

int main(int argc, char** argv) {
  printf("%d\n", Bar(5));
  return EXIT_SUCCESS;
}
```

Linkage Issues

- Every global (variables and functions) is `extern` by default
  - Unless you add the `static` specifier, if some other module uses the same name, you'll end up with a collision!
    - Best case: compiler (or linker) error
    - Worst case: stomp all over each other

- It's good practice to:
  - Use `static` to "defend" your globals
    - Hide your private stuff!
  - Place external declarations in a module's header file
    - Header is the public specification

Static Confusion…

- C has *another, unrelated* use for the word "`static`": to create a persistent *local* variable
  - The storage for that variable is allocated when the program loads, in either the `.data` or `.bss` segment
  - Retains its value across multiple function invocations

static_extent.c

```
void Foo() {
  static int count = 1;
  printf("Foo has been called %d times\n", count++);
}

void Bar() {
  int count = 1;
  printf("Bar has been called %d times\n", count++);
}

int main(int argc, char** argv) {
  Foo(); Foo(); Bar(); Bar(); return EXIT_SUCCESS;
}
```

Additional C Topics

- Teach yourself!
  - **man pages** are your friend!
  - String library functions in the C standard library
    - `#include <string.h>`
      - strlen(), strcpy(), strdup(), strcat(), strcmp(), strchr(), strstr(), …
    - `#include <stdlib.h>` or `#include <stdio.h>`
      - atoi(), atof(), sprint(), sscanf()
  - How to declare, define, and use a function that accepts a variable-number of arguments (`varargs`)
  - `unions` and what they are good for
  - `enums` and what they are good for
  - Pre- and post-increment/decrement
  - Harder: the meaning of the "`volatile`" storage class

Lecture Outline

- C Preprocessor In-Depth
- Visibility of Symbols
- **File I/O with the C standard library**

This is essential material for the next part of the project (HW2)!

File I/O

- We'll start by using C's standard library
  - These functions are part of `glibc` on Linux
  - They are implemented using Linux system calls (POSIX)

- C's `stdio` defines the notion of a stream
  - A sequence of characters that flows **to** and **from** a device
    - Can be either *text* or *binary*; Linux does not distinguish
  - Is *buffered* by default; `libc` reads ahead of your program
  - Three streams provided by default: `stdin`, `stdout`, `stderr`
    - You can open additional streams to read and write to files
  - C streams are manipulated with a `FILE*` pointer, which is defined in `stdio.h`

C Stream Functions (1/2)

- Some stream functions (complete list in `stdio.h`):
  - `FILE* fopen(filename, mode);`
    - Opens a stream to the specified file in specified file access mode
  - `int fclose(stream);`
    - Closes the specified stream (and file)
  - `int fprintf(stream, format, ...);`
    - Writes a formatted C string
      - `printf(...);` is equivalent to `fprintf(stdout, ...);`
  - `int fscanf(stream, format, ...);`
    - Reads data and stores data matching the format string

C Stream Functions (2/2)

- Some stream functions (complete list in `stdio.h`):

  - `size_t fwrite(ptr, size, count, stream);`
    - Writes an array of *count* elements of *size* bytes from *ptr* to *stream*
  - `size_t fread(ptr, size, count, stream);`
    - Reads an array of *count* elements of *size* bytes from *stream* to *ptr*

C Stream Error Checking/Handling

- Some error functions (complete list in `stdio.h`):
  - `int ferror(stream);`
    - Checks if the error indicator associated with the specified stream is set
  - `int clearerr(stream);`
    - Resets error and EOF indicators for the specified stream
  - `void perror(message);`
    - Prints `message` followed by an error message related to `errno` to `stderr`

C Streams Example (1/3)

cp_example.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#define READBUFSIZE 128

int main(int argc, char** argv) {
  FILE* fin;
  FILE* fout;
  char readbuf[READBUFSIZE];
  size_t readlen;

  if (argc != 3) {
    fprintf(stderr, "usage: ./cp_example infile outfile\n");
    return EXIT_FAILURE;       // defined in stdlib.h
  }

  // Open the input file
  fin = fopen(argv[1], "rb");  // "rb" -> read, binary mode
  if (fin == NULL) {
    perror("fopen for read failed");
    return EXIT_FAILURE;
  }
  ...   // next slide's code
```

C Streams Example (2/3)

cp_example.c

```
int main(int argc, char** argv) {

  ...   // previous slide's code

  // Open the output file
  fout = fopen(argv[2], "wb");  // "wb" -> write, binary mode
  if (fout == NULL) {
    perror("fopen for write failed");
    fclose(fin);
    return EXIT_FAILURE;
  }

  // Read from the file, write to fout
  while ((readlen = fread(readbuf, 1, READBUFSIZE, fin)) > 0) {
    // Test to see if we encountered an error while reading
    if (ferror(fin)) {
      perror("fread failed");
      fclose(fin);
      fclose(fout);
      return EXIT_FAILURE;
    }
    ...   // next slide's code
}
```

C Streams Example (3/3)

cp_example.c

```
int main(int argc, char** argv) {

  ...   // two slides ago's code

  ...   // previous slide's code

    if (fwrite(readbuf, 1, readlen, fout) < readlen) {
      perror("fwrite failed");
      fclose(fin);
      fclose(fout);
      return EXIT_FAILURE;
    }
  }

  fclose(fin);
  fclose(fout);

  return EXIT_SUCCESS;
}
```

Extra Exercise #1

- Modify the linked list code from the previous lecture's extra exercise
  - Add static declarations to any internal functions you implemented in `linkedlist.h`
  - Add a header guard to the header file

Extra Exercise #2

- Write a program that:
  - Uses `argc`/`argv` to receive the name of a text file
  - Reads the contents of the file a line at a time
  - Parses each line, converting text into a `uint32_t`
  - Builds an array of the parsed `uint32_t`'s
  - Sorts the array
  - Prints the sorted array to `stdout`

```
bash$ cat in.txt
1213
3231
000005
52
bash$ ./extra1 in.txt
5
52
1213
3231
bash$
```

- Hint: use `man` to read about `getline`, `sscanf`, `realloc`, and `qsort`

Extra Exercise #3

- Write a program that:
  - Loops forever; in each loop:
    - Prompt the user to input a filename
    - Reads a filename from `stdin`
    - Opens and reads the file
    - Prints its contents to `stdout` in the format shown:

```
00000000 50 4b 03 04 14 00 00 00 00 00 9c 45 26 3c f1 d5
00000010 68 95 25 1b 00 00 25 1b 00 00 0d 00 00 00 43 53
00000020 45 6c 6f 67 6f 2d 31 2e 70 6e 67 89 50 4e 47 0d
00000030 0a 1a 0a 00 00 00 0d 49 48 44 52 00 00 00 91 00
00000040 00 00 91 08 06 00 00 00 c3 d8 5a 23 00 00 00 09
00000050 70 48 59 73 00 00 0b 13 00 00 0b 13 01 00 9a 9c
00000060 18 00 00 0a 4f 69 43 43 50 50 68 6f 74 6f 73 68
00000070 6f 70 20 49 43 43 20 70 72 6f 66 69 6c 65 00 00
00000080 78 da 9d 53 67 54 53 e9 16 3d f7 de f4 42 4b 88
00000090 80 94 4b 6f 52 15 08 20 52 42 8b 80 14 91 26 2a
000000a0 21 09 10 4a 88 21 a1 d9 15 51 c1 11 45 45 04 1b
... etc ...
```

- Hints:
  - Use `man` to read about `fgets`
  - Or, if you're more courageous, try `man 3 readline` to learn about `libreadline.a` and Google to learn how to link to it