Poll Everywhere

About how long did Exercise 2 take you?

- A. **[0, 2) hours**
- B. **[2, 4) hours**
- C. **[4, 6) hours**
- D. **[6, 8) hours**
- E. **8+ Hours**
- F. **I didn't submit / I prefer not to say**

C Preprocessor, Linking
CSE 333 Fall 2023

Instructor: Chris Thachuk

Teaching Assistants:

Ann Baturytski	Humza Lala
Yuquan Deng	Alan Li
Noa Ferman	Leanna Mi Nguyen
James Froelich	Chanh Truong
Hannah Jiang	Jennifer Xu
Yegor Kuznetsov	

Relevant Course Information

❖ Exercise 3 out today, due Wednesday (10/4)

▪ First modularized (multi-file) exercise – separate interface, implementation, and tests

▪ *Automated* testing relies on status codes

❖ Homework 1 due next Friday (10/13)

▪ Watch that `HashTable` doesn't violate the modularity of `LinkedList` (*i.e.*, respect the interfaces!)

▪ Watch for pointer to local (stack) variables

▪ ***Draw memory diagrams!***

▪ Use `gdb` and valgrind and fill out your bug journal as you go!

▪ Please leave "`STEP #`" markers for graders!

▪ Timely bonus: don't tag `hw1-final` until you are really ready

Lecture Outline

- ❖ **C Preprocessor**

- ❖ Visibility of Symbols
  - ▪ `extern`, `static`

`#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`

    `" "` — look in local directory
    `< >` — look in library directory

  • *e.g.,* `#define PI 3.1415` *defines* a symbol and replaces later occurrences

    macro text substitution

  • Several others that we'll see soon...

▪ Run automatically on your behalf by `gcc` during compilation

Poll Everywhere

pollev.com/cse333

Exploration: Which of the following text will remain in the preprocessor output?

```
#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

Keep in mind:
1. Pre-processor goes line by line
2. builds up "state" as it processes directives

A. **#define**

B. **BAR**

C. **FOO**

D. **verylong**

E. **// a comment**

C Preprocessor Example

❖ 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

❖ We can manually run the preprocessor:

Pre-processor state

FOO	1

- `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

Pre-processor state

FOO	1

❖ 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

❖ 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

❖ 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;
```

11

C Preprocessor Example

❖ 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

❖ 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;
```

13

C Preprocessor Example

❖ We can manually run the preprocessor:

Pre-processor state

FOO	1
BAR	2 + 1

  ▪ `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

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

14

C Preprocessor Example

❖ We can manually run the preprocessor:

Pre-processor state

FOO	1
BAR	2 + 1

- `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

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

❖ 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;
}
```

Program Using a Linked List

```
#include <stdlib.h>
...
#include "ll.h"

Node* Push(Node* head,
           void* element) {
  ... // implementation here
}
```
— — Node def needed here

ll.c

```
typedef struct node_st {
  void* element;
  struct node_st* next;
} Node;

Node* Push(Node* head,
           void* element);
```

ll.h

```
#include "ll.h"

int main(int argc, char** argv) {
  Node* list = NULL;
  char* hi = "hello";
  char* bye = "goodbye";

  list = Push(list, (void*)hi);
  list = Push(list, (void*)bye);

  ...

  return 0;
}
```
— — Push declaration needed here
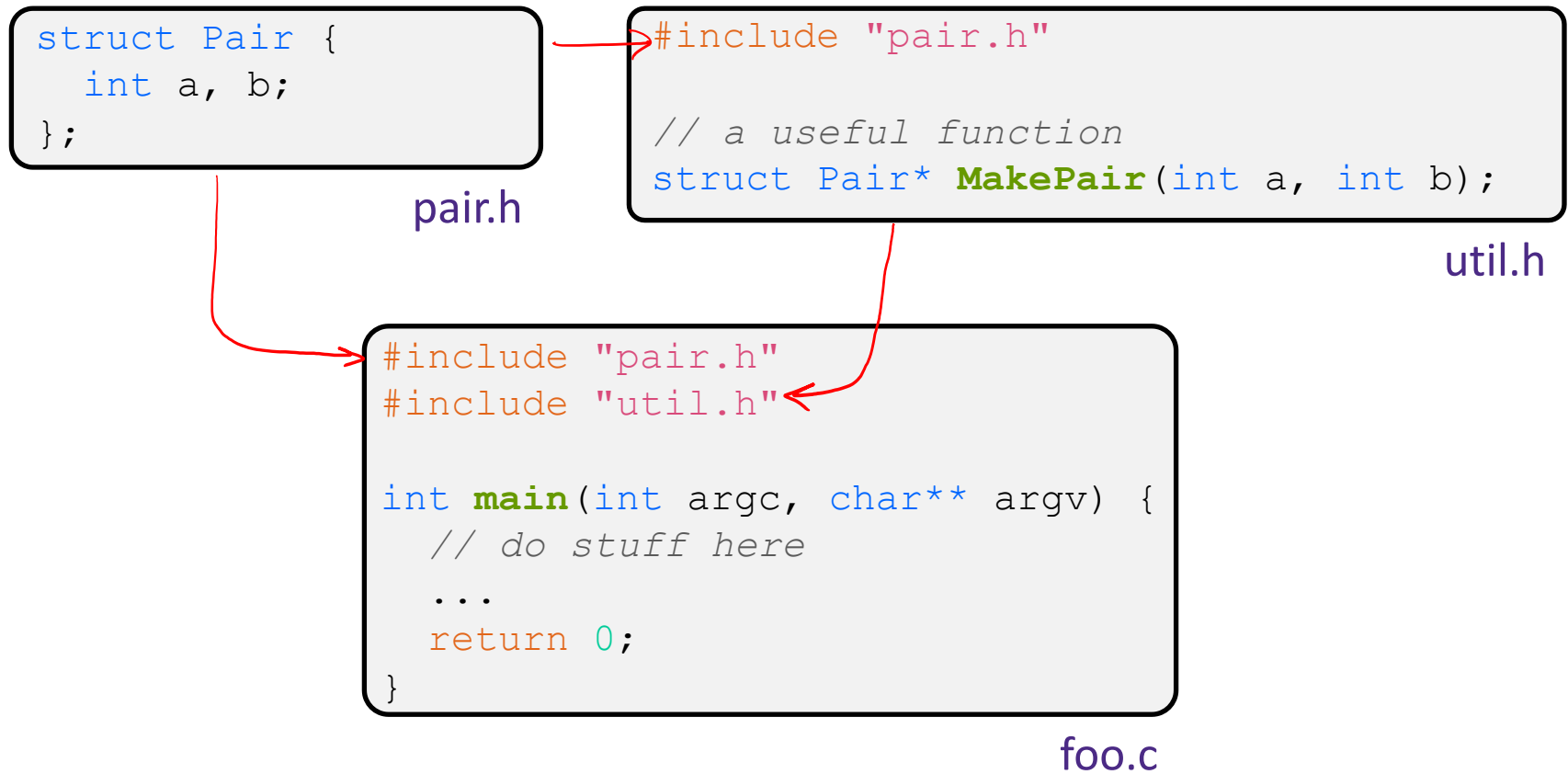
example_ll_customer.c

Compiling the Program

❖ Four parts:

- 1/2) Compile `example_ll_customer.c` into an object file

- 2/1) Compile `ll.c` into an object file

- 3) Link both object files into an executable

- 4) Test, Debug, Rinse, Repeat

```
$ gcc –Wall -g –c -o example_ll_customer.o example_ll_customer.c
$ gcc –Wall –g –c –o ll.o ll.c
$ gcc -g –o example_ll_customer ll.o example_ll_customer.o
$ ./example_ll_customer
Payload: 'yo!'
Payload: 'goodbye'
Payload: 'hello'
$ valgrind –leak-check=full ./example_ll_customer
... etc ...
```

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;
}
```
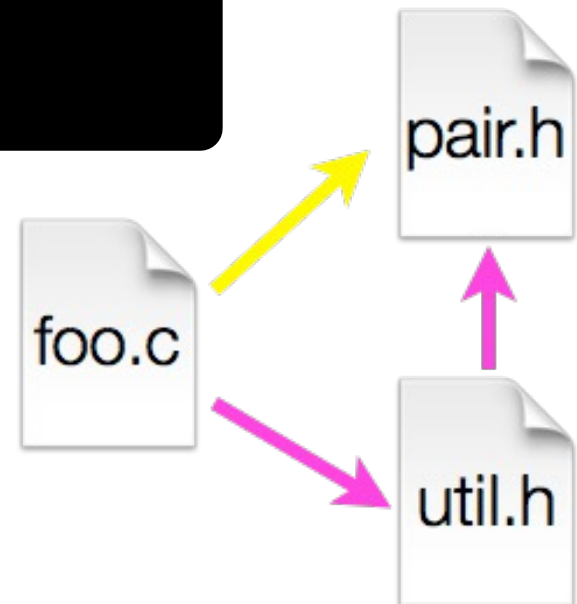foo.c

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: Header Guards

STYLE TIP

- ❖ A standard C Preprocessor trick to deal with this
    - Uses macro definition (#define) in combination with conditional compilation (#ifndef and #endif)

```
#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"

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

#endif   // UTIL_H_
```
util.h

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

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

Preprocessor Tricks: Constants

STYLE TIP

❖ A way to deal with "magic constants"

```c
int global_buffer[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)

```c
#define BUFSIZE 1000
#define PI 3.14159265359

int global_buffer[BUFSIZE];

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

Better code

Preprocessor Tricks: Macros

treated as just text

❖ 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 TIP

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 #defines in the code, preprocessor values can be given as part of the gcc command:

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

-D   define
-U   undefine

❖ assert can be controlled the same way – defining NDEBUG causes assert to expand to "empty"

▪ It's a macro – see assert.h

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

Preprocessor Tricks: Conditional Compilation

#ifdef = "if defined"
#ifndef = "if not defined"

❖ You can change what gets compiled

- 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

Lecture Outline

❖ C Preprocessor

❖ **Visibility of Symbols**

▪ **`extern, static`**

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

```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);
}
```
bar.c

bar() can invoke foo() because in same file

```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;
}
```
main.c

not explicitly needed, but indicates that definition is elsewhere

Linkage Issues

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

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

Static Confusion…

❖ C has a *different* 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 (Static Data)
- Retains its value across multiple function invocations

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

void Bar() {
  int count = 1;  // re-initialized each time
  printf("Bar has been called %d times\n", count++);
}

int main(int argc, char** argv) {
  Foo(); Foo(); Bar(); Bar(); return EXIT_SUCCESS;
}   1 times    2 times     1 times    1 times
```

static_extent.c

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

Extra Exercise #1

❖ Extend the linked list program we covered in class:

■ Add a function that returns the number of elements in a list

■ Implement a program that builds a list of lists

  • *i.e.* it builds a linked list where each element is a (different) linked list

■ Bonus: design and implement a "Pop" function

  • Removes an element from the head of the list

  • Make sure your linked list code, and customers' code that uses it, contains no memory leaks

Extra Exercise #2

❖ Modify the linked list code from Extra Exercise #1

- Add static declarations to any internal functions you implemented in `linkedlist.c`

- Add a header guard to the header file