

pollev.com/cse333a

About how long did Exercise 3 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

Systems Programming

Modules, C Preprocessor

Instructors:

Amber Hu

Justin Hsia

Teaching Assistants:

Ally Tribble

Blake Diaz

Connor Olson

Grace Zhou

Jackson Kent

Janani Raghavan

Jen Xu

Jessie Sun

Jonathan Nister

Mendel Carroll

Rose Maresh

Violet Monserate

Relevant Course Information

- ❖ Exercise 4 out today, due Friday before 11 AM
- ❖ Homework 1 due a week from Thursday
 - You should be well under way now
 - Be sure to read headers *carefully* while implementing
 - Use git add/commit/push regularly to save work – easier to share with partner and course staff
- ❖ Section this week will involve debugging!

- ❖ ***Correction from Lecture 4***

```
int a[2];  
a[2] = 0; //undefined
```

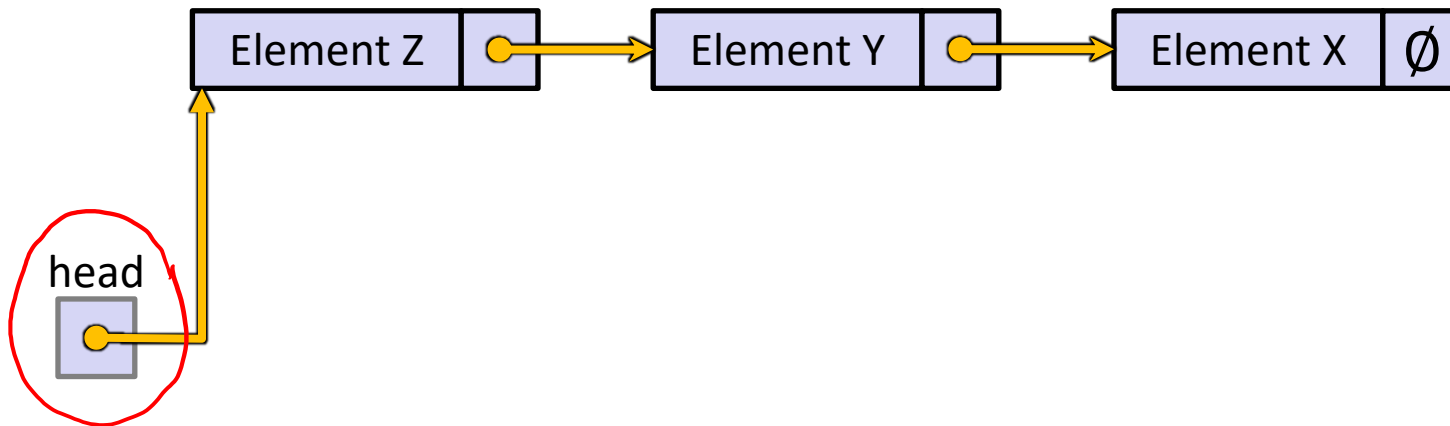
- Accessing one past the end of an array is undefined behavior

Lecture Outline (1/3)

- ❖ **Generic Data Structures in C**
- ❖ Modules & Interfaces
- ❖ C Preprocessor Intro

Simple Linked List in C

- ❖ Each node in a linear, singly-linked list contains:
 - Some element as its payload
 - A pointer to the next node in the linked list
 - This pointer is **NULL** (or some other indicator) in the last node in the list

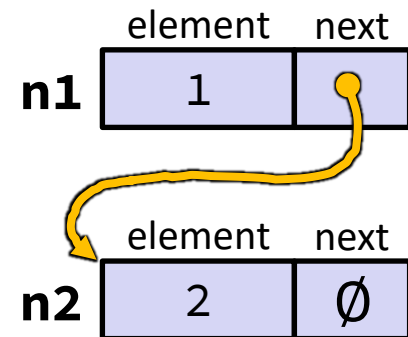


Linked List Node

- ❖ Let's represent a linked list node with a struct
 - For now, assume each element is an `int`

```
typedef struct node_st {  
    int element;  
    struct node_st* next;  
} Node;  
  
int main(int argc, char** argv) {  
    Node n1, n2; ← on stack  
  
    n1.element = 1;  
    n1.next = &n2;  
    n2.element = 2;  
    n2.next = NULL;  
    return EXIT_SUCCESS;  
}
```

tagname is necessary because
pointer to it is part
of struct definition



manual_list.c

Push Onto List (1/14)

Arrow points to
next instruction.

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

Node* Push(Node* head, int e) {
    Node* n = (Node*) malloc(sizeof(Node));
    assert(n != NULL); // crashes if false
    n->element = e;
    n->next = head;
    return n;
}

int main(int argc, char** argv) {
    Node* list = NULL;
    list = Push(list, 1);
    list = Push(list, 2);
    return EXIT_SUCCESS;
}
```

(main) list ∅

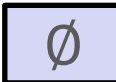
push_list.c

Push Onto List (2/14)

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

Node* Push(Node* head, int e) {
    Node* n = (Node*) malloc(sizeof(Node));
    assert(n != NULL); // crashes if false
    n->element = e;
    n->next = head;
    return n;
}

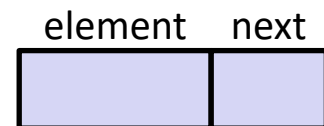
int main(int argc, char** argv) {
    Node* list = NULL;
    list = Push(list, 1);
    list = Push(list, 2);
    return EXIT_SUCCESS;
}
```

(main) list 

(Push) head 

(Push) e 

(Push) n 



push_list.c

Push Onto List (3/14)

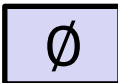
```
typedef struct node_st {
    int element;
    struct node_st* next;
} Node;

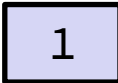
Node* Push(Node* head, int e) {
    Node* n = (Node*) malloc(sizeof(Node));
    assert(n != NULL); // crashes if false
    n->element = e;
    n->next = head;
    return n;
}

int main(int argc, char** argv) {
    Node* list = NULL;
    list = Push(list, 1);
    list = Push(list, 2);
    return EXIT_SUCCESS;
}
```

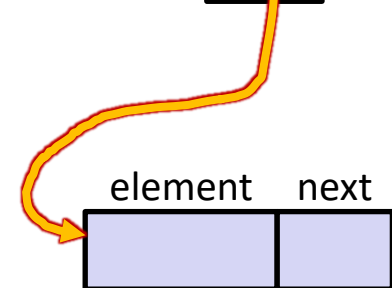
*only use assert in
testing code!*

(main) list 

(Push) head 

(Push) e 

(Push) n 



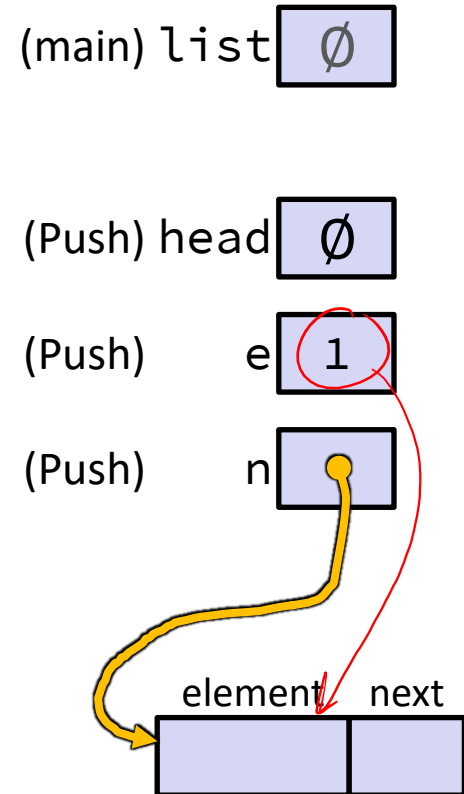
push_list.c

Push Onto List (4/14)

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

Node* Push(Node* head, int e) {
    Node* n = (Node*) malloc(sizeof(Node));
    assert(n != NULL); // crashes if false
    n->element = e;
    n->next = head;
    return n;
}

int main(int argc, char** argv) {
    Node* list = NULL;
    list = Push(list, 1);
    list = Push(list, 2);
    return EXIT_SUCCESS;
}
```



push_list.c

Push Onto List (5/14)

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

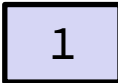
Node* Push(Node* head, int e) {
    Node* n = (Node*) malloc(sizeof(Node));
    assert(n != NULL); // crashes if false
    n->element = e;
    n->next = head;
    return n;
}

int main(int argc, char** argv) {
    Node* list = NULL;
    list = Push(list, 1);
    list = Push(list, 2);
    return EXIT_SUCCESS;
}
```

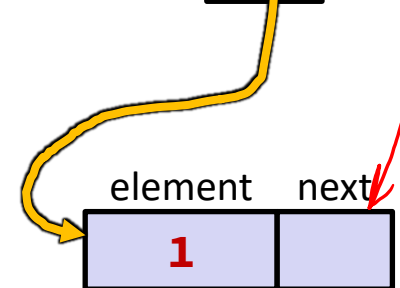


(main) list 

(Push) head 

(Push) e 

(Push) n 



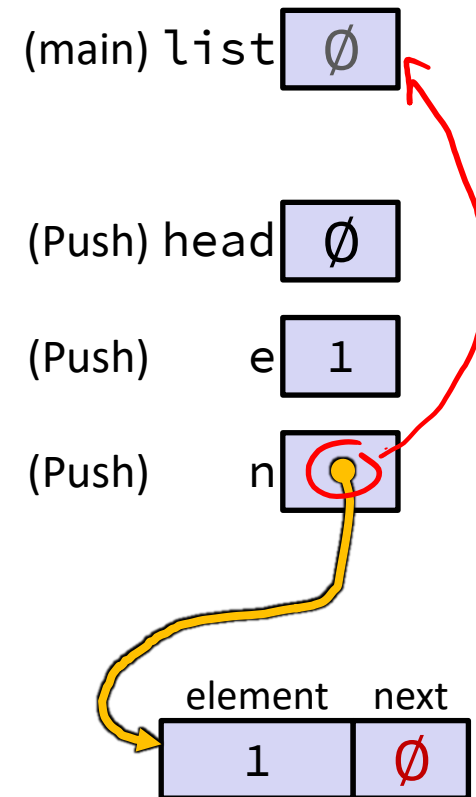
push_list.c

Push Onto List (6/14)

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

Node* Push(Node* head, int e) {
    Node* n = (Node*) malloc(sizeof(Node));
    assert(n != NULL); // crashes if false
    n->element = e;
    n->next = head;
    return n;
}

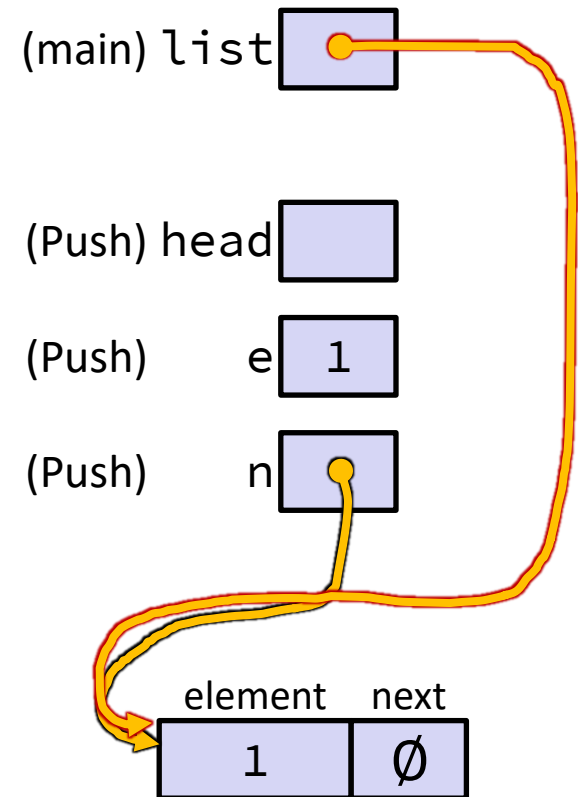
int main(int argc, char** argv) {
    Node* list = NULL;
    list = Push(list, 1);
    list = Push(list, 2);
    return EXIT_SUCCESS;
}
```



push_list.c

Push Onto List (7/14)

```
typedef struct node_st {  
    int element;  
    struct node_st* next;  
} Node;  
  
Node* Push(Node* head, int e) {  
    Node* n = (Node*) malloc(sizeof(Node));  
    assert(n != NULL); // crashes if false  
    n->element = e;  
    n->next = head;  
    return n;  
}  
  
int main(int argc, char** argv) {  
    Node* list = NULL;  
    list = Push(list, 1);  
    list = Push(list, 2);  
    return EXIT_SUCCESS;  
}
```



push_list.c

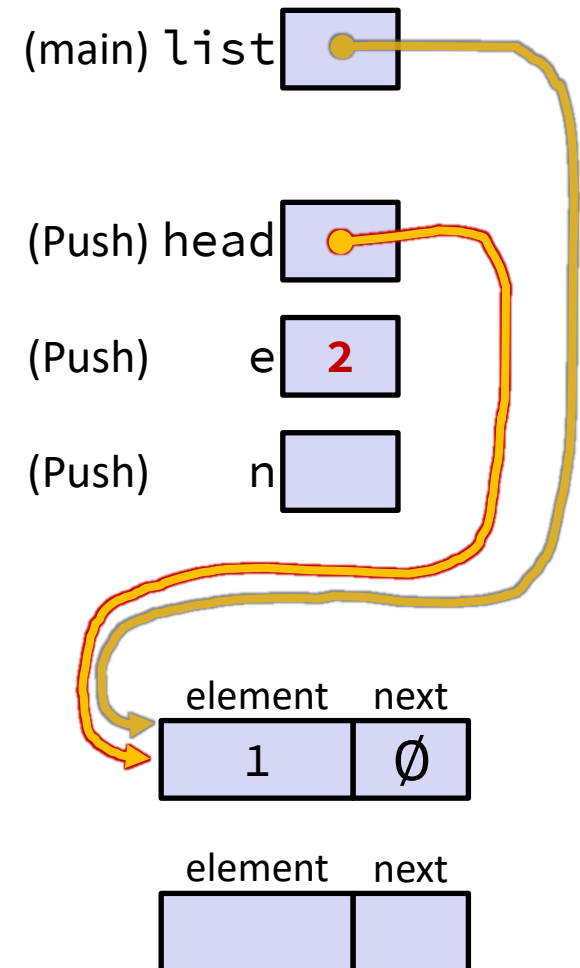
Push Onto List (8/14)

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

Node* Push(Node* head, int e) {
    Node* n = (Node*) malloc(sizeof(Node));
    assert(n != NULL); // crashes if false
    n->element = e;
    n->next = head;
    return n;
}

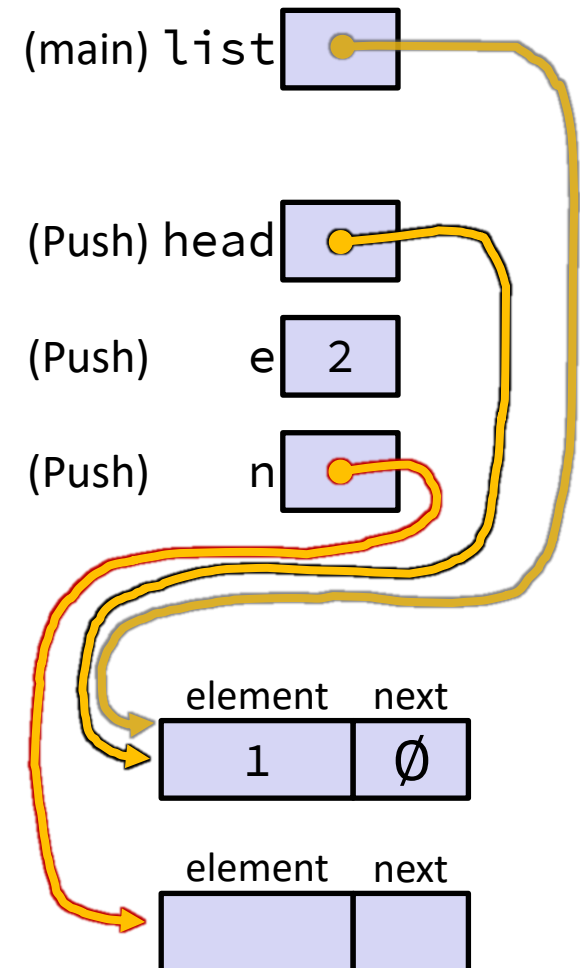
int main(int argc, char** argv) {
    Node* list = NULL;
    list = Push(list, 1);
    list = Push(list, 2);
    return EXIT_SUCCESS;
}
```

push_list.c



Push Onto List (9/14)

```
typedef struct node_st {  
    int element;  
    struct node_st* next;  
} Node;  
  
Node* Push(Node* head, int e) {  
    Node* n = (Node*) malloc(sizeof(Node));  
    assert(n != NULL); // crashes if false  
    n->element = e;  
    n->next = head;  
    return n;  
}  
  
int main(int argc, char** argv) {  
    Node* list = NULL;  
    list = Push(list, 1);  
    list = Push(list, 2);  
    return EXIT_SUCCESS;  
}
```

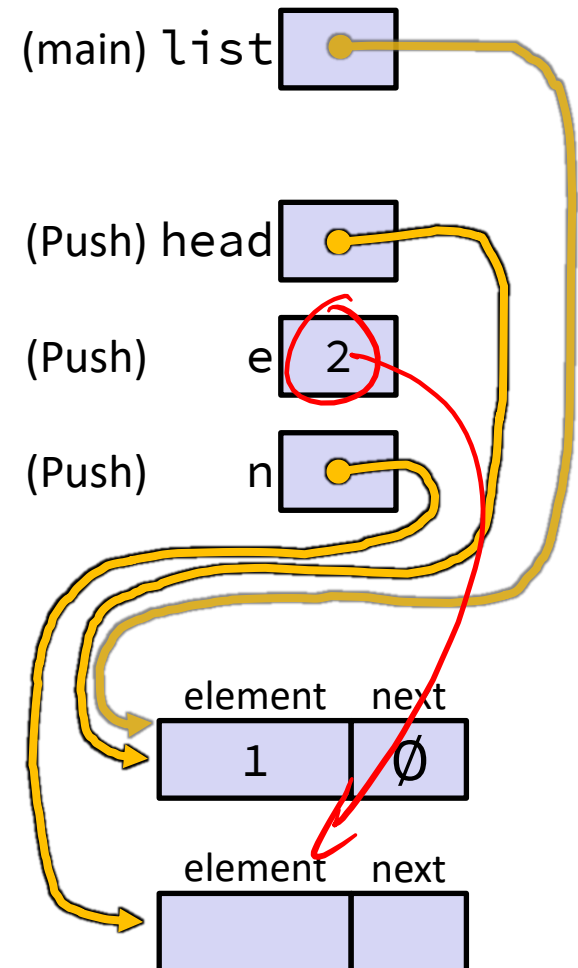


push_list.c

Push Onto List (10/14)

```
typedef struct node_st {  
    int element;  
    struct node_st* next;  
} Node;  
  
Node* Push(Node* head, int e) {  
    Node* n = (Node*) malloc(sizeof(Node));  
    assert(n != NULL); // crashes if false  
    n->element = e;  
    n->next = head;  
    return n;  
}  
  
int main(int argc, char** argv) {  
    Node* list = NULL;  
    list = Push(list, 1);  
    list = Push(list, 2);  
    return EXIT_SUCCESS;  
}
```

push_list.c

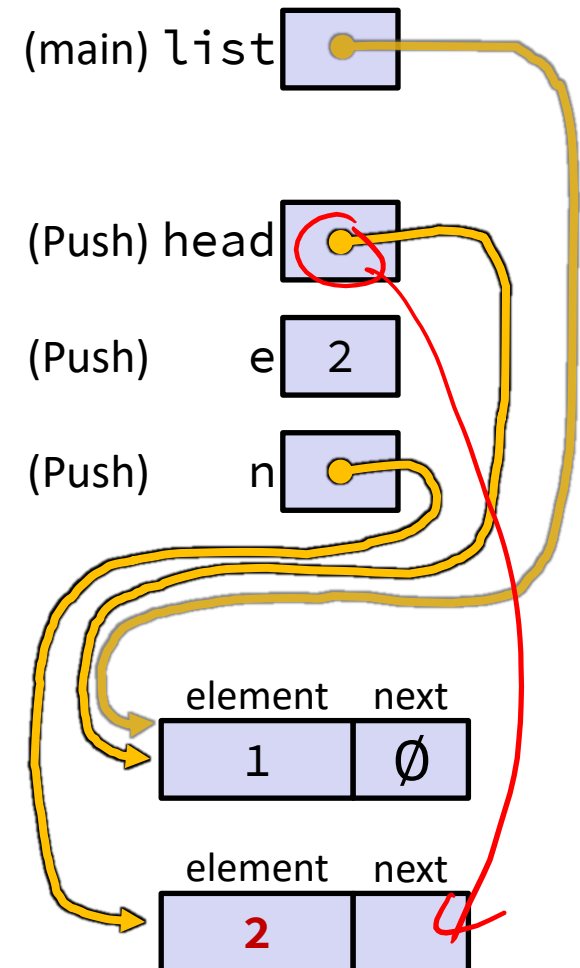


Push Onto List (11/14)

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

Node* Push(Node* head, int e) {
    Node* n = (Node*) malloc(sizeof(Node));
    assert(n != NULL); // crashes if false
    n->element = e;
    n->next = head;
    return n;
}

int main(int argc, char** argv) {
    Node* list = NULL;
    list = Push(list, 1);
    list = Push(list, 2);
    return EXIT_SUCCESS;
}
```



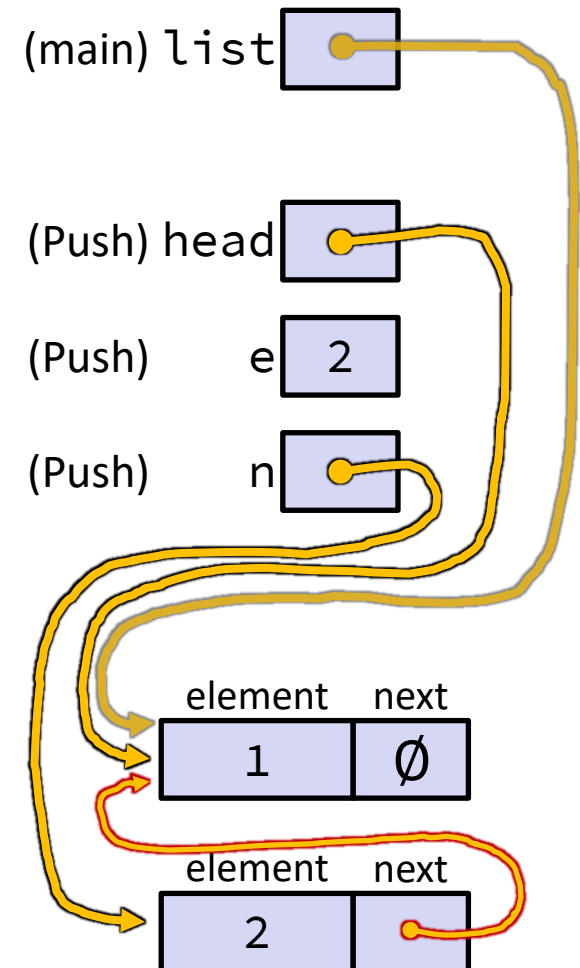
push_list.c

Push Onto List (12/14)

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

Node* Push(Node* head, int e) {
    Node* n = (Node*) malloc(sizeof(Node));
    assert(n != NULL); // crashes if false
    n->element = e;
    n->next = head;
    return n;
}

int main(int argc, char** argv) {
    Node* list = NULL;
    list = Push(list, 1);
    list = Push(list, 2);
    return EXIT_SUCCESS;
}
```



push_list.c

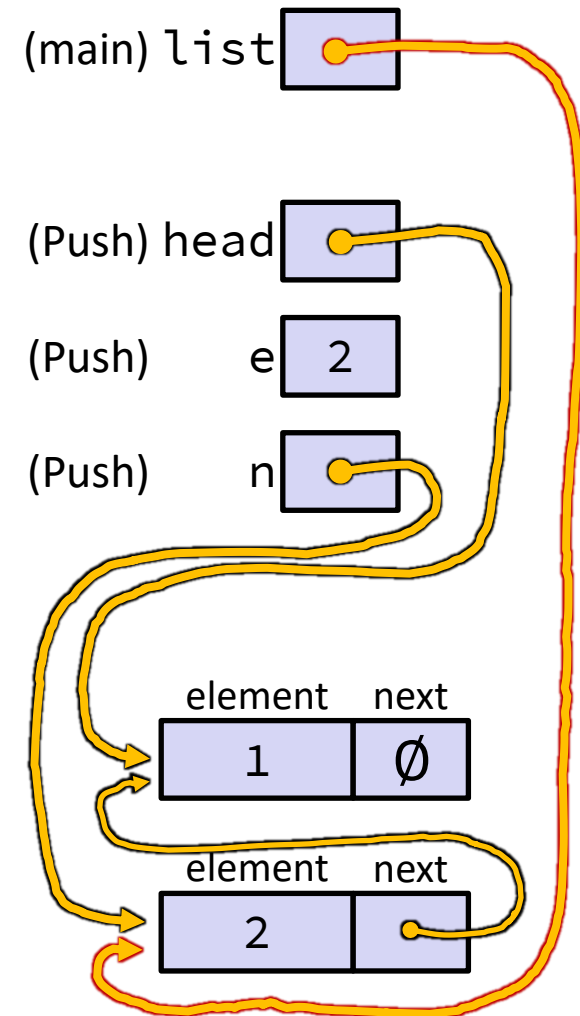
Push Onto List (13/14)

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

Node* Push(Node* head, int e) {
    Node* n = (Node*) malloc(sizeof(Node));
    assert(n != NULL); // crashes if false
    n->element = e;
    n->next = head;
    return n;
}

int main(int argc, char** argv) {
    Node* list = NULL;
    list = Push(list, 1);
    list = Push(list, 2);
    return EXIT_SUCCESS;
}
```

push_list.c



Push Onto List (14/14)

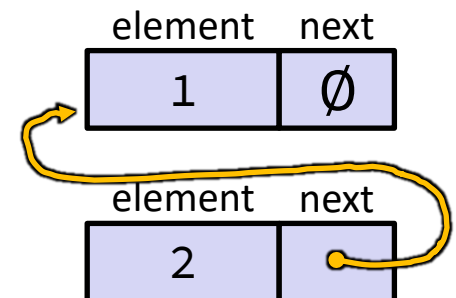
```
typedef struct node_st {
    int element;
    struct node_st* next;
} Node;

Node* Push(Node* head, int e) {
    Node* n = (Node*) malloc(sizeof(Node));
    assert(n != NULL); // crashes if false
    n->element = e;
    n->next = head;
    return n;
}

int main(int argc, char** argv) {
    Node* list = NULL;
    list = Push(list, 1);
    list = Push(list, 2);
    return EXIT_SUCCESS;
}
```



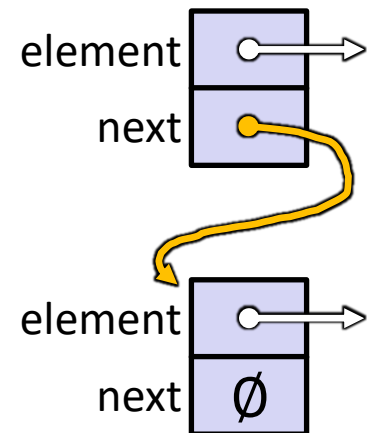
push_list.c



A Generic Linked List

- ❖ Let's generalize the linked list element type
 - Let customer decide type (instead of always `int`)
 - Idea: let them use a generic pointer (*i.e.*, a `void*`)

```
typedef struct node_st {  
    void* element;  
    struct node_st* next;  
} Node;  
  
Node* Push(Node* head, void* e) {  
    Node* n = (Node*) malloc(sizeof(Node));  
    assert(n != NULL); // crashes if false  
    n->element = e;  
    n->next = head;  
    return n;  
}
```





Using a Generic Linked List

- ❖ Type casting should be used with `void*` (raw address)
 - Before pushing, should convert to `void*`
 - Convert back to data type when accessing

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

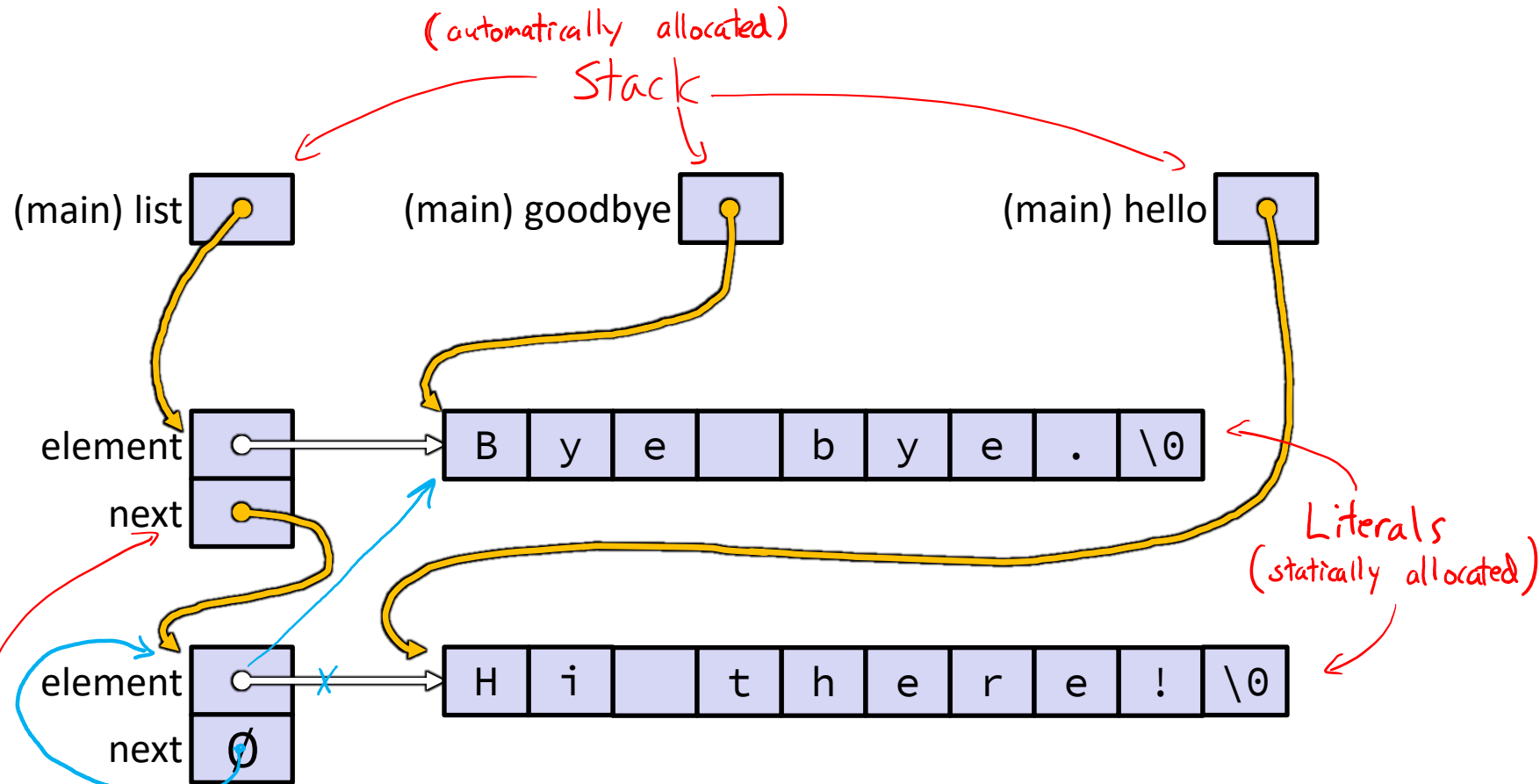
Node* Push(Node* head, void* e);    // assume last slide's code

int main(int argc, char** argv) {
    char* hello = "Hi there!";
    char* goodbye = "Bye bye.";
    Node* list = NULL;

    list = Push(list, (void*) hello);
    list = Push(list, (void*) goodbye);
    printf("payload: '%s'\n", (char*) ((list->next)->element) );
    return EXIT_SUCCESS;
}
```

manual_list_void.c

Resulting Memory Diagram



What would happen if we execute `*(&list->next) = *list;`?

Something's Fishy...

- ❖ A (benign) memory leak!

```
int main(int argc, char** argv) {  
    char* hello = "Hi there!";  
    char* goodbye = "Bye bye.";  
    Node* list = NULL;  
  
    list = Push(list, (void*) hello);  
    list = Push(list, (void*) goodbye);  
    return EXIT_SUCCESS;  
}
```



- ❖ Try running with Valgrind:

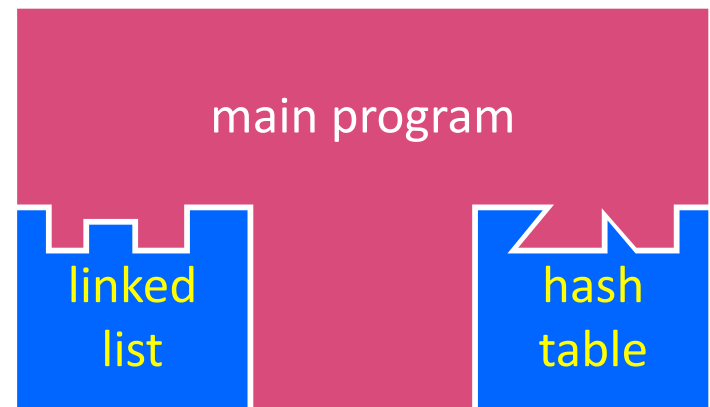
```
$ gcc -Wall -g -o manual_list_void manual_list_void.c  
$ valgrind --leak-check=full ./manual_list_void
```

Lecture Outline (2/3)

- ❖ Generic Data Structures in C
- ❖ **Modules & Interfaces**
- ❖ C Preprocessor Intro

Multi-File C Programs

- ❖ Let's create a linked list *module*
 - A module is a self-contained piece of an overall program
 - Has externally visible functions that customers can invoke
 - Has externally visible typedefs, and perhaps global variables, that customers can use
 - May have internal functions, typedefs, or global variables that customers should *not* look at
 - Can be developed independently and re-used in different projects
- ❖ The module's *interface* is its set of public functions, typedefs, and global variables



C Header Files

- ❖ **Header:** A file whose only purpose is to be `#include`'d
 - Generally has a filename `.h` extension
 - Holds the variables, types, and function prototype declarations that make up the interface to a module *(not definitions)*
 - There are `<system-defined>` and `"programmer-defined"` headers
 - `#include <stdio.h>`*
 - `#include "my_header.h"`*
- ❖ **Main Idea:**
 - Every name `.c` is intended to be a module that has a name `.h`
 - `name.h` declares the interface to that module
 - Other C files can use module name by `#include`-ing `name.h`
 - They should assume as little as possible about the implementation in `name.c`



C Module Conventions (1/2)

❖ File contents:

- .h files only contain *declarations*, never *definitions*
- .c files never contain prototype declarations for functions that are intended to be exported through the module interface
- Public-facing functions are ^{Generic Linked List_Push()} **ModuleName_FunctionName**() and take a pointer to “this” as their first argument

❖ Including:

- **NEVER** #include a .c file – only #include .h files
- #include all of headers you reference, even if another header (transitively) includes some of them

❖ Compiling:

- Any .c file with an associated .h file should be able to be compiled (together via #include) into a .o file



C Module Conventions (2/2)

❖ Commenting:

- If a function is declared in a header file (.h) and defined in a C file (.c), *the header needs full documentation because it is the public specification*
 - Don't copy-paste the comment into the C file (don't want two copies that can get out of sync)
- If prototype and implementation are in the same C file:
 - School of thought #1: Full comment on the prototype at the top of the file, no comment (or “declared above”) on code
 - School of thought #2: Prototype is for the compiler and doesn't need comment; comment the code to keep them together

e.g., 333

project code

Lecture Outline (3/3)

- ❖ Generic Data Structures in C
- ❖ Modules & Interfaces
- ❖ **C Preprocessor Intro**

#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

pollev.com/cse333a

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 (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;  
}
```

Program Using a Linked List

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

Node* Push(Node* head,
           void* element) {
    ... // implementation 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;
}
```

example_ll_customer.c

Compiling the Program

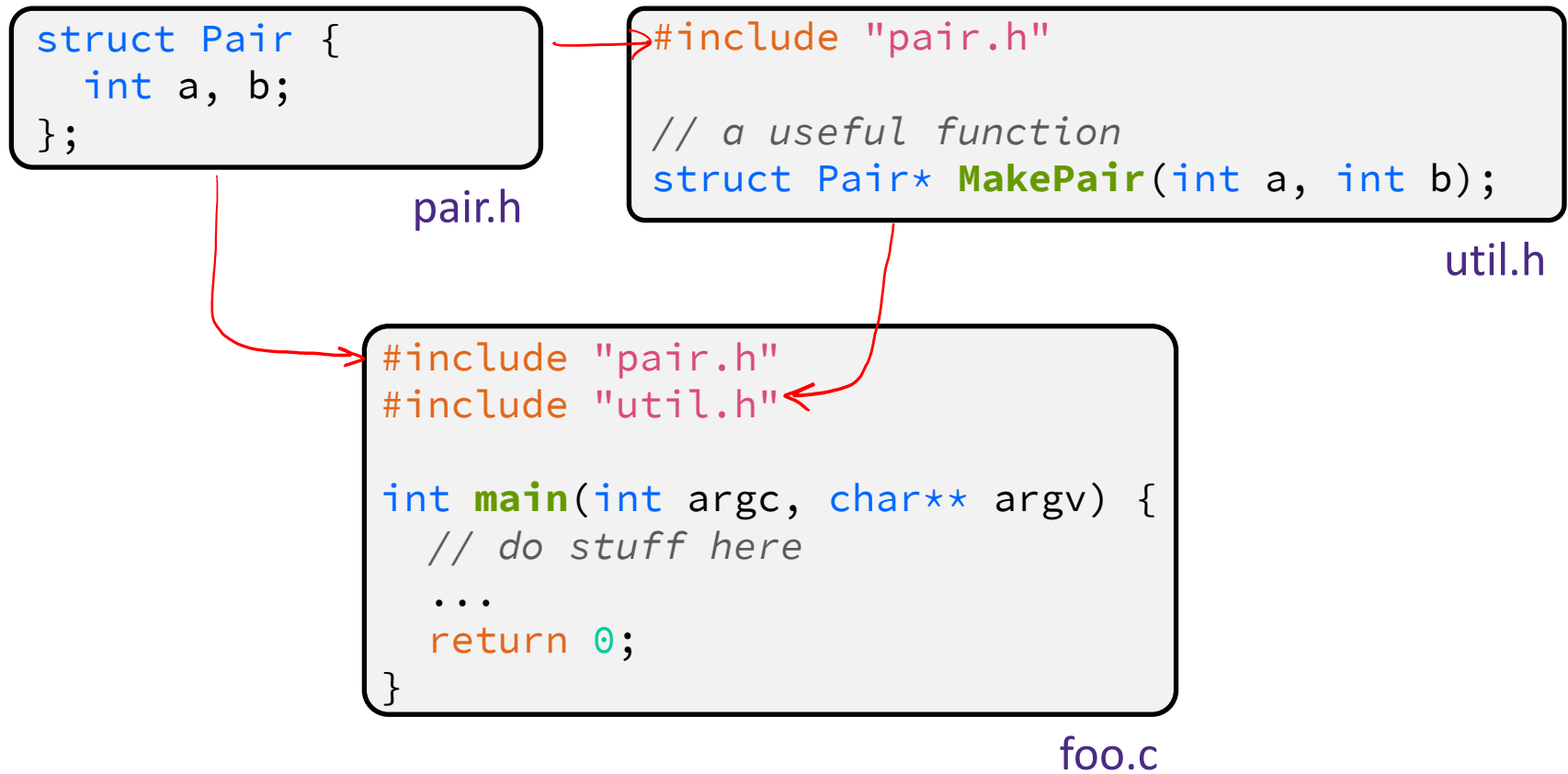
❖ Four parts:

- 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

```
① $ gcc -Wall -g -o example_ll_customer.o -c example_ll_customer.c
② $ gcc -Wall -g -o ll.o -c ll.c
③ $ gcc -g -o example_ll_customer ll.o example_ll_customer.o
④ $ ./example_ll_customer
Payload: 'yo!'
Payload: 'goodbye'
Payload: 'hello'
④ $ valgrind --leak-check=full ./example_ll_customer
... etc ...
```

But There's a Problem with #include

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

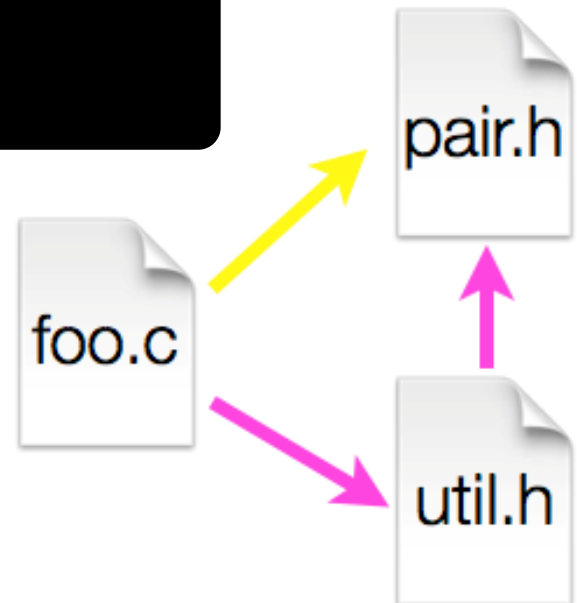


A Problem with #include

- ❖ 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

❖ A standard C Preprocessor trick to deal with this

preprocessor state

① PAIR_H_ ✓
② UTIL_H_ ✓

- Uses macro definition (**#define**) in combination with conditional compilation (**#ifndef** and **#endif**)

① not defined ③ defined
~~#ifndef PAIR_H_ {~~
~~#define PAIR_H_~~
 ① now define
 struct Pair {
 int a, b;
 };
 }#endif // PAIR_H_

pair.h

includes struct pair

empty
include!

② not defined
 #ifndef UTIL_H_
 #define UTIL_H_
 ② now define
 ③ #include "pair.h" ← even if #included twice,
 header guard PAIR_H_
 saves us!
 // a useful function
 struct Pair* MakePair(int a, int b);
 #endif // UTIL_H_

util.h

foo.c

① ~~#include "pair.h"~~
 ② ~~#include "util.h"~~
 int main(int argc, char** argv) {



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

*More preprocessor tricks
next lecture!*

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

- ❖ Implement and test a binary search tree
 - https://en.wikipedia.org/wiki/Binary_search_tree
 - Don't worry about making it balanced
 - Implement key insert() and lookup() functions
 - Bonus: implement a key delete() function
 - Implement it as a C module
 - bst.c, bst.h
 - Implement test_bst.c
 - Contains main() and tests out your BST

Extra Exercise #3

- ❖ Implement a Complex number module
 - `complex.c`, `complex.h`
 - Includes a typedef to define a complex number
 - $a + bi$, where a and b are doubles
 - Includes functions to:
 - add, subtract, multiply, and divide complex numbers
 - Implement a test driver in `test_complex.c`
 - Contains `main()`