

Data Structures, Modules, and CPP

CSE 333 Spring 2020

Instructors: Travis McGaha, Justin Hsia

Teaching Assistants:

Arthava Deodhar

Callum Walker

Cosmo Wang

Dylan Hartono

Elizabeth Haker

Kyrie Dowling

Leo Liao

Markus Schiffer

Neha Nagvekar

Nonthakit Chaiwong

Ramya Challa



pollev.com/cse333travis

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

Administrivia

- ❖ Exercise 2 was due this morning
 - There may be a tweak to the autograder “correctness” grade

- ❖ Exercise 3 out today, due Monday morning
 - Depends on content from Today’s and Monday’s lecture

- ❖ Section code “handout” for tomorrow will be added to the calendar later today
 - Will be about gdb, valgrind, and structs
 - GDB is very helpful!
 - Suggestion: download/print a copy ahead of time

Administrivia

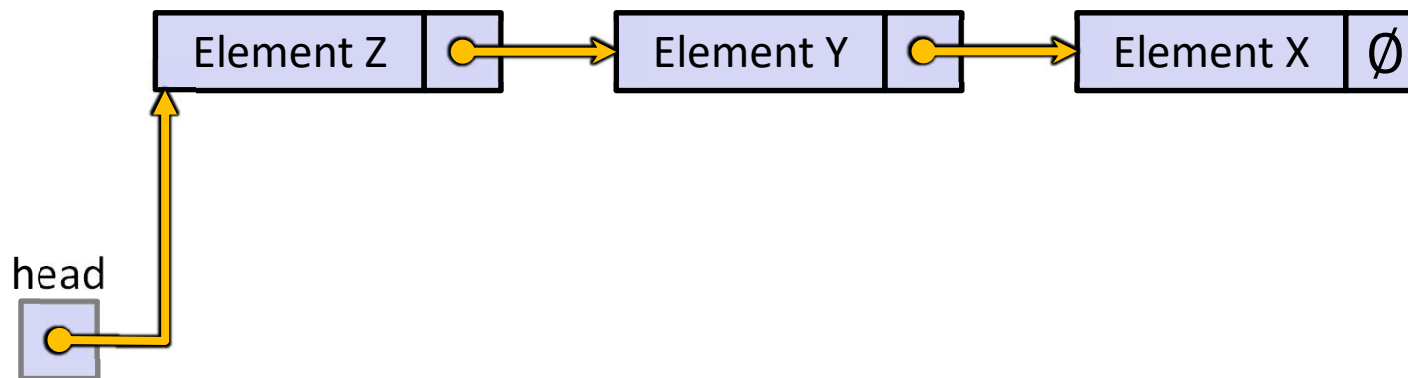
- ❖ HW1 due a week from Thursday
 - You should be well under way now
 - 🙄 🙄 🙄 😏 if not
 - Be sure to read headers *carefully* while implementing
 - Use git add/commit/push regularly to save work when you finish another part of the job
 - And also please do this before contacting TA during office hours if you want help with your code

Lecture Outline

- ❖ **Implementing 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`

```

#include <stdio.h>
typedef struct node_st {
    int element;
    struct node_st* next;
} Node;

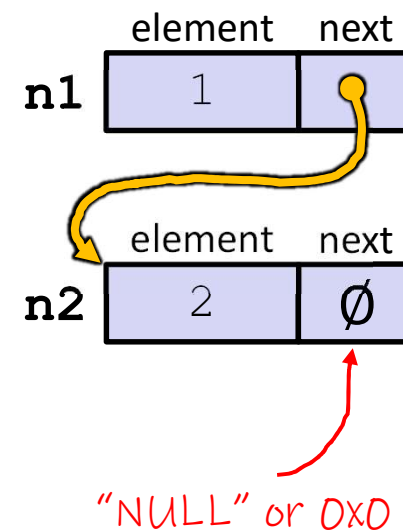
int main(int argc, char** argv) {
    Node n1, n2;

    n1.element = 1;
    n1.next = &n2;
    n2.element = 2;
    n2.next = NULL;
    return 0;
}
    
```

Handwritten annotations:

- tagname needed* (points to `node_st`)
- Pointer is part of definition* (points to `struct node_st*`)
- On the stack* (points to `Node n1, n2;`)

manual_list.c



Push Onto List

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 0;
}
```

(main) list



push_list.c


Push Onto List


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;
}

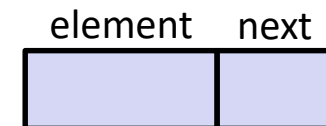
int main(int argc, char** argv) {
    Node* list = NULL;
    list = Push(list, 1);
    list = Push(list, 2);
    return 0;
}
```

(main) list 

(Push) head 

(Push) e 

(Push) n 



push_list.c

Push Onto List

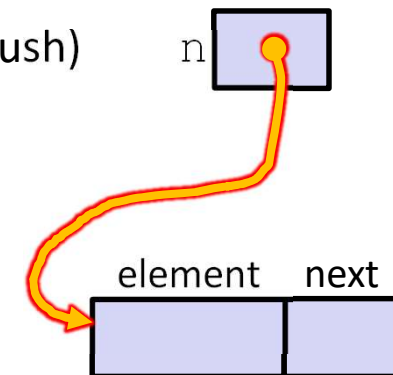
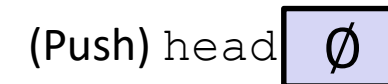
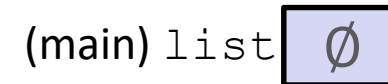
Arrow points to *next* instruction.

```

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 0;
}
    
```



push_list.c

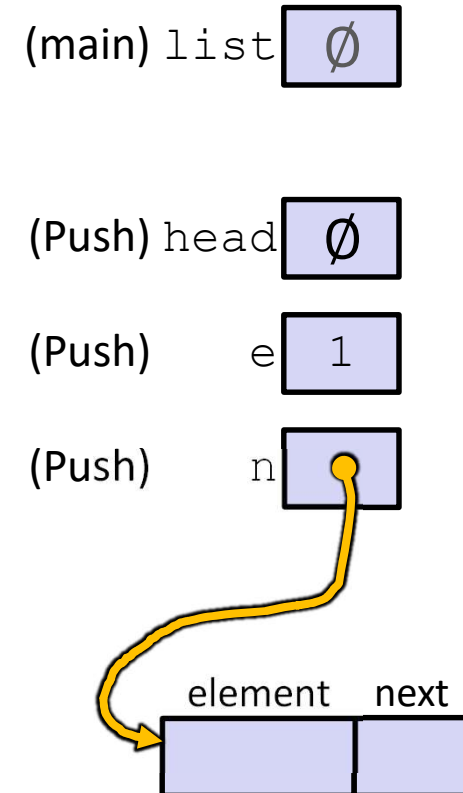
Push Onto List

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 0;
}
```



push_list.c

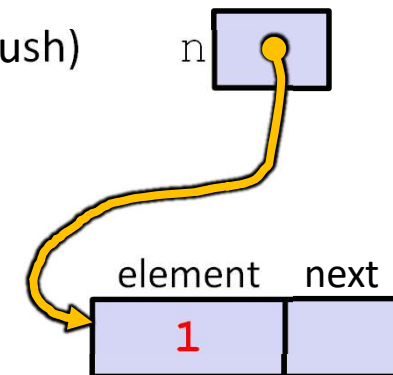
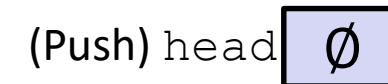
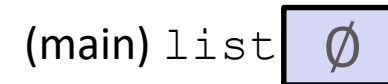
Push Onto List

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 0;
}
```



push_list.c

Push Onto List

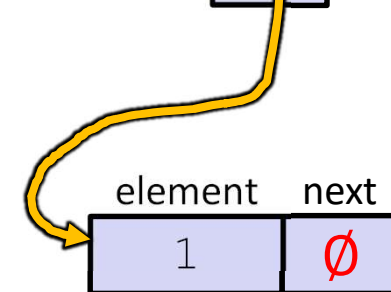
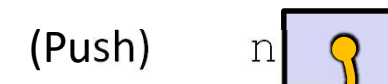
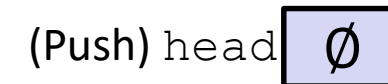
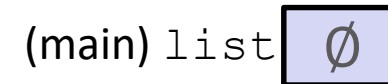
Arrow points to *next* instruction.

```

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 0;
}
    
```



push_list.c

Push Onto List

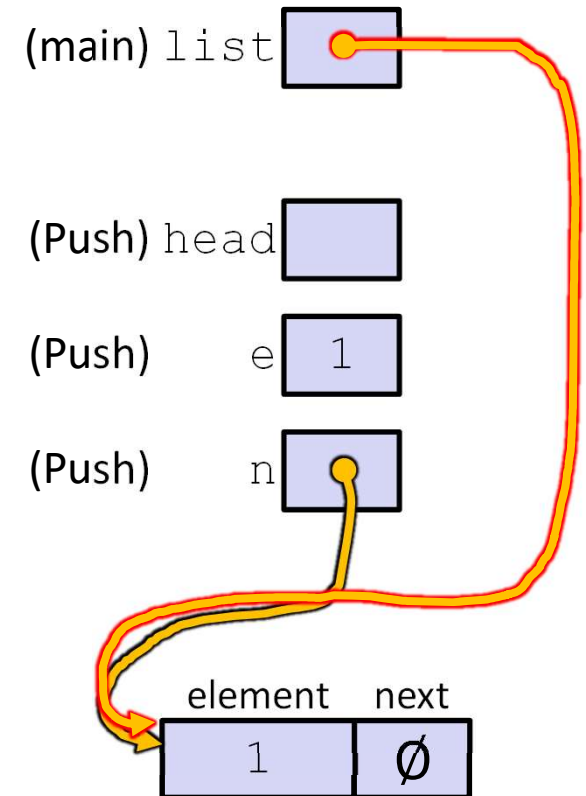
Arrow points to *next* instruction.

```

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 0;
}
    
```



push_list.c

Push Onto List

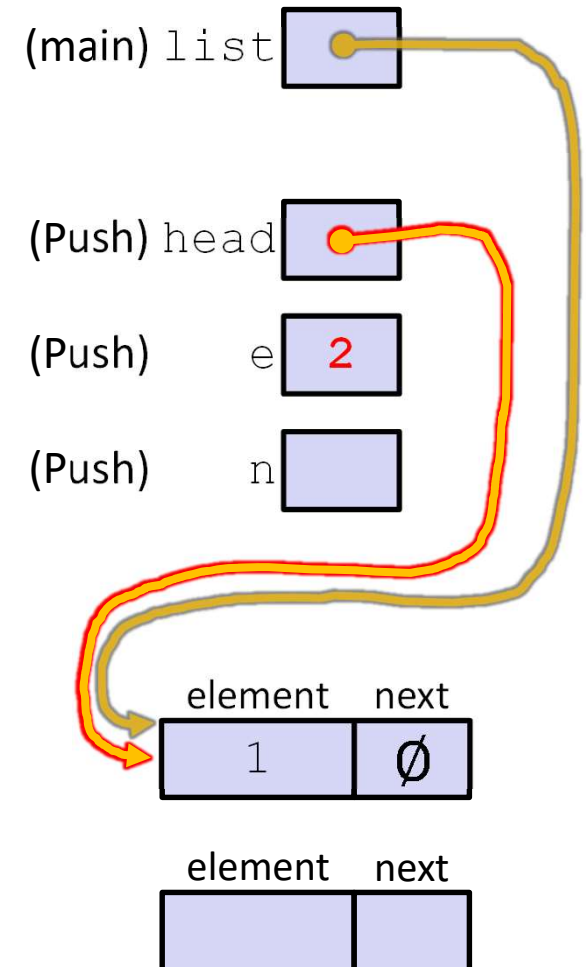
Arrow points to *next* instruction.

```

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 0;
}
    
```



push_list.c

Push Onto List

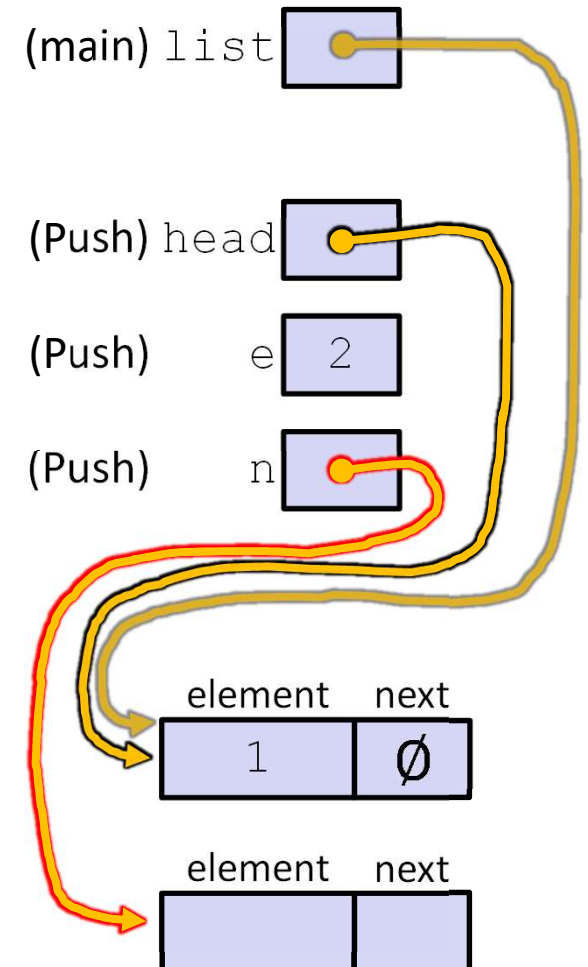
Arrow points to *next* instruction.

```

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 0;
}
    
```



push_list.c

Push Onto List

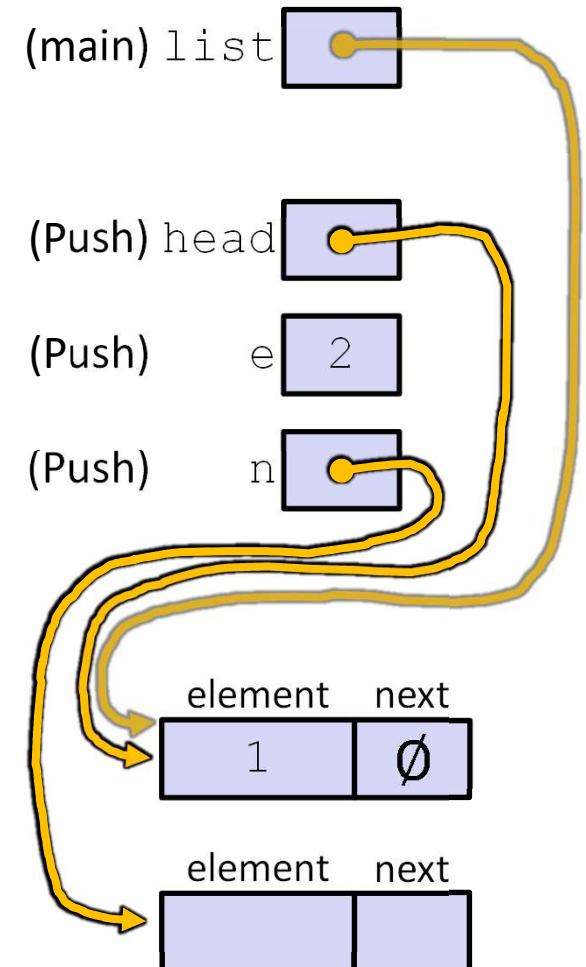
Arrow points to *next* instruction.

```

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 0;
}
    
```



push_list.c

Push Onto List

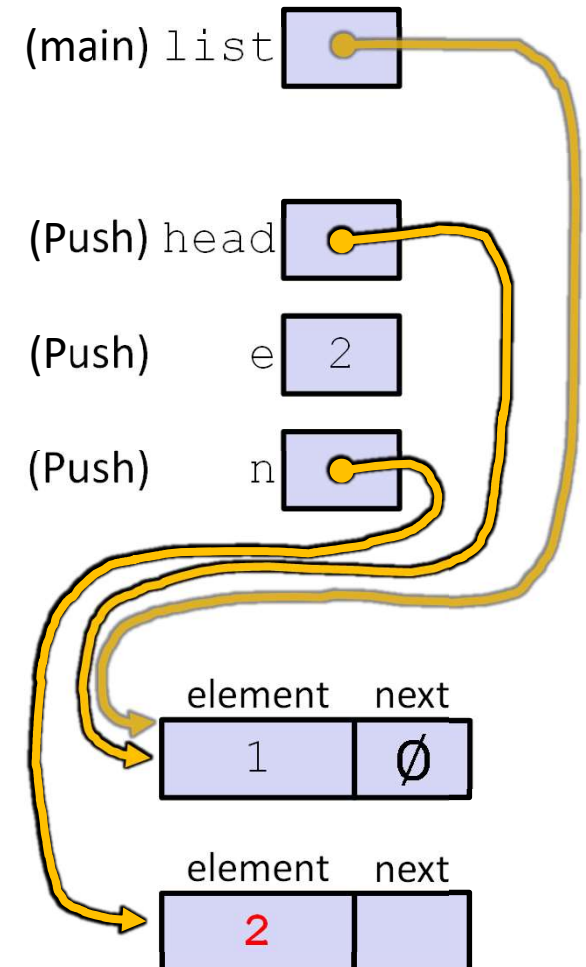
Arrow points to *next* instruction.

```

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 0;
}
    
```



push_list.c

Push Onto List

Arrow points to *next* instruction.

```

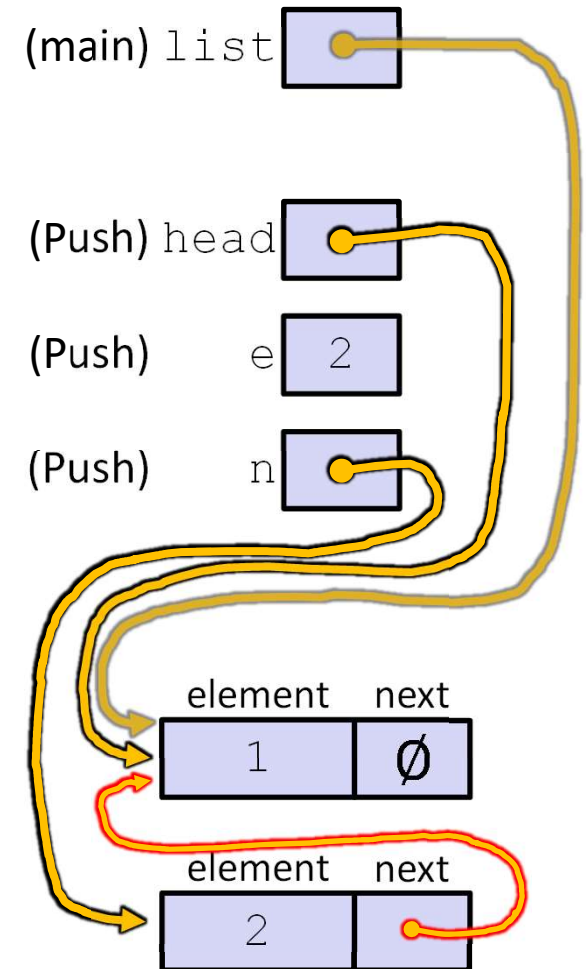
typedef struct node_st {
    int element;
    struct node_st* next;
} Node;

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 0;
}
    
```



push_list.c



Push Onto List

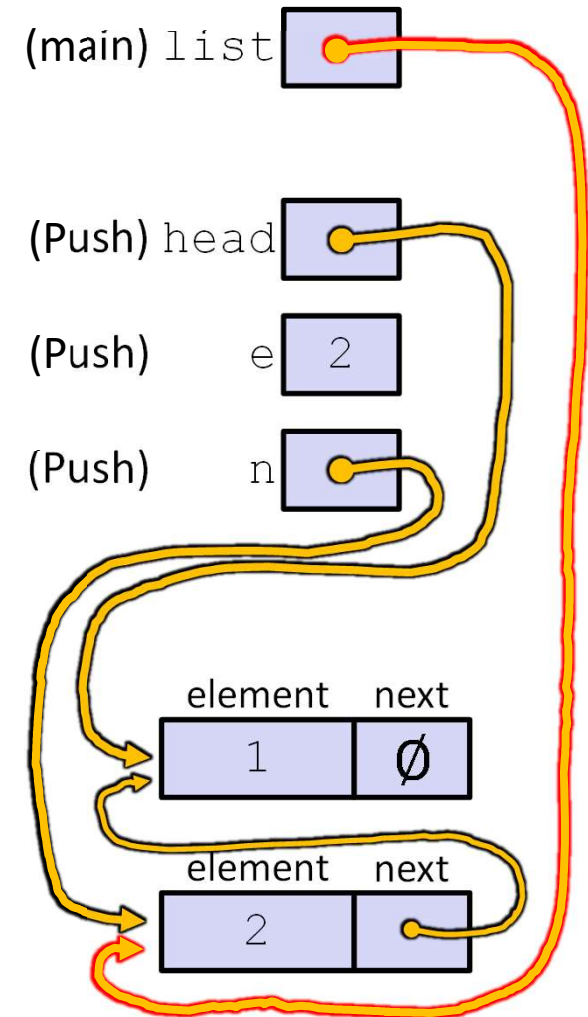
Arrow points to *next* instruction.

```

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 0;
}
    
```



push_list.c

Push Onto List

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;
}

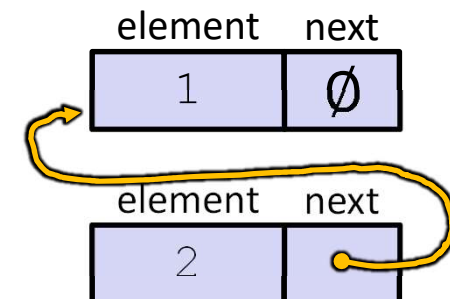
int main(int argc, char** argv) {
    Node* list = NULL;
    list = Push(list, 1);
    list = Push(list, 2);
    return 0;
}
```

push_list.c

A (benign) memory leak!
Try running with Valgrind:

```
bash$ gcc -Wall -g -o
push_list push_list.c

bash$ valgrind --leak-
check=full ./push_list
```

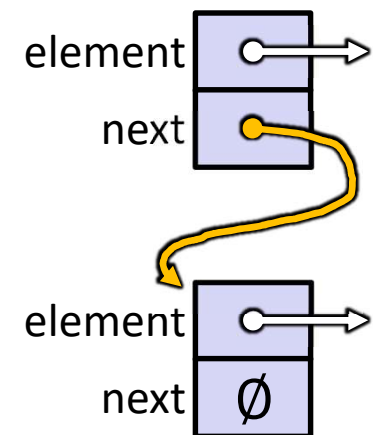


A Generic Linked List

- ❖ 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 needed to deal with `void*` (raw address)
 - Before pushing, need to 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

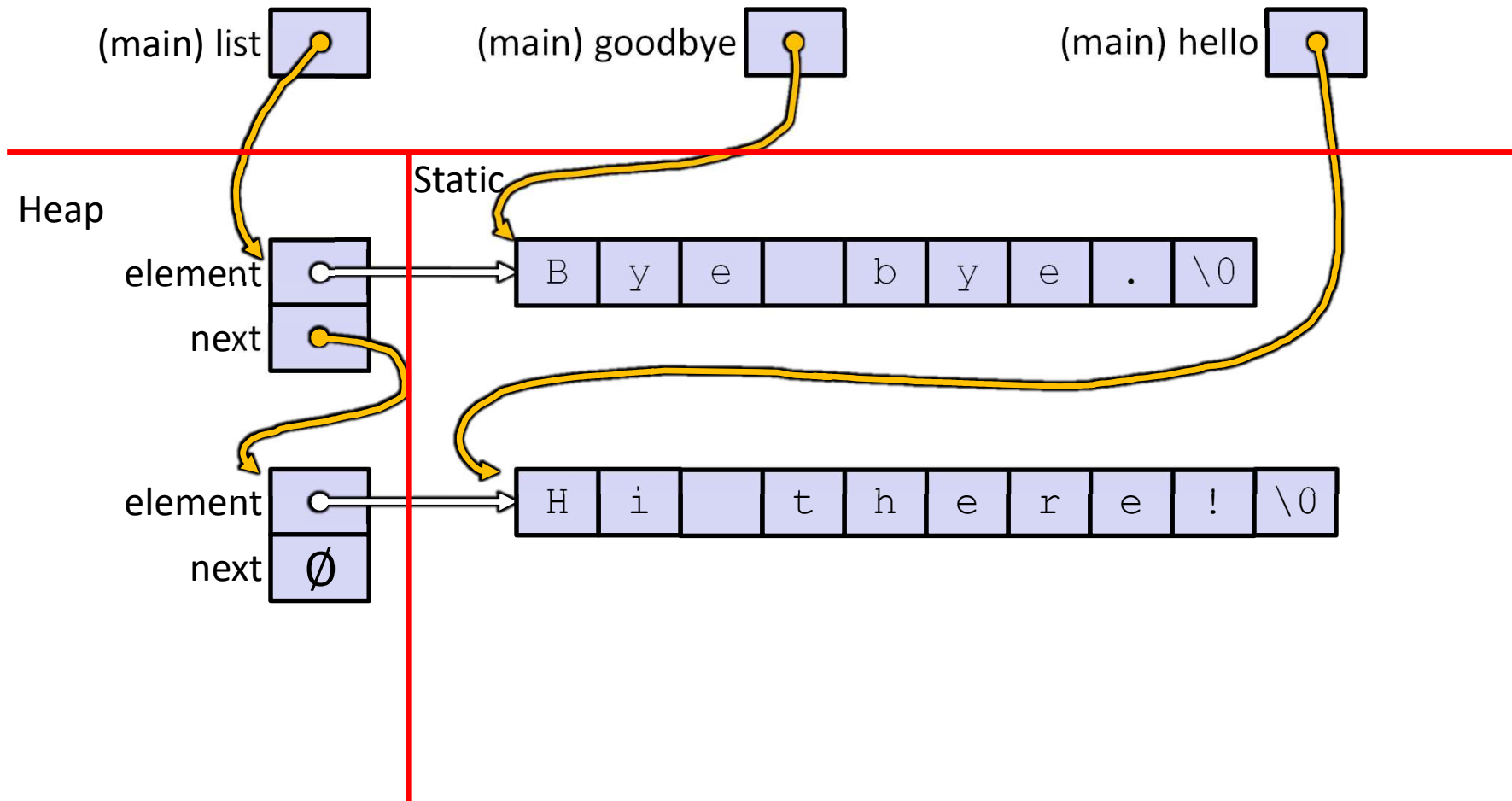
int main(int argc, char** argv) {
    char* hello = "Hi there!";
    char* goodbye = "Bye bye.";
    Node* list = NULL;

    list = Push(list, (void*) hello);
    list = Push(list, (void*) goodbye);
    printf("payload: '%s'\n", (char*) ((list->next)->element) );
    return 0;
}
```

manual_list_void.c

Resulting Memory Diagram

Stack



Lecture Outline

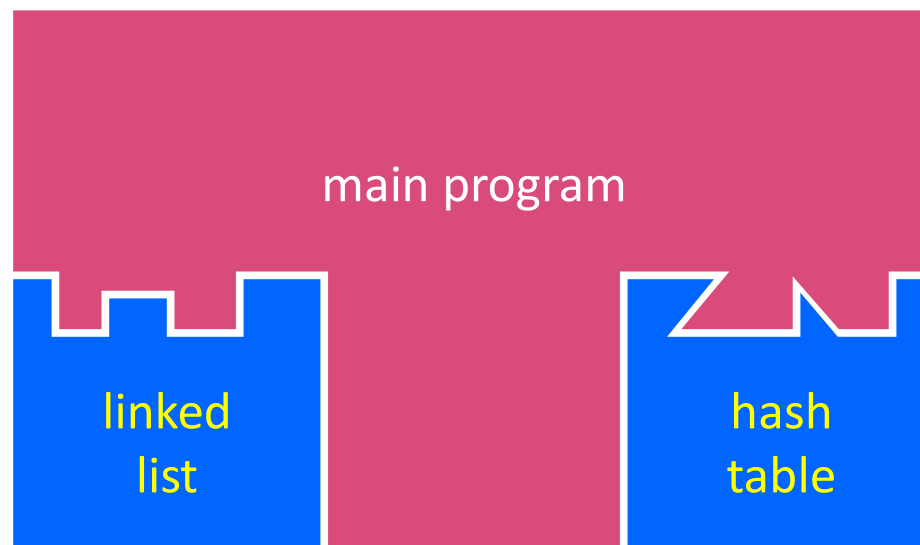
- ❖ Implementing Data Structures in C
- ❖ **Modules & Interfaces**
- ❖ C Preprocessor Intro

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
 - The module's *interface* is its set of public functions, `typedefs`, and global variables

Modularity

- ❖ The degree to which components of a system can be separated and recombined
 - “Loose coupling” and “separation of concerns”
 - Modules can be developed independently
 - Modules can be re-used in different projects



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
 - There are <system-defined> and "programmer-defined" headers
 - `#include <stdio.h>`
 - `#include "my_linkedlist.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 modules can use `name` by `#include-ing` `name.h`
 - They should assume as little as possible about the implementation in `name.c`



C Module Conventions (1 of 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 `ModuleName_LinkedList_Pushfunctionname()` and take a pointer to “`this`” as their first argument

❖ Including:

The “object” we are calling the function on

- **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 into a `.o` file
 - The `.c` file should `#include` the `.h` file; the compiler will check definitions and declarations for consistency



C Module Conventions (2 of 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

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



#include and the C Preprocessor

- ❖ The C preprocessor (`cpp`) transforms your source code before the compiler runs – it's a simple copy-and-replace text processor(!) with a memory
 - Input is a C file (text) and output is still a C file (text)
 - 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 (a string!) and replaces later occurrences
 - Several others that we'll see soon...
 - Run on your behalf by `gcc` during compilation
 - Note: `#include <foo.h>` looks in system (library) directories; `#include "foo.h"` looks first in current directory, then system

C Preprocessor Example

- ❖ What do you think the preprocessor output will be?

```
#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

C Preprocessor Example

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`

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

C Preprocessor Example

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

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

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

C Preprocessor Example

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

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

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

C Preprocessor Example

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

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

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

C Preprocessor Example

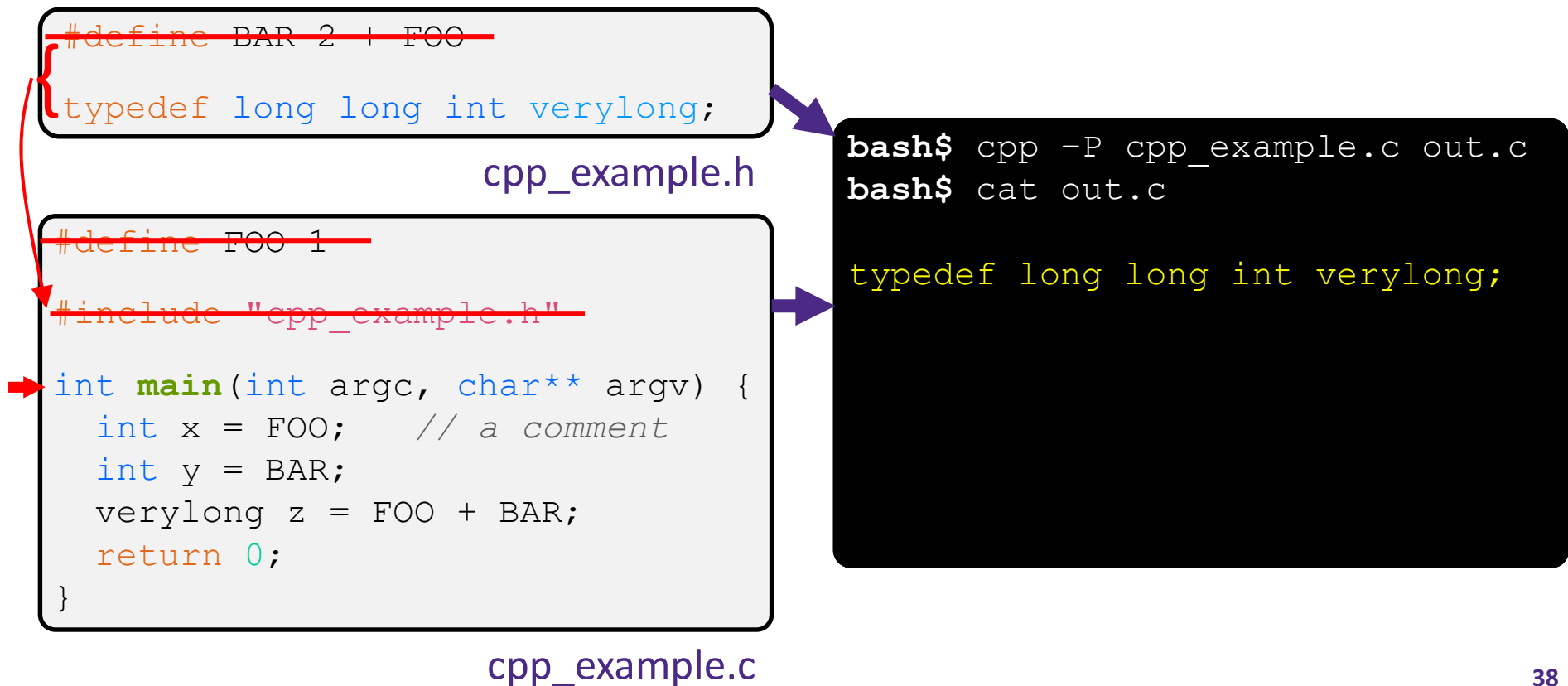
Arrow points to next line to process

❖ We can manually run the preprocessor:

- `cpp` is the preprocessor (can also use `gcc -E`)
- “-P” option suppresses some extra debugging annotations

Pre-processor state

FOO	1
BAR	2 + 1



C Preprocessor Example

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

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

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

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

C Preprocessor Example

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

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

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

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

C Preprocessor Example

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

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

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

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

C Preprocessor Example

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

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

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

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

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

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

Need definitions for code to compile

```
#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

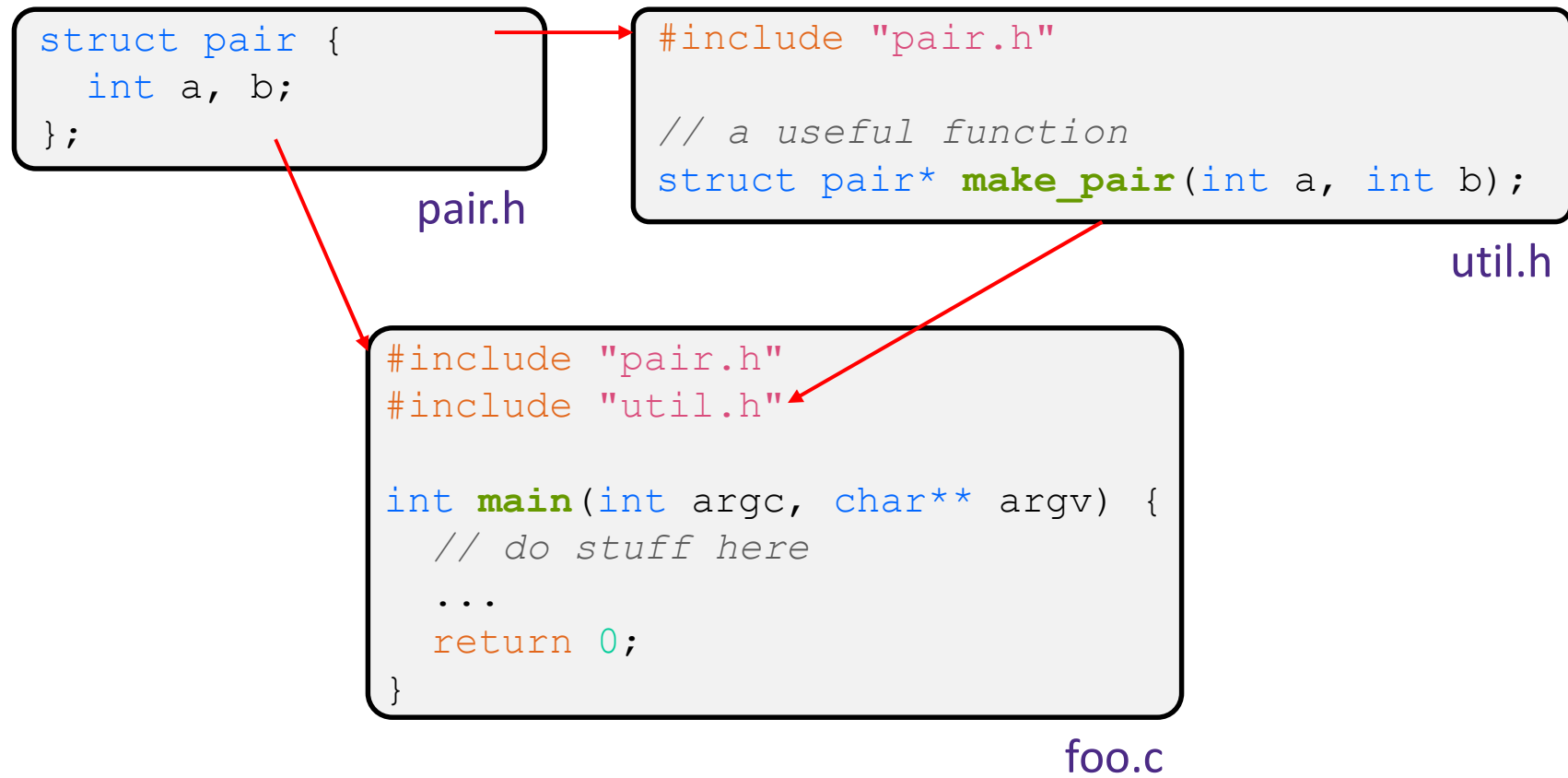
❖ Four parts:

- 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 bash$ gcc -Wall -g -c -o example_ll_customer.o example_ll_customer.c
2 bash$ gcc -Wall -g -c -o ll.o ll.c
3 bash$ gcc -g -o example_ll_customer ll.o example_ll_customer.o
4 bash$ ./example_ll_customer
Payload: 'yo!'
Payload: 'goodbye'
Payload: 'hello'
4 bash$ 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`?

```
bash$ gcc -Wall -g -o foo foo.c
In file included from util.h:1:0,
                from foo.c:2:
pair.h:1:8: error: redefinition of 'struct pair'
  struct pair { int a, b; };
                ^
In file included from foo.c:1:0:
pair.h:1:8: note: originally defined here
  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 Pre-processor state
 - 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* make_pair(int a, int b);

#endif // UTIL_H_
```

util.h

foo.c

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

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



Preprocessor Tricks: Header Guards

- ❖ A standard C Preprocessor trick to deal with this **Pre-processor state**
 - 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* make_pair(int a, int b);

#endif // UTIL_H_
```

util.h

foo.c

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

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



Preprocessor Tricks: Header Guards

- ❖ A standard C Preprocessor trick to deal with this **Pre-processor state**
 - 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* make_pair(int a, int b);

#endif // UTIL_H_

```

util.h

foo.c

```

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

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

```



Preprocessor Tricks: Header Guards

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

PAIR_H_	defined

```
#ifndef PAIR_H_
#define PAIR_H_

struct pair {
    int a, b;
};

#endif // PAIR_H_
```

```
#ifndef UTIL_H_
#define UTIL_H_

#include "pair.h"

// a useful function
struct pair* make_pair(int a, int b);

#endif // UTIL_H_
```

pair.h

util.h

foo.c

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

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



Preprocessor Tricks: Header Guards

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

PAIR_H_	defined

```
#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* make_pair(int a, int b);

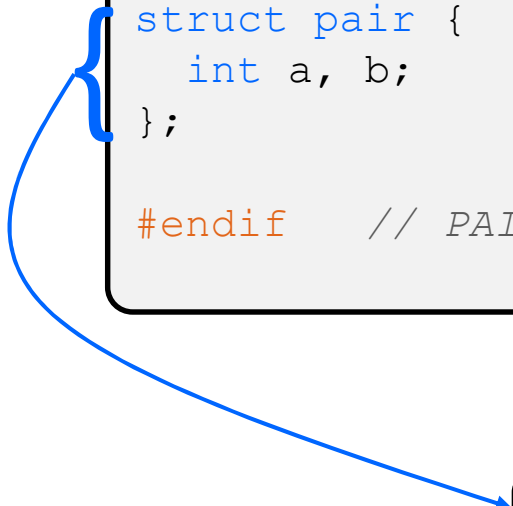
#endif // UTIL_H_
```

util.h

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

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

foo.c





Preprocessor Tricks: Header Guards

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

PAIR_H_	defined

```
#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* make_pair(int a, int b);

#endif // UTIL_H_
```

util.h

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

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

foo.c



Preprocessor Tricks: Header Guards

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

PAIR_H_	defined

```
#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* make_pair(int a, int b);

#endif // UTIL_H_
```

util.h

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

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

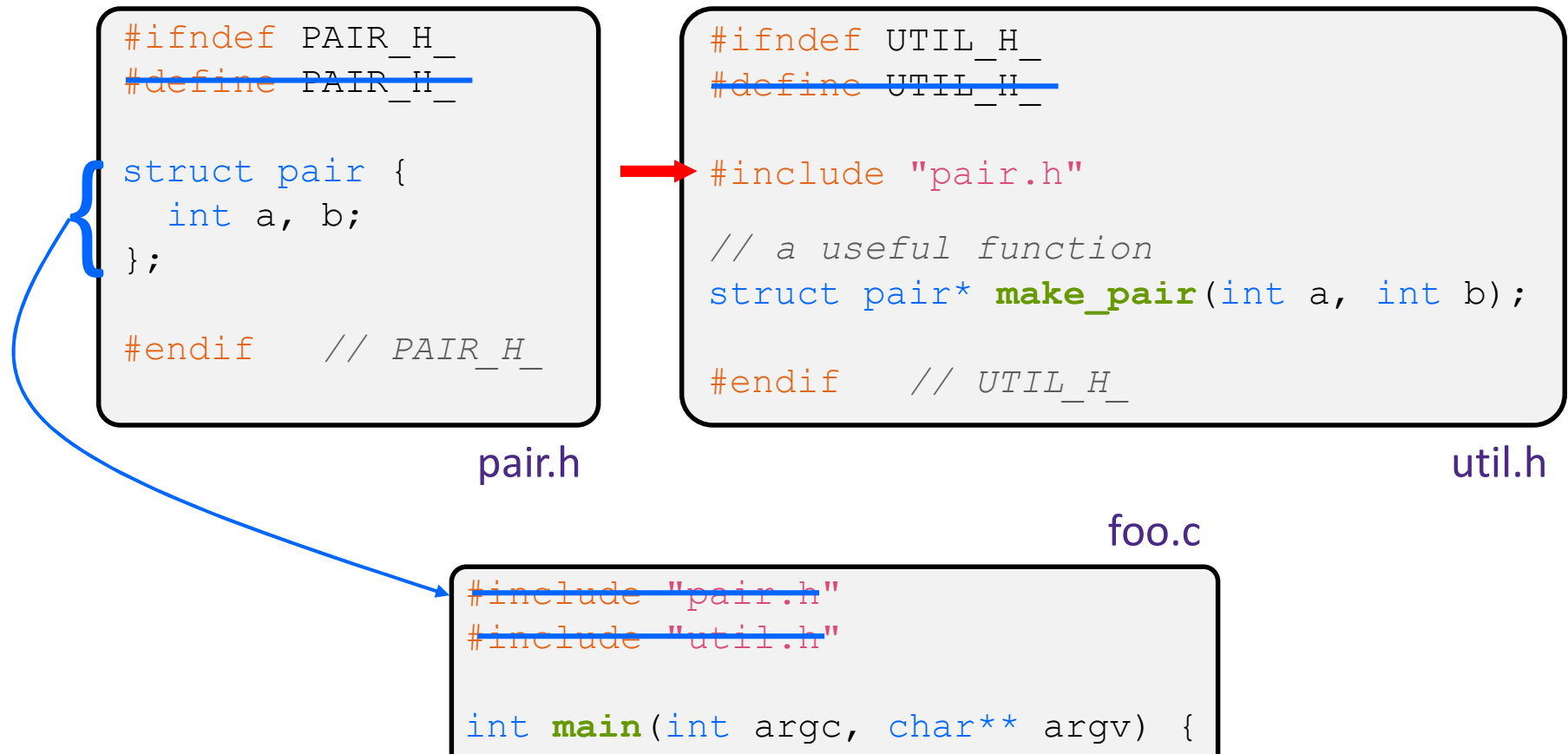
foo.c



Preprocessor Tricks: Header Guards

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

PAIR_H_	defined
UTIL_H_	defined





Preprocessor Tricks: Header Guards

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

PAIR_H_	defined
UTIL_H_	defined

```

#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* make_pair(int a, int b);

#endif // UTIL_H_
    
```

util.h

```

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

int main(int argc, char** argv) {
    // ...
}
    
```

foo.c



Preprocessor Tricks: Header Guards

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

PAIR_H_	defined
UTIL_H_	defined

```
#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* make_pair(int a, int b);

#endif // UTIL_H_
```

util.h

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

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

foo.c



Preprocessor Tricks: Header Guards

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

PAIR_H_	defined
UTIL_H_	defined

```
#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* make_pair(int a, int b);

#endif // UTIL_H_
```

util.h

foo.c

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

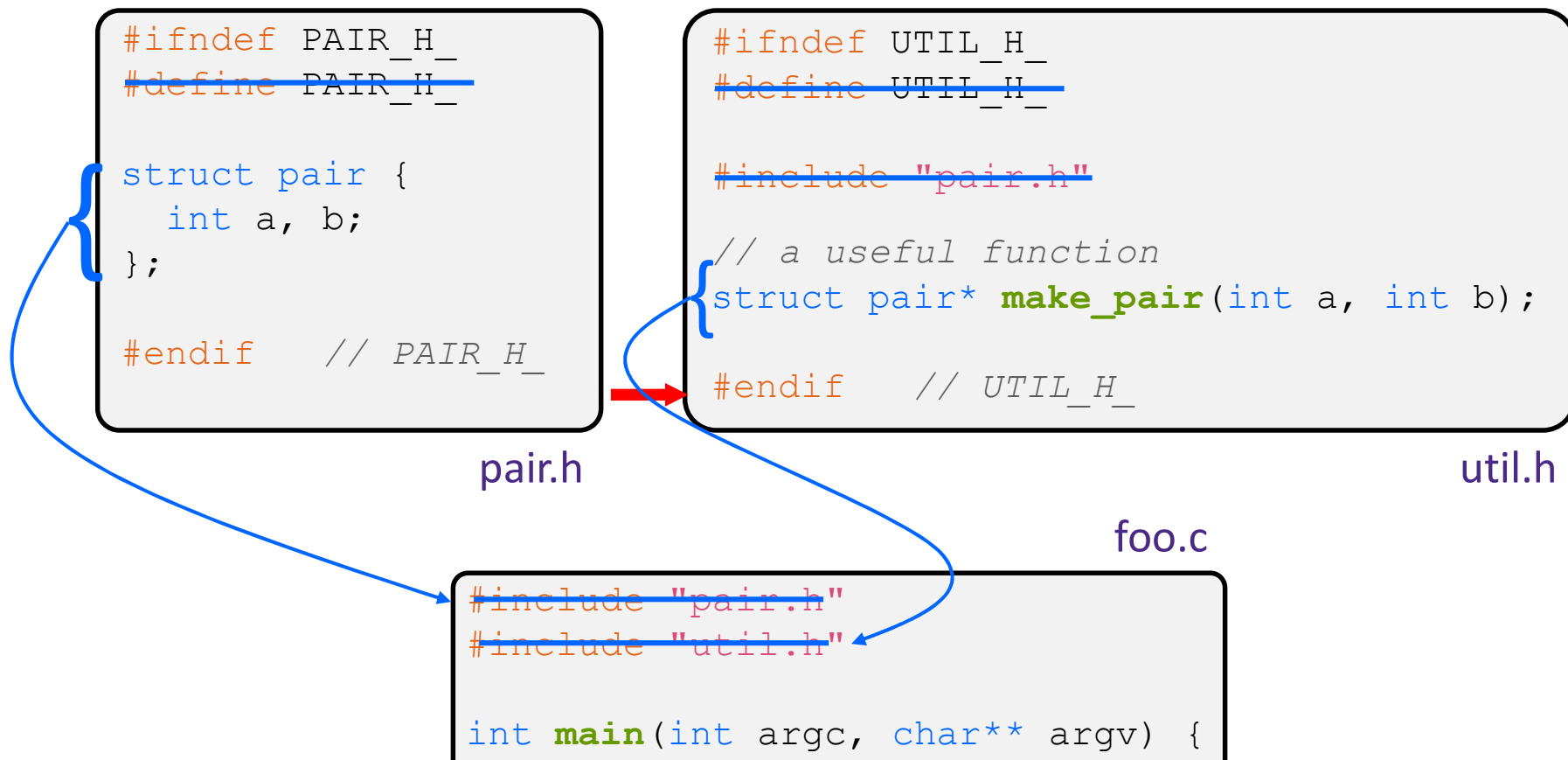
int main(int argc, char** argv) {
```



Preprocessor Tricks: Header Guards

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

PAIR_H_	defined
UTIL_H_	defined





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

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()`