

Data Structures and Modules

CSE 333 Autumn 2025

Instructors: Naomi Alterman, Chris Thachuk

Teaching Assistants:

Ann Baturytski	Derek de Leuw	Blake Diaz
Rishabh Jain	Chendur Jel Jayavelu	Lucas Kwan
Irene Xin Jie Lau	Nathan Li	Maya Odenheim
Advay Patil	Selim Saridede	Deeksha Vatwani
Angela Wu	Jiexiao Xu	

Administrivia

- ❖ Exercise 3 was due this morning
 - Sample solution posted after class

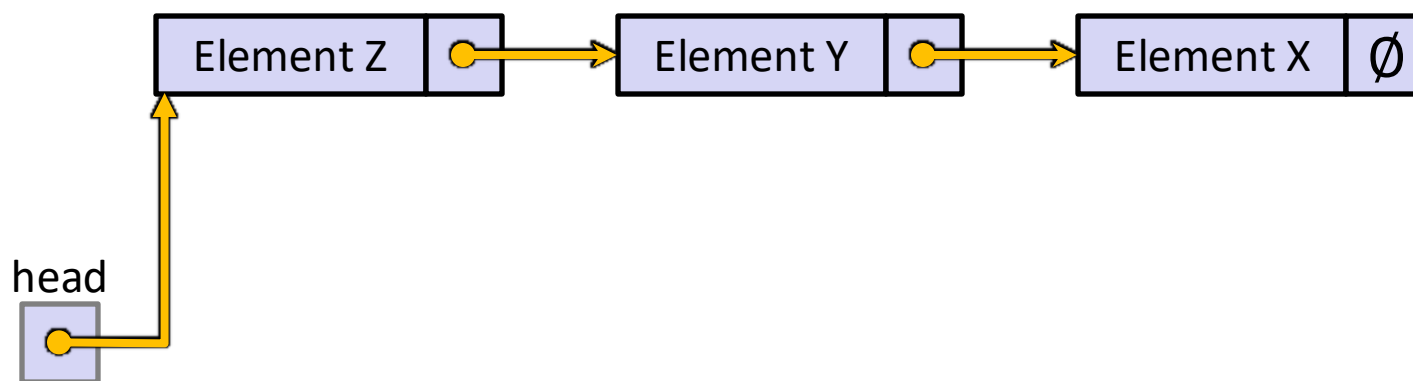
- ❖ New exercise out later today, due Monday 10 am
 - A multi-file program.

Lecture Outline

- ❖ **Implementing Data Structures in C**
- ❖ Multi-file C Programs
- ❖ Header Guards and Preprocessor Tricks
- ❖ Visibility of Symbols
 - `extern, static`

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`

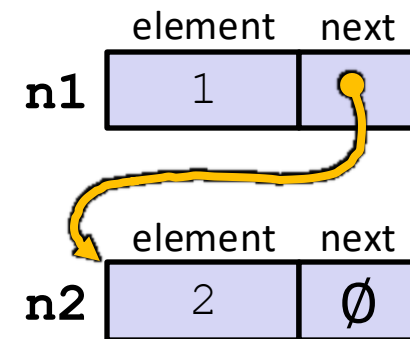
```
#include <stdio.h>

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 EXIT_SUCCESS;
}
```

Need to use `struct node_st` here. Node not defined until after end of `typedef`.



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 EXIT_SUCCESS;
}
```

(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;
}

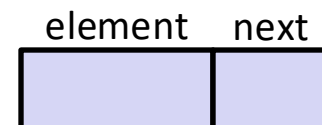
int main(int argc, char** argv) {
    Node* list = NULL;
    list = Push(list, 1);
    list = Push(list, 2);
    return EXIT_SUCCESS;
}
```

(main) list 

(Push) head 

(Push) e 

(Push) n 



push_list.c

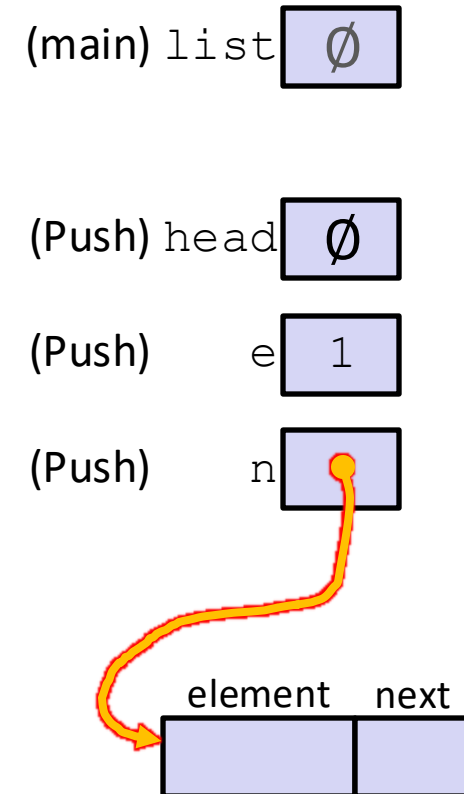
Push Onto List

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 EXIT_SUCCESS;
}
```



push_list.c

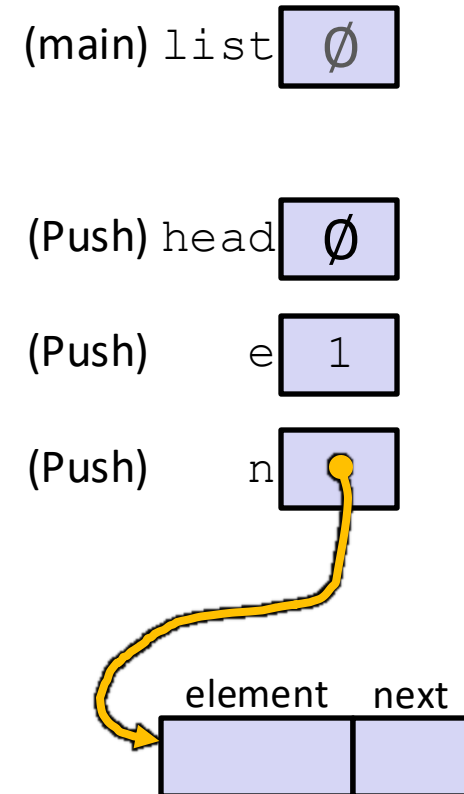
Push Onto List

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 EXIT_SUCCESS;
}
```



push_list.c

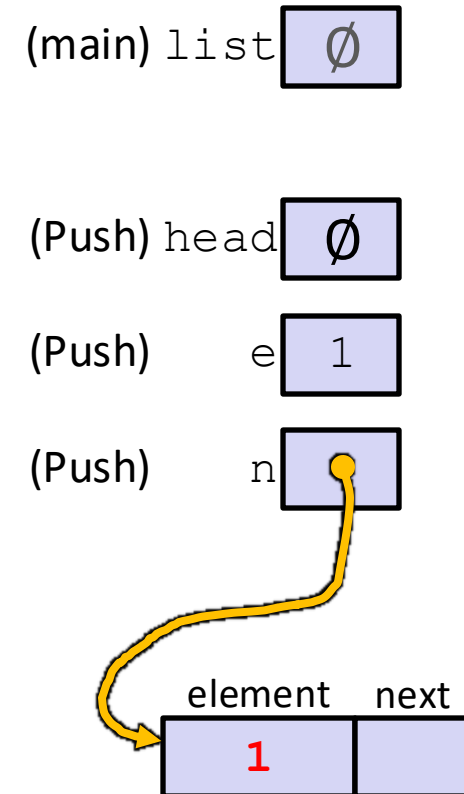
Push Onto List

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 EXIT_SUCCESS;
}
```



push_list.c

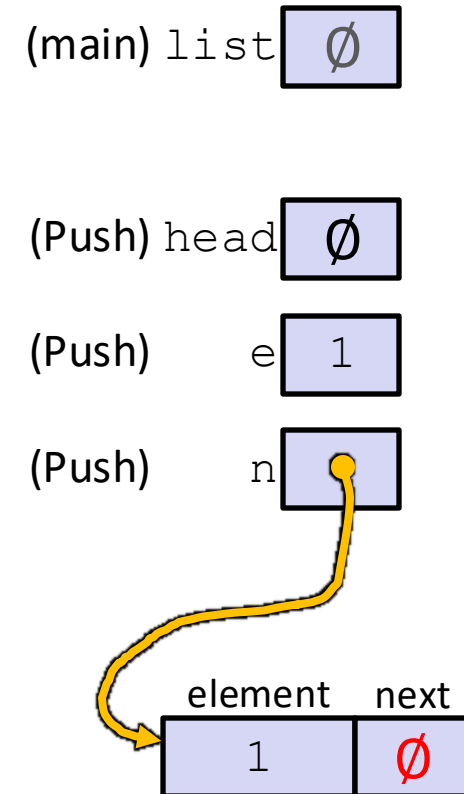
Push Onto List

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 EXIT_SUCCESS;
}
```



push_list.c

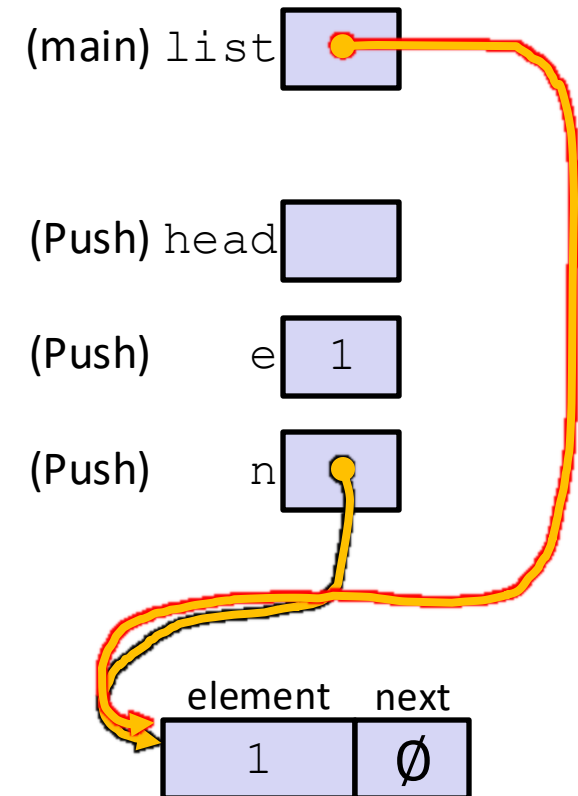
Push Onto List

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 EXIT_SUCCESS;
}
```



push_list.c

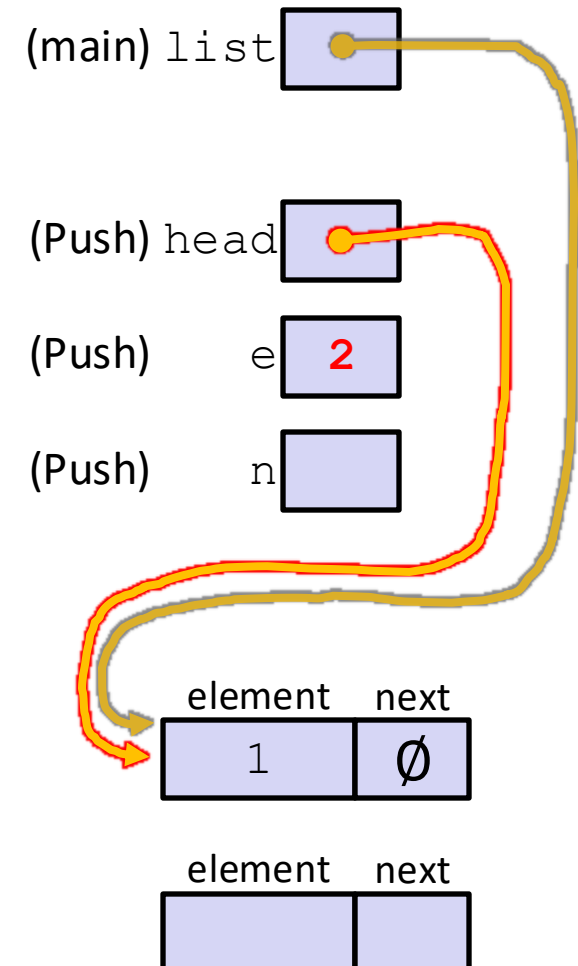
Push Onto List

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 EXIT_SUCCESS;
}
```



push_list.c

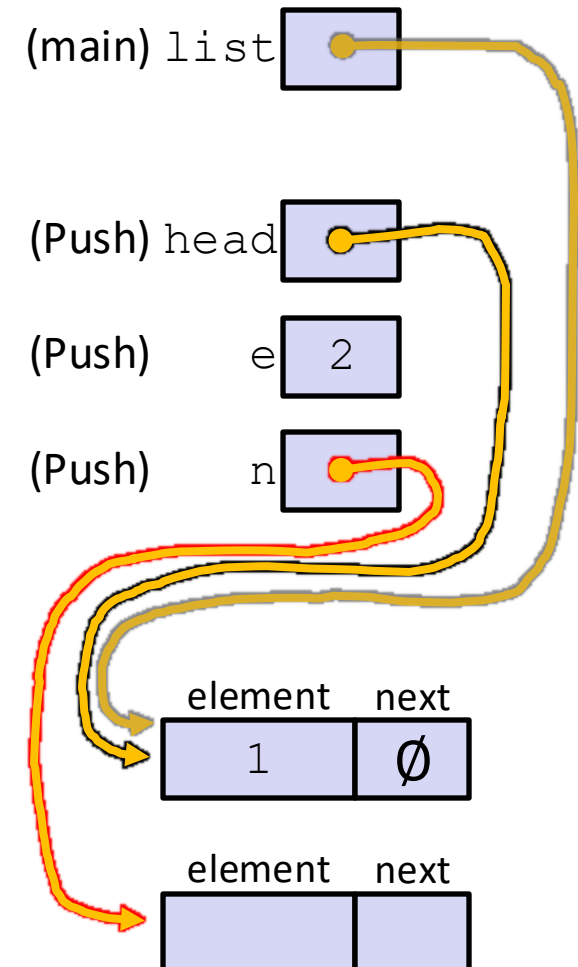
Push Onto List

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 EXIT_SUCCESS;
}
```



push_list.c

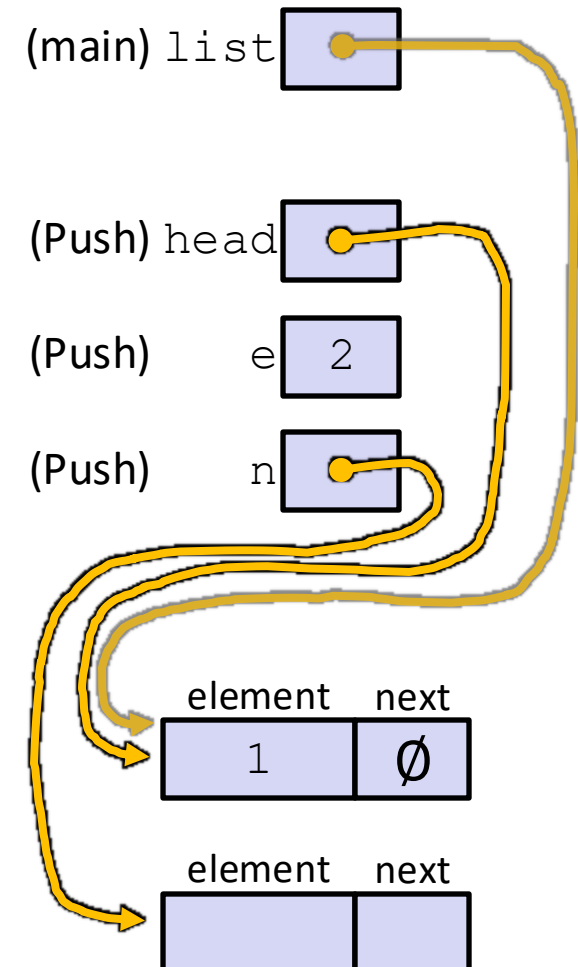
Push Onto List

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 EXIT_SUCCESS;
}
```



push_list.c

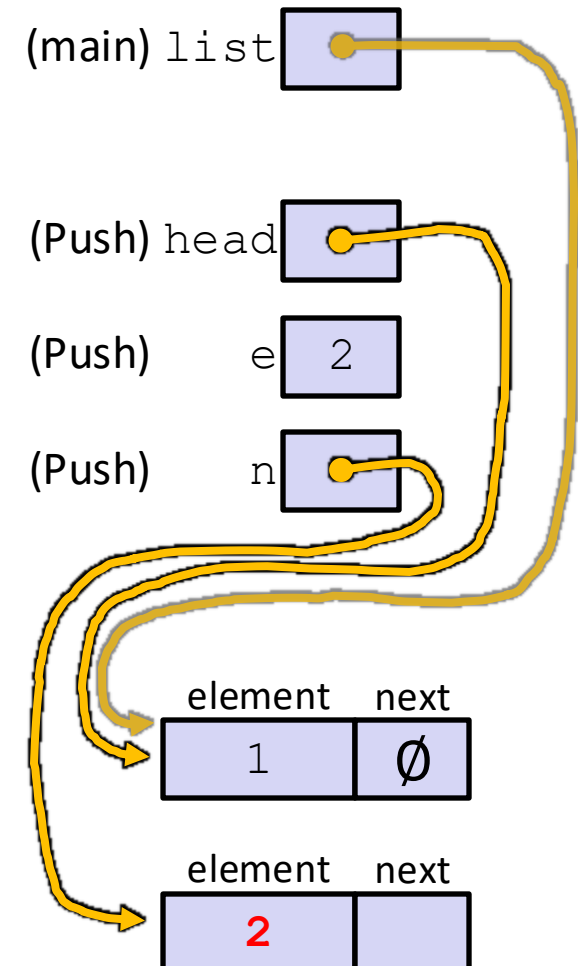
Push Onto List

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 EXIT_SUCCESS;
}
```



push_list.c

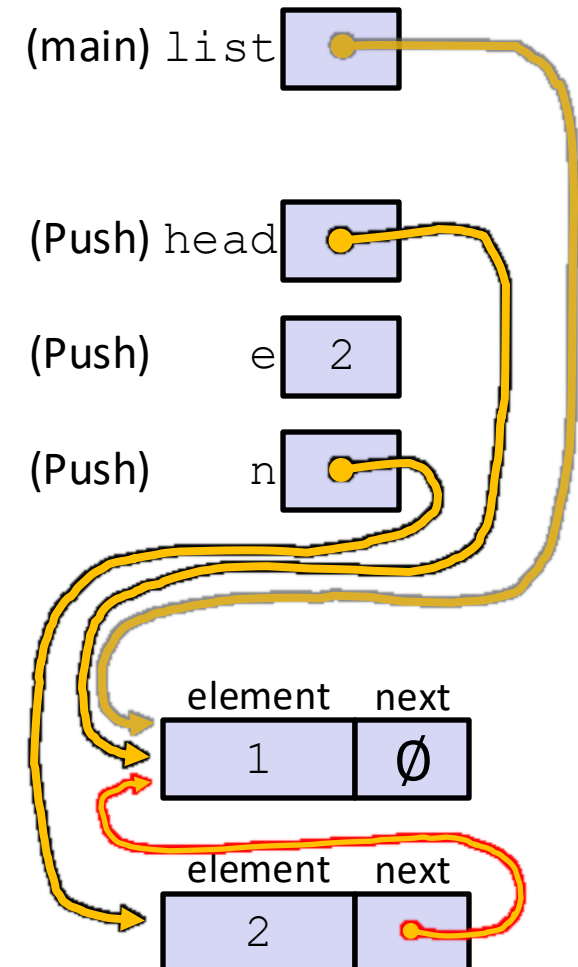
Push Onto List

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 EXIT_SUCCESS;
}
```



push_list.c

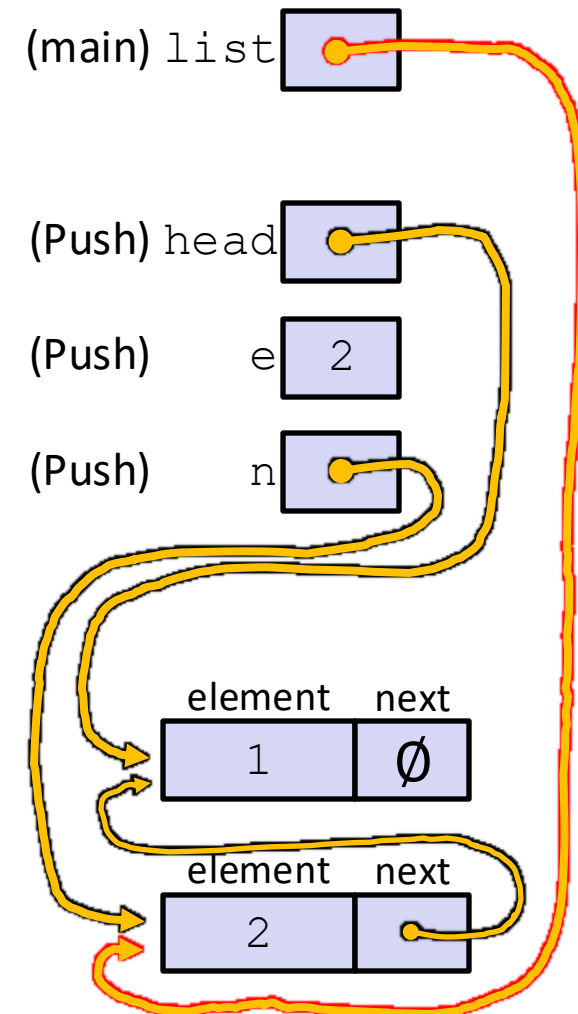
Push Onto List

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 EXIT_SUCCESS;
}
```



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;
}

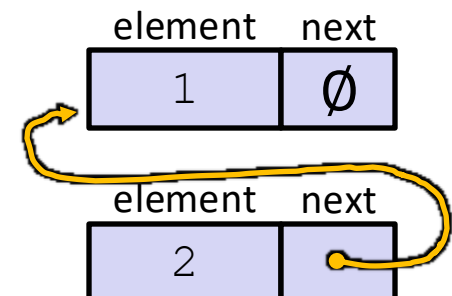
int main(int argc, char** argv) {
    Node* list = NULL;
    list = Push(list, 1);
    list = Push(list, 2);
    return EXIT_SUCCESS;
}
```

push_list.c

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

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

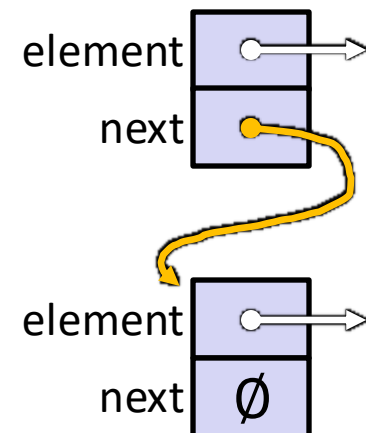
```
bash$ valgrind --leak-
check=full ./push_list
```



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;  
}
```



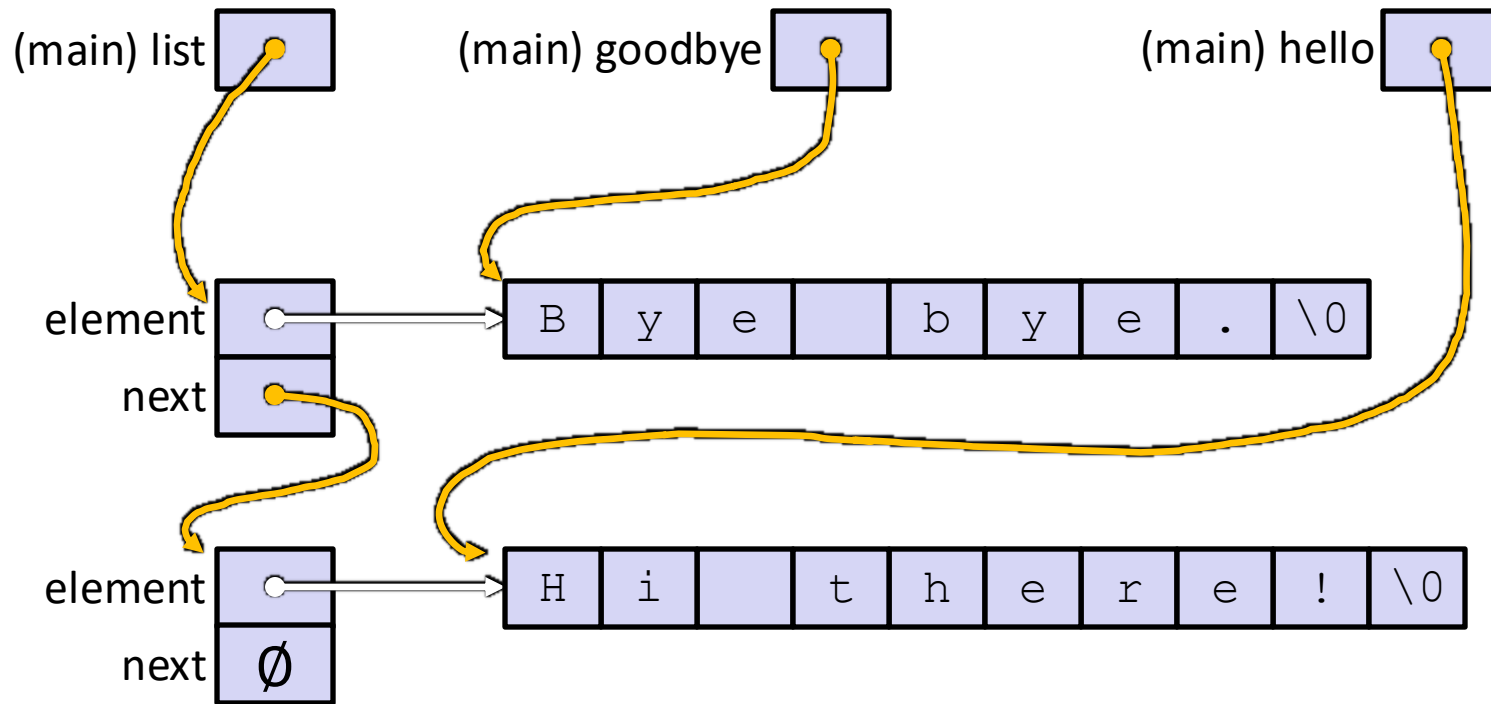
Using a Generic Linked List

- ❖ 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 EXIT_SUCCESS;  
}
```

manual_list_void.c

Resulting Memory Diagram



Lecture Outline

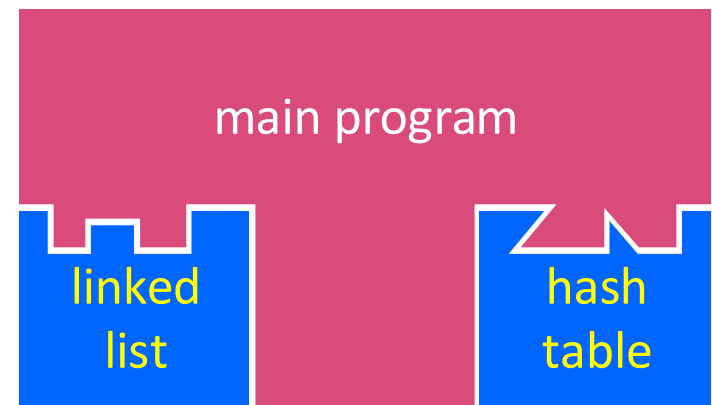
- ❖ Implementing Data Structures in C
- ❖ **Multi-file C Programs**
- ❖ Header Guards and Preprocessor Tricks
- ❖ Visibility of Symbols
 - `extern, static`

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
 - There are `<system-defined>` and "programmer-defined" headers
- ❖ **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_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 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
- ❖ Multi-file C Programs
- ❖ **Header Guards and Preprocessor Tricks**
- ❖ 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`
 - 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

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

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`

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

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

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

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

```
typedef long long int verylong;
```

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

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

```
$ 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

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

```
$ 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

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

```
$ 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

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

```
$ 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
}
```

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

```
$ 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;  
}
```

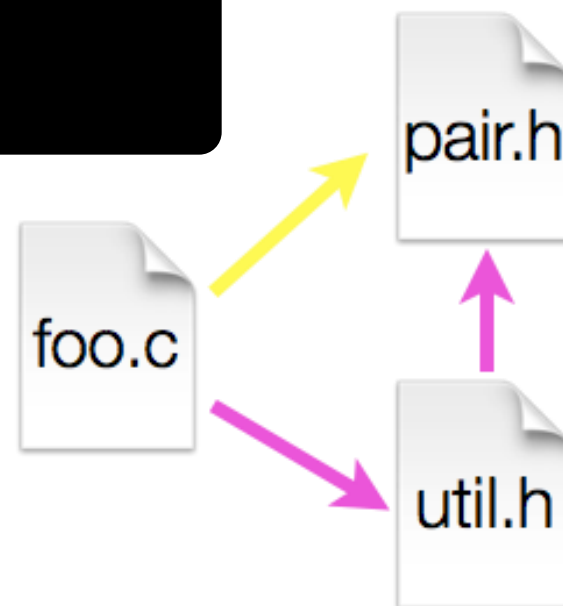
foo.c

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

- ❖ 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) {
```

Lecture Outline

- ❖ Implementing Data Structures in C
- ❖ Multi-file C Programs
- ❖ Header Guards and Preprocessor Tricks
- ❖ **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

```
#include <stdio.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

```
#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("(b): counter = %d\n",
           counter);
}
```

bar.c

Internal Linkage

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

```
#include <stdio.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

```
#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("(b): counter = %d\n",
           counter);
}
```

bar.c

Function Visibility

```
// 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 call it.  
int bar(int x) {  
    return 2*foo(x);  
}
```

bar.c

```
#include <stdio.h>  
  
extern int bar(int x); // "extern" is default, usually omit  
                        // should be in .h file, but effect is same  
  
int main(int argc, char** argv) {  
    printf("%d\n", bar(5));  
    return EXIT_SUCCESS;  
}
```

main.c

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

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()`