Data Structures and Modules

CSE 333

Instructor: Alex Sanchez-Stern

Teaching Assistants:

Audrey Seo

Deeksha Vatwani

Derek de Leuw

Katie Gilchrist

Administrivia

- Exercise 3 was due this morning
- Exercise 5 out now, due Monday
  - There is no exercise 4 this quarter

- Sections tomorrow:
  - visualizing, diagraming, & debugging memory
  - gdb and valgrind, how to use them effectively, and how to interpret their output

Crucial debugging tools! If your program isn't doing what you expect, use these ***before*** asking for help

Administrivia (2)

- No class on Friday! 🎉

Administrivia (2)

- Some issues in HW0 submissions!
  - 5 submissions missing README.md
  - 16 submissions with tagging issues
- If you can't figure out what you did wrong, come to office hours!

HW1 Reminders

- HW1 is due 1 week from tomorrow
- Reminders about hw1:
  - You ***may not*** modify interfaces (`.h` files)
  - But ***do*** read the interfaces while you're writing code(!)
  - Suggestion: look at `example_program_{ll|ht}.c` for typical usage of lists and hash tables

- Watch that `HashTable.c` doesn't violate the modularity of `LinkedList.h` (i.e., don't access private/hidden implementation details of linked lists)

Debugging Hints

- Beware pointers to local (stack) variables (0x7fff… addresses)
  - Symptom: variables appear to spontaneously change values for no reason
- Keep track of types of things – draw memory diagrams
  - Is this variable a Thing, Thing*, Thing**, typedefed Thing*?
- GDB
  - Use a debugger (*e.g.* `gdb`) if you're getting segfaults – fix reality!
  - Write and run ***small*** tests to track down problems (don't kill lots of time trying to debug large test_suite code)
  - gdb hint: What if `Verify333` fails? How can you debug it? Answer: look at the `Verify333` macro (`#define`), figure out what function it calls on failure, and put a breakpoint there

Debugging Hints

- Keep track of these things when debugging:
  - Symptom: what is happening that lets you know there is a bug?
    - Segfault, wrong value printed, etc
  - Local cause hypothesis: What's something nearby that could be wrong?
    - Variable has the wrong value, code took the wrong branch, etc
  - Test: What is something you can do to tell if your hypothesis is right?
    - Check a variable with gdb? Print something? Add an assert?

Debugging Hints

- If the test indicates your hypothesis is wrong, make a new one.
- If the test indicates your hypothesis is right:
  - Restart the process
  - Your old cause is now your symptom
  - Look for a new cause slightly farther back

Git/Submission Hints

- Don’t push .o and executable files or other build products
  - Clutter, makes it harder to do clean rebuilds, not portable, etc.
- Don’t use git as a file transfer program (don’t edit on one machine, commit/push/pull to another, compile, and repeat every few minutes)
  - That’s what scp is for
- Advice: use git add/commit/push often to save your work
  - Not one massive commit at the end!
- Late days: don’t tag `hw1-final` until you are really ready (then check your work – clone repo – and re-read assignment to be sure you didn’t miss anything!)

Yet more hw1 hints

- Extra Credit: if you add unit tests, put them in a new file and adjust the Makefile and be sure to tag the extra credit part with `hw1-bonus`

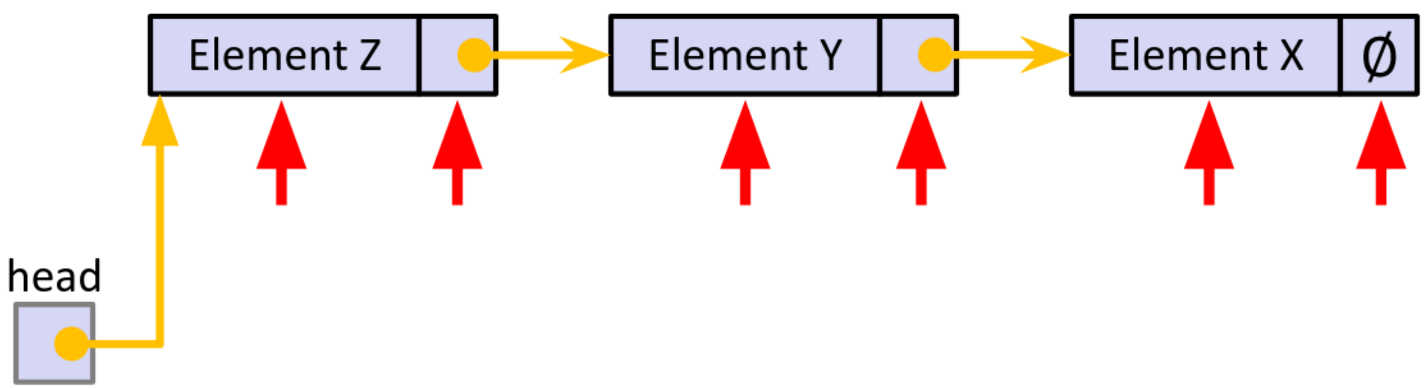
- Remember: the only supported systems for the class are the Current Allen School Linux machines using gcc 11. You should use these systems
  - The projects you build ***must*** work there!

Lecture Outline

- **Implementing Data Structures in C**
- Multi-file C Programs
  - C Preprocessor Intro
- Header Guards
- 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



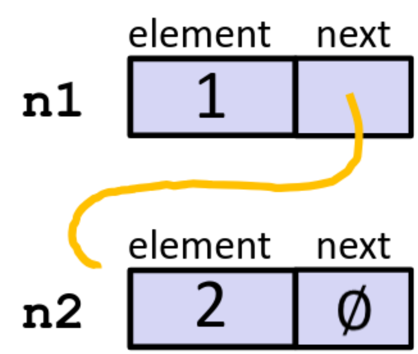
Arrow points to *next* instruction.

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`.



manual_list.c

Next Step: Abstraction

- This works, but lacks abstraction
- The `main` function has to know the implementation of the linked list to use it

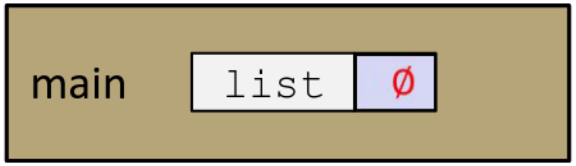
Push Onto List

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

Stack

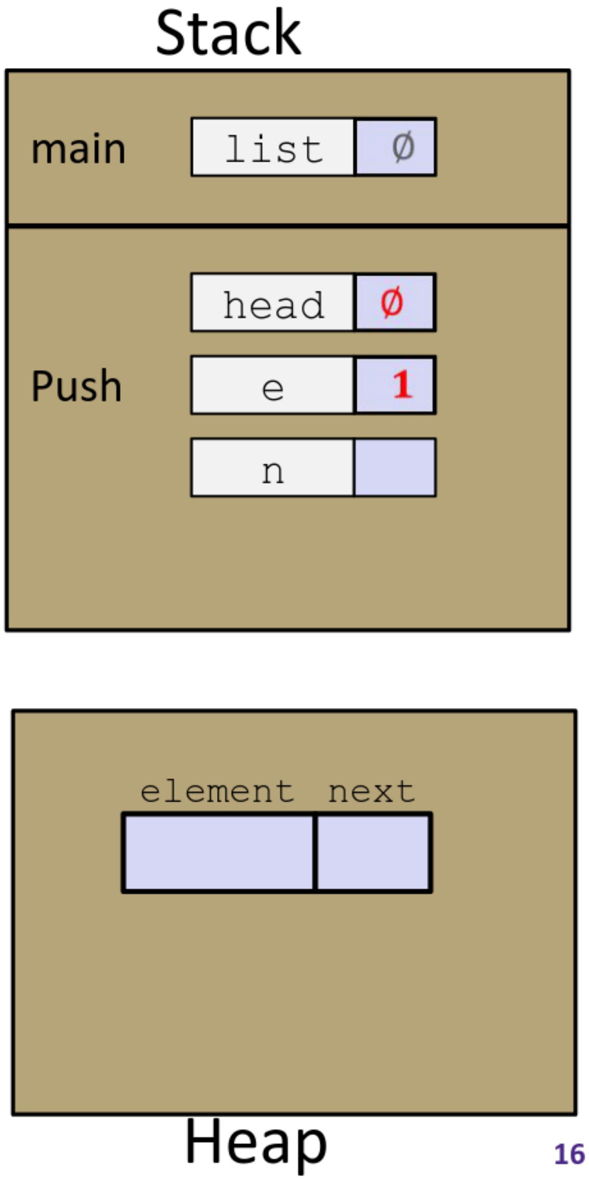


Push Onto List

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

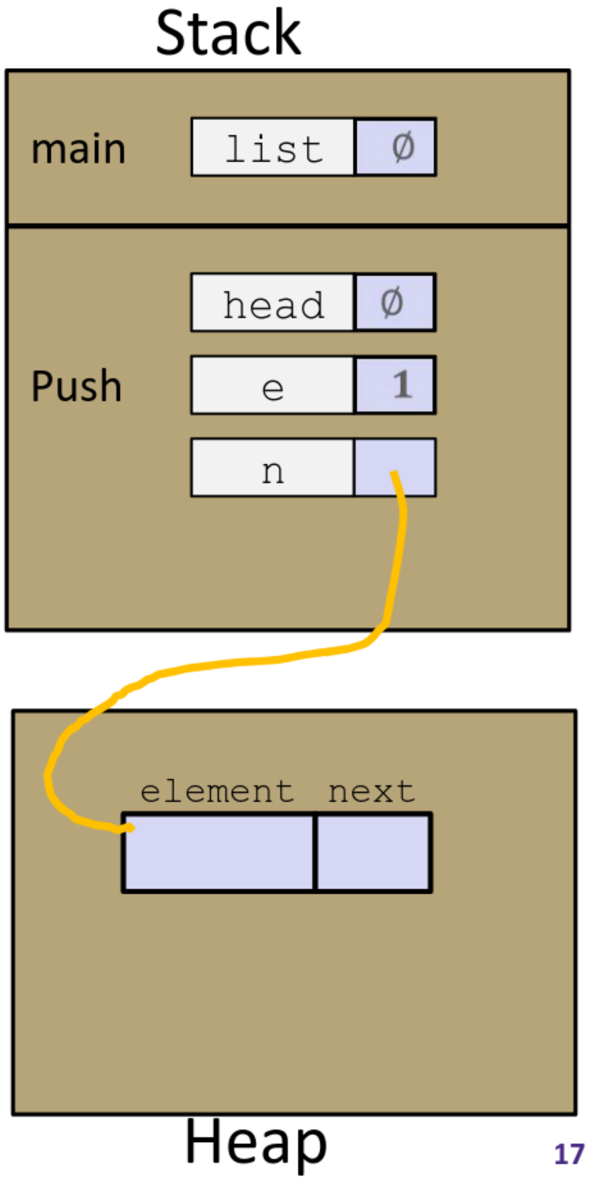


Arrow points to *next* instruction.

Push Onto List

```
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

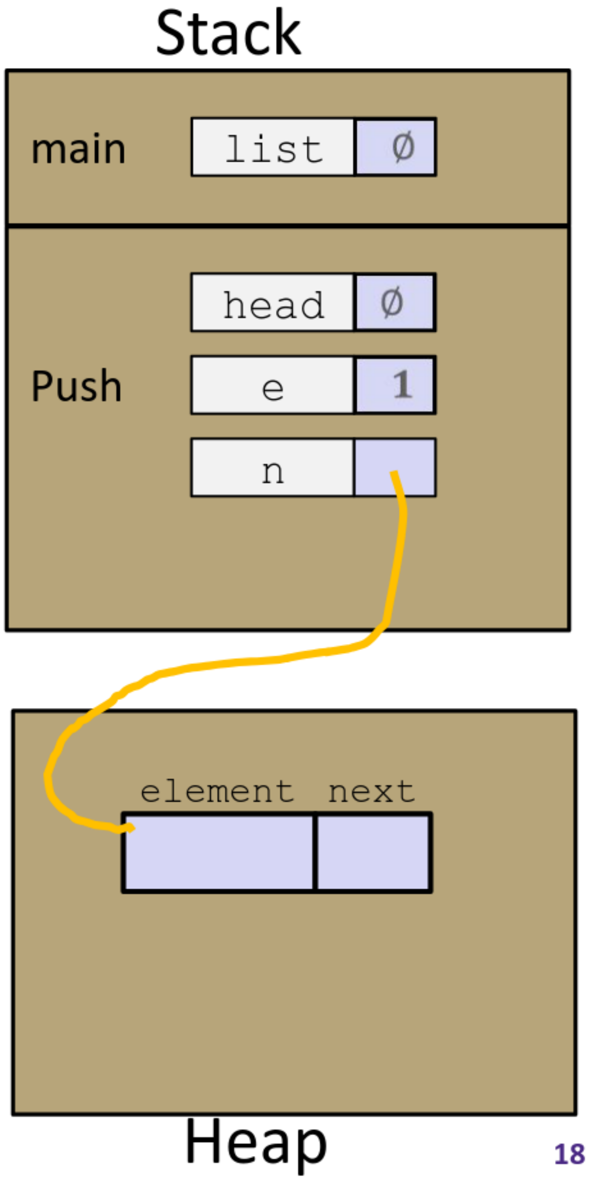


Push Onto List

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

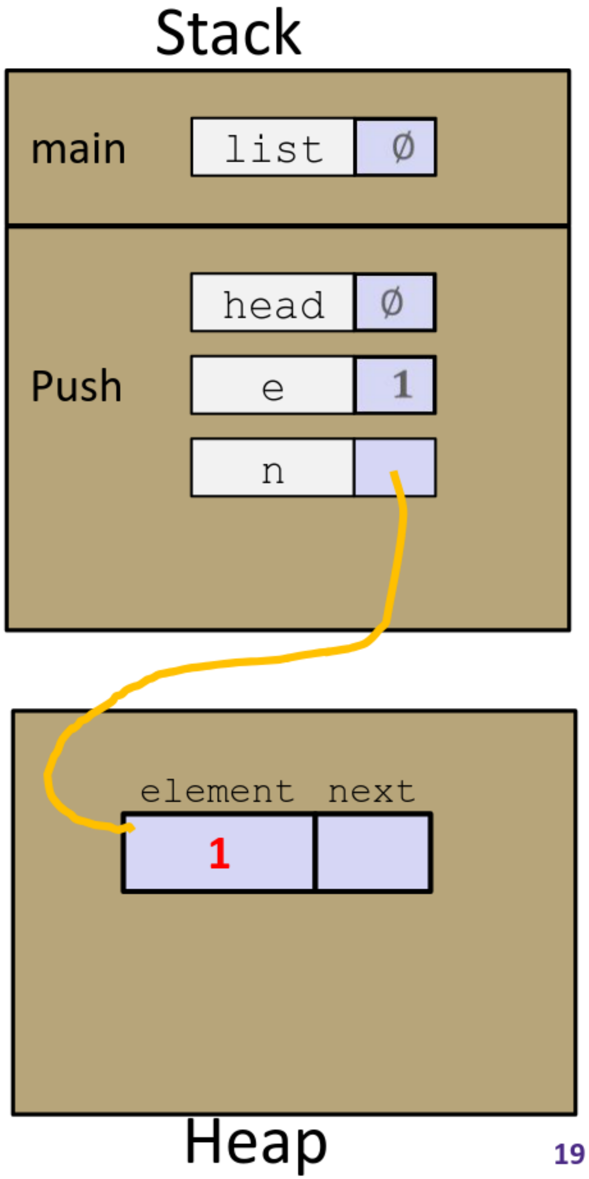


Push Onto List

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

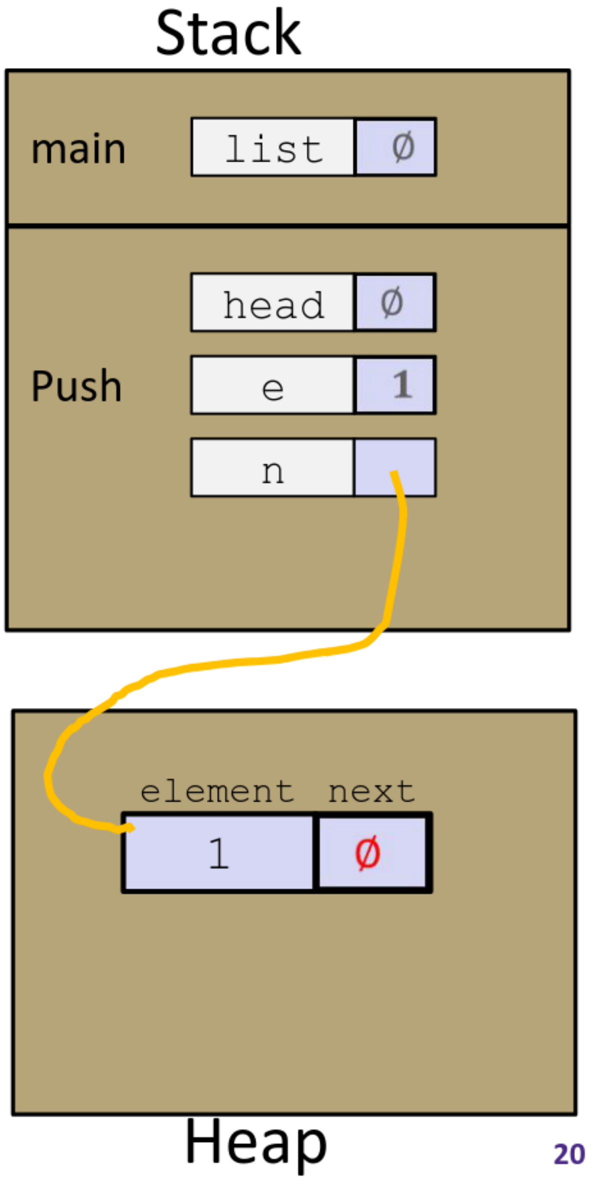


Arrow points to *next* instruction.

Push Onto List

```
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

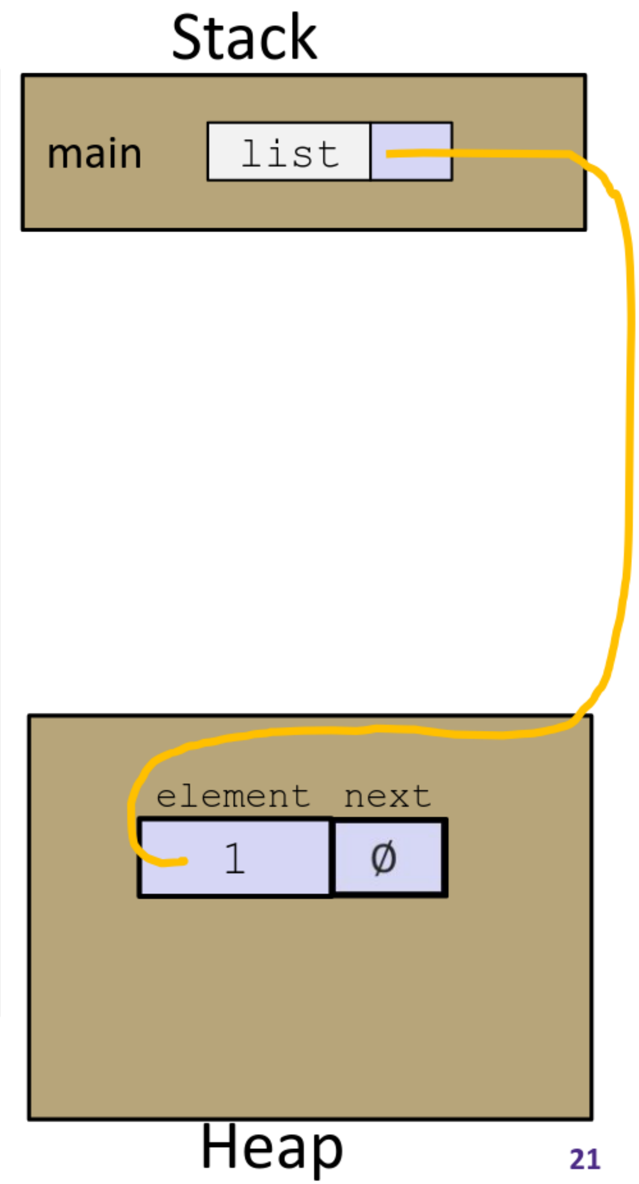


Push Onto List

Arrow points to *next* instruction.

```c
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

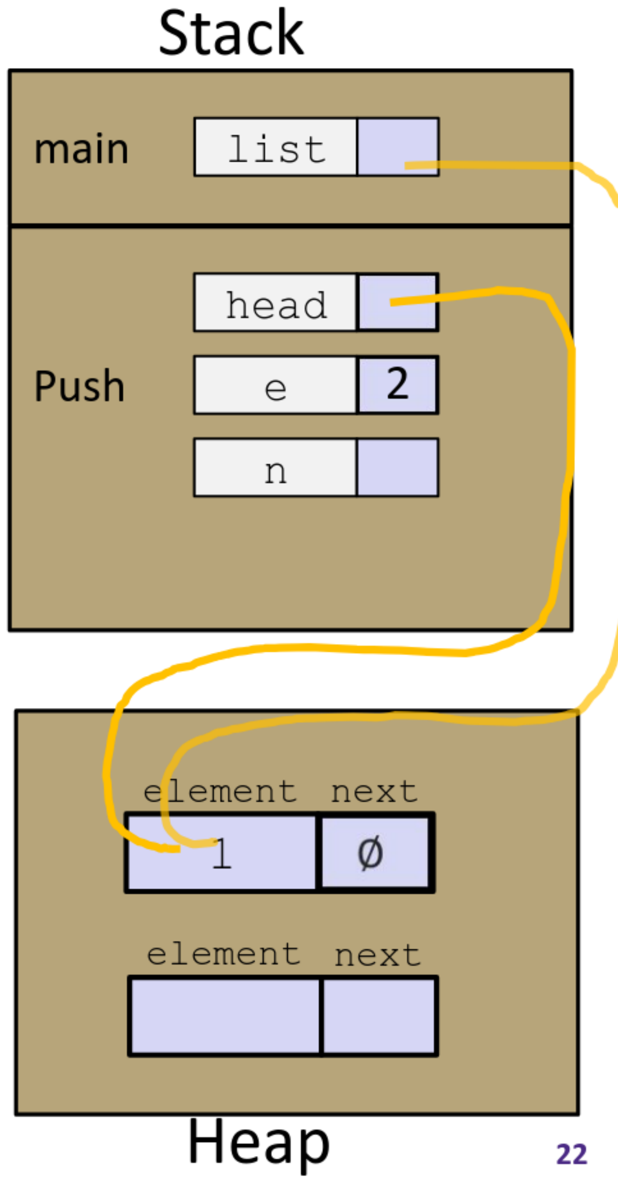


Arrow points to *next* instruction.

Push Onto List

```c
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

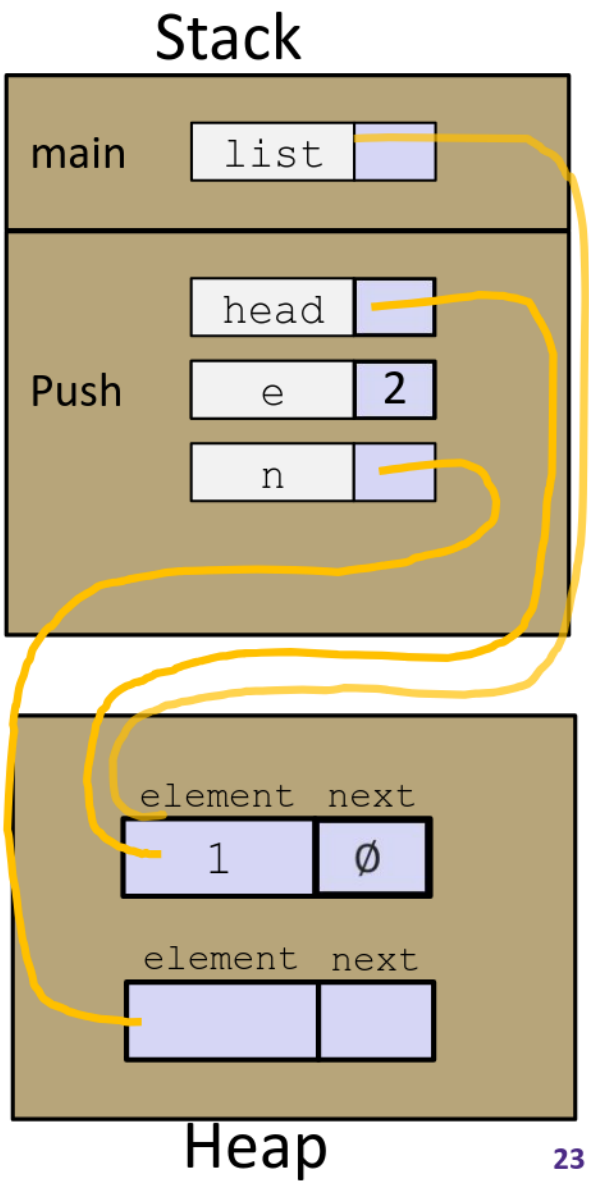


Push Onto List

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

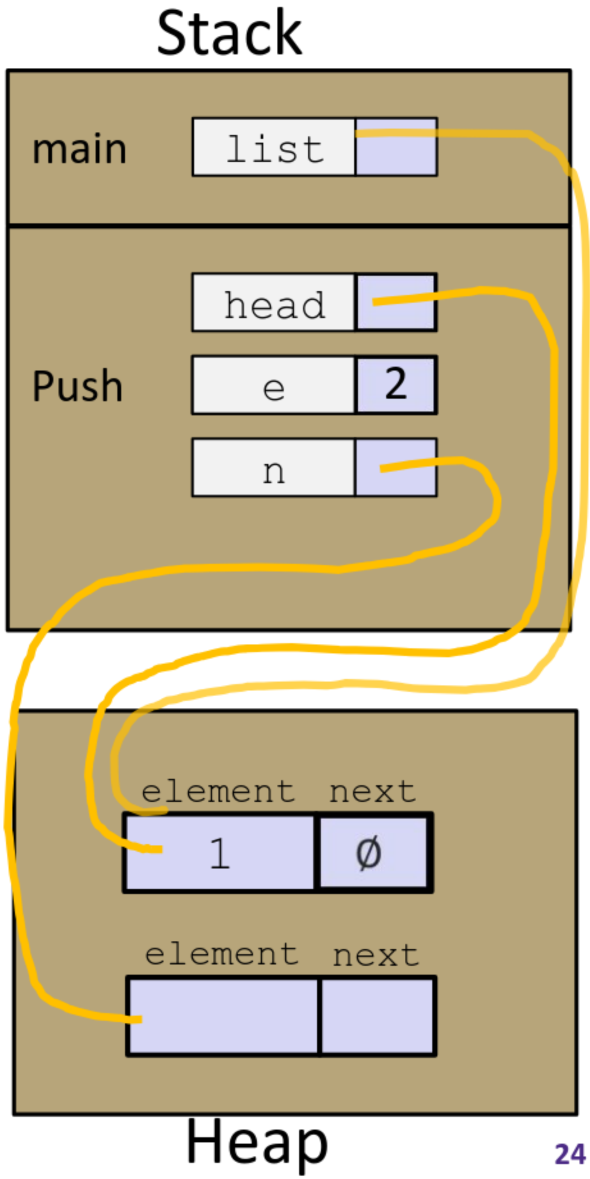


Push Onto List

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

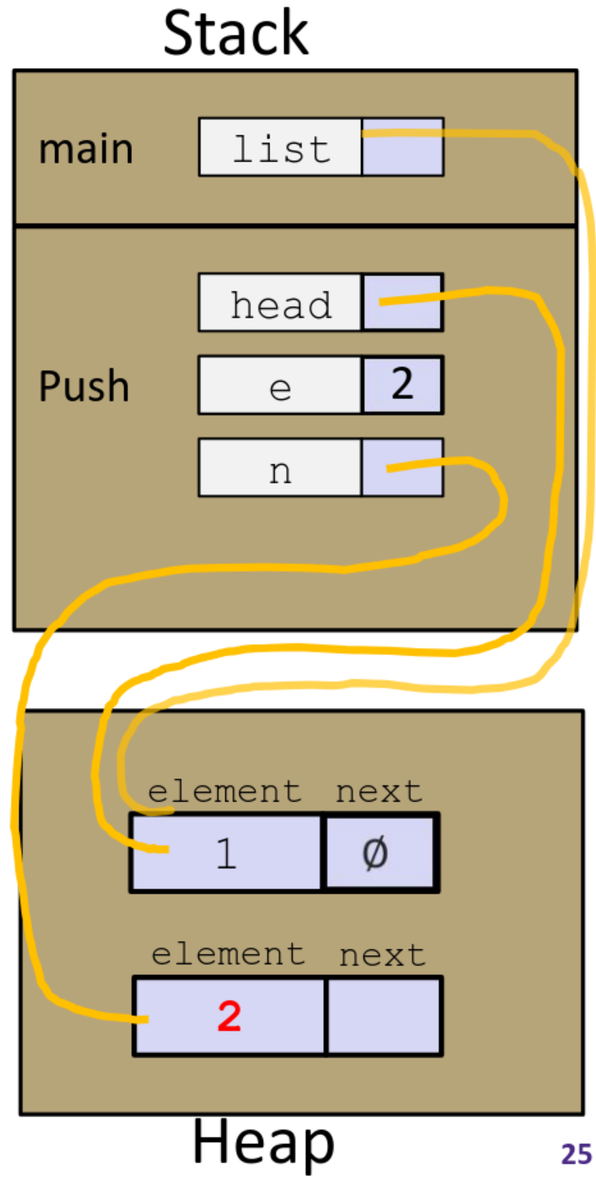


Push Onto List

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

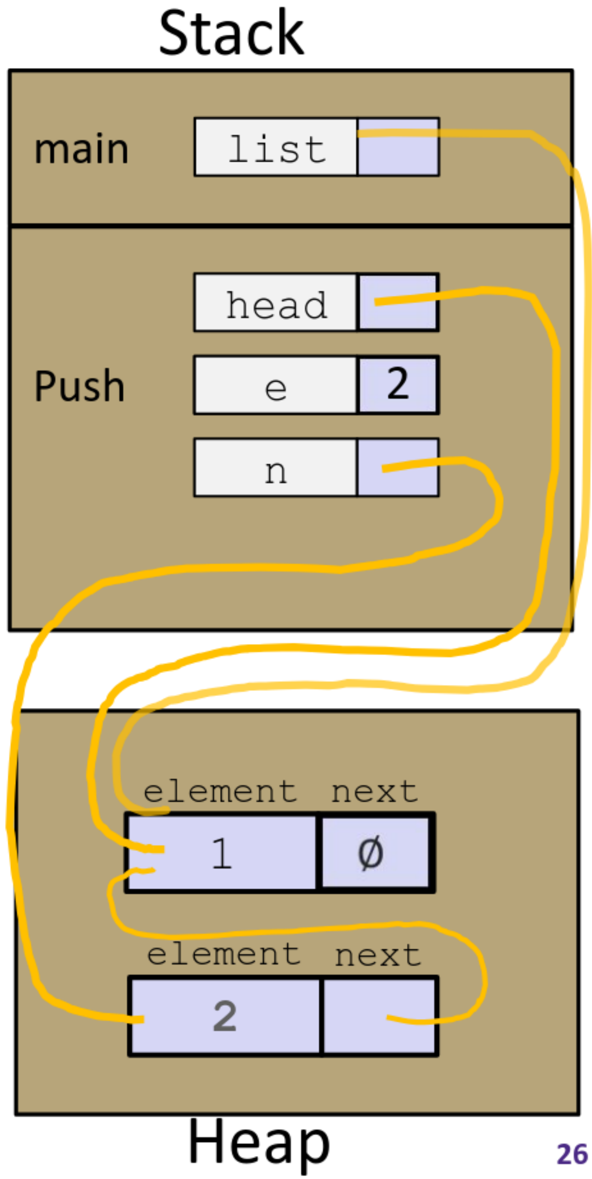


Arrow points to *next* instruction.

Push Onto List

```
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

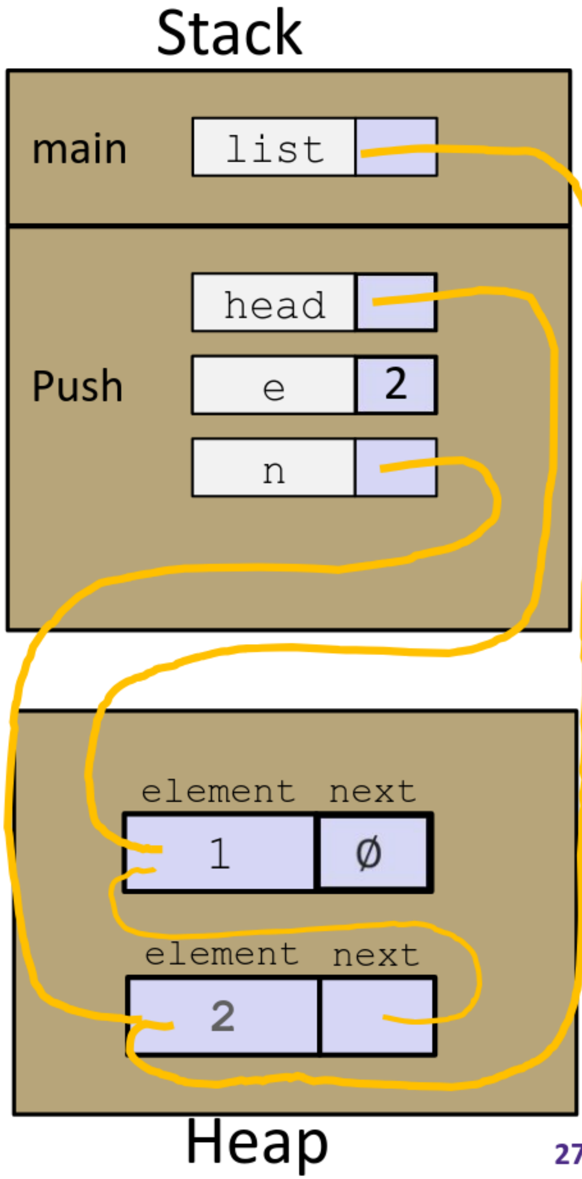


Push Onto List

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

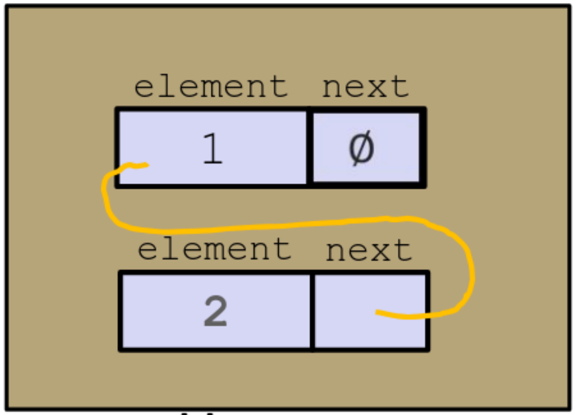
Arrow points to *next* instruction.

```
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
```



Heap

Next Step: More Abstraction

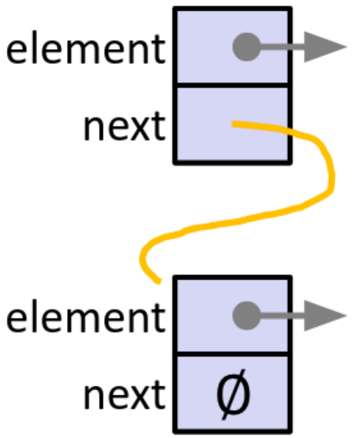
- `main` no longer needs to know the implementation of the linked list, `Push` takes care of that.

- But the linked list needs to know what type of data `main` wants to store.

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;
}
```



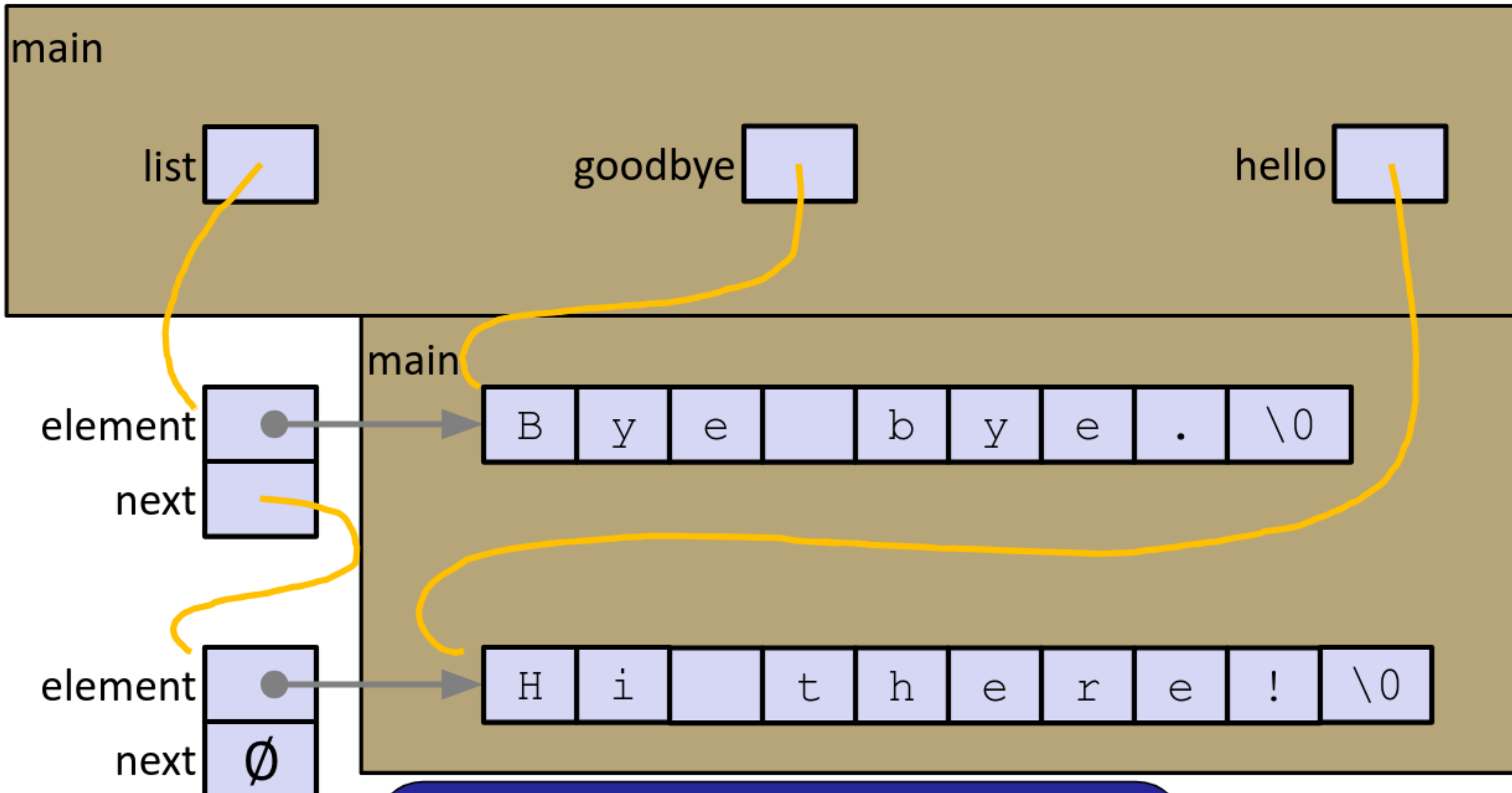
Using a Generic Linked List

- 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



Important: Caller must now allocate persistent storage for elements

Even More Abstraction

- main still has access to all the details of the linked list

- Let's create a linked list *module*

Lecture Outline

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

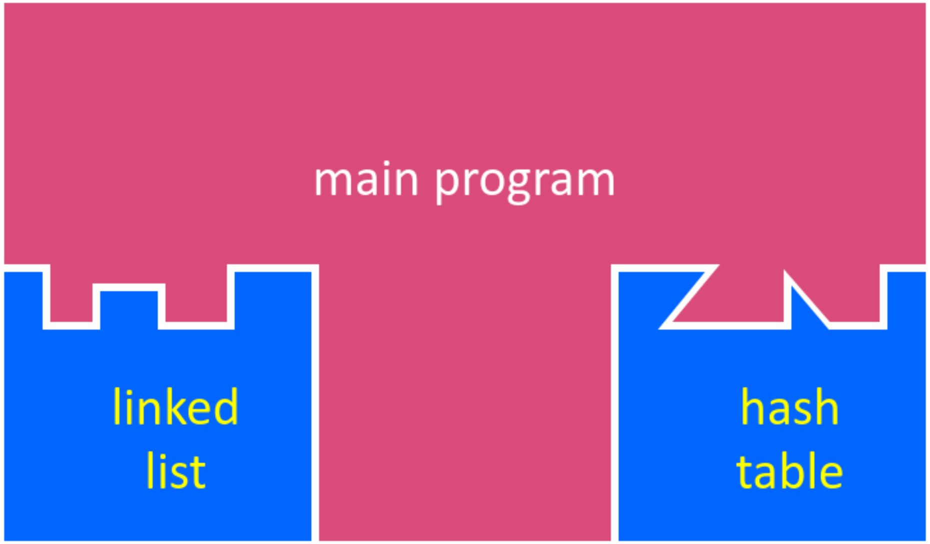
Modular Programming in C

- A module is a self-contained piece of an overall program
  - Has externally visible functions that customers can invoke
  - Has externally visible `typedef`s, and perhaps global variables, that customers can use
  - May have internal functions, `typedef`s, or global variables that customers should *not* look at
  - In object-oriented languages, classes act as modules

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

"Modularity"

- The degree to which components of a system can be separated and recombined
  - "Loose coupling" and "separation of concerns"
  - Good modules can be developed independently
  - Good modules can be re-used in different projects



C Header Files

- The main mechanism of modularity is header files.
- Header: a C file which declares the module interface
  - Generally has a filename `.h` extension
- Main Idea:
  - Every `name.c` is intended to be a module that has a `name.h`
  - `name.h` Holds the variables, types, and function prototype declarations that make up the interface to a module
  - Other modules wanting to use `name` then `#include name.h`
    - Makes the module **interface** visible to the using module
    - Using module should assume as little as possible about the implementation in `name.c`

C to Machine Code

```c
void sumstore(int x, int y,
              int* dest) {
  *dest = x + y;
}
```

C source file (sumstore.c)

C compiler (gcc -S)

```
sumstore:
        addl    %edi, %esi
        movl    %esi, (%rdx)
        ret
```

Assembly file (sumstore.s)

Assembler (gcc -c or as)

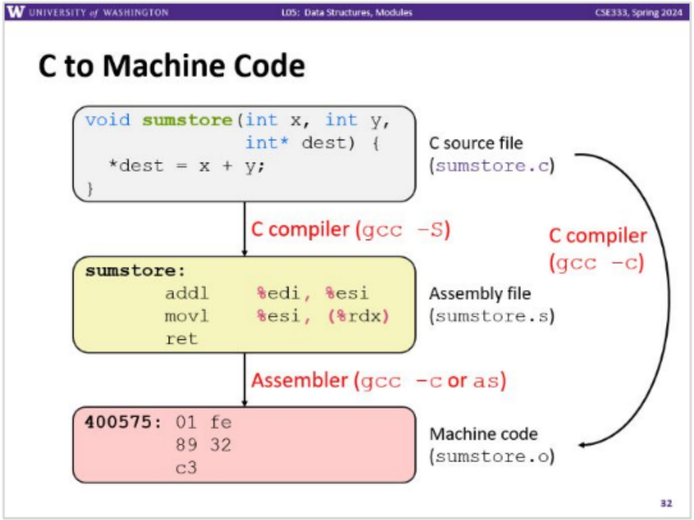
```
400575: 01 fe
        89 32
        c3
```

Machine code (sumstore.o)

C compiler (gcc -c)

Including C Header Files

- How do these `#include`s work? Recall that "compilation" actually comprises multiple steps!



- The first step is the C preprocessor (cpp)
  - It transforms your source code before the compiler runs
  - Simple copy-and-replace text processor(!) with a memory
  - `#include` just copies the text of the included file into the current one
  - Preprocessor directives generally take the form `#directive`
    - *e.g.* `#include "LinkedList.h"`
    - *e.g.* `#define PI 3.1415`

(Ab)uses of C Preprocessor In the Wild

- Google Pack used the C Preprocessor for localization
- Amazon used to render their main webpage using the C Preprocessor

`#include` and the C Preprocessor

- Input is a C file (text) and output is still a C file (text)
- Example `#directive`s
  - e.g. `#include "LinkedList.h"` replaced by post-processed content of LinkedList.h
  - e.g. `#define PI 3.1415` defines a symbol (a string!) and replaces later occurences
  - Several others that we'll see soon …
- Run on your behalf by gcc during compilation
  - `#include <foo.h>` looks in system (library) directories
  - `#include "foo.h"` looks first in current directory, then system

C Header/Module Conventions

- Most C projects adhere to the following rules:
  - `.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
    - Those function prototype declarations belong in the `.h` file
    - The `.c` file should `#include` its own `.h` file; the compiler will check declarations and definitions for consistency
  - ***NEVER*** `#include` a `.c` file – only `#include` `.h` files
  - `#include` all of headers you reference, even if another header (accidentally or not) includes some of them
  - A `.c` file with an associated `.h` file should be considered a single unit of code/compilation

C Preprocessor Example

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

```c
#define BAR 2 + FOO

typedef long long int verylong;
```

cpp_example.h

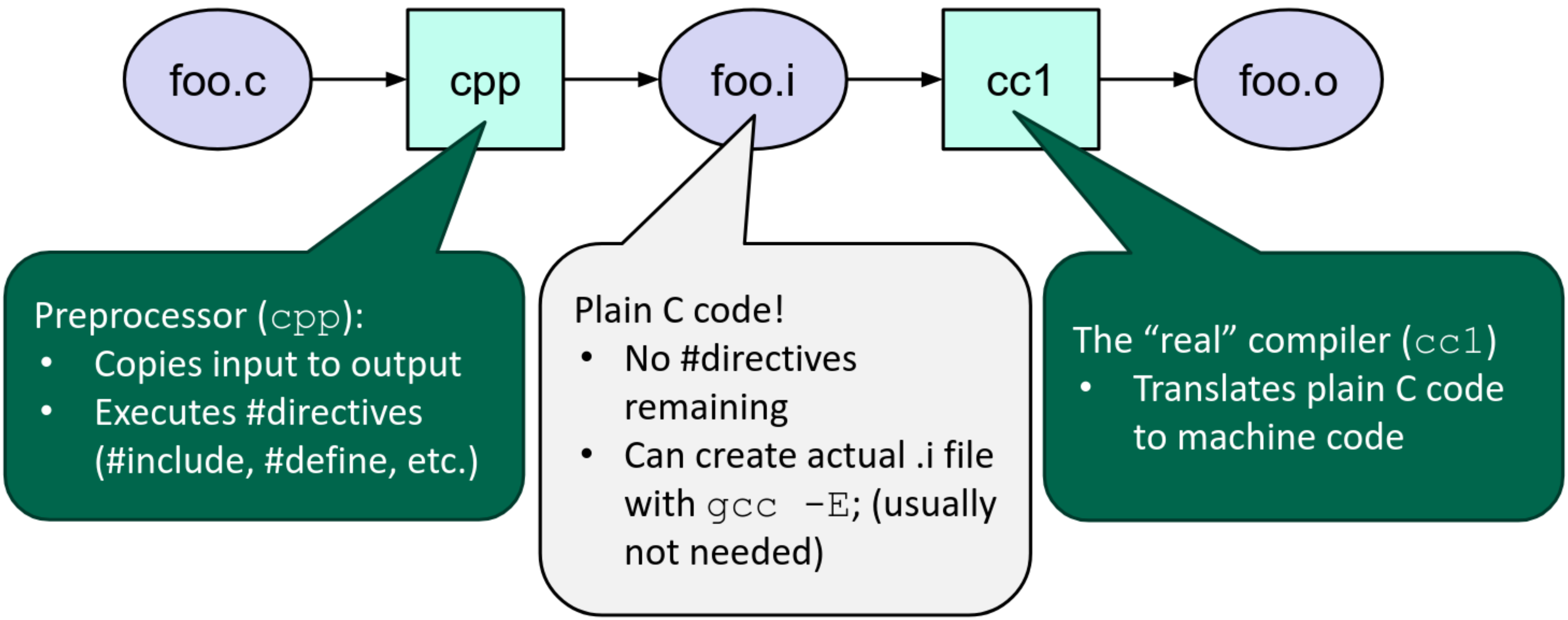
```c
#define FOO 1

#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;
}
```

What Is gcc Really Doing?

- `gcc` runs other programs that do the "real work"
- Here's what `gcc` runs to translate `foo.c` to `foo.o`
  - `gcc -c foo.c`



Program Using a Linked List

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

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

LinkedList.h

```c
#include <stdlib.h>
#include <assert.h>
#include "LinkedList.h"

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

LinkedList.c

```c
#include "LinkedList.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 EXIT_SUCCESS;
}
```

example_ll_customer.c

Compiling the Program

- Four parts:
  - 1/2) Compile `example_ll_customer.c` into an object file
  - 2/1) Compile `LinkedList.c` into an object file
  - 3) Link both object files into an executable
  - 4) Test, Debug, Rinse, Repeat

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

Where Do the Comments Go?

- 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*
  - No need to copy/paste the comment into the C file
    - Don't want two copies that can get out of sync
    - Recommended to leave "specified in <filename>.h" comment in C file code to help the reader

Where Do the Comments Go?

- If a (local) function has its prototype and implementation in same C file:
  - One school of thought: Full comment on the prototype at the top of the file, no comment (or "declared above") on code
    - 333 project code is like this
  - Another school: Prototype is for the compiler and doesn't need comment; put the comments with the code to keep them together
    - Not used in 333

Lecture Outline

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

An `#include` Problem

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

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

pair.h

```
#include "pair.h"

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

util.h

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

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

foo.c

An `#include` Problem

- 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`

Header Guards

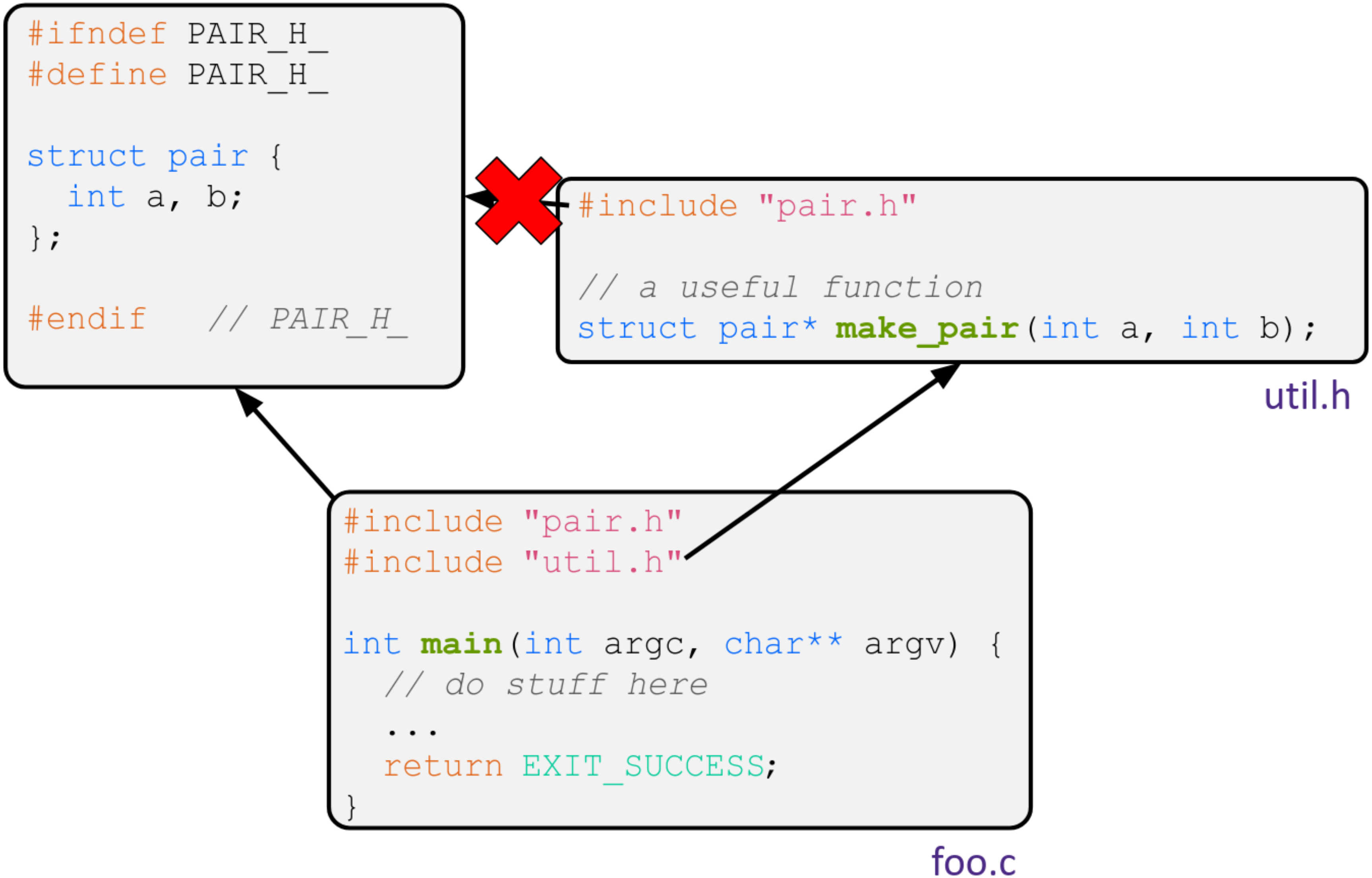
- A standard C Preprocessor trick to deal with this
  - Uses macro definition (`#define`) in combination with conditional compilation (`#ifndef` and `#endif`)

```c
#ifndef PAIR_H_
#define PAIR_H_

struct pair {
  int a, b;
};

#endif  // PAIR_H_
```

pair.h

```c
#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


#ifndef PAIR_H_
#define PAIR_H_
struct pair {
int a, b;
};
#endif // PAIR_H_
#include "pair.h"
// a useful function
struct pair* make_pair(int a, int b);
util.h
#include "pair.h"
#include "util.h"
int main(int argc, char** argv) {
// do stuff here
...
return EXIT_SUCCESS;
}
foo.c

Lecture Outline

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

Headers Aren’t Enough

- So far, we’ve been using header declarations to provide encapsulation (private data/function hiding)
- But code can get around these by re-declaring the variables!
  - The linker will happily link the two variables together
- We need a way to tell the linker which definitions should be only accessible by their module

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* (default)
    - 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* refer to something externally-visible elsewhere

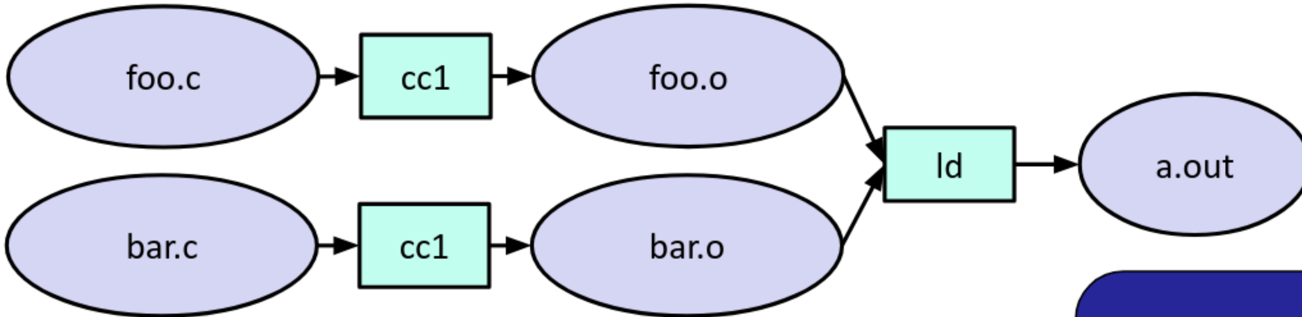
```c
#include <stdio.h>

// 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

```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

External Linkage




```c
#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

```c
#include <stdi

// "counter" i
// initialized
// 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

If you forget to add this, you'll get a linker error!

Internal Linkage

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

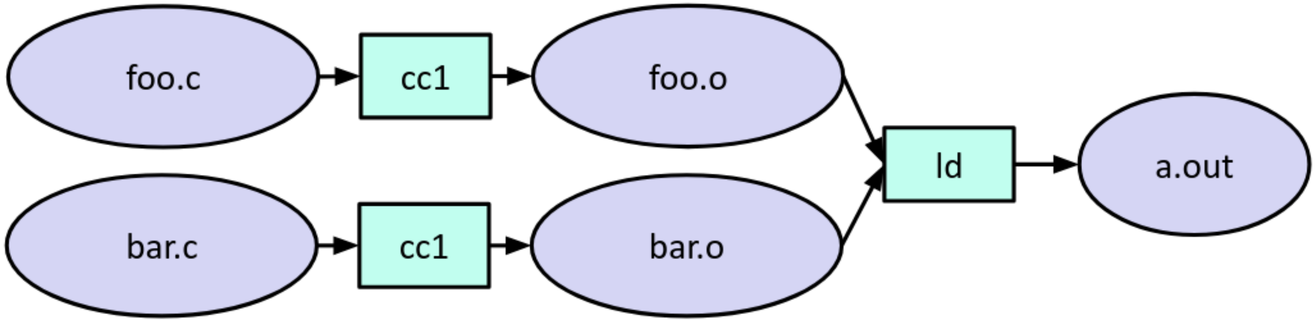
```c
#include <stdio.h>

// 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

```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

Internal Linkage




```c
#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

```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

Closing Thoughts on Data Visibility

- Don't do either of these with an unchanged NthPrime.c!

```c
#define NUM_PRECALC 20
static int16_t
kPrecalculated[NUM_PRECALC] =
    {2, 3, 5, 7, /* etc */ };

int16_t NthPrime(int16_t n);
```

NthPrime.static.h

```c
extern int16_t kPrecalculated[];


int16_t NthPrime(int16_t n);
```

NthPrime.extern.h

- Static variable in the header: every file that #includes the header will have its own private copy of the variable
  - Unnecessary data duplication!
- Extern variable in the header: the accompanying .c file must define the extern'ed variable for successful linkage

Function Visibility

bar.c

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

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

main.c

```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;
}
```

Linkage Issues

- Every global (variables and functions) is `extern` by default
  - Unless you add the `static` specifier, if some other module uses the same name, you'll end up with a collision!
    - Best case: compiler (or linker) error
    - Worst case: stomp all over each other
- It's good practice to:
  - Use `static` to "defend" your globals
    - Hide your private stuff!
  - Place external declarations in a module's header file
    - Header is the public specification

Static Confusion…

- C has a *different* use for the word "`static`": to create a persistent *local* variable
  - The storage for that variable is allocated when the program loads, in either the `.data` or `.bss` segment
  - Retains its value across multiple function invocations
  - Confusing! Don't use!! (But you may see it ☹)

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

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

int main(int argc, char** argv) {
  foo(); foo(); bar(); bar(); return EXIT_SUCCESS;
}
```

static_extent.c

Static Confusion…

- C has a *different* use for the word "`static`": to create a persistent
  - The storage
    `.data` or
  - Retains its
  - Confusing!

```
bash$ gcc -Wall -g -o static_extent static_extent.c
bash$ ./static_extent
foo has been called 1 times
foo has been called 2 times
bar has been called 1 times
bar has been called 1 times
```

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

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

int main(int argc, char** argv) {
  foo(); foo(); bar(); bar(); return EXIT_SUCCESS;
}
```

static_extent.c

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()`