CSE333 Systems Programming
Modules and the Preprocessor

Instructors:

Naomi Alterman

Teaching Assistants:

Ann Baturytski	Barbora Buzkova	Mendel Carroll
Camden Harris	Hannah Hempstead	Rishabh Jain
Irene Lau	Jonathan Nister	Advay Patil
Aviel Wood	Angela Wu	Jennifer Xu

♫ Administriviaa ♫

- Exercise 2 done (good job!)
- Exercise 3 out today
- Next week:
  - Ex3 due Wednesday
  - HW1 due Thursday
  - Plan ahead! You got this 💪!

HW1 tips

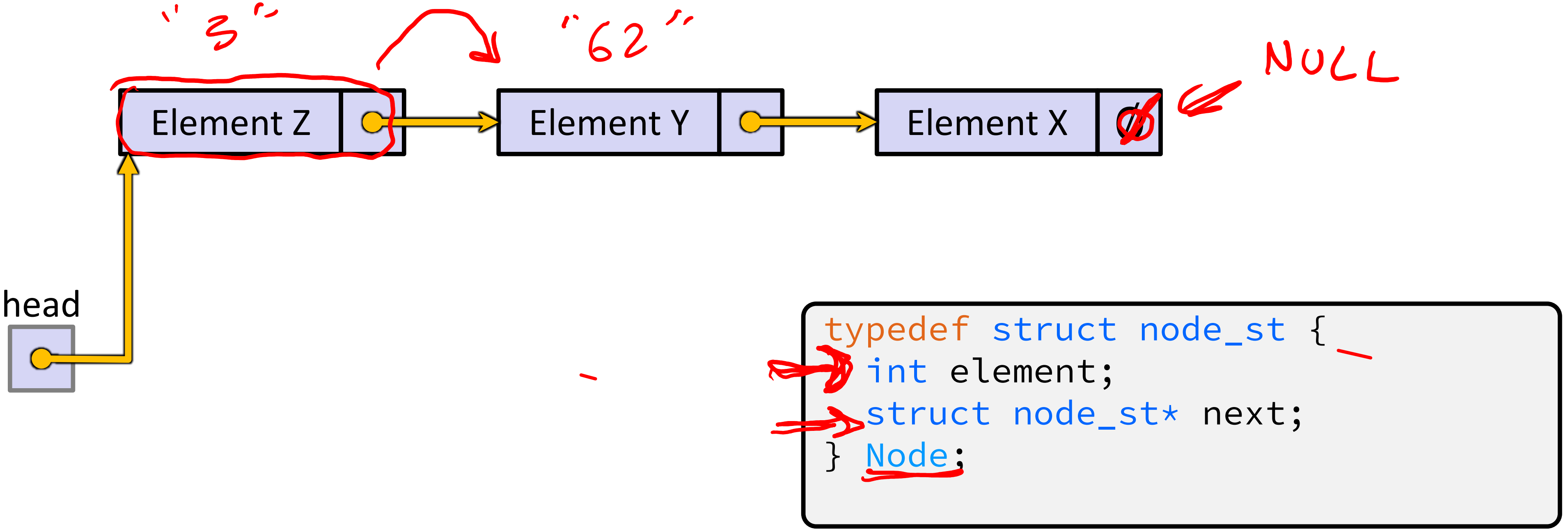
- Watch that `HashTable` doesn't violate the modularity of `LinkedList` (*i.e.*, respect the interfaces!)
- Watch for pointers to local (stack) variables
- Use gdb and valgrind and ***draw memory diagrams!***
- Please leave "`STEP #`" markers for graders!
- Light grading of your git commit history

Lecture Outline

- **Generic Data Structures in C**
- Multi-File C Programs
- C Preprocessor Intro

Simple Linked List in C

- Each node in a linear, singly-linked list contains:
  - An `int` as its payload
  - A pointer to the next node in the linked list
    - The last node's pointer is `NULL` (or some other special value)



Linked List Node

```
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;
}
```

manual_list.c

OS kernel [protected]
Stack
main
n1: element 1, next
n2: element 2, next Ø
Heap (malloc/free)
Read/Write Segment
Read-Only Segment
(**main, copy**)

stack vars!
offset
+
-

Arrow points to *next* instruction.

Push Onto List (1/14)

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

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

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

new element's payload

push_list.c

OS kernel [protected]
Stack
main — list ∅
Heap (malloc/free)
Read/Write Segment
Read-Only Segment (**main, copy**)

Push Onto List (2/14)

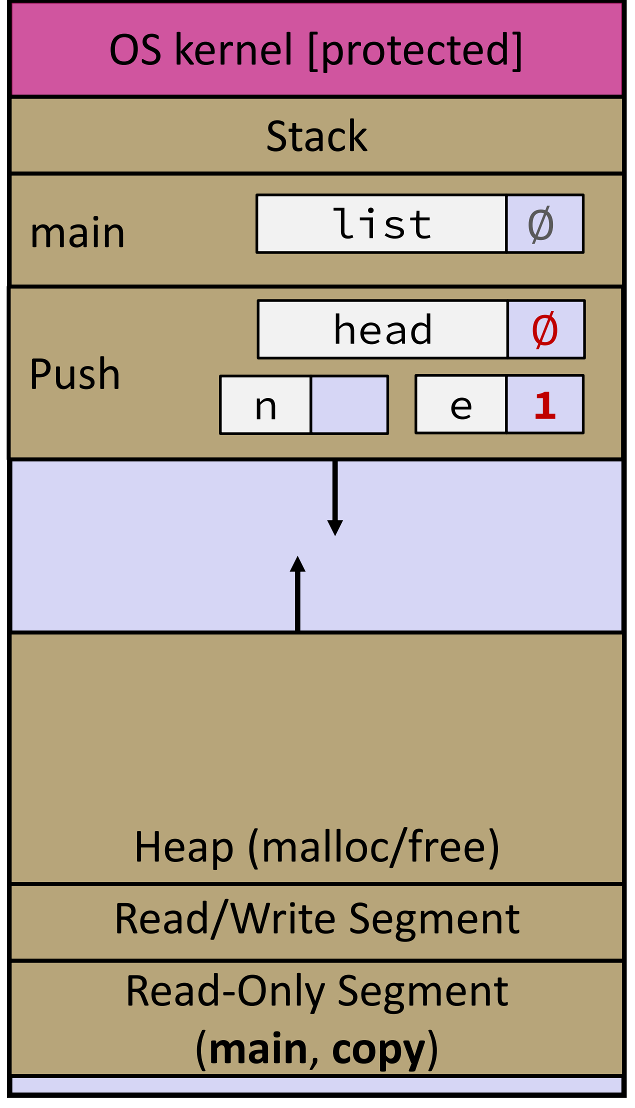
Arrow points to *next* instruction.

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

Node* Push(Node* head, int e) {
  Node* n = (Node*) malloc(sizeof(Node));   ← (arrow)
  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 (3/14)

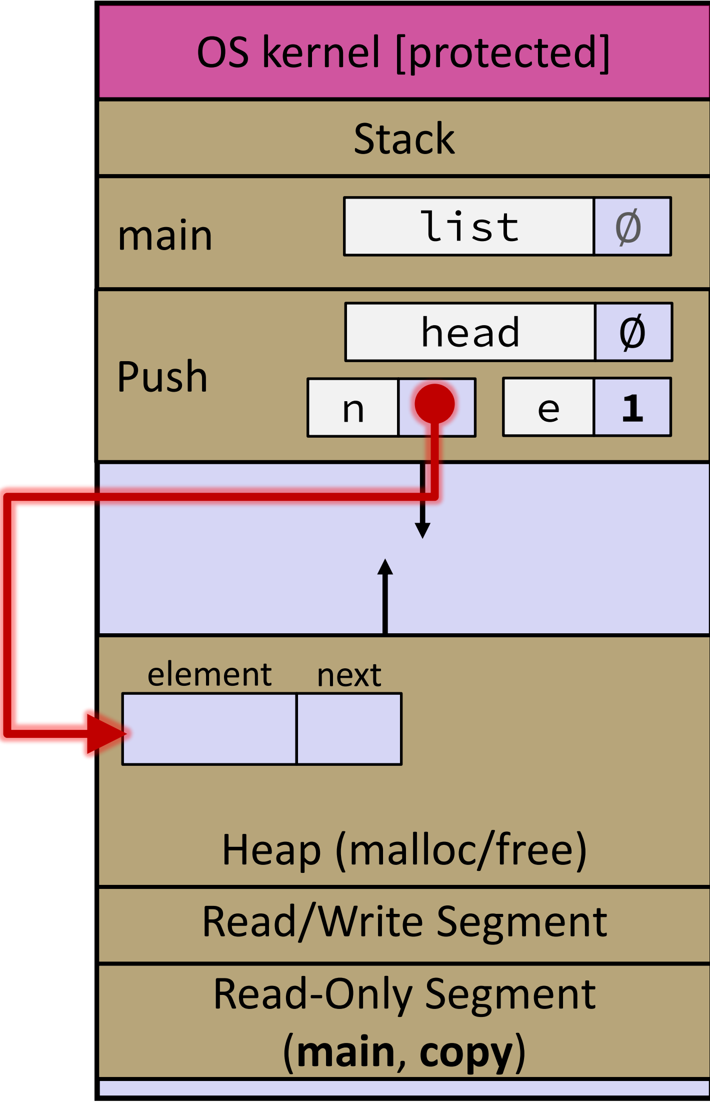
Arrow points to *next* instruction.

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

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

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

push_list.c



Push Onto List (4/14)

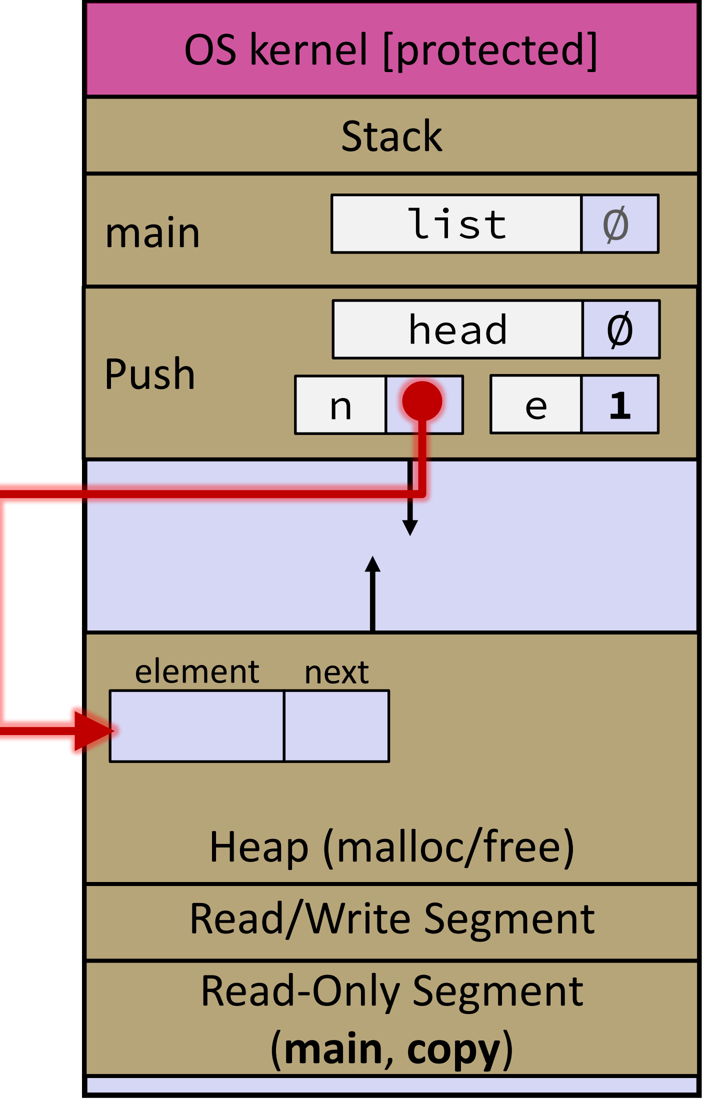
Arrow points to *next* instruction.

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

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

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

push_list.c



Push Onto List (5/14)

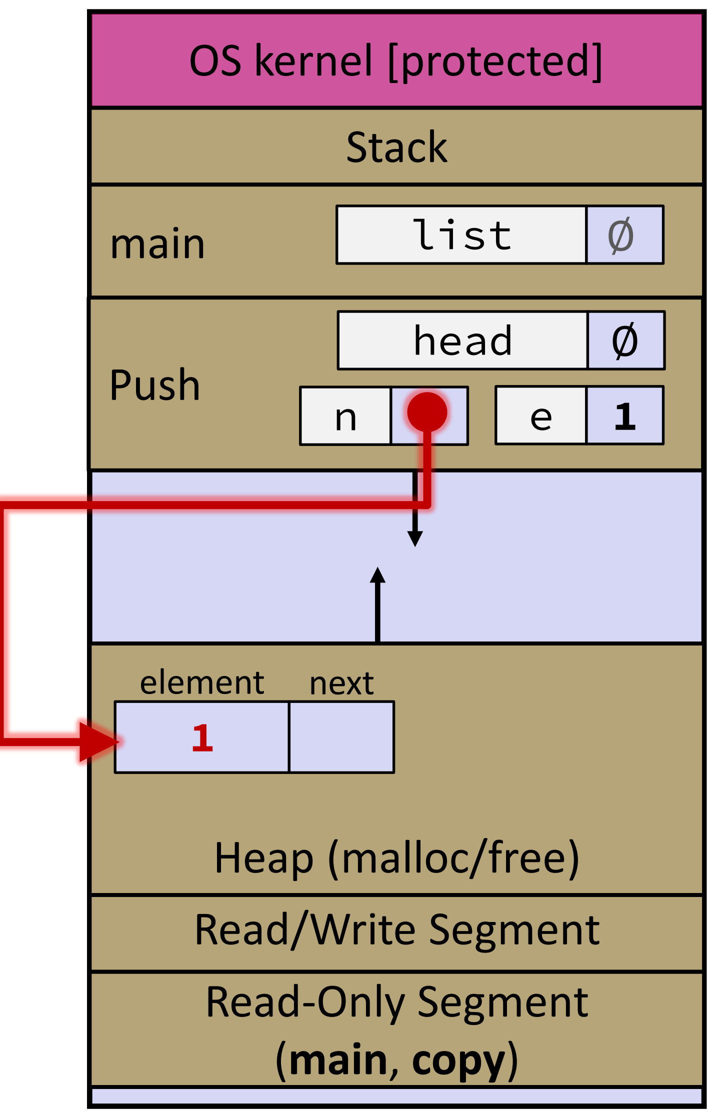
Arrow points to *next* instruction.

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

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

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

push_list.c



Push Onto List (6/14)

Arrow points to *next* instruction.

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

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

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

push_list.c

OS kernel [protected]

Stack

main: list

Push: head Ø, n, e 1

element 1, next Ø

Heap (malloc/free)

Read/Write Segment

Read-Only Segment (**main, copy**)

Arrow points to *next* instruction.

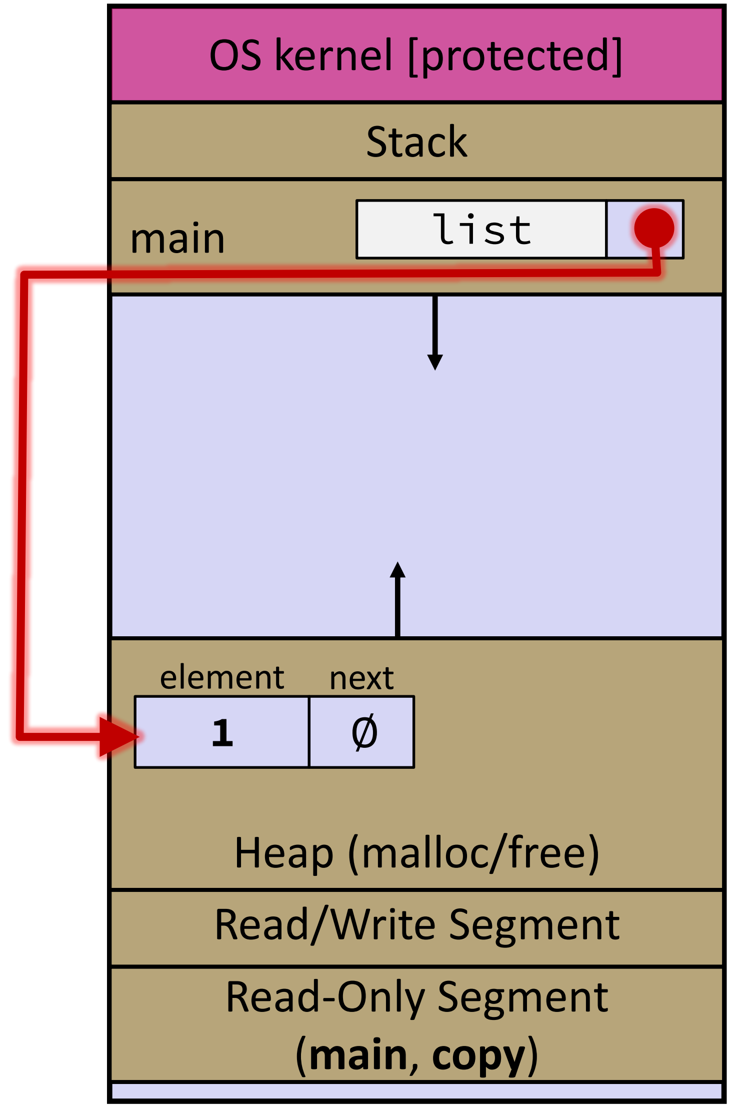
Push Onto List (7/14)

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

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

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

push_list.c



Push Onto List (8/14)

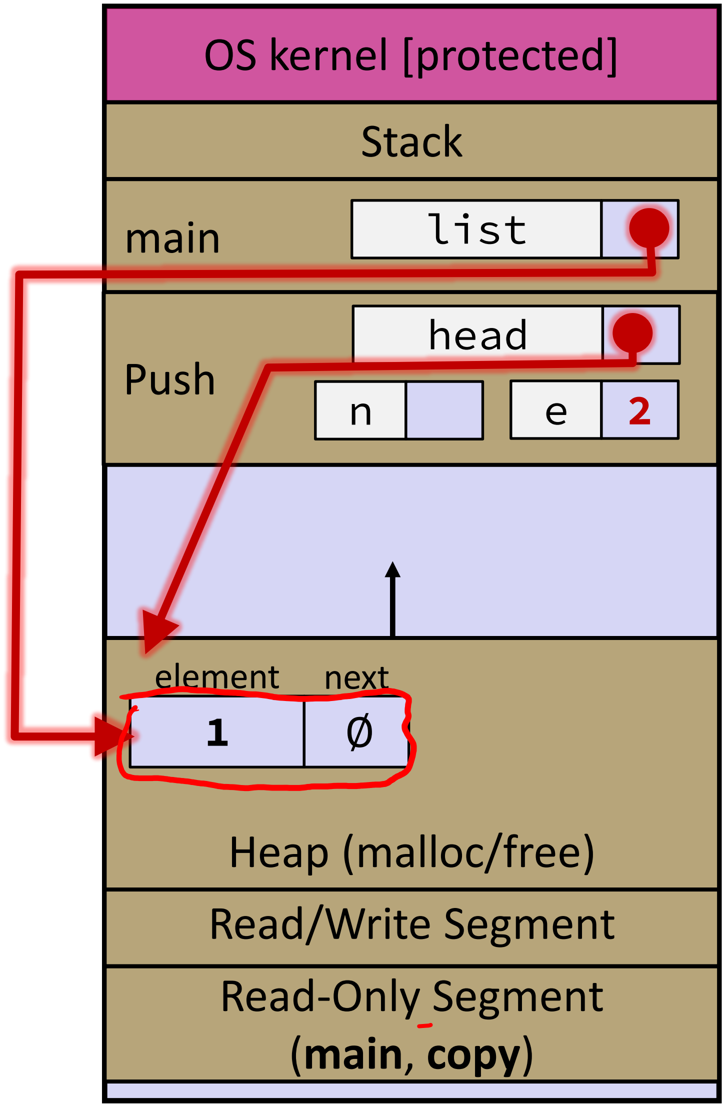
Arrow points to *next* instruction.

```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.

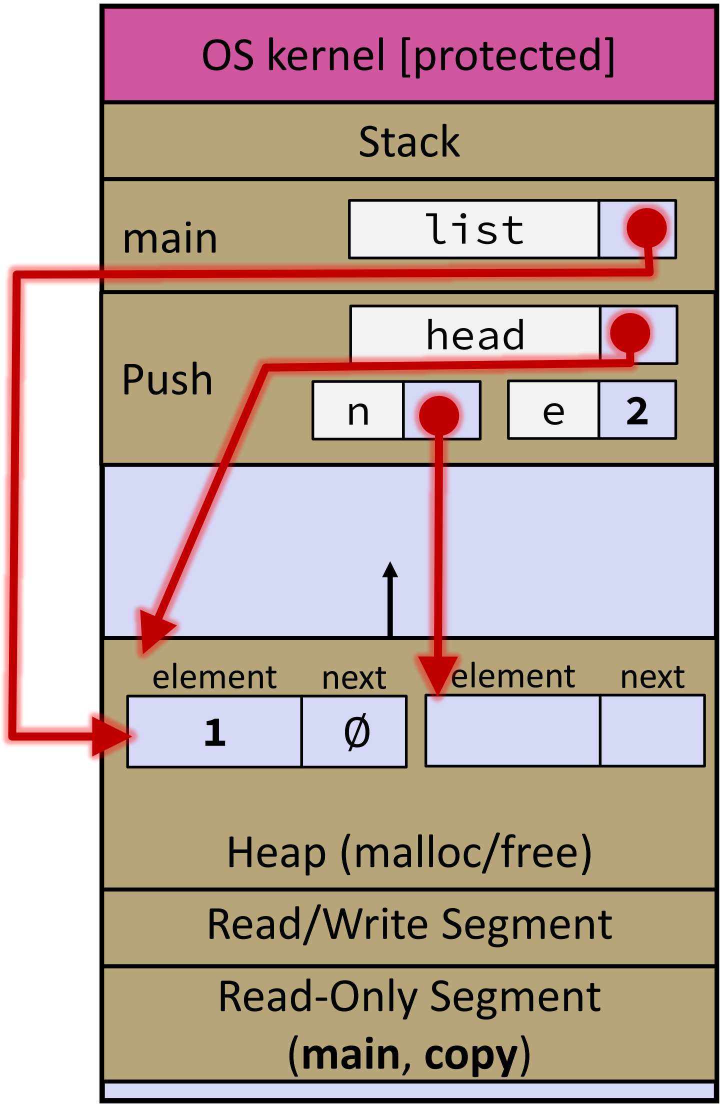
Push Onto List (9/14)

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

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

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

push_list.c



Push Onto List (10/14)

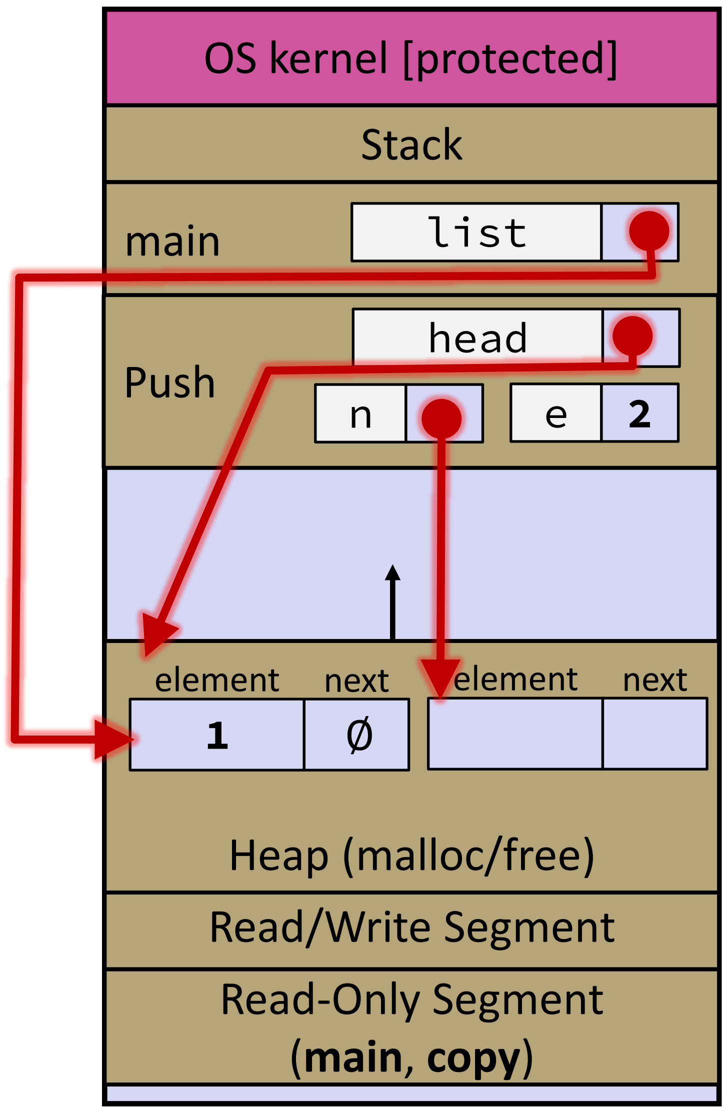
Arrow points to *next* instruction.

```
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 (11/14)

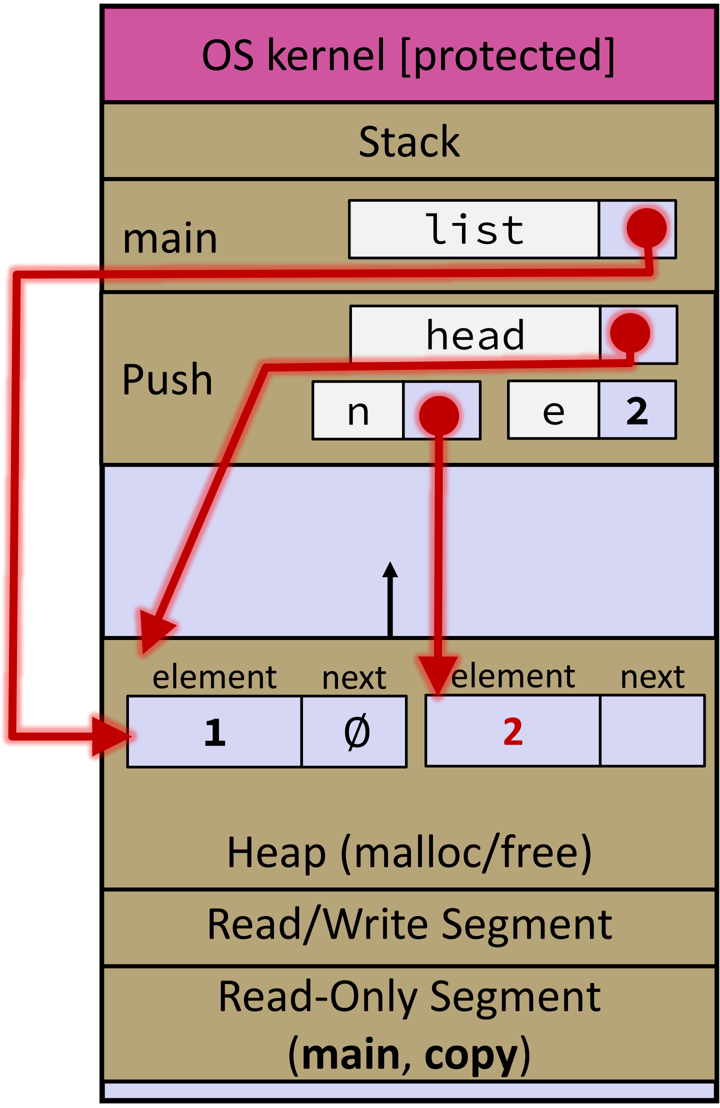
Arrow points to *next* instruction.

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

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

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

push_list.c



Arrow points to *next* instruction.

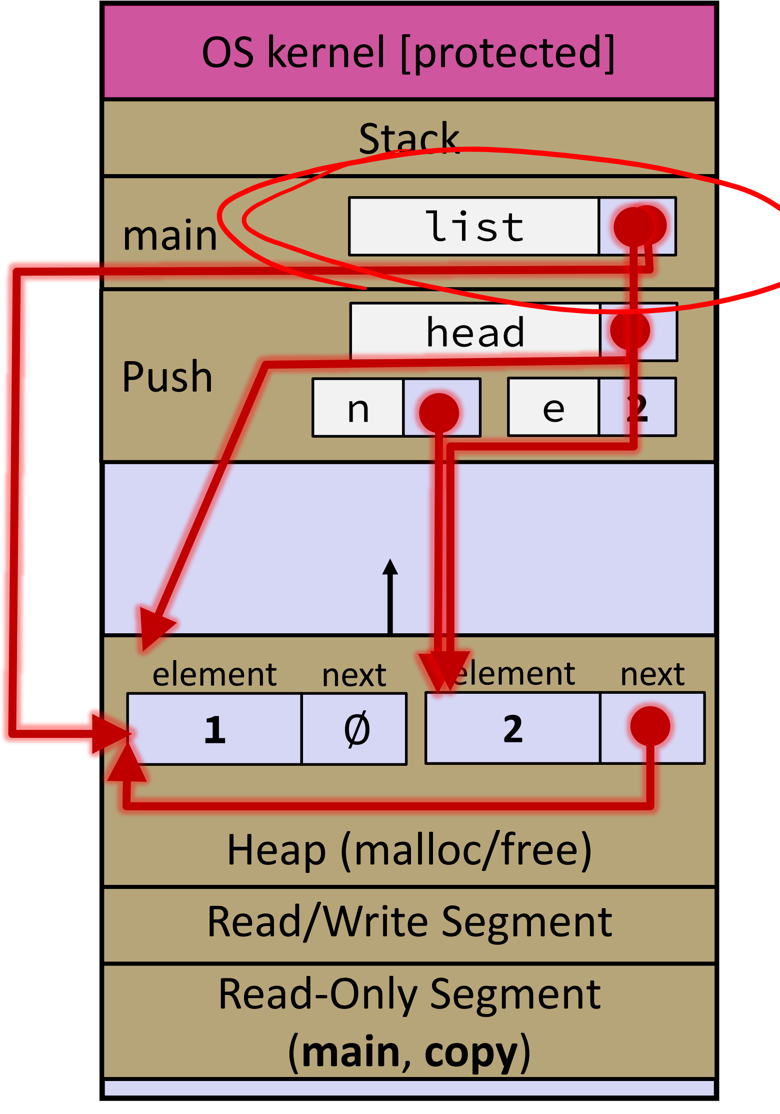
Push Onto List (12/14)

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

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

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

push_list.c



Push Onto List (13/14)

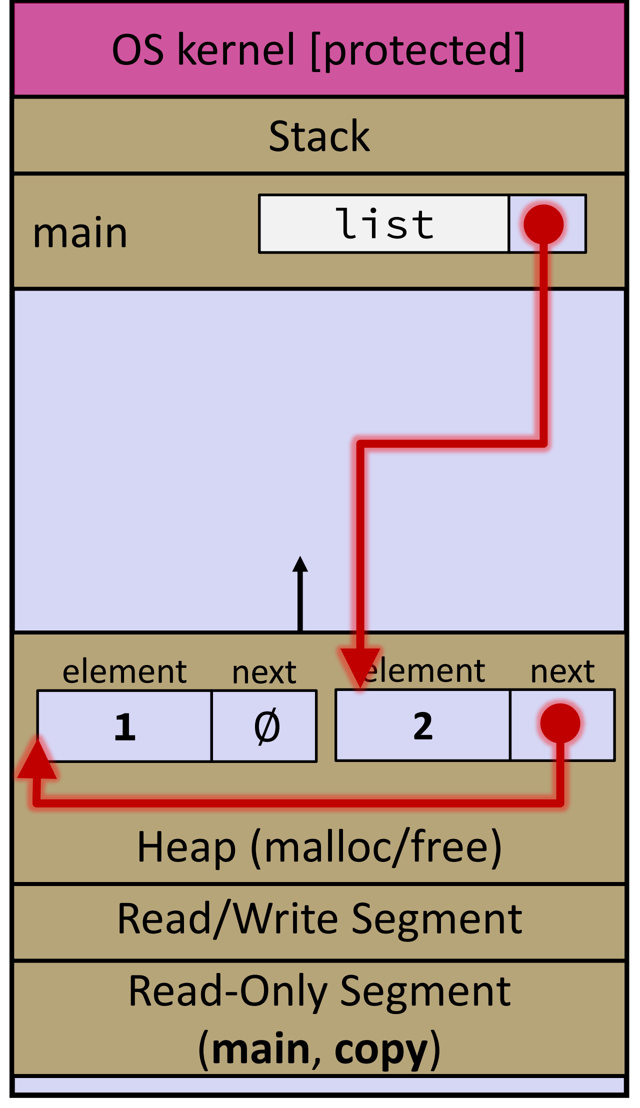
Arrow points to *next* instruction.

```
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 (14/14)

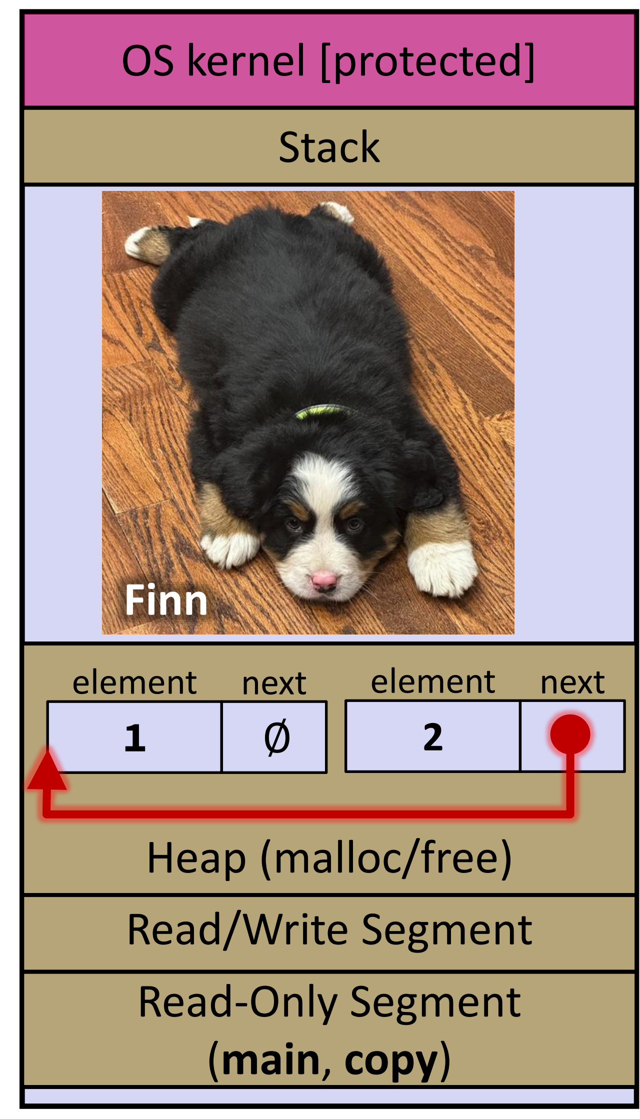
Arrow points to *next* instruction.

```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

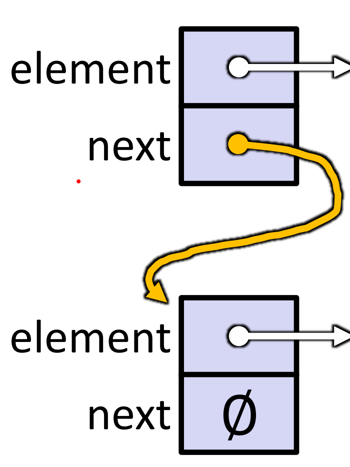


A Generic Linked List

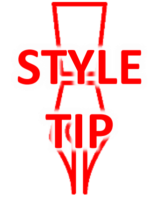
- Let's generalize the linked list element type
  - Let customer decide type (instead of always `int`)
  - Idea: let them use a generic pointer (*i.e.*, a `void*`)

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

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



Using a Generic Linked List



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

manual_list_void.c

```
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!";
  char* goodbye = "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;
}
```

Resulting Memory Diagram

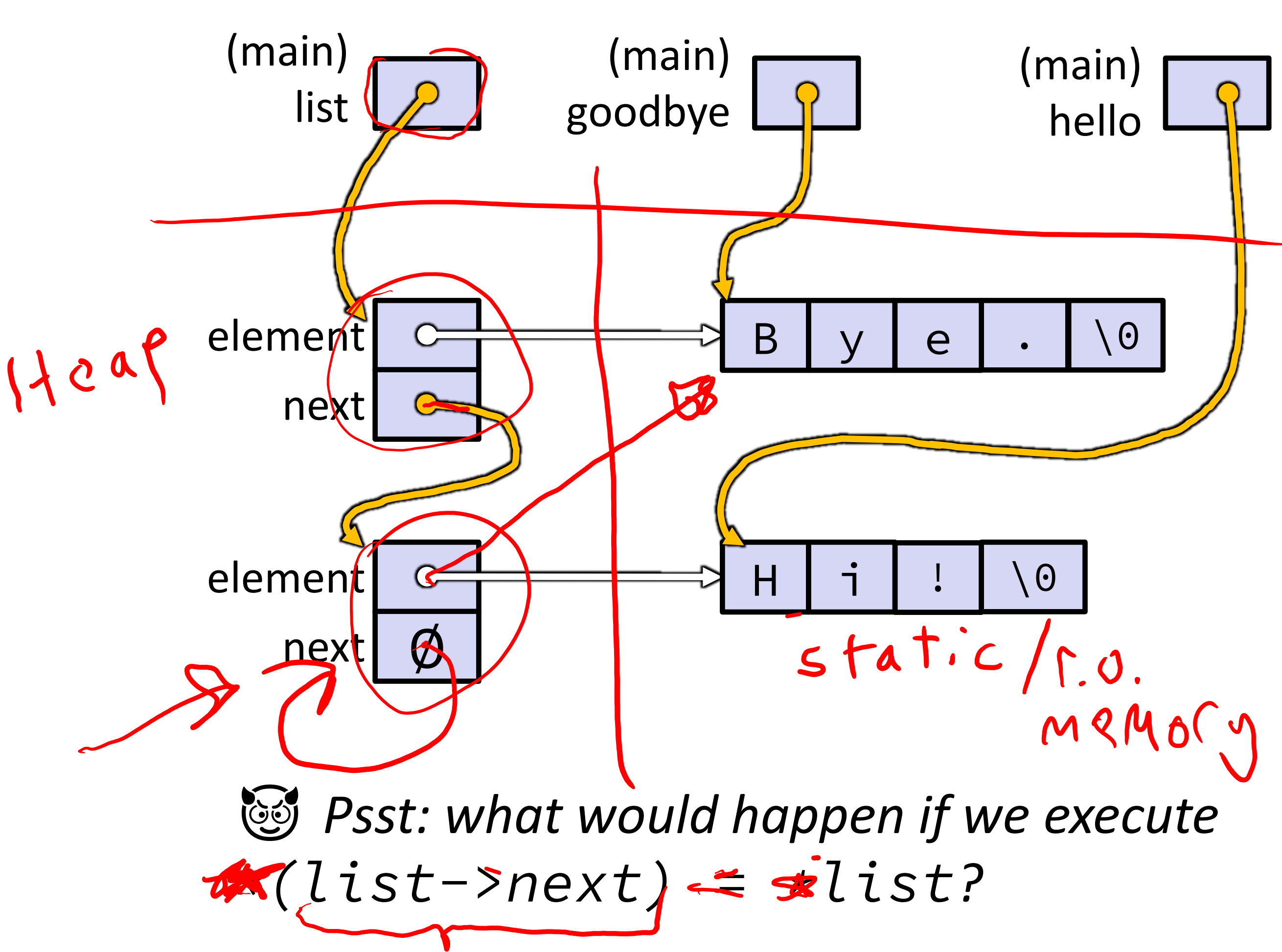
manual_list_void.c

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

Node* Push(Node* head, void* e);

int main(int argc, char** argv) {
  char* hello = "Hi!";
  char* goodbye = "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;
}
```



Psst: what would happen if we execute

`(list->next) = list?`

pollev.com/naomila

Let's finish this function prototype for Pop()

Function takes the head of a linked list of Node, then removes and returns the first node:

```
________ Pop(________ head);
```

1. Should the return type be (a) Node or (b) Node*?

2. Should the parameter be (a) Node, (b) Node*, or (c) Node**?

Something's Fishy…

- A (benign) memory leak!

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

  list = Push(list, (void*) hello);
  list = Push(list, (void*) goodbye);
  return EXIT_SUCCESS;
}
```

- Try running with Valgrind:

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

Lecture Outline

- Generic Data Structures in C
- **Multi-File C Programs**
  - **The C toolchain**
  - **Modules & Interfaces**
- C Preprocessor Intro

Multi-File C Programs

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

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

C Header Files

- Header: A file whose only purpose is to be `#include`'d
  - Generally has a filename `.h` extension
  - Holds the variables, types, and function prototype declarations that make up the interface to a module
  - 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 C files can use module name by `#include`-ing `name.h`
    - They should assume as little as possible about the implementation in `name.c`

Program Using a Linked List

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

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

ll.c

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

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

ll.h

```c
#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

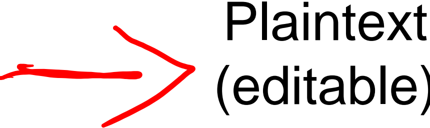
- Four parts:
  - 1) Compile `example_ll_customer.c` into an object file
  - 2) Compile `ll.c` into an object file
  - 3) Link both object files into an executable
  - 4) Test, Debug, Rinse, Repeat

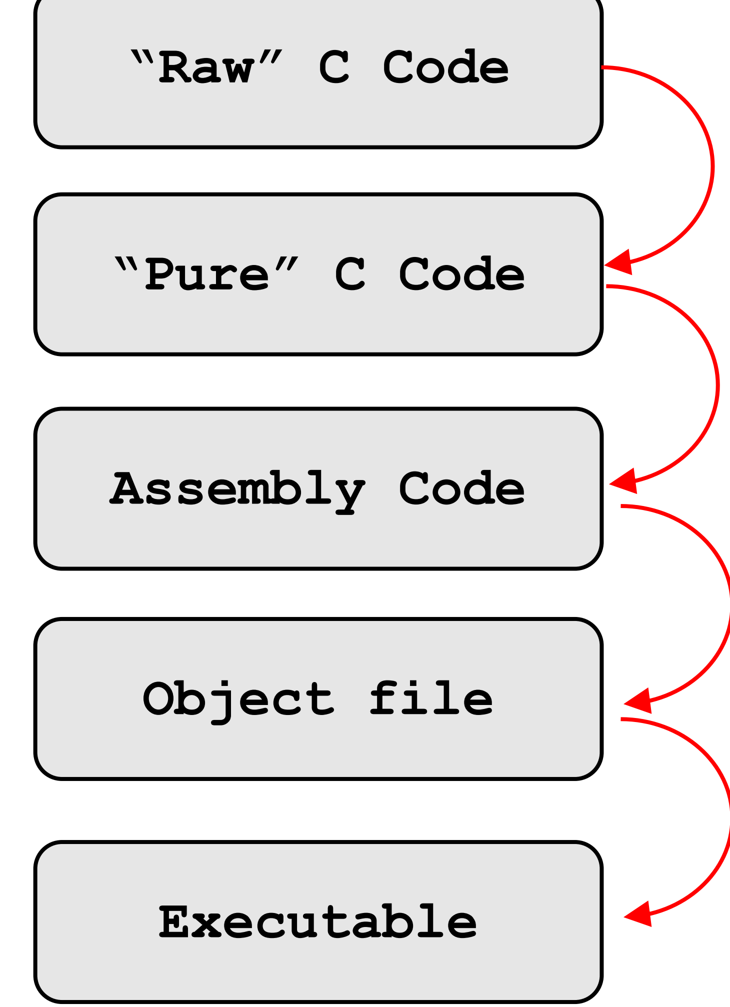
```
$ gcc -Wall -g -o example_ll_customer.o -c example_ll_customer.c
$ gcc -Wall -g -o ll.o -c ll.c
$ gcc -g -o example_ll_customer ll.o example_ll_customer.o
$ ./example_ll_customer
Payload: 'yo!'
Payload: 'goodbye'
Payload: 'hello'
$ valgrind -leak-check=full ./example_ll_customer
... etc ...
```

output filename

"don't make a final executable"

Managing all these C and H files





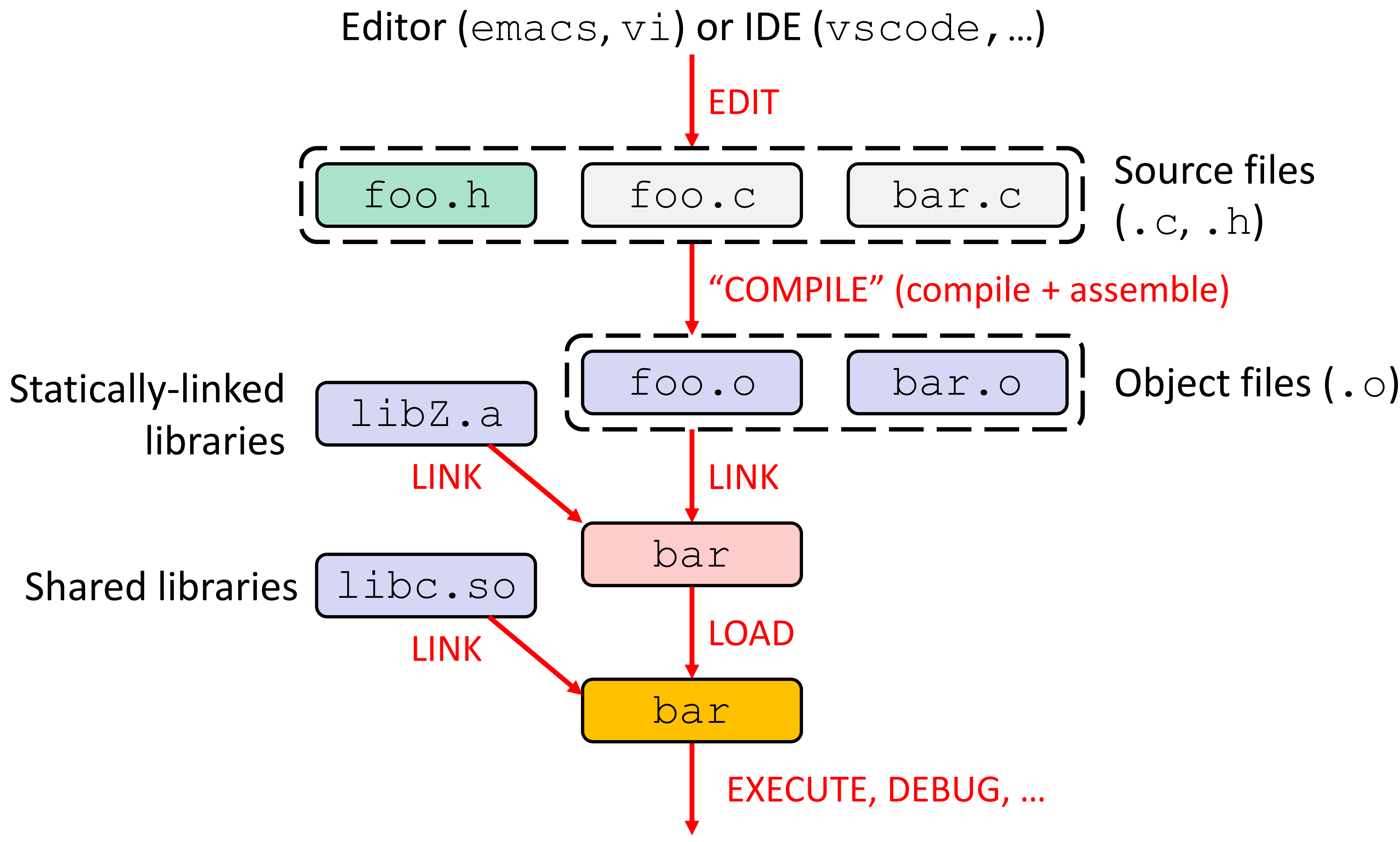
Pre-processor goes through each C file and carries out its "#directive"s. Header files are copy-and-pasted in. Output C file can be compiled without any other files.

Compiler turns each preprocessed C file into Assembly code. All vars/fns are left as symbolic references.

Assembler turns assembly files intoobject files. Where possible, symbolic references are turned into addresses. Where not, symbols are left as placeholders.

Linker gloms all objects together and resolves the address placeholders. We get our final program.

C Workflow (revisited)



C Module Conventions (1/2)

- File contents:
  - `.h` files only contain *declarations*, never *definitions*
  - `.c` files never contain prototype declarations for functions that are intended to be exported through the module interface
  - Public-facing functions are **`ModuleName_FunctionName()`** and take a pointer to "`this`" as their first argument
- Includes:
  - ***NEVER*** `#include` a `.c` file – only `#include` `.h` files
  - `#include` all of headers you reference, even if another header (transitively) includes some of them
- Compiling:
  - Any `.c` file with an associated `.h` file should be able to be compiled (together via `#include`) into a `.o` file

C Module Conventions (2/2)

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

e.g., 333 project code

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 `double`s
  - Includes functions to:
    - add, subtract, multiply, and divide complex numbers
  - Implement a test driver in `test_complex.c`
    - Contains `main()`