

Data Structures and Modules

CSE 333 Spring 2020

Instructor: Hal Perkins

Teaching Assistants:

Ramya Challa

Greg Guo

Travis McGaha

Cosmo Wang

Haoran Yu

Mengqui Chen

Zachary Keyes

Arjun Singh

Yifan Xu

Velocity Yu

John Depaszthory

CJ Lin

Guramrit Singh

Robin Yang

Administrivia

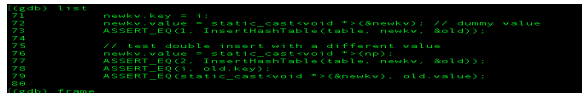
- ❖ Exercise 3 was due this morning
- ❖ Exercise 4 out today, due Friday morning
- ❖ Exercise 5 will be based on code from tomorrow's sections
 - Due Monday morning, out late tomorrow
 - Section code “handout” for tomorrow will be added to the calendar later today
 - Suggestion: download/print a copy 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

Discussion board and email hints

Please help your readers (both for cse333 and elsewhere)

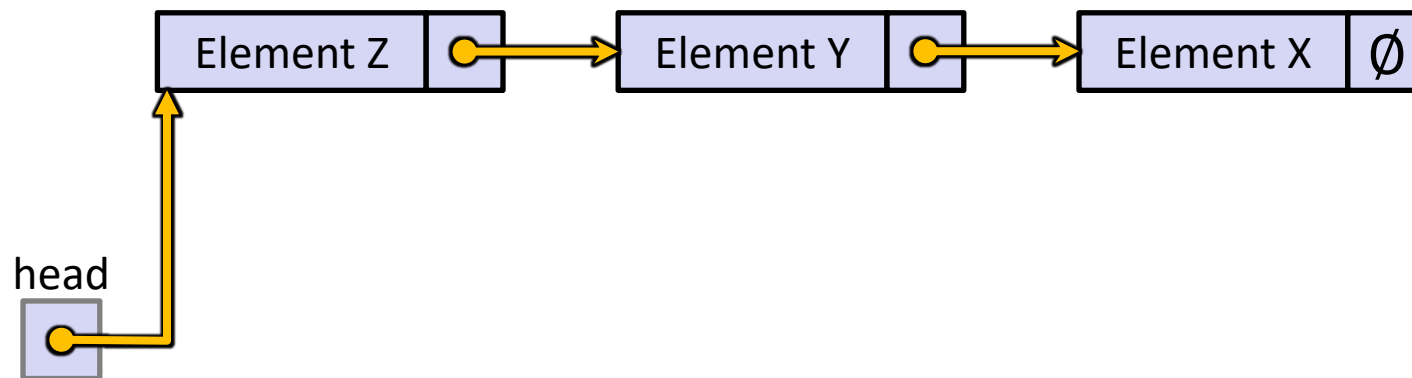
- ❖ Use descriptive titles and provide enough context in the question so readers don't need to go on a treasure hunt
 - Not great: Subject: About exercise
 - Only ε better: Subject: exercise
 - Much better: Subject: is sizeof(type) ok in ex42?
- ❖ Please don't post screenshots of text 
 - Hard to read and/or requires opening an extra window
 - If it's text, copy text (drag to select in terminal or dialog boxes) and paste it into your message/question if you can
 - Images are fine if they actually help explain the question/posting
- ❖ Your readers thank you for your help 😊

Lecture Outline

- ❖ **Implementing 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:
 - 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`

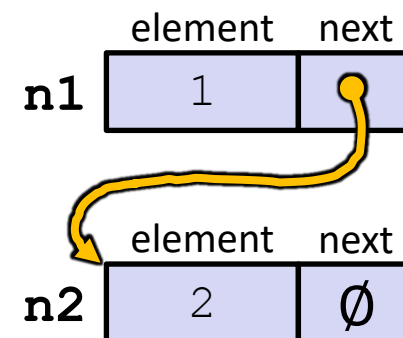
```
#include <stdio.h>

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;
}
```

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;
}

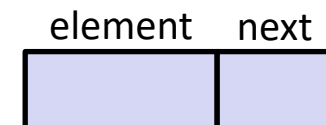
int main(int argc, char** argv) {
    Node* list = NULL;
    list = Push(list, 1);
    list = Push(list, 2);
    return 0;
}
```

(main) list 

(Push) head 

(Push) e 

(Push) n 



push_list.c

Push Onto List

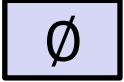
Arrow points to
next instruction.

```
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;
}

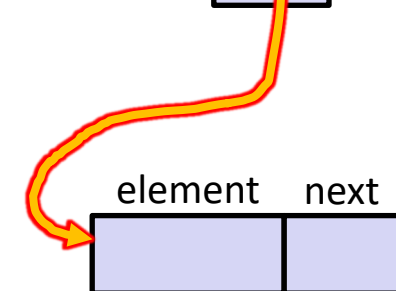
int main(int argc, char** argv) {
    Node* list = NULL;
    list = Push(list, 1);
    list = Push(list, 2);
    return 0;
}
```

(main) list 

(Push) head 

(Push) e 

(Push) n 



push_list.c

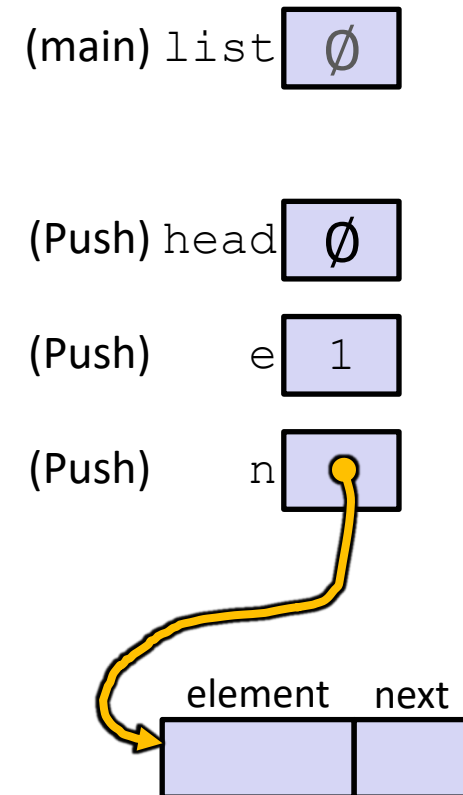
Push Onto List

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

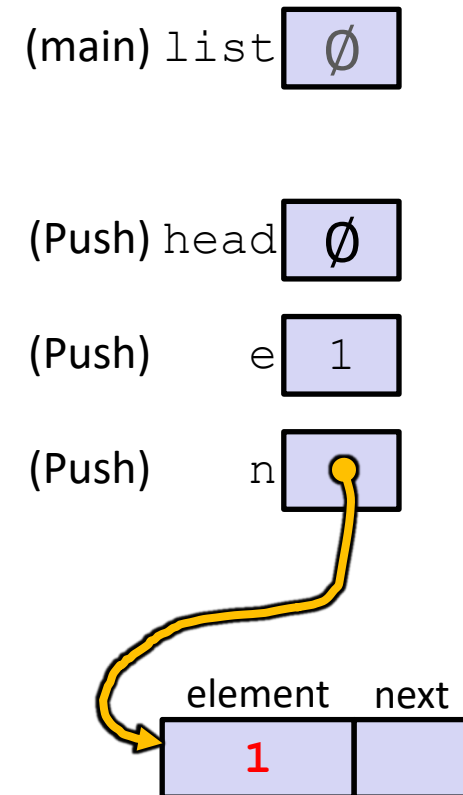
Push Onto List

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

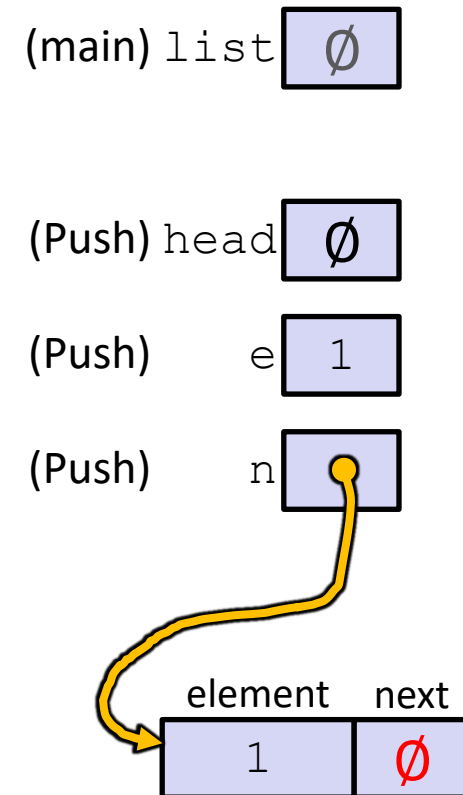
Push Onto List

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

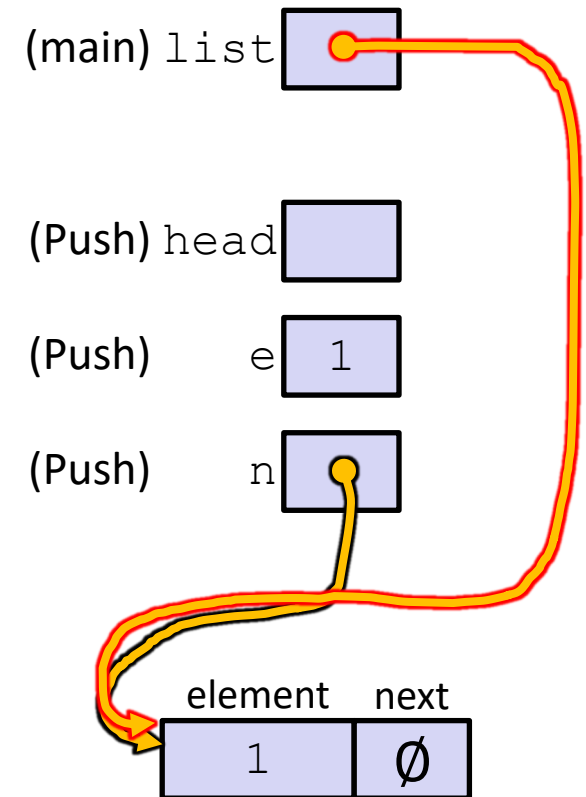
Push Onto List

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

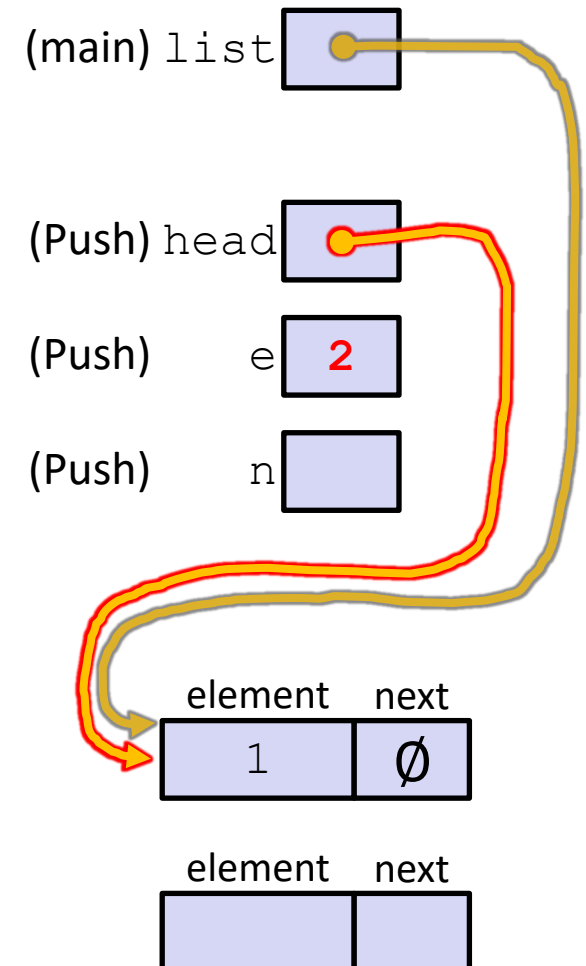
Push Onto List

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

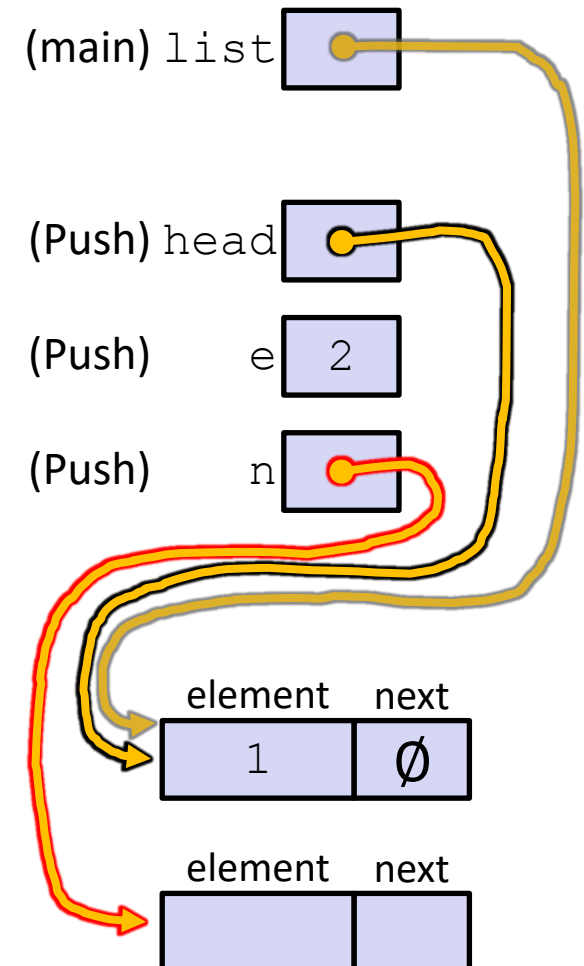
Push Onto List

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

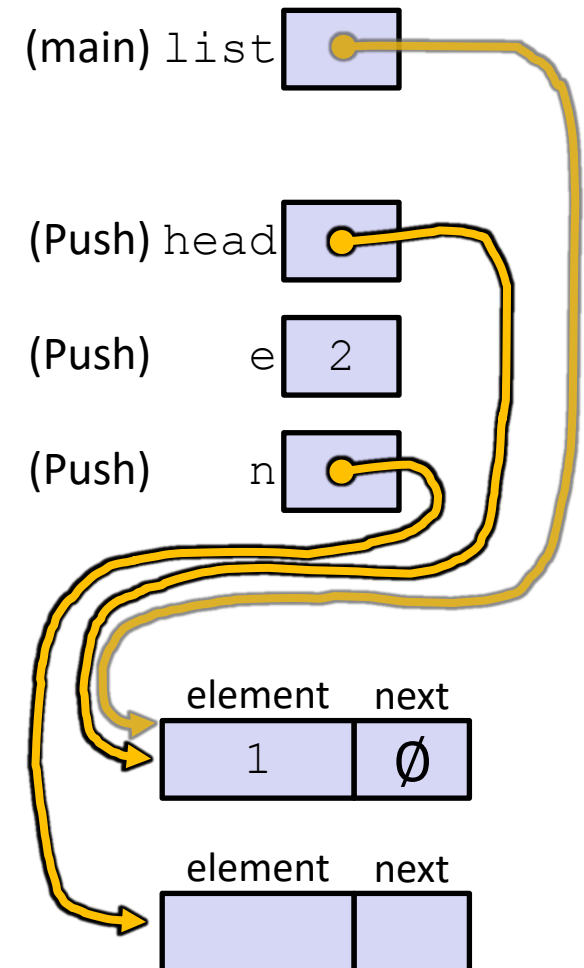
Push Onto List

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

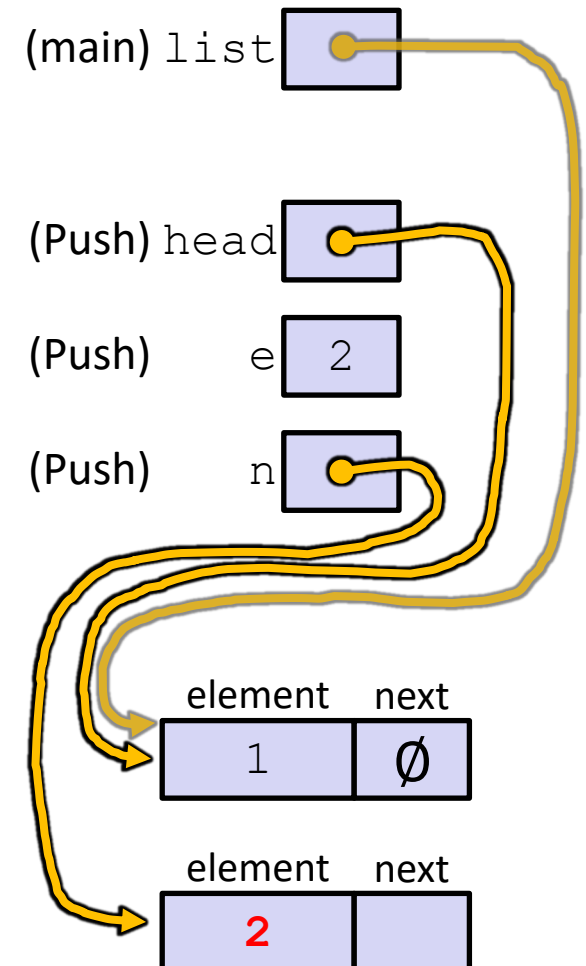
Push Onto List

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

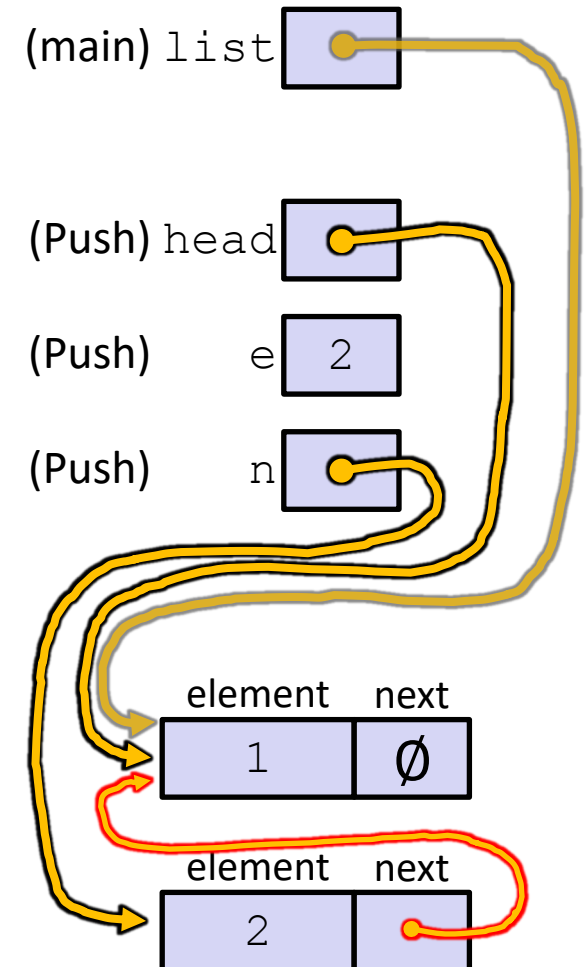
Push Onto List

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

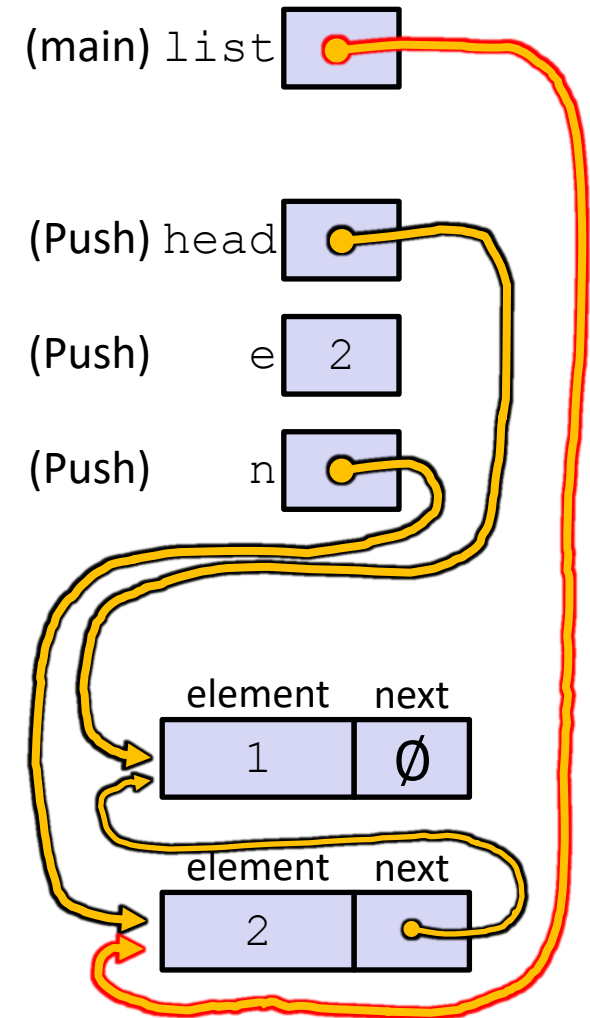
Push Onto List

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;

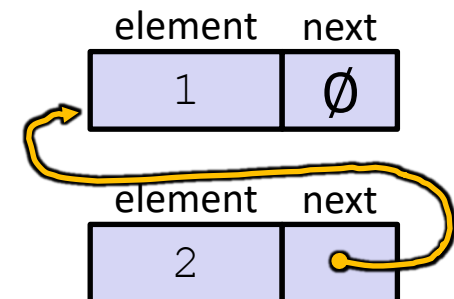
Node* Push(Node* head, int e) {
    Node* n = (Node*) malloc(sizeof(Node));
    assert(n != NULL); // crashes if false
    n->element = e;
    n->next = head;
    return n;
}

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

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
```

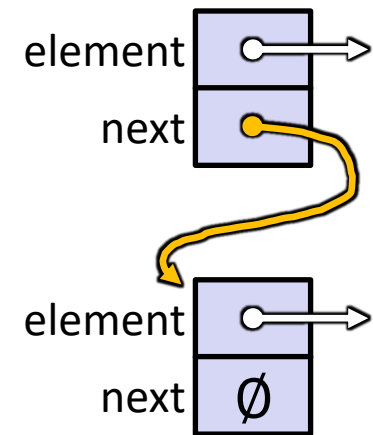


push_list.c

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;  
}
```



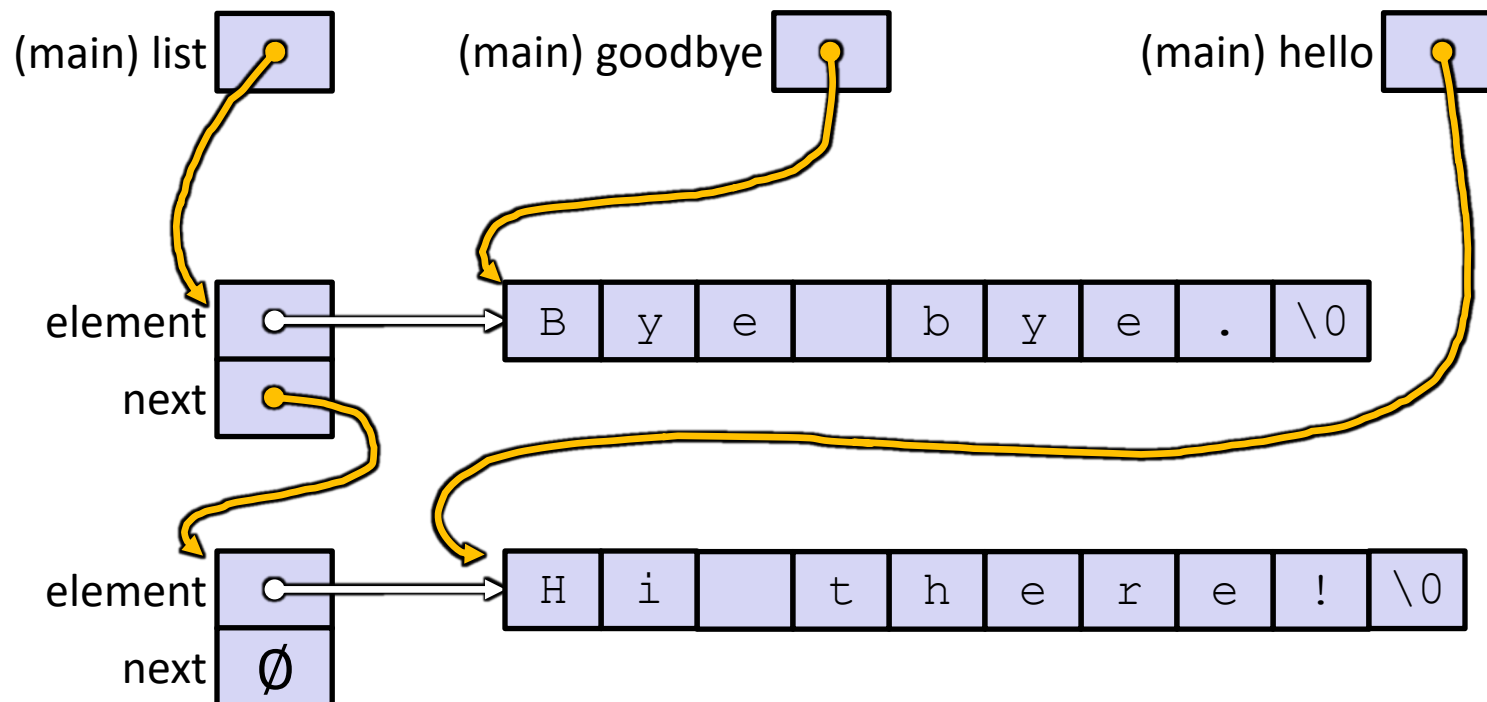
Using a Generic Linked List

- ❖ Type casting 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



Lecture Outline

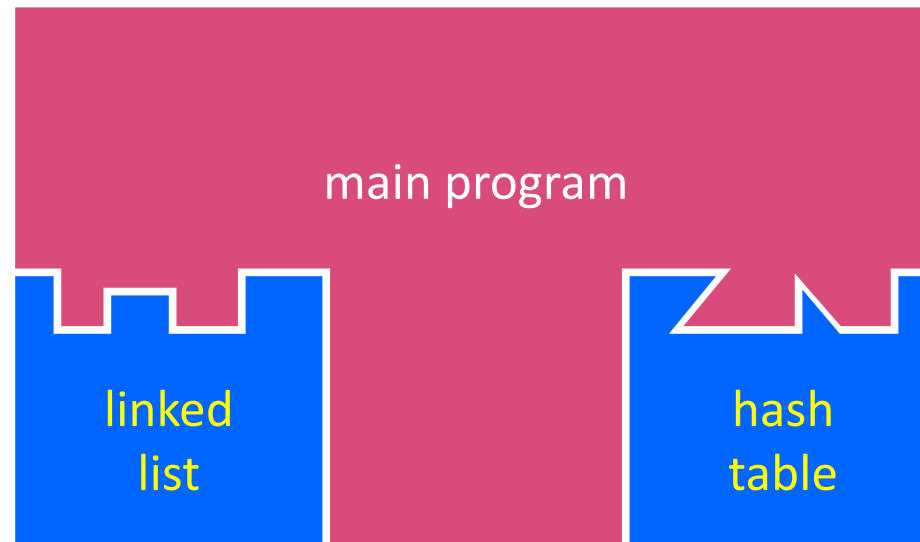
- ❖ Implementing Data Structures in C
- ❖ **Multi-file C Programs**
 - **C Preprocessor Intro**

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 C 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
- ❖ **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

- ❖ 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
 - **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
 - 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

#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

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

```
#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;  
}
```

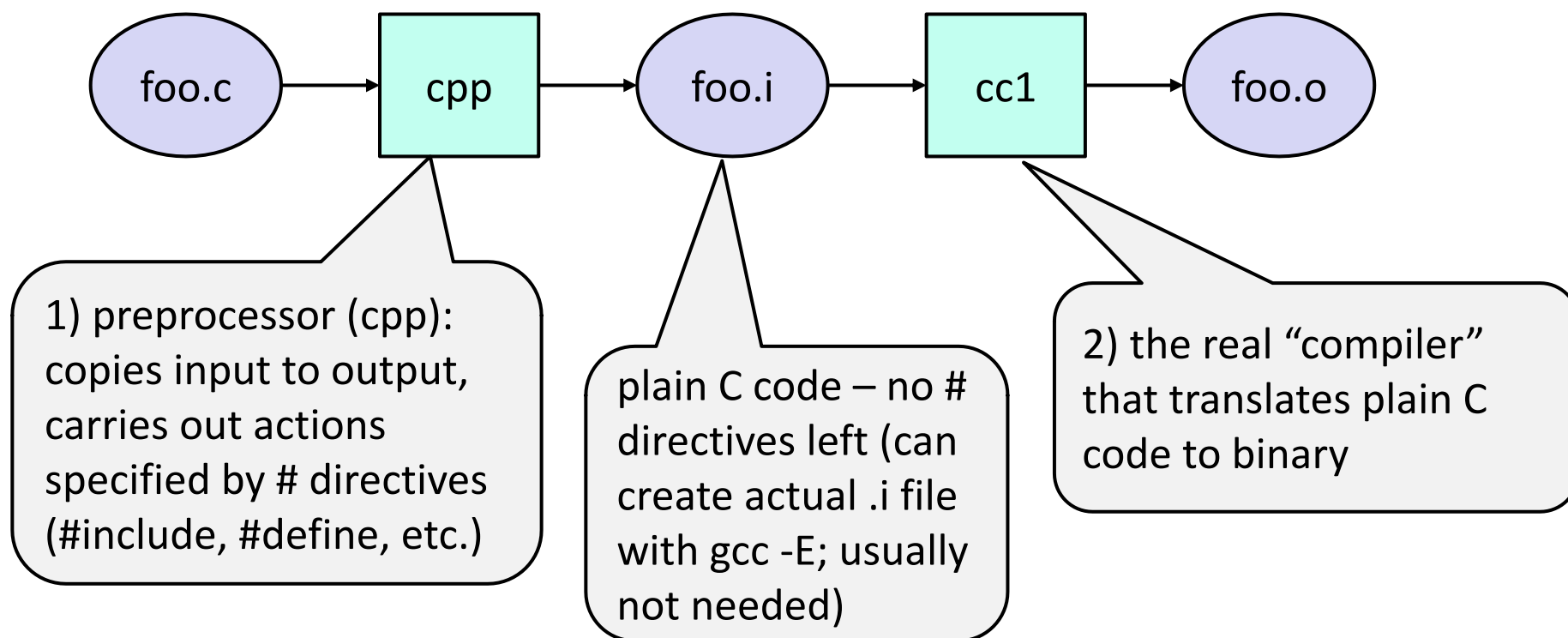
`cpp_example.c`

```
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;  
}
```


What Is gcc Really Doing?

- ❖ gcc is actually a pretty simple program that runs the actual 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

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

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

ll.c

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

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

ll.h

```
#include "ll.h"

int main(int argc, char** argv) {
    Node* list = NULL;
    char* hi = "hello";
    char* bye = "goodbye";

    list = Push(list, (void*)hi);
    list = Push(list, (void*)bye);

    ...

    return 0;
}
```

example_ll_customer.c

Compiling the Program

❖ Four parts:

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

```
bash$ gcc -Wall -g -c -o example_ll_customer.o example_ll_customer.c
bash$ gcc -Wall -g -c -o ll.o ll.c
bash$ gcc -g -o example_ll_customer ll.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

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()`