Data Structures and Modules

CSE 333 Autumn 2020

Instructor: Hal Perkins

Teaching Assistants:

Rehaan Bhimani
Ramya Challa
Eric Chan
Mengqi Chen
Ian Hsiao
Pat Kosakanchit
Arjun Singh
Guramrit Singh
Sylvia Wang
Yifan Xu
Robin Yang
Velocity Yu

Administrivia

- Exercise 3 was due this morning

- There is no exercise 4 this quarter!!! ☺

- Exercise 5 is based on code from yesterday’s sections
  - Due Monday morning, out yesterday afternoon
  - Will take a bit longer than usual – suggest not leaving it until Sunday night. ☺ ☺

Administrivia

- More about hw1 before the weekend
  - 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
  - Suggestion: have more fun, less anxiety: pace yourself and make steady progress; don't leave it until the last minute!
- Remember: the only supported systems for the class are the Allen School Linux machines (workstations, attus, home VM – Centos 8 + gcc 9). You should be working on those systems and the projects you build ***must*** work there.
  - We do not have the cycles to try to support other Unix-like things or chase bugs due to configuration or software differences (including file transfers to/from Windows systems and editing in native Windows environments)

More Administrivia

- Gitlab repo usage
  - Commit things regularly
    - Newly completed units of work / milestones / project parts
    - End-of-day when wrapping up on one computer so you can later pull changes to a different machine
    - And: for this remote quarter, before “visiting” office hours to make it easier for you and TA to browse code
    - etc.
  - Provides backup: protection against lost files and ability to go back in time to retrieve old versions before they got messed up ☺
  - There shouldn’t be one massive commit the day hw is due
  - But: use it properly
    - 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)

Communications

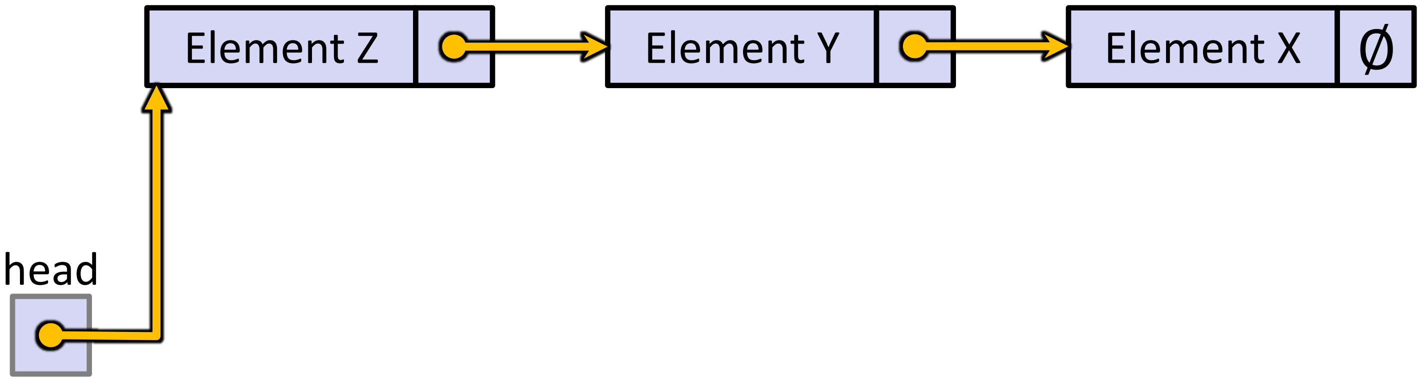
- Please use the discussion board for general questions, helping each other out, checking answers/postings from others, etc.
- We've asked that if you need to contact the staff, please send email to cse333-staff@cs
  - But, for some reason, this quarter we're seeing lots of private messages sent via the discussion board for the first time
  - These are harder to track/follow-up/be-sure-resolved, so…
  - We're going to disable private postings on the discussion board right after class. Please use email for private messages and we'll be better able to help everyone. Use Gradescope for exercise grading requests.
  - And yeah, it's really annoying that different classes all seem to do things in different ways. Will try to fix eventually, but it'll take time…

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

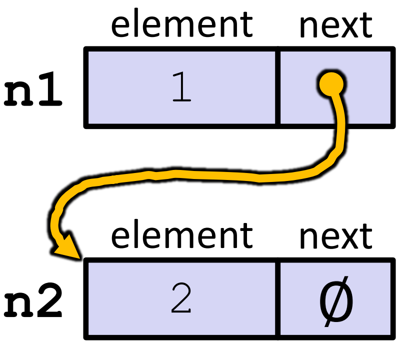


Linked List Node

- Let's represent a linked list node with a struct
  - For now, assume each element is an `int`

```c
#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

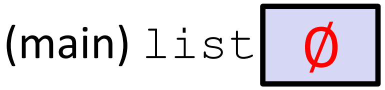


Push Onto List

Arrow points to *next* instruction.

```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 0;
}
```

push_list.c

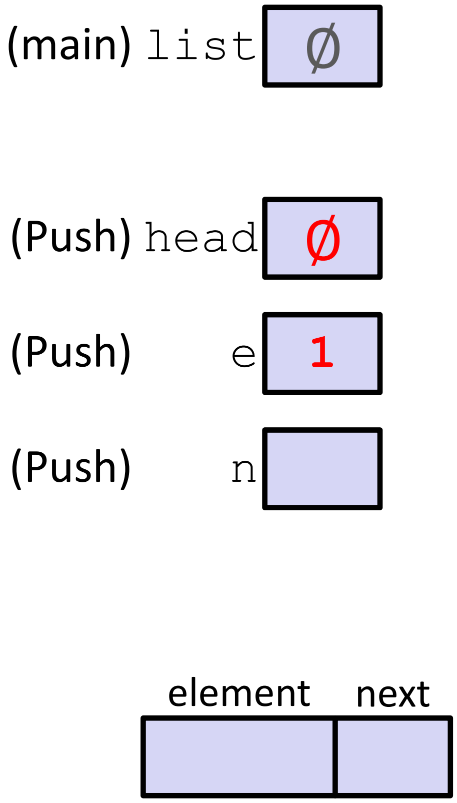


Push Onto List

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

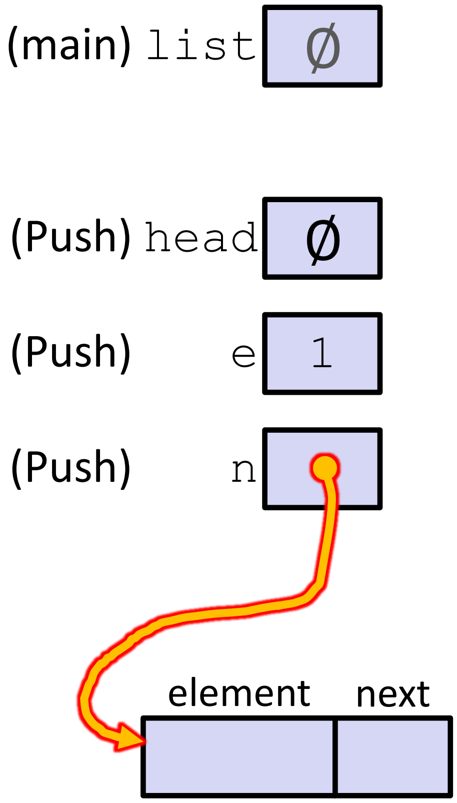


Push Onto List

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

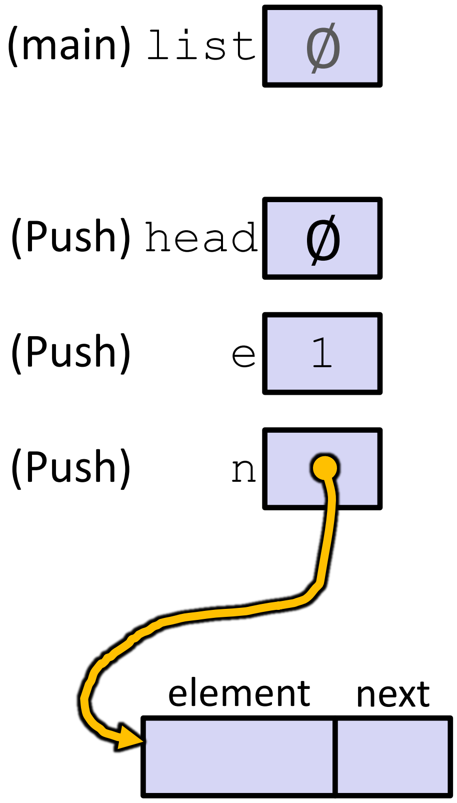


Push Onto List

Arrow points to *next* instruction.

```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 0;
}
```

push_list.c

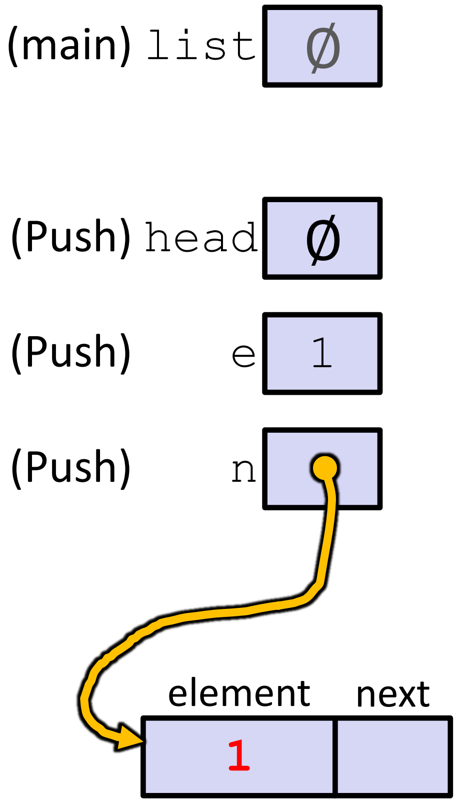


Push Onto List

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

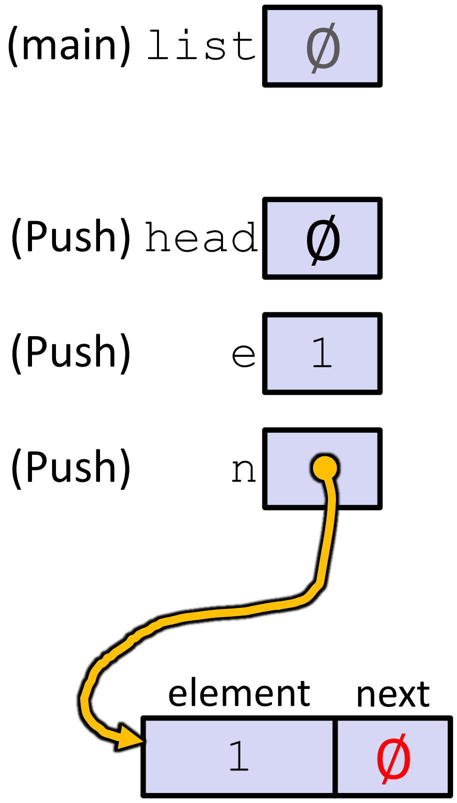


Push Onto List

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

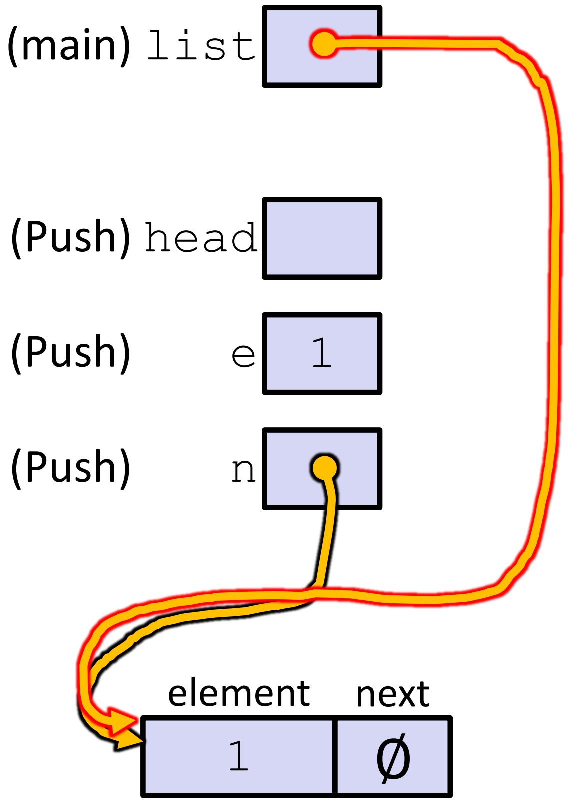


Push Onto List

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

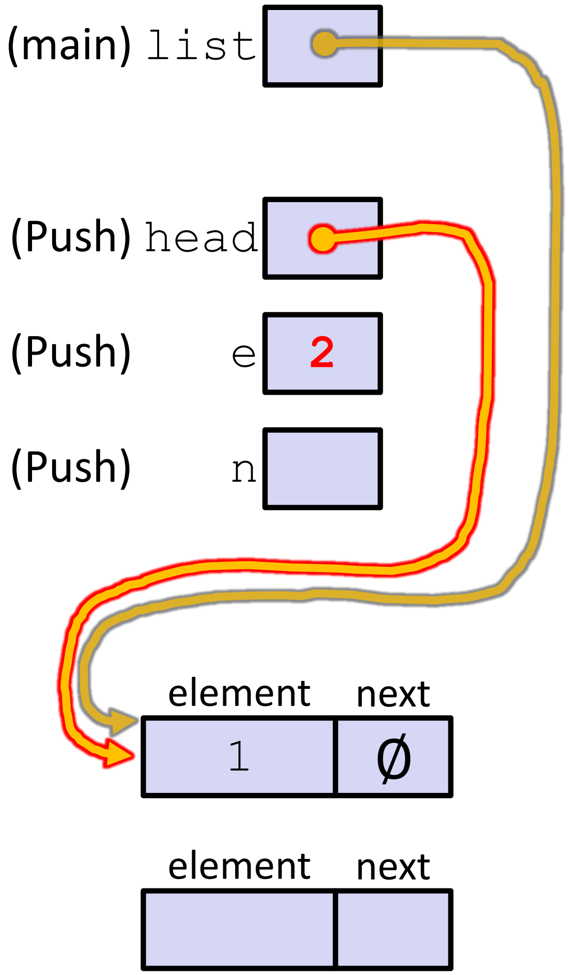


Push Onto List

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

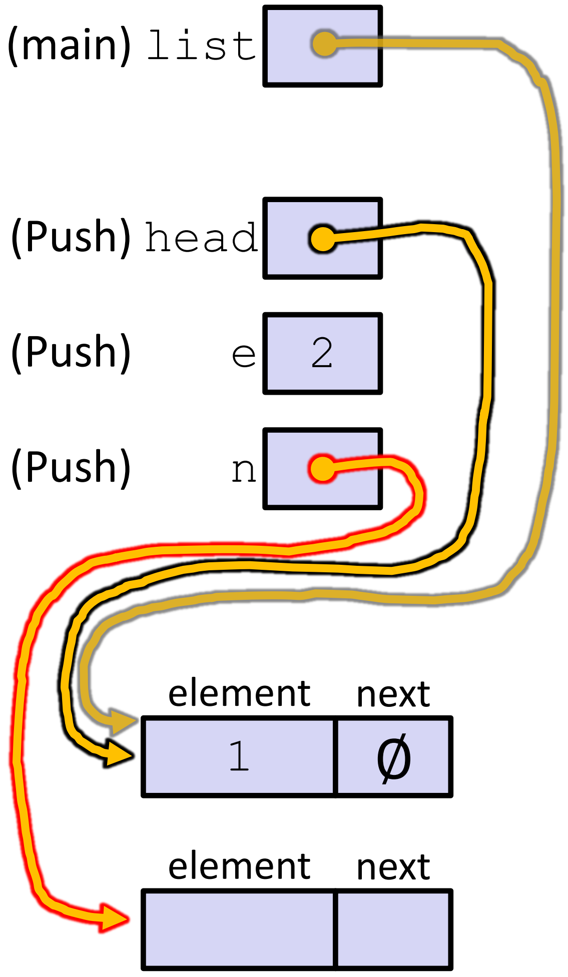


Push Onto List

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

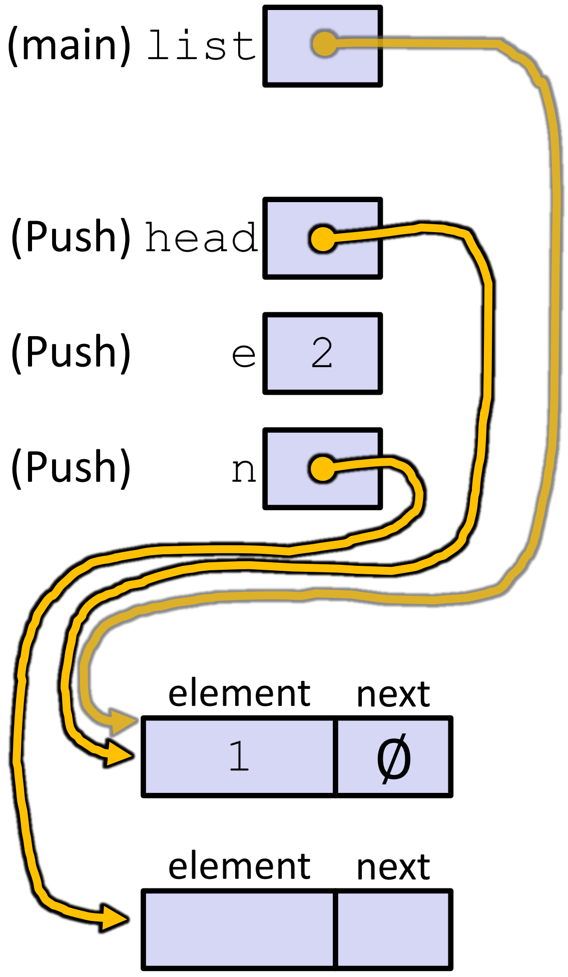


Push Onto List

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

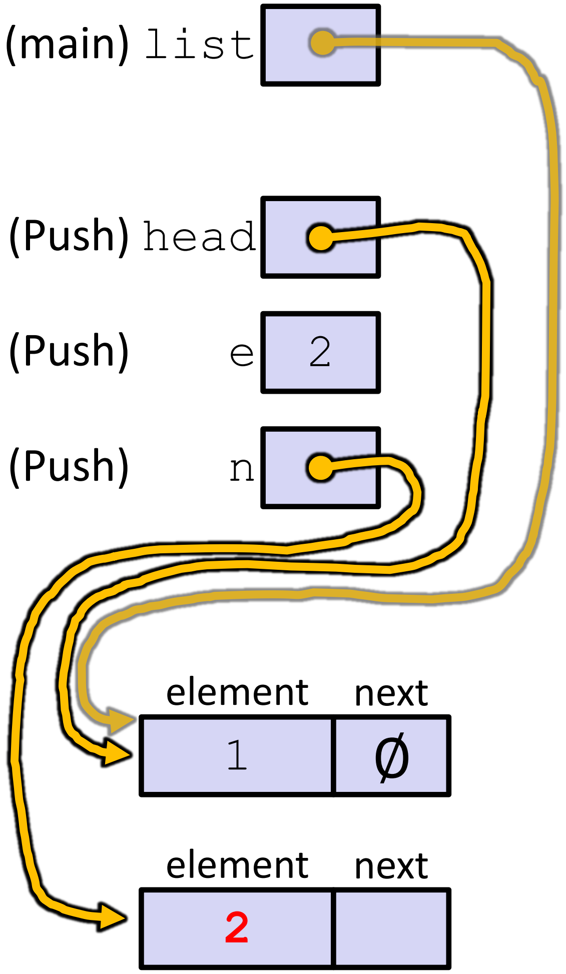


Push Onto List

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

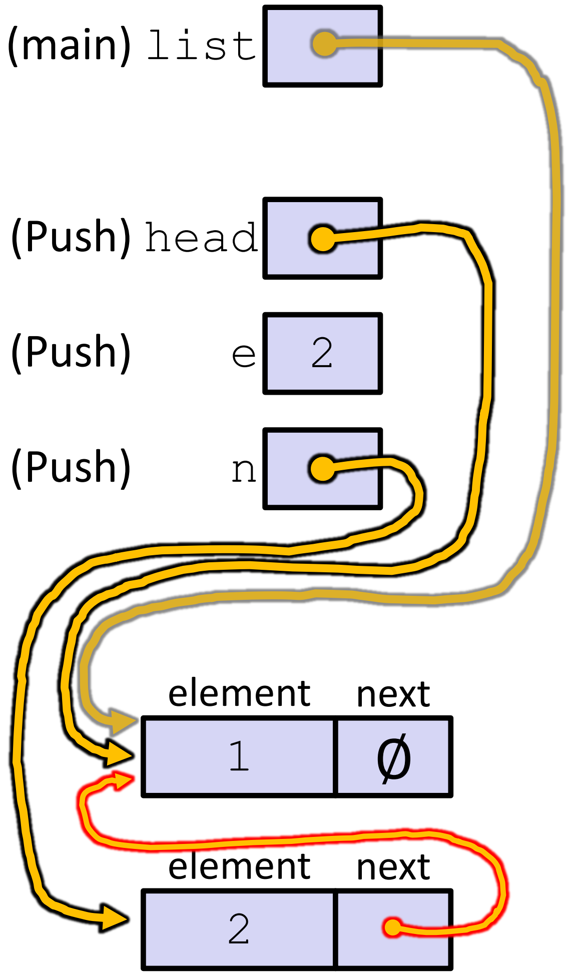


Push Onto List

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

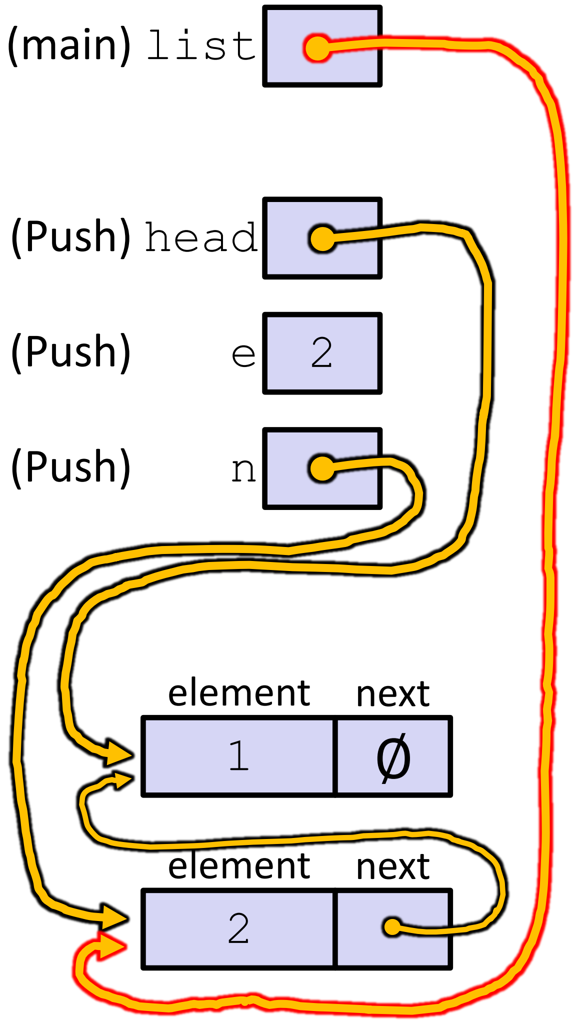


Push Onto List

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

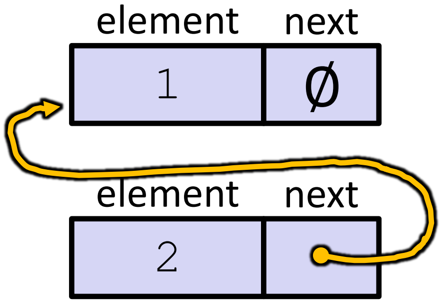
Arrow points to *next* instruction.

```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 0;
}
```

push_list.c

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

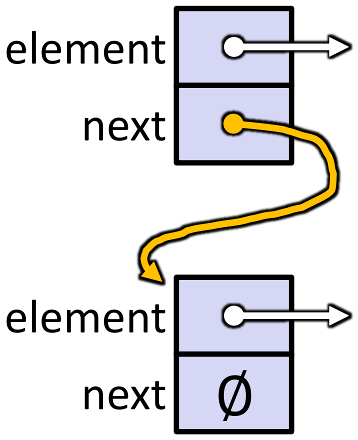
```
bash$ gcc -Wall -g -o
push_list push_list.c

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



A Generic Linked List

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

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

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



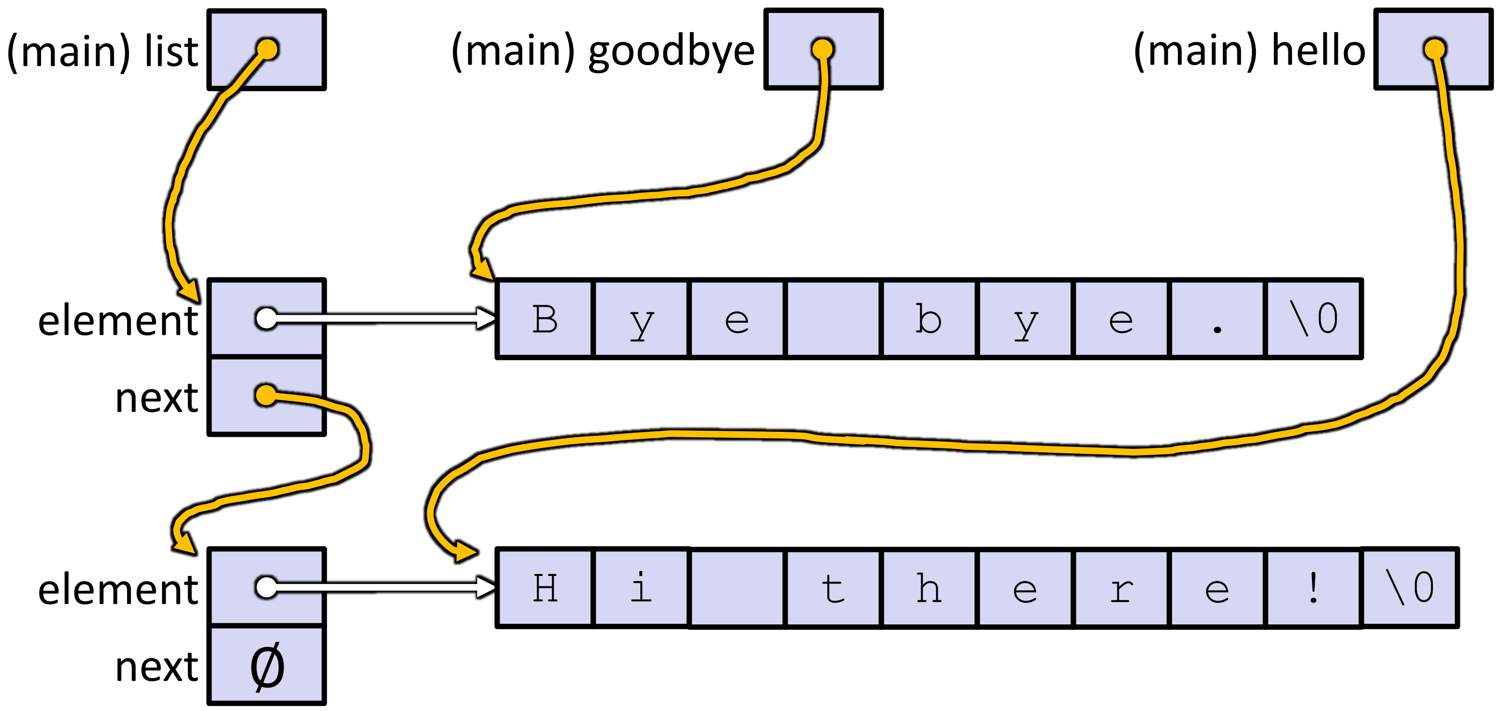
Using a Generic Linked List

- Type casting needed to deal with `void*` (raw address)
  - Before pushing, need to convert to `void*`
  - Convert back to data type when accessing

```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 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

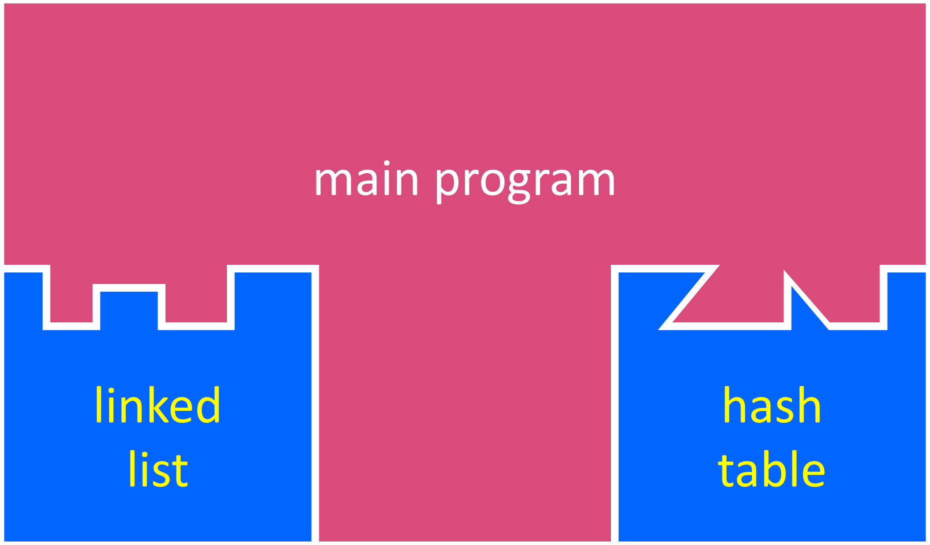
- Implementing Data Structures in C
- **Multi-file C Programs**
  - **C Preprocessor Intro**

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 `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
  - 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"
  - 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;
```

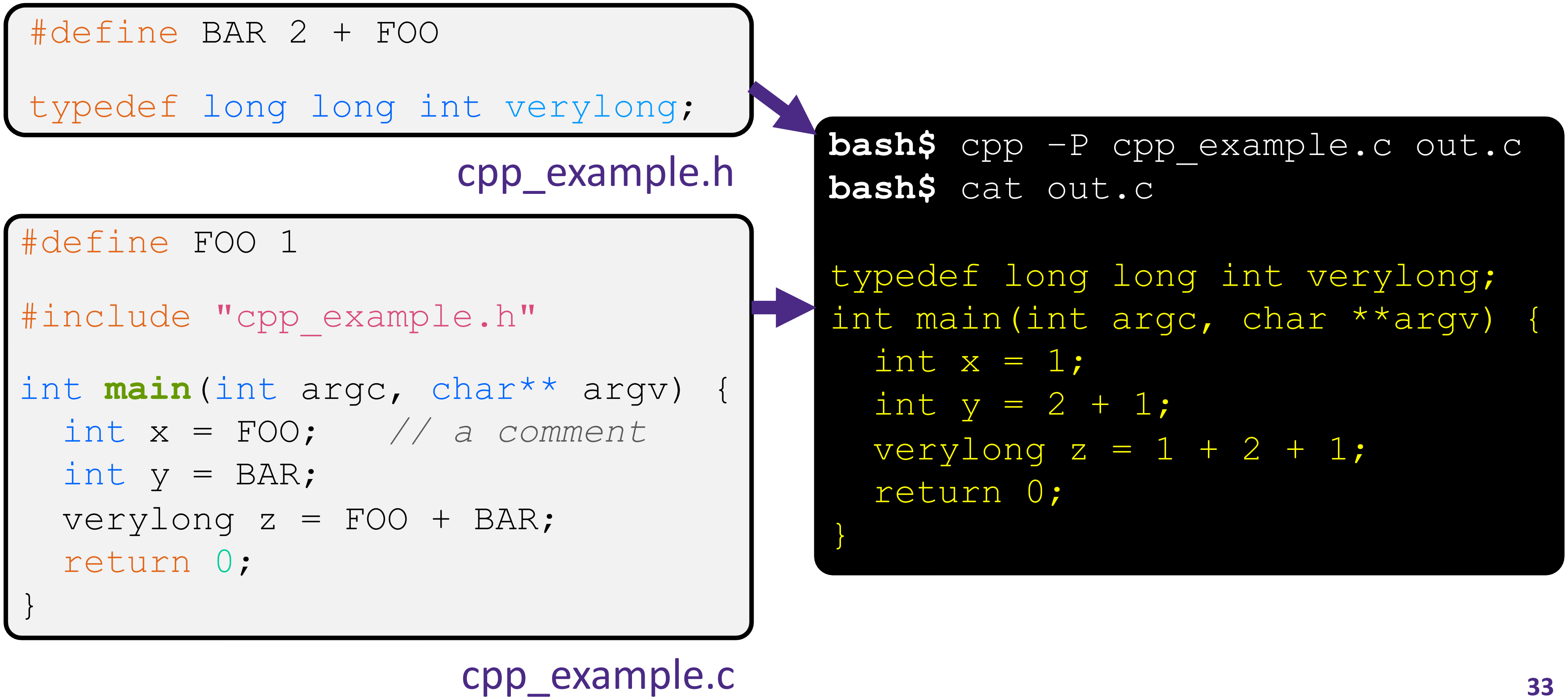
cpp_example.h

```
#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



```c
#define BAR 2 + FOO

typedef long long int verylong;
```

cpp_example.h

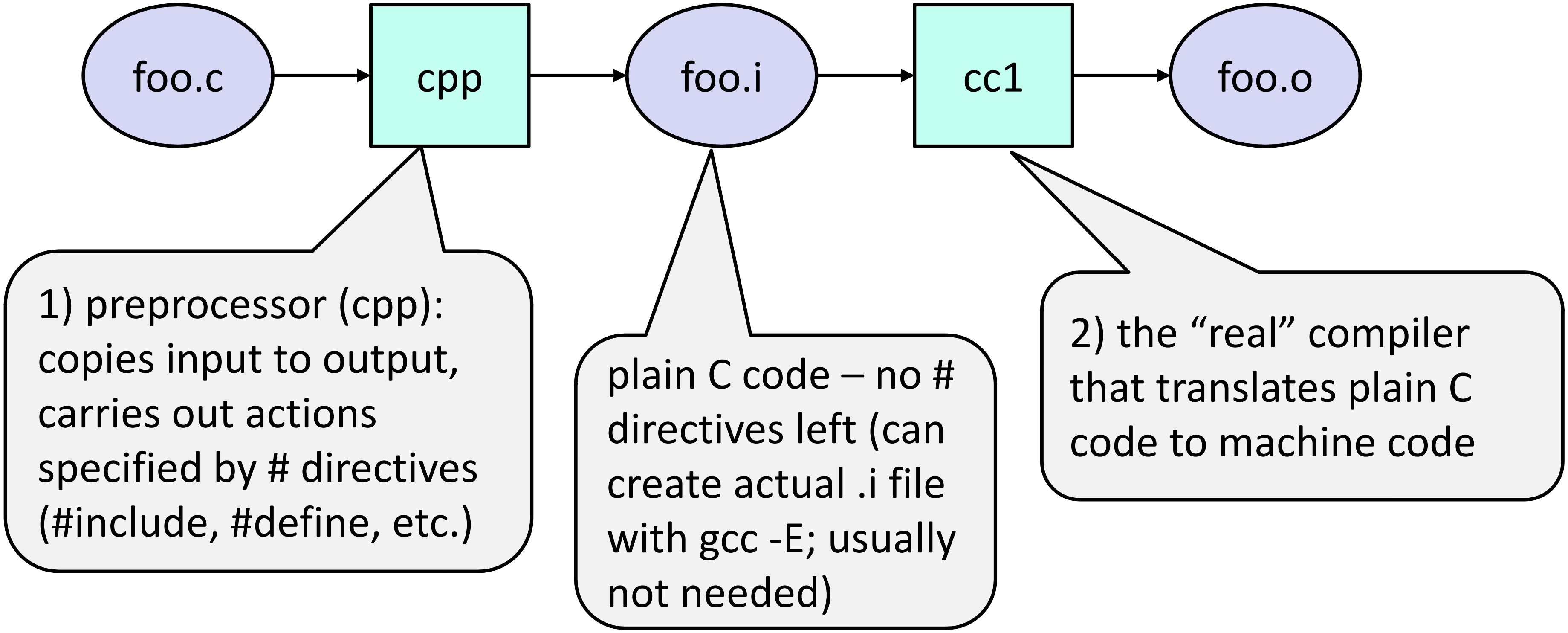
```c
#define FOO 1

#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 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):



1) preprocessor (cpp): copies input to output, carries out actions specified by # directives (#include, #define, etc.)

plain C code – no # directives left (can create actual .i file with gcc -E; usually not needed)

2) the “real” compiler that translates plain C code to machine code

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:
  - 1/2) 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 `double`s
  - Includes functions to:
    - add, subtract, multiply, and divide complex numbers
  - Implement a test driver in `test_complex.c`
    - Contains `main()`