

CSE 333

Lecture 5 - Data Structures



Administrivia

HW:

- HW1 due on Friday; get going NOW if you haven' t yet!!

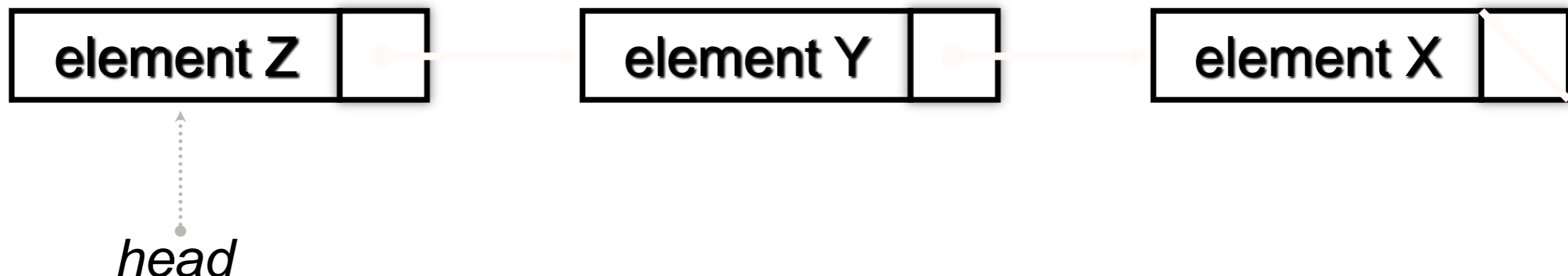
Today's topics:

- implementing data structures in C
- multi-file C programs
- brief intro to the C preprocessor

Let's build a simple linked list

You've seen a linked list in CSE143

- each node in a linked list contains:
 - some element as its payload
 - a pointer to the next node in the linked list
- the last node in the list contains a NULL pointer (or some other indication that it is the last node)



Linked list node

Let's represent a linked list node with a struct

- and, for now, assume each element is an int

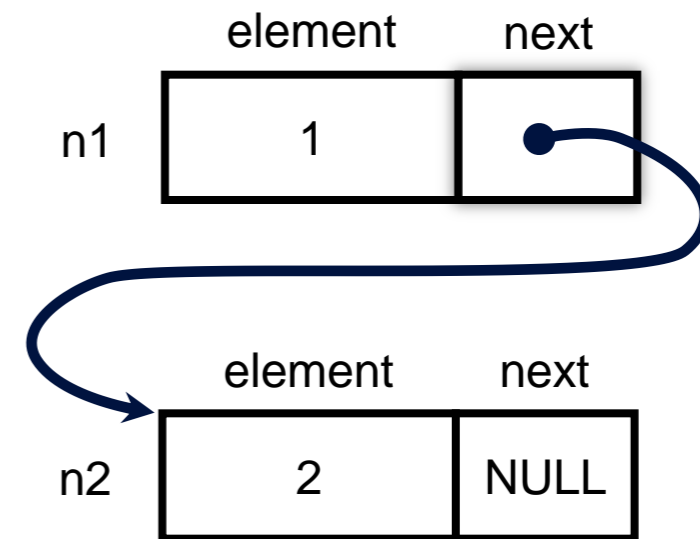
```
#include <stdio.h>

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

int main(int argc, char **argv) {
    Node n1, n2;

    n2.element = 2;
    n2.next = NULL;
    n1.element = 1;
    n1.next = &n2;
    return 0;
}
```

manual_list.c



PushOntoList

push_list.c

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

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

NULL

PushOntoList

push_list.c

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

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

NULL



PushOntoList

push_list.c

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

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

NULL

(*POL*) head

NULL

(*POL*) e

1

(*POL*) n

???

PushOntoList

push_list.c

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

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

NULL

(POL) head

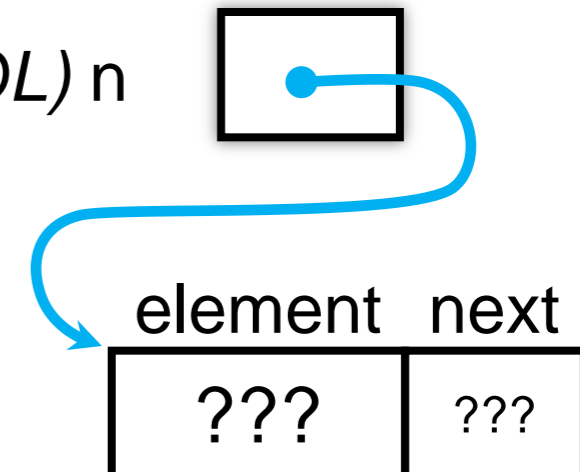
NULL

(POL) e

1

(POL) n

•



PushOntoList

push_list.c

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

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

NULL

(POL) head

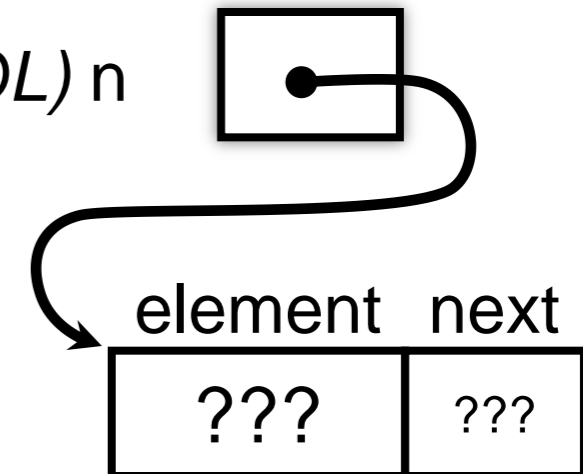
NULL

(POL) e

1

(POL) n

●



PushOntoList

push_list.c

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

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

NULL

(POL) head

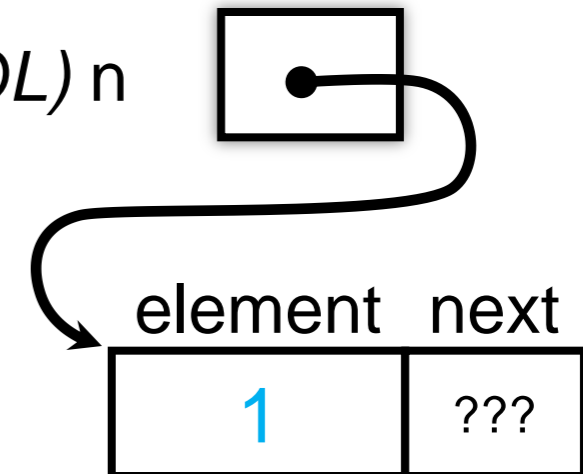
NULL

(POL) e

1

(POL) n

●



PushOntoList

push_list.c

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

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

NULL

(POL) head

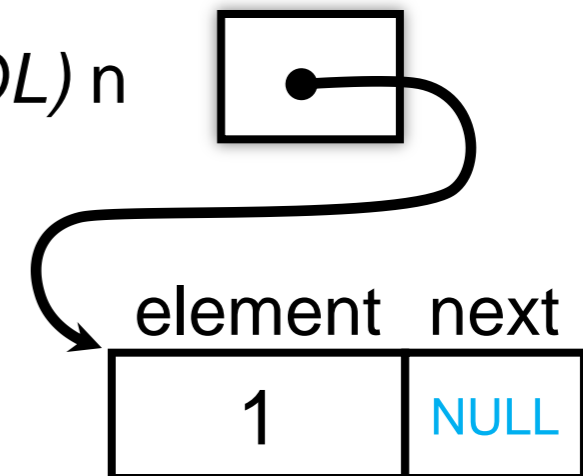
NULL

(POL) e

1

(POL) n

●



PushOntoList

push_list.c

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

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

Node *Push(Node *head, int e) {
    Node *n = (Node *) malloc(sizeof(Node));

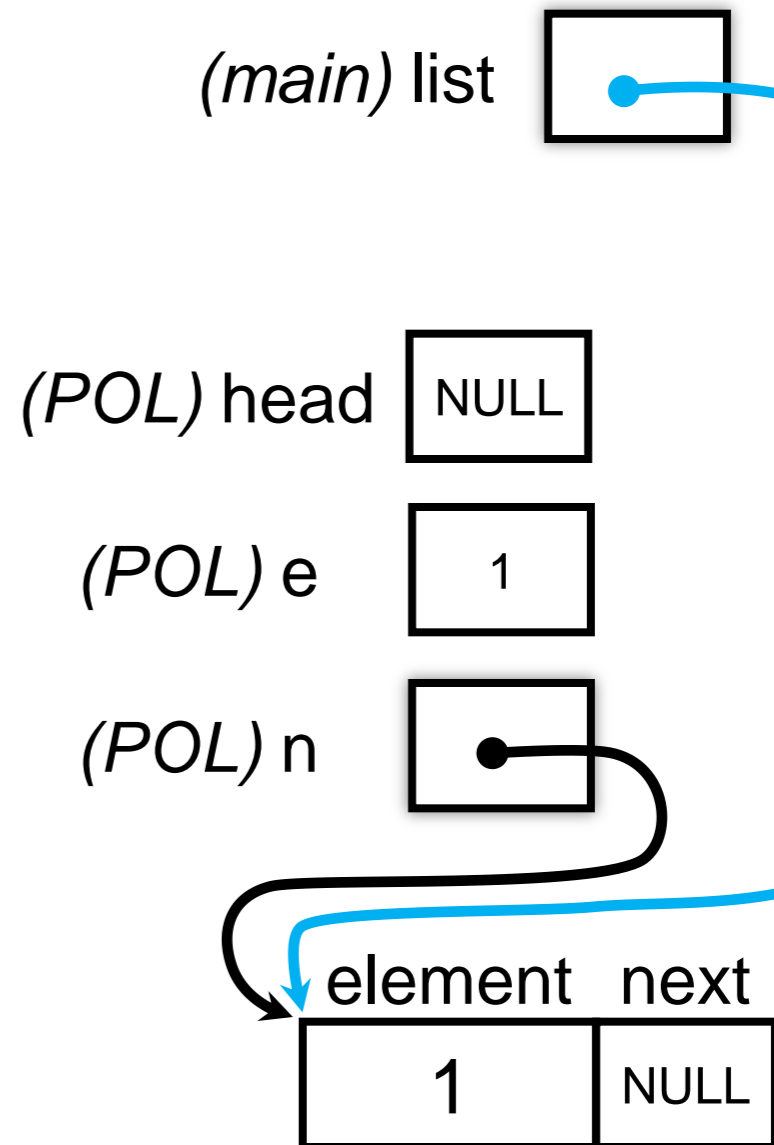
    assert(n != NULL); // crashes if false
    n->element = e;
    n->next = head;

    return n;
}

int main(int argc, char **argv) {
    Node *list = NULL;

    list = Push(list, 1);
    list = Push(list, 2);

    return 0;
}
```



PushOntoList

push_list.c

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

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

Node *Push(Node *head, int e) {
    Node *n = (Node *) malloc(sizeof(Node));

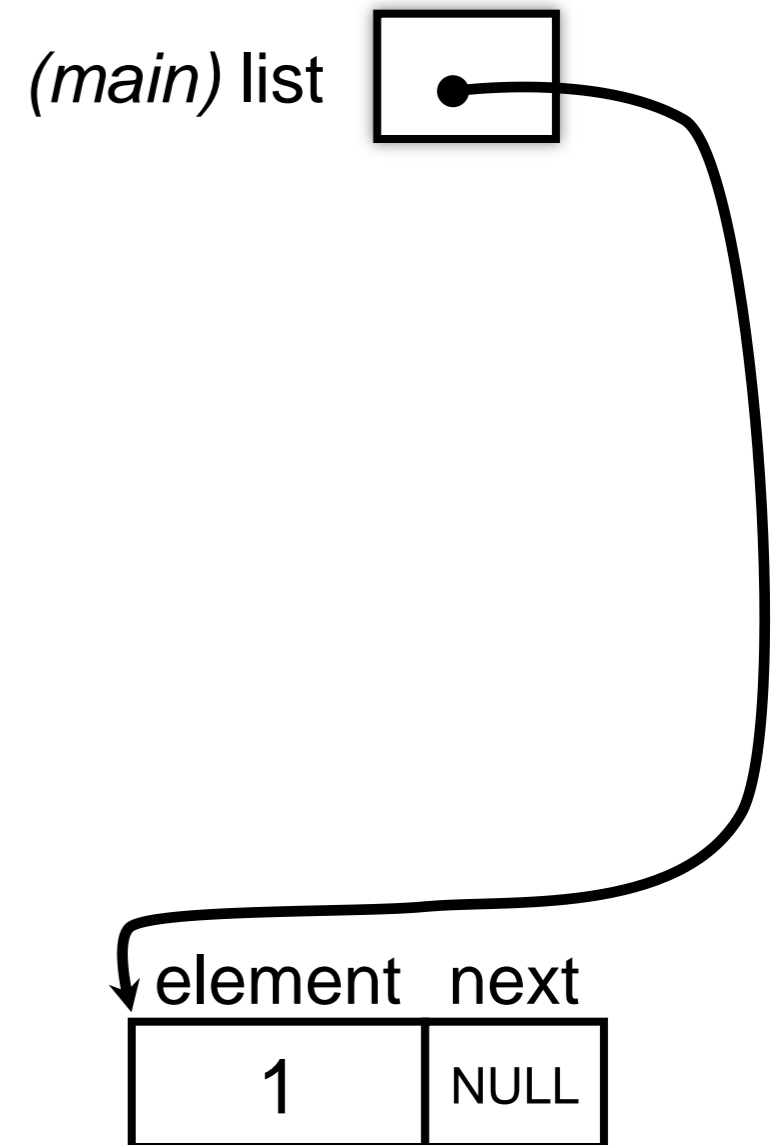
    assert(n != NULL); // crashes if false
    n->element = e;
    n->next = head;

    return n;
}

int main(int argc, char **argv) {
    Node *list = NULL;

    list = Push(list, 1);
    list = Push(list, 2);

    return 0;
}
```



PushOntoList

push_list.c

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

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

Node *Push(Node *head, int e) {
    Node *n = (Node *) malloc(sizeof(Node));

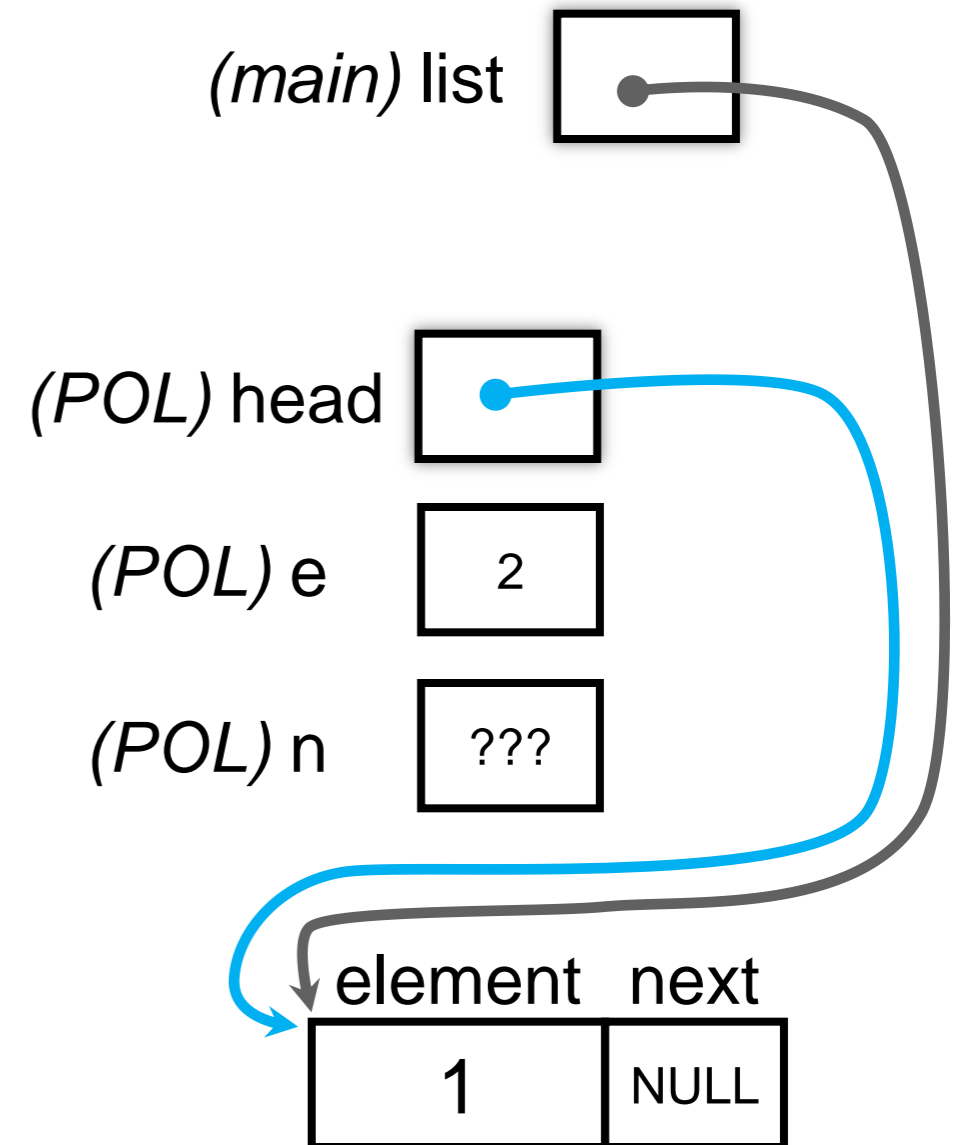
    assert(n != NULL); // crashes if false
    n->element = e;
    n->next = head;

    return n;
}

int main(int argc, char **argv) {
    Node *list = NULL;

    list = Push(list, 1);
    list = Push(list, 2);

    return 0;
}
```



PushOntoList

push_list.c

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

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

Node *Push(Node *head, int e) {
    Node *n = (Node *) malloc(sizeof(Node));

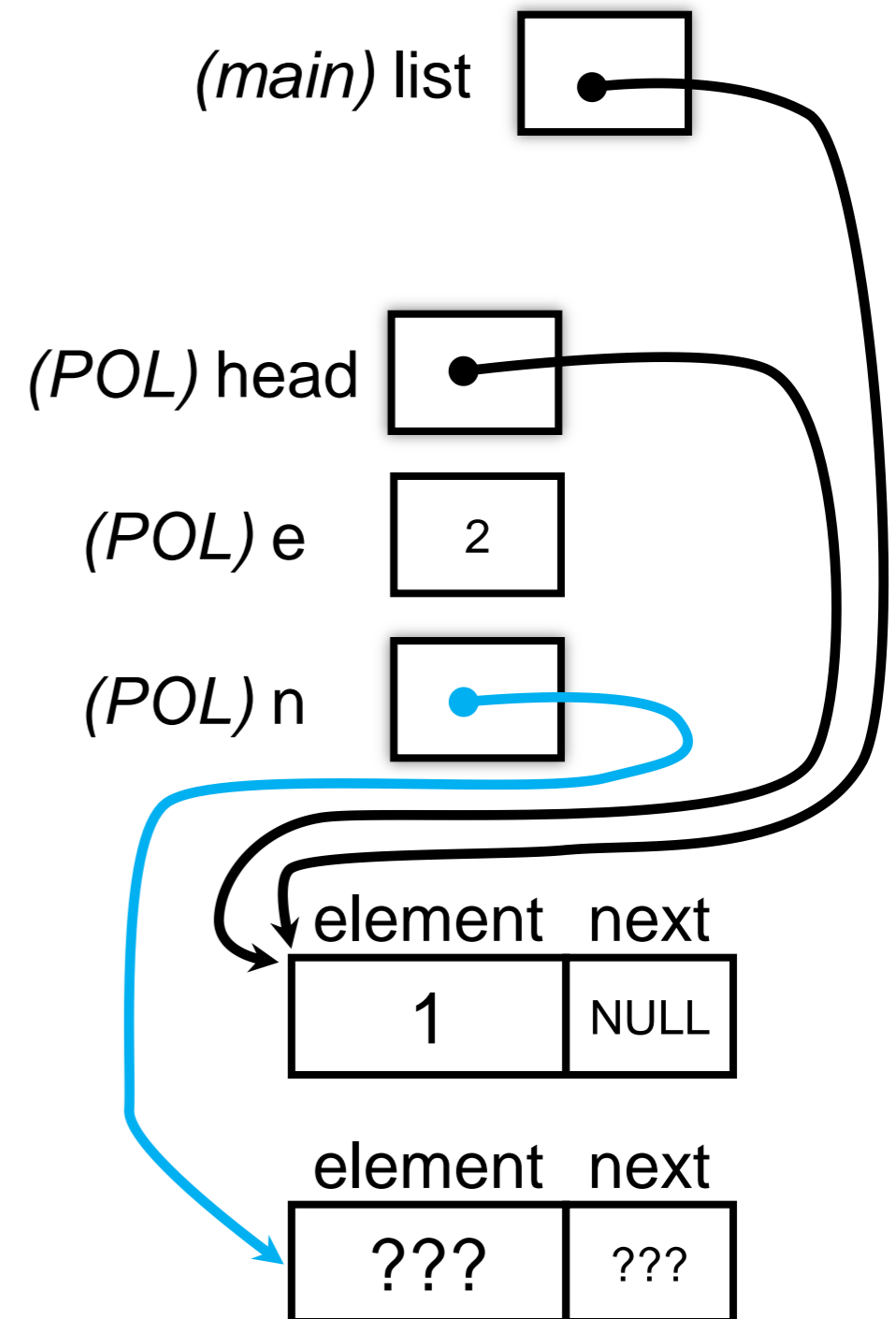
    assert(n != NULL); // crashes if false
    n->element = e;
    n->next = head;

    return n;
}

int main(int argc, char **argv) {
    Node *list = NULL;

    list = Push(list, 1);
    list = Push(list, 2);

    return 0;
}
```



PushOntoList

push_list.c

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

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

Node *Push(Node *head, int e) {
    Node *n = (Node *) malloc(sizeof(Node));

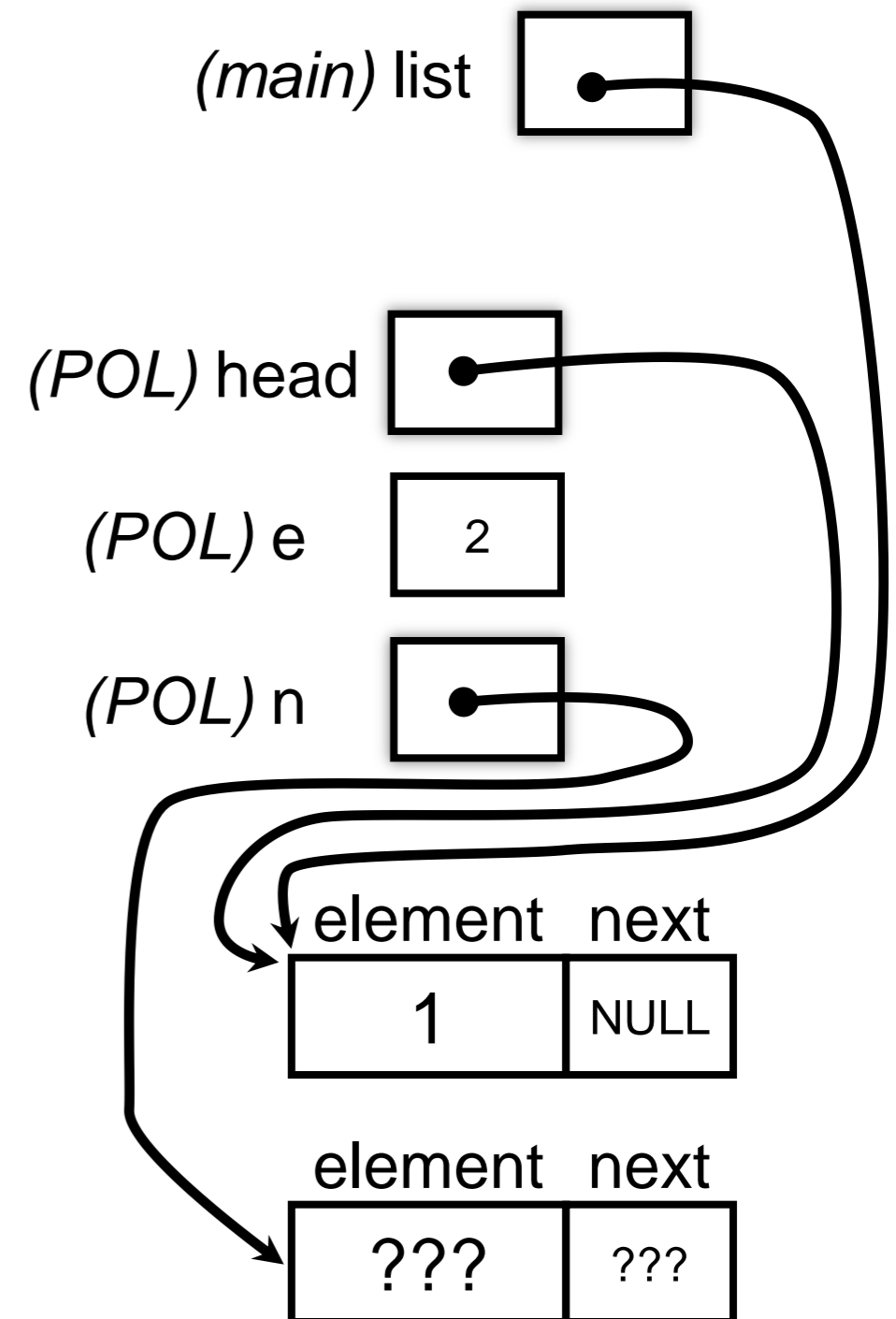
    assert(n != NULL); // crashes if false
    n->element = e;
    n->next = head;

    return n;
}

int main(int argc, char **argv) {
    Node *list = NULL;

    list = Push(list, 1);
    list = Push(list, 2);

    return 0;
}
```



PushOntoList

push_list.c

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

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

Node *Push(Node *head, int e) {
    Node *n = (Node *) malloc(sizeof(Node));

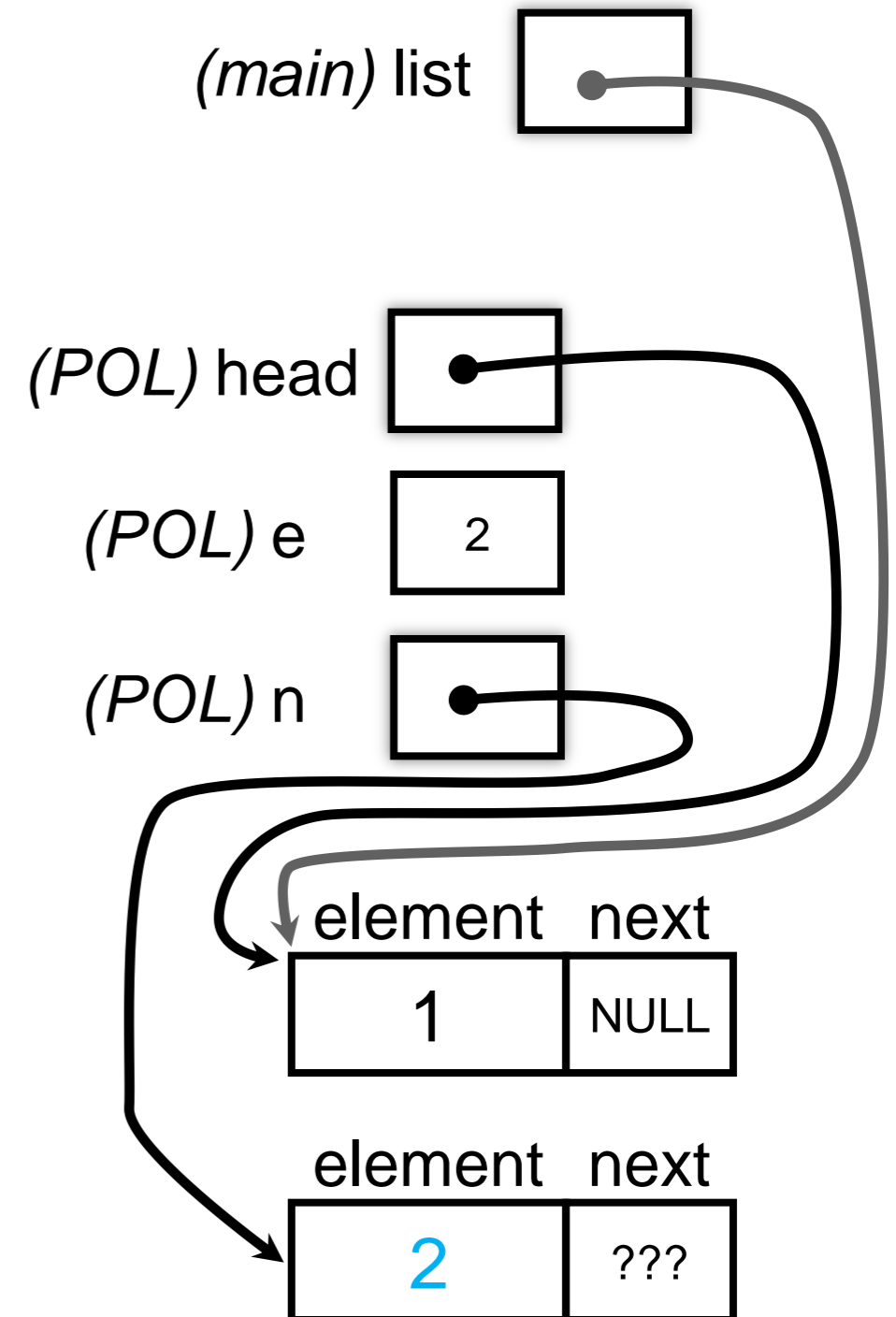
    assert(n != NULL); // crashes if false
    n->element = e;
    n->next = head;

    return n;
}

int main(int argc, char **argv) {
    Node *list = NULL;

    list = Push(list, 1);
    list = Push(list, 2);

    return 0;
}
```



PushOntoList

push_list.c

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

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

Node *Push(Node *head, int e) {
    Node *n = (Node *) malloc(sizeof(Node));

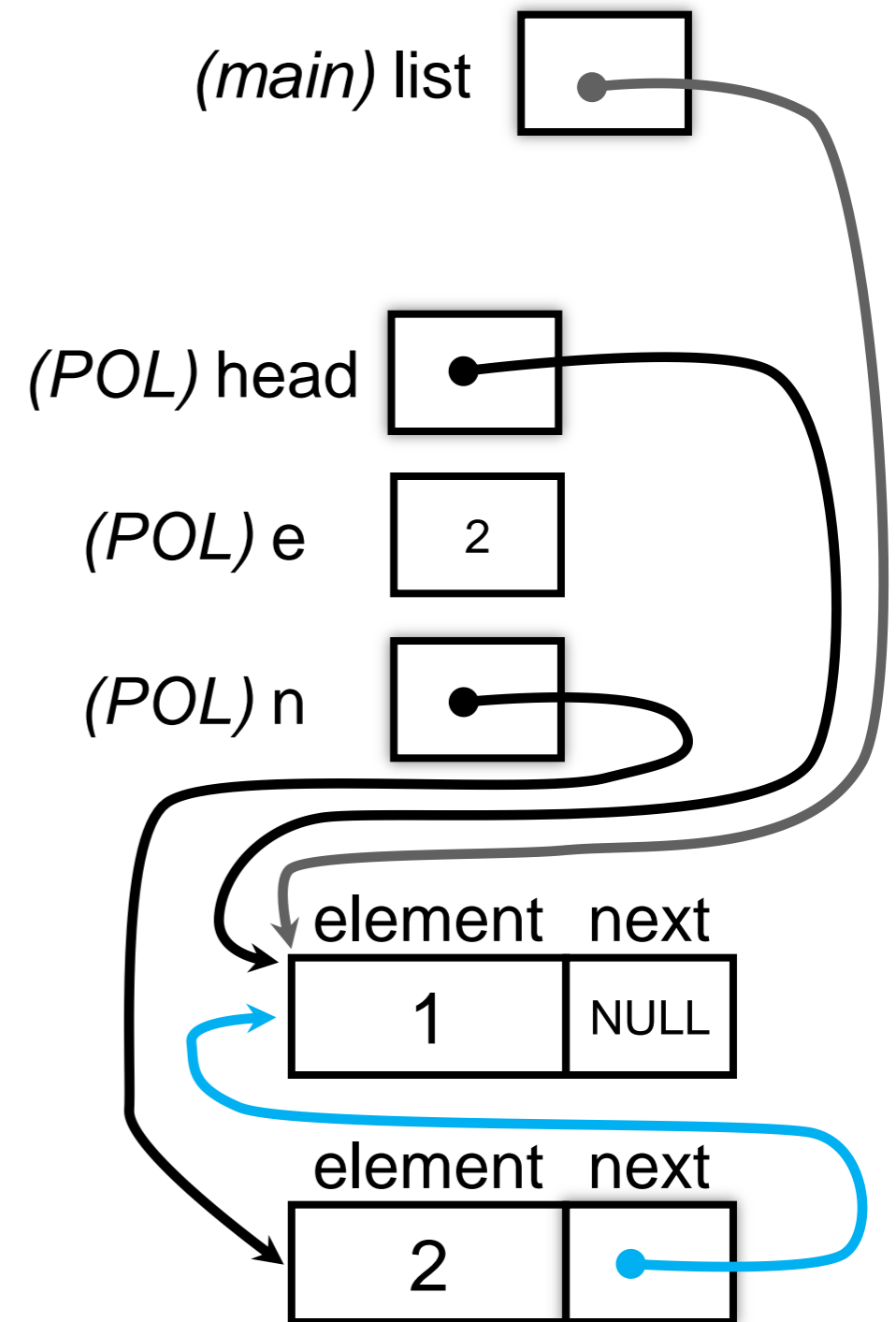
    assert(n != NULL); // crashes if false
    n->element = e;
    n->next = head;

    return n;
}

int main(int argc, char **argv) {
    Node *list = NULL;

    list = Push(list, 1);
    list = Push(list, 2);

    return 0;
}
```



PushOntoList

push_list.c

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

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

Node *Push(Node *head, int e) {
    Node *n = (Node *) malloc(sizeof(Node));

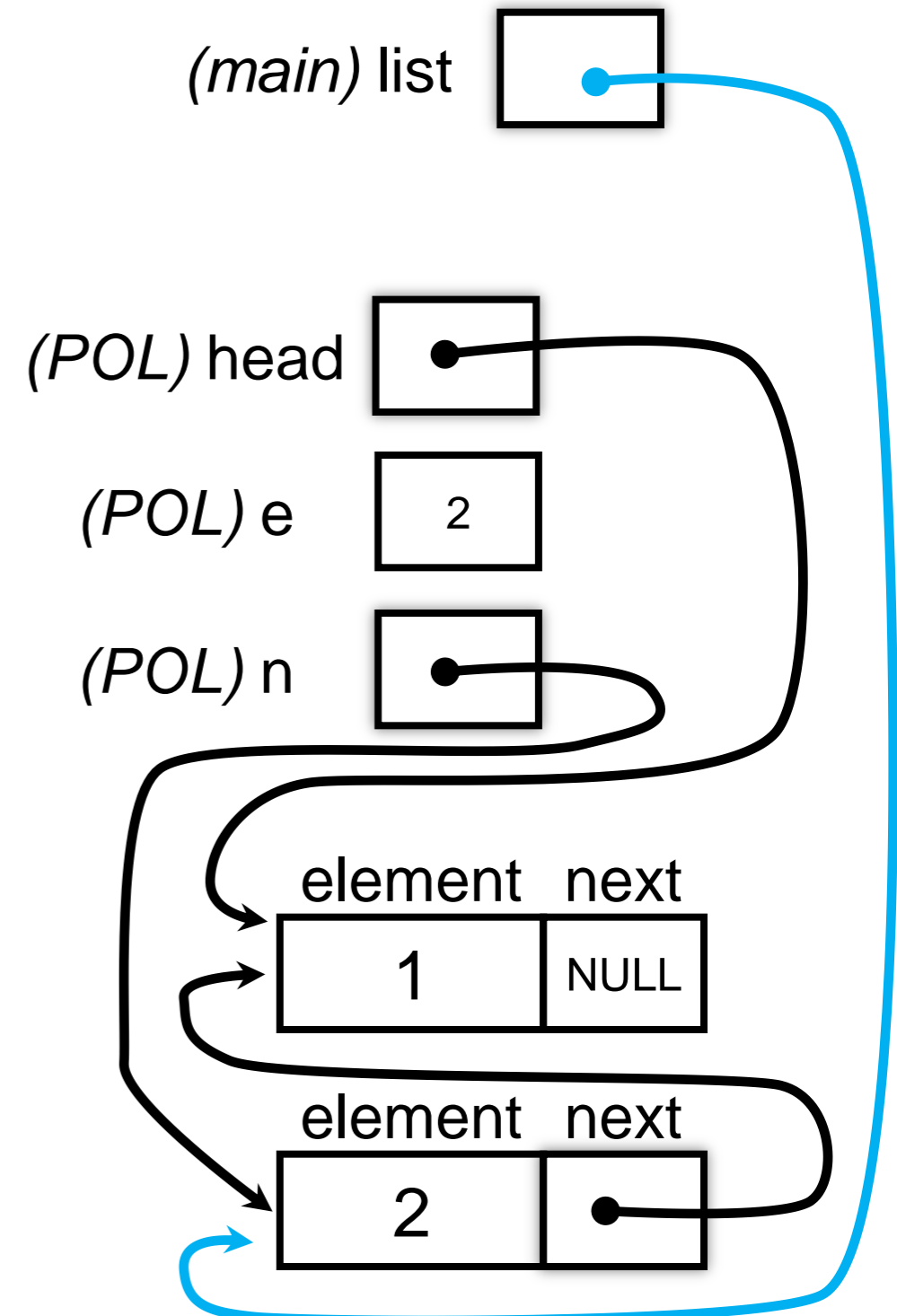
    assert(n != NULL); // crashes if false
    n->element = e;
    n->next = head;

    return n;
}

int main(int argc, char **argv) {
    Node *list = NULL;

    list = Push(list, 1);
    list = Push(list, 2);

    return 0;
}
```



PushOntoList

push_list.c

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

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

Node *Push(Node *head, int e) {
    Node *n = (Node *) malloc(sizeof(Node));

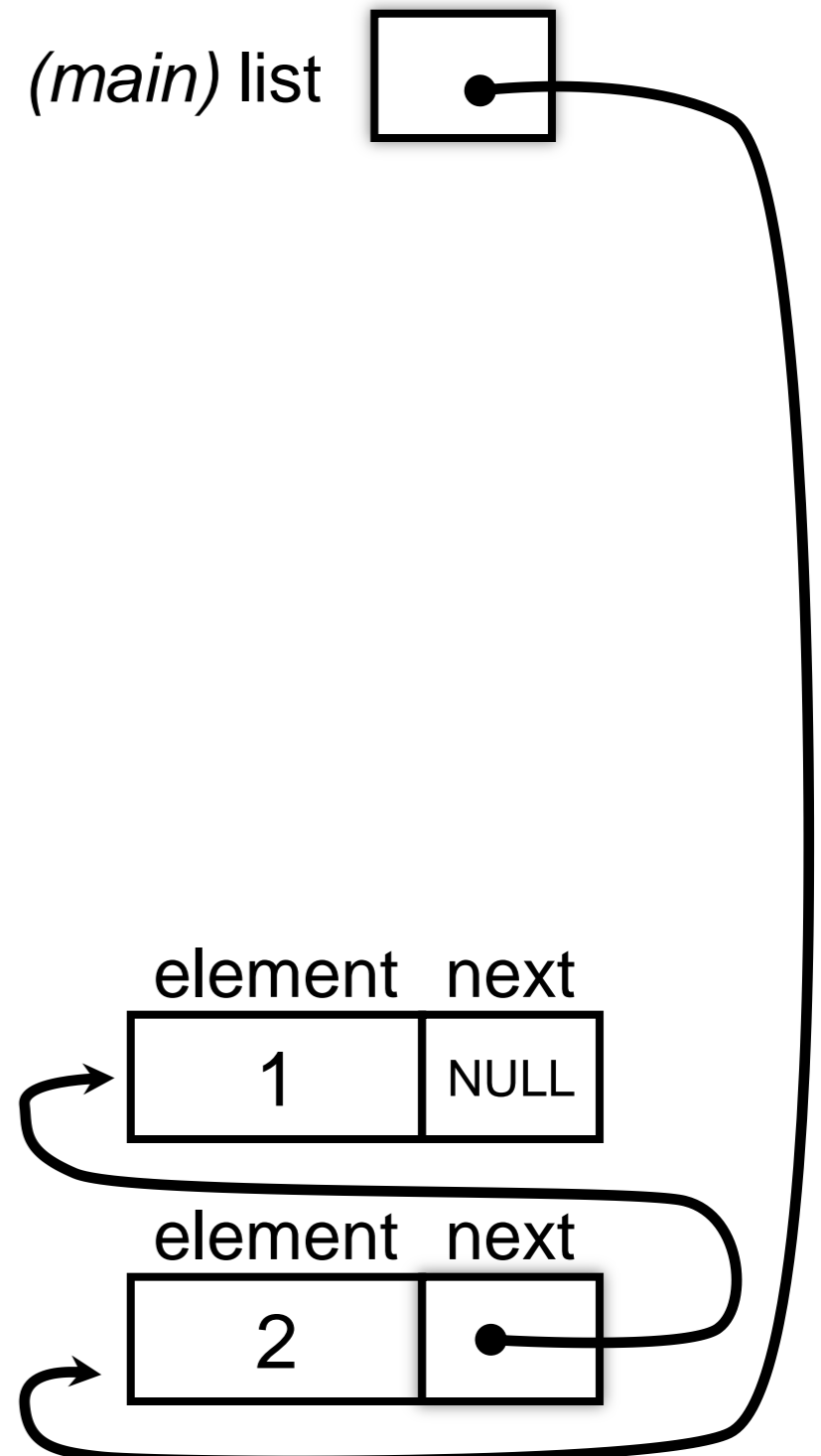
    assert(n != NULL); // crashes if false
    n->element = e;
    n->next = head;

    return n;
}

int main(int argc, char **argv) {
    Node *list = NULL;

    list = Push(list, 1);
    list = Push(list, 2);

    return 0;
}
```



PushOntoList

push_list.c

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

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;

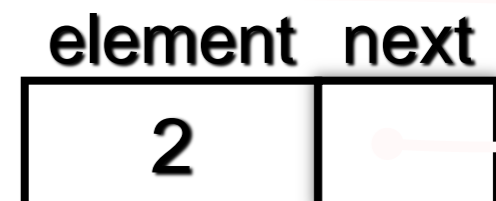
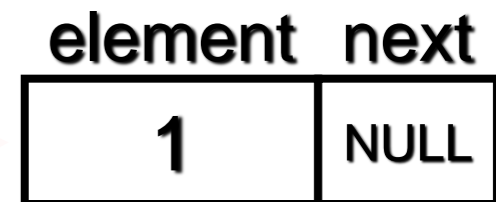
    list = Push(list, 1);
    list = Push(list, 2);

    return 0;
}
```

a (benign) leak!!
try running with valgrind:

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

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



A generic linked list: `void*`

Previously, our linked list elements were of type `int`

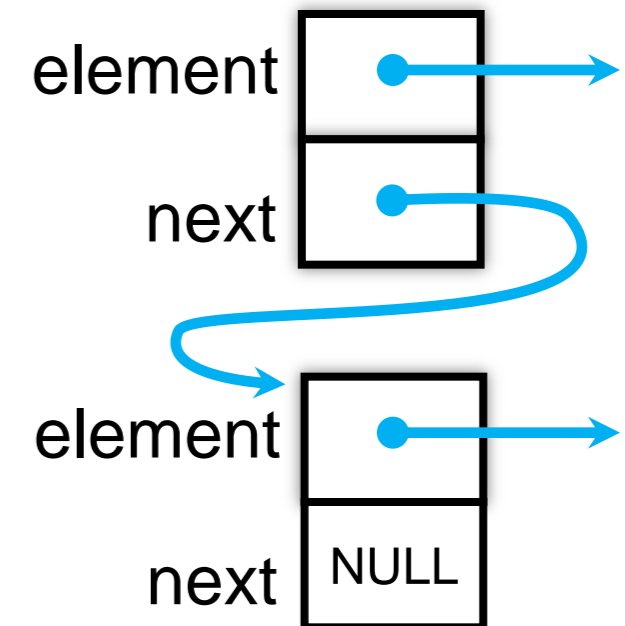
- what if we want to let our customer decide the element type?
- idea: let them push a generic pointer -- i.e., a (`void *`)

```
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

To use it, customers will need to use type casting

- convert their data type to a (void *) before pushing
- convert from a (void *) back to their data type when

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

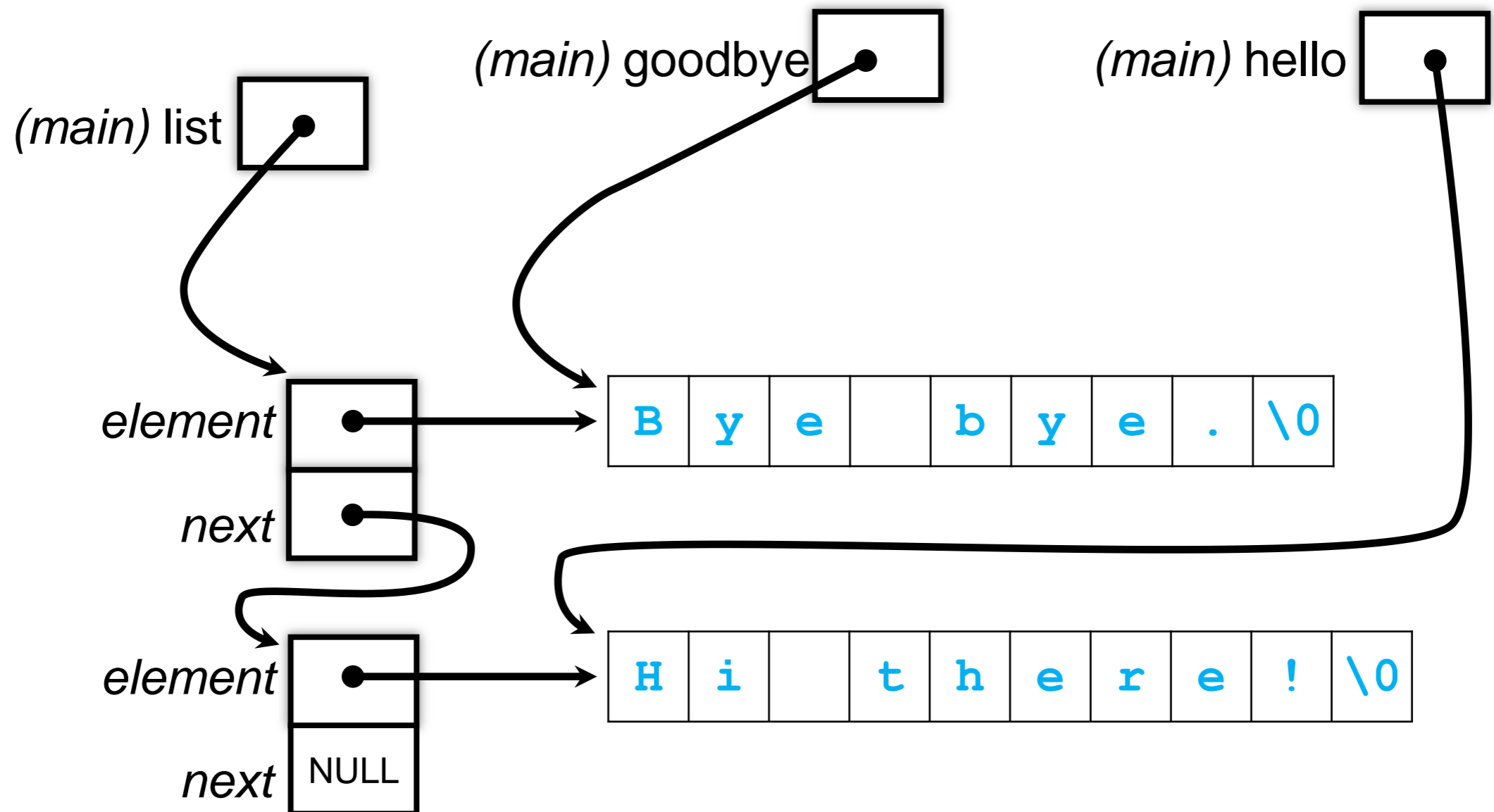
Node *Push(Node *head, void *e); // assume last slide's code

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;
}
```


Using a generic linked list

Results in:

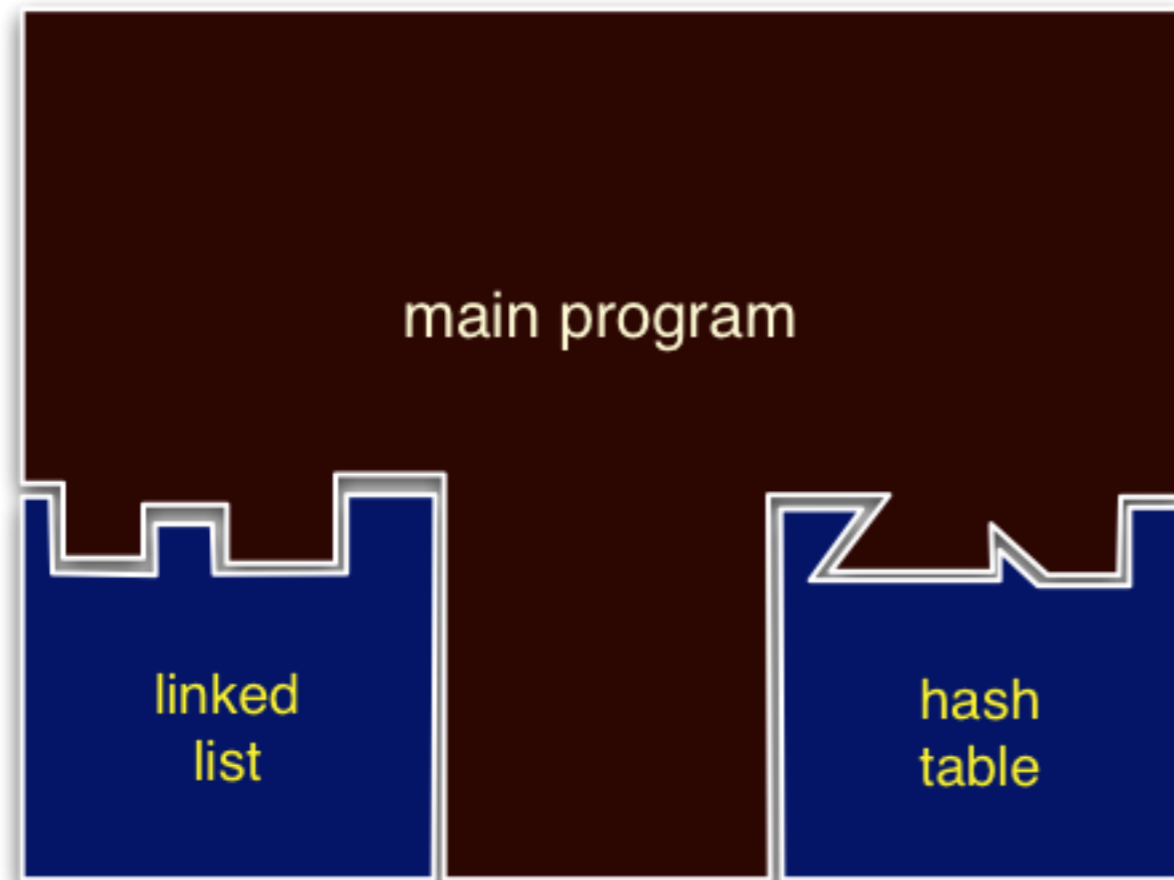


Multi-file C programs

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, 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

- typically the filename ends with `.h`
- holds the declarations (of variables, types, and functions) that make up the (public) interface to a module

the main idea

- **every** `name.c` intended to be a module has a `name.h`
- `name.h` declares the interface to that module
- other modules that want to use `name` will `#include name.h`
 - and 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 never contain definitions, only declarations
- .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
 - Why?
- **Never #include a .c file** -- only #include .h files
- Any .c file with an associated .h file should be able to be compiled into a .o file

#include and the C preprocessor

The C preprocessor (`cpp`) transforms your source code before the compiler runs

- transforms your original C source code into transformed C source code
- processes the directives it finds in your code (*#something*)
 - `#include "ll.h"` -- replaces with post-processed content of ll.h
 - `#define PI 3.1415` -- defines a symbol, replaces later occurrences
 - and there are several others we'll see soon...
- run on your behalf by gcc during compilation

Example

```
#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

Let's manually run the pre-processor on `cpp_example.c`:

- `cpp` is the preprocessor
- “*-P*” suppresses some extra debugging annotations

```
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 Just Happened?

```
#define FOO 1  
  
#include "cpp_example.h"  
  
#define BAR 2 + FOO  
  
typedef long long int verylong;  
  
int main(int argc, char **argv) {  
    int x = FOO 1;    // a comment  
    int y = BAR 2 + FOO 1;  
    verylong z = FOO 1 + BAR 2 + FOO 1;  
    return 0;  
}
```


Program that uses 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, hi);
    list = Push(list, bye);

    return 0;
}
```

example_ll_customer.c

Compiling the program

Four steps:

- compile *example_ll_customer.c* into an object file
- compile *ll.c* into an object file
- link *ll.o*, *example_ll_customer.o* into an executable
- test, debug, rinse, repeat

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

Payload: 'yo!'
Payload: 'goodbye'
Payload: 'hello'
bash$ valgrind --leak-check=full ./example_customer
...etc.
```

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
 - but each element in the list 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

Exercise 2

Implement and test a binary search tree

- http://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(), tests out your BST

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()

See you on Wednesday!