

# Data Structures

## CSE 333 Winter 2020

**Instructor:** Justin Hsia

**Teaching Assistants:**

Andrew Hu

Austin Chan

Brennan Stein

Cheng Ni

Cosmo Wang

Diya Joy

Guramrit Singh

Mengqi Chen

Pat Kosakanchit

Rehaan Bhimar

Renshu Gu

Travis McGaha

Zachary Keyes

# Administrivia

- ❖ ex0 grades released, ex3 released today
  - Regrade requests: open 24 hr after, close 72 hr after release
- ❖ We *highly* recommend doing the extra exercises
  - Also, can Google for “C pointer exercises”
  - You MUST master pointers quickly, or you’ll have trouble the rest of the course (including hw1)
- ❖ hw0 due tonight *before* 11:59 pm (and 0 seconds)
  - Git: add/commit/push, then tag with `hw0-final`, then push tag
    - Then clone your repo somewhere totally different and do `git checkout hw0-final` and verify that all is well

# Administrivia

- ❖ hw1 due Thursday, 1/23
  - You **may not** modify interfaces ( .h files)
  - But **do** read the interfaces while you're implementing them(!)
  - New this quarter: maintain a bug journal
  - Suggestion: look at `example_program_{ll|ht}.c` for typical usage of lists and hash tables
- ❖ GitLab repo usage
  - **Commit things regularly** (not 1 massive commit on due date)
    - Newly completed units of work, milestones, project parts, etc.
  - Provides backup – can retrieve old versions of files 😊
  - Don't push .o and executable files or other build products (not portable)

# Lecture Outline

- ❖ **Heap-allocated Memory**
  - **Memory leaks**
- ❖ `structs` and `typedef`
- ❖ Implementing Data Structures in C

# Dynamic Allocation Function Review

- ❖ malloc: `var = (type*) malloc (size in bytes)`
  - Returns a `void*` to uninitialized heap memory
    - Returns `NULL` to indicate failure
  - Should use `sizeof ()` and explicit cast; error check return value
- ❖ free: `free (pointer) ;`
  - Pointer *must* point to the first byte of heap-allocated memory
    - Will do nothing if passed `NULL`
  - Undefined behavior (often program crash) otherwise
  - Doesn't change the value of the pointer

# Polling Question

- ❖ Which line below is first *guaranteed* to cause either an error or undefined behavior?
  - Vote at <http://PollEv.com/justinh>

- A. Line 1
- B. Line 4
- C. Line 6
- D. Line 7
- E. We're lost...

```
int main(int argc, char** argv) {  
    int a[2];  
    int* b = malloc(2*sizeof(int));  
    int* c;  
  
1    a[2] = 5;  
2    b[0] += 2;  
3    c = b+3;  
4    free(&(a[0]));  
5    free(b);  
6    free(b);  
7    b[0] = 5;  
  
    return EXIT_SUCCESS;  
}
```

# Memory Corruption

- ❖ There are all sorts of ways to corrupt memory in C

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

int main(int argc, char** argv) {
    int a[2];
    int* b = malloc(2*sizeof(int));
    int* c;

    a[2] = 5;    // assign past the end of an array
    b[0] += 2;   // assume malloc zeros out memory
    c = b+3;    // mess up your pointer arithmetic
    free(&(a[0])); // free something not malloc'ed
    free(b);
    free(b);    // double-free the same block
    b[0] = 5;   // use a freed pointer

    // any many more!
    return EXIT_SUCCESS;
}
```

memcorrupt.c

# Memory Leak

- ❖ A **memory leak** occurs when code fails to deallocate dynamically-allocated memory that is no longer used
  - *e.g.* forget to **free** malloc-ed block, lose/change pointer to malloc-ed block
- ❖ What happens: program's VM footprint will keep growing
  - This might be OK for *short-lived* program, since all memory is deallocated when program ends
  - Usually has bad repercussions for *long-lived* programs
    - Might slow down over time (*e.g.* lead to VM thrashing)
    - Might exhaust all available memory and crash
    - Other programs might get starved of memory



# Lecture Outline

- ❖ Heap-allocated Memory
  - `malloc()` and `free()`
  - Memory leaks
- ❖ **structs and typedef**
- ❖ Implementing Data Structures in C

# Structured Data

- ❖ A `struct` is a C datatype that contains a set of fields
  - Similar to a Java class, but with no methods or constructors
  - Useful for defining new structured types of data
  - Behave similarly to primitive variables
- ❖ Generic declaration:

```
struct tagname {  
    type1 name1;  
    ...  
    typeN nameN;  
};
```

```
// the following defines a new  
// structured datatype called  
// a "struct Point"  
struct Point {  
    float x, y;  
};  
  
// declare and initialize a  
// struct Point variable  
struct Point origin = {0.0, 0.0};
```

# Using structs

- ❖ Use “.” to refer to a field in a struct
- ❖ Use “->” to refer to a field from a struct pointer
  - Dereferences pointer first, then accesses field

```
struct Point {
    float x, y;
};

int main(int argc, char** argv) {
    struct Point p1 = {0.0, 0.0}; // p1 is stack allocated
    struct Point* p1_ptr = &p1;

    p1.x = 1.0;
    p1_ptr->y = 2.0; // equivalent to (*p1_ptr).y = 2.0;
    return EXIT_SUCCESS;
}
```

simplestruct.c

# Copy by Assignment

- ❖ You can assign the value of a struct from a struct of the same type – *this copies the entire contents!*

```
struct Point {
    float x, y;
};

int main(int argc, char** argv) {
    struct Point p1 = {0.0, 2.0};
    struct Point p2 = {4.0, 6.0};

    printf("p1: {%f,%f} p2: {%f,%f}\n", p1.x, p1.y, p2.x, p2.y);
    p2 = p1;
    printf("p1: {%f,%f} p2: {%f,%f}\n", p1.x, p1.y, p2.x, p2.y);
    return EXIT_SUCCESS;
}
```

structassign.c

# typedef

- ❖ Generic format: `typedef type name;`
- ❖ Allows you to define new data type *names/synonyms*
  - Both `type` and `name` are usable and refer to the same type
  - Be careful with pointers – `*` before name is part of type!

```
// make "superlong" a synonym for "unsigned long long"
typedef unsigned long long superlong;

// make "str" a synonym for "char*"
typedef char *str;

// make "Point" a synonym for "struct point_st { ... }"
// make "PointPtr" a synonym for "struct point_st*"
typedef struct point_st {
    superlong x;
    superlong y;
} Point, *PointPtr; // similar syntax to "int n, *p;"

Point origin = {0, 0};
```

# Dynamically-allocated Structs

- ❖ You can **malloc** and **free** structs, just like other data type
  - **sizeof** is particularly helpful here

```
// a complex number is a + bi
typedef struct complex_st {
    double real;    // real component
    double imag;   // imaginary component
} Complex, *ComplexPtr;

// note that ComplexPtr is equivalent to Complex*
ComplexPtr AllocComplex(double real, double imag) {
    Complex* retval = (Complex*) malloc(sizeof(Complex));
    if (retval != NULL) {
        retval->real = real;
        retval->imag = imag;
    }
    return retval;
}
```

complexstruct.c

# Structs as Arguments

- ❖ Structs are passed by value, like everything else in C
  - Entire struct is copied – where?
  - To manipulate a struct argument, pass a pointer instead

```
typedef struct point_st {                                     structarg.c
    int x, y;
} Point, *PointPtr;

void DoubleXBroken(Point p)    { p.x *= 2; }

void DoubleXWorks(PointPtr p) { p->x *= 2; }

int main(int argc, char** argv) {
    Point a = {1,1};
    DoubleXBroken(a);
    printf("( %d, %d) \n", a.x, a.y);    // prints: ( 1, 1 )
    DoubleXWorks(&a);
    printf("( %d, %d) \n", a.x, a.y);    // prints: ( 1, 1 )
    return EXIT_SUCCESS;
}
```

# Returning Structs

- ❖ Exact method of return depends on calling conventions
  - Often in `%rax` and `%rdx` for small structs
  - Often returned in memory for larger structs

```
// a complex number is a + bi
typedef struct complex_st {
    double real;    // real component
    double imag;   // imaginary component
} Complex, *ComplexPtr;

Complex MultiplyComplex(Complex x, Complex y) {
    Complex retval;

    retval.real = (x.real * y.real) - (x.imag * y.imag);
    retval.imag = (x.imag * y.real) - (x.real * y.imag);
    return retval; // returns a copy of retval
}
```

complexstruct.c





# Pass Copy of Struct or Pointer?

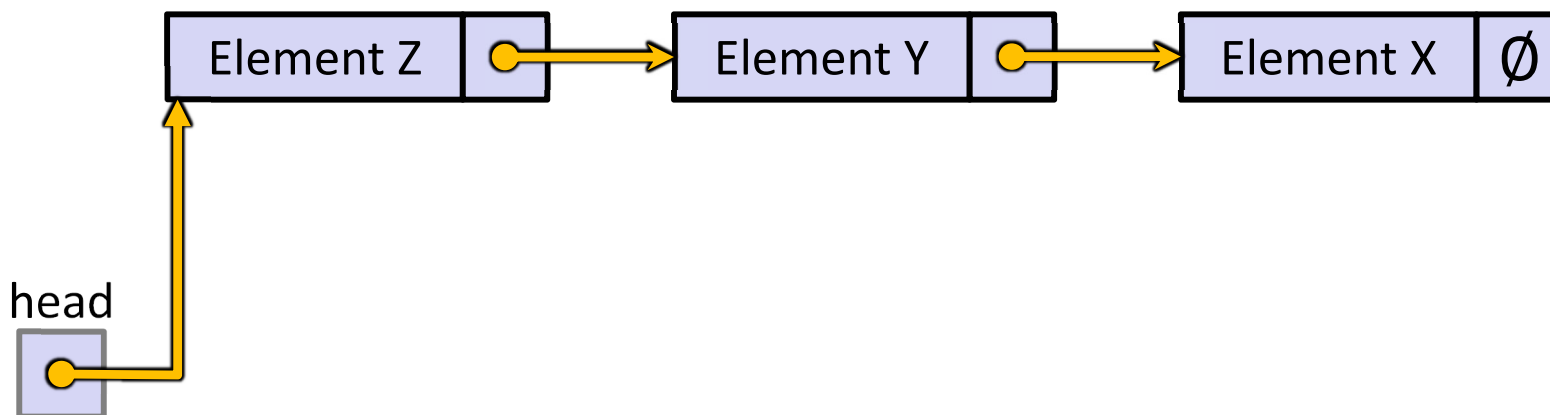
- ❖ Value passed: passing a pointer is cheaper and takes less space unless struct is small
- ❖ Field access: indirect accesses through pointers are a bit more expensive and can be harder for compiler to optimize
- ❖ For small structs (like `struct complex_st`), passing a copy of the struct can be faster and often preferred if function only reads data; for large structs use pointers

# Lecture Outline

- ❖ Heap-allocated Memory
  - `malloc()` and `free()`
  - Memory leaks
- ❖ `structs` and `typedef`
- ❖ **Generic Data Structures in C**

# 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

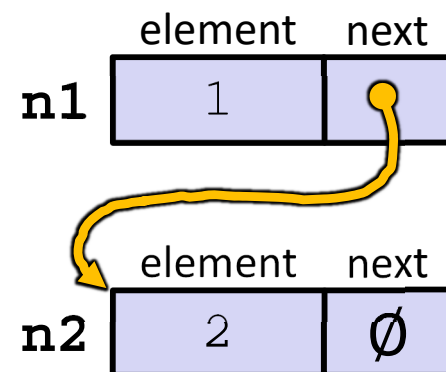
- ❖ Let's represent a linked list node with a struct
  - Assume each element is an `int32_t`

```
typedef struct node_st {
    int32_t 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



# Push Onto List

Arrow points to  
*next* instruction.

```
typedef struct node_st {
    int32_t element;
    struct node_st* next;
} Node;

Node* Push(Node* head, int32_t 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;
}
```

(main) list



push\_list.c

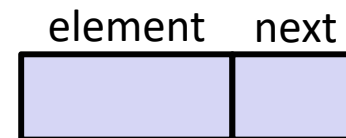
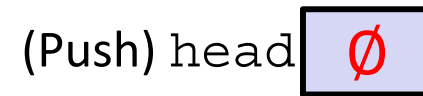
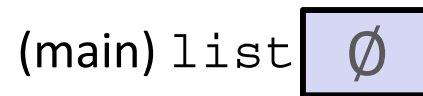
# Push Onto List

Arrow points to *next* instruction.

```
typedef struct node_st {
    int32_t element;
    struct node_st* next;
} Node;

Node* Push(Node* head, int32_t 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

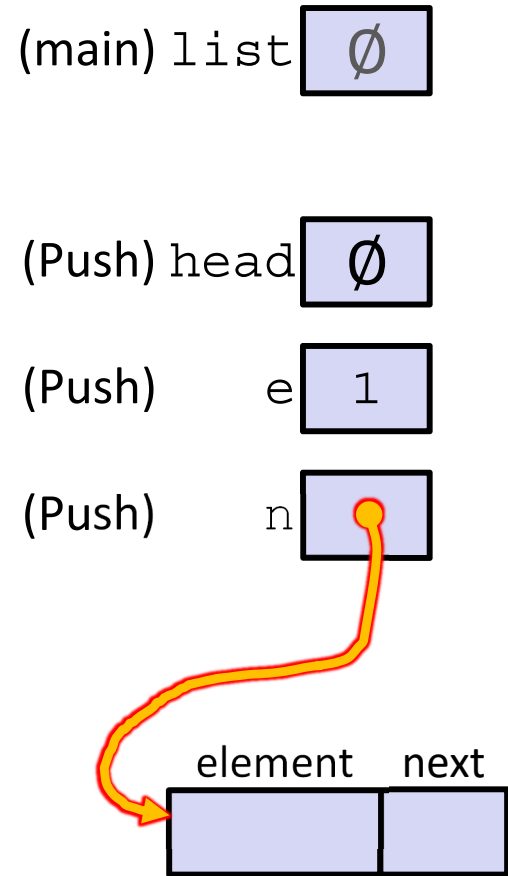
Arrow points to next instruction.

```

typedef struct node_st {
    int32_t element;
    struct node_st* next;
} Node;

Node* Push(Node* head, int32_t 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

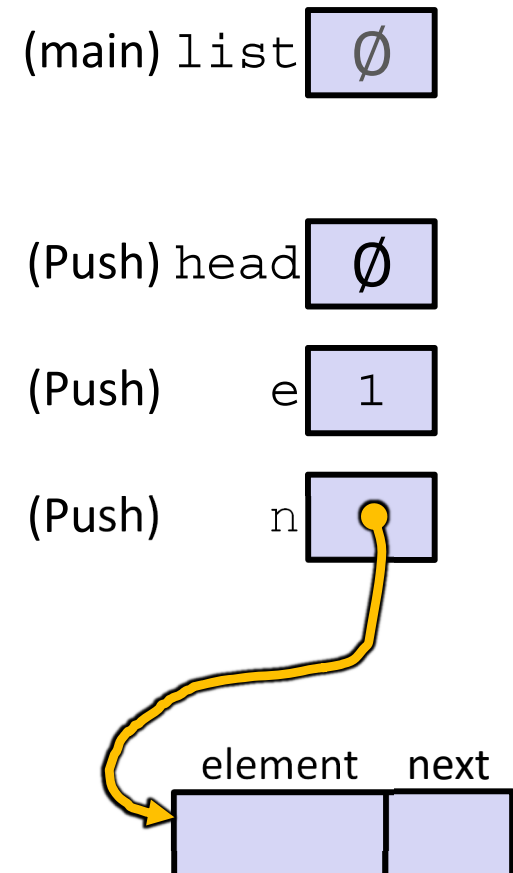
# Push Onto List

Arrow points to  
*next* instruction.

```
typedef struct node_st {
    int32_t element;
    struct node_st* next;
} Node;

Node* Push(Node* head, int32_t 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



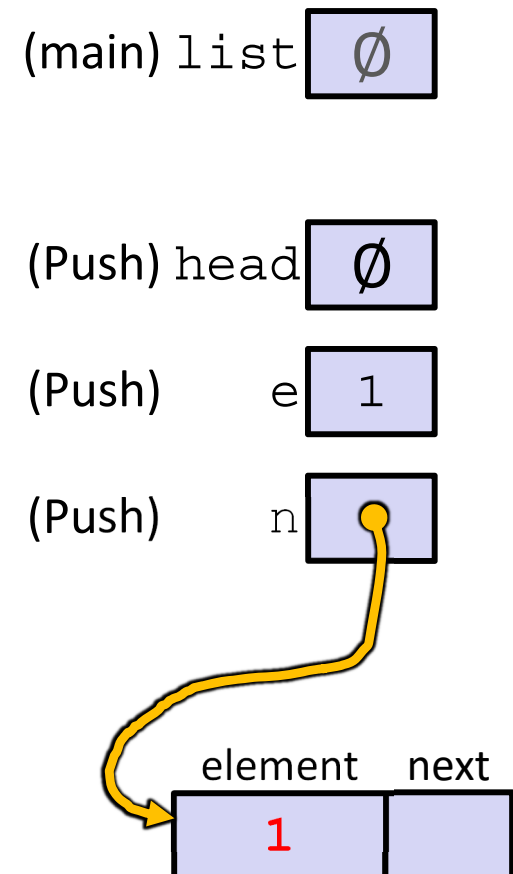
# Push Onto List

Arrow points to  
*next* instruction.

```
typedef struct node_st {
    int32_t element;
    struct node_st* next;
} Node;

Node* Push(Node* head, int32_t 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

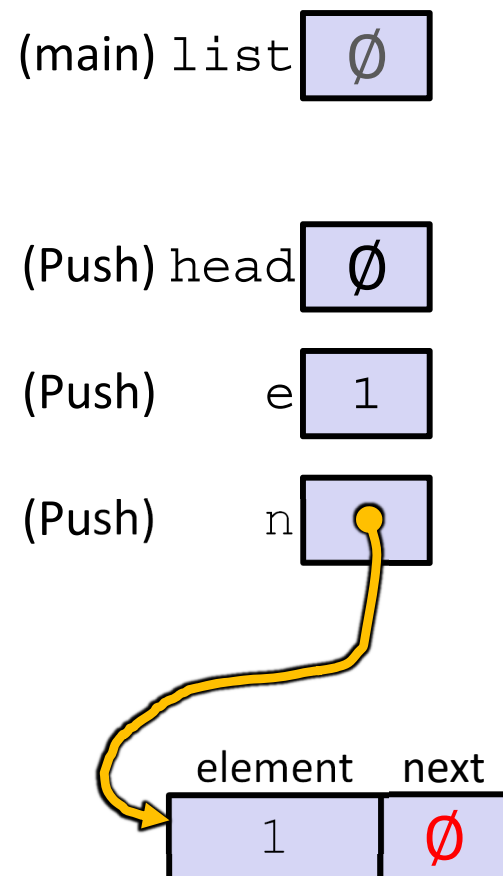
# Push Onto List

Arrow points to  
next instruction.

```
typedef struct node_st {
    int32_t element;
    struct node_st* next;
} Node;

Node* Push(Node* head, int32_t 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

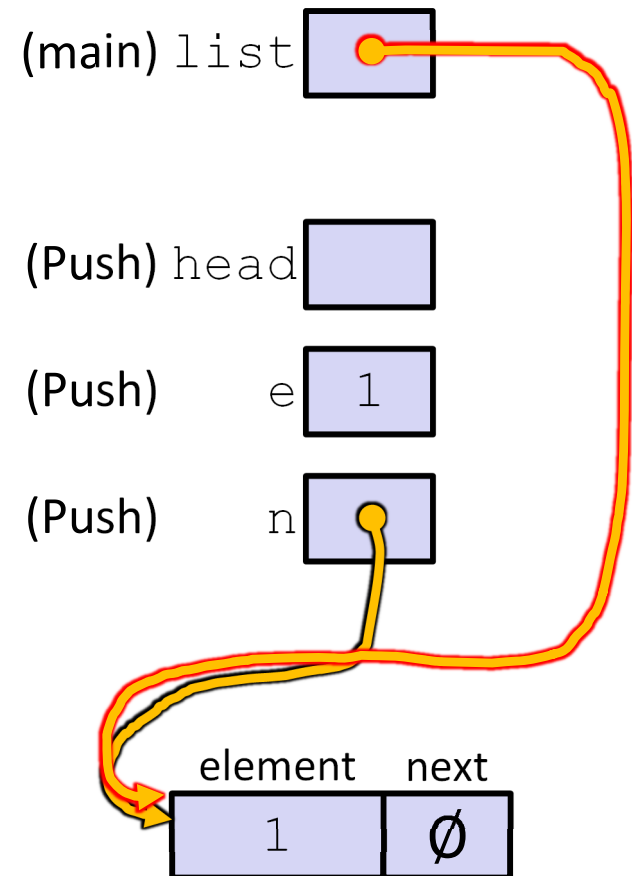
# Push Onto List

Arrow points to  
*next* instruction.

```
typedef struct node_st {
    int32_t element;
    struct node_st* next;
} Node;

Node* Push(Node* head, int32_t 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

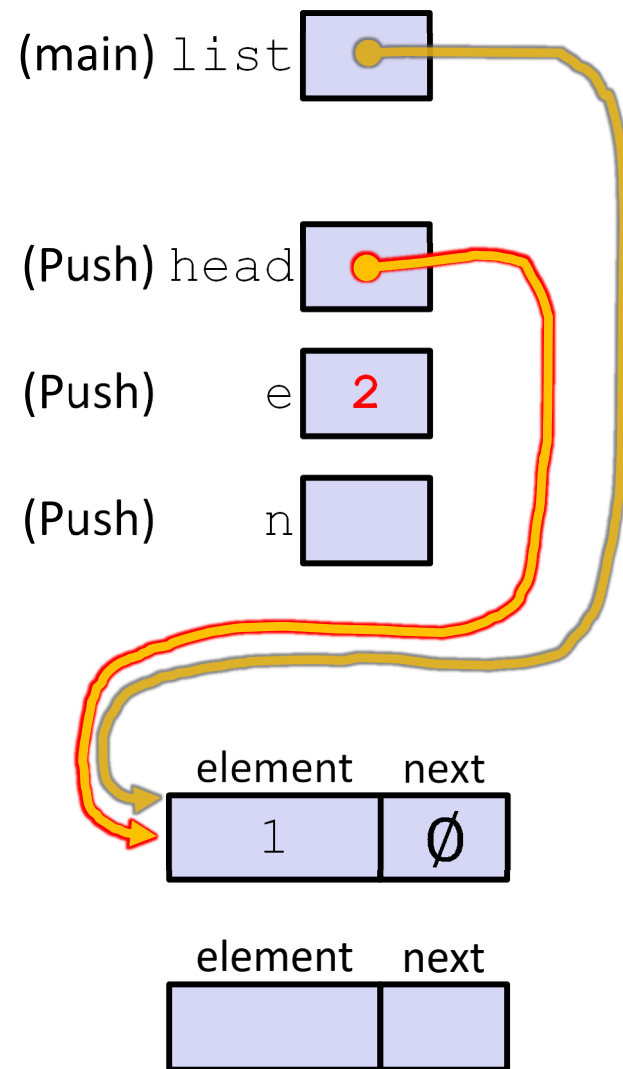
Arrow points to next instruction.

```

typedef struct node_st {
    int32_t element;
    struct node_st* next;
} Node;

Node* Push(Node* head, int32_t 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

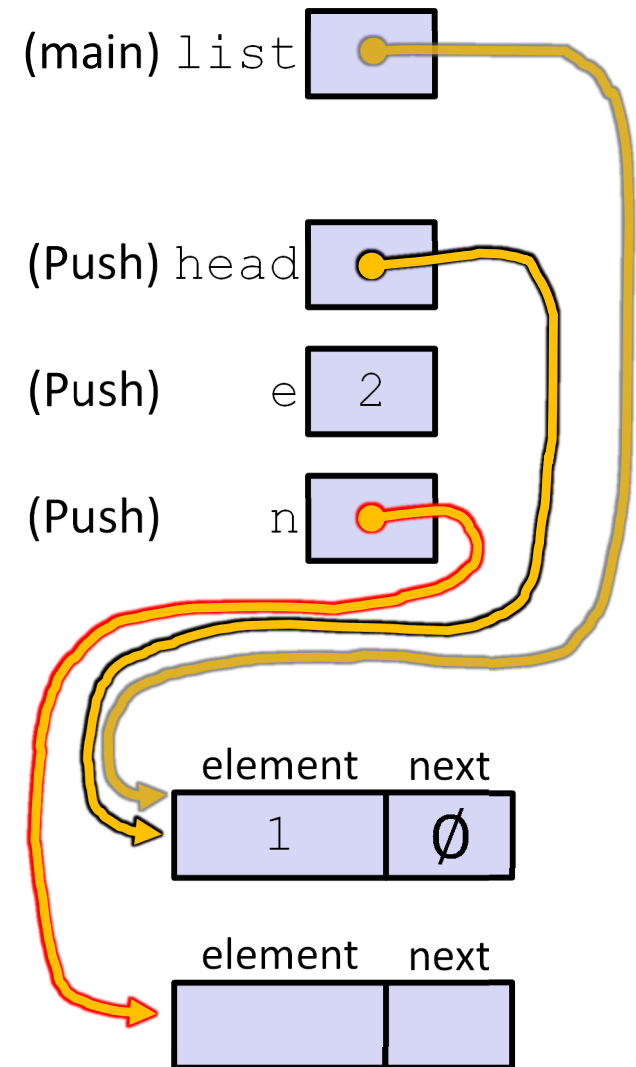
Arrow points to next instruction.

```

typedef struct node_st {
    int32_t element;
    struct node_st* next;
} Node;

Node* Push(Node* head, int32_t 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

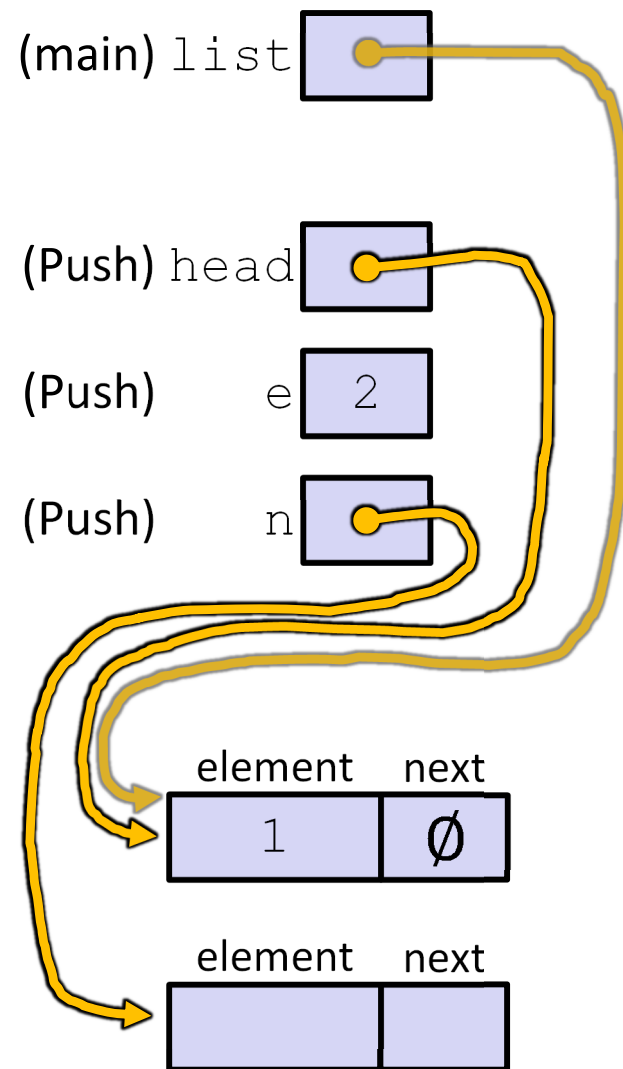
Arrow points to next instruction.

```

typedef struct node_st {
    int32_t element;
    struct node_st* next;
} Node;

Node* Push(Node* head, int32_t 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

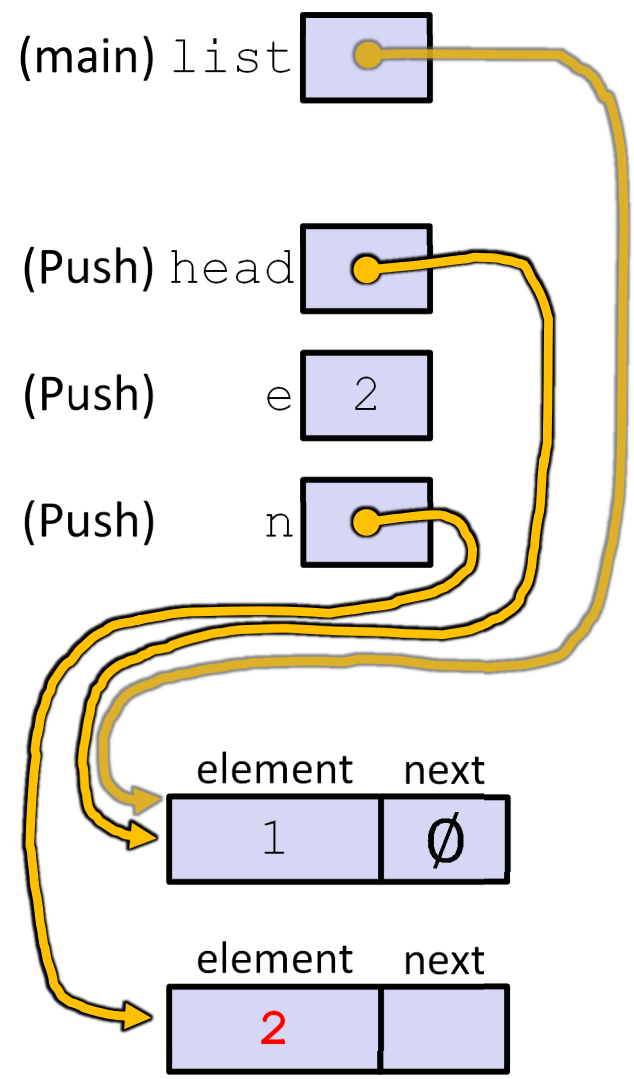
Arrow points to next instruction.

```

typedef struct node_st {
    int32_t element;
    struct node_st* next;
} Node;

Node* Push(Node* head, int32_t 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

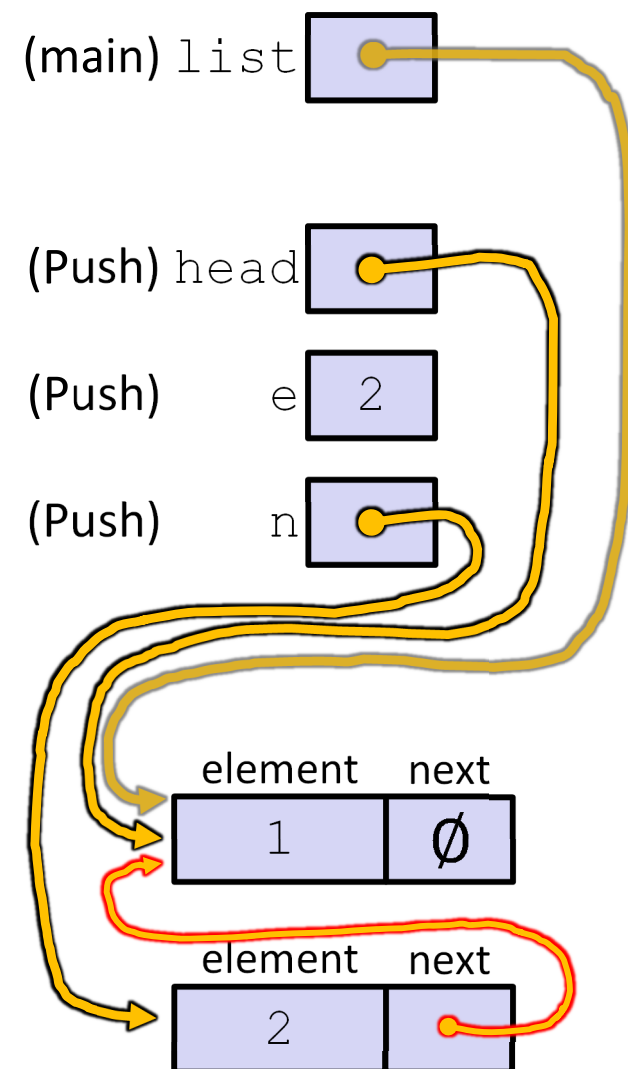
Arrow points to  
*next* instruction.

```
typedef struct node_st {
    int32_t element;
    struct node_st* next;
} Node;

Node* Push(Node* head, int32_t 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

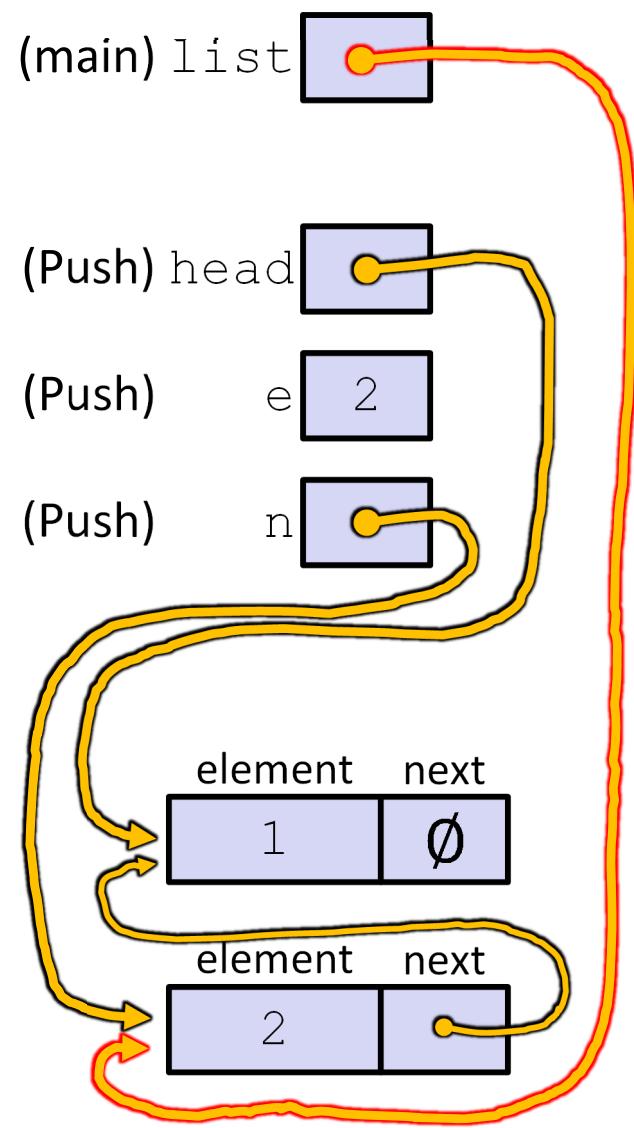
Arrow points to next instruction.

```

typedef struct node_st {
    int32_t element;
    struct node_st* next;
} Node;

Node* Push(Node* head, int32_t 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.

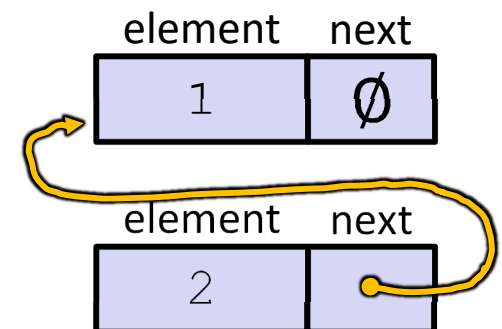
```
typedef struct node_st {
    int32_t element;
    struct node_st* next;
} Node;

Node* Push(Node* head, int32_t 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;
}
```



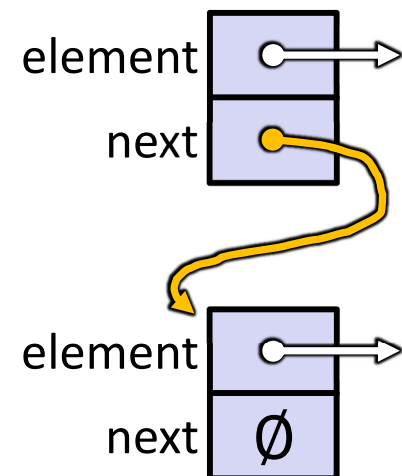
push\_list.c



# A Generic Linked List

- ❖ Let's generalize the linked list element type
  - Let customer decide type (instead of always `int32_t`)
  - 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

# Extra Exercise #1

- ❖ Write a program that defines:
  - A new structured type Point
    - Represent it with `floats` for the x and y coordinates
  - A new structured type Rectangle
    - Assume its sides are parallel to the x-axis and y-axis
    - Represent it with the bottom-left and top-right Points
  - A function that computes and returns the area of a Rectangle
  - A function that tests whether a Point is inside of a Rectangle

## Extra Exercise #2

- ❖ Implement `AllocSet()` and `FreeSet()`
  - `AllocSet()` needs to use `malloc` twice: once to allocate a new `ComplexSet` and once to allocate the “points” field inside it
  - `FreeSet()` needs to use `free` twice

```
typedef struct complex_st {
    double real;    // real component
    double imag;   // imaginary component
} Complex;

typedef struct complex_set_st {
    double    num_points_in_set;
    Complex*  points;    // an array of Complex
} ComplexSet;

ComplexSet* AllocSet(Complex c_arr[], int size);
void FreeSet(ComplexSet* set);
```

# Extra Exercise #3

- ❖ 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