Structs, Modules
CSE 333 Fall 2023

Instructor: Chris Thachuk

Teaching Assistants:

Ann Baturytski   Humza Lala
Yuquan Deng      Alan Li
Noa Ferman       Leanna Mi Nguyen
James Froelich   Chanh Truong
Hannah Jiang     Jennifer Xu
Yegor Kuznetsov
Relevant Course Information (1/2)

- **Exercises**
  - Exercise 1 solution out yesterday, grades released tonight
  - Regrade requests: open 24 hr after, close 72 hr after release
  - Exercise 2 due Thurs by 10pm (10/5)

- **Homework 1 out later today**
  - Not released yesterday, deadline will be extended to compensate
  - Be sure to read headers *carefully* while implementing
  - Use git add/commit/push regularly to save work – easier to share with course staff and good practice when working with a partner on future homework
Relevant Course Information (2/2)

- Section this week will involve group debugging!
  - Be prepared for drawing memory diagrams and using your terminal
Lecture Outline

- structs and typedef
- Generic Data Structures in C
- Modules & Interfaces
Structured Data (351 Review)

- 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:**

```c
struct tagname {
    type1 name1;
    ... // more fields...
    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 (351 Review)

- 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

```c
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!*

```c
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: {%.1f,%.1f}  p2: {%.1f,%.1f}\n", p1.x, p1.y, p2.x, p2.y);
    p2 = p1;
    printf("p1: {%.1f,%.1f}  p2: {%.1f,%.1f}\n", p1.x, p1.y, p2.x, p2.y);
    return EXIT_SUCCESS;
}
```

`structassign.c`
Typedef (351 Review)

- **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`!

```c
// 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

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

Complex* 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

```c
typedef struct point_st {
    int x, y;
} Point;

void DoubleXBroken(Point p) { p.x *= 2; }
void DoubleXWorks(Point* 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: ( , )
    DoubleXWorks(&a);
    printf("(%d,%d)\n", a.x, a.y);  // prints: ( , )
    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

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

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
Check-In Activity

- Write out a C snippet that:
  - Defines a struct for a linked list node that holds (1) a character pointer and (2) a pointer to an instance of this struct
  - Typedefs the struct as `Node`

- Write out the prototype for a function `Pop` that takes the head of a linked list of `Node`, then removes and returns the first node:
Lecture Outline

- structs and typedef
- Generic Data Structures in C
- Modules & Interfaces
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

![Diagram of a simple linked list]

- Head

Element Z → Element Y → Element X
∅
Let’s represent a linked list node with a struct

- For now, assume each element is an `int`

```c
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`
Push Onto List

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
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**

```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;
}
```

*(main) list: ⌀*

*(Push) head: ⌀*

*(Push) e: 1*

*(Push) n: ⌀*

Arrow points to `next` instruction.

*push_list.c*
Push Onto List

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

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

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;
}

Arrow points to next instruction.

push_list.c
**Push Onto List**

```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*
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
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 Onto List

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

```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
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

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

```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;
}
```

Arrow points to `next` instruction.

```
1  Ø
2
```

`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*`)

```c
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 EXIT_SUCCESS;
}
```

`manual_list_void.c`
What would happen if we execute \(*(\text{list} \to \text{next}) = *\text{list}\)?
Something’s Fishy… 🐟

- A (benign) memory leak!

```c
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);
    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

- structs and typedef
- Generic Data Structures in C
- Modules & Interfaces
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 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 (1 of 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

- **Including:**
  - *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 of 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

- Write a program that defines:
  - A new structured type Point
    - Represent it with `float`s 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

```c
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

- Implement and test a 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 #4

- 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()`