

CS 223

Week 5 Lecture 1



Agenda

Announcements

Data structures in C

Valgrind practice

Announcements

Lab this week:

- Go over quiz

Homework #3 will be released Wed Sept 30, due Wed Oct 7

No class Oct 5.

Pointer-based data structures

E.g. linked lists, trees, graphs

No built-in data structures in C!

Nodes in these types of data structures use **structs**

struct pointers connect nodes together

Dive into systems calls these “self-referential structs” (Section 2.7.5)

Linked List Ex

```
struct node {  
    int data;          // used to store a list element's data value  
    struct node *next; // used to point to the next node in the list  
};
```

```
struct node *head, *temp;  
int i;
```

```
head = NULL; // an empty linked list
```

```
head = malloc(sizeof(struct node)); // allocate a node  
assert (head != NULL); // quit if head is null  
head->data = 10; // set the data field  
head->next = NULL; // set next to NULL (there is no next element)
```

```
// add 2 more nodes to the head of the list:  
for (i = 0; i < 2; i++) {  
    temp = malloc(sizeof(struct node)); // allocate a node  
    assert (temp != 0); // quit if temp is null  
    temp->data = i; // set data field  
    temp->next = head; // set next to point to current first node  
    head = temp; // change head to point to newly added node  
}
```

Visualize the linked list from this code

```
struct node {
    int val;
    struct node* next;
};

int main() {
    struct node n1 = {1, NULL};
    struct node n2 = {2, NULL};
    struct node n3 = {3, NULL};

    struct node* list;
    list = &n1;
    n1.next = &n2;
    n2.next = &n3;

    for (struct node* n = list; n != NULL; n = n->next) {
        printf("Val: %d\n", n->val);
    }
}
```

Example: Dynamic Linked list (using malloc/free)

```
struct node {
    int val;
    struct node* next;
};

struct node* insert_front(int val, struct node* head) {
    struct node* n = malloc(sizeof(struct node));
    if (n == NULL) {
        printf("ERROR: Out of space!\n");
        exit(1);
    }
    n->val = val;
    n->next = head;
    return n;
}
```

```
void print(struct node* list) {
    for (struct node* n = list; n != NULL; n = n->next) {
        printf("Val: %d\n", n->val);
    }
}
```

```
int main() {
    struct node* n3 = insert_front(2, NULL);
    struct node* n2 = insert_front(1, n3);
    struct node* n1 = insert_front(0, n2);

    free(n1);
    free(n2);
    free(n3);
}
```

Visualize: insert_front

Assume insert_front is called twice, like so:
struct node* n2 = insert_front(2, NULL);
struct node* n1 = insert_front(1, n2);

```
struct node {  
    int val;  
    struct node* next;  
};  
  
struct node* insert_front(int val, struct node* head) {  
    struct node* n = malloc(sizeof(struct node));  
    if (n == NULL) {  
        printf("ERROR: Out of space!\n");  
        exit(1);  
    }  
    n->val = val;  
    n->next = head;  
    return n;  
}
```

```
void print(struct node* list) {  
    for (struct node* n = list; n != NULL; n = n->next) {  
        printf("Val: %d\n", n->val);  
    }  
}
```

Visualize: print

```
void print(struct node* list) {  
    for (struct node* n = list; n != NULL; n = n->next) {  
        printf("Val: %d\n", n->val);  
    }  
}
```

Assume the list contains the numbers 1, 2, and 3.

Exercise: Draw the stack diagram

```
struct node* insert_front(int val, struct node* head) {  
    struct node* n = malloc(sizeof(struct node));  
    if (n == NULL) {  
        printf("ERROR: Out of space!\n");  
        exit(1);  
    }  
    n->val = val;  
    n->next = head;  
    return n;  
}
```

```
int main() {  
    struct node* n1 = insert_front(1, NULL);  
  
    free(n1);  
    // draw the stack diagram here  
}
```

Example: Dynamic linked list – version 2

```
struct node {  
    int val;  
    struct node* next;  
};  
  
void insert_front(int val, struct node* head) {  
    struct node* n = malloc(sizeof(struct node));  
    if (n == NULL) {  
        printf("ERROR: Out of space!\n");  
        exit(1);  
    }  
    n->val = val;  
    n->next = head;  
    head = n;  
}
```

```
void print(struct node* list) {  
    for (struct node* n = list; n != NULL; n = n->next) {  
        printf("Val: %d\n", n->val);  
    }  
}  
  
int main() {  
    struct node* head = NULL;  
    insert_front(0, head);  
    insert_front(1, head);  
    insert_front(2, head);  
    print(head);  
}
```

This doesn't work. Why?

Example: Dynamic linked list – version 2 (fixed)

```
void insert_front(int val, struct node** head) {
    struct node* n = malloc(sizeof(struct node));
    if (n == NULL) {
        printf("ERROR: Out of space!\n");
        exit(1);
    }
    n->val = val;
    n->next = head;
    *head = n;
}

int main() {
    struct node* head = NULL;
    insert_front(0, &head);
    insert_front(1, &head);
    print(head);
}
```

Exercise: Binary tree

How would you implement a binary tree in C? What structs would you need? What functions would you need?

Tools for C programming

Valgrind - we'll try out today

C Tutor - next time

GDB - next time

C Tutor

- <https://pythontutor.com/c.html#mode=edit>

The screenshot displays the Python Tutor C visualization interface. The browser address bar shows the URL <https://pythontutor.com/visualize.html#mode=display>. The page title is "Python Tutor: Visualize code in [Python](#), [JavaScript](#), [C](#), [C++](#), and [Java](#)".

The code being executed is C (gcc 9.3, C17 + GNU extensions) with the following source code:

```
1 void inc(int* i) {  
2     int local = *i;  
3     local = local + 1;  
4     *i = local;  
5 }  
6 int main() {  
7     int i = 5;  
8     printf("Before: %d\n", i);  
9     inc(&i);  
10    printf("After: %d\n", i);  
11    return 0;  
12 }
```

Below the code, there are navigation controls: a green arrow icon for "line that just executed" and a red arrow icon for "next line to execute". A progress bar indicates the current step, and buttons for "<< First", "< Prev", "Next >", and "Last >>" are provided. The text "Step 7 of 12" is displayed at the bottom.

On the right side, the "Print output (drag lower right corner to resize)" box shows "Before: 5". Below this, the memory stack is visualized, showing the "main" frame and the "inc" frame. The "inc" frame contains a "local" variable with the value 5. A "pointer to int" is shown, pointing to the "local" variable. The "main" frame contains a variable "i" with the value 5. The "Stack" and "Heap" labels are also visible.

At the bottom right, the "C/C++ details:" dropdown menu is set to "none [default view]".

GDB: GNU Debugger

run (r): Start a program and step through it line by line

break (b): Pause the execution of a program when it reaches certain points in its code

break if: Pause the execution of a program on user-specified conditions

print (p): Show the values of variables at the point in execution that a program is paused

continue (c): Continue a program's execution after a pause

where: Examine the program's execution state at the point when it crashes

up/down: Examine the contents of any stack frame on the call stack

Generating debugger symbols

The debugger needs information about the variable and function names in order to present the execution of the program in a human readable way

The `-g` option does this

```
gcc -g <filename>.c
```

Breakpoints

A **breakpoint** is a point in the program where the debugger should pause execution

b <function_name>

b <line_num>

b <filename>:<line_num>

Valgrind

Can help us find common memory errors!

- Reading/writing at unallocated memory (Index out of bounds)
- Freeing already freed memory
- Memory leaks
- Writing to a null pointer

Danger zone #1: Reading from uninitialized memory

```
int main() {  
    int *ptr, x;  
    ptr = malloc(sizeof(int) * 10);  
    x = ptr[3]; // bad!  
    return 0;  
}
```

NOTE: Your program might run without crashing even if there are errors in it!

Danger zone #2: Reading/writing at unallocated memory

```
int main() {  
    int values[10];  
    printf("%d\n", values[11]);  
    values[11] = 5; // memory not allocated!  
}
```

Danger zone #3: Freeing already freed memory

```
int main() {  
    int* values = malloc(sizeof(int) * 10);  
    free(values);  
    free(values);  
}
```

Danger zone? #4: Freeing already freed memory

```
int main() {  
    int* values = malloc(sizeof(int) * 10);  
    free(values);  
    values = NULL;  
    free(values);  
}
```

Memory leaks: Use valgrind to find

```
int main() {  
    int* ptr;  
    ptr = malloc(sizeof(int) * 10);  
    ptr = malloc(sizeof(int) * 5);  
    return 0;  
}
```

Invalid memory read example: bigfish.c

```
// allocate space for two int arrays
bigfish = (int *)malloc(sizeof(int)*10);
littelfish = (int *)malloc(sizeof(int)*10);
for (i=0; i < 10; i++) {
    bigfish[i] = 10+i;
    littelfish[i] = i;
}
print_array(bigfish,10, "bigfish");
print_array(littelfish,10, "littelfish");
for (i=0; i < 13; i++) {
    bigfish[i] = 66+i;
}
printf("\nafter loop:\n");
print_array(bigfish,10, "bigfish");
print_array(littelfish,10, "littelfish");
```

```
bigfish array:
10 11 12 13 14 15 16 17 18 19
littelfish array:
0 1 2 3 4 5 6 7 8 9

after loop:
bigfish array:
66 67 68 69 70 71 72 73 74 75
littelfish array:
78 1 2 3 4 5 6 7 8 9
Segmentation fault (core dumped)
```

Invalid memory read example: bigfish.c

```
// allocate space for two int arrays
bigfish = (int *)malloc(sizeof(int)*10);
littlefish = (int *)malloc(sizeof(int)*10);
for (i=0; i < 10; i++) {
    bigfish[i] = 10+i;
    littlefish[i] = i;
}
print_array(bigfish,10, "bigfish");
print_array(littlefish,10, "littlefish");
for (i=0; i < 13; i++) {
    bigfish[i] = 66+i;
}
printf("\nafter loop:\n");
print_array(bigfish,10, "bigfish");
print_array(littlefish,10, "littlefish");
```

Example: badprog.c

```
int findAndReturnMax(int *array1, int len, int max) {
    int i;
    if (!array1 || (len <= 0) ) {
        return -1;
    }
    max = array1[0];
    for (i=1; i <= len; i++) {
        if (max < array1[i]) {
            max = array1[i];
        }
    }
    return 0;
}
```

```
int main(int argc, char *argv[]) {
    int arr[5] = { 17, 21, 44, 2, 60 };
    int max = arr[0];

    if ( findAndReturnMax(arr, 5, max) != 0 ) {
        printf("strange error\n");
        exit(1);
    }
    printf("max value in the array is %d\n", max);
    return 0;
}
```

What is the output of this program supposed to be?

Aside: Writing C programs with multiple files

A **header** file includes definitions for structs, constants, and functions so that they can be re-used

Examples: `stdio.h`, `stdlib.h`

Define your own headers for your own code:

```
#include "myfile.h"
```

Demo: `mystring.h`

```
// Example header file structure
#ifndef _filename_H_ // define a macro to avoid redefinitions
#define _filename_H_

// define structs and constants
#define ARRAY_SIZE 32
struct dataT
{
    int x;
    char buffer[ARRAY_SIZE];
};

// use extern for functions
extern void init_data(struct dataT* data);

#endif // close #ifndef
```