

This space for rent

David Notkin • Autumn 2009 • CSE303 Lecture 13

Upcoming schedule

Today 10/23	Monday 10/26	Wednesday 10/28	Friday 10/30	Monday 11/2
Finish-up Wednesday Some specifics for HW3 Social implications Friday	Memory management		Midterm review	Midterm

- structured data
 - struct, typedef
 - structs as parameters/returns
 - arrays of structs
- linked data structures
 - stacks
 - linked lists

CSE303 Au09

2

free: releases memory

- `free(pointer);`
- Releases the memory pointed to by the given pointer
 - precondition: pointer must refer to a heap-allocated memory block that has not already been freed
 - it is considered good practice to set a pointer to **NULL** after freeing

```
int* a = (int*) calloc(8, sizeof(int));
...
free(a);
a = NULL;
```

Memory corruption

- If the pointer passed to free doesn't point to a heap-allocated block, or if that block has already been freed, bad things happen
 - you're lucky if it crashes, rather than silently corrupting something

```
int* a1 = (int*) calloc(1000, sizeof(int));
int a2[1000];
int* a3;
int* a4 = NULL;

free(a1); // ok
free(a1); // bad (already freed)
free(a2); // bad (not heap allocated)
free(a3); // bad (not heap allocated)
free(a4); // bad (not heap allocated)
```

Structured data

- **struct**: A type that stores a collection of variables
 - like a Java class, but with only fields (no methods or constructors)
 - instances can be allocated on the stack or on the heap

```
struct Point { // defines a new structured
    int x, y; // type named Point
};
```

Using structs

- Once defined, a **struct** instance is declared just like built-in types (e.g., `int`, `char`) except preceded by **struct**
 - this allocates an instance on the stack
 - name fields of a **struct** using the `.` operator

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

int main(void) {
    struct Point p1;
    struct Point p2 = {42, 3};
    p1.x = 15;
    p1.y = -2;
    printf("p1 is (%d, %d)\n",
        p1.x, p1.y);
    return 0;
}
```

typedef

- Tell C to acknowledge your `struct` type's name with `typedef`

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

int main(void) {
    Point p1;    // don't need to write 'struct'
    p1.x = 15;
    p1.y = -2;
    printf("p1 is (%d, %d)\n", p1.x, p1.y);
    return 0;
}
```

Structs as parameters

- when you pass a `struct` as a parameter, it is copied
 - not passed by reference as in Java

```
int main(void) {
    Point p = {10, 20};
    swapXY(p);
    printf("(%d, %d)\n", p.x, p.y);
    return 0;    // prints (10, 20)
}

void swapXY(Point a) {
    int temp = a.x;
    a.x = a.y;
    a.y = temp;    // does not work
}
```

Pointers to structs

- structs can be passed using pointers
 - must use parentheses when dereferencing a `struct*` (because of operator precedence)

```
int main(void) {
    Point p = {10, 20};
    swapXY(&p);
    printf("(%d, %d)\n", p.x, p.y);
    return 0;    // prints (20, 10)
}

void swapXY(Point* a) {
    int temp = (*a).x;
    (*a).x = (*a).y;
    (*a).y = temp;
}
```

The -> operator

- We often allocate structs on the heap
 - `pointer->field` is equivalent to `(*pointer).field`

```
int main(void) {
    Point* p = (Point*) malloc(sizeof(Point));
    p->x = 10;
    p->y = 20;
    swapXY(p);
    printf("(%d, %d)\n", p->x, p->y);    // (20, 10)
    return 0;
}

void swapXY(Point* a) {
    int temp = a->x;
    a->x = a->y;
    a->y = temp;
}
```

Copy by assignment

- One `struct`'s entire contents can be copied to another with =
 - `struct2 = struct1;` // copies the memory

```
int main(void) {
    Point p1 = {10, 20}, p2 = {30, 40};
    p1 = p2;
    printf("(%d, %d)\n", p1.x, p1.y);    // (30, 40)
    // is this the same as p1 = p2; above?
    Point* p3 = (Point*) malloc(sizeof(Point));
    Point* p4 = (Point*) malloc(sizeof(Point));
    p3->x = 70;
    p3->y = 80;
    p3 = p4;
    printf("(%d, %d)\n", p3->x, p3->y);
}
```

Struct as return value

- We generally pass/return structs as pointers
 - takes less memory (and time) than copying
 - if a `struct` is `malloc`-ed and returned as a pointer, who frees it?

```
int main(void) {
    Point* p1 = new_Point(10, 20);
    ...
    free(p1);
}

// creates/returns a Point; sort of a constructor
Point* new_Point(int x, int y) {
    Point* p = (Point*) malloc(sizeof(Point));
    p->x = x;
    p->y = y;
    return p;    // caller must free p later
}
```

Comparing structs

- relational operators (`==`, `!=`, `<`, `>`, `<=`, `>=`) don't work with **structs**

```
Point p1 = {10, 20};
Point p2 = {10, 20};
if (p1 == p2) { ...      // error
```

- what about this?

```
Point* p1 = new_Point(10, 20);
Point* p2 = new_Point(10, 20);
if (p1 == p2) { ...      // true or false?
```

Comparing structs, cont'd

- the right way to compare two **structs**: write your own

```
#include <stdbool.h>
bool point_equals(Point* a, Point* b) {
    if (a->x == b->x && a->y == b->y) {
        return true;
    } else {
        return false;
    }
}

int main(void) {
    Point p1 = {10, 20};
    Point p2 = {10, 20};
    if (point_equals(&p1, &p2)) { ...
```

Structs and input

- you can create a pointer to a field of a **struct**
 - structs'** members can be used as the target of a `scanf`, etc.

```
int main(void) {
    Point p;
    printf("Please type your x/y position: ");
    scanf("%d %d", &p.x, &p.y);
}
```

```
int main(void) {
    Point* p = (Point*) malloc(sizeof(Point));
    printf("Please type your x/y position: ");
    scanf("%d %d", &p->x, &p->y);
}
```

Arrays of structs

- parallel arrays are conceptually linked by their index
 - parallel arrays are usually bad design; isn't clear that they are related
 - you should often replace such arrays with an array of **structs**

```
int id[50];           // parallel arrays to store
int year[50];        // student data (bad)
double gpa[50];
```

```
typedef struct Student {
    int id, year;
    double gpa;
} Student;
Student students[50];
```

Structs with pointers

- What if we want a **Student** to store a significant other?

```
typedef struct Student {      // incorrect .. Why?
    int id, year;
    double gpa;
    struct Student sigother;
} Student;
```



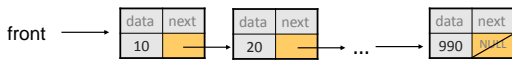
Alternative

```
typedef struct Student {      // correct
    int id, year;
    double gpa;
    struct Student* sigother;
} Student;
```

Linked data structures

- C does not include collections like Java's ArrayList, HashMap
 - must build any needed data structures manually
 - to build a linked list structure, create a chain of structs/pointers

```
typedef struct Node {
    int data;
    struct Node* next;
} Node;
Node* front = ...;
```



Manipulating a linked list

- there is only a node type (`struct`), no overall list class
- list methods become functions that accept a front node pointer:

```
int list_length(Node* front) {
    Node* current = front;
    int count = 0;
    while (current != NULL) {
        count++;
        current = current->next;
    }
    return count;
}
```



Exercise

- Write a complete C program that allows the user to create a basic stack of ints. The user should be able to:
 - push : put a new int onto the top of the stack.
 - pop : remove the top int from the stack and print it.
 - clear : remove all ints from the stack.
- Do not make any assumptions about the size of the stack.
 - Do not allow any memory leaks in your program.

Questions?