C: structs, malloc, free

CSE 413, Autumn 2007
10-24-2007

Topics

- structs
- malloc, free and the heap

Structs

- A struct is a record. (similar to a Java object with no methods.)
  - » x.f is for field access.
  - » (*x).f in C is like x.f in Java.
  - » x->f is an abbreviation for (*x).f.
- There is a huge difference between passing a struct and passing a pointer to a struct.
- (see struct example code)

Review: Passing Objects in Java

- Remember Java: objects are passed "by reference" (a copy of a *reference* to the object is passed)

```
public static void java_example(Point arg1, Point arg2) {
  // Modifies values referred to by arg1
  arg1.x = 100;
  arg1.y = 100;

  // Modifies the arguments
  // (which are copies - does nothing)
  Point temp = arg1;
  arg1 = arg2;
  arg2 = temp;
}
```

Notes:
- This does not make copies of the Point objects themselves, just a copy of the references (this is really like an address or pointer in C)

Passing structs in C

- In C there are two ways we could pass a struct:
  - » Pass the struct itself (this makes a copy of the struct - changes to fields in the struct are **not visible** outside the function).
  - » Pass a pointer to the struct (this makes a copy of the pointer - changes to fields in the struct are **visible** outside of the function)

```
// p is unchanged after this call
void wrong_update_x(struct Point p, int new_x) {
  p.x = new_x;
}

// p is changed here
void update_x(struct Point * p, int new_x) {
  p->x = new_x;
}
```


Returning structs in C

- In C there is another way we could change a struct inside of a function even if we pass in the struct itself:

```
// Modifies its own copy of p1, but then
// returns that modified copy.
struct Point change_point(struct Point p1) {
  p1.x = 15;
  p1.y = 26;
  return p1; // p1 is copied back
}
```

- This works, but copying entire structs can be less efficient than copying a single pointer/address.

Java vs. C again

In Java:

```
Point arg1, Point arg2;
arg1 = new Point(0, 0);
arg2 = new Point(5, 6);

arg1 = arg2; // arg1 refers to the same object as arg2
arg2.x = 78; // arg1.x and arg2.x now both contain 78
```

In C:

```
struct Point arg1;
struct Point arg2;
initpoint(&arg1);      // set x and y to zero
initpoint(&arg2);      // set x and y to zero
modifypoint(&arg2, 5, 6); // set x=5 and y=6

arg1 = arg2;   // the x and y fields from arg2 are
               // copied into the x and y fields of arg1
arg2.x = 78; // arg2.x contains 78, arg1.x contains 5
```

7

Review: Creating objects in Java

```
arg2 = new Point(5, 6);
```

This does several different things:

- Allocate space for a Point
- Initialize the fields to null or 0
- Call the user-written constructor function
- Return a reference (pointer) to the new object

We can then pass this reference to and from functions and the object lives "forever".

8

Lifetimes in C

```
struct Point* bad_idea() {
  struct Point ans;
  ans.x = 0;
  ans.y = 0;
  return &ans;
}
```

What if we want:

- the efficiency of returning a pointer to a **`Point`**
- the **`Point`** to live beyond the lifetime of this function!

9

Malloc

```
void *malloc(size_t size);
```

- **`size_t`** is an unsigned long, indicates how many bytes of memory are requested.
- Returns a pointer to the newly-allocated memory.
- Returns NULL on failure.
- Does not initialize the memory.
- You should cast the result to the pointer type you want.

10

Malloc

Example:

```
char *char_array;
char_array =
  (char*)malloc(MAX_SIZE*sizeof(char));
```

- Returns a pointer to a chunk of memory on the heap
- Large enough to hold an array of length **`MAX_SIZE`** with elements of type **`char`**.
- The memory is still not initialized!

11

Free

```
void *free(void* ptr);
```

- Returns the chunk of memory pointed to by ptr to the heap.

Example:

```
int *buffer1;
buffer1 = (int*) malloc(50*sizeof(int));
free(buffer1);
```

You should **`free`** what you **`malloc`**. Why?

Q: What is the value of buffer1 now?

12