2. Arrays, Pointers, and Memory Management

Will Qian

January 15, 2015
Today’s Agenda

1. Review of Data Structures
2. Arrays
3. Pointers and References
4. Dynamic memory allocation
Today’s Agenda

1. Review of Data Structures
2. Arrays
3. Pointers and References
4. Dynamic memory allocation
Using `struct`

```c
1 struct date_t {
2     uint64_t year, month, day;
3 };
4
5 int main() {
6     struct date_t today;
7     return 0;
8 }
```
Using `struct` and `typedef`

```c
1 typedef struct date_t Date;
2
3 struct date_t {
4     uint64_t year, month, day;
5 };
6
7 int main() {
8     Date today;
9     return 0;
10 }
```
Using `struct` and `typedef`

1 typedef struct date_t {
2     uint64_t year, month, day;
3 } Date;

4

5 int main() {
6     Date today;
7     return 0;
8 }
Initializing an object

```c
1 typedef struct date_t {
2     uint64_t year, month, day;
3 } Date;
4
5 int main() {
6     Date today = (Date){ .year = 2014 };  
7     return 0;
8 }
```
Object functions

5 Date Date_init(...) {
6     ...
7 }
8
9 Date Date_inc(Date date) {
10     ...
11 }
12
13 Date Date_dec(Date date) {
14     ...
15 }
Object functions

16
17 int main() {
18     Date today = Date_init(2015, 1, 15);
19     Date tomorrow = Date_inc(today);
20     return 0;
21 }
Date is a non-recursive data structure; what about a recursive one?

```c
1 typedef struct node_t {
2     struct node_t next;
3     int value;
4 } Node;
```
Date is a non-recursive data structure; what about a recursive one?

```c
1 typedef struct node_t {
2   struct node_t next;
3   int value;
4 } Node;
```

Uh-oh, compiler claims that `next` has an incomplete type.
Recursive data structures

Adding this magic asterisk seems to help.

1 typedef struct node_t {
  2   struct node_t *next;
  3   int value;
  4 } Node;
Adding this magic asterisk seems to help.

```c
1 typedef struct node_t {
2     struct node_t *next;
3     int value;
4 } Node;
```

Asterisk indicates that `next` is actually a pointer.
Today’s Agenda

1. Review of Data Structures
2. Arrays
3. Pointers and References
4. Dynamic memory allocation
Notation

```c
1 int arr1[5];
2
3 int arr2[5] = { 0, 1, 2, 3, 4 };  // arr2 is now { 0, -1, 2, 3, 4 }
4 arr2[1] = -1;
5
```
Physical representation

int:

(4 bytes)

int[5]:

(20 bytes)

(5 × 4 bytes)
Strings are arrays, too!

"6.179"

`'6'  '.'  '1'  '7'  '9'  '\0'`
Strings are arrays, too!

"6.179"

| '6' | ' ' | '1' | '7' | '9' | '\0' |

- "Null-terminated strings" use the null character to signify end of string
- 5 visible chars + 1 null char = 6 elements in array
Two new string functions

Library: <string.h>

- `strcmp(str1, str2)` – compares the contents of `str1` and `str2`
- `strlen(str)` – returns the number of characters in `str`, excluding the null terminator

Also, don’t forget about `sprintf()`!
Returning an array is dangerous!

- Array memory is allocated within the scope of the function
- Upon exiting function, memory is deallocated
- Future variables can write over returned memory block!

Instead, if you want to output to an array (e.g. a string), you should pass it in as a parameter instead!
Buffer overflow is when you write to memory that isn't allocated to your array.

```
1 char arr[4];
2 int num = ~0;  // all bits are 1
3 arr = "6179";
4 printf("%d\n", num == ~0);
```

Line 3 overwrites the first byte of num to be 0 with the null terminator. Note that this won't necessarily happen, but the code is set up so that it can happen.
Today’s Agenda

1. Review of Data Structures
2. Arrays
3. Pointers and References
4. Dynamic memory allocation
Motivations

- Using very large objects (e.g. 1GB structs)
- Recursive data structures
- Mutating an object in-place
Motivations

- Using very large objects (e.g. 1GB structs)
- Recursive data structures
- Mutating an object in-place

The overarching theme: saving and reusing memory
Memory Address

The number of the first byte that the variable is stored in.

e.g. 0x0000FEED

(64-bit machines have 8-byte addresses, 32-bit machines have 4-byte addresses.)
Memory Address

Use the ampersand (&) to obtain the address of a variable.

```
1 int a = 0;
2 printf("%p\n", &a);  // prints the address of a
```
Pointers

A “type” whose value indicates the address of the value that it’s pointing to.

Indicated by an asterisk (*) after a data type (primitive, object type, even another pointer type).
Pointers

1 int a = 0;
2 // initialize b and c, declare d
3 int *b = &a, c = 10, *d;
4 *b = 1; // a’s value is now 1
5 b = 1; // b now points to address 0x00000001
6 *b = 2; // a’s value is still 1

(Also, line 6 will likely seg fault because address 0x00000001 is usually allocated to the OS.)
Pointers are just numbers

So you can add or subtract with them! (No multiplication or division, though.)

Result: offsets from the original pointer, depending on the pointer type.
Pointers are just numbers

1 int a = 0, *b = &a;
2 char c = 0, *d = &d;
3 printf("%d\n", (int)(b + 1) - (int)b);
4 printf("%d\n", (int)(d + 1) - (int)d);

Ignoring all the compiler warnings, line 3 prints 4 and line 4 prints 1. Why? (Hint: an int is 4 bytes, a char is 1 byte.)
Pointers and arrays

As it turns out, an array is actually a pointer to the first element of the array. This explains why an array doesn’t know its own size, as well as why `scanf()` doesn’t require an ampersand in front of strings.

Furthermore, when returning a pointer (or an array) from a function, you are actually returning the address – that’s why the actual array itself gets deallocated.
Today’s Agenda

1. Review of Data Structures
2. Arrays
3. Pointers and References
4. Dynamic memory allocation
Dynamic memory allocation

- Refers to allocating memory for a variable when the compiler doesn’t know how much to allocate
- Allows allocation of arrays of non-constant lengths
- Prevents deallocation of memory after exiting function
malloc() and free()

Library: <stdlib.h>

1 int *arr = malloc(sizeof(*arr) * 100);
2 arr[99] = 6179;
3 free(arr);
Malicious `malloc`ing

After memory is `malloc`ed, it will not be freed until `free()` is called upon it, even after program execution ends. So, when memory that is `malloc`ed is not `free`ed by the end of the program, we call that a memory leak.

As more and more memory is leaked, less of your RAM is available, until you are forced to restart your computer (thus forcibly freeing the memory).

Taking advantage of this, one can `malloc` a lot of memory at once and never free it, so that the machine is rendered useless due to the lack of available RAM.
Despite what bad things `malloc` can result in, it’s actually incredibly useful if used properly. Here are some basic use cases:

- Returning arrays
- Variable-length data structures
- Recursive data structures

To sum it all up: to do anything particularly useful with C, you will need to dynamically allocate memory.
Overview

As a case study, we will consider the problem of writing a linked list data structure to store int’s. We will want five functions for our object:

- **List_init()** – creates an empty list
- **List_size(List*)** – returns the length of the list
- **List_append(List*, int)** – appends value list
- **List_get(List*, int)** – returns the value at index
- **List_remove(List*, int)** – removes the value at index
The object

typedef struct list_t {
    struct list_t *next;
    int value;
} List;
List_init

```c
List* List_init() {
    List *node = malloc(sizeof(*node));
    node->next = NULL;
    return node;
}
```
List_size

```c
size_t List_size(List* node) {
    if (node->next == NULL)
        return 0;
    return List_size(node->next) + 1;
}
```
void List_append(List* node, int value) {
    if (node->next != NULL)
        List_append(node->next, value);
    else {
        node->value = value;
        node->next = List_init();
    }
}

List_append
List_get

int List_get(List* node, int index) {
    if (node->next == NULL)
        return -1;
    if (index == 0)
        return node->value;
    return List_get(node->next, index - 1);
}
List_remove

void List_remove(List* node, int index) {
    if (node->next == NULL)
        return;
    if (index == 0 || index == 1) {
        if (index == 0)
            node->value = node->next->value;
        List* next = node->next;
        node->next = node->next->next;
        free(next);
    }
    else
        List_remove(node->next, index - 1);
}"
Takeaway

“Giving someone a copy your address is easier than giving that person a copy of your house!”