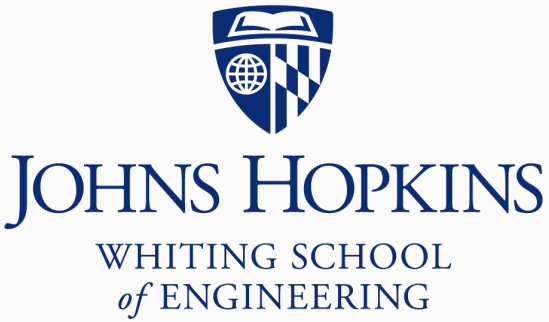
Structures

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

Structures

A collection of related variables, bundled into one

```
struct card {
    char rank;
    char suit;
};
```

Structures

Variables in a `struct` are *fields*

```
struct cc_receipt {
    float amount;
    char cc_number[16];
};
```

Two fields: `float` named `amount`, and `char[16]` named `number`

Structures

Say we're programming a checkers game.

We want a `struct` describing everything about a game piece

```
struct checkers_piece {
    // ???
};
```

Structures

```
struct checkers_piece {
    int x; // horizontal offset
    int y; // vertical offset
    int black; // 0 = white, non-0 = black
};
```

Structures

```c
#include <stdio.h>

struct date {
    int year;
    int month;
    int day;
};

int main() {
    struct date today; // like 3 variables in 1!
    today.year = 2016; // use . to refer to fields
    today.month = 2;
    today.day = 26;
    printf("Today's date: %d/%d/%d\n",
           today.month, today.day, today.year);
    return 0;
}
```

Structures

```
$ gcc -c struct_eg1.c -std=c99 -pedantic -Wall -Wextra
$ gcc -o struct_eg1 struct_eg1.o
$ ./struct_eg1
Today's date: 2/26/2016
```

Structures

The `struct name { ... };` syntax defines a new struct

```c
struct date {
    int year;
    int month;
    int day;
};

// declare variable 'today' with type 'struct date'
struct date today;
```

Structures

`struct` variable can be initialized in similar way to an array:

```
struct date {
    int year;
    int month;
    int day;
};

...

struct date today = {2016, 2, 26};
```

Structures

struct fields can be other structs

```
struct date {
    int year;
    int month;
    int day;
};

struct cc_transaction {
    // struct within a struct is fine!
    struct date purchase_date;
    float amount;
    char cc_number[16];
};
```

Structures

struct fields can be pointers

```
struct player {
    int home_runs;
    int strikeouts;
    int walks;
};

struct team {
    struct player *catcher;
    struct player *first_baseman;
    struct player *second_baseman;
    ...
};
```

Structures

`sizeof(struct player)` returns total size of all fields

- With a caveat we'll see later

```
struct date {
    int year;
    int month;
    int day;
};
```

What is `sizeof(struct date)`?

Structures

```c
#include <stdio.h>

struct date {
    int year;  // 4 bytes
    int month; // 4 bytes
    int day;   // 4 bytes
};

int main() {
    printf("%d\n", (int)sizeof(struct date));
}
```

```
$ gcc -c struct_sizeof.c -std=c99 -pedantic -Wall -Wextra
$ gcc -o struct_sizeof struct_sizeof.o
$ ./struct_sizeof
12
```

Structures

A `struct` can be a function parameter and/or return type

```
struct date next_day(struct date d) {
    if((++d.day) > 30) { // assume 30-day months
        d.day = 1;
        if((++d.month) > 12) {
            d.month = 1;
            d.year++;
        }
    }
    return d;
}
```

Structures

What if it were a `void` function, without `return d` at the end?

```
void next_day(struct date d) {
    if((++d.day) > 30) { // assume 30-day months
        d.day = 1;
        if((++d.month) > 12) {
            d.month = 1;
            d.year++;
        }
    }
}
```

Structures

`structs` are passed by value

- `next_day` on previous slide has no effect
- Alternative 1: return a new `struct`
- Alternative 2: pass *pointer* to `struct`

Structures

```
void next_day_in_place(struct date *d) {
    if((++(*d).day) > 30) {
        (*d).day = 1;
        if((++(*d).month) > 12) {
            (*d).month = 1;
            (*d).year++;
        }
    }
}
```

Structures

d->day is a synonym for (*d).day

```
void next_day_in_place(struct date *d) {
    if((++d->day) > 30) {
        d->day = 1;
        if((++d->month) > 12) {
            d->month = 1;
            d->year++;
        }
    }
}
```

Structures

```c
#include <stdio.h>
#include "date.h" // "struct date" defined here

struct date next_day(struct date d) {
    if(++d.day > 30) {
        d.day = 1;
        if(++d.month > 12) {
            d.month = 1;
            d.year++;
        }
    }
    return d;
}

int main() {
    struct date today = {2016, 2, 26};
    struct date tomorrow = next_day(today);
    printf("Tomorrow's date: %d/%d/%d\n",
           tomorrow.month, tomorrow.day, tomorrow.year);
}
```

```
$ gcc -c struct_next_day_1.c -std=c99 -pedantic -Wall -Wextra
$ gcc -o struct_next_day_1 struct_next_day_1.o
$ ./struct_next_day_1
Tomorrow's date: 2/27/2016
```

Structures

```c
#include <stdio.h>
#include "date.h" // "struct date" defined here

void next_day_in_place(struct date *d) {
    if((++d->day) > 30) {
        d->day = 1;
        if((++d->month) > 12) {
            d->month = 1;
            d->year++;
        }
    }
}

int main() {
    struct date today = {2016, 12, 30};
    next_day_in_place(&today);
    printf("Tomorrow's date: %d/%d/%d\n",
           today.month, today.day, today.year);
    return 0;
}
```

```
$ gcc -c struct_next_day_2.c -std=c99 -pedantic -Wall -Wextra
$ gcc -o struct_next_day_2 struct_next_day_2.o
$ ./struct_next_day_2
Tomorrow's date: 1/1/2017
```

Structures

You can have an array of structs

```
struct album {
    const char *name;
    const char *artist;
    double length;
};

struct album music_collection[99999];
music_collection[0].name = "The Next Day";
music_collection[0].artist = "David Bowie";
music_collection[0].length = 41.9;
music_collection[1].name = "Hunky Dory";
...
```

Structures

What is `sizeof(struct album)`?

```
struct album {
    const char *name;
    const char *artist;
    double length; // 8 bytes
};
```

Structures

```
$ gcc -c struct_sizeof_album.c -std=c99 -pedantic -Wall -Wextra
$ gcc -o struct_sizeof_album struct_sizeof_album.o
$ ./struct_sizeof_album
sizeof(struct album) = 24
```

24 bytes

- `const char *`s are just (8-byte) pointers
- Strings themselves not stored in the `struct`

Structures

You can have a `struct` with an array in it:

```
struct cc_receipt {
    float amount;
    char cc_number[16];
};
```

Structures

What is `sizeof(struct cc_receipt)`?

```
struct cc_receipt {
    float amount; // 4 bytes
    char cc_number[16];
};
```

Structures

```
$ gcc -c sizeof_receipt.c -std=c99 -pedantic -Wall -Wextra
$ gcc -o sizeof_receipt sizeof_receipt.o
$ ./sizeof_receipt
sizeof(struct cc_receipt) = 20
```

Answer: 20 bytes. `char cc_number[16]` is inside the struct, taking up 16 bytes.

Structures

```c
#include <stdio.h>

struct ten_ints {
    int ints[10];
};

void func1(struct ten_ints ints) {
    printf("func1 sizeof(ints)=%d\n", (int)sizeof(ints));
}

void func2(int *ints) {
    printf("func2 sizeof(ints)=%d\n", (int)sizeof(ints));
}

int main() {
    struct ten_ints ints;
    func1(ints);
    func2(ints.ints);
    return 0;
}
```

```
$ gcc -c struct_sizeof_receipt.c -std=c99 -pedantic -Wall -Wextra
$ gcc -o struct_sizeof_receipt struct_sizeof_receipt.o
$ ./struct_sizeof_receipt
func1 sizeof(ints)=40
func2 sizeof(ints)=8
```

Structures

When a `struct` is passed to a function, everything inside is copied, including arrays

This means an array wrapped in a `struct` is actually passed by value!

- In contast to normal arrays, which are passed by pointer

Structures

We might get tired of writing `struct` over and over:

```
struct cc_receipt {
    float amount;
    char cc_number[16];
};

struct cc_receipt lunch_receipt;
struct cc_receipt dinner_receipt;
```

Structures

We can use typedef to make the type name shorter:

```
typedef struct { // no name here
    float amount;
    char cc_number[16];
} cc_receipt;    // name down here

cc_receipt lunch_receipt;
cc_receipt dinner_receipt;
```

Now the type simply cc_receipt instead of struct cc_receipt

Structures

Size of `struct` is *at least* the sum of the sizes of its fields

It can be bigger if the compiler decides to add "padding"

```
struct plane {
    int passengers;
    double cargo_weight;
};
```

Structures

```
$ gcc -c sizeof_plane.c -std=c99 -pedantic -Wall -Wextra
$ gcc -o sizeof_plane sizeof_plane.o
$ ./sizeof_plane
sizeof(struct plane) = 16
```

For obscure efficiency reasons, the compiler put 4 bytes of "spacer" between the `int` & `double`, making total size $= 16$

Structures

Structures can be defined in a nested way:

```
typedef struct {
    struct {    // this struct type doesn't have a name;
        int r;  // it's just used once to declare a
        int b;  // field named color
        int g;
    } color;
    struct {    // again, no name
        int x;
        int y;
    } position;
} pixel;

pixel p;
p.color.r = 255;
p.position.x = 40;
p.position.y = 50;
```