Lesson 3

Christian Schwarz, Jakob Krebs

11.11.2019

Contents

Source Code and Solutions

Sources and Solutions

- we publish all code written in this course at `https://github.com/jkrbs/c_lessons`
- we will publish example solutions of the tasks on same site
- send us questions or your solutions to c-lessons@deutschland.gmbh

Appendix to last time

Variable in switch statements

```c
switch(1){
    case 1:
        int x = 3;
        printf("%d\n", x);
}
```

(Compiler) **error**: a label can only be part of a statement and a declaration is not a statement

Solution:

```c
switch(1){
    case 1:{
        int x = 3;
        printf("%d\n", x);
    } break;
}
```

assert

`assert` is a function from the c standard library that is extremely useful for debugging:

```c
#include<assert.h>
void main(){
    assert(3 == 3); // ok
    assert(3 == 4); // RUNTIME error when in DEBUG mode
}
```

When in debug mode, assert will produce a **runtime** error if the expression does not evaluate to `true`. An assertion fail can be nicely caught by a debugger like gdb or an IDE.

Large C codebases oftentimes contain hundreds of asserts at critical points to ensure that the programmers assumptions about the state of the programm at the point of execution are correct.

When producing an optimized release binary, asserts are taken out by the compiler.

bool

We have been using `true` and `false` for quite some time now. There is a type that actually tries to encode just these two values:

```c
#include <stdbool.h> //not a native type

void main(){
    bool b = true;

    //BUT (no warnings!):
    b = 27;
    int x = false;
}
```

Since the smalles addressable unit is a byte, a `bool` still uses one byte, despite fitting in one bit. It documents the programmers intention nicely though. In C, `true` really means 1 (or in conditions: **not 0**) and `false` means 0.

Complex Data Types

Enumerations

Oftentimes in Code you want to differentiate between a fixed set of Options. Using Integer values for this can get confusing very quickly. That's why C has `enum`, which can be thought of as restricted integer types that alias certain integer values as constants.

```c
enum day_of_week{
    MONDAY = 1,
    TUESDAY, WEDNESDAY, THURSDAY,
    FRIDAY, SATURDAY, SUNDAY
}; // <- this trailing semicolon is necessary!

// this creates a variable called myday, whose value can
// be any of the enum elements whe defined above
enum day_of_week myday = TUESDAY;
```

If not specified, enum values are just **the previous value plus one**, beginning at 0. We can override values if we want, but be careful with the continuation afterwards.

Enumerations in Switch Statements

Enumerations and switch statements often go hand in hand. They are also one of the cases where we can make good use of the fallthrough behaviour.

```c
switch(myday){
    case SATURDAY:
    case SUNDAY: {
        puts("weekend :)");
    } break;
    default: {
        puts("its a weekday :(");
    } break;
}
```

Structs

The basic building block of all data handling in C is the `struct`. Essentially it's just the idea to bundle up multiple variables as a reusable group. Like enums, structs are types and you can have variables of that are **instances** of the struct.

```c
struct point{
    int x;
    int y;
}; // <- again, this trailing semicolon is necessary!

// this creates a variable called mypoint, which is an instance of
// the struct point
struct point mypoint;
```

You can access elements of a struct using the `.` operator.

```c
mypoint.x = 3;
printf("\%i\n", mypoint.y);
```

Shorthand Struct initialization

There exists a shorthand notation for creating instances of structs:

```c
struct point{
    int x;
    int y;
};

//this specifies the names explicitly
struct point myotherpoint = {x: 3, y: 5};

// this depends on the order of the struct members
struct point mypoint = {3, 5};

// !! but this doesn't work, the shorthand syntax
// is only available during declaration!
// mypoint = {3, 4};
```

Structs and Enums in Functions

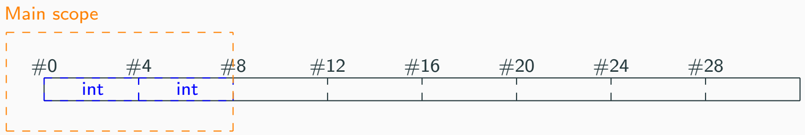
Structs and Enums can also be Function Parameters:

```c
void foo(enum day_of_week day, struct point p){
    //...
}
struct point p = {1,2};

//we can use enum values as literals, no need for a variable
foo(TUESDAY, p);

// !! again, this doesn't work
// foo(TUESDAY, {1,2});
```

pointers

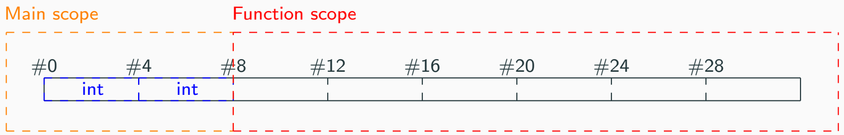
Memory

- You have two int variables in your main function.



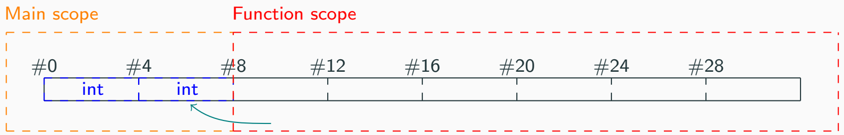
Memory

- You have two int variables in your main function.
- Now you call a function



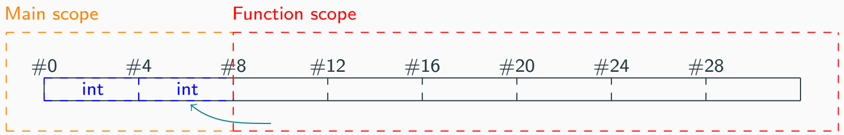
Memory

- You have two int variables in your main function.
- Now you call a function
- You want to change the value of a variable in the main scope



Memory

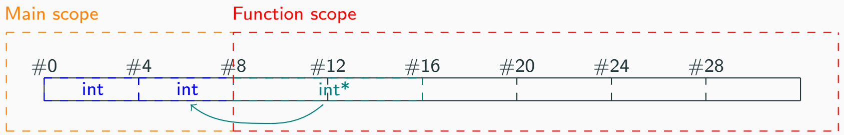
- You have two int variables in your main function.
- Now you call a function
- You want to change the value of a variable in the main scope



- You'll have to pass the address of this variable

Memory

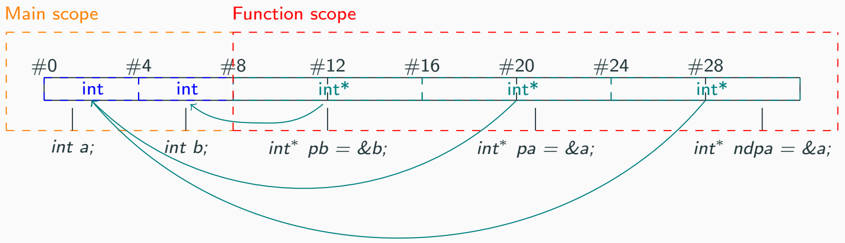
- You have two int variables in your main function.
- Now you call a function
- You want to change the value of a variable in the main scope



- You'll have to pass the address of this variable
- This address is stored in a *pointer* variable

Memory

- You have two int variables in your main function.
- Now you call a function
- You want to change the value of a variable in the main scope



- You'll have to pass the address of this variable
- This address is stored in a *pointer* variable

Operators

- To declare a Pointer, use the *dereference operator* *****
- To get the address of a variable, C offers the *address operator* **&**
- To access the variable a pointer points to, dereference it with the *dereference operator* *****

```c
int a = 42;
int *pa;     /* declare an int pointer*/
pa = &a;     /* initialize pa as pointer to a */
*pa = 13;     /* change a */
```

Increment and decrement

If you want to increment or decrement the variable a pointer points to, you have to use Parentheses.

```
int a = 42;
int *pa = &a;     /* define pa as pointer to a */
(*pa)++;         /* increment a */
(*pa)--;         /* decrement a */
```

If you had not used the parentheses, you would have in-/decremented the pointer, not the variable it points to.

Returning pointers

Pointers can be return values, too.

But

```c
int *someFunction(void) {
    int a = 42;
    return &a;
}
void main(){
    *someFunction() = 12;
}
```

Segmentation Fault

```
$ ./a.out
Segmentation fault (core dumped)
```

- What did just happen?

A segmentation fault is very common when working with pointers.
It means you were trying to write on memory your program didn't own.

argc and argv

You can pass strings to your program from the command line:

```
$ ./a.out string1 longer_string2
```

You

will have to use an alternative definition of *main()*:

```c
int main(int argc, char *argv[]) {
```

- The arguments are stored in *argv*[1]
- *argv* is an array of pointers to the first character of a string
- **Caution:** *argv[0]* is the name by which you called the program
- *argc*[2] is the number of strings stored in *argv*

[1]Short for *argument value*
[2]Short for *argument count*

Nested Structs

- Structs can be forward declared like functions: `struct foo;`
- But instances can't be declared or used before they are defined.
- What works is declaring pointers to the forward declared struct. Since pointers have a constant size, the compiler doesn't need to know about the struct.

```
1 struct b;
2 struct a{
3     //struct b bar; //! compiler error
4     struct b* b_ptr; //works
5 };
6 struct b{
7     struct a foo; //works, since a is defined
8 };
```

These rules have the side effect of preventing an interesting case: What would be the `sizeof(struct a)` if line 3 was legal?

Arrays

Arrays are a contiguous block of memory filled with instances of the array type. We actually have been using them already but didn't even notice: we call `"foo"` a string, but it's really just an array of chars:

```c
char foo[3] = {'f', 'o', 'o'};
```

The code above creates an array called foo of three chars. Like with structs, the initialization syntax is not available for assignment. (For the string case, this is a slight simplification).

We can also have the compiler figure out the size during initialization:

```c
int bar[] = {1, 2, 3};
```

This is semantically equivalent to writing 3 in the array brackets.

The combined size of bar is `3 * sizeof(int)`. Assuming `int` is 4 bytes large, `sizeof(bar)` $= 3 * 4 = 12$ [Bytes].

Accessing Array Elements

You can access array elements like this:

```c
int bar[3] = {0, 1, 2};
bar[0] = 12; // ! the indexing is zero based
printf("%d\n", bar[1]);
```

We can iterate over arrays like this:

```c
//we don't have to initialize our arrays
int bar[5];
for(int i = 0; i < sizeof(bar) / sizeof(bar[0]); i++){
    bar[i] = 2 * i;
    printf("%d\n", bar[i]);
}
```

Like with structs, we can use named initialization:

```c
int bar[] = {[9] = 4, [1] = 7, [2] = 1};
```

Arrays decaying to pointers

What if we want to pass an array to a function?

```c
void foo(int arr[3]); // ?!!
```

Bad idea. It compiles, but what the compiler **really** sees is this:

```c
void foo(int* arr);
```

Really:

```c
void foo(int arr[3]){
    arr[12] = 27; // this is "fine", no warnings!
    assert(sizeof(arr) == sizeof(int*)); // true !!!!!11eleven
}
```

Why does it do this? Because C is an old language. Does this make any sense at all? Kind of. An array is really just a region in memory. And a pointer **points** to a region of memory.

Arrays decaying to pointers 2

What C does here is basically assume you wanted a **Reference** to the array, **not a Copy** of the array. The same conversion happens when you assign an array to a pointer. An array itself is simply not assignable. What if you wanted a copy? Wrap the array in a struct and pass that.

But since C thinks of arrays as pointers, it allows us to do this:

```c
void foo(int* arr){
    arr[27] = 12;
}
```

This is all fine if you passed an array with at least **28** elements, but what if it was smaller? Well let's always check then:

```c
void foo(int arr[]){
   for(int x = 0; i< sizeof(arr) / sizeof(arr[0]); i++){
      printf("%d\n", x); // !! potential segfault
   }
}
```

Living with Arrays decaying to pointers

What happens in reality is quite simple:

```c
#include <stddef.h> //for size_t

void foo(int* arr, size_t arr_len){
    //iterate until we reach arr_len...
}

void main(){
    int arr[] = {1,2,3};
    //just always store the length of the array
    size_t arr_len = sizeof(arr) / sizeof(arr[0]);
    foo(arr, arr_len);
}
```

Living with Arrays decaying to pointers 2

Alternatively:

```c
void foo(int* arr_start, int* arr_end){
    for(int* ip = arr_start; ip != arr_end; ip++){
        *ip = 12;
    }
}

void main(){
    int arr[] = {1,2,3};
    int* arr_end = arr + (sizeof(arr) / sizeof(arr[0]));
    foo(arr, arr_end);
}
```

Pointer arithmetic

Noticed how we 'added' to an array on the last slide? That's because the array decayed to a pointer again (like it always does when you do anything with it). But how can we add to pointers?

```c
char x = 'x';
char* xp = &x; //0x7fffd164611c
char* xpp = x + 1;  //0x7fffd164611c + 1 = 0x7fffd164611d
```

But:

```c
int x = 1;
int* xp = &x; //0x7fffd164611c
char* xpp = x + 1;  //0x7fffd164611c + 4  !!!
```

Here arrays being pointers comes up agin. Adding `x` to a `type*` really means adding `x * sizeof(type)`.

Pointer subtraction

You can also subtract two pointers. `type* - type*` really means `(type* - type*) / sizeof(type)`. Because of this, we can use pointer subtraction to get the number of elements in an array (often called length, differentiating from size):

```c
void foo(int* arr_start, int* arr_end){
   int arr_len = arr_end  - arr_start; //number of elements
}

void main(){
    int arr[] = {1,2,3};
    // !notice: arr_end points AFTER the last element
    // NEVER DEREFERENCE arr_end
    int* arr_end = arr + (sizeof(arr) / sizeof(arr[0]));
    foo(arr, arr_end);
}
```