C-Lessons

Pointers and Memory

Lecturers: Mirko Jantschke, Pascal Scholz

9. Januar 2019

Contents

Pointers

RGB

Consider a function that calculates the RGB values of a hex color string:

```
int calcRGB(char hexString[]) {
    ...          /* converting hexString into RGB values */
    return ???;
}
```

- It is not possible to return 3 values.

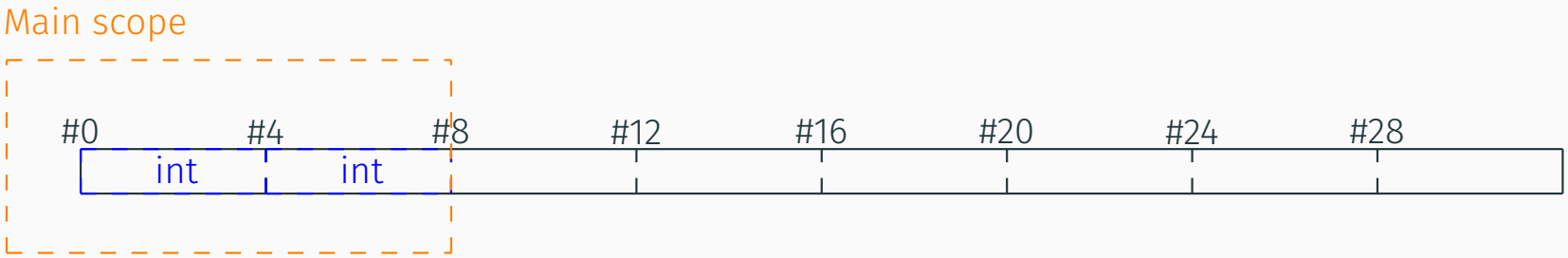
We could write 3 different functions:

```
int calcR(char hexString[]) { ... } /* returns R value */
int calcG(char hexString[]) { ... } /* returns G value */
int calcB(char hexString[]) { ... } /* returns B value */
```

Or we declare the 3 variables before the function call and just tell the function were to put the values.

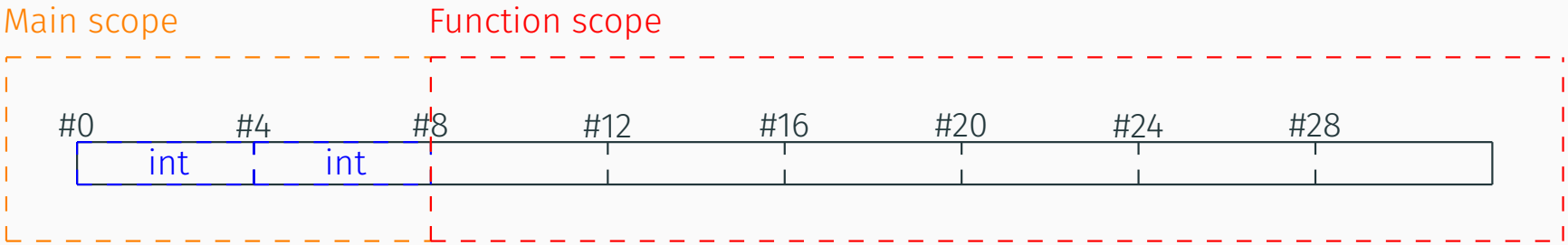
Memory again

- You have two int variables in your main function.



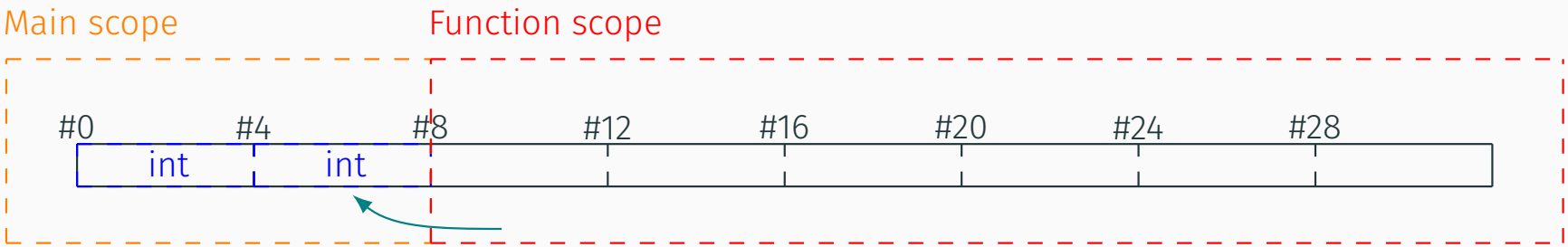
Memory again

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



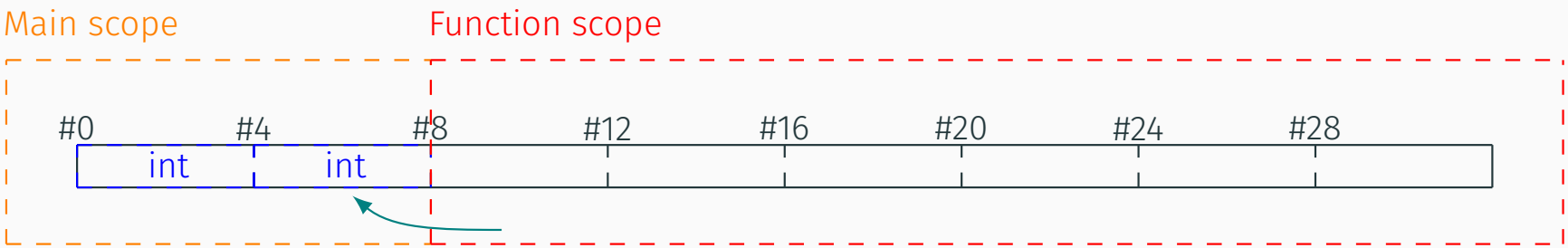
Memory again

- 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 again

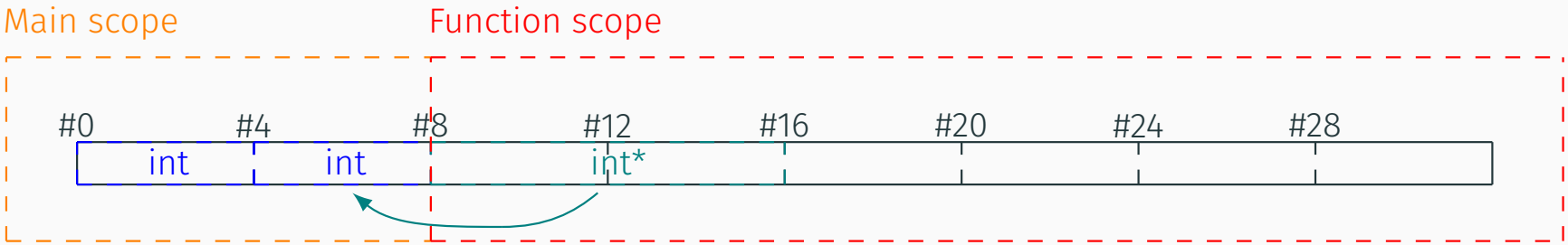
- You have two int variables in your main function.
- Now you call a function
- You want to change the value of a variable in the main scope



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

Memory again

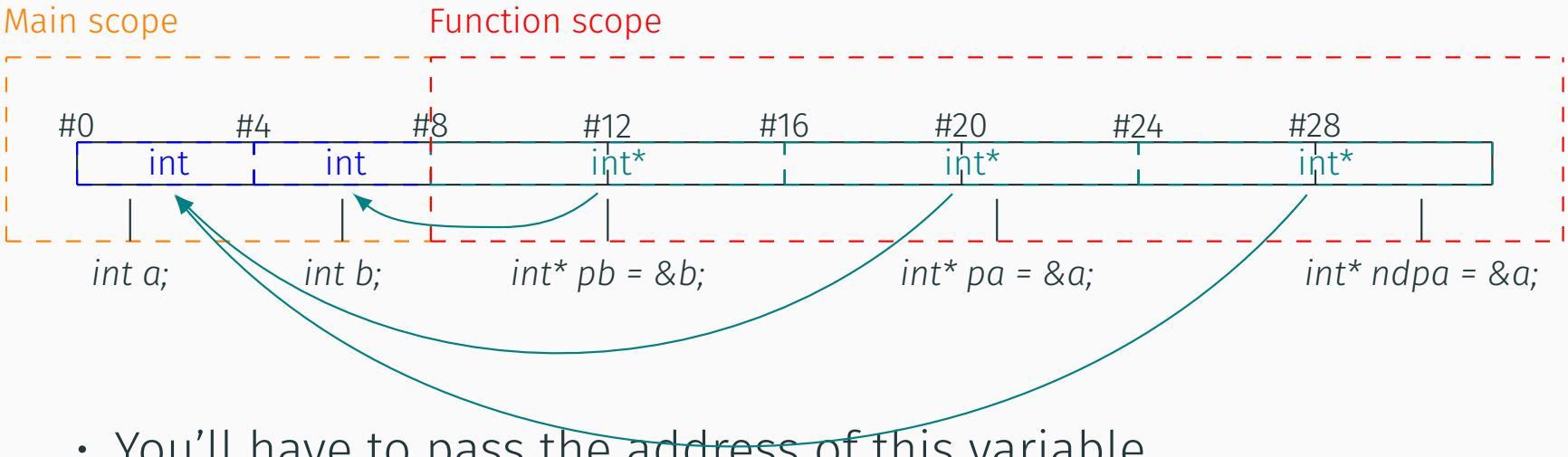
- You have two int variables in your main function.
- Now you call a function
- You want to change the value of a variable in the main scope



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

Memory again

- 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
- This method is called *call by reference*

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.

```c
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. Congratulations, you just invented pointer arithmetic but we will talk later about that.

Back to RGB

Now we can think of the RGB function as one function, taking the hexString and 3 Pointers:

```
void calcRGB(char hexString[], int *r, int *g, int *b) {
    ...
    *r = calculatedRValue;
    *g = calculatedGValue;
    *b = calculatedBValue;
}
```

Call it with

```
int r, g, b;
calcRGB("ffffff", &r, &g, &b);
```

- You now should understand how scanf works.

Returning pointers

Pointers can be return values, too.

But

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

- Dafuq did just happen?

Pointer arithmetic

p++

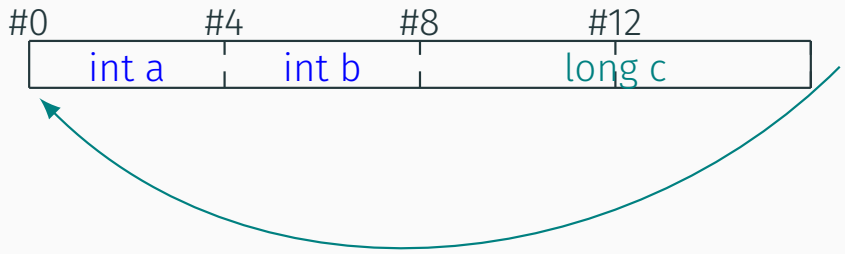
You can in-/decrement a pointer. If you do so, the address it points to will change.
The address changes by the size of the pointer type.

```
int a, b;
long c;
int *p = &a;
p++;
p++;
p++;
```

p++

You can in-/decrement a pointer. If you do so, the address it points to will change.
The address changes by the size of the pointer type.

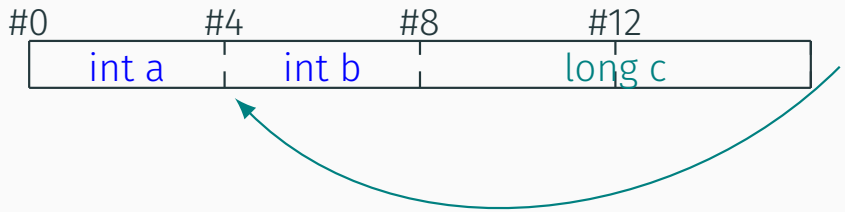
```
int a, b;
long c;
int *p = &a;
p++;
p++;
p++;
```



p++

You can in-/decrement a pointer. If you do so, the address it points to will change.
The address changes by the size of the pointer type.

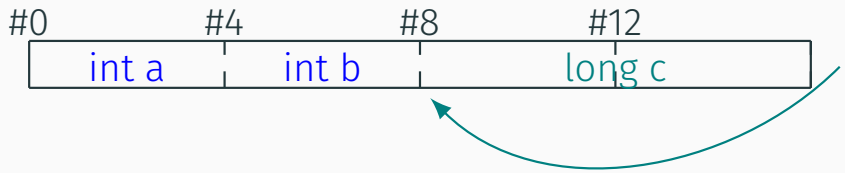
```
int a, b;
long c;
int *p = &a;
p++;
p++;
p++;
```



p++

You can in-/decrement a pointer. If you do so, the address it points to will change.
The address changes by the size of the pointer type.

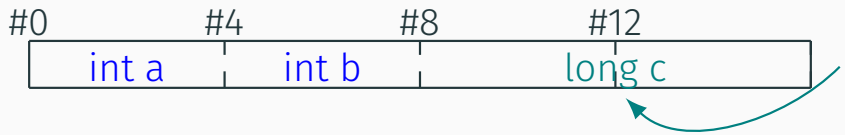
```
int a, b;
long c;
int *p = &a;
p++;
p++;
p++;
```



p++

You can in-/decrement a pointer. If you do so, the address it points to will change.
The address changes by the size of the pointer type.

```
int a, b;
long c;
int *p = &a;
p++;
p++;
p++;
```



- Since the pointer is of type *int* *, the target address moves only the size of *int*

Pointers and arrays

The identifier of an array can be considered a pointer.
This means we can consider the index as an offset for the pointer and access array elements trough pointer arithmetic:

```c
int leet[4] = {1, 3, 3, 7};
int *pleet = leet;
*(pleet++) = 2;
printf("%d %d\n", *pleet, *(pleet + 2));
```

- What is the output?

Pointers and arrays

The identifier of an array can be considered a pointer.
This means we can consider the index as an offset for the pointer and access array elements trough pointer arithmetic:

```
int leet[4] = {1, 3, 3, 7};
int *pleet = leet;
*(pleet++) = 2;
printf("%d %d\n", *pleet, *(pleet + 2));
```

- What is the output?

```
3 7
```

- Why?

Pointers and arrays

The identifier of an array can be considered a pointer.
This means we can consider the index as an offset for the pointer and access array elements trough pointer arithmetic:

```c
int leet[4] = {1, 3, 3, 7};
int *pleet = leet;
*(pleet++) = 2;
printf("%d %d\n", *pleet, *(pleet + 2));
```

- What is the output?

```
3 7
```

- Why?
  - Hint: Wasn't there a difference between *c++* and *++c* ?

Features of pointers

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()*:

```
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*

Misc.

As pointers hold addresses, print them as positive hexadecimal numbers!
Printf() has a special placeholder **%p** for that.

```c
int a;  /* assume address 42 */
int *b = &a;
printf("%p\n", b);  /* output: 0x2a */
```

Misc.

As pointers hold addresses, print them as positive hexadecimal numbers!
Printf() has a special placeholder **%p** for that.

```c
int a;  /* assume address 42 */
int *b = &a;
printf("%p\n", b);  /* output: 0x2a */
```

Caution: the * operator always refers to the next identifier:

```c
int *a, *b; /* two pointers */
int* a, b;  /* pointer and int */
```

Some people prefer *int* a* over *int *a*. This is fine, but avoid declarations as the one above.

Misc.

As pointers hold addresses, print them as positive hexadecimal numbers!
Printf() has a special placeholder **%p** for that.

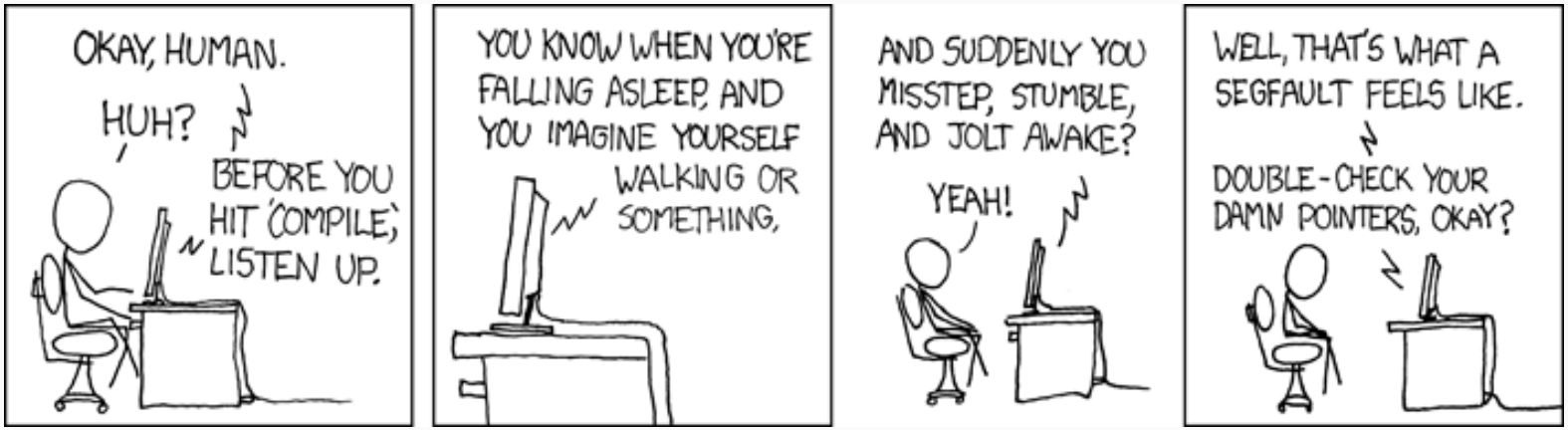
```c
int a;  /* assume address 42 */
int *b = &a;
printf("%p\n", b);  /* output: 0x2a */
```

Caution: the * operator always refers to the next identifier:

```c
int *a, *b; /* two pointers */
int* a, b;  /* pointer and int */
```

Some people prefer *int* a* over *int *a*. This is fine, but avoid declarations as the one above. And keep it consistent.

Segmentation Fault



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

Try to avoid those errors, backtracing is hard!

Related Task

Swap

Task as online

Write a function that swaps the values of 2 variables.

Experts: Write a function, that rotates 3 values in a given direction.

Calculating circles

Task as online

Write a function that takes the radius of a circle and returns the area and the circumference. Returns really means it have to make the result accessible and does no printing or something else.

Implement the circle as a structure use some constant for PI. But don’t take the structure as a parameter type.