Professorship for Big Geospatial Data Management
Department of Aerospace and Geodesy
Technical University of Munich

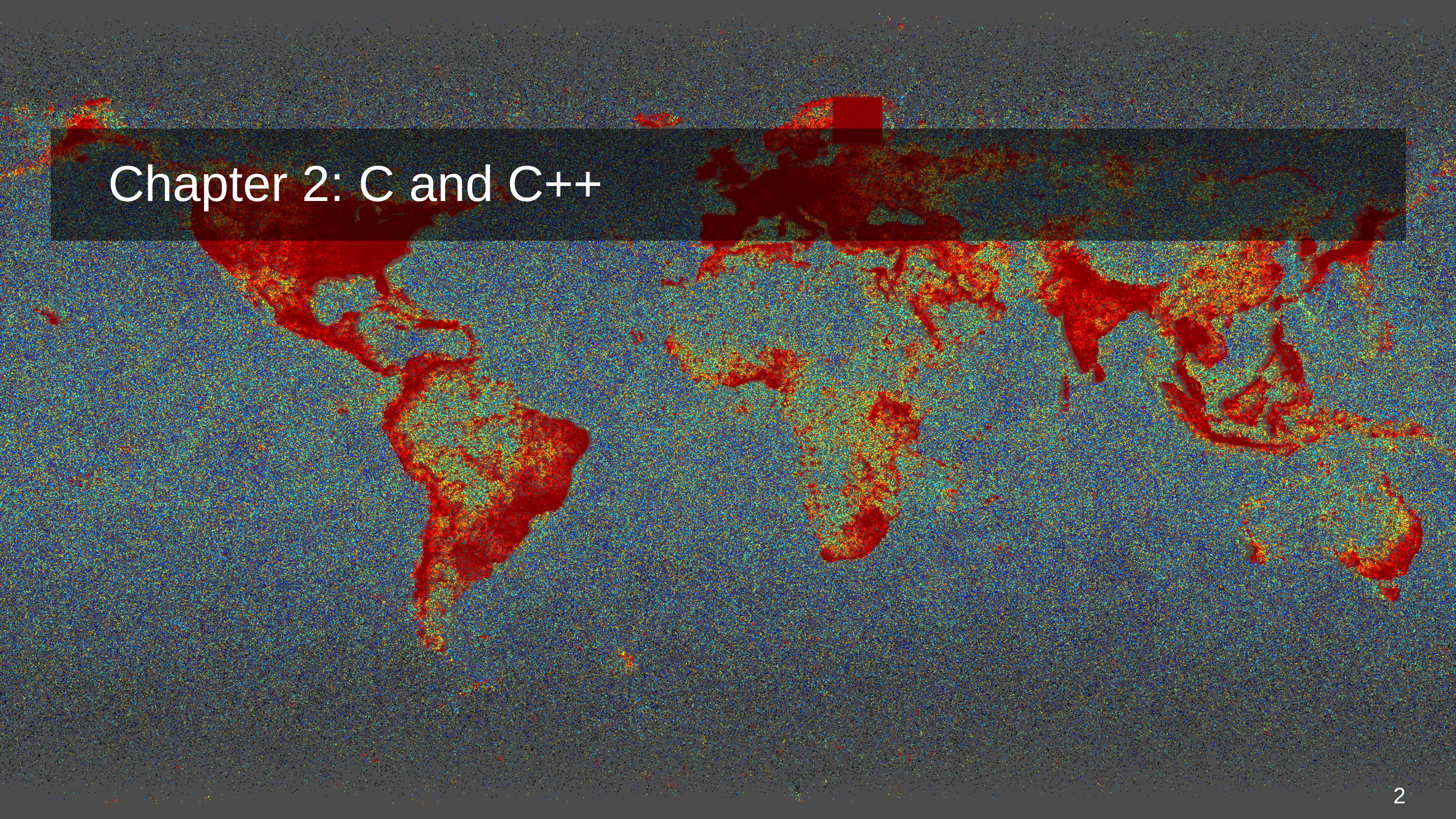
Computational Foundations I

Prof. Dr. Martin Werner

Technical University of Munich
TUM School of Engineering and Design
Department of Aerospace and Geodesy
Professorship of Big Geospatial Data Management

Mail: martin.werner@tum.de

Winter semester 2022/23



Chapter 2: C and C++

Why C and C++

Bjarne Stroustroup: C++ is a language for developing and using elegant and efficient abstractions.

➔elegance
➔efficiency
➔abstraction

Goals of C++:

- General Purpose Language: no specialization to specific use case or area
- No Oversimplification: At least allow for explicit exploitation of hardware by experts
- Leave no room for lower level language
- What you don‘t use, you don‘t pay for

How it began: ANSI C

Language	**Year**	**Developed By**
Traditional C	1972	Dennis Ritchie
K & R C	1978	Kernighan & Dennis Ritchie
ANSI C	1989	ANSI Committee
ANSI/ISO C	1990	ISO Committee

History & Facts

- C was invented to program Unix utilities (small programs for managing a computing system)
- At that time, new computers did not have compilers, hence, operating systems for new hardware were usually written in assembly language, a text representation of machine code.
- It was so clean and nice that the whole **Unix operating system** was reimplemented in this language
- The **Linux kernel** is as well implemented in plain C

A recent analysis also shows that C is the most energy-efficient programming language across a wide variety of problems, see https://greenlab.di.uminho.pt/wp-content/uploads/2017/10/sleFinal.pdf?utm_source=thenewstack&utm_medium=website&utm_campaign=platform

This makes it an ideal candidate for embedded systems.

C is also the language to program small microcontrollers (Atmel AVR, Arduino, etc.) where memory efficiency is essential.

Why then C++?

C++ has been invented mainly as an object-oriented extension to the C language. It provides higher levels of abstractions and automation, we will cover later.

What does C++ gain over C

- Reusable code
- High-Quality Libraries
- IT Security (e.g., automating pointers, semaphores, etc.)
- Readability
- Going beyond object orientation

Why is C still there?

- Energy efficiency
- Less to **no unclear constructs** (in C++, some things you can do are just not (fully) defined)
- Legacy Code

The C Programming Language

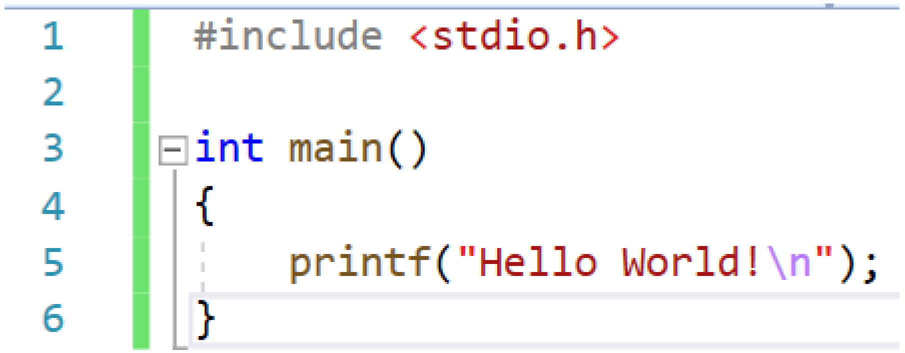
Hello World (this time on Linux and Windows)

C Programs are written in two types of files:

Source Code (*.c or *.cpp) contain the implementation of functions.

Header Files (*.h) contain signature information.

The signature consists of the return type and the number and types of arguments. It is also known as the prototype of a function. It is exactly the information that is needed to create the assembler code for calling the function and interpreting the stack afterwards, but does not include the function body itself.



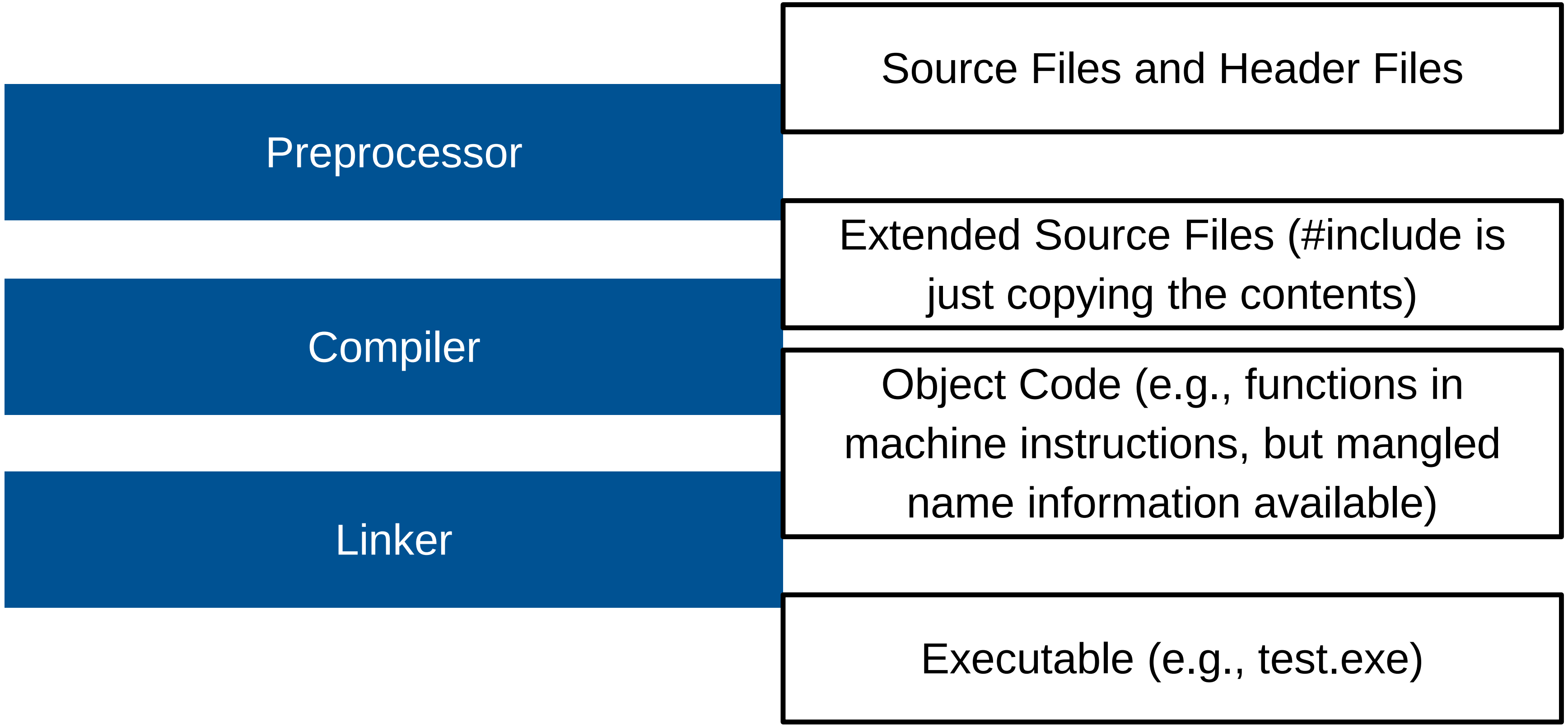
```
1   #include <stdio.h>
2
3   int main()
4   {
5       printf("Hello World!\n");
6   }
```

Line 1 includes the Stanard I/O Library, which contains the definition of printf.

Line 3 -6 implement a function which calls printf (whose signature is found in stdio.h)

A C program (command line program) start by running a function main. There are no instructions outside of functions!

Compilation Process



Source Files and Header Files
Preprocessor
Extended Source Files (#include is just copying the contents)
Compiler
Object Code (e.g., functions in machine instructions, but mangled name information available)
Linker
Executable (e.g., test.exe)

Notes

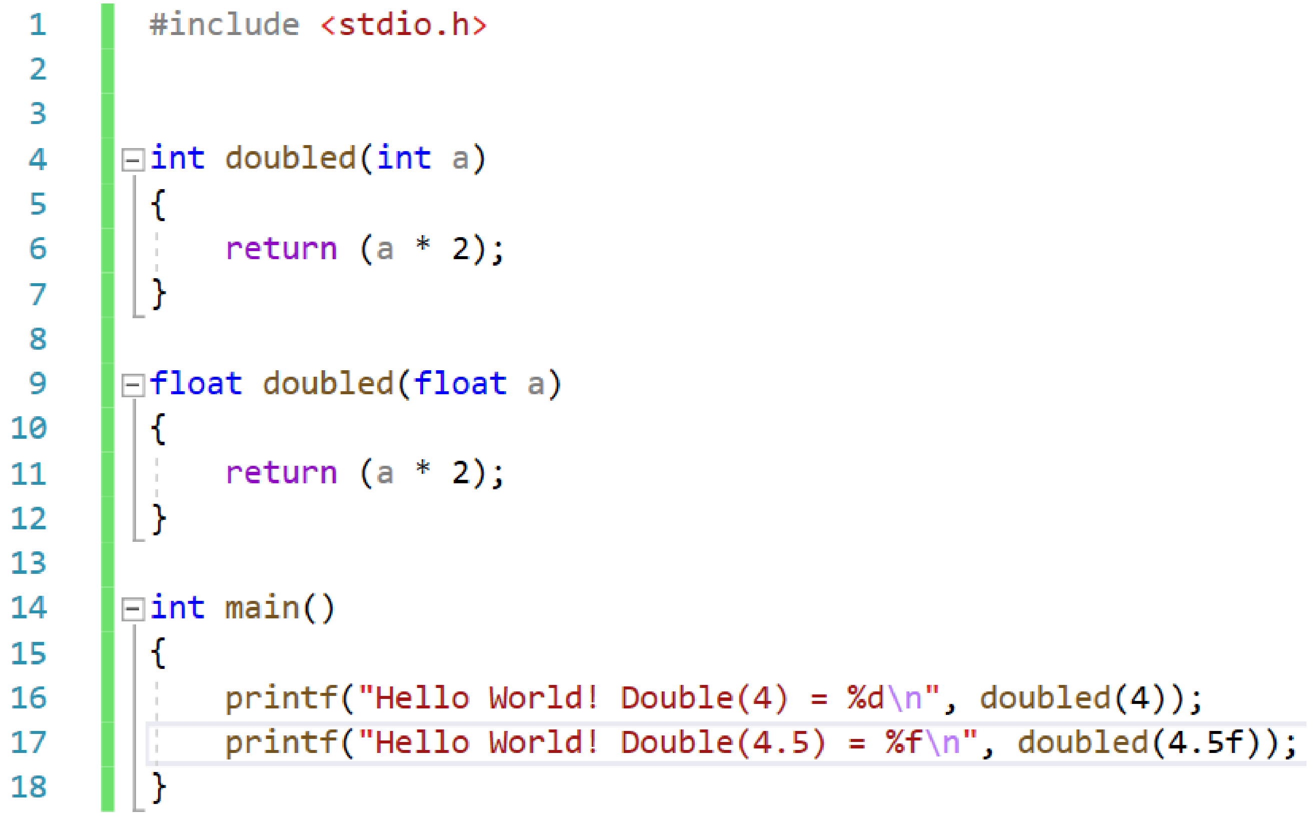
- In order to compile a C-file, **dependencies can be missing.** A C file can call a function that is not implemented in it. This includes operating system APIs as well as functions you implement yourself in a different file.

- In order to compile a file, however, the type information (signature) of all functions is needed. This can be used to derive a name (called mangled name) under which the function needs to be provided from a different C file or an external library

- In order to link a program, all names must exist. Otherwise you get a linker error

- If the program is an exetuable, it must contain exactly one main function. In general, a function with the same signature is not allowed to exist more than once. A function with the same name, but different arguments is allowed to exist. A function with the same name, but different return type is not allowed.

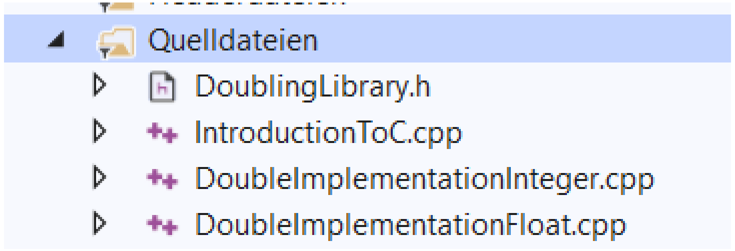
Example:



```c
#include <stdio.h>


int doubled(int a)
{
    return (a * 2);
}

float doubled(float a)
{
    return (a * 2);
}

int main()
{
    printf("Hello World! Double(4) = %d\n", doubled(4));
    printf("Hello World! Double(4.5) = %f\n", doubled(4.5f));
}
```

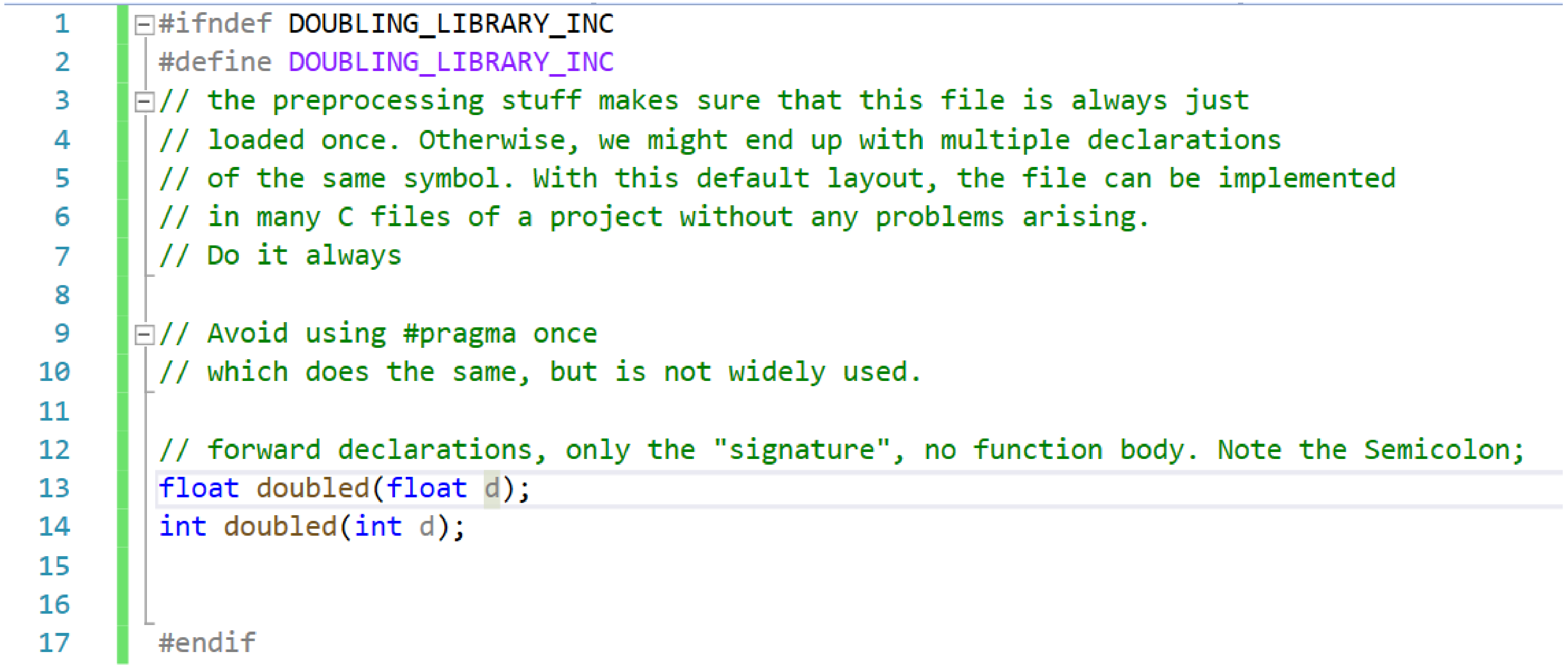
Complex Project Layout Example

We will split now our basic program into two libraries, both with a joint header file, and the implementation referring to the header file:



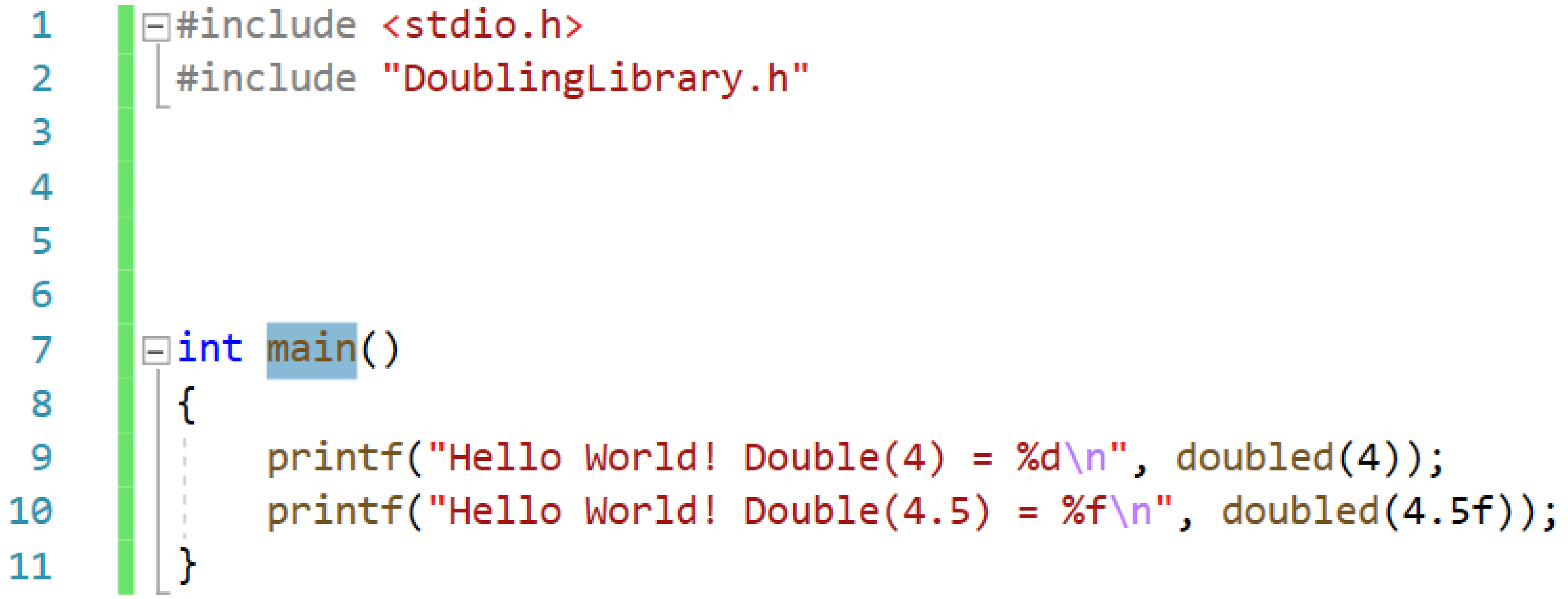
Quelldateien
DoublingLibrary.h
IntroductionToC.cpp
DoubleImplementationInteger.cpp
DoubleImplementationFloat.cpp

Header File



```c
#ifndef DOUBLING_LIBRARY_INC
#define DOUBLING_LIBRARY_INC
// the preprocessing stuff makes sure that this file is always just
// loaded once. Otherwise, we might end up with multiple declarations
// of the same symbol. With this default layout, the file can be implemented
// in many C files of a project without any problems arising.
// Do it always

// Avoid using #pragma once
// which does the same, but is not widely used.

// forward declarations, only the "signature", no function body. Note the Semicolon;
float doubled(float d);
int doubled(int d);


#endif
```

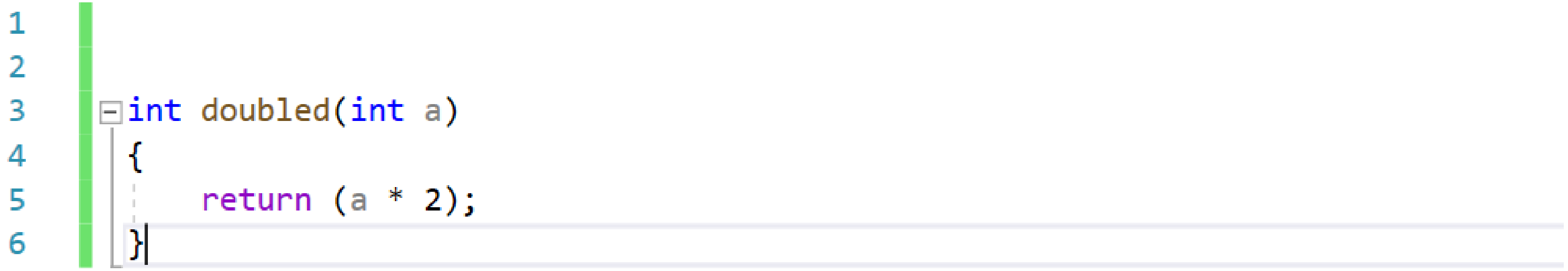
Main C File with reference to Header File



```c
#include <stdio.h>
#include "DoublingLibrary.h"




int main()
{
    printf("Hello World! Double(4) = %d\n", doubled(4));
    printf("Hello World! Double(4.5) = %f\n", doubled(4.5f));
}
```

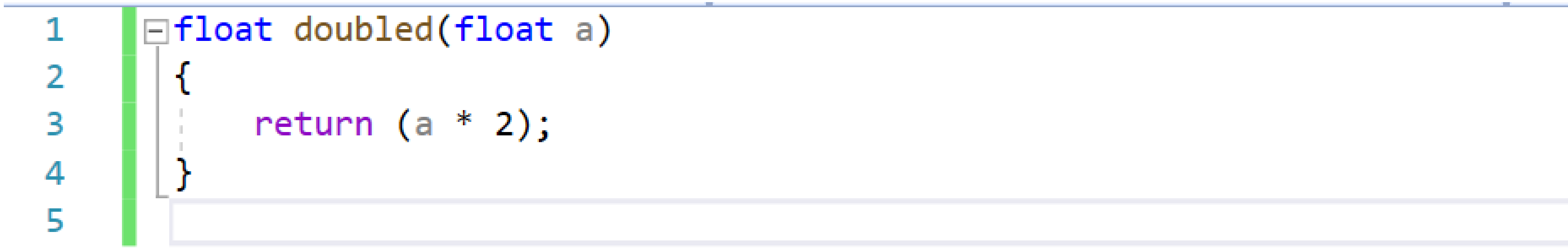
Implementations

First C file for Integer Implementation



```c


int doubled(int a)
{
    return (a * 2);
}
```

Second C file for Float Implementation



```c
float doubled(float a)
{
    return (a * 2);
}

```

Example (Linux + Mac Users)

On Linux, you just create the same text files (use any editor you like)

And then

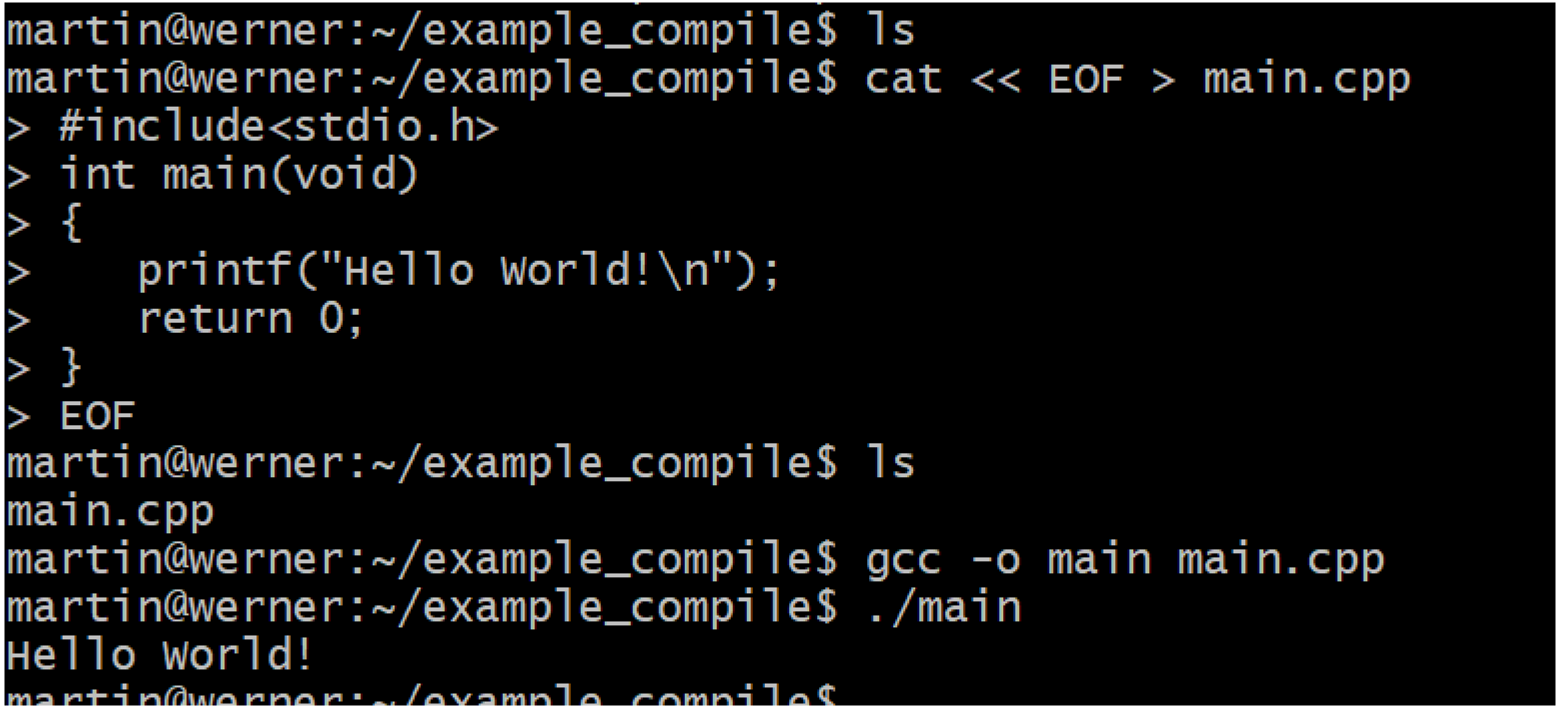
```
g++ -o <program name> -std=c++11 <list all c files to compile>
```

And run the program

```
./<program name>
```

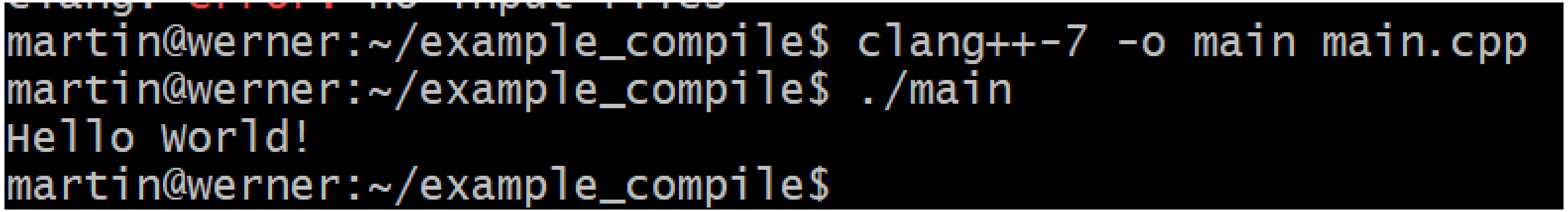
If gcc is not found, try cc which
should find all compilers.
For MAC users, look at clang.

Default example on Linux



```
martin@werner:~/example_compile$ ls
martin@werner:~/example_compile$ cat << EOF > main.cpp
> #include<stdio.h>
> int main(void)
> {
>     printf("Hello World!\n");
>     return 0;
> }
> EOF
martin@werner:~/example_compile$ ls
main.cpp
martin@werner:~/example_compile$ gcc -o main main.cpp
martin@werner:~/example_compile$ ./main
Hello World!
martin@werner:~/example_compile$
```

Using clang (default on Mac)



```
martin@werner:~/example_compile$ clang++-7 -o main main.cpp
martin@werner:~/example_compile$ ./main
Hello World!
martin@werner:~/example_compile$
```

Chapter 3.1: C Language Elements

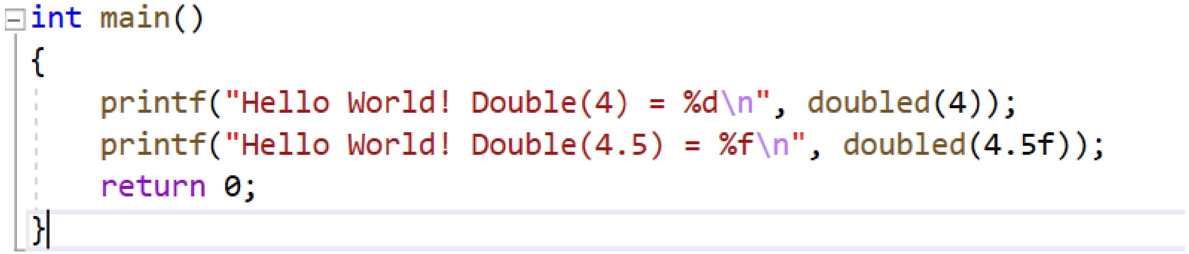
Cheat Sheet / Terms

Things	
Source File	A file that contains the implementation of functions
Header File	A small file that contains the signatures of external functions
Standard Library	A library that contains all default functions like printf and is silently linked into all programs
Project	A definition of the files that have to be combined. Not part of C/C++, meaning varies
Actions	
Compile	To translate from C code to machine code, but not resolving external variables and function calls. Input: C files, output .o files
Link	To combine multiple object files into a library or executable resolving dependencies and external symbols (functions or variables)
Debug Version	A program that contains information that can be used to analyze it while it is running and to react on fatal errors during runtime
Release Version	A program that omits all the information needed for debugging and is, therefore, by an order of magnitude faster

The Main Function

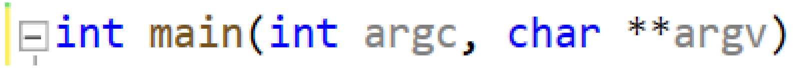
All programs start in the main function which can be used in two forms (more forms maybe accepted by your compiler, but should be avoided)

Main without arguments:



```c
int main()
{
    printf("Hello World! Double(4) = %d\n", doubled(4));
    printf("Hello World! Double(4.5) = %f\n", doubled(4.5f));
    return 0;
}
```

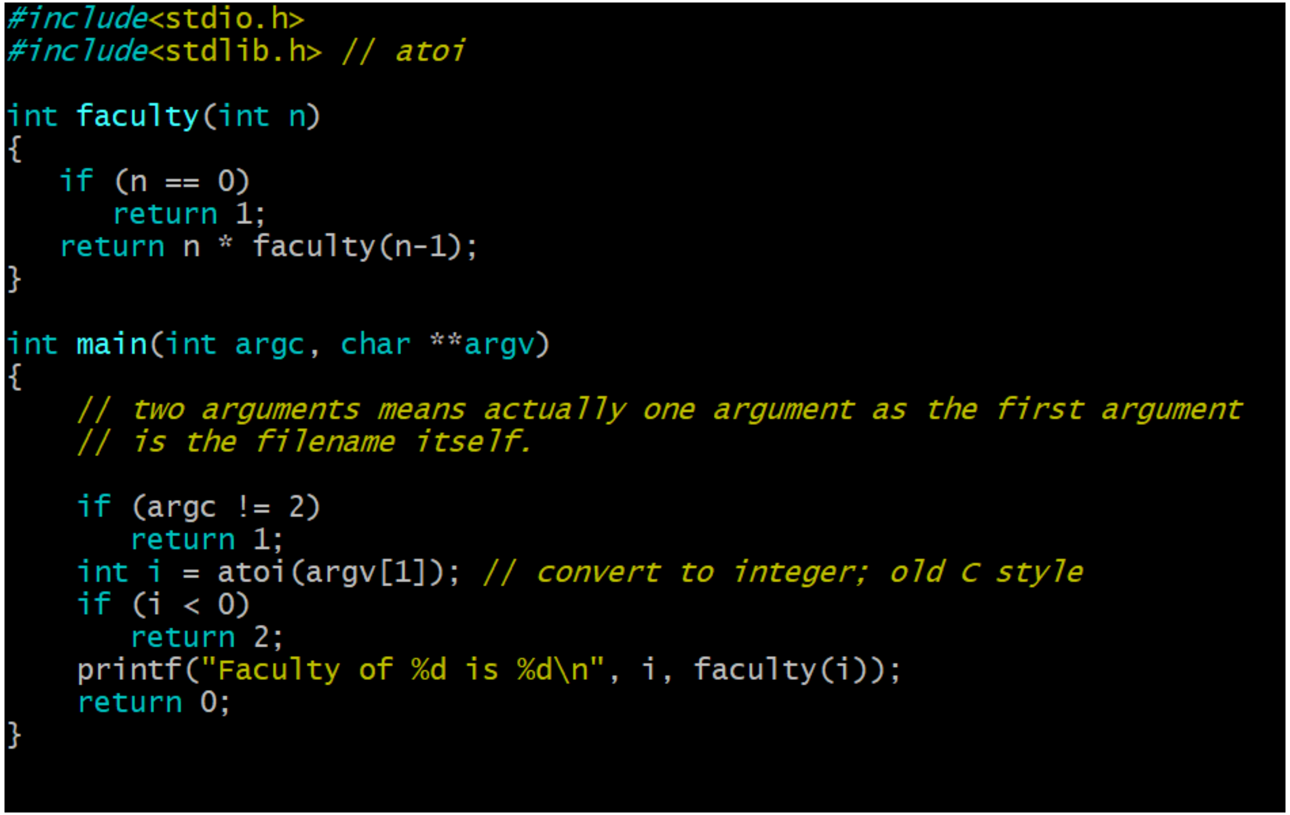
Main with arguments. Here, arguments can be given on the command line, but first, we need to learn about strings in C, we will come to it later.



```c
int main(int argc, char **argv)
```

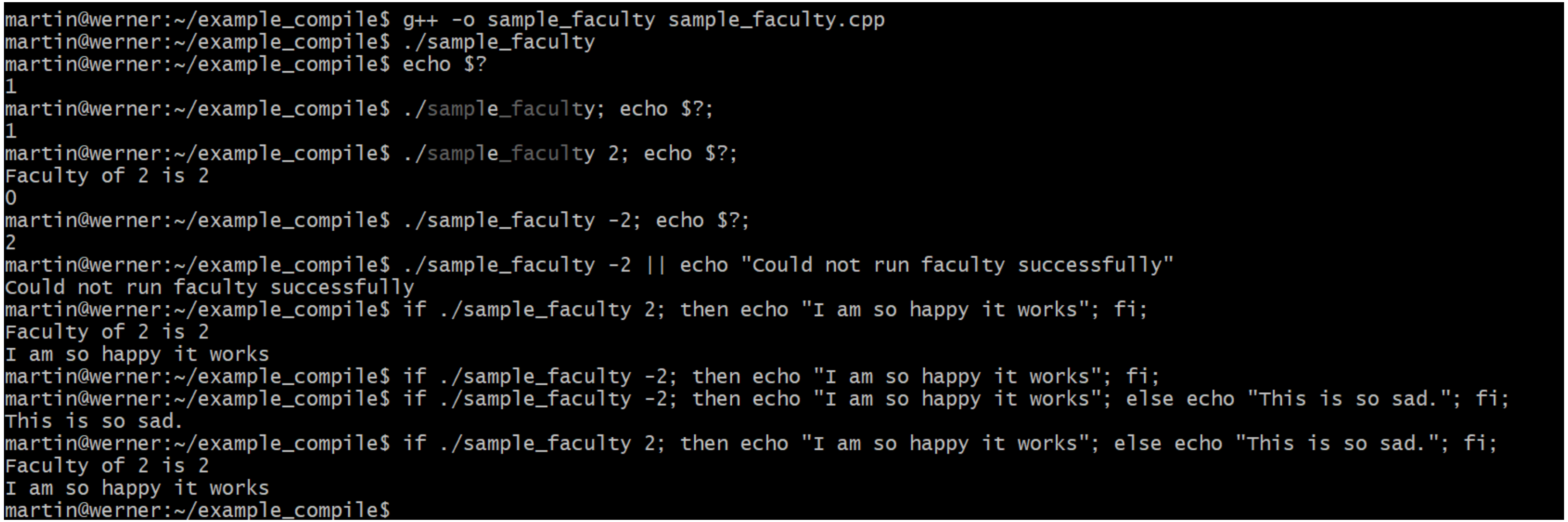
The return value signals information to the surroundings, e.g., the caller. According to POSIX, it should be zero if everything is fine and non-zero in case of error. This is mainly used on Linux / Unix

The return value of main



```
#include<stdio.h>
#include<stdlib.h> // atoi

int faculty(int n)
{
   if (n == 0)
      return 1;
   return n * faculty(n-1);
}

int main(int argc, char **argv)
{
    // two arguments means actually one argument as the first argument
    // is the filename itself.

    if (argc != 2)
       return 1;
    int i = atoi(argv[1]); // convert to integer; old C style
    if (i < 0)
       return 2;
    printf("Faculty of %d is %d\n", i, faculty(i));
    return 0;
}
```

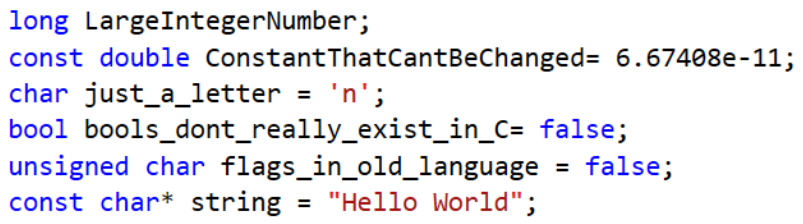
Ways to interact with the return value on Linux



```
martin@werner:~/example_compile$ g++ -o sample_faculty sample_faculty.cpp
martin@werner:~/example_compile$ ./sample_faculty
martin@werner:~/example_compile$ echo $?
1
martin@werner:~/example_compile$ ./sample_faculty; echo $?;
1
martin@werner:~/example_compile$ ./sample_faculty 2; echo $?;
Faculty of 2 is 2
0
martin@werner:~/example_compile$ ./sample_faculty -2; echo $?;
2
martin@werner:~/example_compile$ ./sample_faculty -2 || echo "Could not run faculty successfully"
Could not run faculty successfully
martin@werner:~/example_compile$ if ./sample_faculty 2; then echo "I am so happy it works"; fi;
Faculty of 2 is 2
I am so happy it works
martin@werner:~/example_compile$ if ./sample_faculty -2; then echo "I am so happy it works"; fi;
martin@werner:~/example_compile$ if ./sample_faculty -2; then echo "I am so happy it works"; else echo "This is so sad."; fi;
This is so sad.
martin@werner:~/example_compile$ if ./sample_faculty 2; then echo "I am so happy it works"; else echo "This is so sad."; fi;
Faculty of 2 is 2
I am so happy it works
martin@werner:~/example_compile$
```

Types

In C++, everything has to have a type and the types can‘t change.


```
long LargeIntegerNumber;
const double ConstantThatCantBeChanged= 6.67408e-11;
char just_a_letter = 'n';
bool bools_dont_really_exist_in_C= false;
unsigned char flags_in_old_language = false;
const char* string = "Hello World";
```

A **type** is a declaration of how much memory (in bytes) is needed and how the memory is interpreted (float, integer, string, signed or unsigned)

An **instance** is refers to a value of a given type in memory.

An **variable** is a named instance of a type reserving the memory and behaving as expected (e.g., numbers provide + and -, strings do not)

Every expression has a type.

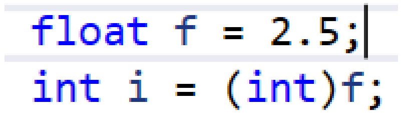
Fundamental Types and Their Interpretation

Type	**Keyword**	**Bytes**	**Range**
character	char	1	-128 .. 127
unsigned character	unsigned char	1	0 .. 255
integer	int	2	-32 768 .. 32 767
short integer	short	2	-32 768 .. 32 767
long integer	long	4	-2 147 483 648 .. 2 147 483 647
unsigned integer	unsigned int	2	0 .. 65 535
unsigned short integer	unsigned short	2	0 .. 65 535
unsigned long integer	unsigned long	4	0 .. 4 294 967 295
single-precision floating-point (7 Stellen)	float	4	1.17E-38 .. 3.4E38
double-precision floating-point (19 Stellen)	double	8	2.2E-308 .. 1.8E308

All of these integer types can be used to declare variables, return types, constants and arguments. They support all common operators (+,*,…), comparisons, etc. They are converted into each other in expressions as needed. Coversions lowering accuracy should raise warnings. Float to Int is always truncating.
Modulo is written as a%b (a modulo b)

Integer Types (ctd.)

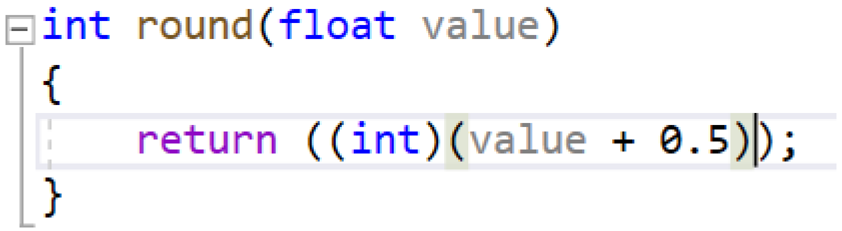
Explicit type conversion can be done (traditionally) using (<type>)



```
float f = 2.5;
int i = (int)f;
```

This should not be used in C++ unless you know what you are doing!

As integer conversions are truncating, we can write a round function as follows:

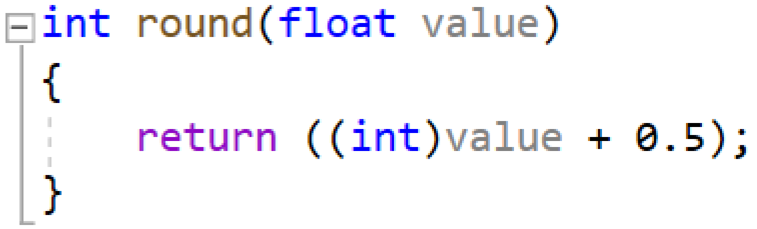


```
int round(float value)
{
    return ((int)(value + 0.5));
}
```

This example shall also remind you that type conversion is binding quite strongly. (see next slide),

Precedence of Type Conversion

This variant is wrong:



```
int round(float value)
{
    return ((int)value + 0.5);
}
```

It takes the float parameter (maybe 25.5), turns it into integer (25), adds 0.5 (25.5) and converts this all implicitly to integer for the return value (25)

Variable Scope

In traditional C, a variable can be local to a function (allocated on the stack during invocation, not visible outside the function, not existent before / after running the function). It is local, when it is declared within the brackets making up the function.

In traditional C, a variable can also be global. Global functions can be shared by declaring them as „external" in header files. That is, in a C file, one declares

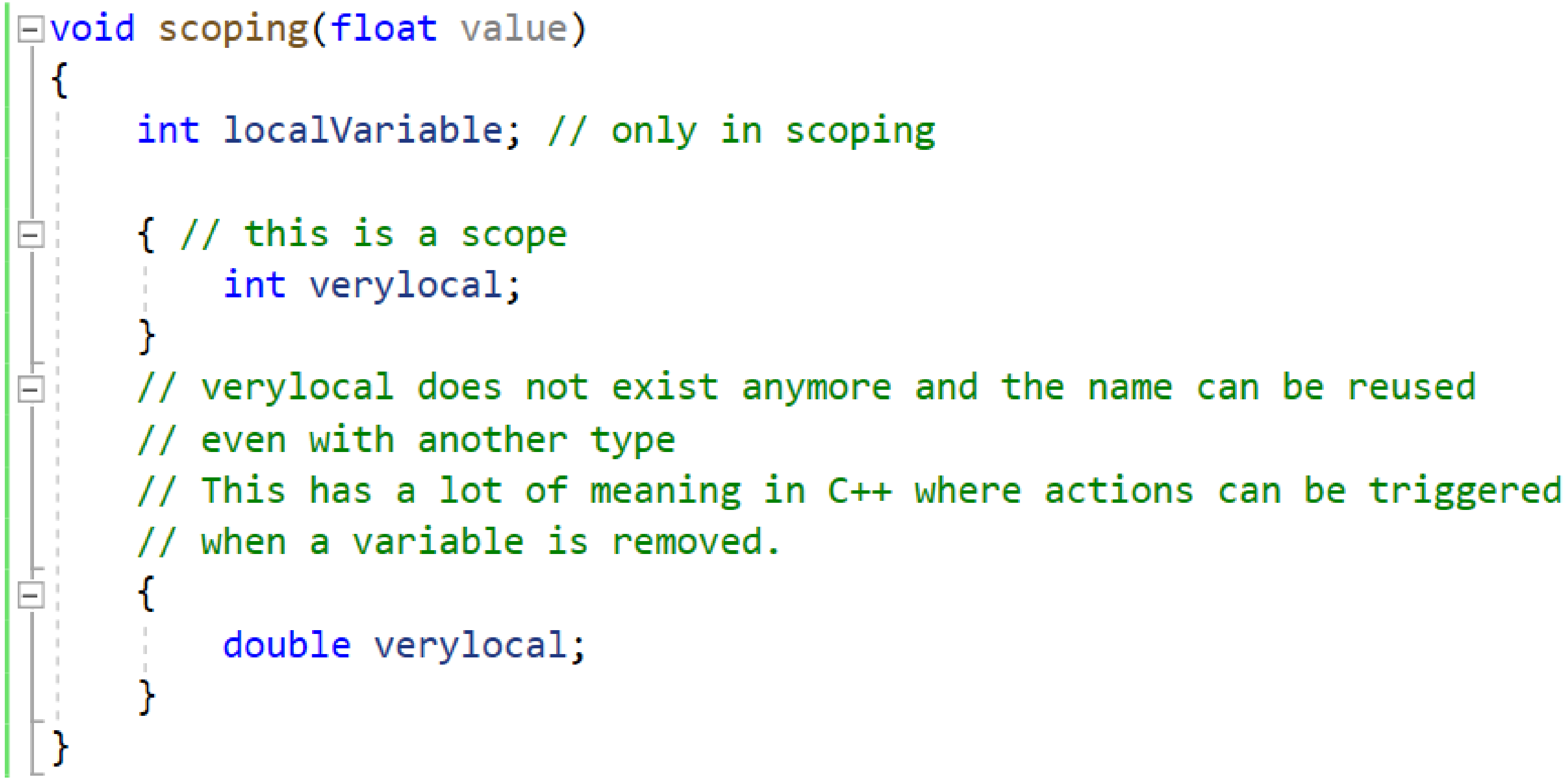
```
          int veryGlobal;
```

In the header, one cannot write `int veryGlobal;` because then the symbol would be doubled. One can, however, inform the compiler (similar to functions) that it should exist somewhere by writing

```
extern int veryGlobal;
```

In modern C++, variables can be even more local (and they are if you just use the C subset on Visual C++) to the current scope (next slide)

Variable Scopes



```
void scoping(float value)
{
    int localVariable; // only in scoping

    { // this is a scope
        int verylocal;
    }
    // verylocal does not exist anymore and the name can be reused
    // even with another type
    // This has a lot of meaning in C++ where actions can be triggered
    // when a variable is removed.
    {
        double verylocal;
    }
}
```

Chapter 3.2: C Control Structures

Control Structures in C

For the beginning, we need to implement sequence, conditional branch, and loops.

- Sequences are just concatenated statements;

```
first_thing();
second_thinkg();
x = third + as + an + expression;
```

Conditional Branch

The if statement evaluates an expression and compares it with the integer 1 representing true.

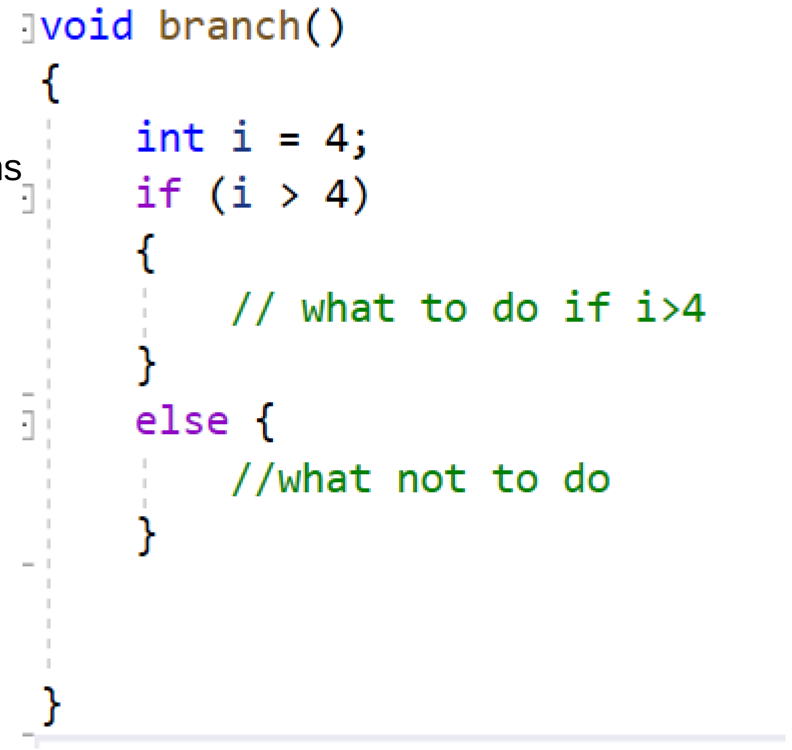
Comparisons are available, but also logical operations

! (expr) = not <expr>

Expr1 && Expr2	Logical AND
Expr 1 || Expr 2	Logical Or

Lazy evaluation applies usually from left to right. That is, for && if the first is wrong, the second is not evaluated.

Other branch constructs exist… Later ;-)



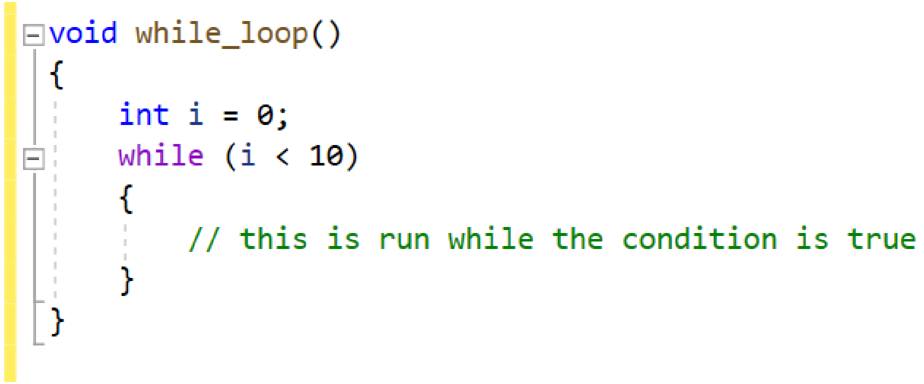
```
void branch()
{
    int i = 4;
    if (i > 4)
    {
        // what to do if i>4
    }
    else {
        //what not to do
    }


}
```

Loops

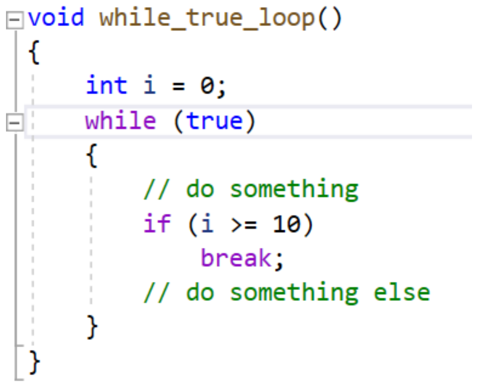
A while loop has a boolean expression and The body is run each time that at the beginning of the loop the condition is true.

A break statement is available to break out of a loop.

Note that if an if triggers a single expression or statement, the braces { and } can be left out.



```
void while_loop()
{
    int i = 0;
    while (i < 10)
    {
        // this is run while the condition is true
    }
}
```



```
void while_true_loop()
{
    int i = 0;
    while (true)
    {
        // do something
        if (i >= 10)
            break;
        // do something else
    }
}
```

The universal for loop

As C tries to be very near to a machine,
the for loop looks a bit complicated at first.

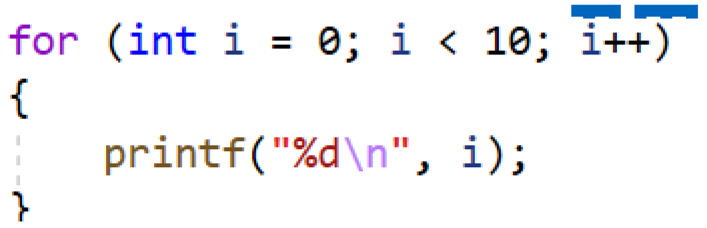
It contains three expressions.

- Initialization expression to be run before the loop
- Test to be tested before each loop body
- Loop Update to be run after each loop body

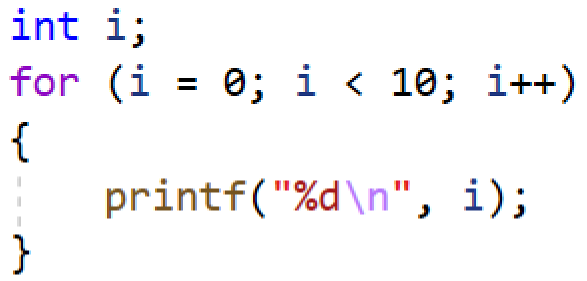
This is very flexible as they can be empty. But first a default for loop. Here `i++` just increments by one.

If you want the loop counter to remain visible (e.g., if you have smart break statements), you can move the int out of the for loop scope.

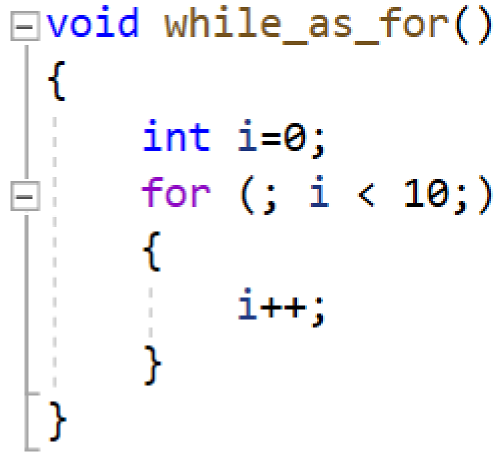
A while loop can be written as a for like so:



```
for (int i = 0; i < 10; i++)
{
    printf("%d\n", i);
}
```



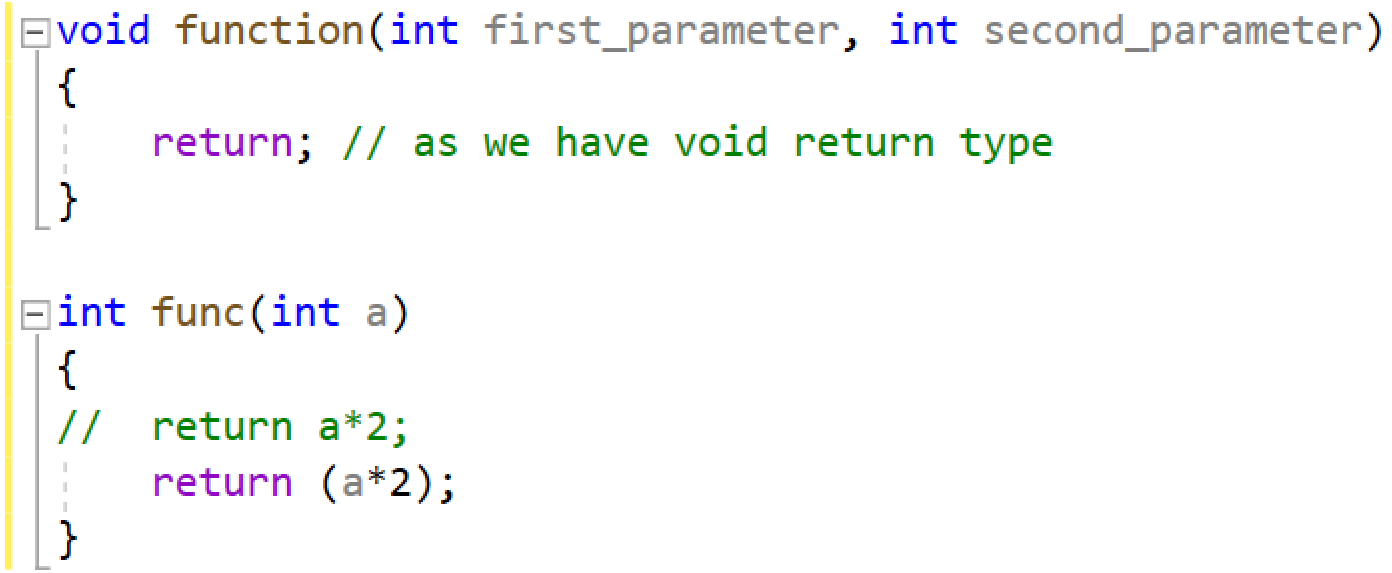
```
int i;
for (i = 0; i < 10; i++)
{
    printf("%d\n", i);
}
```



```
void while_as_for()
{
    int i=0;
    for (; i < 10;)
    {
        i++;
    }
}
```

Functions

A C function has a return value. The special type void (0 bytes) is used to declare that the function does not return a value. A value is returned by using the keyword return which can be used without brackets (it is not a function call). Brackets, however, don‘t harm as they just „bracket“ the expression.



```c
void function(int first_parameter, int second_parameter)
{
    return; // as we have void return type
}

int func(int a)
{
//  return a*2;
    return (a*2);
}
```

Task

Starting to Work with C/C++ is a real challenge. But when you master it, you are able to learn all programming languages with ease.

Some practical tips:

- Try to **compile as often as possible** (maybe after writing one line, F7 in Visual Studio)
- Always **read the first error first.** Many times, an error implies hundreds of messages out of which only the first one is actually relevant.
- Learn the **types table** and implement some of the algorithms we have seen so far.
- Formulate questions carefully (do screenshots in case you have a really complex matter)
- If you really **fail on something**, create a minimum package to share (e.g., the source code and the project) such that we can help you. Maybe put it onto Google Drive or somewhere else and put a link into Etherpad. We will try to help (in the given ressource constraints that 2 PhDs take care of 200 students).

3.3 Arrays and Pointer

Variables

Recall, that variables in C reserve memory of a given type and remain uninitialized until a first value is assigned.

```
int variable;
…
variable = 3;
```

One can as well directly initialize the variable in the declaration

```
int variable=3;
```

Such variables are usually placed on the stack and are available / defined within the current scope (global variables can be defined outside function. Every variable inside a function is scoped to the function and its lifetime ends as soon as the function ends.

Arrays

In the C family of languages, arrays can be declared with brackets. For example

int myArray[10];

Declares and reserves memory for an array with 10 elements number from 0..9 on the stack.
Warning: myArray[10] does not belong to the array!

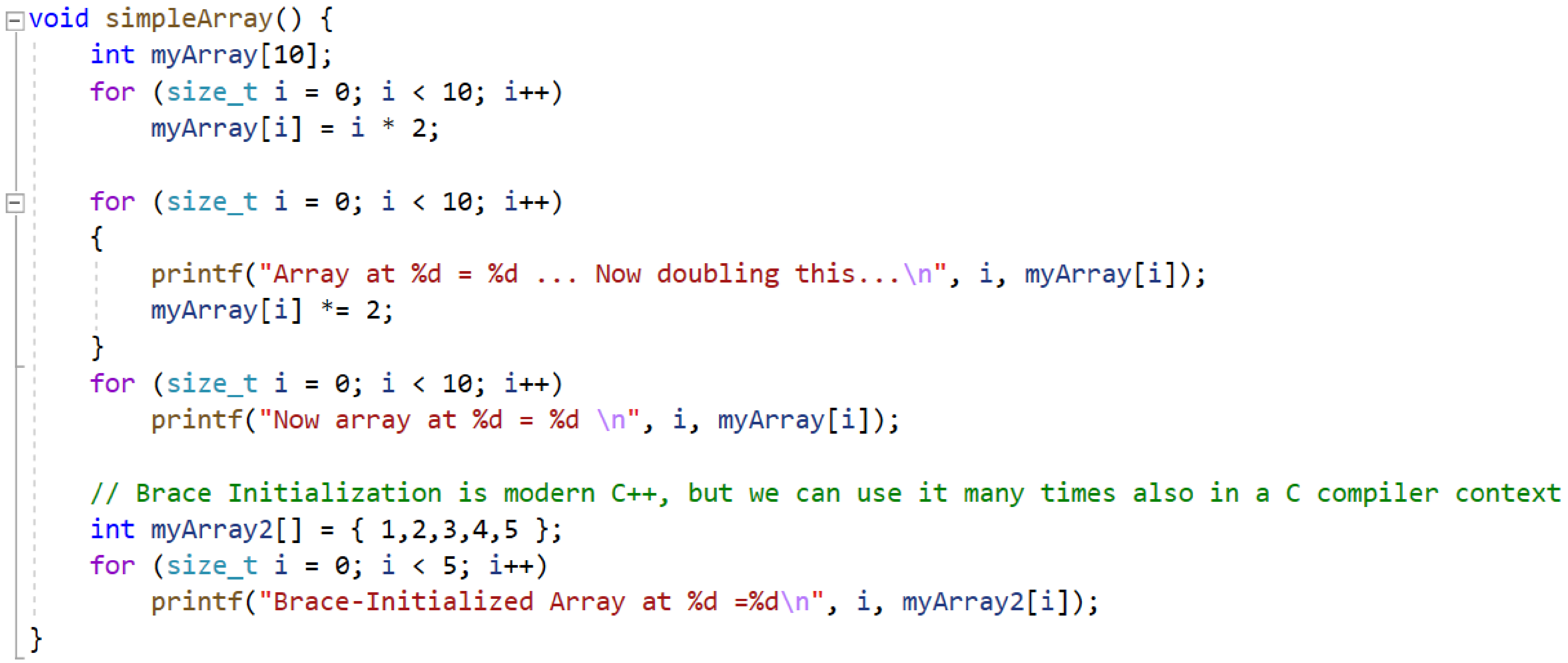
Example:

```
for (int i=0; i< 10; i++)
  myArray[i] = 2*i;
```

Initialization of Arrays is impossible in C, but the C++ extension of brace initilaiztion can be applied in modern compilters

```
int myArray[] = {1,2,3,4}
```

Example with Loops

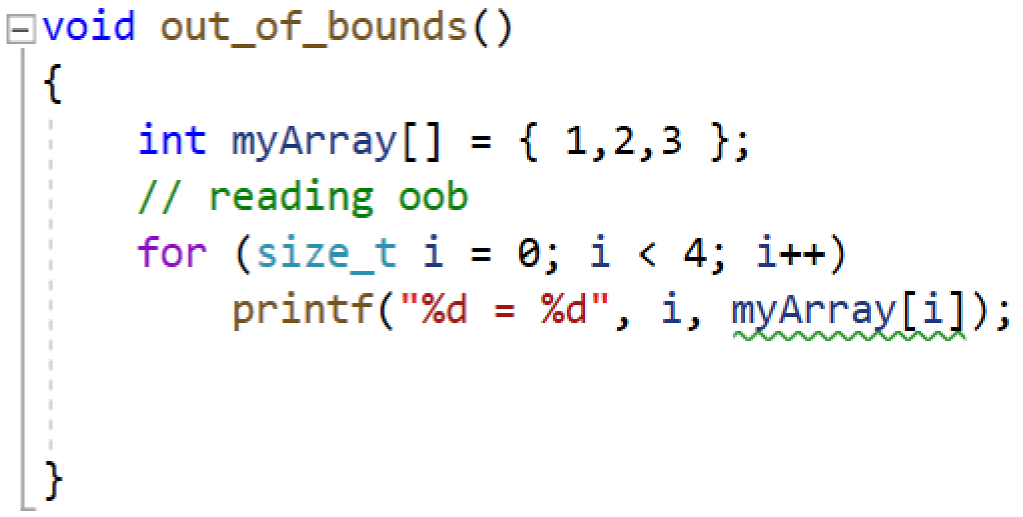


```
void simpleArray() {
    int myArray[10];
    for (size_t i = 0; i < 10; i++)
        myArray[i] = i * 2;

    for (size_t i = 0; i < 10; i++)
    {
        printf("Array at %d = %d ... Now doubling this...\n", i, myArray[i]);
        myArray[i] *= 2;
    }
    for (size_t i = 0; i < 10; i++)
        printf("Now array at %d = %d \n", i, myArray[i]);

    // Brace Initialization is modern C++, but we can use it many times also in a C compiler context
    int myArray2[] = { 1,2,3,4,5 };
    for (size_t i = 0; i < 5; i++)
        printf("Brace-Initialized Array at %d =%d\n", i, myArray2[i]);
}
```

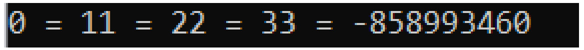
Out of Bounds

Warning! As C is optimized for performance, there is no bound checking in place. That is, if you try to access an element with an index larger than the reserved space, you end up somewhere else in the stack, for example the following snippet works.

Sometimes (depending on the memory layout, this code could as well crash with an access violation).

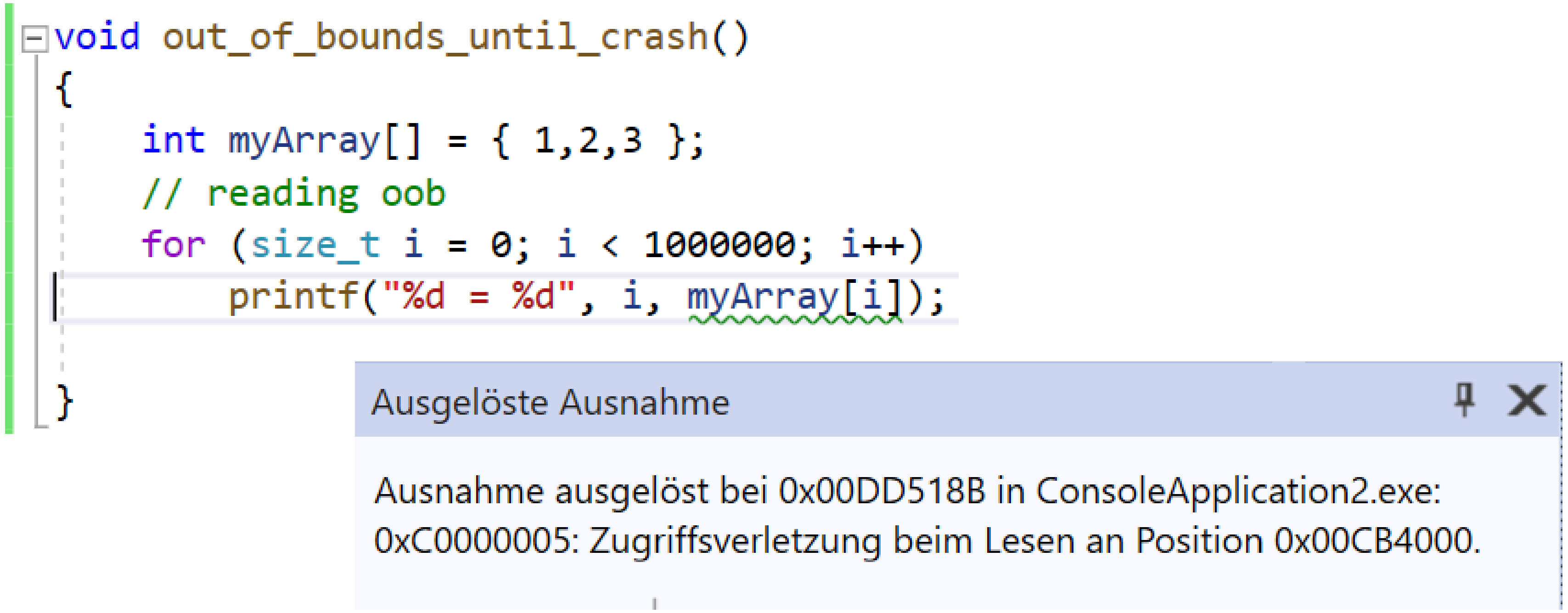


```c
void out_of_bounds()
{
    int myArray[] = { 1,2,3 };
    // reading oob
    for (size_t i = 0; i < 4; i++)
        printf("%d = %d", i, myArray[i]);


}
```



```
0 = 11 = 22 = 33 = -858993460
```

Access Violations



```
void out_of_bounds_until_crash()
{
    int myArray[] = { 1,2,3 };
    // reading oob
    for (size_t i = 0; i < 1000000; i++)
        printf("%d = %d", i, myArray[i]);

}
```

Ausgelöste Ausnahme

Ausnahme ausgelöst bei 0x00DD518B in ConsoleApplication2.exe: 0xC0000005: Zugriffsverletzung beim Lesen an Position 0x00CB4000.

On Linux and Mac, this exception is known as SEGFAULT (segmentation fault) and is one of the more difficult to repair problems. For deeply learning about memory debugging, consider learning about **valgrind. (Beyond the scope of an introduction!)**

How does this come to be?

In fact, in classical C, an array is not stored as a container with a certain length. Instead, we store a pointer to the beginning of the array. The length of the array needs to be taken care from the source code.

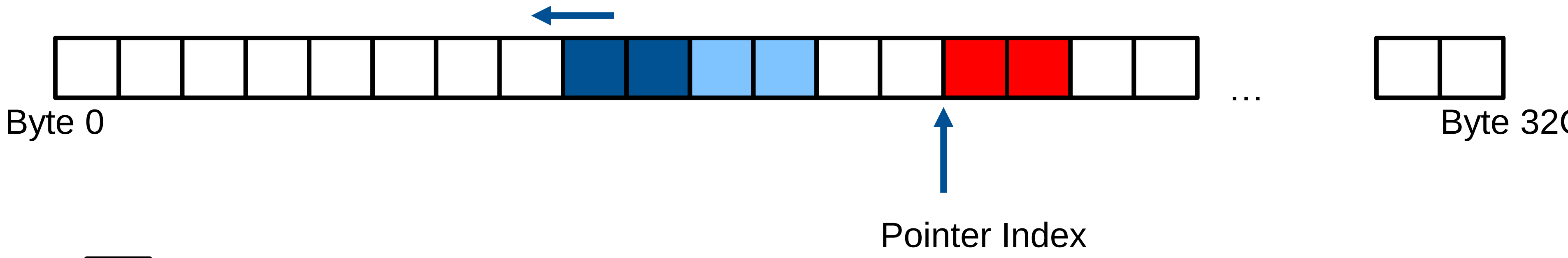
Note that this is one of the reasons why arrays are seldom used in high-level C++ code. There are alternatives!

The memory model of a basic computer is a global array (just like the Turing machine).

A location in this array is represented as an unsigned integer (64 bits for 64 bit machines).

C allows to **reference each memory address** with a type.
$\Rightarrow$ We can interpret any location in main memory as a certain type like int, char, long, double.
$\Rightarrow$ We can use the left and right neighboring cell in some situations. This is how arrays are built.

Basic Memory Model



🟦 Stack Cells (A stack is represented by the start and grows towards 0

🟥🟥 A two byte variable outside the stack („on the heap“); we learn how to declare those in a minute

🟦🟦 A two-byte variable on the stack (this is just a local variable)

In order to be able to access the red variable, we will usually have a local variable 64 bits essentially containing the location of the start of the memory of the variable. Such a variable is called a pointer.

Pointer

A variable is declared as in
int var = 4;

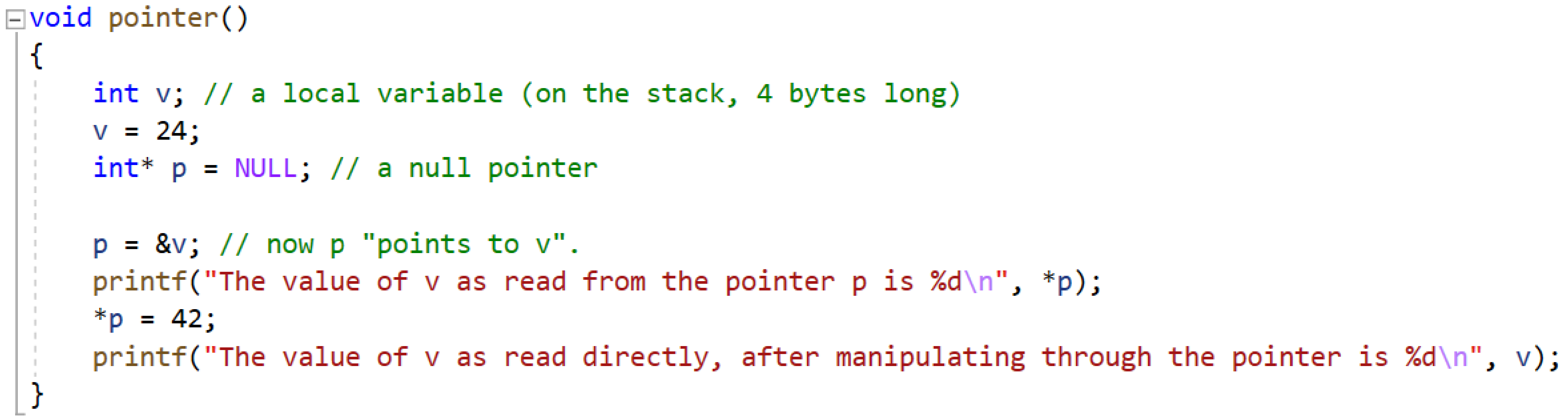
A pointer is declared as
int *p;

A pointer should be initialized to point to the start of memory (which is always „invalid“)
int *p=NULL;

The adress of a variable can be generated with the & operator:
p = &a;

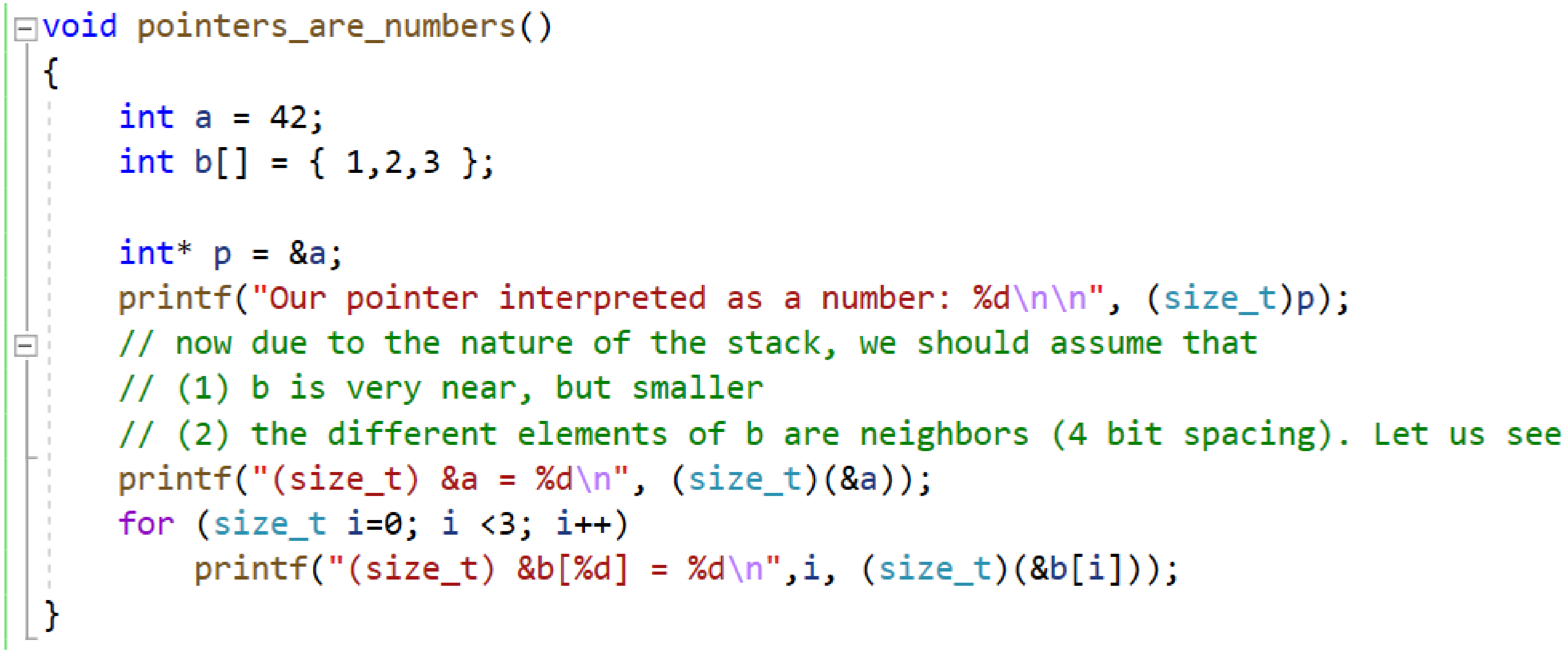
Afterwards ,the value of a as an integer can be referenced to as *p („where p points to“). This is called to dereference a pointer.
*p=8;

Pointer Example



```c
void pointer()
{
    int v; // a local variable (on the stack, 4 bytes long)
    v = 24;
    int* p = NULL; // a null pointer

    p = &v; // now p "points to v".
    printf("The value of v as read from the pointer p is %d\n", *p);
    *p = 42;
    printf("The value of v as read directly, after manipulating through the pointer is %d\n", v);
}
```

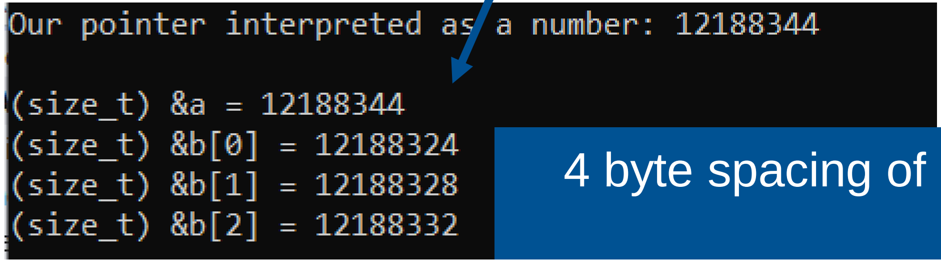
Pointers as Numbers



```
void pointers_are_numbers()
{
    int a = 42;
    int b[] = { 1,2,3 };

    int* p = &a;
    printf("Our pointer interpreted as a number: %d\n\n", (size_t)p);
    // now due to the nature of the stack, we should assume that
    // (1) b is very near, but smaller
    // (2) the different elements of b are neighbors (4 bit spacing). Let us see
    printf("(size_t) &a = %d\n", (size_t)(&a));
    for (size_t i=0; i <3; i++)
        printf("(size_t) &b[%d] = %d\n",i, (size_t)(&b[i]));
}
```

Result

a has a higher number than b
(declared before a)



```
Our pointer interpreted as a number: 12188344

(size_t) &a = 12188344
(size_t) &b[0] = 12188324
(size_t) &b[1] = 12188328
(size_t) &b[2] = 12188332
```

4 byte spacing of array elements

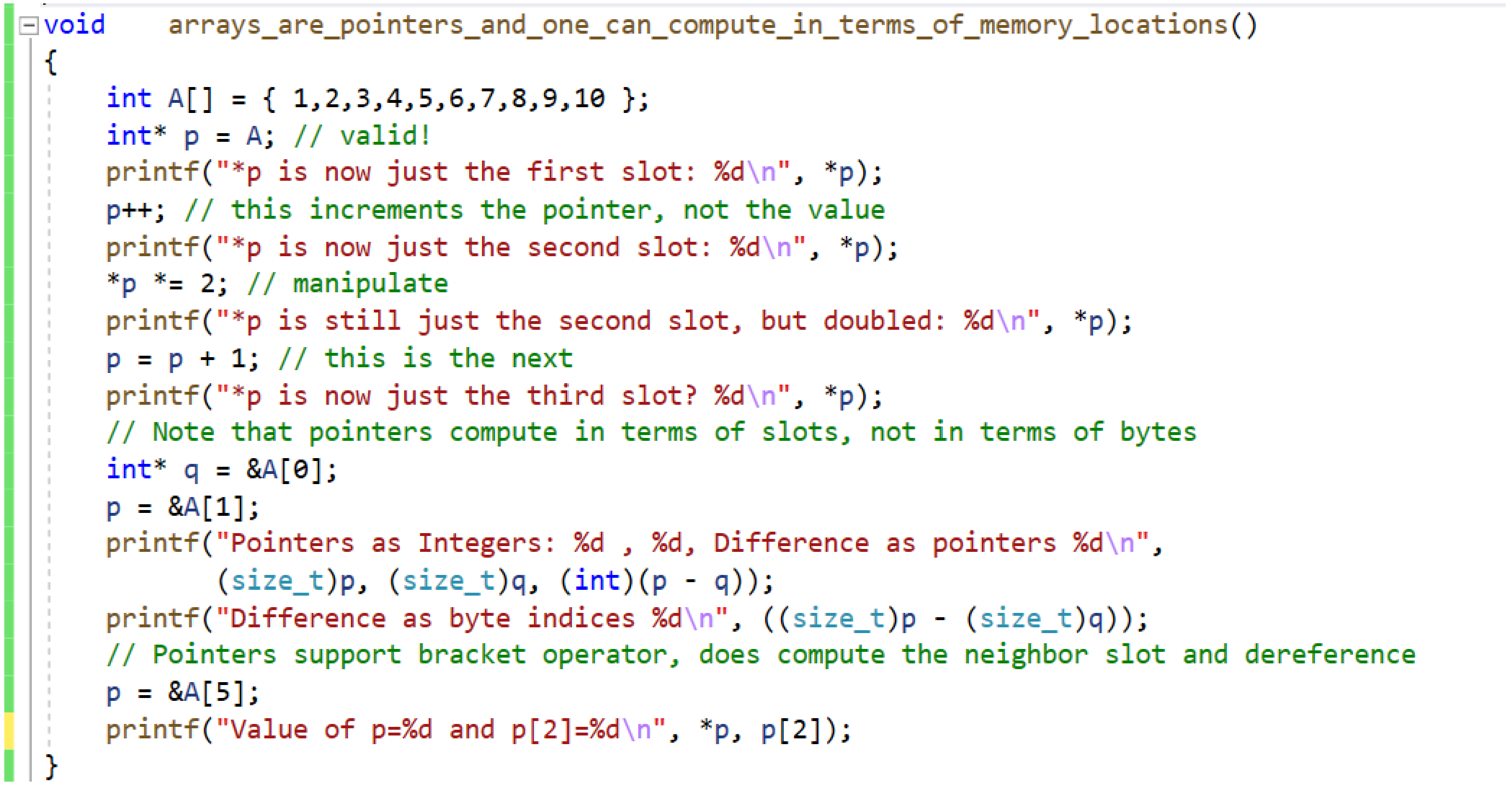
Arrays are Pointers

In practice, this means that arrays and pointers can be identified. In fact, the array variable itself is easily interpreted as a pointer to the beginning.

As pointers are numbers, one can compute with them (pointer arithmetics) to compute location of neighboring values.

In fact, the [-] operator is defined for pointers and what it does, is it **advances** the pointer by the number in bracket times the size of value type of the pointer and returns a variable which „references“ this location.

Pointer Arithmetic

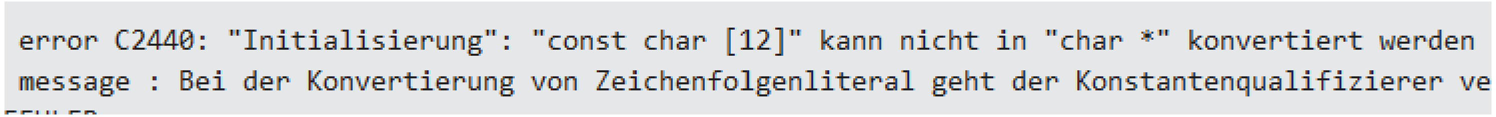


```
void    arrays_are_pointers_and_one_can_compute_in_terms_of_memory_locations()
{
    int A[] = { 1,2,3,4,5,6,7,8,9,10 };
    int* p = A; // valid!
    printf("*p is now just the first slot: %d\n", *p);
    p++; // this increments the pointer, not the value
    printf("*p is now just the second slot: %d\n", *p);
    *p *= 2; // manipulate
    printf("*p is still just the second slot, but doubled: %d\n", *p);
    p = p + 1; // this is the next
    printf("*p is now just the third slot? %d\n", *p);
    // Note that pointers compute in terms of slots, not in terms of bytes
    int* q = &A[0];
    p = &A[1];
    printf("Pointers as Integers: %d , %d, Difference as pointers %d\n",
           (size_t)p, (size_t)q, (int)(p - q));
    printf("Difference as byte indices %d\n", ((size_t)p - (size_t)q));
    // Pointers support bracket operator, does compute the neighbor slot and dereference
    p = &A[5];
    printf("Value of p=%d and p[2]=%d\n", *p, p[2]);
}
```

Chapter 3.4
Strings in C
(note that C++ strings are different and maybe better)

C strings: a playground for pointer arithmetics

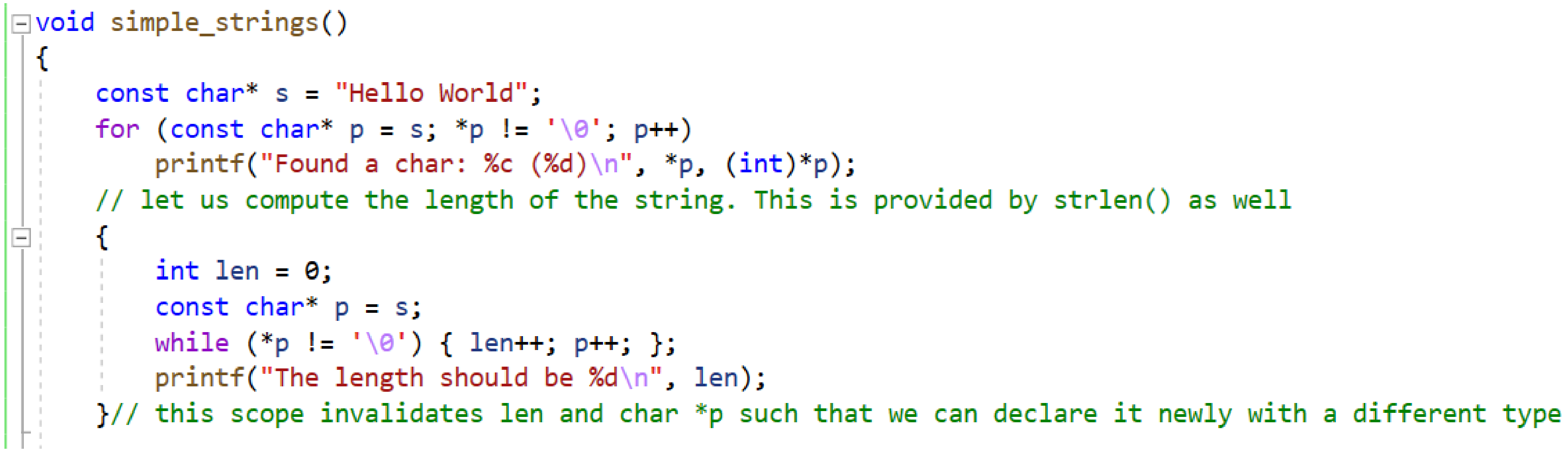
- In C, a character is represented as an unsigned char, which is an integer type ranging from 0..255.
- The meaning of the character follows the ASCII table.
- A string in C is now represented as a char * to the beginning of the string. The end of string is (by agreement) marked with a NULL character (\0)
- Traditionally, char* was used, but unsigned char * is more correct.
- Constants in the code must be const char * (can be some warnings or error messages when you start. The const just means that this must not be changed by code).



```
error C2440: "Initialisierung": "const char [12]" kann nicht in "char *" konvertiert werden
message : Bei der Konvertierung von Zeichenfolgenliteral geht der Konstantenqualifizierer ve
```

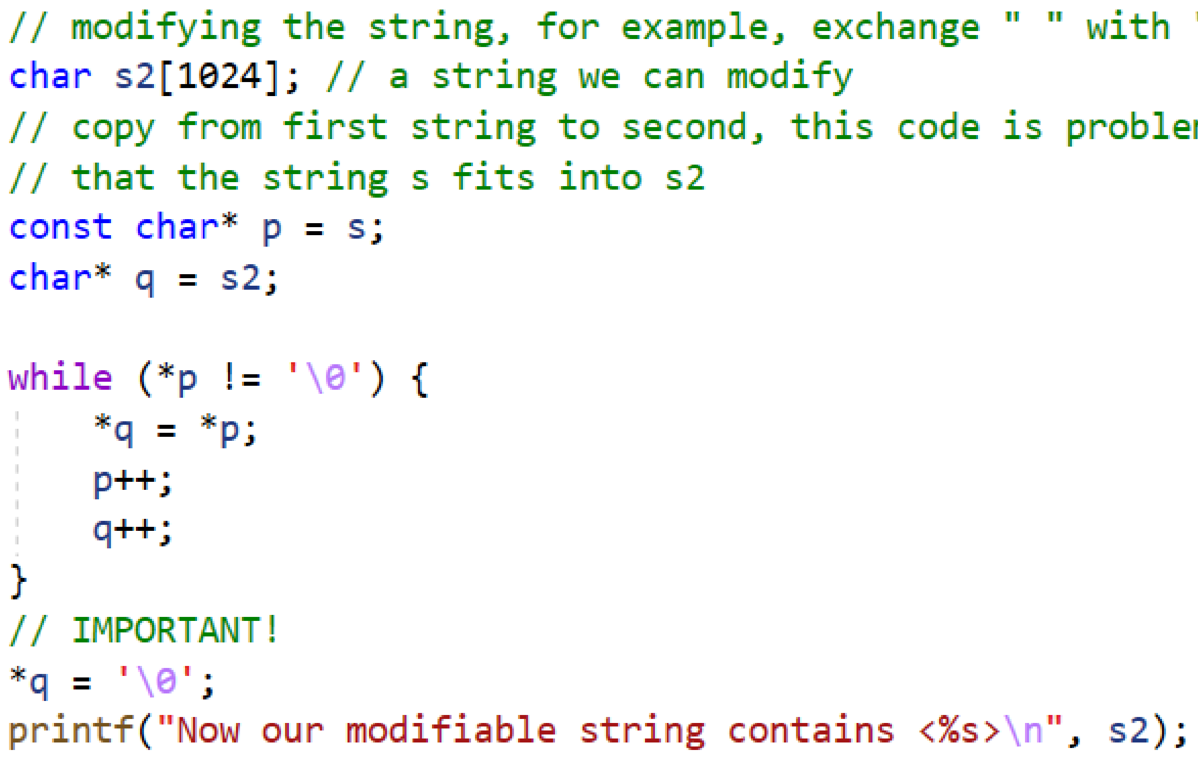
See the following slides for example code.

A simple string printing out characters and their codes



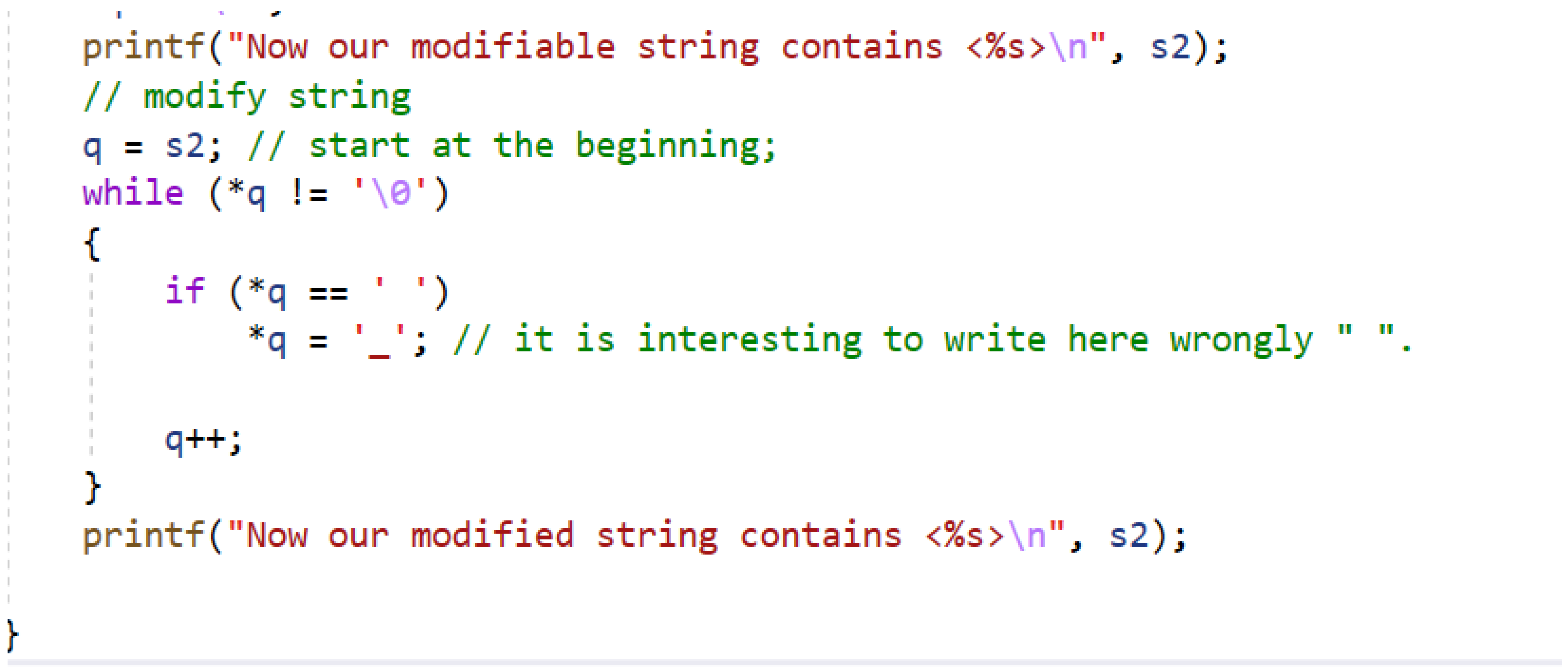
```
void simple_strings()
{
    const char* s = "Hello World";
    for (const char* p = s; *p != '\0'; p++)
        printf("Found a char: %c (%d)\n", *p, (int)*p);
    // let us compute the length of the string. This is provided by strlen() as well
    {
        int len = 0;
        const char* p = s;
        while (*p != '\0') { len++; p++; };
        printf("The length should be %d\n", len);
    }// this scope invalidates len and char *p such that we can declare it newly with a different type
```

Copying a string (see strcpy as well)



```c
// modifying the string, for example, exchange " " with "_"
char s2[1024]; // a string we can modify
// copy from first string to second, this code is problematic if we did not knew
// that the string s fits into s2
const char* p = s;
char* q = s2;

while (*p != '\0') {
    *q = *p;
    p++;
    q++;
}
// IMPORTANT!
*q = '\0';
printf("Now our modifiable string contains <%s>\n", s2);
```

Conditional Replacement of Characters (Example)



```
    printf("Now our modifiable string contains <%s>\n", s2);
    // modify string
    q = s2; // start at the beginning;
    while (*q != '\0')
    {
        if (*q == ' ')
            *q = '_'; // it is interesting to write here wrongly " ".

        q++;
    }
    printf("Now our modified string contains <%s>\n", s2);

}
```

The string library (string.h)

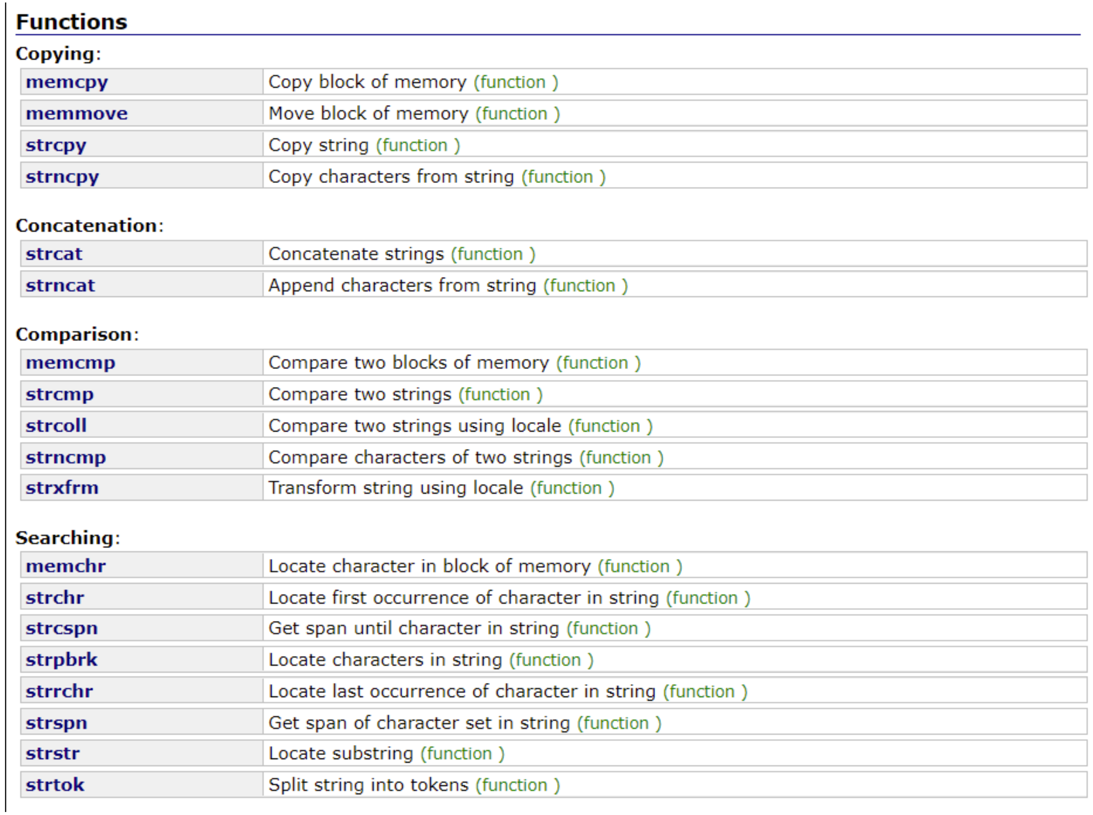
Basic functions for manipulating C strings are given in string.h

The problem here is that with „bad“ input strings, for example very long ones, one might end up with problems. If the strings come from untrusted sources (files, users, network, Internet, etc.), a family of functions has been proposed that solves these issues.

Therefore, some additional functions (with an n in the name) take the maximal length of the output and ensure zero-terminated strings.

https://www.cplusplus.com/reference/cstring/

String Library (as defined in C++ standards)



Functions

Copying:

memcpy	Copy block of memory (function)
memmove	Move block of memory (function)
strcpy	Copy string (function)
strncpy	Copy characters from string (function)

Concatenation:

strcat	Concatenate strings (function)
strncat	Append characters from string (function)

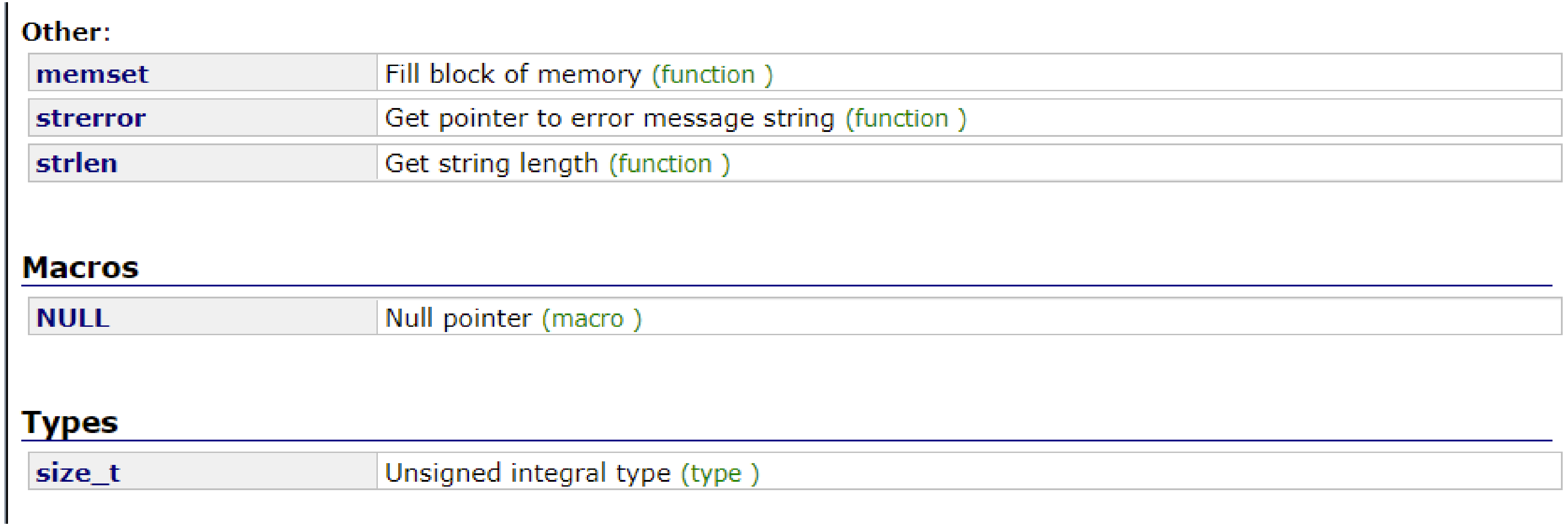
Comparison:

memcmp	Compare two blocks of memory (function)
strcmp	Compare two strings (function)
strcoll	Compare two strings using locale (function)
strncmp	Compare characters of two strings (function)
strxfrm	Transform string using locale (function)

Searching:

memchr	Locate character in block of memory (function)
strchr	Locate first occurrence of character in string (function)
strcspn	Get span until character in string (function)
strpbrk	Locate characters in string (function)
strrchr	Locate last occurrence of character in string (function)
strspn	Get span of character set in string (function)
strstr	Locate substring (function)
strtok	Split string into tokens (function)

String Library (as defined in C++ standards)



Other:

memset	Fill block of memory (function)
strerror	Get pointer to error message string (function)
strlen	Get string length (function)

Macros

NULL	Null pointer (macro)

Types

size_t	Unsigned integral type (type)

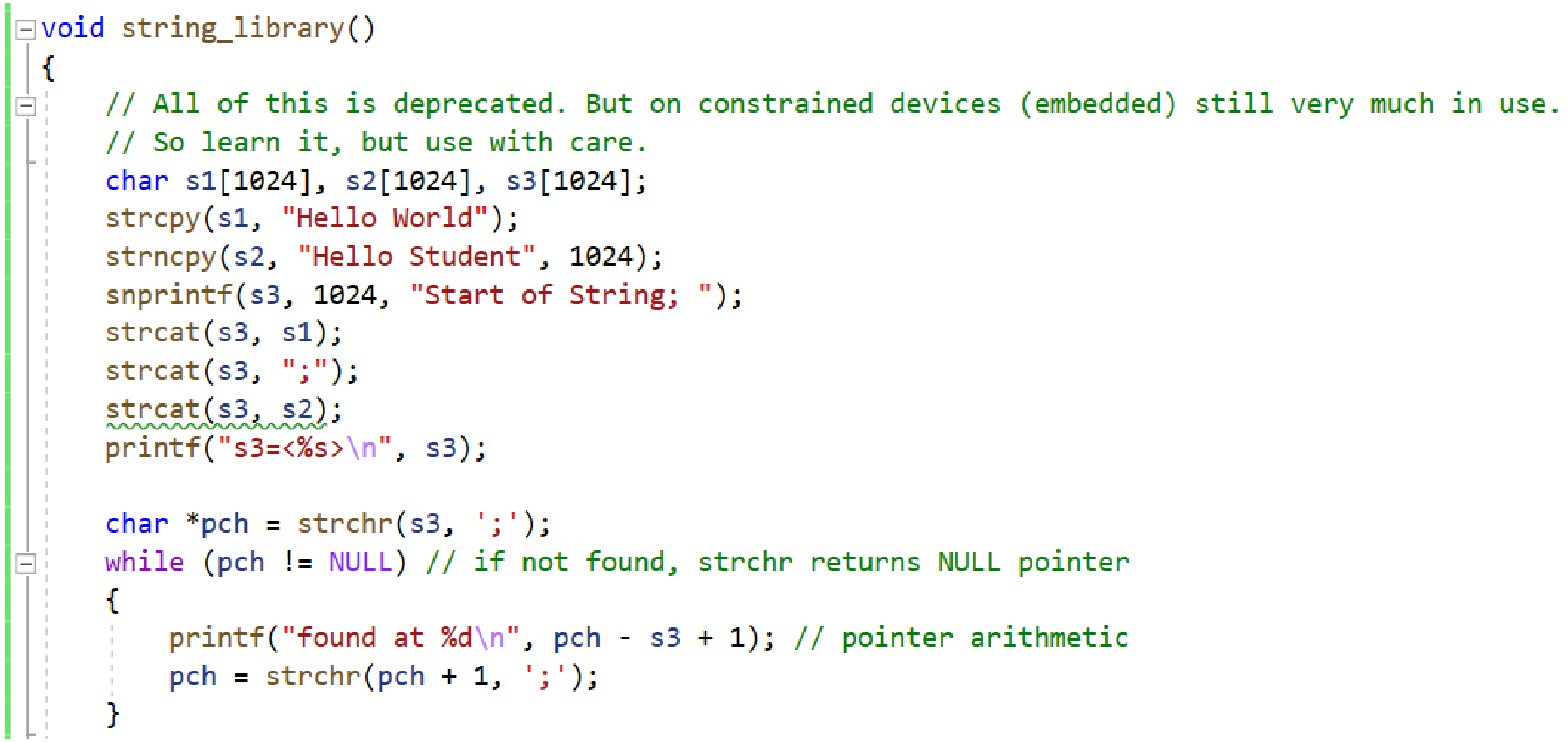
Most important

strcat	concatenate two strings.
strchr	string scanning operation.
strcmp	compare two strings.
strcpy	copy a string.
strncpy	safely copy a string
strlen	get string length.
strncat	concatenate one string with part of another.
strncmp	compare parts of two strings.

Additionally, from stdio.h the function snprintf is helpful:

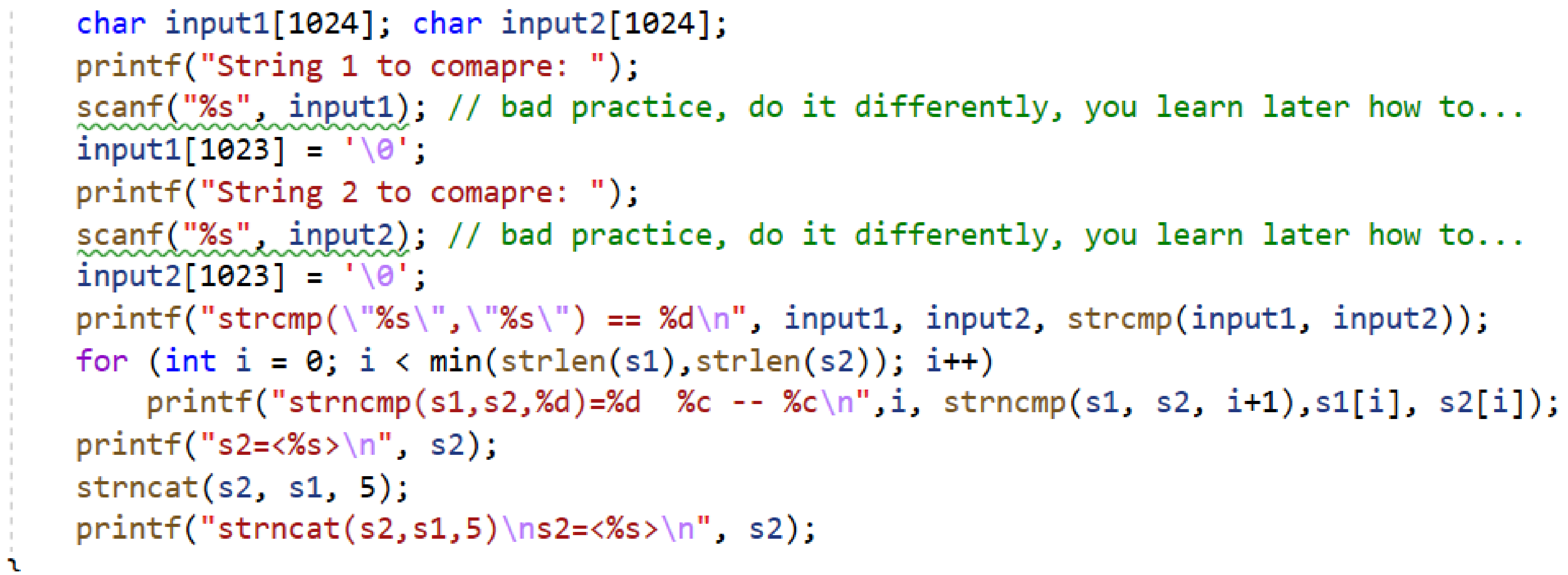
https://www.cplusplus.com/reference/cstdio/snprintf/?kw=snprintf

String Library (I)



```c
void string_library()
{
    // All of this is deprecated. But on constrained devices (embedded) still very much in use.
    // So learn it, but use with care.
    char s1[1024], s2[1024], s3[1024];
    strcpy(s1, "Hello World");
    strncpy(s2, "Hello Student", 1024);
    snprintf(s3, 1024, "Start of String; ");
    strcat(s3, s1);
    strcat(s3, ";");
    strcat(s3, s2);
    printf("s3=<%s>\n", s3);

    char *pch = strchr(s3, ';');
    while (pch != NULL) // if not found, strchr returns NULL pointer
    {
        printf("found at %d\n", pch - s3 + 1); // pointer arithmetic
        pch = strchr(pch + 1, ';');
    }
```

String Library (2)



```
char input1[1024]; char input2[1024];
printf("String 1 to comapre: ");
scanf("%s", input1); // bad practice, do it differently, you learn later how to...
input1[1023] = '\0';
printf("String 2 to comapre: ");
scanf("%s", input2); // bad practice, do it differently, you learn later how to...
input2[1023] = '\0';
printf("strcmp(\"%s\",\"%s\") == %d\n", input1, input2, strcmp(input1, input2));
for (int i = 0; i < min(strlen(s1),strlen(s2)); i++)
    printf("strncmp(s1,s2,%d)=%d  %c -- %c\n",i, strncmp(s1, s2, i+1),s1[i], s2[i]);
printf("s2=<%s>\n", s2);
strncat(s2, s1, 5);
printf("strncat(s2,s1,5)\ns2=<%s>\n", s2);
```

Chapter 3.5 References

References – Better Pointers?

Sometimes, pointer dereferencing (*-operator) and pointer arithmetics are annoying. Can we have a concept in which a variable name is assigned to the memory location of another variable, but without using pointers?

In C++, therefore, references are introduced.

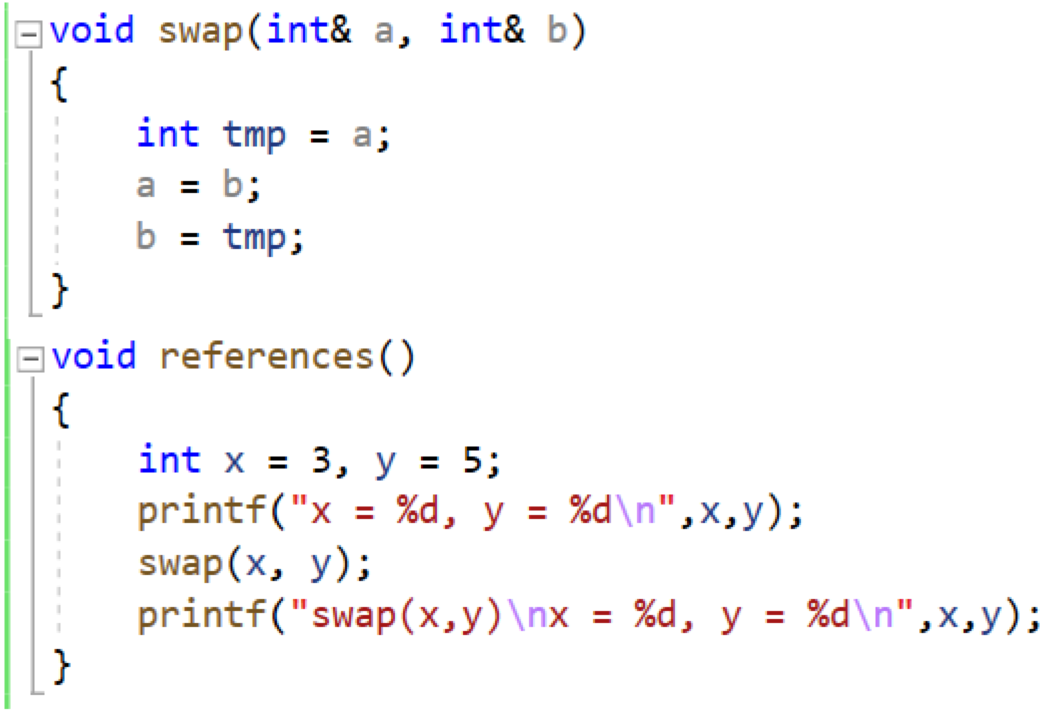
A reference behaves like the type itself, pointers are not involved.
A reference can be assigned the location it refers to only during construction (afterwards it behaves like the original variable)

It is not possible to create references to arbitrary values as they might not exist in memory at all times.

References are mainly used as parameters of functions, but can be useful otherwise as well.

Reference

A reference is declared using & in the declaration (not in an instruction, where it generates a pointer).
It makes the variable a placeholder for the input. For example (call by reference, the function does not get a copy of the parameters, it can change them)



```
void swap(int& a, int& b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
void references()
{
    int x = 3, y = 5;
    printf("x = %d, y = %d\n",x,y);
    swap(x, y);
    printf("swap(x,y)\nx = %d, y = %d\n",x,y);
}
```

Returning a reference

If you return a reference, the value is not copied to the stack, but refers to the original one.

This can be used in advanced patterns like method chaining and accessor functions.

In this case, the function call can stand on the left hand side of an assignment.

When to use references

References can be very helpful an have an extreme impact on code performance. Here is a basic guideline for how to declare the parameters of a function you are writing.

- If you want to change the values of the parameters, **call by reference**
- If you don‘t want to change the values of the parameters
  - Call **by value if they are small** (few bytes)
  - Call **by const reference otherwise** (avoids a costly copy operation of huge data)

Examples will follow in the tutorial.