(4) C++ Procedural Programming

Nico Ludwig (@ersatzteilchen)

TOC

- (4) C++ Procedural Programming
  - Functions – Overview
  - Recursion
  - Functional Programming
  - Namespaces and separated Function Definitions
  - A Glimpse of Separated Compilation and Translation Units
  - Documentation of Functions

- Sources:
  - Bruce Eckel, Thinking in C++ Vol I
  - Bjarne Stroustrup, The C++ Programming Language

Understanding Reusability

- <u>Repetition of code is really bad!</u>
  - We have to <u>copy</u> code to <u>reuse</u> it, or to <u>share</u> it with other developers.
  - If repeated code contains a bug, this bug is present in <u>all occurrences of that code!</u>
  - In general: <u>D</u>on't <u>R</u>epeat <u>Y</u>ourself! <u>DRY</u>

- We already discussed the usage of <u>loops to support the DRY idea</u>, e.g. to <u>repeatedly query numbers from the user</u>:

```
std::cout<<"Please enter a number:"<<std::endl;
std::cout<<"The number should be greater than ten:"<<std::endl;
int number1;
std::cin>>number1;
std::cout<<"You entered "+number1+"!"<<std::endl;

std::cout<<"Please enter a number:"<<std::endl;
std::cout<<"The number should be greater than ten:"<<std::endl;
int number2;
std::cin>>number2;
System.out.println("You entered "<<number2<<"!"<<std::endl;

std::cout<<"Please enter a number:"<<std::endl;
std::cout<<"The number should be greater than ten:"<<std::endl;
int number3;
std::cin>>number3;
std::cout<<"You entered "<<number3<<"!"<<std::endl;
```

```
// This for-loop executes the block below for three times.
for (int i = 1; i <= 3; ++i) {
    std::cout<<"Please enter a number:"<<std::endl;
    std::cout<<"The number should be greater than ten:"<<std::endl;
    int number;
    std::cin>>number;
    std::cout<<"You entered "<<number<<"!"<<std::endl;
}
```

- But what if <u>another programmer</u> wants to <u>use this code in his own program/cpp-file</u> and <u>own *main()*-function</u>?   3
  - In fact we implemented DRY aware code, but it is <u>not yet reusable</u>! This can be achieved by programming <u>own functions</u>!

```
std::cout<<"Please enter a number:"<<std::endl;
std::cout<<"The number should be greater than ten:"<<std::endl;
int number;
std::cin>>number;
std::cout<<"You entered "<<number<<"!"<<std::endl;
```

- But, what if programmer B wants to use this code in his own program/cpp-file "programB" and own *main()*-function?
  - The code needs to be transported from our program "programA" to his "programB".
  - The problem of this code: it is not reusable! – Therefor we need to copy the code from "programA" to "programB":

```
// <programA.cpp>
#include <iostream>

int main() {
    std::cout<<"Please enter a number:"<<std::endl;
    std::cout<<"The number should be greater than ten:"<<std::endl;
    int number;
    std::cin>>number;
    std::cout<<"You entered "<<number<<"!"<<std::endl;
}
```

```
// <programB.cpp>
#include <iostream>

int main() {
    std::cout<<"Please enter a number:"<<std::endl;
    std::cout<<"The number should be greater than ten:"<<std::endl;
    int number;
    std::cin>>number;
    std::cout<<"You entered "<<number<<"!"<<std::endl;
```

copied code

- But we just heard, that repetition of code is really bad:
  - We have to copy code to reuse it, or to share it with other developers. We call this "copy-pase reuse".
  - If repeated code contains a bug, this bug is present in all occurrences of that code!
  - In general: Don't Repeat Yourself! DRY

4

- When a piece of code, which is repeated all over in the code needs a bugfix or other modifications (e.g. so called refactoring) this is sometimes called shotgun refactoring. The term underscores the fact, that pieces of code spread all over the program must be modified, an effect like firing a shotgut, which spreads its pellets.

Making Code reusable with Functions – Part I

- We will gradually enhance the reusability of this code during the following slides.
  - We begin with putting the code prompting for user input into a reusable unit:

```
// This is the code we want to reuse:
std::cout<<"Please enter a number:"<<std::endl;
std::cout<<"The number should be greater than ten:"<<std::endl;
```

```
// <programA.cpp>
#include <iostream>

void printPrompt() {
        std::cout<<"Please enter a number:"<<std::endl;
        std::cout<<"The number should be greater than ten:"<<std::endl;
}
```

- We put the code into a so called function with the name *printPrompt()*.
  - In the function *main()* we now call the function *printPrompt()*, which executes the code contained in that function.
  - Functions have names for verbs, activities or actions.

Good to know
Function, from latin *functio*, meaning "task"

```
// <programA.cpp>
#include <iostream>

void printPrompt() {
        std::cout<<"Please enter a number:"<<std::endl;
        std::cout<<"The number should be greater than ten:"<<std::endl;
}

int main() {
        for (int i = 1; i <= 3; ++i) {
                printPrompt();
                int number;
                std::cin>>number;
                std::cout<<"You entered "<<number<<"!"<<std::endl;
        }
}
```

5

Making Code reusable with Functions – Part II

- Actually *printPrompt()* is called for three times, i.e. for each "round" in the for loop.
  - So *printPrompt()*'s code is reused for three times.

- So in retrospective, these two programs are functionally equivalent:

```cpp
// <programA.cpp>
#include <iostream>

void printPrompt() {
    std::cout<<"Please enter a number:"<<std::endl;
    std::cout<<"The number should be greater than ten:"<<std::endl;
}

int main() {
    for (int i = 1; i <= 3; ++i) {
        printPrompt();
        int number;
        std::cin>>number;
        std::cout<<"You entered "<<number<<"!"<<std::endl;
    }
}
```

```cpp
// <programA.cpp>
#include <iostream>

int main() {
    for (int i = 1; i <= 3; ++i) {
        std::cout<<"Please enter a number:"<<std::endl;
        std::cout<<"The number should be greater than ten:"<<std::endl;
        int number;
        std::cin>>number;
        std::cout<<"You entered "<<number<<"!"<<std::endl;
    }
}
```
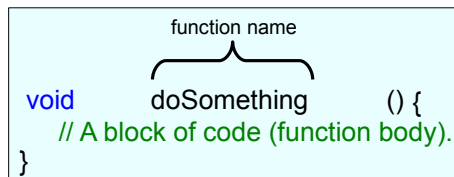
- A function is a bunch of statements, which can be called using a name. – This makes the reusability!
  - *printPrompt()* represents a bunch of statements printing the prompt to the console.

6

- In fact programming functions is the central activity of C/C++ programmers!

Anatomy of Functions

- Let's discuss the definition of the function *doSomething()*:

```
                    function name
                   ⌣⎵⌣

void       doSomething       () {
        // A block of code (function body).
}
```

- Names of functions, e.g. *doSomething()* or *printPrompt()* obey to the <u>same rules as all identifies</u> in C++ do.
    - Function names are <u>case sensitive</u> and are <u>not allowed to begin with a digit etc.</u>
        - <u>Function names are identifiers like variables, thus they obey the same rules.</u>
    - A function with a specific name and signature <u>must only occur once in the same namespace</u>. – We'll discuss this aspect soon.
    - The <u>parentheses do not belong to the function names</u>, they are written to distinguish function names from, e.g., variable names.
    - <u>Function names do not follow a standardized convention in C++</u>, in this course, we'll use the <u>camelCase</u> naming convention.
    - Function names usually read like <u>verbs</u> or <u>activities</u> like *doSomething()* or *printPrompt()*.
        - In opposite to variables, which often read like things, values or containers.
        - The name *main()* is somewhat different from this "activity-naming" scheme.

7

Anatomy of Function Definitions

- Hence, we'll occasionally use complete function definitions in the code snippets.

- If the code contained in a function is not of interest, or unknown to us, we just write the comment // pass as a placeholder:

```
void        doSomething        () {
    // pass
}
```

  - This is just a convention, we follow for the examples in this course.

- Braces (i.e. a block) are mandatory to surround function bodies, even if a function body has not any statements!

- For the way we program procedurally in C++ right now, the keyword void is required. Let's just accept this for now.

8

Calling Functions – Part I

```cpp
void printPrompt() {
    std::cout<<"Please enter a number:"<<std::endl;
    std::cout<<"The number should be greater than ten:"<<std::endl;
}

int main() {
    for (int i = 1; i <= 3; ++i) {
        printPrompt();
        int number;
        std::cin>>number;
        std::cout<<"You entered "<<number<<"!"<<std::endl;
    }
}
```

`doSomething () ;`
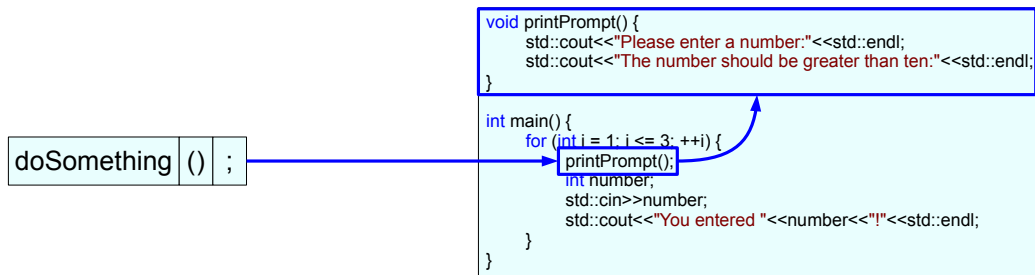
- <u>To make use of the code contained in a function</u>, we have to <u>call</u> the function, <u>then its code will be executed</u>.
  - The <u>call itself is just a statement</u> and <u>it can also be used as expression</u>, which we'll discuss later.
  - As can be seen, we can virtually <u>write any code in *printPrompt()*</u> we would have written in *main()*.
    - E.g. we can use *std::cout* either in *main()* or *printPrompt()*.

- We call "*printPrompt()* in *main()*" or "from *main()*" -> <u>*main()* is the caller</u> and <u>*printPrompt()* is the callee</u>.

- <u>The call expression is put into effect by just writing the name of the function we want to call and a pair of empty parens.</u>

9

Calling Functions – Part II

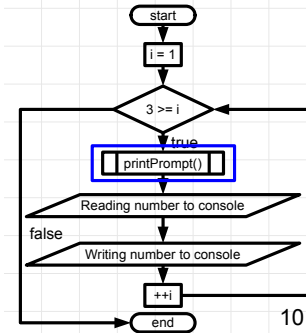- With the introduction of functions, we leave the ground of flowchart diagrams and NSDs.
    - The idea of those diagram types is to show the imperative code in a function.
    - In other words: a function encapsulates imperative code.

- However, flowchart diagrams and NSDs have symbols to represent function calls from within imperative code:
    - In both diagram types function calls, formally procedure calls, are represented by boxes having left and right borders.

```
void printPrompt() {
    std::cout<<"Please enter a number:"<<std::endl;
    std::cout<<"The number should be greater than ten:"<<std::endl;
}

int main() {
    for (int i = 1; i <= 3; ++i) {
        printPrompt();
        int number;
        std::cin>>number;
        std::cout<<"You entered "<<number<<"!"<<std::endl;
    }
}
```

NSD (function call)

for i = 1 to 3 do
printPrompt()
Reading number from console
Writing number to console
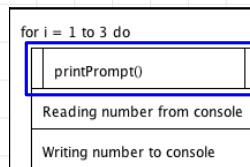
Flowchart Diagram (function call)

10

The Theory behind Procedural Programming

- In <u>engineering</u>, <u>complex problems are getting separated into smaller problems</u>.
  - In <u>programming</u>, <u>complex code is broken into subroutines, subprograms, procedures or functions</u>.
  - Then we have a <u>piece of code</u>, which can be used to <u>solve a "category" of problems</u>.
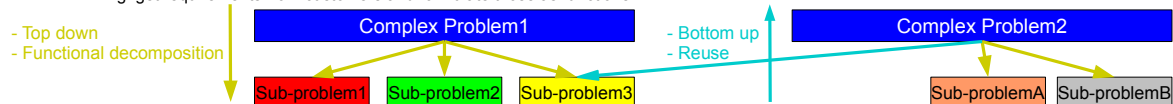 - We call this the <u>"Top-Down" approach</u>, or <u>functional decomposition</u>.

- Procedural programming is all about the ideas of "Top-Down" and "Bottom-Up".
  - Bottom-Up: Programmers <u>code new functions that solve sub-problems</u>.
    - These functions can then be reused, i.e. <u>called</u>, by other complex code to solve its sub-problems of the same category.
  - Top-Down: Programmers <u>decompose their code into functions to make sub problems solvable and make code reusable</u>.
    - E.g. get requirements from customers and formulate those as functions.

Definition
Procedural programming means to program considering a problem as being expressed by reusable sub-problems, so called procedures.

Good to know:
Procedure from latin *procedere* – to go forward.

- Top down
- Functional decomposition

Complex Problem1

- Bottom up
- Reuse

Complex Problem2

Sub-problem1 Sub-problem2 Sub-problem3

Sub-problemA Sub-problemB

- Procedural programming extends imperative programming with the concept of (reusable) functions.
  - <u>Reusability of code from the perspective of procedural programming is the ability to call functions from other functions.</u>
  - <u>In functions, statements can be executed imperatively.</u>
  - <u>And in functions, further functions can be called as well.</u>

- The C/C++ standard library provides a plethora of functions for us.
    - The C/C++ standards team already had their "Top-Down" specs on... :-)
    - These functions allow reusing foreign code to solve own problems.
        - E.g: *std::pow()*, *std::qsort()*, *std::bsearch()*, *std::transform()* etc.
    - Some functions open the gate to the OS, like *std::system()*.

- How to use functions from the standard library?
    - (1) #include the respective standard h-file, where the function declaration resides.
    - (2) Call the standard function within own code.
    - (3) Mind to pass the o-file/library file where the functions' definition resides, to the linker.

- Let's see how this works in practice...

12

- <u>"There is a problem with the usage of *std::system()*."
  What does this statement mean?</u>
    - Well, *std::system()* accepts commands that are highly platform-dependent in most cases.

Using predefined Functions in own Code – Part II

- Let's assume, we want to write a program, which <u>squares the input base</u>:
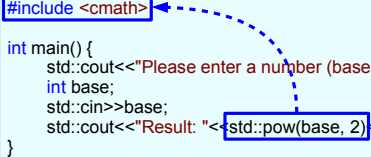
```cpp
// <main.cpp>
#include <iostream>

int main() {
    std::cout<<"Please enter a number (base) to be squared:"<<std::endl;
    int base;
    std::cin>>base;
    std::cout<<"Result: "<<(base * base)<<std::endl;
}
```

  - Here, we have written the code squaring the input on our own, namely just multiplying the *base* with itself.

- The <u>C++ standard library</u> provides the function *std::pow()* in the h-file <cmath>, which we can use instead of the multiplication:

```cpp
// <main.cpp>
#include <iostream>
#include <cmath>

int main() {
    std::cout<<"Please enter a number (base) to be squared:"<<std::endl;
    int base;
    std::cin>>base;
    std::cout<<"Result: "<<std::pow(base, 2)<<std::endl;
}
```

  - We call *std::pow()* and pass *base* and the exponent (2 for squaring) as arguments. (We'll discuss arguments soon.)

  - We have to <u>#include <cmath></u>, because this <u>h-file</u> contains <u>information about *std::pow()*</u>, <u>so that it can be called</u>.

13

Using predefined Functions in own Code – Part III

- The h-file <cmath> contains the declaration of the function *std::pow()*:

```
// <cmath> _very_ simplified
namespace std {
    double pow(double base, double exponent);
    // a lot more declarations exist in <cmath>
}
```

- A function declaration makes the name of a function known to the compiler.
  - A function declaration does just have no function-body, but the declaration is terminated with a semicolon.
  - Optionally, function declarations can be marked with the extern keyword.

- Fair enough, we didn't clarify, where the definition, i.e. the body of *std::pow()* resides. This explanation is yet to come.

- Furthermore, <cmath> encloses *pow()*'s declaration into another block, a so-called namespace with the name *std*.
  - Hence the function is named *std::pow()*, i.e. *pow() in the namespace std*. The name of the namespace separates the name with ::.
  - We'll discuss namespaces in short.

- In fact also using others' functions is a central activity of C/C++ programmers!
  - The idea is to combine other's functions/libraries and my own functions/libraries, to solve sub-problems in my program.

14

Declaration and Definition of Functions

- After we have seen *std::pow()*, it is time to show a more formal picture of a function's declaration and definition.

Declaration:

extern	int	myFunction	(int x, int y = 0);

Type of returned value or void.   Function name   Parameter list   Default argument

Definition:

int	myFunction	(int x, int y)

```
{ // A block of code (function body).

    return 42;
}
```

Return value from function.   Function body

int (*)(int, int)

The signature of *myFunction()*.

C++11 – trailing return type
```
auto myFunction(int x, int y) -> int {
    return 42;
}
```

15

The local Variable Scope – Part I

- Let's continue enhancing the reusability of our program just by <u>increasing the amount of decomposed code</u>.
  - In the next step, we'll put reading the user's input into the function *printPrompt()* along with the prompt itself:

```cpp
void printPrompt() {
    std::cout<<"Please enter a number:"<<std::endl;
    std::cout<<"The number should be greater than ten."<<std::endl;
    int number;
    std::cin>>number;
}

int main() {
    for (int i = 1; i <= 3; ++i) {
        printPrompt();
        std::cout<<"You entered "<<number<<"!"<<std::endl;
    }
}
```

- But this code <u>will not compile</u>!
  - The compiler has a problem here, because in *main()* we refer to a variable *number*, which is <u>not visible in *main()*</u>!
    - The compiler message reads "Use of undeclared identifier 'number'".
  - *number* is defined in *printPrompt()* and <u>only visible in the scope of *printPrompt()*</u>.

- We say, the variable *number* <u>is a local variable (in short "a local") in *printPrompt()*</u>.

16

The local Variable Scope – Part II

- Up to now, we have only discussed <u>local variables</u>, which are <u>only known to the function, in which they are defined</u>.
  - We can also say, that a variable defined in a function <u>is only visible in the local scope (of that function)</u>.
  - Remember, that <u>a scope defines when/where a variable is known to its environment</u> and to which environment is it known.

- <u>Local variables ("locals") are only known in the function, in which they were defined!</u>

```cpp
void functionA() {
    int numberA = 41;
    std::cout<<numberA<<std::endl;
    // >41
}

void functionB() {
    functionA();
    int numberB = 234;
    std::cout<<numberB<<std::endl;
    // >234
    std::cout<<numberA<<std::endl; // Invalid! numberA not visible here!
}
```

  - <u>Calling *functionA()* in *functionB()* will not make its locals (i.e. *numberA*) visible to *functionB()*!</u>
  - Also the written <u>order</u> of function definitions or variable definitions or types <u>is irrelevant</u> for the variables' scope or "locality".

17

The Idea of global Variables – Part I

- We can solve the "scoping problem" of the local *number* by making it a <u>global variable</u>, in short <u>"a global"</u>.

- It means, that we simply put *number* into <u>a scope, which is common to *main()* and *printPrompt()*</u>.

```cpp
void printPrompt() {
    std::cout<<"Please enter a number:"<<std::endl;
    std::cout<<"The number should be greater than ten."<<std::endl;
    int number;
    std::cin>>number;
}

int main() {
    for (int i = 1; i <= 3; ++i) {
        printPrompt();
        std::cout<<"You entered "<<number<<"!"<<std::endl;
    }
}
```

(1) →

```cpp
int number; // global variable number

void printPrompt() {
    std::cout<<"Please enter a number:"<<std::endl;
    std::cout<<"The number should be greater than ten."<<std::endl;
    std::cin>>number;
}

int main() {
    for (int i = 1; i <= 3; ++i) {
        printPrompt();
        std::cout<<"You entered "<<number<<"!"<<std::endl;
    }
}
```

(2)

- Now, *number* is visible to *main()* and *printPrompt()*, because <u>*number*, *main()* and *printPrompt()* reside in the same scope</u>.
    - (1) <u>*number* is written in *printPrompt()*</u>, i.e. <u>the user's input is stored in *number*</u>.
        - <u>We say *printPrompt()* has a side-effect on *number*</u>.
    - (2) And <u>*number* is read in *main()*</u>, where <u>*number* is printed to the console</u>.

Good to know
In C++, globals are automatically 0-initialized.

- <u>More exactly, *number*, *main()* and *printPrompt()* reside in the global scope now</u>.

The Idea of global Variables – Part II

- Let's temporarily add another function *printNumber()*, which prints the content of the global *number* to the console:

  - *printPrompt()* prompts the user for input and stores the input in the global *number*.

  - *printNumber()* prints the value of the global *number* to the console.

  - *main()* only needs to call *printPrompt()* and then *printNumbers()*.

  - The communication between *printPrompt()* and *printNumbers()* is handled via the global *number*.

```
int number; // global variable number

void printPrompt() {
    std::cout<<"Please enter a number:"<<std::endl;
    std::cout<<"The number should be greater than ten."<<std::endl;
    std::cin>>number;
}

void printNumber() {
    std::cout<<"You entered "<<number<<"!"<<std::endl;
}

int main() {
    printPrompt();
    number = 7865; // deliberately modify number
    printNumber();
}
```

```
Terminal
NicosMBP:src nico$ ./main
Please enter a number:
The number should be greater than ten:
23
You entered 7865!
NicosMBP:src nico$
```

- Communication via globals can be very dangerous!

  - All functions can access the global scope and they have access and can modify all variables in the global scope!

  - I.e. *main()* can directly access and modify *number*, in between the calls of *printPrompt()* and *printNumber()*.

  - Here, the code deliberately modifies *number* in between *printPrompt()* and *printNumber()* overwriting the user's input!

  - *printPrompt()* has a side effect on *number* and *main()* has a side effect on *number*, i.e. both functions change the *number*'s value.

  - What we see is, that code of functions could overwrite the value of globals.

  - In practice, this way of programming will often evolve to unmanageable code with unpredictable side effects and disasters!

19

Functions returning a Value – Part I

- One can say, that <u>globals are a way for functions to communicate with its environment</u>, e.g. its callers.
  - Using globals means to use <u>side effects</u> for <u>communication</u>.

- But the <u>golden path to create manageable procedural code</u> is to <u>reduce side effects on the global scope</u>.
  - We just hinted how <u>unstructured access and modification</u> of *number* can lead to problems.

- Another core feature of functions beyond code reuse is the idea of functions <u>providing a result</u>, by <u>returning data</u>.
  - Therefor we're going to discuss functions, which <u>return</u> values, instead of modifying global values.

- Now we'll continue enhancing the reusability of *printPrompt()* by making it a function <u>returning a value</u>:

```cpp
int printPrompt() {
    std::cout<<"Please enter a number:"<<std::endl;
    std::cout<<"The number should be greater than ten."<<std::endl;
    int number; // local variable number
    std::cin>>number;
    return number;
}

int main() {
    int number = printPrompt();
    std::cout<<"You entered "<<number<<"!"<<std::endl;
}
```

20

Functions returning a Value – Part II

- The updated function *printPrompt()* shows two significant modifications:
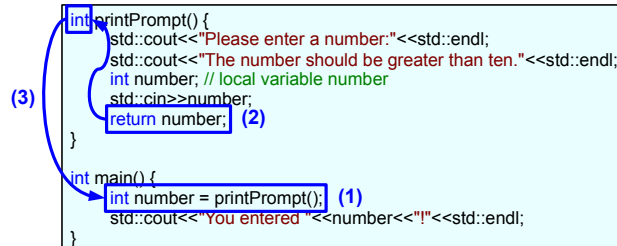
```cpp
int printPrompt() {
    std::cout<<"Please enter a number:"<<std::endl;
    std::cout<<"The number should be greater than ten."<<std::endl;
    int number; // local variable number
    std::cin>>number;
    return number;
}
// ...
```

- (1) The keyword void was replaced by the keyword int. So, <u>the function *printPrompt()* itself has a type now</u>!

- (2) There is a new statement in the function body, a <u>return statement</u>.

- *printPrompt() has no longer a side effect on a global*, instead it <u>return</u>s a value as a result of its call.

- The return statement is one way to get rid off side effects, the <u>side channel of the global has vanished</u>!
  - Now, *number* is a local variable in *printPrompt()*.

- Let's have a closer look at the call of *printPrompt()* to understand what is going on here.

Functions returning a Value – Part III

```cpp
int main() {
    int number = printPrompt();
    std::cout<<"You entered "<<number<<"!"<<std::endl;
}
```

- When we inspect the call of *printPrompt()*, we'll notice, that its call returns a value, which can be assigned to a variable!
    - In fact, the function call of *printPrompt()* is an expression, which yields a value.

```cpp
int printPrompt() {
    std::cout<<"Please enter a number:"<<std::endl;
    std::cout<<"The number should be greater than ten."<<std::endl;
    int number; // local variable number
    std::cin>>number;
    return number;        (2)
}
int main() {
    int number = printPrompt();        (1)
    std::cout<<"You entered "<<number<<"!"<<std::endl;
}
```
(3)

- The type of the function defines the type of the value, which is returned by the function.
    - We call it the return type of the function.

Functions returning a Value – Part IV

- The idea of function with return type changes the general anatomy of a function somewhat.
  - Functions can return a value and show this by <u>carrying the type in front of the function name</u>:

```
<Type>        identifier        () {
    // A block of code (function body).
    return valueOfType;
}
```

```
int        doSomething        () {
    // A block of code (function body).
    return intValue;
}
```
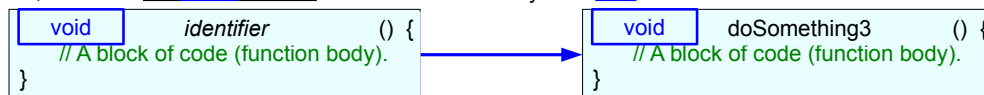
```
double        doSomething2        () {
    // A block of code (function body).
    return doubleValue;
}
```
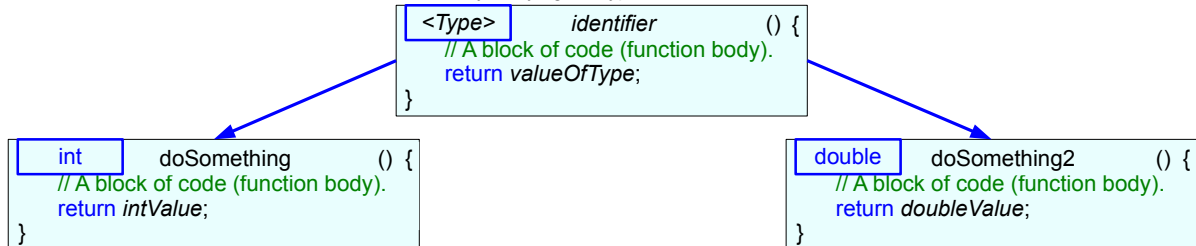
  - In the <u>function body</u> there is a <u>return</u> statement, <u>which return</u>s the result from the function to the caller and <u>ends function execution</u>.
  - <u>Any fundamental or compound type can be used as return</u> type.

- Functions, which do <u>not return a value</u> show this with the keyword <u>void</u> in front of the function name instead of a type:

```
void        identifier        () {
    // A block of code (function body).
}
```

```
void        doSomething3        () {
    // A block of code (function body).
}
```

23

  - void functions don't have a result, so they might have a side effect to communicate to its environment.

How the return Statement works – Part I

- When control flow within a function <u>reaches a return statement</u>, <u>the function will return immediately to the caller function</u>.

- <u>This means, that when a return statement is hit, no code beyond that return statement will be executed.</u>
  - There are exceptions to this rule, following the idea of the so called "RAII principle", which we will discuss in a future lecture.

```cpp
int giveMeAnInt() {
    int number;
    std::cin>>number;
    if (0 <= number) {
        std::cout<<"The input number was greater than or equal to 0"<<std::endl;
        return number;
    }
    std::cout<<"The input number was less than 0"<<std::endl;
    return number;
}

int main() {
    std::cout<<"Please enter a number"<<std::endl;
    std::cout<<"Entered number: "<<giveMeAnInt()<<std::endl;
}
```

```
Terminal
NicosMBP:Debug nico$ ./main
Please enter a number
Entered number: 5
The input number was greater than or equal to 0
5
NicosMBP:Debug nico$
```

- The int 5 was entered by the user, which is greater than 0, in this case:
  - (1) Only "The input number was greater than or equal to 0" was printed to the console.
  - (2) The following return statement was executed, it exited from *giveMeAnInt()* and transported the int 5 to the caller *main()*.

24

How the return Statement works – Part II

- When a function is called, a value of type "return type" will be returned as a result of that function.
  - The returned value is then <u>accepted</u> by the caller, e.g. stored (stuck into a variable) or otherwise consumed.
  - <u>Accepting a returned value on the caller side is basically the same like initializing/assigning a variable:</u>

```cpp
int giveMeAnInt() {
                              returning a value ≈
                              implicit assignment
        return 56;
}

int main() {
        int acceptedInteger = giveMeAnInt();              // store returned value in a variable
        std::cout<<"Number: "<<giveMeAnInt()<<std::endl;  // consume the returned value in an expression
}
```

- C++ allows <u>implicit conversions while returning values from functions</u> as we know it from initialization/assignment:

```cpp
int giveMeADouble() {
        return 56; // This int will be converted to a double value.
}

int main() {
        std::cout<<"Number: "<<giveMeADouble()<<std::endl;
}
```

How the return Statement works – Part III – Edge Cases

- Basically, this example shows, that functions can have more than one return statement:

```cpp
int giveMeAnInt() {
    int number;
    std::cin>>number;
    if (0 <= number) {
        std::cout<<"The input number was greater than or equal to 0"<<std::endl;
        return number;
    }
    std::cout<<"The input number was less than 0"<<std::endl;
    return number;
}
```

- C++ also allows to have multiple return statements in a sequence, i.e. unconditionally!
    - This is remarkable, because all unconditional statements following a return statement are unreachable:

```cpp
int suspiciousFunction() {
    int number;
    std::cin>>number;
    std::cout<<"Before"<<std::endl;
    return number;
    std::cout<<"After"<<std::endl;    // Unreachable code
    return number;                    // Unreachable code
}
```

    - Usually, C++ compilers accept this code, but will log a compiler warning like "code will never be executed".
    - Getting a bug warning here at least is good, because this is a serious source of bugs!

26

How the return Statement works – Part IV – Edge Cases

- A function, that declares a return type, must return a value on all of its control flow paths, e.g. consider:

```cpp
int printSomething(bool b) {
    if (b) {
        std::cout<<"printing something"<<std::endl;
        return 42;
    }
} // Invalid: Control may reach end of non-void function
```

  - In this case the compiler can clearly tell us, that not all paths return a value, here the function only returns if *b* evaluates to true.
  - However, the C++ compiler cannot always find such errors, i.e. a non-void function could really be executed not returning a value.
    - In such a case, the result of the function on the caller side is undefined!

- An exception to this restriction is *main()*. *main()* declares an int return type, but is not required to return a value!

```cpp
int main() {
    std::cout<<"Hello World!"<<std::endl;
}
```

  - If *main()* does not explicitly return a value, it'll implicitly return 0.

- Although it is allowed not to return a value from *main()*, it is not a good practice.
  - The return value of *main()*, the so called exit code, is sent to the executor of the program, e.g. to the terminal.

27

How the return Statement works – Part V – main() and Exit Codes

- The exit code of a program tells the executor, in which <u>state</u> the program <u>terminated</u> as an <u>integer code</u>.
  - Just to check this out, let's return *main()* with the code 42:

```cpp
int main() {
    std::cout<<"Hello World!"<<std::endl;
    return 42;
}
```

  - The <u>exit code is not automatically written to the console</u>! Instead we must <u>read it ourselves from the memory of the console</u>.
  - On a unixoide terminal, e.g. bash, the exit code of the last command is stored in the environment variable *$?*:

```
Terminal
NicosMBP:Debug nico$ ./main
Hello World!
NicosMBP:Debug nico$  echo $?
42
```

- If a program can exit with different exit codes, <u>those should be documented</u> (e.g. in its help output or man page).
  - But in most cases, it is sufficient to tell <u>successful termination</u> from <u>termination due to failure</u>.
  - The convention is to assume the <u>exit code 0 for successful termination</u> and <u>all other codes for termination due to failure</u>.
  - C++ defines two handy <u>constants</u> to denote successful or failure-termination: *EXIT_SUCCESS* and *EXIT_FAILURE* in <cstdlib>.

```cpp
#include <cstdlib>

int main() {
    std::cout<<"Hello World!"<<std::endl;
    return EXIT_SUCCESS;
}
```

28

How the return Statement works – Part VI – void

- The return statement can also be applied in functions w/o return type, i.e. in void functions!
  - As we already know, when control flow <u>reaches the return statement</u>, <u>it will return immediately to the caller and exit the function</u>.
  - This means, that if a return statement is hit, no code beyond that return statement will be executed in that function.

```
void enterAnInt() {
    int number;
    std::cin>>number;
    if (0 <= number) {
        std::cout<<"The input number was greater than or equal to 0"<<std::endl;
        return;
    }
    std::cout<<"The input number was less than 0"<<std::endl;
    return;
}

int main() {
    std::cout<<"Please enter a number"<<std::endl;
    enterAnInt();
}
```

```
Terminal
NicosMBP:src nico$ ./main
Please enter a number
5
The input number was greater than or equal to 0
NicosMBP:src nico$
```

- The int 5 was entered by the user, which is greater than 0, in this case
  - (1) only "The input number was greater than or equal to 0" was printed to the console, and
  - (2) the following return statement was executed, it exited from *enterAnInt()* to the caller *main()*.

Procedural Programming in C++

- Procedural code in C++:
    - Program execution <u>starts in the function *main()*</u>, from where <u>other functions are being called</u>.
    - <u>After the execution of a function ends, control flow will continue in the calling function.</u>
    - <u>Effectively the program itself ends, when the last function called from *main()* returns to *main()*.</u>
    - I.e. a C++ program is basically a <u>hierarchy of function calls</u>!

```
double printPrompt() {
    std::cout<<"Please enter a number (base) to be squared:"<<std::endl;
    double base;
    std::cin>>base;
    return std::fabs(base);
}

int main()  {
    double base = printPrompt();
    std::cout<<"Result: "<<std::pow(base, 2)<<std::endl;
}
```

calls predefined *std::fabs()*

calls "our" *printPrompt()*

calls predefined *std::pow()*

High level

High level code
Calls low level code

Low level

main()

printPrompt()   std::pow()
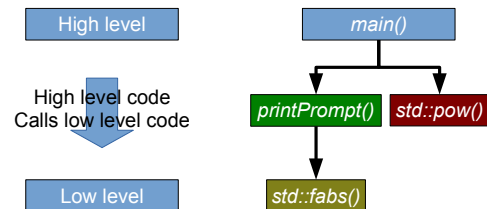
std::fabs()

- In fact coding functions is the <u>central activity of C++ programmers</u>!
    - <u>C++ programers write new functions and call them</u>, or they <u>call predefined functions.</u>

- The function *std::fabs()* is defined in <cmath>. It calculates the absolute value of its argument.

Functions with Parameters – Part I

- Let's continue enhancing the reusability of our program just <u>by increasing the amount of decomposed code</u>.

- It would be good, if we could <u>exchange the prompted text</u>, which is fixed to "The number should be greater than ten." now.
    - We can achieve this with a global variable of type cstring (const char*), <u>that stores the prompted text</u>:

```cpp
const char* promptText; // global variable promptText

int printPrompt() {
    std::cout<<"Please enter a number:"<<std::endl;
    std::cout<<promptText<<std::endl;
    int number;
    std::cin>>number;
    return number;
}

int main() {
    promptText = "The number should be greater than ten.";
    int number = printPrompt();
    std::cout<<"You entered "<<number<<"!"<<std::endl;
}
```

- <u>This idea works the same, as when we use a global to communicate the *number* via a side effect in an earlier example.</u>
    - The only difference is that we are <u>setting the global *promptText* from *main()*</u> and <u>read it from *printPrompt()*</u>.
    - But as we already know: side effects via globals can be really bad. How can we improve the situation?   31

Functions with Parameters – Part II

- Similar to returning values, functions support <u>accepting values</u>, so called <u>arguments (args)</u>, when they are called.

- Therefor C++ allows defining <u>functions with parameters (params)</u>, which <u>transport arguments into function bodies</u>.
    - Let's rewrite our code to use a function <u>with a parameter to handle flexible prompts</u>.

```cpp
int printPrompt(const char* promptText) {
    std::cout<<"Please enter a number:"<<std::endl;
    std::cout<<promptText<<std::endl;
    int number;
    std::cin>>number;
    return number;
}

int main() {
    const char* promptText = "The number should be greater than ten.";
    int number = printPrompt(promptText);
    std::cout<<"You entered "<<number<<"!"<<std::endl;
}
```

Hint
Similar to return statements, parameters contribute side effect free programming (e.g. programming without side channels with global variables).

    - Here we <u>pass the argument *promptText* to the function *printPrompt()*</u>.

    - The output shows, that the <u>passed argument changes the behavior of the function</u> (console output).

- <u>Parameters allow controlling the behavior of functions from outside by passing arguments.</u>
    - <u>… in opposite to globals instead of args, where a function's behavior would be controlled via side effects.</u>

32

- After the introduction of parameters, we have to review the anatomy of functions.
  - This is the anatomy we've seen so far:

parameter

```
void  doSomething1 (int param) {
    // pass
}
```

  - Obviously, the new thing in the anatomy is the variable *param*, the parameter, which is written in the function's parentheses.

- C++ also supports functions with multiple parameters, a so called parameter list, parameter set or just parameters/params.

parameter list

```
void  doSomething2 (int param, int param2) {
    // pass
}
```

Hint
The params are separated by commas, not semicolons!

- C++ doesn't support the variable-list notation for parameters!

variable-list not supported

```
void  doSomething2 (int param, param2) {
    // pass
}
```

33

- C++ allows to specify parameters without a name, this is interesting to keep parameters open for future usage. More often anonymous parameters are used in function declarations, so that only the bare signature of the function remains.

Functions with Parameters – Part IV

- When we strip a function down to its <u>name</u> and <u>ordered list of types in the param list</u>, we get the signature of this function.

```
void  doSomething1 (int param) {
    // pass
}
```

```
void  doSomething2 (int param, int param2) {
    // pass
}
```

signature ↓

signature ↓

doSomething1(int)

doSomething2(int, int)

- So we come to a more complete, but still incomplete and preliminary anatomy of a function:

return type   identifier   parameter list

```
void|<Type1> identifier  ([<Type2> [paramIdentifier1][,<Type3> [param2Identifier], ...]])  {
    // A block of code (function body).
    [return valueOfType1;]
}
```

} function body

  - And of its signature (the return type is no part of the signature):

identifier   parameter list

```
identifier ([<Type2> [paramIdentifier1][,<Type3> [param2Identifier], ...]])
```

34

Features of Functions – Parameters and Arguments – Part I

- Often the terms argument and parameter are used interchangeably, but this is <u>completely wrong</u>!

- An <u>argument</u> is the <u>value</u>, we actually <u>pass to a function</u> on the caller side.
    - It doesn't matter, whether we pass a variable, a value as result of an expression or a literal: <u>in the end only a value is passed</u>.

```
// Variables as arguments:
int takenPressureA = 15, takenPressureB = 78;
int result1 = sum(takenPressureA, takenPressureB); // Arguments: 15 and 78
```

```
// Results of evaluated expressions as arguments:
int toleranceA = 5, toleranceB = 17;
int result2 = sum(takenPressureA + toleranceA, takenPressureB + toleranceB);  // Arguments: 20 and 95
```

```
// Literal values as arguments :
int result3 = sum(12, 31); // Arguments: 12 and 31
```

> **Good to know**
> To drive people completely crazy, arguments are sometimes called <u>actual parameters</u> and parameters are sometimes called <u>formal arguments</u>.

    - <u>Accepting passed values on the callee side (i.e. putting them into variables) is basically the same like initializing a variables.</u>
    - C++ allows implicit conversions while accepting arguments and storing them in parameters.

- A <u>parameter</u> is a <u>variable in a function</u>, <u>which holds the value of an argument, that was passed to the function</u>.
    - When a function is called, the arguments' values are copied into its parameters.

```
int sum(int x, int y) {
    return x + y;
}
```

    - *sum()*'s *x* and *y* will hold the values 15 and 78 to compute *result1*, 20 and 95 to compute *result2* and 12 and 31 to compute *result3*.

- Parameters <u>allow</u> controlling the behavior of functions, arguments <u>do actually</u> control the behavior of functions. 35
    - In *sum()*, *x* and *y* allow specifying summands and the arguments are the actual summands, which control the result of *sum()*.

Features of Functions – Parameters and Arguments – Part II

- To understand params, it makes a lot of sense to discuss the caller side <u>apart</u> from the callee side.

- We called *printPrompt()* with an arg, i.e. the variable *promptText*. We say, that we <u>pass *promptText* to *printPrompt()*</u>:

```
const char* promptText = "The number should be greater than ten.";
int number = printPrompt(promptText);
```

  – Mind, how the code reads like prosaic English: "print prompt with the prompt text"

  – The call works, because <u>the arg *promptText* is of type <u>const char*</u> and *printPrompt()* awaits (or accepts) an arg of type <u>const char*</u>.

- The flexible syntax of C++ allows <u>to replace passing variables with passing literals</u> or <u>more complex expressions</u>.

```
int number = printPrompt("The number should be greater than ten.");
```

- If a function accepts multiple params, we have to pass a <u>list of argument values separated with commas</u>.
  – Here the double values 10 and 2 are passed as a comma separated list of arguments to *std::pow()*:

```
// <cmath> _very_ simplified
namespace std {
    double pow(double base, double exponent);
    // a lot more declarations exist in <cmath>
}
```
```
double result1 = std::pow(10, 2);
```

- <u>If a function specifies params, it is required to pass all arguments to this method to satisfy all params!</u>

```
double result2 = std::pow(10); // Invalid: No matching function for call to 'pow'
```

36

- Sometimes, we are using functions, which are called with the same or partially the same arguments quite often.

- Assume the function *sum4()*, which can add up to 4 arguments:

```
int sum4(int w, int x, int y, int z) {
    return w + x + y + z;
}
```

  – But sometimes, we want to add only two or three summands, then we have to <u>pass 0s to satisfy remaining parameters</u>:

```
int mySum = sum4(2, 3, 0, 0);
```

- We can set <u>default arguments</u> for parameters, so that <u>unsatisfied parameters get a default value</u>:

```
int sum4(int w, int x, int y = 0, int z = 0 ) {
    return w + x + y + z;
}
```

  – With this definition of *sum4()*, we can just leave the last two params alone, not passing any args and <u>let them default to 0</u> each:

```
int mySum = sum4(2, 3);
```

- Important facts about default arguments:
  – The parameters with defaults arguments <u>must be the ones on the very right side of the param list</u>.
  – All params can have default arguments! Default arguments can be added afterwards via function declarations.
  – Default arguments are <u>evaluated at run time</u>, i.e. we can also use <u>expressions</u> or <u>other function calls</u> of <u>almost any type</u>.

- Usually, default arguments are to be put at the function declaration, so that any caller #includes the same declaration, guaranteeing, that all callers use the same default arguments potentially.

Features of Functions – Parameters and Arguments – Part IV

- All right, now to the callee side! So, functions can accept a list of arguments as data to control their behavior.
    - The functions' params store the values, which were passed as arguments, when they were called.

- Let's discuss the function *sum()*, which has two params:
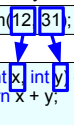
```
int sum(int x, int y) {
    return x + y;
}
```

    - *sum()* calculates the sum of two ints and returns that sum to the caller.

- The accepted arguments are declared as comma separated variables in the parentheses of the function.
    - And … these variables are called parameters.
    - When a function is called, we have to pass the args in exactly the order, in which the "matching" params were declared.

```
int result = sum(12, 31);

int sum(int x, int y) {
    return x + y;
}
```

    - This is called positional argument passing: the position of the args must match the params by their position in the param list.

Features of Functions – Parameters and Arguments – Part V

- Parameters act like local variables in the function.
  - The <u>name of the argument variable has no connection to the name of the param</u>, it can be completely different:

  ```
  int printPrompt(const char* promptText {        const char* blaBlaBla = "The number should be greater than ten.";
      // pass                                       // The passed argument is a variable of name blaBlaBla, but the function's param is named promptText:
  }                                                 int number = printPrompt(blaBlaBla);
  ```

  - It means we <u>cannot have two params with the same name</u>! E.g. the params of the function need to have differing names:

  ```
  int sum(int x, int x) { // variable x is already defined in method sum
      return x + x;
  }
  ```

  - It also means that the <u>params' names need to be different from the names of any other locals</u>:

  ```
  int printPrompt(const char* promptText) {
      const char* promptText = "example text"; // variable promptText is already defined in function printPrompt
      // pass
  }
  ```

- Parameters can be of <u>any C++ type</u>, i.e. fundamental as well as compound types or user defined types.
  - The types of params in the parameter-list can also be <u>mixed freely</u>, e.g. assume the function *sum2()* accepting int and short:

  ```
  int sum2(int x, short y) {
      return x + y;
  }
  ```

- The maximum count of params/arguments can <u>differ</u> from compiler to compiler, usually the maximum is at 256.

39

- Mind that <u>we have to write a pair of empty parens</u>, <u>when we want to call a function without params</u>!

- The order of evaluation of expressions as arguments to functions is not defined in C++!

- Consider the function *sum2()*:

```
int sum2(int x, short y) {
        return x + y;
}
```

```
int i = 0;
int result = sum2(i = 2, i++);
```

> **Good to know**
> The situation with function calls and order of execution is even worse than with simpler expressions:
> E.g. if we have an expression like *h(g(), f())* or *d() * s()* how can we know, which functions are being called first? And this does really matter! It is relevant to know that, because the function calls can have side effects we don't see in the call! This can be nasty, esp. when we use UDT constructor calls!

  - As you may have guessed, the order of evaluation, i.e. whether *i = 2* or *i++* is evaluated first is not known to us!

  - At least all arguments of the function call must be evaluated before the evaluated arguments are passed and the function itself is executed.

    - => It means, after the evaluation of all arguments of a function call, C++ sets a sequence point.

- Actually, C++ defines the so-called sequence operator, which is just represented by a , – i.e. it is the comma-operator.

  - The sequence operator has nothing to do with the comma-separator we use to separate function arguments!

  - But the paradox is, that the sequence operator really is meant to define a certain evaluation order.

```
int i = 0;
i++, i = 2; // The sequence-operator
// i = 2
```

```
int i = 0;
i = 2, i++; // The sequence-operator
// i = 3
```

```
int i = 0;
// No sequence-operator here:
int result = sum2(i = 2, i++); // The comma used as separator!
```

40

- To control how the evaluation of arguments is done, we can order the execution ourselves: We just calculate <u>each argument in a separate statement</u> before the calculated arguments are then passed to the function. – We know this will work, because C++ sets sequence points after each statement!

Experiments on Order of Evaluation – Part I

- Let's consider these functions <u>having side-effects</u> (printing texts to the console) and returning values:

```cpp
int a() {
    std::cout<<"a";
    return 1;
}
```

```cpp
int b() {
    std::cout<<"b";
    return 2;
}
```

```cpp
int c() {
    std::cout<<"c";
    return 3;
}
```

- When we call these functions in an expression to directly add their results, <u>in which order will they be executed</u>?

```cpp
int result = a() + b() * c();
std::cout<<" result: "<<result<<std::endl
```

  - Actually, <u>we don't know</u>, <u>it could be all thinkable permutations of *a()*, *b()* and *c()*</u> resulting in such possible outputs:

Terminal	Terminal
NicosMBP:src nico$ ./main abc result: 7 NicosMBP:src nico$	NicosMBP:src nico$ ./main cab result: 7 NicosMBP:src nico$

Terminal	Terminal
NicosMBP:src nico$ ./main acb result: 7 NicosMBP:src nico$	NicosMBP:src nico$ ./main bca result: 7 NicosMBP:src nico$

Terminal	Terminal
NicosMBP:src nico$ ./main bac result: 7 NicosMBP:src nico$	NicosMBP:src nico$ ./main cba result: 7 NicosMBP:src nico$

> **Mind**
> The undefined order of evaluation does only make the side-effects' order undefined, the order of how operators are applied, i.e. the precedence, is clearly defined!

  - <u>This is so, because the order of evaluation is undefined in C++!</u>

41

- Mind, that the <u>result of the expression *a() + b() * c()* is always 7</u>: <u>although the eval order is undefined, precedence is defined</u>.

Experiments on Order of Evaluation – Part II

- Let's consider the functions *sum3()*, which accepts 3 ints:

```
int sum3(int x, int y, int z) {
    return x + y + z;
}
```

- When we <u>call *sum3()* and pass the function calls to *a()*, *b()* and *c()*</u>, in which order will they be executed?

```
int result = sum3(a(), b(), c());
std::cout<<std::endl;
```

 – Actually, <u>we don't know</u>, <u>it could be all thinkable permutations of *a()*, *b()* and *c()*</u> resulting in such possible outputs:

Terminal	Terminal
NicosMBP:src nico$ ./main abc NicosMBP:src nico$	NicosMBP:src nico$ ./main cab NicosMBP:src nico$

Terminal	Terminal
NicosMBP:src nico$ ./main acb NicosMBP:src nico$	NicosMBP:src nico$ ./main bca NicosMBP:src nico$

Terminal	Terminal
NicosMBP:src nico$ ./main bac NicosMBP:src nico$	NicosMBP:src nico$ ./main cba NicosMBP:src nico$

 – <u>This is so, because the order of evaluation is undefined in C++!</u>

42

Features of Methods – Call by Value

- When a function is called, the <u>passed arguments are copied into "their" parameters</u>.
    - I.e. the <u>variables used as arguments have no "connection" to the values in the function's params</u>, this is called <u>call by value</u>!
    - The general opposite of call by value is <u>call by reference</u>. <u>However, call by value is the default behavior in C++.</u>

- Practical example: the "call by value"-phenomenon leads a method like *swap()* to fail:

```cpp
// Suspicious implementation of swap()!
void swap(int first, int second) {
    int temp = first;
    // Assignment affects only the parameter!
    first = second;
    // Assignment affects only the parameter!
    second = temp;
}
```

```cpp
int a = 12, b = 24;
swap(a, b);
// … but the values of a and b won't be swapped:
std::cout<<"a: "<<a<<", b: "<<b<<std::endl;
//>a: 12, b: 24
```

    - Assigning to the params *first* and *second* in *swap()* <u>won't affect the values in the variables, which were passed as args</u>!

- In a future lecture, we'll discuss, how we can write a function like *swap()* the correct way in C++.
    - Btw: *swap()* could also be implemented with globals and the usage of side effects!

43

Features of Functions – Function Overloading – Part I

- Quite often we want to <u>extend existing functions</u>, so that they support <u>more parameters to control their behavior</u>.
  - *sum()* is a valid example here, <u>up to now, we can only add two summands</u>.

- On the other hand, we can easily write a <u>new function</u>, *sum3()*, which adds three passed int arguments together:

```
int sum3(int x, int y, int z) {
    return x + y + z;
}
```

- Soon, more *sumX()*-like functions will be needed to add more and more ints, with a certain function naming schema:

```
int sum4(int w, int x, int y, int z) {
    return w + x + y + z;
}
    int sum5(int v, int w, int x, int y, int z) {
        return v + w + x + y + z;
    }
        int sum6(int u, int v, int w, int x, int y, int z) {
            return u + v + w + x + y + z;
        }
```

```
// We have multiple sumX() methods, which carry different param lists:
int result1 = sum(12, 13);
int result2 = sum3(12, 13, 14);
int result3 = sum4(12, 13, 14, 15);
int result4 = sum5(12, 13, 14, 15, 16);
int result5 = sum6(12, 13, 14, 15, 16, 17);
```

  - The idea is just to <u>encode the count of parameters in the signature, e.g. in the function name</u>.

- Among other improvements we could do to this function, we can simplify its naming schema with so called <u>overloads</u>.
  - In this lecture we'll only discuss overloads, but in a future lecture we'll learn about more means to esp. enhance functions like *sum()*.

- Ok, so let's discuss function overloads.

44

Features of Functions – Function Overloading – Part II

- The syntactic, or declarative format of function overloads is really simple to get!
  - We'll just have multiple free functions having the same name and return type, but differing in their signature:

```
int sum(int x, int y) {
    return x + y;
}
int sum3(int x, int y, int z) {
    return x + y + z;
}
int sum4(int w, int x, int y, int z) {
    return w + x + y + z;
}
int sum5(int v, int w, int x, int y, int z) {
    return v + w + x + y + z;
}
int sum6(int u, int v, int w, int x, int y, int z) {
    return u + v + w + x + y + z;
}
```

to overloads

```
int sum(int x, int y) {
    return x + y;
}
int sum(int x, int y, int z) {
    return x + y + z;
}
int sum(int w, int x, int y, int z) {
    return w + x + y + z;
}
int sum(int v, int w, int x, int y, int z) {
    return v + w + x + y + z;
}
int sum(int u, int v, int w, int x, int y, int z) {
    return u + v + w + x + y + z;
}
```
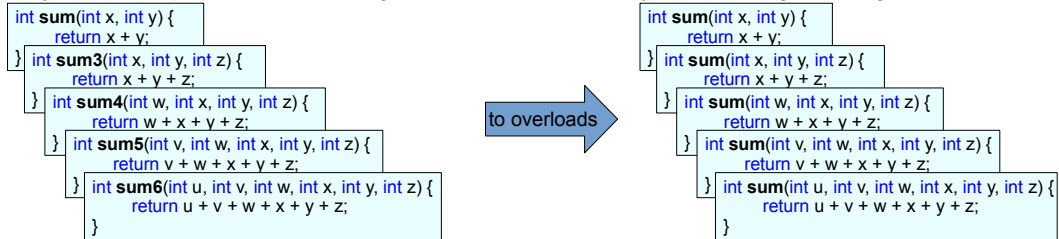
- Instead of differently named *sumX()* functions we have five overloads of a function with the same name: *sum()*.
  - We really only have different variants of the *sum()* function, which only differ in params, their behavior is principally the same.
  - Overloads allows functions having the same name but different parameter lists!

- Therefor overloaded functions read very natural, they are simple to use, or "discover" and program:

```
// Now we call, or "reach" the function sum() having five overloads:
int result1 = sum(12, 13);
int result2 = sum(12, 13, 14);
int result3 = sum(12, 13, 14, 15);
int result4 = sum(12, 13, 14, 15, 16);
int result5 = sum(12, 13, 14, 15, 16, 17);
```

45

Features of Functions – Function Overloading – Part III
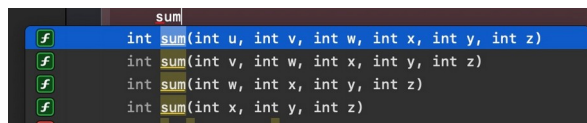
- A popular programming practice to implement overloads is <u>delegation</u>.
  - In this case delegation means, that an <u>overloaded function just calls one or many of its own overloads</u>.

```
int sum(int u, int v, int w, int x, int y, int z) {
    return u + v + w + x + y + z;
}
```

reimplemented
with delegation

```
int sum(int w, int x, int y) {
    return w + x + y;
}

int sum(int u, int v, int w, int x, int y, int z) {
    return sum(u, v, w) + sum(x, y, z);
}
```

- Usually, IDEs support a kind of "fanning out" view to show the overloads of a certain function.
  - For example Xcode:

```
sum
ƒ   int sum(int u, int v, int w, int x, int y, int z)
ƒ   int sum(int v, int w, int x, int y, int z)
ƒ   int sum(int w, int x, int y, int z)
ƒ   int sum(int x, int y, int z)
```

- Actually, the possibilities we have using overloads don't end here!

46

Features of Functions – Function Overloading – Part IV

- Up to now we've only discussed overloads of functions having more and more parameters, i.e. <u>differing param counts</u>.

- C++ also allows overloading functions with completely <u>different parameter types</u>.
    - E.g. let's introduce an other overloads of *sum()* accepting combinations of <u>int and double</u> arguments:

```
double sum(int x, int y) {
    return x + y;
}
    double sum(int x, double y) {
        return x + y;
    }
        double sum(double x, int y) {
            return x + y;
        }
            double sum(double x, double y) {
                return x + y;
            }
```

```
int summand1 = 12, summand2 = 34;
double result = sum(summand1, summand2);
double doubleSummand1 = 65;
double result1 = sum(doubleSummand1, summand2);
```

Good to know
C++ performs a set of implicit conversions, to select candidate functions for the correct overload to call. However, a standard conversion from int to double is not performed here, because an int param is the better match for an int arg than a double param.

   - Here, we changed the return type of *sum()* to double, because mixed arithmetics of int and double should produce double results.

- So, we and up with a lot of *sum()* overloads:

47

- The correct overloaded method to call is <u>determined by the compiler</u>.
  - The compiler <u>selects the matching overload</u> by <u>comparing the list of passed arguments</u> with the <u>available signatures</u>

```
int summand1 = 12, summand2 = 34;
double result = sum(summand1, summand2);
double doubleSummand1 = 65;
double result1 = sum(doubleSummand1, summand2);
```
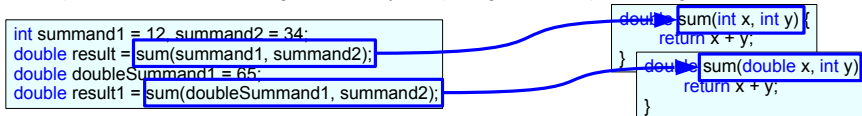
```
double sum(int x, int y) {
    return x + y;
}
double sum(double x, int y) {
    return x + y;
}
```

  - The way the compiler "discovers" the correct overloads introduces some <u>restrictions</u>.

- <u>return</u> types are no part of a function's signature! We cannot overload methods based on <u>return</u> types:

```
int sum(int x, int y) {
    return x + y;
}
// Won't compile: Functions that differ only in their return type cannot be overloaded
double sum(int x, int y) {
    return x + y;
}
```

- That overloads are resolved by the compiler is very important. It means that the selection of overloads happens in a very early build phase.
  - Btw. selecting <u>function overloads at run time</u> can also be implemented in C++ with so called "multiple dispatch" using object oriented features.

Features of Functions – Function Overloading – Part VI

- Sometimes, the compiler performs <u>unexpected overload resolutions</u>. Let's reconsider having following *sum()* overloads:

```
int sum(int x, int y) {
    return x + y;
}
    int sum(short x, int y) {
        return x + y;
    }
```

- Which overload of *sum()* will be called in this example?

```
int result = sum(65, 37);          int sum(int x, int y) {
                                       return x + y;
                                   }
```

  – Indeed the overload *sum(int, int)* is called!

  – Why? The arguments 65 and 37 match better with this overload, because they are int literals!

  – The overload *sum(short, int)* is not chosen, because <u>it would require the compiler to implicitly convert the int 65 to a short</u>.

- <u>The compiler primary resolves overloads, which require no implicit type conversion!</u>

- But we can <u>force</u> the compiler to resolve to *sum(short, int)*!

  – We do so by <u>explicitly converting</u> 65 to short with a <u>cast</u>:

```
int result = sum(static_cast<short>(65), 37);     int sum(short x, int y) {
                                                      return x + y;
                                                  }
```

49

Features of Functions – Function Overloading – Summary

- So, the <u>combination</u> of <u>count</u>, <u>order</u> and <u>types of a method's params</u> makes the <u>signature of the function</u>.
  - With the introduction of overloads, a function can have <u>multiple signatures</u>.
  - Overloads of a specific function should represent <u>similar algorithms</u> working <u>with different types</u> having <u>the same name</u>.

- Vice versa this means, that each <u>function overload needs to have a different signature</u>!
  - The return type does <u>not contribute to the signature</u> in C++! => <u>Overloading cannot be based on different return types!</u>
  - <u>The parameter names play no role for a function's naked signature</u>

- <u>Tips</u>
  - <u>The set of overloaded functions should really just be special cases of each other, but do not do completely different things!</u>
  - <u>Overloading can be combined with default arguments, but this must be done carefully.</u>

50

Functions are called – there is no Code Replacement

- A common misunderstanding: function call leads to code replacement.
    - This is not what's going on!


```cpp
int a() {
    return 2 + 3;
}
```

b() calls a()

```cpp
void b() {
    int result = a();
    std::cout<<result<<std::endl;
}
```

NO!

```cpp
void b() {
    int result = 2 + 3; // No!
    std::cout<<result<<std::endl;
}
```
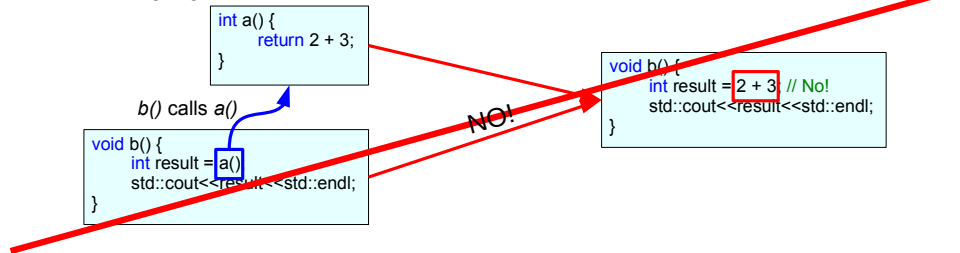
- The functions *a()* and *b()* do always exist, their code is not "merged" somehow.

- Functions are rather called procedurally in C/C++.

- (Code replacement can be achieved with macros in C/C++.)

51

- There is no code replacement going on, it would lead to code bloat, and thus bigger executables after compilation.

Procedural Calling of Functions in Action

- We already discussed that functions can call other functions.
    - When a function <u>completes</u>, <u>execution returns</u> to <u>where the function was called from</u>.
    - "Top-level" functions call "lower-level" functions. So, as we already recognized, there is a <u>call hierarchy</u>.

```
// <cmath.c> _very_ simplified
namespace std {
    double pow(double base, double exponent) {
        // pass
        return value;
    }
}
```

```
int main() {
    double result = square(3);
}
result = 9
```

(3, 2)

9

(3)

```
#include <cmath>

double square(double base) {
    return std::pow(base, 2);
}
```

9
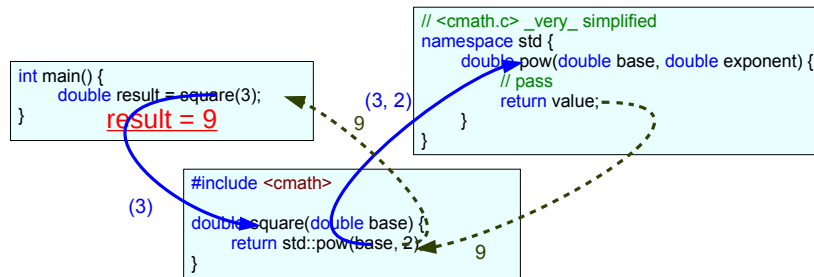
- There is also an alternative of procedural calling of methods: <u>recursive calling of functions</u>! Let's understand <u>recursion</u>...

Recursive Calling of Functions

- Some <u>problems can be described as smaller instances of the same problem</u>.
    - In maths such descriptions are called <u>recursive descriptions</u>.
    - For example the factorial function n! (1 x 2 x 3 x 4 … x n):

$$n! = \begin{cases} 1; & n=0 \\ n \cdot (n-1)!; & n>0 \end{cases} \quad n \in \mathbf{N}$$

> **Good to know**
> Some folks use the blackboard bold notation to name standard number sets, e.g. $\mathbb{N}$. Virtually this notation is only meant to be used on the board and/or with handwriting in order to simulate bold face. In print, the names of the standard number sets are just written in bold face.
> We'll stick to the bold face convention (**N**) instead of the blackboard bold notation ($\mathbb{N}$).

- Factorial can be expressed as a <u>recursive function</u> in C++.
    - In maths, recursion means that a equation is <u>defined in terms of itself</u>.
    - <u>In C++, recursion means that a function calls itself.</u>

- How to write a recursive function?
    - A <u>subproblem of the whole problem</u> must be used as <u>argument for recursive calls</u>.
    - And it is needed to <u>consider the base case</u>, when the <u>recursion terminates</u>!
    - (The progression of the recursive calls must tend to the base case.)
    - If we fail to consider one of these cases, we may end with an <u>infinite recursion</u>!

- <u>Where to use the "factorial" function?</u>
    - E.g. the factorial is used to calculate the count of permutations of a set of elements.
- Infinite recursions run as long as all the stack memory of the executing thread is exhausted.

Recursive Calling of Functions in Action
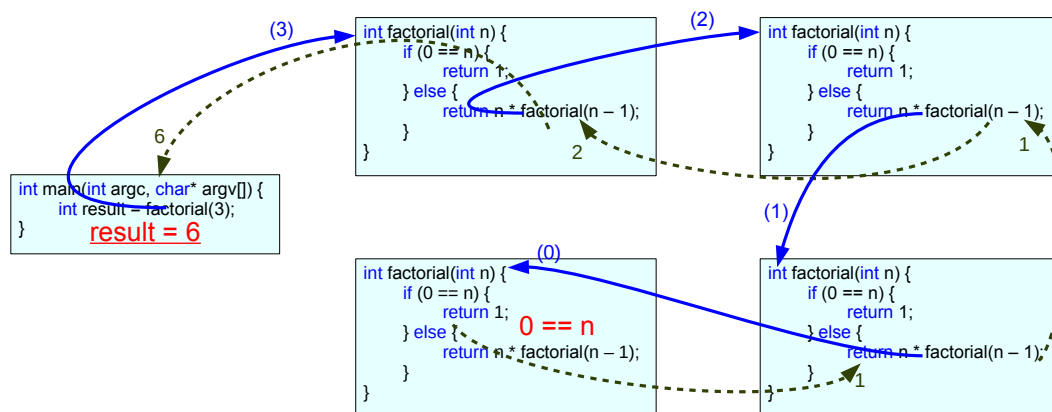
54

- In the beginning, recursion is difficult to understand for most people. – They think about all the calls of a recursive function, in order to understand, if it is correct or not. They rather understand a function as a "machine" that processes inputs and returns outputs. → A better way to understand a function is that a function is a process. E.g. "process all documents and process all documents referred to in these documents".

Various Implementations of the Factorial Algorithm

$$n! = \begin{cases} 1; & n=0 \\ n \cdot (n-1)!; & n>0 \end{cases} \quad n \in N$$

```
int factorialRecursive(int n) {
        if (0 == n) {
                return 1;
        } else {
                return n * factorialRecursive(n - 1);
        }
}
```

```
int factorialIterative(int n) {
        int result = 1;
        for (int i = n; i > 0; --i) {
                result *= i;
        }
        return result;
}
```

```
int factorialNiceRecursive(int n) {
        return (0 == n)
                ? 1
                : n * factorialNiceRecursive(n - 1);
}
```

Good to know
Factorial could also be implemented with a table, which associates each n with n!. Such an implementation wouldn't use a loop/iteration or recursion. – The downside of this approach: we as programmers have to calculate the results and fill in the table. And btw. this table would be really large, ideally infinite in size, but practically limited by the memory of the system.

- Mind, that these implementations <u>do not deal with negative _n_s</u> (n! is only defined for **N**)!
    - We'll solve this problem, <u>i.e. fix this bug</u>, in a future lecture!

55

Recursion and Recursive Functions in the Wild

- Recursion is useful to operate on recursive data. So called data trees!
    - E.g. a graphical TreeView contains trees, that contain subtrees, that contain subtrees...
    - XML data is also a cascading tree of trees.
    - Directory and file hierarchies.
    - Network analyses, e.g. calculation of network efficiency and network load.

- Complex problems and puzzles can be solved easily with recursion.
    - Maths: permutations of a set, numeric differentiation/integration
    - Games: Towers of Hanoi, Sudoku, Chess puzzles etc.

- Definition of subproblems to be solved by multiple CPUs independently.
    - The idea is to use CPU resources most efficiently -> <u>Divide and conquer!</u>
    - Filters in graphics programming.
    - Data analyses in networks and databases.

- Warning: recursive code is often elegant, but it comes with its payoffs.
    - Concretely, too many recursive calls will end in an <u>exhaustion of stack memory (e.g. in C++)</u>, a so called stack overflow.
    - Recursive code is tricky to get for humans, because <u>people rather think iteratively</u>.

56

Functional Programming – Part I

- It makes sense to take another look to the iterative and recursive implementations of n!

```
int factorialIterative(int n) {
    int result = 1;
    for (int i = n; i > 0; --i) {
        result *= i;
    }
    return result;
}
```

```
int factorialRecursive(int n) {
    return (0 == n)
        ? 1
        : n * factorialRecursive(n - 1);
}
```

- *factorialIterative()*, i.e. the iterative n! uses a for loop statement to "drive" the calculation.
    - This loop statement internally modifies the state of the variables *result* and *i*. It uses side effects on those local variables.

- The recursive n! just conditionally calls itself with new values as arguments and consumes its own return values.
    - It is not driven by multiple statements. It uses one single expression. It looks like having no "moving parts".
    - For god's sake function calls are not replaced by their function body: that would be a disaster when recursion is used!

- The idea to code functions without side effects is an aspect of the paradigm of functional programming (fp).
    - Recursion is closely associated with fp, because recursion is used in fp to express loops without side effects.
    - Mind, that "classical", i.e. imperative loops always deal with side effects, e.g. a for loop incrementing its counter variable.

57

Functional Programming – Part II – Relation to Procedural Programming

- Procedural programming:
  - Makes us think in terms of sub programs, which modify data locally or globally.
  - It rather has a focus on side effects and variables.
  - Mentally, it rather makes us think in mathematical procedures/algorithms.

- Functional programming:
  - Makes us think in terms of functions, which are kind of close and never modify global state.
  - It rather has a focus on argument values and return values, side effects are not existent.
  - Mentally, it rather makes us think in mathematical formulas.

- Virtually fp is a very early programming concept, which was used in the first programming languages.
  - Most notable example is Lisp, was introduced in 1958.
  - With the advent of procedural languages, fp was almost pushed away from the mainstream.

- However, today's languages re-adopted fp technologies into their programming idioms.
  - Most notable Java 8 and the .NET framework 3.5 with the LINQ technology in the languages C#3, VB9.0 and F#.

58

- The topic of fp requires is own, maybe multiple lectures.

Functional Programming – Part III

- We have just recapitulated the fact, that functions can have side effects and fp can effectively reduce side effects.
  - Hm, why is that a relevant or important topic?

- One relevant aspect is, that <u>fp gives to us better tools to exploit multi core/CPU computing power</u>.
  - But to understand the ideas behind programming multi-processing code with so called <u>multi-threaded programming</u>.

- Another aspect is, that <u>side effects</u> in functions <u>can introduce code, which behaves surprisingly or even wrongly</u>.

Functional Programming – Part IV

- Let's discuss this code, in which *a()* has a side effect on *globalVariable* and *b()* accesses *globalVariable*:

```
int globalValue;

int a() {
        globalValue = 1;
        return 42;
}

double b(int x) {
        return globalValue * x;
}
```

Definitions
The paradigm of functional programming demands, that calling a specific function must yield the same result, when called with the same arguments no matter when or how often the function is called.
This principle is called referential transparency. Referential transparency cannot be guaranteed, when functions have side effects.
A function obeying referential transparency is called pure function.

- With these functions, we can program two versions of *main()* using *a()* and *b()* for a calculation:

```
// Version 1:
int main() {
        double result = a() * b(15);
        // result = 630.0 (most likely)
}
```

```
// Version 2:
int main() {
        double result = b(15) * a();
        // result = 0.0 (most likely)
}
```

  - The *result*, i.e. the value of the expression with the same functions and arguments, depends on the order of function calls!

  - Mind, that side effects could even be more difficult to predict, when we use multiple operator expressions with mixed precedences...

- The result of the expressions *a() * b(15)* and *b(15) * a()* is even more difficult to predict in C++!

  - This is because the order of evaluation in expressions is undefined in C++. – Oh yes!

  - It actually means, that *a() * b(15)* could be evaluated like *b(15) * a()* and *b(15) * a()* could be evaluated like *a() * b(15)*.

60

Organizing Functions in Namespaces – Part I

- So far all user defined functions we saw, were defined as <u>global</u> functions.

- Global functions may <u>clash</u> with equally named/typed other functions.
    - Therefor functions and other objects, commonly the "names" of the <u>standard library</u> reside in the <u>namespace *std*</u> to prevent this.
        - E.g. *std::cout* or *std::pow()*.

- namespaces group <u>free</u> functions semantically.
    - It turns out, that <u>all free functions should reside in namespaces</u>!
    - <u>Don't define global functions</u> (with the exception of *main()*), <u>use namespaces</u> instead!

- C++ namespaces can be understood as directories and functions as files.
    - In a file system directories do contain files.
    - Files with the same name can reside in different directories. -> No clash!

- Let's examine how to define and use namespaces with the function *factorial()*...

61

- <u>What's a free function?</u>
- <u>And what is its opposite?</u>

- Let's put *factorial()* into the namespace Nico, but leave *main()* a global function.
  - We can call *Nico*'s *factorial()* from *main()* by writing the prefix *Nico::* in front of the function name.
  - The new ::-separator is called <u>scope-resolution operator</u>.
  - Selecting a name by its full namespace <u>path</u> is called <u>full qualification</u> of a name.

- Alternatively, we can make <u>all names of a namespace</u> visible to the caller with a <u>using</u> directive:
  - All names of the specified namespace (here: *Nico*) are visible to the namespace, where the using directive was written (here: global).

- Another alternative: write a <u>using declaration</u> to only <u>make all overloads of a specific function</u> visible to the caller:
  - All functions names *factorial()* of the specified namespace (here: *Nico*) are visible to the namespace, where the using declaration was written (here: global).

namespace *Nico*

Global namespace

```cpp
// <main.cpp>
// Function factorial() in the namespace Nico:
namespace Nico {
    int factorial(int n) {
        return (0 == n)
            ? 1
            : n * factorial(n - 1);
    }
}

// Qualify the function name with the
// namespace's name:
int main() {
    int result = Nico::factorial(3);
    std::cout<<result<<std::endl;
}
```

```cpp
// Make all names in the namespace Nico global:
using namespace Nico; // using directive
int main() {
    int result = factorial(3);
    std::cout<<result<<std::endl;
}
```

```cpp
// Make only functions named Nico::factorial() global:
using Nico::factorial; // using declaration
int main() {
    int result = factorial(3);
    std::cout<<result<<std::endl;
}
```

- Mind that namespaces allow us to have any other function named *factorial()* residing in any other namespace than *Nico*; they will not clash with *Nico::factorial()*!

Organizing Functions in Namespaces – Part III

- With namespaces we can have multiple variants of *factorial()* having the same signature but different implementations.

- Having those functions in namespaces we can call them individually using the scope-resolution operator:

```
int main() {
      factorial(4);           // calls the global factorial()
      ::factorial(4);         // also calls the global factorial() with the ::-prefix.
      recursive::factorial(4);   // calls recursive's factorial()
      iterative::factorial(4);   // calls iterative's factorial()
}
```

  - The scope-resolution operator used as bare prefix refers to the global namespace.

- However, because there is a global *factorial()* we cannot add other "*factorial()*s" from other namespaces via a using directive or declaration:

```
int main() {
      using namespace iterative;
      factorial(4); // Invalid! Nameclash: ::factorial() or iterative::factorial()?
}
```

```
namespace recursive {
      int factorial(int n) {
          if (0 == n) {
              return 1;
          } else {
              return n * factorial(n - 1);
          }
      }
}

namespace iterative {
      int factorial(int n) {
          int result = 1;
          for (int i = n; i > 0; --i) {
              result *= i;
          }
          return result;
      }
}

int factorial(int n) {
      return (0 == n) ? 1 : n * factorial(n - 1);
}
```

63

Organizing Functions in Namespaces – Part IV

- namespaces can be <u>nested</u>, to address the contained names, we must use <u>:: multiply along the</u> namespace's <u>path</u>:

```cpp
namespace recursive {
    int factorial(int n) {
        if (0 == n) {
            return 1;
        } else {
            return n * factorial(n - 1); // addresses recursive::factorial(),
        }                                //                  // not ::factorial()
    }
    namespace nice {
        int factorial(int n) {
            return (0 == n) ? 1 : n * factorial(n - 1);
        }
    }
}

int main() {
    recursive::factorial(4);        // calls the factorial() using branching
    recursive::nice::factorial(4);  // calls the factorial() using ?:
}
```

Warning
Nested namespaces are used rarely, they should only be used to avoid name clashes, not to reflect a design.
When the bare ::-prefix is used in a nested namespace, it'll <u>always refer the very global</u> namespace, <u>not the parent</u> namespace of the current namespace.

- Instead of using :: every time to refer to namespace'd, esp. multiply nested names, we can define <u>namespace aliases</u>:

```cpp
#include <iostream>

int main() {
    namespace standard = std;
    namespace rFactorial = recursive::nice;
    standard::cout<<rFactorial::factorial(4)<<standard::endl;
}
```

64

- # Limitations of namespaces: they are "open" they have no access control.
- # Newer versions of C++ add a lot more features to namespaces.

- Before we go on, we have to understand how programs consisting of <u>multiple source code files</u> are programmed.

- The C++ compiler translates C++ code into machine code.
  - Let's do this via the command line to better see, what's going on:

```
// <main.cpp>
#include <iostream>

int main() {
        std::cout<<"Hello World!"<<std::endl;
}
```

C++ Compiler

<main.o>

```
Terminal
NicosMBP:Debug nico$ g++ -c main.cpp
NicosMBP:Debug nico$
```
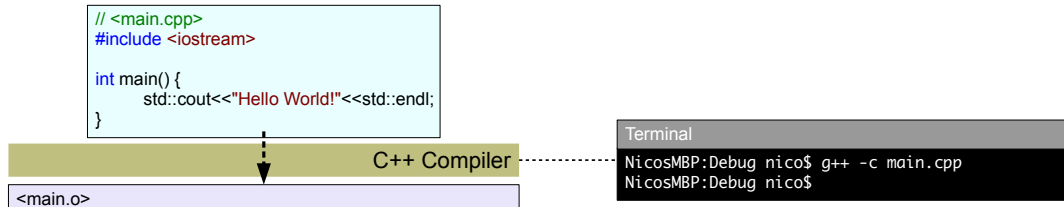
- To compile the file main.cpp we have used g++'s <u>-c option</u> to <u>only compile</u> the file to machine code.
  - C++ allows to <u>compile cpp-files separately</u> this is called <u>separated compilation</u>.

- The result of the <u>compilation is an object-file (o-file)</u>. The o-file contains the machine code of the source cpp-code:

```
Terminal
NicosMBP:Debug nico$ ls
main.cpp  main.o
NicosMBP:Debug nico$
```

65

- However, there is one step, which happens before the compilation step, the preprocessing step. – But usually, the compilation also includes the preprocessing.

Separated Compilation – Part I

- The ideas behind separated compilation:
  - (1) faster compilation: code contained in a cpp-file, e.g. specific functions, <u>must only be compiled once</u> to an o-file
  - (2) better reusability: the specific functions in the o-file can be <u>shared</u>, if they are called from many places.

- To use separated compilation, let's put *Nico::factorial()* into its <u>own separate cpp-file</u> "numeric.cpp" and call it from *main()*:

```
// <main.cpp>
#include <iostream>

int main() {
    std::cout<<Nico::factorial(4)<<std::endl;
}
```

```
// <numeric.cpp>
namespace Nico {
    int factorial(int n) {
        return (0 == n)
            ? 1
            : n * factorial(n - 1);
    }
}
```

```
<main.o>                    <numeric.o>
```

C++ Compiler

```
Terminal
NicosMBP:Debug nico$ g++ -c main.cpp numeric.cpp
main.cpp:5:16: error: use of undeclared identifier 'Nico'
    std::cout<<Nico::factorial(4)<<std::endl;
              ^
1 error generated.
NicosMBP:Debug nico$ ls
main.cpp  numeric.cpp numeric.o
```

- numeric.cpp (we got numeric.o) can be compiled successfully, but main.cpp cannot!
  - So, compilation of main.cpp <u>failed separately</u> and the compilation of numeric.cpp <u>succeeded separately</u>

66

- Why does compilation of main.cpp fail? <u>Because in main.cpp's code, *Nico* or *Nico::factorial(int)* are unknown!</u>

Separated Compilation – Part II

- We must achieve, that main.cpp and numeric.cpp can be compiled separately, but main.cpp doesn't know *Nico::factorial(int)*.

- We can fix this by just presenting the declaration of *Nico::factorial(int)* to main.cpp.
  - Mind that the definition of *Nico::factorial(int)* is still contained in the separate file numeric.cpp!

- So, let's just add the declaration into main.cpp:

```
// <main.cpp>
#include <iostream>

namespace Nico { // declare Nico::factorial(int)
    int factorial(int n);
}

int main() {
    std::cout<<Nico::factorial(4)<<std::endl;
}
```

```
// <numeric.cpp>
namespace Nico {
    int factorial(int n) {
        return (0 == n)
            ? 1
            : n * factorial(n - 1);
    }
}
```

C++ Compiler

`<main.o>`   `<numeric.o>`
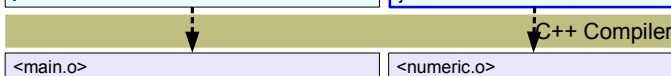
```
Terminal
NicosMBP:Debug nico$ g++ -c main.cpp numeric.cpp
NicosMBP:Debug nico$ ls
main.cpp main.o numeric.cpp numeric.o
```

- To declare *Nico::factorial(int)* properly, a using directive or using declaration would not be enough!
  - using directives/declarations make only names of another namespace visible, but don't unleash the signature of functions! 67

Separated Compilation – Part III

- Now we should think about <u>potential users different from main.cpp</u>, which want to use *Nico::factorial(int)*.
  - Sure, they have just to declare *Nico::factorial(int)* to use this function and compile successfully, <u>but how should they know</u>?
  - Mind, potential users need to know <u>name</u>, <u>namespace</u> and <u>signature</u> of the <u>function to declare</u>!
  - This is a lot of information, esp. if a user wants to use <u>more than only one function</u>!

- To address this <u>lack of information</u> C++ has a <u>convention</u>:
  - (1) A definition cpp-file should have a <u>companion h-file</u>, that contains <u>declarations for each function defined in the cpp-file</u>.
  - (2) The definition cpp-file should <u>#include its companion declaration h-file</u> in the <u>very first #include directive</u>.
  - (3) Other source code files (cpp-/h-files) only #include the declaration h-file to use functions (they know nothing about the definition cpp-file).
  - All right, then let's create numeric.h as companion of numeric.cpp:

```
// <numeric.cpp>
#include "numeric.h"

namespace Nico {
        int factorial(int n) {
                return (0 == n)
                        ? 1
                        : n * factorial(n - 1);
        }
}
```

```
// <numeric.h>

// declare Nico::factorial(int)
namespace Nico {
        int factorial(int n);
}
```

- The more interesting part is, however, <u>where numeric.h is #included</u> in main.cpp and other files.

68

Separated Compilation – Part IV

- All right, so we have to #include numeric.h in main.cpp and as an example also in other.cpp:

```
// <numeric.h>

// declare Nico::factorial(int)
namespace Nico {
    int factorial(int n);
}
```

```
// <main.cpp>
#include <iostream>
#include "numeric.h"

int main() {
    std::cout<<Nico::factorial(4)<<std::endl;
}
```

```
// <numeric.cpp>
#include "numeric.h"

namespace Nico {
    int factorial(int n) {
        return (0 == n) ? 1 : n * factorial(n - 1);
    }
}
```

```
// <other.cpp>
#include "numeric.h"

void calculate(int magic) {
    return Nico::factorial(magic * 34)
}
```

C++ Compiler

<main.o>   <numeric.o>   <other.o>

```
Terminal
NicosMBP:Debug nico$ g++ -c main.cpp numeric.cpp other.cpp
NicosMBP:Debug nico$ ls
main.cpp main.o numeric.cpp numeric.h numeric.o other.cpp other.o
```

- For simplicity, we have put the h-file numeric.h in the same directory as main.cpp, numeric.cpp and other.cpp.
    - If we hadn't, we would have needed to specify a path to numeric.h in the #include directive!
    - Also mind, that because numeric.h is a non-standard h-file, it must be #included written in double quotes, not angle brackets!

69

The C++ Preprocessor

- C'++s <u>preprocessor</u> is a part of the C++ compiler, which <u>loads/includes #included files into the including C++ source code</u>:

```
// <iostream>
// iostream declarations
```

```
// <main.cpp>
#include <iostream>
#include "numeric.h"

int main() {
        std::cout<<Nico::factorial(4)<<std::endl;
}
```

```
// <numeric.h>
// numeric's declarations
```

Good to know
gcc provides the -E flag to create a preprocessed cpp-file:

Terminal
```
NicosMBP:src g++ -E main.cpp > main_pp.cpp
```

C++ Preprocessor

```
// <main_tu.cpp>/
// - all #includes included
// - all macros expanded
```

C++ Compiler
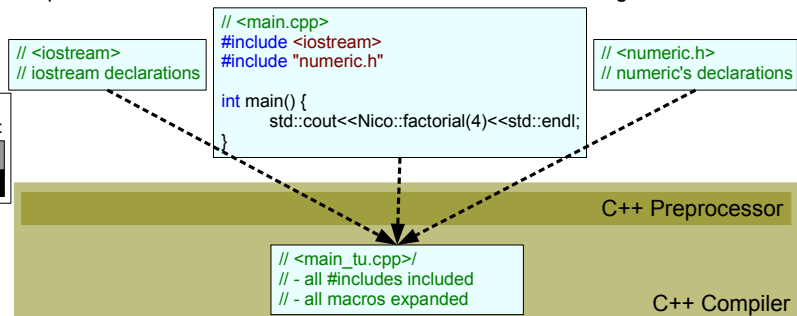
- The preprocessor does a little bit more: <u>it creates more code from our source code</u>, e.g. by <u>expanding</u> so-called <u>macros</u>.
    - The <u>resulting, i.e. preprocessed C++ source code</u> from our source code is summarized as <u>translation unit (tu)</u>.
    - However, the output of the preprocessor is then sent to the C++ compiler's frontend, that continues the translation process.

- For the time being keep in mind: the C++ preprocessor brings our code and declarations together into a tu.

70

- Right after the tu was created, it'll be <u>actually compiled</u>.

The C++ Compiler – Part I

- <u>What</u> the C++ compiler does is simple <u>basically</u>: it translates human readable C++ into machine-readable, object-code.
  - <u>However, how it does the translation is often not so simple</u>.

- Most often a compiler program consists of two components: a <u>frontend</u> and a <u>backend</u>:

  > Frontend
  > - Dissects the passed source code into pieces, this is called <u>parsing</u>.
  > - Checks the syntax of the passed source code, this is called <u>lexical analysis</u>.
  > - => The result is <u>code in an immediate language</u>.
  >
  > Backend
  > - Accepts the frontend's immediate language and <u>builds executable code</u>.
  > - => The result is <u>machine code</u> or <u>binary code</u> or <u>object-code</u>.
  >
  > Compiler

- The frontend:
  - The parsers reads and dissects the sources into an <u>abstract syntax tree (AST)</u> of tokens.
    - Tokens are keywords, operators, literals, symbols and comments.
    - Actually, parsers have a sub-component doing this dissection, which is called <u>tokenizer</u> or <u>lexer</u>.
    - <u>The parsing procedure regards the compiled language's syntax and grammar.</u>
  - The creation of the immediate language is not so spectacular.
    - The idea behind this immediate step is being able to manipulate and optimize immediate code before executable code is created.

71

- When we as human beings listen to speakers, parsing the spoken words, i.e. separating words from the sounds makes the understanding possible.

The C++ Compiler – Part II

- The backend's job is to accept the frontend's immediate code and create an executable file for the target platform.
  - The idea behind having a backend is simple: <u>the backend can be replaced!</u> – <u>Then another type of executable can be generated.</u>

- Usually, we pass a source code file to the compiler, the compiler produces a binary file (after a while of processing):
  - The compilation procedure looks like this:

| Source code files | `// <iostream>`
`// std::cout, std::endl, operator<<`
`// definitions` | `// <main.cpp>`
`// main()'s definition` | `// <numeric.h>`
`// numeric's declarations` | `// <numeric.cpp>`
`// numeric's definitions` |

Preprocessor

| Translation units | `// <main_tu.cpp>`
`// - all #includes included`
`// - all macros expanded` | `// <numeric_tu.cpp>`
`// numeric's definitions`
`// - all #includes included`
`// - all macros expanded` |

Compiler

| Executable files | `<main.o>`
`Object-code`
`0000000100100011010001011011001111000`
`10011010101111001101111011110000000 ...` | `<numeric.o>`
`Object-code`
`1010111100110111001000110100010101100`
`10011011110111100000001001101010111 ...` |

The C++ Compiler – Part III

- C++ is an HLL, which <u>compiles to object-code</u>, thus C++ must be compiled with a compiler matching the target platform.
  - I.e. the code must be compiled for each specific machine-type and OS, we want it to run on.
  - <u>For each machine-type and OS</u> we need <u>a specific compiler,</u> which creates specific object-code.
  - <u>I.e. the source code must not be changed! – We just use another compiler for the target machine.</u>

- <u>With an HLL like C++</u> we compile the <u>very same C++ code</u> into Amiga- or x86-executables with <u>different compilers</u>.
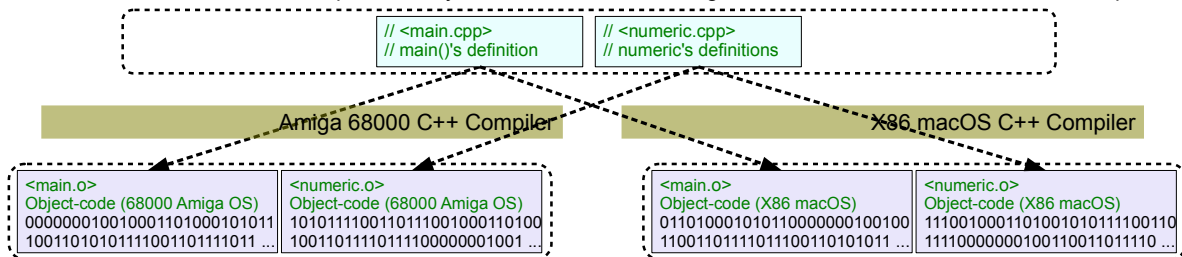
The C++ Linker – Part I

- We are still not at the end! We have several o-files after the compilation, but not an executable program yet!

- We need another step in the build process to link the o-files together to an executable file, the so-called linker-step.

| <main.o> | <numeric.o> | (other o-files or lib-files, esp. standard library) |

C++ Linker

<main>
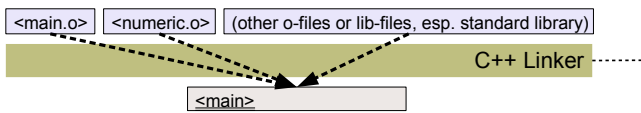
```
Terminal
NicosMBP:Debug nico$ g++ main.o numeric.o -o main -lstdc++
NicosMBP:Debug nico$ ls
main main.cpp main.o numeric.cpp numeric.h numeric.o
NicosMBP:CppExample nico$ ./main
24
NicosMBP:Debug nico$
```

- We must also link the standard library! – The library has definitions for, e.g. *std::cout*, which are only declared in <iostream>.
  - This is done with the -lstdc++ flag on g++' command line.
- The linker also links other libraries into our program to make the code runnable at all. E.g. subsystems for interaction with the OS.
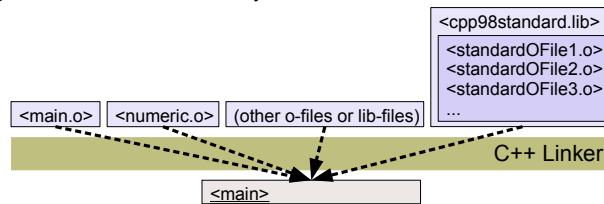
Tipp
g++ allows to combine compilation and linking in one step, i.e. without separated compilation and without explicit creation of o-files:
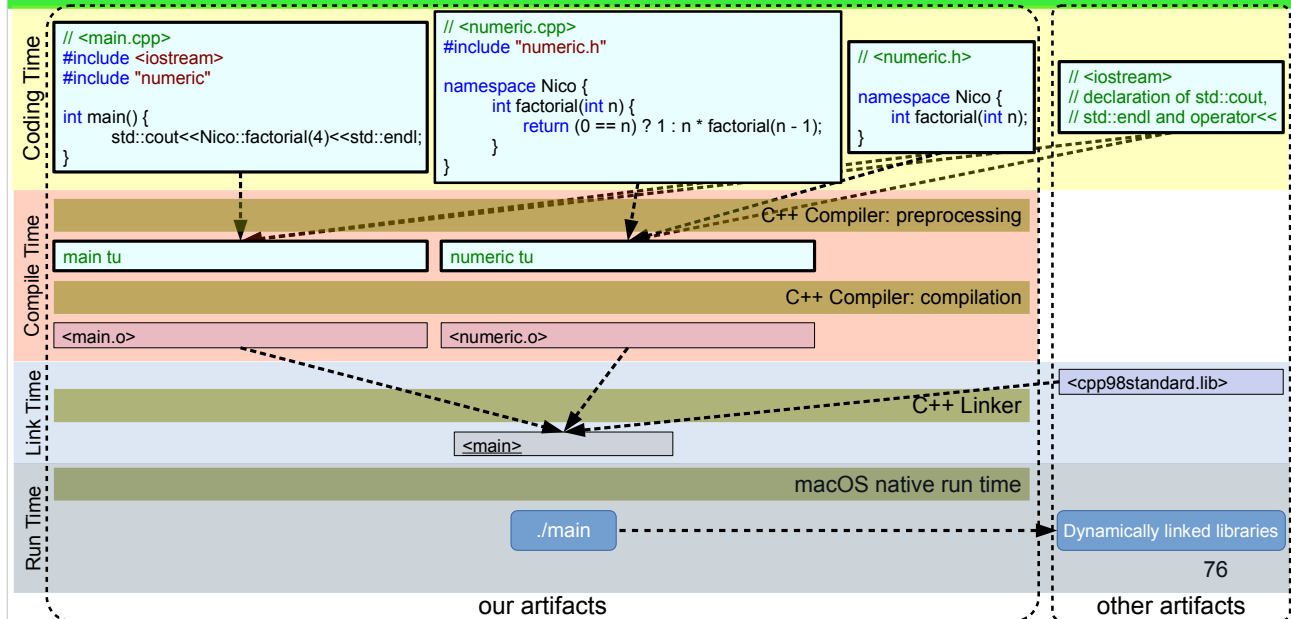
```
Terminal
NicosMBP:Debug nico$ g++ main.cpp numeric.cpp -lstdc++ -o main
NicosMBP:Debug nico$ ls
main main.cpp numeric.cpp numeric.h
NicosMBP:Debug nico$ ./main
24
NicosMBP:Debug nico$
```

74

The C++ Linker – Part II

- Separated compilation also brought the possibility to <u>share compiled o-files among developers</u>.

- In case the same set of o-files must be sent around over and over, we can <u>link o-files together to a library</u>.
    - The advantage of libraries lies in the fact, that they are <u>self-contained</u> (no way to miss an o-file) and can be <u>versioned</u>.

- <u>Basically</u>, a library file (lib-file or lib) is just a <u>collection of o-files</u>.
    - During linking, lib-files can be specified to be linked in <u>just like o-files</u>. This is called <u>static linking</u> of libraries.
    - There also exists <u>dynamic linking</u> of libraries, in which libs are <u>linked at run time</u>, i.e. <u>things come together, when the program runs</u>.
        - However, the support of dynamic linking depends on the platform and is somewhat away from the C++ standard.

- Often the <u>C++ standard library</u> is also a statically linked library.
    - With g++, the flag -lstdc++ does link this library, so we do not need to know its name or location in the file system.

```
                                              <cpp98standard.lib>
                                              <standardOFile1.o>
                                              <standardOFile2.o>
                                              <standardOFile3.o>
  <main.o>  <numeric.o>  (other o-files or lib-files)  ...

                                                    C++ Linker
                                                                        75
              <main>
```

Build Phases Overview

Coding Time

```
// <main.cpp>
#include <iostream>
#include "numeric"

int main() {
        std::cout<<Nico::factorial(4)<<std::endl;
}
```

```
// <numeric.cpp>
#include "numeric.h"

namespace Nico {
        int factorial(int n) {
                return (0 == n) ? 1 : n * factorial(n - 1);
        }
}
```

```
// <numeric.h>

namespace Nico {
        int factorial(int n);
}
```

```
// <iostream>
// declaration of std::cout,
// std::endl and operator<<
```

Compile Time

C++ Compiler: preprocessing

main tu

numeric tu

C++ Compiler: compilation

<main.o>

<numeric.o>

Link Time

C++ Linker

<cpp98standard.lib>

<main>

Run Time

macOS native run time

./main

Dynamically linked libraries

76

our artifacts

other artifacts

Separated Function Definitions

- Related functions should be implemented in separate code files, instead of together with *main()*. – Why that?
  - Each developer can manage just his functions, not interfering with others'.
  - On modification, only the separately modified code files must be compiled separately.
  - Very important: The implemented code can be hidden from its callers.

- Separate and collect function definitions. – What's to do?
  - Collect related function definitions in own cpp/c-files (summarized "c-files", those are the code files).
  - Create companion h-files with function declarations for the functions in each c-files respectively.
  - These "matching" h- and c-files should have the same name apart from the file suffix.
    - The file names should not contain whitespaces or special characters (like umlauts).
  - As C/C++ convention, #include the h-files into the belonging cpp/c-files in the first line.

- Calling the separated functions. – What's to do?
  - In our top-level code file, we've to #include the h-files declaring the functions we need.
  - In the code we can call the functions, because the declarations are visible.
  - However, we have to care for the linker to see the o-files/libs with the definitions.

> **Good to know**
> Header-files are called header-files, because they are usually included at the header section of a file, but the can also included in the body of a source code file!

Separated Compilation and Linkage

- Separated compilation comes for a price: <u>multi-source-file-project are difficulty to handle</u>.
  - In simple projects, we'll only have one code file (with the definition of *main()*).
  - But in multi-file-projects we need to tell the compiler to compile all!
  - Also the linker has to link all the resulting o-files to an executable.

- How to negotiate with the compiler and linker about multiple files?
  - We can, however, <u>compile all c-files separately</u>, e.g. on the console.
  - We can write a <u>make file</u> that defines the build-tasks for compiler and linker.
  - => In the end it is <u>recommended to use an IDE</u> to define <u>this stuff in a project.</u>

- Ok! What does a project within an IDE do for us?
  - <u>In the project we'll just collect all the h/c-files (and other files) we require.</u>
  - <u>The IDE project will be automatically updated, when we add/remove h/c-files.</u>
  - <u>Compiler and linker will be automatically prepared to deal with all the project's files.</u>

- <u>What other kinds of files could be managed within a IDE project?</u>
  - E.g. resource files like icons.

Documentation of Functions

- The separation of function declaration and definition leads to information loss.
  - In some sense this was the aim (callers should not see the definitions).
  - But the bare function declaration is very opaque to us. How to improve the situation?

- To make functions really reusable, it is required to document how these functions work, esp.:
  - What does the function require to work properly: arguments and preconditions.
  - What does the function deliver after the work was done: returned values and postconditions.

- Basically there are two things that help to document something about a function:
  - (1) The most important one: Name the function as well as its parameters in a meaningful/self-explanatory way!
  - (2) Provide a prosaic documentation of what the function does.

- Let's think about the function *factorial()*:
  - The name of the function as well as the name of the parameter (em, just *n*), is already meaningful enough.

```cpp
// <numeric.h>
namespace Nico {
    int factorial(int n);
}
```

  - But, do you remember that the passed argument needs to be a positive number? (n needs to be a natural number)
  - => We should communicate this precondition to potential users of our function!

- The C++ standard doesn't define a kind of formal syntactic support for documentation comments.
    - There exist some conventions, e.g. HeaderDoc. However, the HeaderDoc convention is understood by the Xcode IDE.

- So as programmers we write comments with special HeaderDoc-style markup to a function's declaration:

```
namespace Nico {
    /**
     * Calculates the factorial.
     *
     * @param n the positive integral number for which
     * we want to get the factorial
     * @return the factorial
     */
    int factorial(int n);
}
```

```
int main() {
    Nico::f
}          ƒ  int factorial(int n)
           Calculates the factorial.
```

```
                    Nico::factorial(4);

Summary
Calculates the factorial.

Declaration
  int factorial(int n)

Parameters
  n   the positive integral number for which we want to get the factorial

Returns
the factorial

Declared In
numeric.h
```

Mind
The function, I wrote might also be used by other developers, therefore it must be documented to support the DRY principle!

    - HeaderDoc comments are fringed with /** */ (not /* */) and allow the usage of @-prefixed tags for special documentation aspects.
        - The first lines of the comment can be used for a free prosaic text "Calculates the factorial.". The following lines can be used for the tags.
        - Important HeaderDoc tags for functions: @param, @return, @see and @code/@endcode.
    - The result of these comments is available in the IDE directly, so we have inline documentation during development in the IDE.
    - It also possible to generate a set of HTML pages from HeaderDoc-style commented h-files to get an "IDE-offline" documentation.

                                                                                        80
- Writing code and documentation in the same sources is called "literate programming" after Donald E. Knuth.

- Earlier we learned, that C++ allows to leave the parameter names away from a function declaration. But here we see, that it makes sense to set parameter names, because then we can exactly document each parameter.
- We should avoid adding too many tags and embedded markup to make the generated documentation look awesome. Why? Well, because the comment itself should remain readable as well!
- In the example it is shown, how Xcode shows the documentation of a function from the HeaderDoc comment. This view can be triggered by hitting option-click on macOS.

Thank you!