

(3) Basics of the C++ Programming Language

Nico Ludwig (@ersatzteilchen)

TOC

- (2) Basics of the C++ Programming Language
 - Formatted Output and Input
 - Operators, Precedence, Associativity and Evaluation Order
 - Control Structures for Iteration
 - Forced and unconditional Branching
 - Structured Programming
- Cited Literature:
 - Bruce Eckel, Thinking in C++ Vol I
 - Bjarne Stroustrup, The C++ Programming Language

Initial Words

Yes, my slides are heavy.

I do so, because I want people to go through the slides at their own pace w/o having to watch an accompanying video.

On each slide you'll find the crucial information. In the notes to each slide you'll find more details and related information, which would be part of the talk I gave.

Have fun!

Working with the Console

- All the examples in this course will be console applications.
- Console applications make use of the OS' command line interface.
- To program console applications, we need basic knowledge about the console.
 - Esp. following commands are usually required:

Action	Windows	Unix-like/macOS
Change directory	cd <dirName>	cd <dirName>
Change directory to parent	cd ..	cd ..
List current directory	dir	ls -l
Execute an application	<appName>.exe	./<appName>
Execute an application with three arguments	<appName>.exe a b "x t"	./<appName> a b "x t"

Formatted Output to Console (STL)

- Neither user-input nor user-output are part of C++' language-core, instead this functionality is in a dedicated library.

Good to know

When we talk about console output, the term `stdout` will be used quite often.

- Writing messages to the console is a very important means to communicate:

- We have to use C++' `iostream` library to write to the console, therefore we must `#include` the h-file `<iostream>`.
- Then the output stream `std::cout` can be used to output values to the console.

- The `<<`-operator can be used in a chained manner to output multiple values:

```
std::cout<<"Hello there, You know about "<<42<<"?";  
// >Hello there, You know about 42?
```

- The output streams are buffered, the buffer must be flushed to console:

```
std::cout<<"Hello there, You know about "<<42<<"?"<<std::endl;  
// >Hello there, You know about 42?
```

- To flush the stream, chain `std::flush` or `std::endl` to the output expression.

- Output can be formatted with manipulators (e.g. `std::boolalpha`, `std::hex`), which can be chained as well:

```
// Use std::boolalpha to output boolean values as literal text:  
std::cout<<std::boolalpha<<(23 < 78)<<std::endl;  
// >true
```

- `#include` the h-file `<iomanip>` to use manipulators accepting arguments (e.g. `std::setprecision()`):

```
// Use std::fixed and std::setprecision() to output 2-digit floating point value:  
std::cout<<"The result is "<<std::fixed<<std::setprecision(2)<<2.666<<std::endl;  
// >The result is 2.67
```

5

- If `std::cout` (STDOUT) is buffered or not depends on the console on which it is attached to (read: it is platform dependent). On most OS' the console is buffered.
- There also exists `std::cerr`, which also writes to the console (STDERR), but this one is always unbuffered. Writing to `std::cerr` is like writing to another "channel": if the standard output of a program is e.g. redirected to a file, important messages are still visible, if they are output to `std::cerr`.
- `std::cin` does also flush the output buffer, when it accepts input.
- We'll talk a little about formatting **doubles** on the following slides.

Formatted Output to Console (STL) – general Format

- When we output a large enough `double` to a stream, it'll be formatted according the default precision 6:

```
double value = 2234567890;
std::cout<<value<<std::endl;
// >2.23457e+09
```

- The general format tries to express the value as 2.23457×10^9 , the precision of the format is 6, i.e. the first 6 positions are precise.
- This notation is called the scientific notation, it allows to express very small and very large `double` values in a compact manner.
- The scientific notation expresses values as $m \times 10^n$, where m is called mantissa and n is called exponent.
- But as you see $2.23457 \times 10^9 = 2,234,570,000$ is not exact compared to the input, therefor we can set a higher precision:

```
std::cout<<std::setprecision(10)<<value<<std::endl;
// >2234567890
```

- The precision 10 yield all decimal places.
- => Takeaway (general format): the precision of a floaty value is effective on all decimal places left and right from the decimal point!

- The general formatting of floaty values has some internal logic to switch between fixed point and scientific format:

```
double value = 2234567.890;
// When a double is just sent to a stream, it'll be
// set in the general format with the precision 6:
std::cout<<value<<std::endl;
// >2.23457e+06
```

```
double value = 22345.67
// When a double is just sent to a stream, it'll be
// set in the general format with the precision 6:
std::cout<<value<<std::endl;
// >22345.7
```

- The general format selects the shortest format:
 - it regards the value in scientific representation with only one decimal place left from the decimal point,
 - if in this representation the exponent is greater than -4 or greater than the precision, the scientific format is used, else the fixed point notation.

6

- All C++' floaty formatting functions support some rounding (which can be controlled with `std::setprecision()`, it has no effect on integral types). The rounding behavior (round-to-nearest, round-to-nearest-odd/even etc.) depends on the platform and how the compiler does represent floating point values.
- The general format has "interesting" automatisms, which are not always reasonable. Therefore, C++ provides further formatting capabilities: the scientific format and the fixed point format.

Formatted Output to Console (STL) – scientific and fixed Format

- We can force output streams to use the scientific format for floaty values, therefore we use the manipulator `std::scientific`:

```
double value = 2.23456789;  
std::cout<<std::scientific<<value<<std::endl;  
// >2.234568e+00
```

- The manipulator `std::precision()` let's us specify the count of digits right from the decimal point (the default of 6 leads to rounding):

```
std::cout<<std::setprecision(2)<<value<<std::endl;  
// >2.23e+00
```

- We can force output streams to use a notation with a decimal point at a fixed position with the manipulator `std::fixed`:

```
std::cout<<std::fixed<<value<<std::endl;  
// >2.234568
```

- The manipulator `std::precision()` let's us specify the count of digits right from the decimal point (the default of 6 leads to rounding):

```
std::cout<<std::setprecision(8)<<std::fixed<<value<<std::endl;  
// >2.23456789
```

- And we can also use `std::precision()` to round the value down:

```
double value2 = 2.55555555;  
std::cout<<std::setprecision(2)<<std::fixed<<value2<<std::endl;  
// >2.56
```

The manipulators we have dealt with, will be applied on the stream until the program lifetime ends!

- Therefore we have to reset the output stream explicitly, if we want to go back to the default settings (i.e. the general format):

```
std::cout<<std::resetiosflags(ios_base::floatfield)<<std::setprecision(6);  
std::cout<<value1<<std::endl;  
// >2.55556
```

- ... hence we are back at general format and a precision of 6 (all decimal places, see the output result).

7

- The last slides only showed a few aspects of the possibilities of streams.
- Manipulators are effective, until they are reset or overridden by other "competing" manipulators (e.g. `std::hex` and `std::dec` that override each other). The effectiveness is also spanning multiple statements, when used on the same stream object!

Formatted Input from Console (STL)

- We will use the input stream `std::cin` and the special `>>`-operator (`<iostream>`) to read data from the console.
 - Before a program requires input from the user, it should prompt with an output!
 - Then the input operation blocks program execution and forces user interaction.
 - (In this lecture we'll only discuss inputting fundamental types, in an upcoming lecture we'll discuss textual data).

Good to know

Prompt from latin *promptare*: "to publish something".

- The `>>`-operator can also be used in a chained manner to read multiple values.

```
const int length = 256;
char yourName[length];
int yourAge = 0;
std::cout<<"Please enter your name (max. "<<length - 1<<" characters!) and your age:"<<std::endl; // prompt
std::cin>>yourName>>yourAge; // program execution blocked, until data is entered
std::cout<<"Hello "<<yourName<<" , aged"<<yourAge<<std::endl;
```

Good to know

When we talk about console input, the term `stdin` will be used quite often.

Good to know

When a program prints to the output, what a user has entered, this is called echo. I.e. these programs echo the entered name and age to the console.

- Esp. `cstrings` can also be read with `std::cin`'s member function `getline()`.

```
const int length = 256;
char yourName[length];
std::cout<<"Please enter your name (max. "<<length - 1<<" characters!)"<<std::endl; // prompt
std::cin.getline(yourName, length);
std::cout<<"Hello "<<yourName<<"!"<<std::endl;
```

- `std::cin.getline()` awaits the `cstring` to be filled and `length - 1`, which should be read from the console at maximum.
- If less than `length - 1` are read, the entered text will be stored in `yourName` and 0-terminated.
- If more than `length - 1` are read, the content of `yourName` is undefined, and `std::cin`'s failed bit will be set.

Checking User Input from Console – Part I

- Let's inspect following snippet, when we run it in a program it looks like so on the console:

```
int yourAge;  
std::cin>>yourAge;
```

- Hm ... the program pauses execution until input has ended and enter has been hit.
- The point is: how should the user know, that she should enter an `int`? – Maybe the program did even stop working?

- When the user is meant to input data for processing, the program should prompt!

- Therefore we just add an output, which explains what the user should input:

```
std::cout<<"Please enter your age:"<<std::endl; // prompt  
int yourAge;  
std::cin>>yourAge;
```

```
Terminal  
NicosMBP:src nico$ ./main  
NicosMBP:src nico$ Please enter your age:  
█
```

- So far so good. There is another problem with input validity. E.g. in this case, the entered age must be greater or equal 0.

- With conditional code we can handle this situation easily:

```
std::cout<<"Please enter your age:"<<std::endl;  
int yourAge;  
std::cin>>yourAge;  
if (yourAge < 0) { // Check the input value to meet our expectations.  
    std::cout<<"Your age mustn't be less than 0!"<<std::endl; // another output  
}
```

```
Terminal  
NicosMBP:src nico$ ./main  
NicosMBP:src nico$ Please enter your age:  
-38  
Your age mustn't be less than 0!  
NicosMBP:src nico$ █
```

- Checking user input is very important! Usually we cannot trust user input! It's the most probable source of bugs in programming!
- The example is not perfect, the program should be able to repeat the prompt, until the user's input is valid. We'll handle this soon!

- Formally, we can tell the application prompt from the command prompt. The command prompt is the prompt of the system console or terminal. On most OSes, the command prompt has a special syntax, e.g. the name or path of the current directory and some special character (\$, # or >), which precedes the cursor ready for input.

Checking User Input from Console – Part II

- However, there is still a potential source of problems: What if the user enters a text, although a number ([int](#)) is awaited?

```
std::cout<<"Please enter your age:"<<std::endl;
int yourAge;
std::cin>>yourAge;
std::cout<<"Your age: "<<yourAge<<std::endl;
```

```
Terminal
NicosMBP:src nico$ ./main
Please enter your age:
Nico
Your age: 0
NicosMBP:src nico$
```

- Obviously ... the textual input "Nico" was not regarded an [int](#) and *yourAge* was set to 0 (in this case).
- But, it would be better to get a message from the program, that textual input is not accepted here!

- Alas, this is not so simple in C++, because of the complexity of the i/o stream libraries.

- After an input stream has read unexpected data the stream stays in a failed state.

- The stream is no longer usable, after it failed, e.g. because it read a text, where it expected an [int](#):

```
std::cout<<"Please enter your age:"<<std::endl;
int yourAge;
std::cin>>yourAge;
std::cout<<"Your age: "<<yourAge<<std::endl;
std::cin>>yourAge; // try to reuse the failed stream
std::cout<<"Your age: "<<yourAge<<std::endl;
```

```
Terminal
NicosMBP:src nico$ ./main
Please enter your age:
Nico
Your age: 0
Your age: 0
NicosMBP:src nico$
```

- As can be seen, the try to reuse the failed stream fails as well, the 2nd call to `std::cin>>yourAge` is just ignored.

10

- If the failed bit is set, the next read operation on that stream will fail.
- If the stream is not reset and the buffer is not cleared the input stream is rendered unusable and will not accept inputs anymore.

Checking User Input from Console – Part III

- Back to the bare problem, what we have to do is to check the stream for failed state and tell the user about the situation.

- After reading, input streams should generally be failure-checked.

```
std::cout<<"Please enter your age:"<<std::endl;
int yourAge;
std::cin>>yourAge;
if (std::cin.fail()) { // E.g.: std::cin awaited an int, but somewhat different was read:
    std::cout<<"The entered age was invalid!"<<std::endl;
} else { // All fine:
    std::cout<<"Your age: "<<yourAge<<std::endl;
}
```

Terminal

```
NicosMBP:src nico$ ./main
Please enter your age:
Nico
The entered age was invalid!
NicosMBP:src nico$
```

- The most important new aspect is the failure-check of the input stream.
 - On failure, `std::cin.fail()` evaluates to true, we'll branch into a block, that will tell the user, that the entered age was invalid.
- We still have the problem, that we cannot reuse the input stream after it failed – the user gets no second chance!
 - To make the failed stream usable again, we have to reset the failed status of the stream, let's see how this works.
- But, it should be said, that the user at least gets an information now, that something went wrong, this is quite a progress!

Checking User Input from Console – Part IV

- When an input stream is in failure state, we have to reset it into a good state and clear the wrong data from its buffer:

```
std::cout<<"Please enter your age:"<<std::endl;
int yourAge;
std::cin>>yourAge; // (0) Try accepting valid input for the first time.
if (std::cin.fail()) { // E.g.: std::cin awaited an int, but somewhat different was read:
    std::cout<<"Please enter a valid age!"<<std::endl; // (1) Prompt anew.
    std::cin.clear(); // (2) Clear stream status, esp. reset the fail status.
    std::cin.ignore(std::numeric_limits<int>::max(), '\n'); // (3) Clear buffer, esp. remove the pending newline.
    std::cin>>yourAge; // (4) Try accepting valid input for the second time.
    std::cout<<"Your age: "<<yourAge<<std::endl;
} else { // All fine:
    std::cout<<"Your age: "<<yourAge<<std::endl;
}
```

```
Terminal
NicosMBP:src nico$ java Program
Please enter your age:
Nico
Please enter your age:
39
Your age: 39
NicosMBP:src nico$
```

- In case of `std::cin.fail()`:
 - (1) Prompt the user for a correct age, it's a second chance!
 - (2) Clear the failed bit (`std::cin.clear()`).
 - (3) Clear the unread buffer that remained after the failure (`std::cin.ignore()`).
 - The input stream's buffer is clogged with unread letters, sometimes called characters. The characters in `std::cin` are of type char!
 - The call clears the specified number of characters from the buffer or clears all characters from the buffer until the termination character is reached.
 - With `std::numeric_limits<int>::max()` (`<limits>`) and `'\n'`: an unlimited count of characters is cleared until `'\n'` is found.
 - (4) Try accepting valid input for the second time.

12

- The solution is still far from optimal: the user gets only two chances to enter a correct age! – We'll address this soon!

This is an important Lesson: Usually we cannot trust user input – Part I

- When a user has to enter data, there is a certain probability that the input data is wrong! Therefore we should:
 - (1) write programs, which prompt an output, before it requires input from the user.
 - The text of the prompt should explain use user in simple words, which input and in which format it is required.
 - (Then the following input operation blocks program execution and forces user interaction.)
 - (2) then check the user's input for correctness.
 - Forgetting to check user input is a major source of bugs!
- Usually, programs should behave highly interactive and give the user a chance to:
 - (1) Enter data, until it is valid,
 - (2) or, allow the user to quit data input.
 - ((3) And maybe echo the entered data and request further confirmation from the user.)
 - But to implement programs like this, we have to discuss more means of program control flow.
- So: good quality programs have to prompt for user input and check user input.
 - And may optionally echo user input.
- The discipline to check input before continuing in the program is called defensive programming.
 - Defensive programming is a cornerstone of high quality programs!

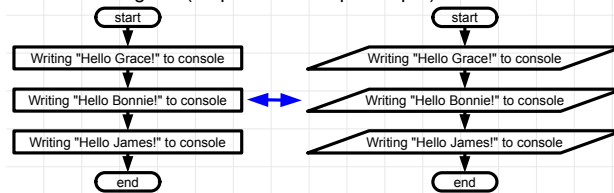
This is an important Lesson: Usually we cannot trust user input – Part II

- Usually checking input is done on multiple levels, usually we tell apart basic checks from checking business rules.
 - A basic check would be "Did the user enter a number?".
 - A business rule would be "The user should enter the age, which is a positive number. – Did the user enter a positive number?"
 - Business rules could also apply blocklists and allowlists to check inputs.
 - If input passes basic checks and business rules checks, the input is said to be plausible.

Input and Output in a Flowchart Diagram

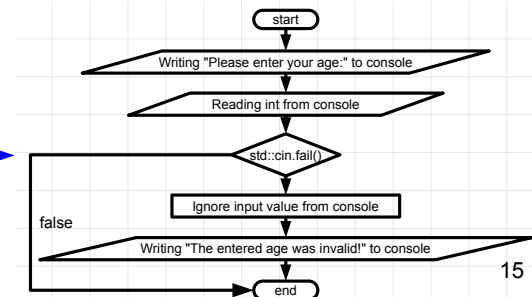
- We've special parallelogram-shaped flowchart symbols for sequence items, which deal with input and output.

Flowchart Diagram (sequence and input/output)



- So we can redesign our last example with this new Flowchart element:

```
std::cout<<"Please enter your age:"<<std::endl;
int yourAge;
std::cin>>yourAge;
if (std::cin.fail()) // E.g.: std::cin awaited an int, but different data was read:
{
    std::cout<<"The entered age was invalid!"<<std::endl;
}
```



Operators – Part I – Mathematical Operators

- The concept of arithmetic operators is well known from mathematics: "+", "-", "*" (multiplication), "/" (division).

`int difference = 5 - 3;`

Good to know

The operator symbol '-' is sometimes called dash or hyphen, although those characters are different.

- Remember, what we said about integral division:

- If in a division operation only `ints` are involved, the division will yield an `int` result and no fractional number!

`double result = 2 / 3;`

`result (0.0)`

- Actually, `result` evaluates to 0.0!

- To correct the expression, so that the division yields a fractional result, at least one of the operands needs to be of floaty type.

- (1) If the division uses `int` literals one of the literals can be just be written as, e.g., `double` literal:

`double result = 2.0 / 3;`

`result (0.6666666666666666)`

- (2) One of the contributed operands (an `int` literal, `int` variable or other `int` expression) can be cast to a `double`:

`double result = static_cast<double>(2) / 3;`

`result (0.6666666666666666)`

- (1) and (2) will evaluate to `result` having the value 0.6.

- Notice: the result of an expression right from assignment is of the type of the largest and floatiest operands' type.

- We already discussed, that the result of the division by 0 has an undefined result.

16

- What does that mean "integral division has an integer result"?
 - I.e. their results are no floating point values!
- The division by 0 is not "not allowed" in maths, instead it is "just" undefined.

Operators – Part II – Mathematical Operators

- "-" and "+" can be used as so called "unary" operators, to mark their operand as negative or positive value:

```
double amount = 4;  
double negativeAmount = -(amount); // makes negativeAmount the negative value of amount
```

negativeValue (-4.0)

- C++ provides support for division with remainder with the %-operator, which is called modulo operator.

- "%" calculates the symmetric modulus (in difference to the mathematical modulus) of its operands.

```
int result1 = 5 % 2;
```

result1 (1)

```
int result2 = -5 % 2;
```

result2 (-1)

- Remainder calculations are very handy, e.g. to program loops doing something at every xth loop.
- % doesn't work for floaty values, instead the function std::modf() must be used, it is declared in <cmath>.

- Special functions, like exponentiation, logarithm and trigonometric functions are not represented by operators in C++.

- Basically all important functions can be found as C++ functions in <cmath>. (We already saw the constant M_PI from <cmath>.)

```
double result1 = std::pow(10, 2);
```

result1 ($10^2 = 100$)

```
double result2 = std::log10(100);
```

result2 ($\log_{10} 100 = 2$)

- The arguments of trigonometric functions need to be passed in the unit radian (rad) or degrees (°) of an arc!

- $360^\circ = 2\pi \text{ rad} \Rightarrow 180^\circ = \pi \text{ rad} \Rightarrow 1 \text{ rad} = 180^\circ/\pi \Rightarrow 90^\circ = \pi/2 \text{ rad}$

```
double result3 = std::sin(M_PI / 2);
```

result3 ($\sin(\pi/2) = 1$)

- <cmath> acts as a library of math-related methods.

Operators – Part III – Operator Notation – Arity

- Binary operators (+, -, *, /, %, ==, <, >, &, |, &&, || etc.) have two operands:

```
// Addition as binary operator:  
int sum = 2 + 3;
```

```
sum (5)
```

- Usually, binary operators are written with whitespaces around operands and operator.
- The operands of binary operators are colloquially called left hand side (lhs), i.e. 2, and right hand side (rhs), i.e. 3.

- Unary operators (-, +, ++, --, ! etc.) have one operand:

```
// Minus as unary operator:  
int i = 4;  
int j = -i; // makes j the negative value of i
```

```
j (-4)
```

- Usually, unary operators are written without whitespaces between operand and operator.

- The ternary operator has three operands:

```
// The conditional operator is the only ternary operator:  
int i = 2;  
int j = 3;  
const char* answer = (i < j) ? "i less than j" : "i not less than j";
```

```
answer ("i less than j")
```

- There are no specific conventions on the notation.
 - It is recommendable (not mandatory) to write the condition in parentheses.

Good to know

The ternary operator ?: is sometimes called Elvis-operator, because it resembles Elvis Presley's iconic pompadour haircut. (In a narrower sense, ?: is called Elvis-operator, if it is used as binary operator, a notation, which is not supported in C++17.)

Operators – Part IV – Operator Notation – Placement

- Prefix operators are written in front of its only argument:

```
// Negation as prefix operator:  
int negatedValue = -value;
```

- This notation is also called (forward) polish notation, or PN.

- Postfix operators are written after its only argument:

```
// Increment as postfix operator:  
int result = item++;
```

- This notation is also called reversed polish notation, or RPN.

- Infix operators are written in between its arguments:

```
// Addition as infix operator:  
int sum = 2 + 3;
```

- The []-operator (brackets) awaits its only argument in between the brackets.

- This operator is used to access array elements. We'll discuss arrays in a future lecture.

- Additionally, parentheses (parens), i.e. (), also await their only argument in between the parentheses.

- Parens are used to structure expressions and force prioritized evaluation.

Good to know

The polish notation was developed by the Polish mathematician Jan Łukasiewicz because, he could formulate his mathematical theories in a better manner.

Originally, the notation was called Łukasiewicz notation, but nobody could pronounce or write his name correctly, so it was called polish notation.

Generally, polish notation is the name of all notations different from infix (and parens-) notation.

There exist programming languages, which only apply PN, e.g., Lisp.

Some pocket calculators (e.g. the legendary Hewlett-Packard HP-12C) use RPN, instead of algebraic notation. The advantage of RPN is, that neither operator priorities nor parentheses required.

Operators – Part V – Operator Notation – Placement (other Languages)

- Esp. mathematical operations are usually expressed as a set of operators and operands.
- The interesting point is, that programming languages in the wild use all thinkable operator placements.

- Postfix operator placement/RPN means, that a list of operands is written in front of an operator.

- Assume the addition of two integers in the programming language PostScript:

```
% PostScript - Addition:  
/sum 2 3 add def
```

- *sum* => variable name, 2 and 3 => operands, add => operator, *def* => keyword (define variable)

- An iconic postfix operator in C++ is the postfix increment:

```
// C++ - Postfix increment:  
i++;
```

- Prefix operator placement/PN means, that an operator is written in front of a list of operators.

- PN is used for mathematical functions like *f(5)* or *g(8, 2.59)* or predefined mathematical operations like *cos(90)*, *sin(18)* and *tan(35)*.
 - Assume the addition of two integers in the programming language Lisp:

```
; Lisp - Addition and init of variable sum:  
(let ((sum (+ 2 3))))
```

- This is C++' notation for function calls like calling *std::pow()*:

```
// C++ - function call:  
double result = std::pow(2, 3);
```

Operators – Part VI – Operator Notation – Placement (other Languages)

- The advantages of RPN/PN:
 - Flexibility: In many languages and situations multiple or lists of operands can be specified
 - Another advantage of RPN: neither operator priorities nor parentheses required.
 - E.g. C++ allows defining functions, which accept a list of operands (also called function arguments).
- Infix operator placement means, that an operator is written in between two operators.
 - This notation is used for all the elementary mathematical operations like addition or multiplication.
 - But, it is the most limited notation: Infix notation is only available with binary operations, i.e. with exactly two operands.

```
// C++ - Addition:  
int result = 2 + 3;
```

- In reality, we'll mainly find combinations off all operator placement variants to write meaningful code:

```
// Java - Addition and prefix method call:  
double result = Math.pow(7, 2 + 3);
```

- There exist three notations, prefix, infix and postfix and parentheses (parentheses, brackets and angle brackets).

Operators – Part VII – Logical Operators

- When we discussed conditional code, we encountered snippets similar to this:

```
int age = 90;
int countOfGreetingsForBonnie = 0;
// Using cascaded if statements:
if (age > 80) {
    if (countOfGreetingsForBonnie <= 0) {
        std::cout<<"Hello Bonnie!"<<std::endl;
    }
}
```

- The condition, under which "Hello Bonnie" is written to the console can be summarized to:
 - "Hello Bonnie" is written to the console, if *age* is greater than 80 and *countOfGreetingsForBonnie* is less than or equal to 0.
 - These two cascaded if statements do reflect a logical and-combination of two conditions, i.e. bool expressions.

- Logical operators can be applied in bool expressions for control structures (conditional code and loops).

- In C++ we can use following logical operators: && (logical "and"), || (logical "or"), ! (logical "not").
- Using && we can reformulate and condense the code with the cascaded if statements shown above to only one if statement:

```
int age = 90;
int countOfGreetingsForBonnie = 0;
// Using logical and (&&):
if (age > 80 && countOfGreetingsForBonnie <= 0) {
    std::cout<<"Hello Bonnie!"<<std::endl;
}
```

- C++' logical operators evaluate to bool results, which are also integral values in C++.

Operators – Part VIII – Logical Operators – Comparison

- Besides logical operators, we have already used C++' comparison operators to compare values:

- Comparison operators: `==`, `!=`, `<`, `>`, `<=`, `>=`

- `"=`" and `"=="` are different operators!

- `"=`" for assignment and `"=="` for equality comparison: i.e. `"=="` evaluates to a bool value!

```
if(a == 0){ // Is the value of a zero?
    // pass
}
```

- Using `"=`" instead of `"=="` for equality comparison will not end in a compiler error:

```
// A typical beginner's error: the "optical illusion":
if(a = 0){ // (1) Maybe wrong! Oops! a = 0 was meant!
    // pass
}
```

Good to know

In maths, the expression $a = 0$ can mean "0 is assigned to a ", or "it is asserted, that a is 0". But maths also provides dedicated notations to distinguish both (very different) meanings:

$a := 0$ " a is defined and set to the value 0"
-> C++ assignment
 $a = 0$ "it is asserted, that a is 0"
-> C++ equality comparison

- We cannot use these operators to compare textual data (cstrings)! For cstrings we must use the function `std::strcmp()`.

- `std::strcmp()` returns 0, if the compared cstrings are equal (i.e. have the same content):

```
// "Incorrect" way to compare two cstrings:
const char* herName = "Gwen";
if (herName == "Gwen"){
    // pass
}
```

```
// "Correct" way to compare two cstrings:
const char* herName = "Gwen";
if (std::strcmp(herName, "Gwen") == 0){
    // pass
}
```

- The crux: using `"=="` to compare cstrings won't issue in a compile time error, but potentially yield wrong results at run time!23

- When we discuss pointers and cstrings, we'll learn, that using `==` would probably compare the addresses of the cstrings and not their contents.

- JavaScript additionally defines the comparison operator `===`. It performs a strict comparison of values: it evaluates to true, if value and type of lhs and rhs are equal, else it evaluates to false. (The comparison operator `==` is less strict, the type of the value must not match.) E.g. `1 === 1` will evaluate to true, whereas `"1" === 1` will be evaluated to false.

Operators – Part IX – Logical Operators – Negation

- In C++ there exist two negation operators "!" and "!=", which are used to express inequality in bool expressions:

```
// Using "!=" for inequality comparison:  
if (count != 0) { // If count is unequal 0:  
    // pass  
}
```

Good to know

The ! (exclamation mark or exclamation point) is sometimes also called "bang" or "shriek" among developers.

- We can also formulate this comparison with the unary negation operator.

```
// Using "!" for inequality comparison:  
if (!(count == 0)) { // If count is unequal 0:  
    // pass  
}
```

- This is an example where we need parens: we've to put the comparison expression in parens to make ! effective on it as a whole.
 - Using ! In this case states "if count == 0 is not true" ...
- We can also not use ! to compare cstrings, instead we use the function std::strcmp().
 - std::strcmp() returns a value different from 0, if the compared cstrings are not equal (i.e. have not the same content):

```
// "Correct" way to compare two cstrings for inequality:  
const char* hisName = "Peter";  
if (std::strcmp(hisName, "Gwen") != 0) {  
    // pass  
}
```


Operators – Part X – Logical Operators – Short-Circuit Evaluation

- The binary logical operators "&&" and "||" support short-circuit evaluation to improve performance:

- Let's think about following snippet:

```
int age = 70;
int countOfGreetingsForBonnie = 0;
if (age > 80 && countOfGreetingsForBonnie <= 0) {
    std::cout<<"Hello Bonnie!"<<std::endl;
}
```

- Short-circuiting: if the lhs of "&&" evaluates to false, the rhs needs not to be evaluated at all, because the overall result is false!
 - age > 80 evaluates to false, countOfGreetingsForBonnie <= 0 needs not to be evaluated, because the result of "&&" must be false.
 - "&&": If the lhs evaluates to false, the rhs need not to be evaluated, because the result of the whole expression must be false!
 - "&&": If the lhs evaluates to true, the evaluated result of rhs is the result of the whole expression!
- For the operator "||" there exists a similar short-circuit evaluation rule:
 - "||": If the lhs evaluates to true, the rhs needs not to be evaluated, because the result of the whole expression must be true!
 - "||": If the lhs evaluates to false, the evaluated result of rhs is the result of the whole expression!
- The idea is to place a more expensive operation into a short-circuit expression, so that it might not necessarily be evaluated!
 - => The more expensive operation could be the rhs for "&&" and "||". It depends on the probability of the lhs leading to short-circuit.

Operators – Part XI – Summary of Logical Operators

- Used to compare values and combine boolean results.

- Comparison: `==`, `!=` (`not_eq`), `<`, `>`, `<=`, `>=`
- Combination: `&&` (`and`), `||` (`or`), `!` (`not`)
- Logical operators return boolean results, which are also integral results in C/C++.

- `&&` (logical and) and `||` (logical or) support short circuit evaluation.

```
// The mathematic boolean expression a = b and c = b:  
if (a == b && c == b) // Ok  
{  
    // pass  
}  
if (a && b == c) // Wrong!  
{  
    // pass  
}
```

- Logical operators are applied in conditional expressions for control structures (branches and loops).

```
// A typical beginner's error: the "optical illusion":  
if (a = 0) // (1) Oops! a == 0 was meant,  
{ // pass // but this is ok for the compiler! It evaluates to false  
    // (0) always, because 0 will be assigned to a!  
}  
if (0 == a) // Better! Tip: write the constant left from the  
{ // pass // equality comparison. -> Yoda Condition style  
}  
if (0 = a) // Invalid! Here (1) could have been caught, as we  
{ // pass // can't assign a constant.
```

Good to know

The style to write conditions with a constant left from the equality comparison is sometimes called "Yoda Condition" or "Yoda Coding" after Yoda's iconic way to apply grammar.

```
// Sometimes assigning and checking values are performed intentionally:  
if (a = b) // Intentional assignment and evaluation.  
{ // pass // On modern compilers this code yields a warning!  
}  
// The warning can be eliminated by writing an extra pair of parentheses:  
if ((a = b)) // Still not a good idea, but the syntax underscores  
{ // pass // the intention in a better way.  
}
```

26

- On many C/C++ compilers the shown "optical illusion" leads to a warning message. Some compilers (e.g. gcc) accept "marking" the assignment to be meant as condition by writing them into an extra pair of parentheses: `if ((a = 0)) { /* pass */ }`.
- In Groovy this way of marking is required.
- In Swift assignment expressions don't return a value at all. With this simple rule a Swift expression like `if a = 0 { /* pass */ }` is a syntax error!

Operators – Part XII – Bit-Wise Operators

- C++ bit-wise operators have no direct representation in mathematics.
 - Those operators work on integral values and operate on their bit representation, i.e. the bit-pattern in computer memory directly.
 - Bit-wise operators apply logical operations on each bit of a value. – There is only an indirect relation to logical operations.

- E.g. let's quickly talk about the bit-and operator, which is expressed with the symbol "&", or the keyword **bitand**:

- (1) Bit-and combines each bit-value of each bit of the two operands with a logical **and** operation.
- (2) Then the resulting bit-pattern is interpreted as integral value.
- => Logical operators (e.g. && and ||) have a boolean result, bit-wise operators (& and |) have integral bit-pattern results

```
unsigned x = 65535;
unsigned y = x & 255;
std::cout<<y<<std::endl;
// >255
```

	1111 1111	1111 1111	x (65535)
&	0000 0000	1111 1111	mask (255)
	0000 0000	1111 1111	y (255)

Good to know

The symbol '&' is usually called ampersand. 'Ampersand' is a corrupted form of the phrase "and per se and", which was used in past to name the '&' itself as specific symbol, e.g. in an enumeration.

- Bit-wise operations work very hardware-near, this is the reason, we use **unsigned** here, we'll discuss this in a future lecture.
- Besides bit-and, there exist some other bit-wise operators: ^ (**xor**), | (**bitor**), ~ (**compl**), << and >>.

Good to know

Usually developers call the operator | ("vertical bar") "pipe". Some fonts represent the pipe with the symbol | ("broken bar" or "parted rule"). Often developers call the operators <, >, << and >> "chevrons", angle brackets or pointy brackets.

27

- Why do we need bit-wise operators?
- Bit shifting operations ($x \ll n$ and $x \gg n$):
 - Shifts all bits to left or right by n positions, the "new" bits will be filled with 0s.
 - A single left shift is a multiplication by 2; a single right shift is a division by 2.

Operators – Part XIII – Precedence

- Let's discuss this snippet:

```
int result = 3 + 4 * 6;  
// result = 27
```

- However, how the result is effectively calculated, if many operators are involved is not obvious!
 - Esp. because we could have mixed expressions with prefix-, postfix- and infix-notated operators, the language must define some rules.
- In this case the multiplication $4 * 6$ is evaluated to 24 before the addition of 3 takes place.
 - We imply operator priority as we know it from maths (PEDMAS), i.e. the "execution order" is independent from their written order in the expression.
- Because C++ has more operators than we know from basic maths, there are more rules if many operators are involved.

- The priority, in which operators are evaluated is called precedence.
 - To simplify matters, operators are grouped into 17 precedence groups respectively.
 - In a specific group all operators have the same precedence.
 - The operators' precedences are mostly as we know from basic maths.

- Precedence is controllable with parentheses as in maths:

```
int result = (3 + 4) * 6;  
// result = 42
```

- If we want to cascade multiple parentheses, we must only use parentheses (round brackets) and no other form of bracket as we sometimes see in maths.

Precedence Group	Operators	Type
1	::	Scope resolution
2	. or -> [] ++	Member access Array subscript Postfix increment ...
...
5	* / %	Multiplication Division Modulus
6	+ - ...	Addition Subtraction ... 28

- The abbreviation **PEDMAS** helps remembering the mathematical rules of order in arithmetic evaluation: **P**arentheses, **E**xponents, **M**ultiplication and **D**ivision, **A**ddition and **S**ubtraction. In C++ exponents are not represented with an idiomatic operator, but as function.

Operators – Part XIV – Associativity

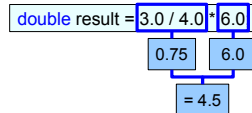
- Having clarified the term precedence, what's about the priority among operators of the same precedence group?

`double result = 3.0 / 4.0 * 6.0;`

- What is the result of this expression? Is it $(3.0 / 4.0) * 6.0 = 4.5$ or $3.0 / (4.0 * 6.0) = 0.125$?

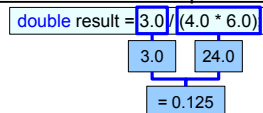
- Associativity defines the evaluation priority among expressions of the same precedence.

- The associativity of precedence groups defines the "direction", in which order expressions are evaluated.
- Because "/" and "*" reside in precedence group 5, we've to apply the associativity "left to right" to get the result:



- So, the result is 4.5!

- Associativity is also controllable with parentheses:



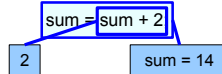
Precedence Group	Operators	Type	Associativity
1	::	Scope resolution	Left to Right
2	. or -> [] ++	Member access Array subscript Postfix increment ...	Right to Left
...
5	* / %	Multiplication Division Modulus	Left to Right
6	+ - ...	Addition Subtraction ...	Left to Right

Operators – Part XV – Combined Assignment

- In C++ we can add sum up a variable by a value very easily:

```
int sum = 12;  
sum = sum + 2; // Add and assign.
```

- The statement `sum = sum + 2;` looks weird, but by applying precedence/associativity, we understand what is going on here:



- `"="` has lower precedence than `+` and `"="` is right-associative, i.e. `"sum + 2"` is evaluated first, which yields the value 14.
 - Finally `"="` assigns the result 14 to `sum`, which overwrites the 12 formerly stored in `sum`.
- However, the syntax above has a little problem: the variable `sum` occurs twice:
 - (1) as summand in the addition operation, and
 - (2) as target for the assignment operation.
 - If this code is modified afterwards, one could exchange only one of two occurrences of `sum` and introduce an error.
- To make such operations more foolproof, C++ offers combined assignment, that combines arithmetic and assignment operators:

```
sum = sum + 2; → sum += 2; // combined assignment and addition
```

 - C++ provides combined assignment for most arithmetic (e.g. `-=`, `*=` ...) and bit-wise (e.g. `|=`, `&=` ...) operators.

Operators – Part XVI – Increment and Decrement

- Programming often requires to add 1 to the value of an integer.
 - We'll use this operation a lot, when we discuss loops to count things in a program.
 - The operation to add 1 to an integer is often called incrementation, subtracting 1 is called decrementation.
 - (There also exist terms like "incrementation of two", i.e. incrementation and decrementation are sometimes more general terms.)

```
// Increment as combined assignment and addition:  
int i = 1;  
i = i + 1; // increments i by 1  
// i = 2
```

- C++ provides an even more condensed syntax as `i = i + 1;` for incrementation/decrementation of an `int` by 1:

<code>i = i + 1;</code>	→	<code>++i;</code>
<code>i = i - 1;</code>	→	<code>--i;</code>

```
// Increment as increment operation:  
int i = 1;  
++i; // increments i by 1  
// i = 2
```

- We call the operators `++/--` increment/decrement operators.

Good to know

The name C++ was derived from the increment operator: C++ is (one) ahead of C or more than C.

Operators – Part XVII – Increment and Decrement

- Increment and decrement operator-expressions come in super handy, because they are very compact.
- However, there is a big difference to unary expressions we have discussed up to now: "++" and "--" have side effects. – What?
 - Well, we can just write `++i;` as a single statement, which internally modifies *i*, i.e. the assignment to *i* is hidden!
 - Think: `++i;` is like `i = i + 1;`! – It is not like `i + 1;`, because `++i` has a side effect!
 - If an expression or statement modifies variables, it is said to have a side effect on that variables.

```
i + 1;    // evaluates to the value of i + 1
++i;      // changes i to the value of i + 1
```

- The increment and decrement operators come in two syntactical and semantical flavors: prefix and postfix.

- As the wording implies, the syntax on how operator/operand are placed is different:

```
++i; // prefix increment
```

```
i++; // postfix increment
```

- The effective semantic difference lies in the value, which is the result of their expressions respectively:

```
int i = 1;
int result = ++i; // prefix increment
// i = 2
// result = 2
```

```
int i = 1;
int result = i++; // postfix increment
// i = 2
// result = 1
```

Advice: `++i` (pre-increment) should be your first choice, not `i++` (post-increment), as it is shown in many books and school exercises! Only use `i++`, if its result (the value before incrementation) is needed and this is rarely the case!

- Both expressions `++i` and `i++` have the same side effect on *i*, after the expression was evaluated respectively.
- But the expression results differ: `i++` yields the value before the incrementation, `++i` yields the value after the incrementation.
- Notice: the side effect of prefix and postfix increment/decrement is the same, but the yielded results are slightly different. 32

Operators – Part XVIII – Precedence, Associativity and Order of Evaluation

- Apart from precedence and associativity, there is still another aspect of operator execution: the order of evaluation.
 - Consider this snippet:

```
int i = 0;  
int result = (i = 2) * i++;
```

- In which order do the subexpressions in the expression "(i = 2) * i++" evaluate? What is the value of *result* and *i*?

// result = 0; i = 2// result = 4; i = 3
 - Virtually, ++ has a higher precedence than *, so the outcome should be result = 0 and i = 2.
 - But, the very dissatisfying answer is: we don't know the outcome, it could also be result = 4 and i = 3! – What?
- Actually, neither precedence nor associativity can answer the question, because i is read and modified in a single expression!
- The effect we see here is, that in C++ the order of evaluation in sub-expressions is not defined!
- Usually, this isn't a problem, but in the snippet above i is modified in each sub-expression of the full expression!
 - The sub-expression i = 2 could be evaluated either before or after the evaluation of i++!
- Lesson learned: don't write expressions, in which the same variable is read and written!
 - Actually, this tip can be formulated more differentiated, because there are exceptions to this rule.

33

- What is "order of execution"? Why is it relevant?
 - More exactly, the order of evaluation is undefined between so-called sequence points, which are positions in a complex expression, on which the results of previous expressions have been fully evaluated. With the expression above, g++ warns with the message "Multiple unsequenced modifications to 'i'" – The placement of those sequence points is standardized in C++:
 - Sequence points are set, when the operators &&, ||, , (i.e. the comma-operator) or ?: are used, then the subexpression of these operators are executed from left to right.
 - Sequence points are set after each statement (of course).
 - Sequence points are also set, after the evaluation of all arguments of a function call was performed.
 - In other words: all arguments of a function call must be evaluated before the evaluated arguments are passed and the function itself is executed.
 - The comma-operator has nothing to do with the comma-separator, that is used, when we pass arguments to a function! The order of evaluation of the arguments of a function call is also undefined in C++!
 - E.g. if we have an expression like *h(g(), f())* or *d() * s()* how can we know, which functions are being called first? This order of execution is undefined in C++! It is relevant to know that, because the function calls can have side effects! – This can be nasty, esp. when we use ctor calls.

Operators – Part XIX – Precedence, Associativity and Order of Evaluation

- How do precedence, associativity and order of evaluation work together?
- It's sufficient to keep this in mind: precedence takes effect first, then associativity:
 - Example: $a = b = c = 1 + 2 + 3;$ $(a = (b = c)_C)_D = ((1 + 2)_A + 3)_B;$
 - "=" has a lower precedence than "+", so the associativity of "=" takes effect ("=" has the lowest precedence at all)
 - "=" is right associative
 - "+" is left associative, so A is executed before B
 - back to "=", which is right associative in $(a = b = c)$, C is executed before D
 - The order of evaluation is only relevant, if side-effects are taking place in an expression.
- Guidelines for associativity: unary operators, -=-operators and ?: are right-associative, others left-associative.
- It makes no sense to learn precedence/associativity tables by heart.
 - If in doubt, just use parentheses to control precedence and associativity explicitly.
 - Parentheses are usually also used by seasoned programmers, it has nothing to do with lack of expertise!
 - Notice: With parentheses we can not control order of evaluation.

Operators – Part XX – Other Operators and Operator Overloading

- Assignment and combined assignment.
 - Operators: =, +=, *=, /= etc.
 - Operators: &= ([and_eq](#)), |= ([or_eq](#)), ^= ([xor_eq](#)), <<=, >>=
- Extra operators:
 - Operators: ,(comma), [], (), ?:, [sizeof](#), [new](#), [delete](#), [delete\[\]](#), [typeid](#), [alignof](#) and [noexcept](#)
 - Operators: * and &
 - Operators: () (c-style cast), [static_cast](#), [const_cast](#), [dynamic_cast](#) and [reinterpret_cast](#)
 - Operators: ., ->, .*, ->*
 - Operators: ::, ::*
- C++ permits to redefine (overload) some operators for user defined types (UDTs).
 - The introduction of new operators is not possible.
 - Arity, placement, precedence and associativity of operators can't be modified.

35

- Esp. [reinterpret_cast](#) is used to get a bitwise view to data, it will be explained and used in future lectures.
- In C++, overloading of canonical operators is often required for UDTs.
- The programming language Haskell allows defining new operators and it allows modifying the precedence and associativity (together called "fixity" in Haskell) of present operators.

Control Structures – Iteration

- In programming we often have to code the same series of statements over and over.

- E.g. reading a couple of numbers from the console:

```
std::cout<<"Please enter 1. number:";
int number1;
std::cin>>number1;
std::cout<<"You entered "+number1+"!";

std::cout<<"Please enter 2. number:"<<std::endl;
int number2;
std::cin>>number2;
std::cout<<"You entered "<<number2<<"!"<<std::endl;

std::cout<<"Please enter 3. number:"<<std::endl;
int number3;
std::cin>>number3;
std::cout<<"You entered "<<number3<<"!"<<std::endl;
```

- But that is not a good practice!

- It is needed to repeat identical code parts all over in *main()*.
 - There exist a general concept concerning programming: Don't Repeat Yourself! DRY

- We can handle repeating parts of code by programming a loop to execute the same block repeatedly:

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

Control Structures – Iteration – for Loop – Part I

- What we just have seen is the most versatile, but also the most complex loop statement in C++: the for loop.

```
// Print the numbers 1 - 5 to console:  
for (int i = 1; 5 >= i; ++i) {  
    std::cout<<i<<std::endl;  
}
```

- The for loop is a flexible, counting and head controlled loop statement. What does that mean?

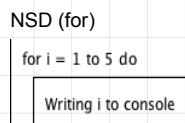
- Basically the for loop consists of two parts: head and body-block:

```
head - - - - - for (int i = 1; 5 >= i; ++i)  
body-block - - - {  
                  std::cout<<i<<std::endl;  
                  }
```

- The head contains the control statements of the loop. This part looks complex, but that is the price for for's flexibility.
 - At least we can understand, what the individual statements in the head do for us!
 - The body-block contains the statements, which are executed repeatedly as it is instructed in the head.
- With loops we introduce another control structure: called iteration (of statements).
 - However, for now we're going to dissect the complexity of the for loop's head.

Control Structures – Iteration – for Loop – Part II

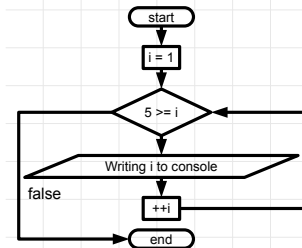
- Counting and head controlled loops like the `for` loop do also have an NSD and flowchart representation:



Good to know

The counter variable of a for loop is an exception to the rule, that variables should have descriptive names. It is commonly accepted that these variables have very short names like *i* (maybe for "index"?), *k*, *j* and so forth.) This tradition was taken over from maths, esp. from index variables.

Flowchart Diagram (for)



- We can use the flowchart diagram to analyze, how the control statements of the `for` loop's head work:

- Initialization expression or statement: `int i = 1;`
- Conditional expression: `5 >= i;`
- Update statement: `++i;`

```

// Print the numbers 1 - 5 to console:
for (int i = 1; 5 >= i; ++i) {
    std::cout<<i<<std::endl;
}
    
```

- The `for` loop as shown here works only, if the conditional expression and the update statement refer to the same state!
 - The state is given as the value *i*, which is evaluated by the conditional expression and modified by the update statement.

- The initialization statement can define multiple variables, but all must be of the same type (it makes sense, because in a Java statement we can generally only define multiple variables of the same type).

Control Structures – Iteration – for Loop – Part III

- The `for` loop's complexity: the statements in the head and the body-block are not executed in the order they are written!
 - (1) If the conditional expression evaluates to `true`, the body-block will be repeated as next step, then the update expression will be evaluated.
 - (2) Then the conditional expression is evaluated anew and we continue at (1).
 - The block of the head controlled for loop is only entered, if the conditional expression evaluates to `true`: we call it a pre-test-loop.

```
// Print the numbers 1 - 5 to console:
for (int i = 1; 5 >= i; ++i) {
    std::cout<<i<<std::endl;
}
```

- We stated, that the `for` loop is a counting loop. This is due to the fact, that we use a counter variable to control the loop.
 - Rule for orientation: use `for` if counting is involved, e.g. because we must know the number of the current iteration.

- Here some variations:

- Use the counter `i` with an increment of 2 starting with 0, until 6 is reached and write the counter values to the console:

```
// Print the numbers 0 - 6 to console:
for (int i = 0; i < 7; i += 2) {
    std::cout<<i<<std::endl;
}
```

- We can write an infinity loop with `for`, if we only write empty statements in the loop head. – It is still required to write semicolons:

```
// Runs forever and writes this text to the console until the program is forcibly stopped:
for (;) {
    std::cout<<"To infinity and beyond!"<<std::endl;
}
```

Good to know:

A statement only consisting of a semicolon, i.e. w/o any expression is called null-statement. A null-statement just does nothing.

39

- Generally, a program "rotating" in an infinity loop needs to be stopped forcibly, e.g. by stopping program execution or stopping the debugger session.

- **Terminate infinity loop:** On Unix shells Ctrl-c sends a SIGINT signal to the running program (more correctly: to its "process group"). This signal "politely" asks the program to end itself. – However a program can reject this signal and just continue execution (usually it should do some cleanup and then exit "gracefully").

If the program doesn't want to stop gracefully, we can send it the SIGTSTP signal by pressing Ctrl-z, then the console prompts again for further OS-related commands, i.e. our program is suspended and "we have the console again". Then we find our program's jobid via `jobs` or its processid via `ps` and execute `kill -TERM <processId>` or `<jobId>` to forcibly terminate the process.

Control Structures – Iteration – for Loop – Closing Notes

- `for` loops are executed sequentially, i.e. one repetition after another, not in parallel, e.g. not on multiple CPU cores.
- `for` loops are most universal and can imitate the functionality of any other loop!
 - We'll discuss other loop forms in C++ in short.
 - A basic rule for orientation: generally use the `for` loop, if counting is involved, i.e. if a counter like *i* is required.
- `for` loops are popular to iterate arrays by index (filling, processing and output), which we will discuss in a future lecture.
 - This is so, because we can use the increasing counter variable's value as index for array-access.
- C++ also supports range-based `for` loops, but we'll discuss them in a future lecture!
 - Using range-base `for` loops makes handling arrays very simple in many cases, but we need to understand more background.
- For completeness: `for` loops can be cascaded of course:

```
// Prints 25 numbers to the console:  
for (int i = 1; 5 >= i; ++i) {  
    for (int j = 1; 5 >= j; ++j) {  
        std::cout<<(i * j)<<std::endl;  
    }  
}
```


Control Structures – Iteration – while and do Loop – Part I

- **while** loops are head controlled like **for** loops, but the syntax and its semantics is different, maybe even simpler.
 - Here we formulated the **for** loop printing **ints** from 1 to 5 to the console with a **while** loop:

```
// Print the numbers 1 - 5 to console:  
for (int i = 1; 5 >= i; ++i) {  
    std::cout<<i<<std::endl;  
}
```

```
// Print the numbers 1 - 5 to console:  
int i = 1;  
while (5 >= i) {  
    std::cout<<i<<std::endl;  
    ++i;  
}
```

head - - - while (5 >= i) {
body-block - - - std::cout<<i<<std::endl;
 ++i;
 }

- Using the **while** loop, the state variable *i* is initialized before the **while** statement, i.e. it is not part of the head.
 - **while**'s conditional expression is next the **while** keyword, similar to **if** statements, it is the only part of the head.
 - The update statement is executed in while's body block, i.e. it is not part of the head.
 - The **while** loop is so simple to understand syntactically (head, body block), that no more explanations are given here.
 - => The important thing to note: the statements in the while loop's head and body are executed in the order they are written!
- There is a key difference to **for** loops: the update-operation controlling the loop condition is done in the body block.

```
for (int i = 1; 5 >= i; ++i) {  
    std::cout<<i<<std::endl;  
}
```

```
int i = 1;  
while (5 >= i) {  
    std::cout<<i<<std::endl;  
    ++i;  
}
```

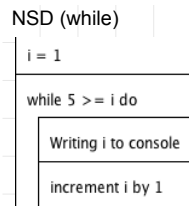
- A typical beginners' error using while loops: the update-statement in **while**'s body-block was forgotten, which leads to an infinity loop.
 - There exist exceptions to the rule, that the update-operation needs to be done in the body block, we won't discuss here.
- **for** and **while** are both head controlled/pre-test-loops: their body is only entered, if the conditional expression evaluates to true.

41

- In which situations is the update of the loop condition not required within the loop?
 - If we have a side effect in the conditional expression in the head or foot (e.g. `++i`).
 - If the data that is checked in the loop condition will be somehow modified from "outside" (e.g. from different threads).

Control Structures – Iteration – while and do Loop – Part II

- The flowchart representation of the `while` loop is equivalent to the that of the `for` loop, however, the NSD looks like this:



- We can write a `while` infinity loop, if we ensure the conditional expression to evaluate to `true` forever:

```
// Runs forever and writes this text to the console until the program is forcibly stopped:  
while (true) {  
    std::cout<<"To infinity and beyond!"<<std::endl;  
}
```

- Remember that a program "rotating" in an infinity loop needs to be stopped forcibly.

- For completeness: `while` loops can be cascaded of course:

```
// Prints 25 numbers to the console:  
int i = 1;  
while (5 >= i) {  
    int j = 1;  
    while (5 >= j) {  
        std::cout<<(i * j)<<std::endl;  
        ++j;  
    }  
    ++i;  
}
```

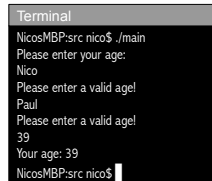
Control Structures – Iteration – while and do Loop – Part III

- Remember, when we discussed console input, esp. checking user input.
 - With the programming skills we had then, we were not able to deal with user input of unexpected data type appropriately:

```
std::cout<<"Please enter your age:"<<std::endl;
int yourAge;
std::cin>>yourAge;
if (std::cin.fail()) { // E.g.: std::cin awaited an int, but somewhat different was read:
    std::cout<<"The entered age was invalid!"<<std::endl;
}
```

- The problem: After the user entered invalid input, the program just continues with the invalid age and the failed input stream!
 - However, it would be better, if we were able to repeatedly ask the user to enter correct input and continue the program correctly.
- However, with **while** we can ask the user for input until it is valid and then continue program execution:

```
std::cout<<"Please enter your age:"<<std::endl;
int yourAge;
std::cin>>yourAge; // (0) Try accepting valid input for the first time.
while (std::cin.fail()) { // While somewhat different than an int was read:
    std::cout<<"Please enter a valid age!"<<std::endl; // (1) Prompt anew.
    std::cin.clear(); // (2) Clear stream status, esp. reset the fail status.
    std::cin.ignore(std::numeric_limits<int>::max(), '\n'); // (3) Clear buffer, esp. remove the pending newline.
    std::cin>>yourAge; // (4) Try accepting valid input for the second time.
}
std::cout<<"Your age: "<<yourAge<<std::endl;
```



```
Terminal
NicosMBP:src nico$ ./main
Please enter your age:
Nico
Please enter a valid age!
Paul
Please enter a valid age!
39
Your age: 39
NicosMBP:src nico$
```

Control Structures – Iteration – while and do Loop – Part IV

- The last loop variant we are going to discuss is the [do while](#) loop (sometimes also called [do-loop](#)).

- In opposite to [while](#) loops, [do while](#) loops are foot controlled.

- Let's reformulate our first example using the [while](#) loop with a [do while](#) loop:

```
// Print the numbers 1 - 5 to console:
int i = 1;
while (5 >= i) {
    std::cout<<i<<std::endl;
    ++i;
}
```

```
// Print the numbers 1 - 5 to console:
int i = 1;
do {
    std::cout<<i<<std::endl;
    ++i;
} while (5 >= i);
```

- Like in the [while](#) loop, the update-operation controlling the loop condition is often done in the body block.
 - (There exist exceptions to the rule, that the update-operation needs to be done in the body block, we won't discuss here.)
 - However, different to [while](#) loops, the loop's control condition is specified in the foot section of [do while](#)'s syntax.

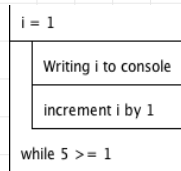
- But with foot controlled loops, we have a different way the control flow is executed:

- (1) The code in [do while](#)'s block is executed.
 - (2) Then the loop condition is evaluated. If it evaluates to [true](#), the control flow starts over at (1).
 - The block of a foot controlled loop is entered at least once, therefor we call them post-test-loops.

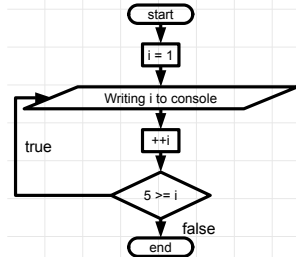
Control Structures – Iteration – while and do Loop – Part V

- The flowchart and NSD representations of the **do while** loop reflect its foot controlled idea:

NSD (do while)

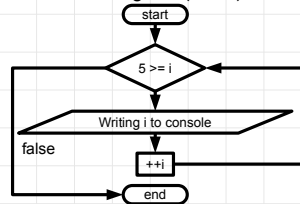


Flowchart Diagram (do while)

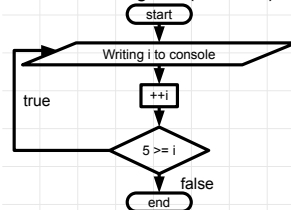


- Head controlled **while** loop vs foot controlled **do while** loop: notice the positions of the control condition in the control flow:

Flowchart Diagram (while)



Flowchart Diagram (do while)



Control Structures – Iteration – while and do Loop – Part VI

- We can write a **do while** infinity loop, if we ensure the conditional expression to evaluate to true forever:

```
// Runs forever and writes this text to the console until the program is forcibly stopped:  
do {  
    std::cout<<"To infinity and beyond!"<<std::endl;  
} while (true);
```

- Remember that a program "rotating" in an infinity loop needs to be stopped forcibly.

- For completeness: **do while** loops can be cascaded of course:

```
// Prints 25 numbers to the console:  
int i = 1;  
do {  
    int j = 1;  
    do {  
        std::cout<<(i * j)<<std::endl;  
        ++j;  
    } while (5 >= j);  
    ++i;  
} while (5 >= i);
```

- There is a remarkable point: **do while** loops must be written with blocks! Also empty **do while** loops must use blocks:

```
// Invalid!  
int i = 1;  
do  
    ++i;  
while (5 >= i);
```

```
// Correct!  
int i = 1;  
do {  
    ++i;  
} while (5 >= i);
```

```
// Invalid!  
int i = 1;  
do  
    ;  
while (5 >= i);
```

```
// Correct!  
int i = 1;  
do {  
    // empty block  
} while (5 >= i);
```

Control Structures – Iteration – while and do Loop – Part VII

- Let's review our example using a **while** loop to handle valid user input:

```
std::cout<<"Please enter your age:"<<std::endl;
int yourAge;
std::cin>>yourAge;
while (std::cin.fail()) { // While somewhat different than an int was read:
    std::cout<<"Please enter a valid age!"<<std::endl;
    std::cin.clear();
    std::cin.ignore(std::numeric_limits<int>::max(), '\n');
    std::cin>>yourAge;
}
std::cout<<"Your age: "<<yourAge<<std::endl;
```

Advice: **do while** is excellent to write menus in a console program: "**do** show the menu **while** the program was not quitted".

- However, there are issues: we have to write the code to print the prompt and accept user input for two times!

- Using **do while** we can rewrite this code to deal with the prompt and user input more efficiently:

```
bool wasValidInput = false;
int yourAge;
do {
    std::cout<<"Please enter your age:"<<std::endl;
    std::cin>>yourAge;
    wasValidInput = std::cin.fail();
    if (!wasValidInput) {
        std::cin.clear();
        std::cin.ignore(std::numeric_limits<int>::max(), '\n');
        std::cout<<"The age must be an int!"<<std::endl;
    }
} while (!wasValidInput); // While somewhat different than an int was read.
std::cout<<"Your age: "<<yourAge<<std::endl;
```

```
Terminal
NicosMBP:src nico$ ./main
Please enter your age:
Nico
The age must be an int!
Please enter your age:
45
Your age: 45
NicosMBP:src nico$
```

- To be frank, we have to use more code, but it better reflects the idea: "**do** the prompt **while** input is invalid".

Control Structures – Iteration – Loop Control – Part VIII

- Occasionally it is required to control the execution of loops in a finer grained manner to execute loops only partially.
- C++ provides the continue statement to skip iterations and provides the break statement, known from switch, to stop loops.
- Let's assume a loop, which prints the first 100 int numbers to the console, but no numbers, which are multiples of 10.
 - There're at least two ways to do it in C++: (1) print only non-multiples of 10 or (2) skip console output, if we have a multiple of 10:

```
// (1) print only non-multiples of 10:  
for (int i = 0; 100 >= i; ++i) {  
    if (0 != i % 10) {  
        std::cout<<i<<std::endl;  
    }  
}
```

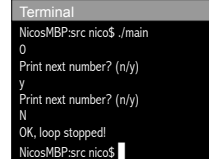
```
// (2) skip console output, if we've a multiple of 10:  
for (int i = 0; 100 >= i; ++i) {  
    if (0 == i % 10) {  
        continue;  
    }  
    std::cout<<i<<std::endl;  
}
```

- The continue statement does the trick, it skips the current iteration and continues with the next iteration.
 - (a) if *i* reaches the value 10, the condition `0 == i % 10` evaluates to true and continue is executed
 - (b) continue skips the console output and executes for's head, the loop condition evaluates to true and *i* is incremented to 11
 - (c) for's body block is executed anew ...
- continue can be used with for, while and do while loops (also with range-based for loops).

Control Structures – Iteration – Loop Control – Part IX

- C++ also provides a means to stop a running loop completely with the `break` statement.
- Assume an infinity loop, which prints all `ints` beginning from 0 to the console with a prompt, unless the user stops this cycle:

```
int i = 0;
while (true) {
    std::cout<<i<<std::endl;
    std::cout<<"Print next number? (n/y)"<<std::endl;
    char answer;
    std::cin>>answer;
    if (answer == 'n') {
        break;
    }
    ++i;
}
std::cout<<"OK, loop stopped!"<<std::endl;
```



```
Terminal
NicosMBP:src nico$ ./main
0
Print next number? (n/y)
y
Print next number? (n/y)
n
OK, loop stopped!
NicosMBP:src nico$
```

- Here, the `break` statement does the trick, it stops the loop completely and control flow continues after the loop.

Good to know:

Because infinity loops don't apply conditions, only `break`, `continue`, `return` and `throw` can end an infinity loop programmatically.

Control Structures – Iteration – Loop Control – Part X

- It should be said, that the same behavior can be achieved, by getting rid off the dangerous infinity loop:

```
// (1) Uses an infinity loop and break:
int i = 0;
while (true) {
    std::cout<<i<<std::endl;
    std::cout<<"Print next number? (n/y)"<<std::endl;
    char answer;
    std::cin>>answer;
    if (answer == 'n') {
        break;
    }
    ++i;
}
std::cout<<"OK, loop stopped!"<<std::endl;
```

```
// (2) The same behavior without break:
int i = 0;
bool userAbortedOutput = false;
while (!userAbortedOutput) {
    std::cout<<i<<std::endl;
    std::cout<<"Print next number? (n/y)"<<std::endl;
    char answer;
    std::cin>>answer;
    if (answer == 'n') {
        userAbortedOutput = true;
    } else {
        ++i;
    }
}
std::cout<<"OK, loop stopped!"<<std::endl;
```

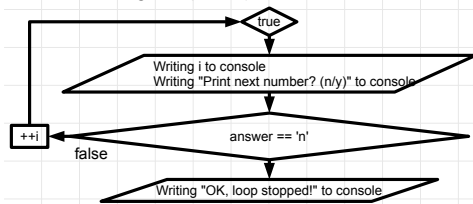
- In (2) we just use the status variable *userAbortedOutput* to control the loop, instead of an infinity loop.
- `break` can be used with `for`, `while` and `do while` loops (also with range-based `for` loops).

Control Structures – Iteration – Loop Control – Part XI

- Code containing **break** and/or **continue** can basically only be illustrated with flowchart diagrams:

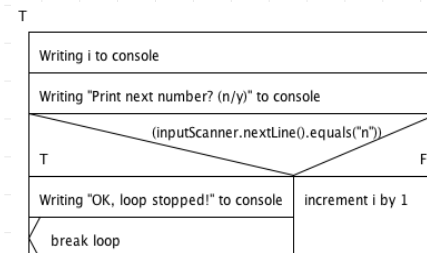
```
// (1) Uses an infinity loop and break:
int i = 0;
while (true) {
    std::cout<<i<<std::endl;
    std::cout<<"Print next number? (n/y)"<<std::endl;
    char answer;
    std::cin>>answer;
    if (answer == 'n') {
        break;
    }
    ++i;
}
std::cout<<"OK, loop stopped!"<<std::endl;
```

Flowchart Diagram (break)



- Sometimes, people use the "EXIT" symbol to express **break** in NSDs:
 - (Variations of that symbol (different edges etc.) are sometimes used to express **continue**, but those are not standardized for NSDs).

NSD (break)



Control Structures – Iteration – Loop Control – Part XII

- C++ provides more not so well-known means to control loop execution, when we use [break/continue](#) together with labels.

- What is a "label"?

- C++ code can be marked with special identifiers, i.e. labels, to give the code a more "row-wise" structure.

```
acceptInput:
std::cout<< "Please enter an int larger than 0:"<<std::endl;
int number;
std::cin>>number;
checkInput:
if (number <= 0) {
    std::cout<<"This is not what I expected."<<std::endl;
} else {
    allFine:
    std::cout<<"OK!"<<std::endl;
}
```

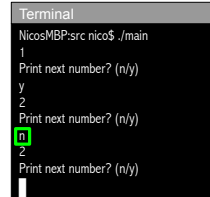
- In the code above, we have three labels: *acceptInput*, *checkInput* and *allFine*.
 - As can be seen, a label is a valid and unique C++ identifier followed by a colon, which can be placed almost anywhere in the code.
 - E.g. there is the restriction, that they must be written before another statement.
 - Labels per se have no functional meaning for the program, but have a documentary meaning like comments.
 - Labels can be thought of as replacement for "line numbers" as it is known from some Basic dialects.

- As mentioned, we can do more with labels, when loops, esp. cascaded loops come into play.

Control Structures – Iteration – Loop Control – Part XIII

- In programming, we will occasionally encounter cascaded loops.
 - Let's assume a cascaded loop, which counts up two numbers that are multiplied.
 - The user is prompted to stop generating the products and console output after each product respectively.

```
for (int i = 1; i < 10; ++i) {  
    for (int j = 1; j < 10; ++j) {  
        std::cout<<(i * j)<<std::endl;  
        std::cout<<"Print next product? (n/y)"<<std::endl;  
        char answer;  
        std::cin>>answer;  
        if (answer == 'n') {  
            break;  
        }  
    }  
    std::cout<<"OK, loop stopped!"<<std::endl;  
}
```



```
Terminal  
NicosMBP:src nico$ ./main  
1  
Print next number? (n/y)  
y  
2  
Print next number? (n/y)  
n  
2  
Print next number? (n/y)
```

- No, it does not work! The program keeps generating products, even though the user answered with "n"!
 - After the user entered "n", we would have assumed, that the program stops and prints "OK, loop stopped!" to the console!
 - The problem: break does only stop the closest loop, i.e. the for loop counting up j!

Control Structures – Iteration – Loop Control – Part XIV

- For exactly such cases, i.e. cascaded loops must be stopped from the innermost loop, we can use goto with a label.

```
for (int i = 1; i < 10; ++i) {  
    for (int j = 1; j < 10; ++j) {  
        std::cout<<(i * j)<<std::endl;  
        std::cout<<"Print next product? (n/y)"<<std::endl;  
        char answer;  
        std::cin>>answer;  
        if (answer == 'n') {  
            goto endOutput;  
        }  
    }  
    endOutput:  
    std::cout<<"OK, loop stopped!"<<std::endl;  
}
```

```
Terminal  
NicosMBP:src nico$ ./main  
1  
Print next number? (n/y)  
y  
2  
Print next number? (n/y)  
n  
Print next number? (n/y)  
OK, loop stopped!  
NicosMBP:src nico$
```

- We have set the label endOutput after the outermost loop of the cascaded loops and before the "OK, loop stopped!" message.
- The goto statement, addresses the label endOutput.
- When the goto statement is executed, control flow will jump behind the outermost loop, hitting the label endOutput.
- We can also set a label before an outermost loop of the cascaded loops.
 - When hitting this label from a goto of an inner loop, control flow would jump before the outermost loop.
- Tip: gotos should not be used, because the code look/behaves weird. Better rewrite your code!

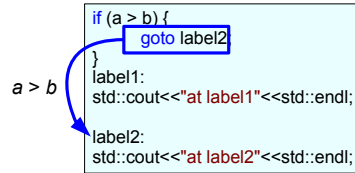
54

- goto** cannot jump behind a declaration/initialization of a variable:

```
goto lab; // Invalid!  
int c = 42;  
lab:  
std::cout<<c<<std::endl;
```

Excursus: forced Branching – Part I

- Actually, we can do more with labels apart from branching: we can use them to code forced branching.
- Here an (contrived) example:



- if $a > b$, then the control flow skips `label1` and its following statement(s) right below the statement marked with `label2`:

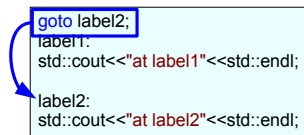
```
Terminal  
NicosMBP:src nico$ ./main  
at label2  
NicosMBP:src nico$
```

- if $a \leq b$, then the control flow just continues to the end, hitting `label1` and `label2`.

```
Terminal  
NicosMBP:src nico$ ./main  
at label1  
at label2  
NicosMBP:src nico$
```

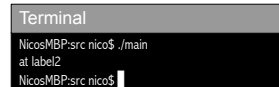
Excursus: forced Branching – Part II

- Remarks on forced branching with labels:
 - Forced branching is different to conditional branching, in that it uses the `if` keyword **and jumps** to labels.
 - Using forced branching practice is meanwhile unconventional in C++ and leads to weird code! → Don't use it!
 - (But such code could be result of automatic code generation. – It is simple to generate such code.)
- Forced branching is one thought away from unconditional branching.



```
goto label2;  
label1:  
std::cout<<"at label1"<<std::endl;  
label2:  
std::cout<<"at label2"<<std::endl;
```

A diagram illustrating forced branching. A blue box highlights the `goto label2;` statement. A blue arrow points from this statement down to the `label2:` line of code, bypassing the `label1:` block.



```
Terminal  
NicosMBP:src nico$ ./main  
at label2  
NicosMBP:src nico$
```

A terminal window showing the execution of a program. The prompt is `NicosMBP:src nico$`. The user enters `./main`, and the output is `at label2`. The prompt returns to `NicosMBP:src nico$`.

- In this example, control flow will **always jump** directly and under all circumstances to *label2*!
 - The statements below *label1* will never be executed!
- Unconditional branching with the `goto` keyword should be avoided!
 - When is `goto` used anyhow:
 - On the automatic generation of code.
 - For algorithms that need to run really fast.

Control Structures – Exiting Iteration – Part I

- There are even more ways to "exit" a loop in C++, which we will discuss in future lectures.

- The `return` statement exits the surrounding function and thus all surrounding loops:

```
int main() {  
    for (int i = 1; i < 10; ++i) {  
        for (int j = 1; j < 10; ++j) {  
            std::cout<<(i * j)<<std::endl;  
            std::cout<<"Print next product? (n/y)"<<std::endl;  
            char answer;  
            std::cin>>answer;  
            if (answer == 'n') {  
                return 0;  
            }  
        }  
    }  
}
```

- A `return` statement could also transport a value to the caller of the function, which acts as a result of the function.
 - In this case, the `main()`-function is exited with the value 0, which means "program ended successfully" for C++-programs.
 - We will discuss functions in depth soon.
- Similar to `return` statements are `throw` statements, in that they also exit the surrounding function and all surrounding loops.
 - `throw` statements are used to emit exceptions. Exceptions are a powerful, yet relatively simple way to deal with run time errors.

Control Structures – Exiting Iteration – Part II

- Special statements for loop control:
 - `continue` – skips the current loop.
 - `break` – leaves the next surrounding loop completely.
- Labels and `goto`
 - The unconditional jump statement.
 - Do not use `goto` statements! Leads to spaghetti programming "pasta oriented programming".
- When is `goto` used anyhow:
 - To escape from the innermost loop of cascaded loops, which is avoidable.
 - On the automatic generation of code.
 - For algorithms that need to run really fast.
- `return` and `throw`
 - Return from a function with or without value.
 - Throw an exception, which leaves the function as well.

We should always use Blocks and Indentation – True Example using **goto**

- With a solid understanding of conditional code and indentation of blocks, the hideous **goto** stresses their relevance.
- Apple introduced a serious and hard-to-see bug in 2014 in their SSL/TLS library (C/C++), the code looked like this:

Hint

C and C++ support **goto** statements, whereas Java doesn't!

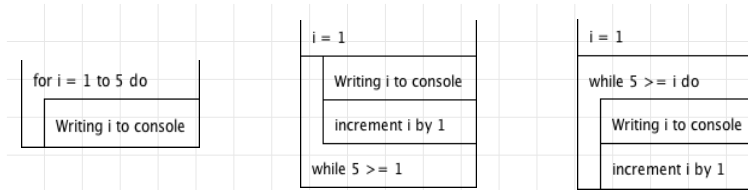
```
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0) // (1)
    goto fail;
goto fail; // (2)
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;
// ...
fail:
```

- The problem: the second **goto** (2) was always executed, because it was not executed under the condition of (1)!
- Apple's developers avoided blocks to structure the code, which might have unleashed this serious bug:

```
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0) { // (1)
    goto fail;
}
goto fail; // (2) Oops!
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0) {
    goto fail;
}
// ...
fail:
```

- Indentation is only a visual help to understand the code, but it is important to understand code from a distance.
 - Blocks are important to understand the scope of statements, in this example see how the **goto** goes horribly wrong!

Control Structures – Iteration – Summary



- General remarks on loops:
 - `for`, `while` and `do while` are statements, which can be cascaded in any thinkable way like `if` statements.
 - Beware of unwanted infinity loops!
 - Infinity loops are a very common error due to wrong conditional expressions or unwanted modifications of the counter variable in the body-block.
 - You should know how to stop program execution from the console, on the IDE and platform you use (Unix/Linux/macOS/Windows).
 - Basically, you can use all loop variants for everything, however a certain loop variant can be beneficial over the others.
 - `break`, `continue` and `goto` should be used sparingly. Often, loops can be rewritten without those control statements!
 - `return` and `throw` statements can also be used to exit all surrounding loops and the surrounding function.
 - We will discuss functions in a future lecture.
 - `for`, `while` and `do while` loops are executed sequentially, not in parallel, e.g. not on multiple CPUs or cores.
- 60
- Later, we'll discuss the range-based `for` loop, that can solve problems like infinity loops and counter issues in `for` loops.

- C++20's `std::ranges` API will also simplify code a lot to avoid many loops.

Imperative Programming – Closing Words

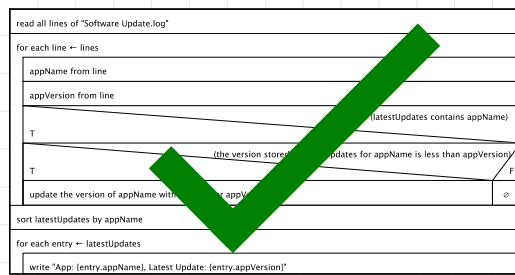
Definition

Imperative programming means to program with statements being executed sequentially to change the state of the program.

- Statements are executed in the order they where written.
 - Whereas in functional programming so-called lambdas and deferred execution this is not necessarily the case.
- Statements change the state of the program by modifying variables.
 - Those modified variables control the execution of following statements, esp. control structures, e.g. if's and loops.
 - We say, that variables communicate via side-effects with the statements, that are executed next.
 - Where in functional programming side effects are disallowed and control structures work completely differently, if any.

Structured Programming

- Programming only using sequences, branches and loops is called structured programming.
 - Very traditionally, structured programming means, that code using unconditional branching (**gotos**) is prohibited.
 - Code with **gotos** often evolves to so called spaghetti code, i.e. very intertwined code of low maintainability(/quality).
- The initiative of structured programming begun in the 1970ies and was coined by Mr. Edsger Wybe Dijkstra ['dɛɪk,stra].
 - Basically, during that time the era of high-level (programming) languages (HLL) rose.
 - Lower level (programming) languages like assembly languages use jump/branch statements, i.e. unconditional branching, inherently.
 - That time also the need to document algorithms in a structured way, with flowcharts and esp. NSDs came into play.



62

- Spaghetti code: program execution is on position x and I know, from where I jumped here, and now program execution will jump to y.

Thank you!