(2) Basics of the C++ Programming Language

Nico Ludwig (@ersatzteilchen)

TOC

- (2) Basics of the C++ Programming Language
  - Style and Conventions in this Course
  - Elements of imperative Programming
  - Values, Expressions, Statements, Variables and Declarations
  - Syntactic Style and Scope
  - Conditional Code, the conditional Operator, if/else Branching, switch-Branching
  - Identifiers and Comments
  - Constants
  - Fundamental Types
  - Implicit and explicit Type Conversion
  - Integral Division
  - First Look on Cstrings

- Cited Literature:
  - Bruce Eckel, Thinking in C++ Vol I
  - Bjarne Stroustrup, The C++ Programming Language
  - https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#Rp-Cplusplus

2

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!

Starting to write Programs

- Well, we discussed a lot of theory up to now. And now it is time to get our feet wet!
  - Assume this C++ program, which just prints the text "Hello, World!" to the console:
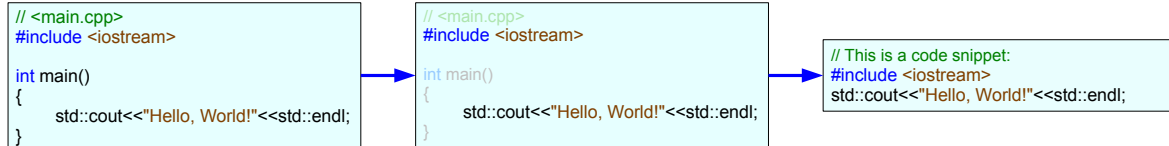
```
// <main.cpp>
#include <iostream>

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

  - The text, which is the program's <u>code</u>, looks weird:
    - What do the different English words do?
    - What do the different parentheses mean?
    - And what is the meaning of colors?

- We'll learn, that there are rules in the C++ programming language, which are not difficult to understand.
  - These rules span two things: the <u>syntax</u> of the language and the <u>semantics</u> of the language.
  - <u>The first hurdle to get is getting our code compiled</u>, therefor we have to learn how to write correct or valid C++ programs.

- In this lecture, we'll get a feeling for the <u>syntax</u> and gradually understand the <u>semantics</u> of syntactic elements.
  - Keep in mind, that high level languages like C++ are specially designed to be human-readable.

4

Code Snippets

- Hence we'll begin using <u>code snippets</u> (or simply called "<u>snippets</u>") as examples.
    - C++ code is written into ordinary text files, they just have the suffix .cpp (or .cc).
    - A C++ program also brings some of "fluff", which has no meaning to us right now.
    - Therefor we will strip C++ code down to snippets:

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

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

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

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

```
// This is a code snippet:
#include <iostream>
std::cout<<"Hello, World!"<<std::endl;
```

- In C++ snippets, we'll get rid off the <u>surrounding *main()-function*</u>.
    - (We'll learn what functions are for in future lectures.)
    - Then only the "essence" of the code remains.
    - But: A snippet is <u>no runnable program code by itself</u>, it is <u>still required to put its code into a *main()-function.*</u>
    - <u>=> I.e. for running C++ programs we used a *main()*-function, without knowing why it is required right now.</u>

- We'll <u>use mostly snippets</u>, instead of <u>fully blown C++ programs</u> in upcoming lectures!

5

- C++ code is written in text files. Sometimes the code is called "source code" or programmers just call the whole source code as "the sources".
- An important thing to keep in mind is that the execution of C++ program starts in the free *main()-*function. – Hence we accept that we will leave the explicit definition of *main()* away in the code examples of this course.
    - There can be only one *main()*-function in a C++-application.

#includes

- #include-directives <u>include files with information for the C++ compiler</u>. Included files are called <u>header-files (h-files)</u>.
  - #include-directives are usually just called "pound-includes".
  - They are called header-files, because its <u>#includes are usually written on the very head of a cpp-file</u>.

- The h-file \<iostream\> adds information about *std::cout*, *std::endl* and about the special <u>\<\<-operator</u> into main.cpp.

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

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

  - <u>Standard h-files, that are part of the C++ libraries are written in angle-braces \< and \>.</u>
  - We can (and must) define own h-files, whose file names must be included written in <u>double quotes " and "</u>.
    - User defined h-files usually have the extension .h or .hpp or .hh

- C++ supports some other directives, e.g. #define, #undef, #ifdef, #endif etc., they all have their #-prefix in common.
  - (These directives must be written without a terminating semicolon, an aspect we'll discuss soon.)

- The C++-compiler doesn't evaluate directives! They are evaluated by the so-called <u>C++-preprocessor</u>!
  - Therefor we call them <u>preprocessor directives</u>.
  - <u>The result of the C++preprocessor is then passed to the C++-compiler.</u>

6

Compile and Run our Program on the Console

- Before we can put the presented C++ program into effect, it needs to be compiled.
  - Compilation means, that the symbolic C++ code is transformed into machine code.
  - The C++ code is just written in a text file with the extension .cpp. Let's assume our program code resides in main.cpp.
  - A compiler is a program, which performs this transformation. A specific C++ compiler is gcc (for GNU Compiler Collection).

- Compile the C++ program in main.cpp separately, the compiled machine code is put into a binary file named main.o:

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

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

```
Terminal
NicosMBP:src nico$ g++ -c main.cpp -std=c++11
NicosMBP:src nico$
```

- We need a further immediate step in C++: we have to link main.o to the effective program file called "main":

```
Terminal
NicosMBP:src nico$ g++ main.o -o main
NicosMBP:src nico$
```

- After linking, we can execute main and the message "Hello, World!" is written to the console:

```
Terminal
NicosMBP:src nico$ ./main
Hello, World!
NicosMBP:src nico$
```

7

- In upcoming lectures, we will discuss separated compilation and linking in detail.
- Here the full command lines for separated C++ compilation and linking with gcc (macOS 10.14):
- Here we use gcc's variant g++ on the command line. Often, g++ in an alias for gcc, which is predefined with some C++-standard options.
  - Compile (g++ is preferred):
    ```
    g++ -c main.cpp -std=c++11
    gcc -c main.cpp -std=gnu++11
    ```
- Link (g++ is preferred):
  ```
  g++ main.o -o main
  gcc main.o -o main -lstdc++
  ```

Highlighting Code in Colors

- If code in a snippet produces command line output, it will occasionally be shown in the code with the "// >"-notation:

```
std::cout<<"Hello, World!"<<std::endl;
```
→
```
std::cout<<"Hello, World!"<<std::endl;
//> Hello, World!
```

Good to know
The symbol "/" is usually called (forward) "slash" or "whack" (but "whack" is virtually the term for the backslash). Sometimes it is also called solidus. However, "slant" is the official ASCII name for "/".

- As can be seen, the text line starting with "// " was highlighted in green color.
    - In C++ a line starting with two slashes "//" is a C++-comment.

- We use colors in the snippets to highlight elements in the code: brown for text, blue for keywords and green for comments.
    - We use colors in the code only to highlight different elements of the language.
    - We'll discuss the meaning of texts, keywords and comments in the upcoming slides.

- The highlighting (i.e. coloring) of language elements could look completely different in the code editor you use:

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

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

← →

⊞ < > 📄 CppExample 📁 CppExample 📄 main.cpp No Selection
```
// <main.cpp>
#include <iostream>

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

8

    - All code editors allow to configure the coloring of language elements to fit our need.

Console Output

- We have to discuss *std::cout*, *std::endl* and the <<-operator because we use them often: they write output to the console.

- When we compile, link and run this snippet of code (of course it must be enclosed in *main()* etc.), we get this output:

```
std::cout<<"Hello, World!"<<std::endl;
```

```
Terminal
NicosMBP:src nico$ ./main
Hello, World!
NicosMBP:src nico$
```

- Alternatively, we get the same console output with this code:

```
std::cout<<"Hello, "
        <<"World!"
        <<std::endl;
```

```
Terminal
NicosMBP:src nico$ ./main
Hello, World!
NicosMBP:src nico$
```

  - Actually, we can mix the <<-operator with any textual data.
  - The <<-operator kind of concatenates values, which are written to the console. It must just be "<<-put" "into" *std::cout* in the end.
  - Notice: *std::cout* is an object, which represents the console and with the <<- operator we send stuff out to the console.

- The object *std::endl* represents a line break. We can also put a line break in between "Hello, " and "World" via <<:

```
std::cout<<"Hello, "<<std::endl<<"World!"<<std::endl;
```

  - Mind, that the placement of the <<-operator and *std::endl* in the code doesn't matter for the console output!

```
Terminal
NicosMBP:src nico$ ./main
NicosMBP:src nico$
Hello,
World!
NicosMBP:src nico$
```

9

- To learn a programming language its syntax and its semantics must be learned.

- For example the syntax of this code snippet:

```
std::cout<<"Hello, World!"<<std::endl;    // line (1)
//> Hello, World!                         // line (2)
```

  - On line (1) we have the words *std*, *cout* and *endl* separated by two colons.
  - *std::cout and std::endl* enclose the text "Hello, World!" written in quotes.
  - The line ends with a semicolon.
  - Line (2) starts with // and contains the text '> Hello World!'

Good to know
The english term for the symbol : is colon.

- What we've described here is the structure of the code, its words and individual characters. This structure is called syntax.
  - Therefor, the (colored) highlighting of the code's structure is also called syntax highlighting.

- What this structure really does, when the program runs is not obvious. The meaning of the structure is called semantics.

Syntax: structure of the code ↔ **Semantics**: meaning of the syntax

- The special quality of high level languages is, that their syntax allows to guess its semantics pretty reliably. 10

- # Esp. the need to write semicolons to terminate statements scares C++ newbies.

C++ Syntax Cornerstones – Part I – Imperative Elements

- We'll begin by discussing <u>imperative programming</u> in C++.

> **Definition**
> *<u>Imperative programming</u> means to program with <u>statements</u> being <u>executed sequentially</u> to <u>change the state</u> of the program.*

- Imperative programming is a general <u>programming paradigm</u>.
    - Other paradigms: procedural programming, functional programming and object-oriented programming.

- Imperative programming includes following (imperative) <u>elements</u> in general:

> **Good to know**:
> Imperative from latin *imperare* – to command someone/something.

  - <u>Values</u> – values, which <u>represent the state</u> or possible states of the program, which have a <u>type</u> and <u>consume memory</u>
  - <u>Variables</u> – <u>hold the state</u> of the program, they are abstract locations (or "cells") in memory with names
  - <u>Operators</u> – connect values and variables to <u>express an operation</u>
  - <u>Expressions</u> – formulate a <u>combination of values, variables and operators that can be evaluated</u>, <u>yielding another value</u>
  - <u>Statements</u> – executable instructions, that execute expressions, which typically <u>change the state</u> (i.e. the contents of variables) of the program
  - <u>Conditional branches</u> – execute statements <u>depending an a condition</u>
  - <u>Unconditional branches</u> – <u>jumps</u> between statements <u>unconditionally</u>
  - <u>Loops</u> – execute statements <u>repeatedly</u>
  - <u>Input and output</u> – <u>communicate</u> with the "world outside of the program" (the user, the file system or a network)

- Values represent a very important, yet simple concept in any programming language.
  - E.g. in the "Hello, World!" program we saw the value "Hello, World!".

    `std::cout<<"Hello, World!"<<std::endl;` → `// The literal value "Hello, World!"` `"Hello, World!"`

- Values simply represent data in a program.
  - All values in C++ have a certain type and a certain memory consumption.
    - The text "Hello, World!" is of type const char*, which is the type to hold textual data in C++.
    - Textual data are called strings in most programming languages, so in C++. Strings are of type const char* in C++.
    - The memory consumption of "Hello, World!" is relevant, but not yet important, however, it depends on the string's length.

- If we write a value directly in our source code, e.g. "Hello, World!", we call such a value literal value, or just literal.
  - C++' fundamental types can have literal values. E.g. writing a text into double quotes makes it a literal string value or string-literal.
  - In this course, we highlight string-literals in brown color.

- However, values of fundamental types have an important limitation: Those values can not be modified in C++!

- To do something useful with unmodifiable values, we have to understand the ideas of C++' variables and expressions.

12

- In assembly languages literals are sometimes called immediate constants. In opposite to most HLLs, immediate constants are written immediately into the instruction stream. In HLLs, literals are usually stored into a kind of data segment, from which the literal values are loaded.

C++ Syntax Cornerstones – Part III – Expressions

- Expressions represent another important, but simple concept in any programming language.
  - E.g. consider this expression:

```
// Expression:
3 + 4
```

- Obviously, expressions just represent "calculations" in a program, in this case an addition calculation.
  - C++' +-operator does expectedly add values, as we know it from mathematics.
  - C++ has to evaluate an expression (i.e. "do the addition" in this case), to get its resulting value (i.e. the "sum" in this case).
    - To talk about the value of an expression we say the expression "evaluates to a value" or it "yields a value", or that the expression "returns a value".
  - This expression evaluates to the value 7. – Notice, that "the 7" is not written as literal value, it is rather a computed value.
  - On the other hand, we used two literals, 3 and 4, to formulate this expression.
  - 3 and 4 are int-literals. C++' fundamental type int, which is a type to hold integer numbers.

- I didn't call expressions "mathematical terms", because sometimes they aren't.
  - C++' expressions can yield values and modify the state of a program. The latter concept is basically unknown in mathematics.
  - C++' uses types and operators, which have no counterpart in ("elementary") mathematics.

- Elementary facts about expressions:
  - A literal value is also an expression. A literal is an expression evaluating to itself.

```
3 // this is an int-literal and an expression
```

  - An expression can be built up from other expressions.

```
3 + 4 - 7 // multiple expressions
```

13

C++ Syntax Cornerstones – Part IV – Output, Intermediate Results and Variables

- From the "Hello, World!" program we know, that *std::cout and std::endl* is a syntax to output information to the outside world:
  - We know how to output textual values, e.g. the string-literal "Hello, World!" to the command line from a C++ program:

    ```
    std::cout<<"Hello, World!"<<std::endl;
    // >Hello, World!
    ```
    - A new syntactic need is, that we have to write a ; (semicolon) at the end of *std::cout and std::endl*, that executes the output of "Hello, World!".
  - Well, it is also possible to output numeric values, e.g. the int-literal 42 to the command line:

    ```
    std::cout<<42<<std::endl;
    // >42
    ```
  - We can also output the immediate result of an expression to the command line:

    ```
    std::cout<<45 * 3 + 21 + 5 + 45 * 3<<std::endl;
    // >296
    ```
    - The operator * means multiplication! When we progress seeing more C++ code, we'll see more of C++' operators.

- Above, the expression *45 * 3* is used in the calculation for two times. C++ allows storing intermediate results in variables.
  - We can calculate the expression *45 * 3* and store its intermediate result in a variable named *product*:

    ```
    int product = 45 * 3;
    ```
  - Then we can use *product* in further expressions to get the effective result:

    ```
    std::cout<<product + 21 + 5 + product<<std::endl;
    // >296
    ```
  - Here we can clearly see, that the line calculating *product* must be executed sequentially before *product* is used in the next line!
  - As can be seen, we've defined a variable *product* with the keyword int and initialized it with the result of the expression *45 * 3*.
  - Using the keyword "int" defines a variable being of type integer. Integer variables can only hold integer values!

- C++ reserves <u>symbols</u> for its <u>grammar</u>, these symbols are called <u>keywords</u>.
  - In upcoming snippets all keywords are written in <u>blue color</u>. We already saw some keywords: <u>void</u>, <u>int</u> etc..

- We see ; all over in C++ code. They must be written to <u>execute expressions</u>. Expression terminated with ; are called <u>statements</u>.
  - Remember, that we didn't terminate preprocessor directives like #include with semicolons: <u>they are no C++-statements</u>!

- Initialization and assignment statements are excellent examples to consider syntax versus semantics.
  - <u>Initialization</u> and <u>assignment statements</u> have <u>similar C++-syntax</u>, but their <u>meaning is different</u>: they have <u>different semantics</u>:

```
int product = 45 * 3; // Initialization
product = 42;          // Assignment
```

- Compile time errors versus run time errors:

```
49 = 3 + 5; // Invalid! initialization/assignment to a constant value
```

  - This code is <u>no correct C++-syntax</u>! The compiler will <u>reject it</u>, <u>issue a compile time error</u> and <u>abort the compilation-process</u>.
    - An important function of the C++-compiler is to <u>check the syntax</u> for syntax errors. <u>Syntax errors result in compile time errors and abort compilation.</u>
    - The syntax error here: a constant literal value can not be assigned! <u>Sounds logical...</u>

```
int zero = 0;
int oddResult = 42/zero;
```

  - Both statements are <u>ok for the compiler</u>, but the last one <u>behaves undefined</u> at run time.
    - The result of the division by 0 is undefined in C++! <u>Sounds familiar...</u>
    - There can also exist <u>link time errors</u> in the C-family languages. We'll discuss this kind of complex matter in a future lecture.

15

- <u>What keywords does the audience know from any programming languages?</u>
  - Uninitialized variables for experts: Uninitialized auto locals have arbitrary values, statics (globals) have default values, e.g. 0.
- <u>Syntax versus semantics; what's that?</u>
  - Grammar/keywords versus their meaning.
- <u>Programming errors can occur on different "times" in development. Can anybody explain this statement?</u>
  - The time <u>during programming</u> is often called "design time".

C++ Syntax Cornerstones – Part VI – Variables

- Besides values, the most elementary syntax cornerstones are variables. Variables hold data/values/state in a program.

  `int width = 32;`

  - In C++, variable definitions make variables applicable in the code.
  - Variables need to have a type on the definition, this is called static typing.
  - We can define multiple variables of the same type in one statement as a comma-separated list:

    `int width = 32, height = 67;`

variable type	variable name	initialization
int	width	[= 32];
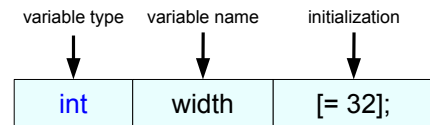
- A defined variable has data/value/state and consumes memory.
  - The compound of a portion of memory and the data/value/state it holds is called object.

  C++11 – uniformed initializers
  `int age{19};`

- Besides definition, variables can be initialized, assigned to and read.
  - Initialization sets initial values to a variable.

    `int age = 19; int age(19); int age = {19}; // Initialization`

  - A variable needs not to be initialized in C++ or assigned to before usage, in this case its value is undefined (in most cases)!
    - Therefor, defined variables should be assigned to as soon as possible, if they are not initialized.

```
int diameter;
// Variable diameter has not been initialized!
std::cout<<diameter<<std::endl;
//> -98129837244
```
diameter (-98129837244)

```
int diameter = 56;
// Fine! diameter has been initialized!
std::cout<<diameter<<std::endl;
//> 56
```
diameter (56)

  - Assignment sets a variable to a new value.

    `age = 0; // Assignment`

  - Reading retrieves the value of a variable.

    `int hisAge = age; // read age and initialize hisAge.`

16

- The same variable should only be used for exactly one purpose!
- After definition variables are referenced just by their name (prefixes or sigils (like the '$' sigil) as found in scripting languages are not used).

- Besides definitions, C++ offers the concept of <u>declarations</u>.

- <u>A declaration makes a name/symbol, e.g. a variable known to the compiler, but not yet usable.</u>
  - Declarations are expressed with the extern keyword (the extern keyword is optional for functions):

    ```
    extern int anotherAge; // variable declaration
    ```
  - The same symbol (e.g. variable) can be declared for multiple times, but it cannot be used without definition:

    ```
    extern int anotherAge; // variable declaration
    extern int anotherAge; // twice
    std::cout<<anotherAge<<std::endl; // Invalid (linker): Undefined symbol: _anotherAge
    ```
  - <u>Mind</u>: in opposite to declarations, <u>definitions cannot be repeated: redefinitions are not allowed</u>!

- Why are declarations needed? They don't make anything usable to the compiler!
  - C++ supports <u>separated compilation of cpp-files</u>, so each cpp-file must be compile-able on its own.
  - For compilation it is often enough for the compiler <u>just to see a symbol declared</u>.
  - Later on, for the symbols, which were left only declared, the C++-linker must find an o-file with the definition.
    - Assume a definition of the same symbol was given in a h-file included by two cpp-files, the linker would see two definitions, which is not allowed.

- H-files usually contain variable and function declarations, type definitions and preprocessor-directives.

17

- <u>What is the difference between declaration and definition?</u>
- In principle there is only one difference: we can have multiple declarations of the same entity in the same translation unit, but no multiple definitions. Each entity can be defined only once, this is called <u>one definition rule (ODR)</u> in C++.
- A definition of a symbol implies the declaration of that symbol in C++.
- In C++, multiple variables of the same type can be defined/declared in one statement.
- Some compilers call their o-files obj-files.

C++ Syntax Cornerstones – Part VIII – Syntactic Style and free Form

- The C++ syntax allows a lot of freedom as far as syntactical formatting of code is concerned.

- Spacing: C++ allows to add as many whitespaces around syntactic elements as we want to, the semantics stays the same.
    - A whitespace is usually just a blank space, i.e. the character we get, when we hit the space-key.

Good to know
Whitespaces, esp. spaces are sometimes called "blanks".

    - In a broader sense esp. vertical tab and linefeed are also considered as being whitespace characters.
    - In sounds academic, but it boils down to this: All three statements are equivalent for the compiler, i.e. have the same semantics:

    ```
    std::cout<<3 + 4<<std::endl;
    ```
    ≙
    ```
    std::cout << 3 +4<<std :: endl;
    ```
    ≙
    ```
    std ::cout<<
            3    + 4<<  std::endl;
    ```

    - Also these statements are all equivalent for the compiler:

    ```
    if (answerIsOk)
    {
        std::cout<<3 + 4<<std::endl;
    }
    ```
    ≙
    ```
    if    (  answerIsOk)     { std::cout<<3 + 4<<std::endl; }
    ```
    ≙
    ```
    if     (  answerIsOk) {
        std::cout << 3 +4<<std :: endl;
    }
    ```

    - We will discuss the meaning of the if-statement and the usage of braces (as so called blocks) in short.

- C++ is a so called free form language: The same syntax with the same semantics but different formatting.

- Freedom is good! But chaos is not! If each programmer would follow its taste on free formatting, we'll end in chaos!
    - Just compare the snippets above, were I really went crazy on formatting freedom...   18
    - On the bottom-line, programmers agree upon so-called coding conventions to define rules for free from languages.

- An <u>expression</u> is <u>like a mathematical term</u>: "something, that yields a value".

```
// Expression:
3 + 4
```

- A <u>statement</u> is a <u>set of expressions</u> to take effect, it doesn't need to yield a value.
  - Statements are like <u>phrases</u>, <u>sentences</u> or <u>commands</u>.
  - Individual statements need to be <u>terminated with a semicolon</u>.

```
// Statement:
std::cout<<3 + 4<<std::endl;
```

- A <u>block</u> is a <u>set of statements</u> within curly braces ({}).
  - C++ code is written in blocks, this is <u>a main C++ style feature</u>.

```
if (answerIsOk)
{
    std::cout<<3 + 4<<std::endl;
}
```

- Blocks fringe <u>method and type definitions</u>, <u>scopes</u> and <u>control structures</u>.
  - <u>Control structures</u> must be <u>cascaded</u> to code meaningful programs.
    - I.e. control structure blocks must be cascaded!
  - <u>Cascaded blocks should be indented to enhance readability!</u>
    - <u>We should use common conventions for indenting and bracing!</u>

```
// BSD style:
if (answerIsOk)
{
        // In the block.
}
```

```
// We'll use the 1TSB style:
if (answerIsOk) {
        if (answerIsOk) {
                // In the cascaded
                // if-block.
        }
}
```

- Blocks are important for <u>control structures</u>.
  - Now we're going to see blocks in action with if/else- and switch-statements.

19

- Blocks:
  - It is absolutely required to indent cascading blocks to make the code readable!
  - Empty blocks should be used instead of empty statements to denote empty control structures:

```
// Empty statement:
// (Not so good.)
if (answerIsOk);
```

```
// Empty statement, pair of empty braces:
// (We'll use this syntax in this course.)
if (answerIsOk) {
        // pass
}
```

- Bracing styles:
  - BSD (Berkley Software Distribution) style: should generally be used, when beginning programing, as it has a very good readability.
  - 1TSB style, which is at least basically also the K&R style (used in the legendary book "The C Programming Language"): "Only true bracing style", it is used in this course, as this style saves some lines (it is a "line-saver" style). It is the only style allowing to write correct JavaScript code to avoid mistakes because of semicolon insertion.
  - There exist many more bracing styles, which is often a source of coding wars among programmers, therefor we need coding conventions to cool down the minds.
  - Within blocks we can use a free syntax in C++, but please adhere to coding conventions.
    - In C++, blocks are mandatory for do-loops, try-blocks and catch-clauses.

C++ Syntax Cornerstones – Part X – Conditional Code

- Now we are going to discuss <u>control structures</u> to write <u>conditional code</u>.

- C++ code is executed <u>synchronously</u>: execution is performed <u>from one statement to the next in the order they are written</u>.
    - Synchronous execution means, that <u>one statement needs to complete execution, before the next statement can start execution</u>.
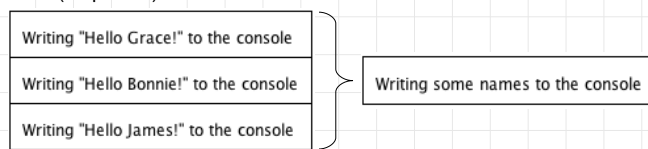
```
// Writing some names to the console:
std::cout<<"Hello Grace!"<<std::endl;
std::cout<<"Hello Bonnie!"<<std::endl;
std::cout<<"Hello James!"<<std::endl;
```

    - C++ allows writing <u>multiple statements into a single line</u>, but each statement <u>needs an individual semicolon</u> at its end!
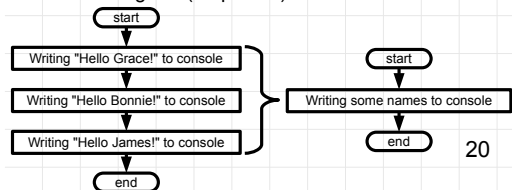
```
std::cout<<"Hello Grace!"<<std::endl; std::cout<<"Hello Bonnie!"<<std::endl; std::cout<<"Hello James!"<<std::endl;
```

- As a matter of fact, what we see here is the <u>most fundamental</u> <u>control structure</u>: the <u>sequence (of statements)</u>.

- Sequences (sequence of statements, or "operations") are written as simple <u>boxes</u> in NSDs and flowchart diagrams.

NSD (sequence)

Writing "Hello Grace!" to the console
Writing "Hello Bonnie!" to the console
Writing "Hello James!" to the console

Writing some names to the console

Flowchart Diagram (sequence)

```
    ( start )
       ↓
┌─────────────────────────┐
│ Writing "Hello Grace!" to console │
└─────────────────────────┘
       ↓
┌─────────────────────────┐
│ Writing "Hello Bonnie!" to console │
└─────────────────────────┘
       ↓
┌─────────────────────────┐
│ Writing "Hello James!" to console │
└─────────────────────────┘
       ↓
    ( end )
```

```
    ( start )
       ↓
┌─────────────────────────┐
│ Writing some names to console │
└─────────────────────────┘
       ↓
    ( end )
```

20

C++ Syntax Cornerstones – Part XI – Conditional Code

- <u>Control structures</u> are expressed as <u>syntactic elements</u>, which define how the <u>logic flow of the algorithm executes</u>.
    - If we do not use any syntactic element to define this flow, we'd end up with the just presented control structure of a sequence.
    - Now its time to discuss the <u>conditional flow/execution</u> of code.
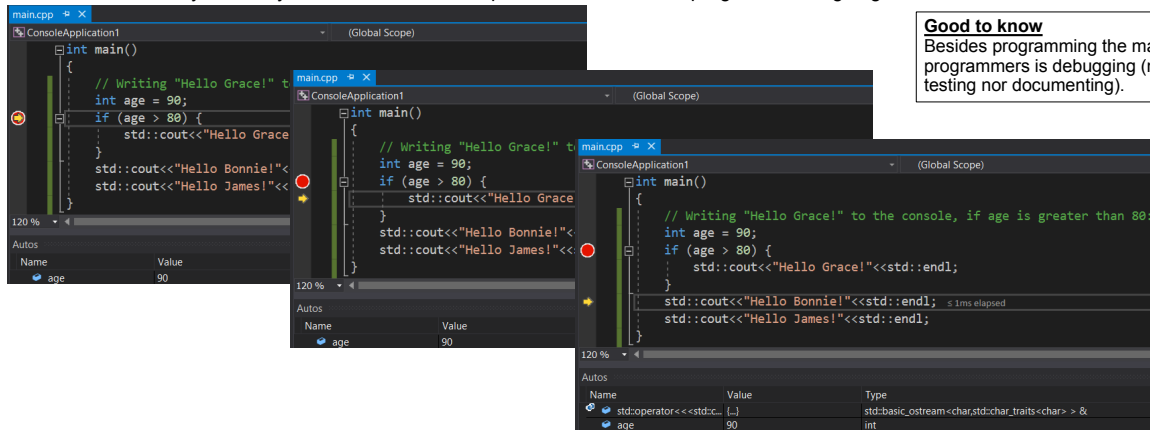
- if statements allow to <u>execute statements or sequences under a certain condition</u>:

```
// Writing "Hello Grace!" to the console, if age is greater than 80:
int age = 90;
if (age > 80) {
        std::cout<<"Hello Grace!"<<std::endl;
}
std::cout<<"Hello Bonnie!"<<std::endl;
std::cout<<"Hello James!"<<std::endl;
```

    - An if statement awaits a <u>conditional expression</u> in parentheses (age > 80) and a <u>sequence of statements written in a block</u>.
        - The if statement branches the logic of the code, so that the code in the block is executed conditionally.
        - In this case "Hello Grace!" and "Hello James!" will be printed to the console <u>always</u>, because the <u>value of *age* is 90</u>, <u>which is greater than 80</u>!
        - To distinguish the conditional block in a clear way from the unconditional code, the <u>conditional block's statement (or sequence) is indented</u>.
    - The statements, <u>following the if statement's block</u> <u>will not be executed conditionally</u>!
        - In this case "Hello Bonnie!" and "Hello James!" will be executed <u>always</u>, the value of *age* doesn't matter!
    - An important point to mention here: C++' <u>if statements read very similar to a english prosaic text</u>.
        - Here we clearly see the quality of C++ as high level language: the if-syntax exactly matches the semantics of a spoken language!

- What we see here is another fundamental control structure: <u>branching statements</u>.

C++ Syntax Cornerstones – Part XII – Conditional Code

- C++ supports two <u>syntactical basic constructs</u>: <u>sequence</u> and <u>cascade</u> (using blocks).

- Most IDEs support to <u>execute a program stepwise</u> among so called <u>breakpoints</u>, this is called <u>debugging</u>.
  - This is a very neat way to understand how sequential and cascaded program flow is going forward.


Good to know
Besides programming the main work of programmers is debugging (neither testing nor documenting).

  - During debugging, <u>the line which is selected</u>, <u>is the next line, which will be executed in the program</u>!

Java Syntax Cornerstones – Part XIII – Conditional Code

```
// Writing "Hello Grace!" to the console, if age is greater than 80:
int age = 90;
if (age > 80) {
      std::cout<<"Hello Grace!"<<std::endl;
}
std::cout<<"Hello Bonnie!"<<std::endl;
std::cout<<"Hello James!"<<std::endl;
```

- if statements are written as <u>tip-down triangles</u> in NSDs and as <u>diamonds</u> in flowchart diagrams:

NSD (if)
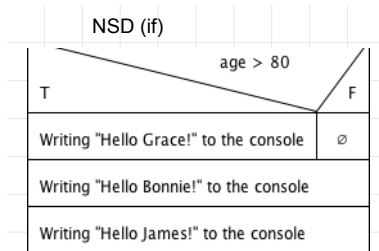
Flowchart (if)
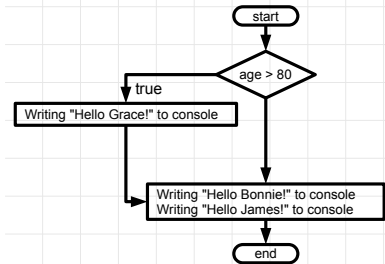
- These notations show, how the <u>flow of control</u> during program execution is virtually <u>forked</u> <u>following the condition</u>.
- The "wing" of the decision elements, which the flow of controls follows, if the decision is met is <u>marked with "T"/"true"</u>.

23

C++ Syntax Cornerstones – Part XIV – Conditional Code

- In order to widen our understanding of if statements, we'll take a look at <u>cascaded if statements</u>:

```cpp
int age = 90;
int countOfGreetingsForBonnie = 0;
if (age > 80) {
    std::cout<<"Hello Grace!"<<std::endl;
    if (countOfGreetingsForBonnie <= 0) {
        std::cout<<"Hello Bonnie!"<<std::endl;
    }
}
std::cout<<"Hello James!"<<std::endl;
```
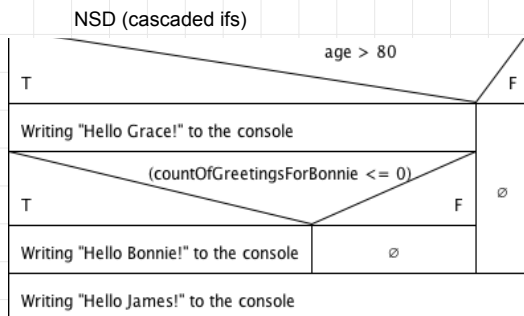
  – This snippet writes "Hello Grace!" if *age* is greater than 80 and "Hello Bonnie!" if *countOfGreetingsForBonnie* is less than or equals 0.

  Good to know
  In C++ we notate the mathematical ≤ condition as "<=" and ≥ as ">=".

  – As can be seen, the <u>cascaded relation of the if statements</u> is expressed by the <u>cascaded blocks</u> and by <u>cascaded block-indentation</u>.

  – In C++ we'll have to write <u>a lot of code</u> in this cascading style using <u>cascaded blocks</u>!

- The cascaded if statements are reflected as <u>cascaded tip-down triangles and diamonds</u> in NSDs and statechart diagrams:

NSD (cascaded ifs)

Flowchart (cascaded if)

24
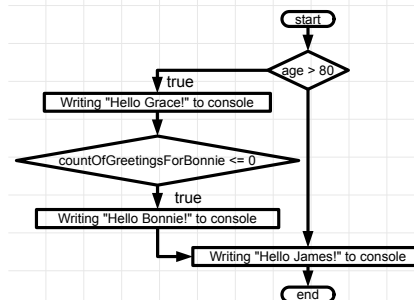
C++ Syntax Cornerstones – Part XV – Conditional Code

- This code writes "Hello Grace!" to the console only conditionally, but always "Hello Bonnie!" and "Hello James!":

```cpp
// Writing "Hello Grace!" to the console, if age is greater than 80:
int age = 90;
if (age > 80) {
    std::cout<<"Hello Grace!"<<std::endl;
}
std::cout<<"Hello Bonnie!"<<std::endl;
std::cout<<"Hello James!"<<std::endl;
```

- If we want to print "Hello Bonnie!", only if *age* is <u>not greater</u> than 80, we could program it like so:

```cpp
// Writing "Hello Grace!" to the console, if age is greater than 80:
int age = 90;
if (age > 80) {
    std::cout<<"Hello Grace!"<<std::endl;
}
// Writing "Hello Bonnie!" to the console, if age is not greater than 80:
if (age <= 80) {
    std::cout<<"Hello Bonnie!"<<std::endl;
}
std::cout<<"Hello James!"<<std::endl;
```

  - "Hello James!" is always written to the console, because it is <u>unconditional code</u>, i.e. it is <u>not contained in an if-block</u>!

- Well, this solution is somewhat cumbersome, because <u>we have to code the negated ("if not") condition of greater than</u>.

- In programming a <u>conditional pair "if"/"if not"</u> is <u>a very common situation</u>, therefor C++ provides <u>if/else statements</u>.

25

C++ Syntax Cornerstones – Part XVI – Conditional Code

- Now it's time to re-formulate our last conditional code example with if/else statements:

```
// Writing "Hello Grace!" to the console, if age is greater than 80:
int age = 90;
if (age > 80) {
        std::cout<<"Hello Grace!"<<std::endl;
}
// Writing "Hello Bonnie!" to the console, if age is not greater than 80:
if (age <= 80) {
        std::cout<<"Hello Bonnie!"<<std::endl;
}
std::cout<<"Hello James!"<<std::endl;
```
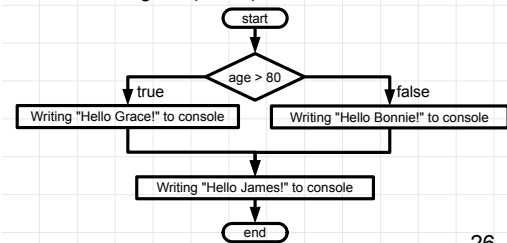
```
// Writing "Hello Grace!" to the console, if age is greater than 80,
// else write "Hello Bonnie!":
int age = 90;
if (age > 80) {
        std::cout<<"Hello Grace!"<<std::endl;
} else {
        std::cout<<"Hello Bonnie!"<<std::endl;
}
std::cout<<"Hello James!"<<std::endl;
```

- Conditional code with if/else statements can also be expressed as NSDs and statechart diagrams:

NSD (if/else)

age > 80	
T	F
Writing "Hello Grace!" to the console	Writing "Hello Bonnie!" to the console
Writing "Hello James!" to the console	
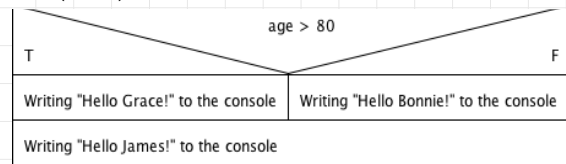
Statechart Diagram (if/else)

26

  - This time the "wings" of the decision elements each are explicitly marked with the values "T"/"true" and "F"/"false".

C++ Syntax Cornerstones – Part XVII – Conditional Code

- Let's discuss <u>yet another constellation</u>:
  - write "Hello Grace!" if *age* is greater than 80,
  - if not, write "Hello Bonnie!" if *greetingsForBonnie* is less than or equals 0,
  - if not, write "Hello James!".

- Meanwhile we can write code to express this constellation with a set of cascaded if and else statements:

```cpp
int age = 90;
int greetingsForBonnie = 0;
if (age > 80) {
    std::cout<<"Hello Grace!"<<std::endl;
} else {
    if (greetingsForBonnie <= 0) {
        std::cout<<"Hello Bonnie!"<<std::endl;
    } else {
        std::cout<<"Hello James!"<<std::endl;
    }
}
```

- However, there is one <u>new aspect in this cascaded code</u>: we cascaded an if statement in <u>another else</u> block.
  - I.e. no if statement in <u>another if block</u>.

- Such constellations appear so <u>often</u>, that C++ programmers usually <u>condense</u> the cascading to if/<u>else if</u>/else.   27
  - Now we'll discuss how this condensed syntax works.

C++ Syntax Cornerstones – Part XVIII – Conditional Code
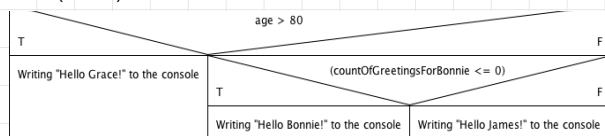
- Here it is. We can condense another if statement within an else block to an else if statement:

```
int age = 90;
int greetingsForBonnie = 0;
if (age > 80) {
        std::cout<<"Hello Grace!"<<std::endl;
} else {
        if (greetingsForBonnie <= 0) {
                std::cout<<"Hello Bonnie!"<<std::endl;
        } else {
                std::cout<<"Hello James!"<<std::endl;
        }
}
```
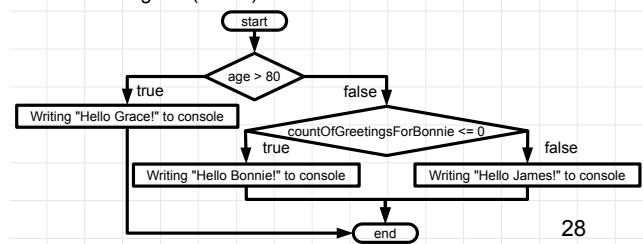
```
int age = 90;
int countOfGreetingsForBonnie = 0;
if (age > 80) {
        std::cout<<"Hello Grace!"<<std::endl;
} else if (greetingsForBonnie <= 0) {
        std::cout<<"Hello Bonnie!"<<std::endl;
} else {
        std::cout<<"Hello James!"<<std::endl;
}
```

- Conditional code with else if statements can also be expressed as NSDs and statechart diagrams with cascading forks:

Statechart Diagram (else if)

NSD (else if)

age > 80		
T		F
Writing "Hello Grace!" to the console	(countOfGreetingsForBonnie <= 0)	
	T	F
	Writing "Hello Bonnie!" to the console	Writing "Hello James!" to the console

```
start
   │
   ▼
age > 80
 true ◄──┴──► false
   │              │
Writing "Hello Grace!" to console   countOfGreetingsForBonnie <= 0
                          true ◄──┴──► false
                           │              │
              Writing "Hello Bonnie!" to console   Writing "Hello James!" to console
   │
   ▼
  end
```

28

- For <u>some control</u> structures, e.g. if/else, C++ allows to <u>leave braces away</u>, if <u>a belonging to block has only one statement</u>.

```
// (1) meets our conventions: recommendable!
if (age > 80) {
      std::cout<<"Hello Grace!"<<std::endl;
} else if (greetingsForBonnie <= 0) {
      std::cout<<"Hello Bonnie!"<<std::endl;
} else {
      std::cout<<"Hello James!"<<std::endl;
}
```

≙

```
// (2) omitted braces: not recommendable!
if (age > 80)
      std::cout<<"Hello Grace!"<<std::endl;
else if (greetingsForBonnie <= 0)
      std::cout<<"Hello Bonnie!"<<std::endl;
else
      std::cout<<"Hello James!"<<std::endl;
```

≙

```
// (3) omitted braces and line breaks: not recommendable!
if (age > 80) std::cout<<"Hello Grace!"<<std::endl;
else if (greetingsForBonnie <= 0) std::cout<<"Hello Bonnie!"<<std::endl;
else std::cout<<"Hello James!"<<std::endl;
```

- A <u>word of warning from the trenches</u>: <u>Generally use braces/blocks</u>, <u>even if it is not required</u> (only one statement in a block)!
  - Bracing allows the <u>best readability</u>, while <u>omitting braces is very error prone</u>! Also experienced programmers mess it up!
  - Many well-known coding conventions <u>explicitly forbid to leave braces away</u>!
  - => In this course: We'll generally use braces!

- <u>There is one exception</u>: this recommendable formatting of if/else plus else if does already omit braces actually:

```
// (1) meets our conventions, recommendable!
if (age > 80) {
      std::cout<<"Hello Grace!"<<std::endl;
} else {
      if (greetingsForBonnie <= 0) {
            std::cout<<"Hello Bonnie!"<<std::endl;
      } else {
            std::cout<<"Hello James!"<<std::endl;
      }
}
```

≙

```
// (2) else if, maybe better readability, also recommendable!
if (age > 80) {
      std::cout<<"Hello Grace!"<<std::endl;
} else if (greetingsForBonnie <= 0) {
      std::cout<<"Hello Bonnie!"<<std::endl;
} else {
      std::cout<<"Hello James!"<<std::endl;
}
```

29

- According to if/else: We should always use blocks! Matching if/else pairs are clear to the reader when we use blocks. This avoids the "dangling else" problem:

```
if (a == 1)
        if (b == 1)
                a = 42;
else
        b = 42;
```

  - Some developers think, that the else belongs to *if (a == 1)*, but this is not the case! – It belongs to its <u>nearest if</u> (i.e. *if (b == 1)*). Consequent usage of braces and indentation avoids the dangling else "optical illusion".
- The programming language Perl explicitly disallows leaving away braces on any control flow statement. – This is a bit strange restriction in Perl, because Perl allows a lot of (error prone) freedom in other places of the language.

- Assume this constellation:
  - if *billsAge* is less than or equal to 10, Bill will get a birthday gift worth 50€, else he'll get a birthday gift worth 75€.
  - No problem with if/else:

```
int billsAge = 12;
int birthdayGiftPrize = 0;
if (billsAge <= 10) {
    birthdayGiftPrize = 50;
} else {
    birthdayGiftPrize = 75;
}
```

- There is a clever way to abbreviate such a if/else constellation with a compact expression, the conditional expression:

```
int billsAge = 12;
int birthdayGiftPrize = (billsAge <= 10) ? 50 : 75; // (1)
```

  - When statement (1) is executed and *billsAge* is less than or equal to 10 then *birthdayGiftPrize* is initialized to 50, otherwise to 75.

- Conditional expressions apply the conditional operator **?:**.

```
// Conditional expression:
condition ? expression1 : expression2;
```

  - The conditional operator is the only ternary operator, i.e. it accepts three arguments.
  - The arguments are *condition*, *expression1* and *expression2*.
  - If *condition* evaluates to true *expression1* is evaluated, otherwise *expression2*.

30

- The operator ?: is more difficult to debug than if/else, because it does not consist of alternative statements to be executed.

C++ Syntax Cornerstones – Part XXI – Conditional Code

- Conditional expressions allow writing <u>conditional code without statements</u>, i.e. <u>without blocks and cascading</u>.
  - But, sometimes programmers want the indented style of if/else with the conditional operator. So, let's just reformat the code:

```
int birthdayGiftPrize = (billsAge <= 10) ? 50 : 75;
```
→
```
int birthdayGiftPrize = (billsAge <= 10)
                           ? 50
                           : 75;
```

- In opposite to if-statements, it is not required to put parentheses around the condition:

```
int birthdayGiftPrize = (billsAge <= 10) ? 50 : 75;
```
→
```
int birthdayGiftPrize = billsAge <= 10 ? 50 : 75;
```

  - But in most cases, parentheses are not bad and should be kept for clarity.

- Now let's assume, that we have to deal with more variants of ages and prizes for birthday gifts:
    - (1) If the person's age is 10 years, the gift may cost 50€ and for an age of 15 years it may cost 75€.
    - (2) For all other ages, the gift may only cost 25€. We can code this with a simple piece of conditional code:

```cpp
int personsAge = 10;
int birthdayGiftPrize = 0;
if (personsAge == 10) {
        birthdayGiftPrize = 50;
} else if (personsAge == 15) {
        birthdayGiftPrize = 75;
} else {
        birthdayGiftPrize = 25;
}
```

    - The specialty of this code: the comparison is only based on the equality of constant int values, even literal ints in this example.
    - In C++ we use the comparison operator "==" to express equality expressions, not "="!

- Alternatively we can use the switch statement to execute statements conditionally on mutually exclusive int constants:

```cpp
switch (personsAge) {   // Switch over the personsAge variable ...
    case 10:            // against the constant 10 or ...
        birthdayGiftPrize = 50;
        break;
    case 15:            // the constant 15 or ...
        birthdayGiftPrize = 75;
        break;
    default:            // all other ages.
        birthdayGiftPrize = 25;
}
```

32

- According to switch:
    - Floating point values cannot be used.
    - In C/C++, switch can only be used with integral types.
    - In C/C++, switch statements do not need braces surrounding all cases and the default-case needs no extra break statement.
    - switch statements are very popular in Win32 programming. Esp. traditionally programmed Win32 message loops make excessive use of switch statements.
    - (Opinion of [NLu]: Don't use switch! It looks temptingly simple, but is rather unstructured and can introduce goofy bugs. Also it leads to longer function bodies. It should be used only in the most low level code.)
        - Here two examples: (Everything is valid C++ code!)

```cpp
switch (argc) // Valid in C/C++: switch needs no braces at all
        default:
                if (is_prime(argc))
                        process_prime(argc);
                else
                        process_nonprime(argc);
```

```cpp
switch (pc) // Valid in C/C++: mix of inter-cascaded control
            // structures with switch
case START:
        while (true)
                if (!lchild()) {
                        pc = LEAF;
                        return true;
case LEAF:
                        while (true)
                                if (sibling())
                                        break;
                                else
                                        if (parent()) {
                                        pc = INNER;
                                        return true;
case INNER:
                                                ;
                                        } else {
                                        pc = DONE;
case DONE:
                                                return false;
                                        }
                }
```

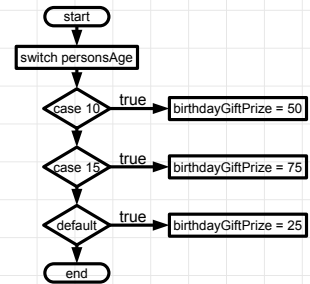# C++ Syntax Cornerstones – Part XXIII – Conditional Code

- Before we continue discussing the switch statement, let's show its NSD and flowchart representation.

```cpp
switch (personsAge) { // Switch over the personsAge variable ...
    case 10:          // against the constant 10 or ...
        birthdayGiftPrize = 50;
        break;
    case 15:          // the constant 15 or ...
        birthdayGiftPrize = 75;
        break;
    default:          // all other ages.
        birthdayGiftPrize = 25;
}
```

NSD (switch)

	personsAge	
10	15	default
birthdayGiftPrize = 50	birthdayGiftPrize = 75	birthdayGiftPrize = 25

Flowchart (switch)

33

- The readability of switch statements is really good and the mechanics should be clear as well:

```
switch (personsAge) {  // Switch over the personsAge variable ...
    case 10:              // against the constant 10 or ...
        birthdayGiftPrize = 50;
        break;
    case 15:              // the constant 15 or ...
        birthdayGiftPrize = 75;
        break;
    default:              // all other ages.
        birthdayGiftPrize = 25;
}
```

- The switch statement
  - does switch among a set of branched code blocks,
  - equality compares the statement's input (the "candidate"), that is passed as parameter to switch (*personsAge*), against a set of constants,
  - and jumps to a labeled case section, often called case-label, if the evaluated switch expression matches to the case constant.
  - The (optional) break statement limits the conditional code belonging to case-label, similar to an if's block.
  - If none of the case-labels met, the code after the optional default-label will be executed.
  - The order of case-labels doesn't matter, because the candidate will only match one of them.
  - The switch statement works with int values (and enum values).

- In fact, switch looks temptingly simple, but one can introduce goofy errors, because of switch's dangerous parts.
  - We'll discuss these parts now.

34

- Currently (C++17), the order of case-labels in switch doesn't matter. However in future versions of switch in C++, more complex expressions could be used than just comparison against constants – a feature called pattern matching. With pattern matching the order of case-"labels" will matter, because the candidate could match to more than only one of them.

- Let's assume another variant of the *birthdayGiftPrize* calculation: if the person gets 10 or 15 years old it may cost 75€.
  - We can modify the switch statement to a switch statement with case-fall through to cover this variant:

```
switch (personsAge) {  // Switch over the personsAge variable ...
    case 10:             // against the constant 10 or ...
        birthdayGiftPrize = 50;
        break;
    case 15:             // the constant 15 or ...
        birthdayGiftPrize = 75;
        break;
    default:             // all other ages.
        birthdayGiftPrize = 25;
}
```

```
switch (personsAge) {  // Switch over the personsAge variable ...
    case 10:             // against the constant 10 or ...
    case 15:             // the constant 15 or ...
        birthdayGiftPrize = 75;
        break;
    default:             // all other ages.
        birthdayGiftPrize = 25;
}
```

- Problems with switch: actually it is not simple at all and the readability is also not so good! – switch is a noisy statement.
  - switch statements with case-fall through are potentially dangerous:
    - Fall throughs are introduced by just leaving away the break statement, which formally belonged to a specific case statement.
    - Where the break statement is left away, case-labelled code will fall through. i.e. a break statement then belongs to more than one case statement.
    - From another perspective, if one unintentionally forgets to write the break statement, the control flow will surprisingly fall through!
    - This is a common source of bugs in Java and C/C++.
  - Other problems with switch statements:
    - Actually they are more complicated than if/else: they use case, break and default statements and allow application of fall throughs.
    - Inflexible: (1) Only equality comparison is supported, i.e, no other conditional expressions. (2) Only int- and enum-values can be handled.

35

- => The switch statement will be avoided in the examples used in this course!

- Refactoring code from using switch to if/else can be error prone.
- Good, when used with only returns instead of any breaks in the case sections.
- If an oo-design is used to solve a problem, all used switch statements are at least a smell, that the oo-design was not consequent here. In many cases the case-labels represent "type-codes", which could be expressed as abstracted types and polymorphic algorithms, which might lead to a cleaner design.

- The <u>region</u>, in which a <u>variable has a meaning and can be applied/used</u> is called <u>scope</u>.
  - C++ <u>limits the scope</u> of a variable by the <u>positioning in the code</u> and by <u>blocks</u>.

- Example: the <u>scope of a variable in an if-statement's block</u>:

y's scope ----
```
int x = 78; // Define x in the outer scope.
if (x == 78) {
    std::cout<<x<<std::endl; // Fine! x is defined before if's scope/block, i.e. in the outer scope.
    int y = 56; // Define y in the inner scope.
    std::cout<<y<<std::endl;  // Fine! y is defined in if's scope/block, i.e. in the inner scope.
}
std::cout<<y<<std::endl; // Invalid! y is not known/has no meaning in the outer scope, it was defined in if's scope!
```

- Example: the <u>scope of a variable in a scope-limiting block</u>:

y's scope ----
```
int x = 78; // Define x in the outer scope.
{
    std::cout<<x<<std::endl; // Fine! x is defined before the scope-limit block, i.e. in the outer scope.
    int y = 56; // Define y in the inner scope.
    std::cout<<y<<std::endl;  // Fine! y is defined in the scope-limit block, i.e. in the inner scope.
}
std::cout<<y<<std::endl; // Invalid! y is has no meaning in the outer scope, it was defined in the scope-limit block!
```

- Example: the definition of a variable must be <u>unique in a scope, if not we get a name clash, resulting in a compile time error</u>:

```
if (x == 78) {
    int y = 23;
    int y = 50; // Invalid! ODR violation: variable y is already defined …
}
```

36

- <u>What is a scope?</u>
  - A scope defines an area of code, in which a variable has certain meaning.
  - Other example: The scope of a variable in JavaScript is the scope of the function, in which the variable is defined. – There are no other sub scopes (e.g. curly braces). Instead JavaScript's variables are said to be <u>hoisted</u> from a nested scope to the function's scope.

- Values, e.g. those held in variables, have types: All values in C++ have a certain type! This is a profound concept of C++,

- Fundamental types are types, which are "built into" C++, whose values can be represented with literals.
  - We already know the fundamental type int. int variables can be created with integer literals:

    `int intValue = 23;`
  - C++' fundamental types have own keywords: int, double, bool etc.
  - We'll learn other fundamental C++ types in short.

- Compound types are types, which are "more involved" (a topic, we'll discuss in future lectures):
  - 1. Array declarators of fundamental and user defined types.
  - 2. Pointer declarators of fundamental and user defined types types.
  - 3. User defined types (UDTs): enums, structs, bit-fields, unions, classes and typedefs
  - 4. Functions

- Types of variables (and member functions) can be qualified with const/volatile, we call those c/v-qualifiers.

- The size of types and objects can be retrieved with the sizeof-operator.
  - sizeof provides the size of its argument as multiple of sizeof(char), which is 1. Simply spoken, sizeof(char) ~ 1 byte.

37

- <u>What are fundamental types?</u>
  - These types are integrated into C++ (as keywords).
- <u>What is a literal?</u>
  - A literal is a value of specific type that can be written out in source code directly.
  - In a sense a literal is the opposite of a symbol.
- In C++11 we can define user defined literals.
- <u>What is "const" and "volatile"?</u>
  - If an object is declared volatile, the C/C++ compiler assumes that its value could change anytime by any thread of execution. – Accessing a volatile variable is interpreted as direct memory read/write w/o caching.
- <u>In which "unit" does the sizeof operator return its result?</u>
  - A *std::size_t* of value 1 represents the sizeof(char). The type *std::size_t* is defined in <cstddef>.
- The operator sizeof can be used for variables and types. In the latter form we are required to write the type argument for sizeof in parentheses.

- <u>We already mentioned variables</u>
  - C++' variables are used like those in algebra, but if we use only short names like 'x' and 'y' we'll run out of variable names soon.
  - We'll have to deal with the more variables, the bigger a program gets. Therefor it is needed to give variables meaningful names.

- <u>Names</u> of variables are also called the <u>identifiers</u> of variables.
  - Identifiers are <u>case sensitive</u>: *aValue* is not the same identifier as *aVaLuE*!
  - C++ keywords <u>mustn't be used as identifiers</u>.
  - Special characters (except '_' (underscore)) are not allowed in identifiers!

Good to know
The _ (underscore) is sometimes also called "underbar" or just "under" among developers.

  - Digits are not allowed as first character.
  - Identifiers may contain only ASCII characters, but no special characters (but _) no umlauts and no ideograms.

    ```cpp
    // Valid C++ identifiers:
    int _n = 23;          // with an underscore
    int count_ = 58;      // with an underscore
    int G_U_uis = 566;    // with mixed casing
    ```

    ```cpp
    // Invalid C++ identifiers:
    int n*c = 23;         // invalid special character * used
    int 2gone = 58;       // starts with a digit
    int void = 17;        // C++ keyword cannot be identifier
    int nKäufer = 2;      // with an umlaut
    ```

  - The identifiers of variables must be <u>unique in the same scope</u>:

    ```cpp
    if (x == 78) {
        int y = 23;
        int y = 50; // Invalid! ODR violation: variable y is already defined …
    }
    ```

    - Yes, we already mentioned this, when we discussed "scopes".

38

- All identifiers are case sensitive.
  - <u>What's an identifier?</u>
  - Use *PascalCase* for types.
  - private, protected, public and local identifiers should be written in *camelCase*.
  - For identifiers of variables, there also exist the <u>Hungarian notation (HN)</u>. HN mandates to put a specific type-dependent prefix in front of a variable name, f for float (*farea*), i for int and (e.g. *iage*) sz (e.g. *szname*) for "string zero terminated". HN is called Hungarian notation, because its inventor, Charles Simonyis is Hungarian. HN is used when programming the Win32 SDK, but esp. the .NET Framework Design Guidelines prohibits using HN for other (non-Microsoft) developers (however, the guidelines make no statement about private fields). The problem using HN is, that the name of such variables must be changed every time their types change, and when developing oo, changing of a variable's type is quite common. But HN can be useful:
    - It could make sense, when using a dynamically typed language (but I doubt this).
    - With Win32 programming it makes sense, because a lot of handle-types and constants are just ints with a typedef, whose semantics might not be understood by functions (it is just an int…). HN can help here, because one can spot passing wrong arguments at least visually, if variables carry prefixes to tell handle-types from ints.
- For compile time constants, we should use *SCREAMING_SNAKE_CASE*, sometimes also called *MACRO_CASE*, is a variant of snake_case. – We will discuss constants in short.
- <u>What makes up a variable?</u>
  - A name, a type and a value.
  - Variable names should not be short names like in maths, rather use meaningful names. – Alas in the examples of this course often short variable names will be used. In maths, variables often apply identifiers following some rules:
    - Most identifies are reduced to only one letter of the alphabet.
    - The first letters of the alphabet are often used to name known values. E.g. *a*, *b*, *c* and *d* to name the sides of a 4-polygon.
    - The last letters of the alphabet are often used to name unknown values. E.g. *x*, *y* and *z* to name parameters of a function.
    - The letters "in between" are often used to name natural numbers, esp. *n* and *m*, but also *i* and *j* to name indexes.
    - Angles are named using greek lower case letters.
- Declaration, definition and initialization:
  - <u>What's that?</u>
  - A declaration introduces a symbol; <u>a declaration of a symbol can be repeated in code</u>.
  - A definition reserves memory for a value of a symbol; <u>a definition of a symbol mustn't be repeated in code!</u>
  - <u>What is a symbol?</u>
    - <u>Symbols are identifiers with a special meaning in code (e.g. identifiers of variables, constants or functions).</u>
  - An assignment sets the value of a symbol of a variable; <u>assignments of variables can be repeated in the code.</u>
  - An initialization combines the definition of a symbol with the assignment of a value. <u>An initialization of a symbol mustn't be repeated in the code!</u>

- Some special identifier-patterns mustn't be used as identifiers.
  - Identifiers mustn't start with underscore+upper-case-letter or contain a double-underscore:

    `int _Eva = 34; // Invalid!`   `bool may__be = true;  // Invalid!`

- Free identifiers can be put into namespaces to built some kind of own scopes.

- Because of all the freedom we have with identifier-naming, we need to tame the freedom somewhat with some conventions:
  - Convention, or coding convention means, that we've to agree upon a common notation for the identifiers we introduce in programs.
  - The most important identifiers in C++ are those for variables, functions and types.

- For variable names, we use *camelCase* for naming as convention! Camelcase means:
  - (1) names of variables consist of full words
  - (2) if the name consists of more than one word, all words are jammed together and each word starts with an upper case letter, but
  - (3) the very first word is always written in lower case.
  - (4) No separators like '_' are allowed.

```
// Examples for the camelCase notation:
int age = 90;
int birthdayGiftPrize = 75;
int countOfGreetingsForBonnie = 0;
```

39

- The special underscore rules are used for (future) reserved names in C/C++. The "__"-prefix is usually used for non-standardized compiler extensions.

C++ Syntax Cornerstones – Part XXX – Comments

- In most of the examples we have already used comments.

- Comments allow to write all kinds of human readable annotations into the code, without introducing syntax errors.
    - Comments can be applied everywhere in code. They are simple to spot in our examples, because they're highlighted in green.
    - But besides the highlighting, there is of course a syntax concept behind comments in C++, there are even two sorts of comments:
        - (1) Single line comments, which make text from the beginning of "//" until the end of the line a comment:

        ```
        // comment
        std::cout<<r * r * 3.14<<std::endl; // Explanation: Prints the area of the circle to the console.
        ```
        - (2) Multiline comments as comment brackets "/*" and "*/", which makes all enclosing text, even if spanning multiple lines, a comment:

        ```
        /* comment
         Explanation: Prints the area of the circle to the console:*/ std::cout<<r * r * 3.14<<std::endl;
        ```

- Esp. when beginning programming, we should use comments to be able to understand code when looking at it after a while!
    - E.g. the explanation of the console output in the examples above, explaining the formula, can prove useful.

- Technically, comments are ignored by the compiler. The compiler handles comments like whitespaces.
    - "Ignored" means, that the compiler will leave comments in the code, the compiler never modifies the code, but ignores its content.
    - => Seen from another perspective, this means, that valid C++ code can be "deactivated" by making it a comment:

    ```
    // This statement will not be compiled and thus also never be executed:
    //std::cout<<r * r * 3.14<<std::endl;
    ```

40

- Esp. programming newbies should use comments very often to remember what the code does. Later in the life as programmer, produced code should document itself.
- Multiline comments are often called C-style comments, whereas single line comments are often called C++-style comments (they were introduced with C++).
- Multiline comments act like a set of parentheses that enclose the commented text. Comments will be converted into whitespaces before the preprocessing phase starts.
- Prefer: Single line comments concern only one line.
- Avoid multiline comments as they can not be cascaded. The first occurrence of /* is always matched with the first occurrence of */.
- Virtually, comments are not that good, because they never change!
    - Writing good comments is difficult, a comment should not just repeat what the code in a verbose manner.
    - We as programmers have to change comments along with changes in the code.
    - Better than comments is self describing code!
    - Also commented code can be a problem for refactoring. Because in most cases commented identifiers are not refactored, when identifiers are renamed via a refactoring tool.

Constants – Part I

- Sometimes, we have to deal with the <u>same value over and over again</u>!

- 1. We can use <u>literal values</u>, e.g. the value 3.14 (pi) calculating a circle's area:

  ```
  double a = r * r * 3.14;
  ```

  - Here <u>we can't use int values/variables</u>, because pi is a <u>floating point number</u>! Floating point numbers are of type <u>double</u> in C++!
    - So, 3.14 is <u>double literal</u>, whereby the '.' represents the <u>decimal point</u> (not the ',', as we find it in, e.g., Germany).
  - Will we <u>remember the meaning</u> of the literal 3.14 in that expression/formula? Such literals are called <u>magic numbers</u>.
  - <u>Magic numbers should be avoided, as we could forget their meaning</u> and <u>other readers may not understand their meaning!</u>

- 2. We can use a <u>variable</u> for pi to give the value's variable a <u>memorable name/identifier</u>:

  ```
  // The double variable PI:          // Better memorable, eh?
  double PI = 3.14;                    double a = r * r * PI;
  ```

  - <u>But we can not prevent programmers from assigning to variables, replacing the value 3.14 with something different!</u>

- But we can solve these problems with the introduction of <u>constants</u>.

  ```
  PI = 4; // Ouch! Changes the meaning of PI!
  ```

- 3. In C++ we define constants as <u>variables with the new const keyword</u>.

```
// Constant double (compile time constant) PI:
const double PI = 3.14;
```

```
// This line remains valid:
double a = r * r * PI;
```

  - Such <u>symbols that can not be modified</u> are called <u>constants</u>.

```
PI = 4; // Invalid! PI is a constant, not a variable!
```

Good to know
// In C++ we do not need to define PI explicitly. The constant
// M_PI is <u>already defined</u> in the h-file <cmath>
// (_USE_MATH_DEFINES must be #defined as well):
double a = r * r * M_PI;

- Constants prevent us from doing <u>coding errors</u> and provide <u>self documentation</u>.
  - Constants are like variables, <u>which cannot be assigned to more than once</u>, <u>they replace magic numbers perfectly</u>.

- The constant PI is a so called <u>compile time constant</u>. A … What?
  - Well, the <u>constant value of *PI*</u> is <u>known to the compiler</u>.
    - => PI was initialized with the literal 3.14, and <u>the value of that literal is known at compile time</u>.
  - The value of a compile time constant must be explicitly set for exactly once in its lifetime, if not we'll get a compile time error.
    - => Well, after PI was set to 3.14, it can never be set to another value.
  - For compile time constants, we use *SCREAMING_SNAKE_CASE* for naming as <u>convention</u>, e.g.:

```
// Approximate speed of light in km/s in vacuum (traditional syntax):
const double SPEED_OF_LIGHT = 300000.0;
```

```
// Approximate speed of light in km/s in vacuum (C++ standard syntax):
double const SPEED_OF_LIGHT = 300000.0;
```

C++11 – type inference for constants:
const **auto** SPEED_OF_LIGHT = 300000.0;

Good to know
// We can also use conditional initialization for constants:
const double diameter = weHaveMonday ? 4.56 : 87.933;
// This cannot be done with if/else!

42

- The alternative C++ syntax to define constants, i.e. basically switching the name of the type with keyword const, does better match the general C++-syntax of constants for compound types, e.g. const pointers, or oo constness, e.g. const member functions.

Constants – Part III

- We can also define <u>run time constants</u> in C++.
  - Those are constants, whose <u>values are "fixed" at run time</u>, i.e. not already at compile time.
    - (Mind, compile time happens before run time.)
  - A run time constant is defined, by using const as before, but initializing it with a value, which is <u>not known at compile time</u>.
  - It sounds complicated, but is really easy to get:

    ```
    // rectangleArea as run time constant:
    const double rectangleArea  = a * b;
    ```

    - In this statement, the variables *a* and *b* have no constant values, maybe they were input from the user.

- Sometimes (compile time) constants are called <u>symbolic constants</u> to separate the wording from literals (literal constants):

  ```
  // 3.14 is just a literal (constant) here:
  double a = r * r * 3.14;
  ```
  ```
  // PI is now a symbolic constant:
  double a = r * r * PI;
  ```

- Another way to define constants are <u>enumerations (enums)</u>, which we will discuss in a future lecture.

- Yet another way to define compile time constants are <u>macros</u> created with preprocessor-#defines.
  - However, using const or enums is the better way to do it.

C++ Fundamental Integral Datatypes

- int, long, short, char/wchar_t and bool

```
// Definition and initialization of an int:
int numberOfEntries = 16;
```

Good to know
The term "integer" is for "value of integrity", i.e. a non-divisible value.

- Integral types default to be signed, can be marked to be unsigned with the keyword unsigned.

- Signed types generally use the <u>two's complement</u> representation in memory.
  - So the range of the value domain is broken into a negative and a positive wing.
  - The 0 (zero) counts as positive value.

Good to know
The 0 is sometimes written as 0̷ (slashed zero) or 0̇ (zero with point). In earlier days of computing it was required to tell the letter O from the digit 0, but the display and printer resolution was not very high and fonts were monospaced, therefor slashed zero and zero with point were introduced.

- Literals and sizes:
  - char {'A', 65, 0x41, 0101}; 1 = sizeof(char), at least 1B underneath
  - int/short {42, -0x2A, 052}; sizeof(char) ≤ sizeof(short), at least 2B ≤ sizeof(int)
  - long {42L, -0x2al, 052L}; sizeof(int) ≤ sizeof(long), at least 4B
  - wchar_t {L'A', 65, 0x41, 0101}; sizeof(char) ≤ sizeof(wchar_t) ≤ sizeof(long)
  - bool {true, false; <u>falsy</u>: 0; <u>truthy</u>: 42, -0x2A, 052}; 1 ≤ sizeof(bool) ≤ sizeof(long)

C++11 – new char types
char16_t c1 = u'c';
char32_t c2 = U'c';

C++14 – binary integer literal
int x = 0b00101001;

C++11 – min. 64b integer
long long x = 24LL;
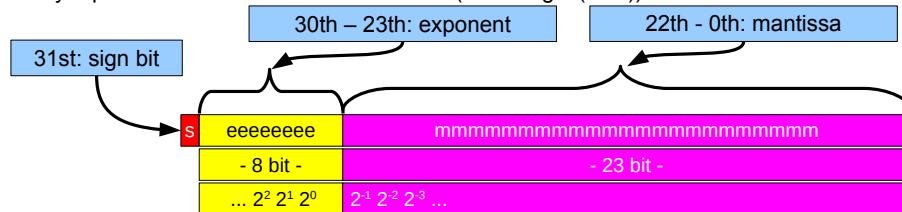
C++14 – digit separators
int x = 1'000'000;

- Integral data seems to be the most basic and important data type in programming. – Most basic programming lectures usually start using integral data.
- C99's <stdint.h> defines integer types of guaranteed size, <inttypes.h> defines integer types of specific size and
  <stdbool.h> defines an explicit boolean type.
- <u>What are "integral" types?</u>
- <u>What is the two's complement?</u>
  - Negative ints could be represented with the one's complement. The one's complement of a positive number is just the inversion of all the bits, which represent the number. – The problem with this approach is, that we'll end up with two representations of the 0, one for -0 and one for +0. To overcome this problem, we just add 1 to the one's complement, which makes the sign of the number clear, the result os the two's complement. – We will discuss this in more depth in a future lecture.
- We should always use int as our default integral type.
  - The unsigned is needed, if we have to deal with raw memory or bit-fields.
  - Never use unsigned as it is a source of nasty bugs in innocent looking code! – We'll revisit this topic in a later lecture.
  - Never use short or char as long as there is no compelling reason. – They will be automatically <u>promoted to int</u> in all expressions, safe sizeof or taking their address.
  - We should never use long, if we think int is too small. Because in this case long is mostly also becoming too small. – Better introduce a user defined type. – "int is the biggest (integral) type that is efficient" (John Lakos).
- Literals of type long should always be written with an upper case L to avoid "optical illusions", because a lower case l could be confused with the digit 1 depending on the editor's font.
- C++11 officially introduced long long with at least 8B for really large numbers, but we have to think twice if we want to use it (see above).
- wchar_t literals need to start with an upper case L!

- <u>What does the term "<u>double</u>" exactly mean?</u>
  - The double precision of float, the C/C++ type float has single precision.
- A double can represent bigger and more precise numbers than float.
- Floating point types try to display <u>rational</u> numbers of various precision.
- Institute of Electrical and Electronics Engineers (IEEE, pronounced as "i triple e").
- We should always use double as our default floating point type!
  - The hardware (FPU) is typically optimized for doubles.
  - APIs for embedded systems do often not support double precision. E.g. Open GL ES (ES for Embedded Systems) provides only float in its interfaces.
- We have to be aware that we have to use "." instead of "," in the floating point literals! -> Therefore these datatypes are called floating point datatypes!

Implicit Type Conversion of integral Types

- Type conversion: Under certain circumstances a value of a specific type can be used to <u>create a value of another type</u>.

- C++ supports <u>standard integral conversions</u>, which are <u>done implicitly, when values are assigned (or passed to functions)</u>.
  - <u>Non-lossy</u> implicit conversions are done <u>from a smaller to a larger integral type</u>:

```
// Some standard integral (implicit) conversions:
char c = 'k';
// c = 'k'
short s = c;
// s = 107
int i = c;
// i = 107
long l = i;
// l = 107L
```

Good to know
chars and shorts are implicitly promoted to int (or unsigned) in expressions.
Exceptions are taking the size (sizeof) and taking the address of char/short.

  - In the char-to-short-example, the conversion is non-lossy, because <u>the range of values of char fits into short</u>, this is called <u>widening</u>.

  - Conversions from char, signed char, unsigned char, short, unsigned short to int or unsigned int, are called <u>promotions, they guarantee not to change the value.</u>

- C++ also supports <u>lossy</u> standard conversions <u>from larger to smaller types</u>, e.g. long to int, this is called <u>narrowing</u>:

```
// Some standard integral (implicit) conversions:
long l = 107L;
// l = 107L
int i = l; // Narrowing conversion
// i = 107
```

  - Narrowing conversions can lead to <u>lost data</u> and <u>may change values</u>, so type <u>narrowing conversion in C++ is not safe!</u>   46
  - <u>Simply spoken, all conversions between fundamental related types (like long/int/short/char) are implicit and possibly not safe!</u>

- **int**-promotion is performed, because **int** is the biggest (integral) type that is efficient, and it is guaranteed that no information will be lost.
- Esp. **int** to **long** is not an integral promotion, but an integral standard conversion (this is the only one that is guaranteed to be non-lossy).

Implicit Type Conversion of floaty Types

- C++ supports <u>standard floaty conversions</u>, which are <u>done implicitly, when values are assigned (or passed to functions)</u>.
  - <u>Non-lossy implicit conversions</u> are done <u>from a less precise to a more precise floaty type</u>:

    ```
    // Some standard floaty (implicit) conversions:
    float f = 1.85F;
    // f = 1.85F
    double d = f; // Promotion
    // d = 1.85
    ```

    - float is promoted to double in this case. Floating-point promotions are different, as floats are not always promoted to double.

- C++ also has <u>lossy</u> standard conversions from floaty <u>types to less precise floaty types</u>, i.e. <u>narrowing</u>, e.g. double to float:

    ```
    // Some standard floaty (implicit) conversions:
    long double ld = 1.85L;
    // ld = 1.85L
    double d = ld; // Narrowing conversion
    // d = 1.85
    float f = d; // Narrowing conversion
    // f = 1.85F
    ```

    - Narrowing conversions can lead to <u>lost precision or data</u> and <u>may change values</u>, so type conversion in C++ <u>is not safe</u>!

    - <u>Simply spoken, all conversions between fundamental related types (like long double/double/float) are implicit and possibly not safe!</u>

Explicit Type Conversions

- (Implicit) standard conversions work also between integral and floaty types:

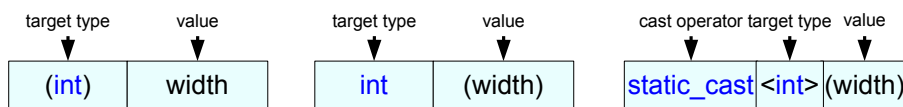 | // Standard (implicit) conversion: | // Standard (implicit) conversion: |
  |---|---|
  | double d = 25; | int i = 2.78; // Narrowing conversion |

  - We can assume from these examples, that implicit widening as well as narrowing (i.e. lossy) conversions are allowed.
  - Yes, esp. those implicit narrowing conversions are not safe, because we loose precision and possibly data.

- In opposite to implicit standard type conversions, C++ also supports explicit conversions between unrelated types.
  - Explicit type conversion is needed to convert between "unrelated" types, e.g. from int* to int), or more relevant, from void* to int*.
  - We haven't yet discussed those types, therefor, we'll show it by expressing standard conversions as explicit conversions.

- Explicit type conversions are expressed with so-called cast-operators.

// Standard (implicit) conversion:	// Explicit conversion with casts:
int i = 2.78;	int i = (int)2.78; // C-style cast
	int j = int(2.78); // function-style cast
	int k = static_cast<int>(2.78); // C++-cast/new-style cast

  target type   value   target type   value   cast operator target type value

  | (int) | width | | int | (width) | | static_cast | <int> | (width) |

  48

  - Although all cast-syntaxes have the same effect (semantics), we'll prefer C++-casts, i.e. static_cast.

- ## Note that conversions between floaty and integral types are standard conversions and no promotions!

- Multiplication of <u>integers is no problem</u>: the <u>largest type is the resulting type</u>.

- But the <u>division of integers does not have a floaty result</u>!

```
int x1 = 5;
int x2 = 3;
double result = x1 / x2; // result will just contain an int value expressed as double
// result = 1.0
```

  - The result is of the type of the <u>largest contributed integer</u>.
  - <u>Even if the result variable's declared type is floaty, the result is integral (*result* is declared to be a <u>double</u> above)</u>.
  - Also the <u>result won't be rounded, instead the places after the period will be clipped</u>!

- To correct it, <u>explicitly convert any of the integral operands to a floaty type with the <u>static_cast</u> operator</u>:

```
int x1 = 5;
int x2 = 3;
double result = static_cast<double>(x1) / x2;
// result = 1.6666666666666667
```

- <u>The result of the division by 0 is implementation specific.</u>

```
double result = 5.0 / 0;
// result = ?
int result2 = 5 / 0;
// Whatever ...
```

  - Floaty values divided by 0 may result in "not-a-number" (NaN) or infinity.
  - Integral values divided by 0 may result in a run time error (e.g. EXC_ARITHMETIC).
  - => <u>Code defensively</u>: We should <u>always check the divisor for being not 0</u> in our code <u>before dividing</u>.

- The integral result of the integral division makes sense, because how should the integral division know something about floaty types?
- Clipping: the result of 5 : 2 is 1.6, but the part after the period will be clipped away.
- The division by 0 is <u>not</u> "not allowed" in "ordinary" maths, instead it is "just" undefined.

- In some examples we already saw the application of text values, e.g. to write text to the command line.

- The type, which represents text in C++, is called <u>string</u>, more specifically <u>cstring</u>.
  - Cstrings are represented as "strings of characters", which are <u>arrays of char</u>s in C++ lingo.

> **Good to know**
> char is usually pronounced [ˈtʃɒr] not [ˈkɔr]!

- Initialization, literals and <u>literal concatenation</u>:

  ```
  char aName[] = "Arthur";
  ```
  ```
  const wchar_t* anotherName = L"Ford";
  ```
  ```
  char fullName[] = "Arthur " "and" " Ford"; // Concatenation of literals (also over multiple lines)
  ```

> **C++11 – new string literals/raw strings**
> char16_t s1[] = u"st";
> char32_t s2[] = U"st";
> char s3[] = u8"st";
> char rawString[] = R"(s\t)";

  The <u>assignment of cstrings isn't defined</u>, the function *std::strcpy()* must be used:

  ```
  // Erroneous:
  aName = "Marvin"; // Invalid!
  ```
  ```
  // Correct: Copy "Marvin" to aName:
  std::strcpy(aName, "Marvin"); // Ok!
  ```

- The concepts of cstrings are simple in theory, but hard in practice! Why cstrings are <u>hard to use</u>:
  - Esp. assignment and <u>non-literal concatenation</u> must be done with functions!
  - Cstring manipulation involves working with pointers and <u>raw memory</u>.
  - Cstrings are <u>0-terminated</u>, which makes them error prone as well...
  - => We'll discuss the concepts of <u>memory</u>, arrays and cstrings <u>in depth</u> in a future lecture!

50

- <u>What is an array?</u>
- Concerning the term "string" mind the German term "Zeichen<u>kette</u>", which means "string of characters".
- Cstrings can also be represented as arrays of wchar_t.
- <u>What is the difference between *aName* and *anotherName*?</u>
  - The contents of *aName* can be modified, the contents of *anotherName* not. *anotherName* is a "real" cstring (of type const char*).
- Btw: we can not assign cstrings, because they are arrays and arrays can't be assigned.
- The expression "*std::strcpy()*" is a so called function-call.
- <u>What does the prefix "*std::*" in front of the function name "*strcpy()*" mean?</u>
  - Every object of the C++ standard library is contained in the namespace *std*. The namespace name must be prefixed (incl. the scope operator "::") in front of the function name to find the specified function. More on this topic in future lectures.

Thank you!