Advanced Programming in C++

Kristof Van Laerhoven
kvl@eti.uni-siegen.de

```
#include <iostream>
#include <vector>

int main() {
  vector<string> msg {"Welcome", "to", "advanced", "C++"};
  for (const std::string & word : msg) {  // C++11 standard
    std::cout << word << ' ';
  }
  sdt::cout << '\n';
}
```

Contents

Contents

Week 6
Week 7
Week 8
Week 9

This course in a nutshell

In this course, you learn advanced themes in C++ programming

The course consists of:

- Lecture: 2h per week, basic concepts
- Lab: 2h per week, getting / correcting programming tasks
- Homework: ± 2h per week, solving assignments

This course in a nutshell

Links to lecture slides and assignments are available on moodle:

- https://moodle.uni-siegen.de/course/view.php?id=34345
- updates to the slides will be made available during the term

We learn C++ by doing, hence: *follow lectures and exercises*

Disclaimer:

This course's material was inspired by similar courses from colleagues Roland Wismueller (Uni Siegen), Hannah Bast (Uni Freiburg), Federico Busato (NVIDIA, Modern C++ Programming)

Examination and grading

See the description of this course here

Enroll for both lecture *and* exercises, as well as the course work (4INFMA307-S):

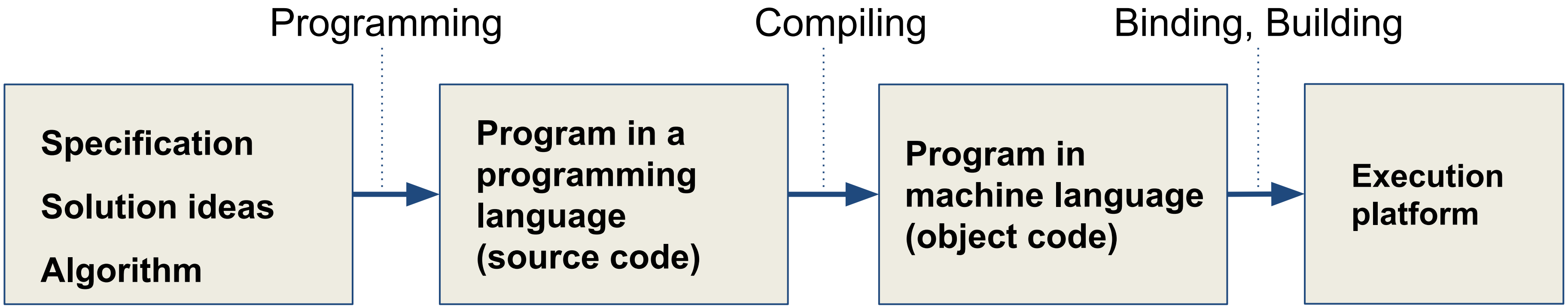
- Set of programming assignments during term: programming on paper
- These need to be delivered in-class => your presence required
- You need to pass this course work to be able to enroll for the exam

A 1-hour written exam (enroll more than 2 weeks before: 4INFMA307-P):

- 1 handwritten A4 (double-sided) page is allowed
- Bring a photo ID and a pen (blue/black ink only)
- Structure: Programming tasks *very similar* to the exercises

1. Designing programs

Steps in creating a program:



1. Designing programs

Steps in creating a program:

Specification
Solution ideas
Algorithm

→ Programming →

Program in a programming language (source code)
`program.cpp`

→ Compiling →

Program in machine language (object code)
`program.o`

→ Binding, Building →

Execution platform
`program`

Editor:
```
nano program.cpp
```

GCC compiler:
```
g++ program.cpp
```

```
./program
```

1. Designing programs

1.1. Program structure:

- A program is essentially a series of *machine instructions* that tell the system what to do, step by step, similar to a recipe
- Source code allows humans to formulate instructions in an understandable fashion, which then can be translated to machine instructions → In this course, we use C++ to create source code
  - C++ source code files are text files that end with **.cpp** or **.h**
  - A *compiler* then translates these to a program that consists out of machine instructions
- Creating source code needs a file system and operating system

1.2. Using GitHub Repository, g++, editors

We will use a github repository: https://github.com/kristofvl/AdvancedCPP as a code base for the slides, all exercises, and home works from the lecture

- You will need a github account
- It allows keep track of changes, and distributing larger projects

We'll use GNU g++. For Linux or macOS, you can directly use g++
In Windows, you can install Windows Subsystem for Linux (WSL), to have a Linux environment to use g++ inside it

1. Designing programs

1.2. Using GitHub Repository, g++, editors

You are free in the use of editor. Here are a few options:

Microsoft Visual Studio Code (VSCode) , Sublime , Lapce, Zed

Text-based coding editors: Vim, Emacs, NeoVim, Helix

Not suggested: Notepad, Gedit, and other similar editors

1. Designing programs

BDay.cpp

```
/* The Birthday Paradox -- an illustration
   Author: kvl,    Date: first week     */

int main() {

  return 0;
}
```

1. Every program should mention who programmed it

2. Every program needs a function "main" that contains all code to be executed

3. The function "main" will exit and send a number (usually zero, for "no error") to the operating system

1. Designing programs

BDay.cpp

```
/* The Birthday Paradox -- an illustration
   Author: kvl,   Date: first week     */
#include <iostream>
class BDay {
 /* p(x) --> probability of x happening,  p(x) = 1 - p(not x)
    p(2 persons have same birthday) =  1 - (365-1)/365 * (365-2)/365 * ... * (365 - (n-1))/365 */
 public:
  double prob(int n);
};
double BDay::prob(int n) {
  double p = 1.0;   // probability that out of n people, 2 have the same birthday
  for (int i = 0; i < n; i++) {  // i = 1, 2, ..., n-1
     p = p * (365-i)/365;
  }
  return 1 - p;
}
int main() {
  int n = 23;
  BDayP b;  // create an object to access the prob() method:
  std::cout << "The chance that 2 of " << n << " people have a same birthday is " << b.prob(n);
  return 0;
}
```

1.3. Compiling and building a program:

- Type `g++` to launch the GCC c++ compiler: `g++ BDay.cpp⏎`
  - `g++` creates the program, a file with the default name `a.out` in the current directory, which can be executed in the terminal: `./a.out⏎`
- Add `-o` to specify another name: `g++ BDay.cpp -o BDay⏎`
  - `g++` creates the file `BDay` , which can be executed: `./BDay⏎`
- The compiler tries to compile all other needed source files and needs a `main` function in the files that specifies what your program does
  - `g++` can compile multiple files, e.g.: `g++ ex1.cpp ex2.cpp⏎`
  - `g++` can compile code for later use in a program with `-c` :
    `g++ -c BDay.cpp⏎`, which creates `BDay.o`
    `g++ BDayx.o⏎` then creates the program `a.out`

1.3. Compiling and building a program:

- Add `--std` to specify which C++ standard your code is for. If for instance you use features for C++11 you need to add `--std=c++11`:

  ```
  g++ BDay.cpp -o BDay --std=c++11
  ```

  Features you use may be in different standards, and you can specify them by the option `--std` to let the compiler compile the source code in different standards

1.4. Including libraries:

- The compiler can include others' code as a library that is mentioned in the source code (for example `#include <iostream>`)
  - In this course, we will see mostly such *standard* libraries, `g++` knows where to search for their files and links these:
    `g++ BDaySimple.cpp`

BDaySimple.cpp

```
/* The Birthday Paradox -- a simplification */
#include <iostream>
int main() {
  std::cout << "The chance that 2 out of 23 people have";
  std::cout << " a same birthday is about 0.5 \n";
  return 0;
}
```

1.4. Including libraries:

- The compiler can include others' code as a library that is mentioned in the source code (for example ncurses: `#include <ncurses.h>`)
  - Other non-standard libraries need to be linked explicitly with `-l`:
    `g++ BDayCentre.cpp -l ncurses`

BDayCentre.cpp

```
/* The Birthday Paradox -- in the *middle*  */
#include <ncurses.h>  // draw text in terminal screen
int main() {
  initscr();     // initialize ncurses window
  mvaddstr(LINES/2,   COLS/3, "The chance that 2 out of 23 have");
  mvaddstr(LINES/2+1, COLS/3, "the same birthday is about 0.5. wow.");
  getch();       // capture the user's pressed key
  endwin();      // close the ncurses window
  return 0;
}
```

1.5. Indenting your code:

- To make everyone's code look the same, we recommend using [cpplint]
- We *also* recommend **2-space indentation** (see *all* examples in slideset):

```
void myFunction(bool exec, bool size) {  // indent after each {
◇◇int ret = 0;
◇◇if (exec) {  // indent after each {
◇◇◇◇for (int i = 0; i < size; i++) {  // indent after each {
◇◇◇◇◇◇ret += i;
◇◇◇◇}  // de-indent before each }
◇◇} else {  // indent after each {
◇◇◇◇ret = 12;
◇◇}  // de-indent before each }
}  // de-indent before each }
```

2.1. The basic components of a program:

- Reserved keywords
- Preprocessor directives
- Names
- Constants
- Operators
- Braces
- Separators
- Comments

```cpp
/* The Birthday Paradox -- in short */
#include <iostream>
int main() {
  std::cout << "The chance that 2 out of";
  std::cout << " 23 people have a same";
  std::cout << " birthday is about ";
  std::cout << 0.5 << "\n";
  return 0;  // 0 back to operating system
}
```

2.1.1. Reserved C++ keywords

Reserved keywords are words that are reserved for special meaning by the language standard and cannot be used as identifiers (names for variables, functions, classes, etc.). E.g.:

`bool`	`do`	`namespace`	`switch`
`break`	`double`	`new`	`this`
`case`	`else`	`private`	`true`
`catch`	`false`	`protected`	`using`
`char`	`float`	`public`	`virtual`
`class`	`for`	`return`	`void`
`const`	`if`	`short`	`while`
`delete`	`int`	`sizeof`	

2.1.2. Preprocessor directives

Source code can also contain so-called *preprocessor directives* that start with a `#` : We will use solely:

- `#include`, followed by a source file either:
  - surrounded by `<` and `>` , for example: `#include <ncurses.h>` for source code from standard libraries
  - surrounded by `"` and `"` , for example: `#include "myCode"` for source code in the current directory
- **header guards** to ensure that code is included only once:

```
#ifndef HEADERFILE
#define HEADERFILE
```

```
#endif
```

2.1.3. Names:

You can define names to variables, parameters, functions, etc.

- these names need to be unique (so no keywords)
- they can contain only letters, digits and underscores (_)
- they need to start with letters or an underscore
- their length is nearly unlimited
- examples:

correct	wrong	reason
`Sum`	`get Name`	no empty spaces
`getName`	`Ver2-1`	minus '-' not allowed
`_all4you`	`2exp4`	starts with a digit
`__1_2`	`while`	C++ keyword

2. Data: Variables, Types, Constants

2.1.4. Constants:

`15` (integer) `3.14159f` (float) `3.14159` (double)
`'p'` (character) `"brb"` (string) `true` (boolean)

2.1.5. Operators:

`+` `-` `*` `/` `&&` `||` `=` `==` `>=` `<=` `<<` `>>` `<` `>` …

2.1.6. Braces:

`(` `)` `[` `]` `{` `}`

2.1.7. Separators:

`,` `;` `.` ` ` (space) as well as tabs or new lines

2.1.8. Comments:

- `// comment till the end of the line`
- ```
  /* comment that spans across
     multiple lines */
  ```
- note that a multi-line comment cannot contain */:
  ```
  /* this example comment */ would
     cause an error */
  ```

```
// probability that out of n people, 2

/* p(x) --> probability of x happening
   p(x) = 1 - p(not x)
   p(2 persons have same birthday) =
…*/

/** The Birthday Paradox -- illustration
*   Author:  kvl
*   Date:    last week
*/
```

IMPORTANT: Comments should explain your code but never be trivial

good: `int scrMaxWidth; // maximum screen width`

bad: `float aspectRatio; // this variable holds the aspect ratio`
```

2.1. Example:

```
/* An interactive example */
#include <iostream>
int main() {
   char name[80];   // symbols array for the user's name
   std::cout << "Hi there, what's your name?" << "\n";
   std::cin >> name;   // read the name from terminal
   std::cout << "Welcome " << name;
   if (name[0] == 'K') {
      std::cout << ", I like your name!"
   }
   std::cout << "\n";
   return 0;   // return a zero
}
```

- comments
- preprocessor
- separators
- operators
- braces
- constants
- names
- keywords

2.2. Variables:

- represent *memory space*, where data of a certain type can be stored
- need to be declared and ideally initialized before use:
```
int keyPressCounter = 0;   // how often did user press a key?
```
- have values that after declaration can be 'read out' and changed:
```
if (keyPressCounter > 27)  // after 27 key presses,
  keyPressCounter = 0;     // we set it back to zero
```
- can't be changed afterwards, when declared (and initialized) as **constants**: `const int answer = 42; // after this, answer stays 42`
- live in a certain *scope*, typically the function in which it was declared. After this function ends, the variable is deleted from memory.

2.3. Data types: Integral

Native type	Bytes	Range	Fixed width types <cstdint>
bool	1	true, false	
char	1	see ASCII table	
signed char	1	-128 -- 127	`int8_t`
unsigned char	1	0 -- 255	`uint8_t`
short	2	-2^{15} -- $2^{15}-1$	`int16_6`
unsigned short	2	0 -- $2^{16}-1$	`uint16_t`
int	4	-2^{31} -- $2^{31}-1$	`int32_t`
unsigned int	4	0 -- $2^{32}-1$	`uint32_t`
long int	4/8		`int32_t / int64_t`
long unsigned int	4/8		`uint32_t / uint64_t`
long long int	8	-2^{63} -- $2^{63}-1$	`int64_t`
long long unsigned int	8	0 -- $2^{64}-1$	`uint_64_t`

2.3. Data types: Floating-Point

Native type	Bytes	Range	Fixed width types C++23 <stdfloat>
(bfloat16)	2	$\pm 1.18 \times 10^{-38}$ -- $\pm 3.4 \times 10^{38}$	`std::bfloat16_t`
(float16)	2	0.00006 -- 65536	`std::float16_t`
float	4	$\pm 1.18 \times 10^{-38}$ -- $\pm 3.4 \times 10^{38}$	`std::float32_t`
double	8	$\pm 2.23 \times 10^{-308}$ -- $\pm 1.8 \times 10^{308}$	`std::float64_t`

- See also IEEE754 for more information

2.3. Data types

- A type defines: what *values* the variable can have,
  how much memory is allocated for it, and
  which *operations* are possible
- All variables and constants have a type in C++
  - for constants, you can tell the type by their form
  - for variables, this is explicit in the declaration

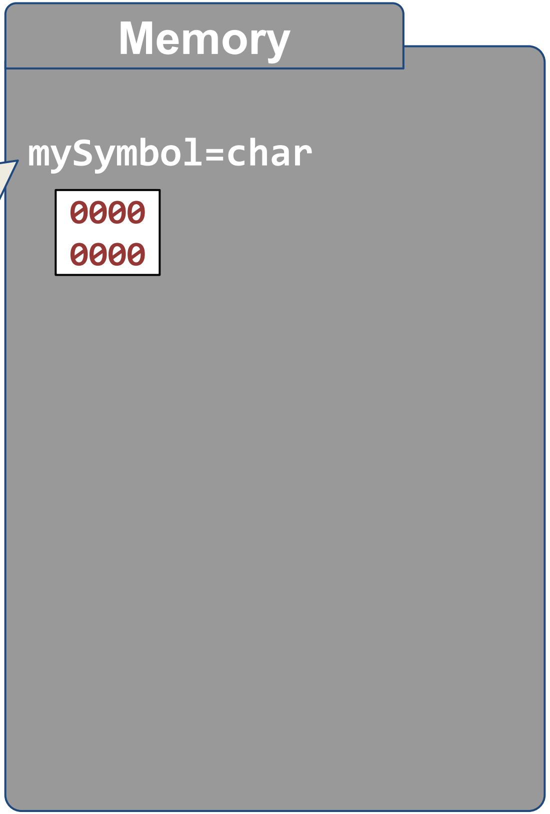
2.3. Data types: A (simplified) Memory View

```
/* reserving variables */
int main() {
  char mySymbol;    // store one character
  int myInteger;    // store an integer
  bool myBoolean;   // store a boolean
  float myFloat;     // store a floating point
  myInteger = 12;   // 12 = constant integer
  myFloat = 12.0f;  // 12.0f= constant floating point
  mySymbol = '@';   // '@' = constant character
  myBoolean = true;  // true = constant boolean
  return 0;
}
```

Memory

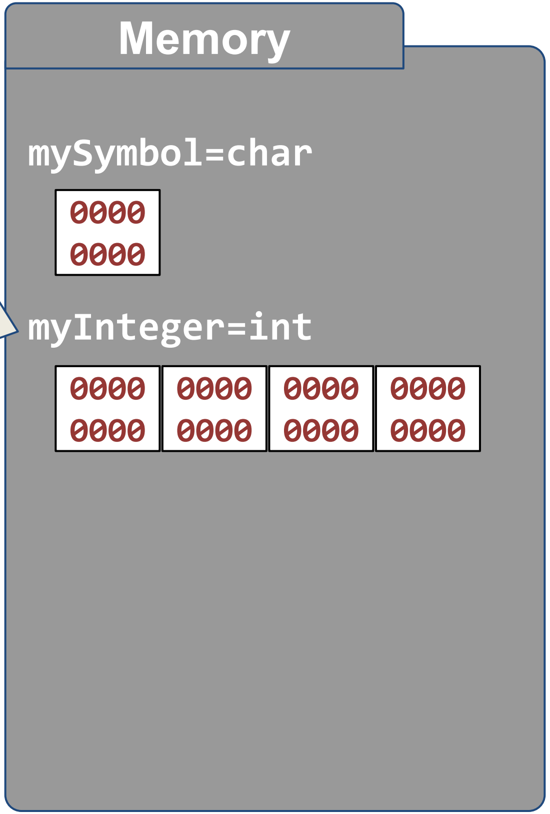
2.3. Data types: A (simplified) Memory View

```
/* reserving variables */
int main() {
   char mySymbol;    // store one character
   int myInteger;    // store an integer
   bool myBoolean;   // store a boolean
   float myFloat;     // store a floating point
   myInteger = 12;   // 12 = constant integer
   myFloat = 12.0f; // 12.0f= constant floating point
   mySymbol = '@';   // '@' = constant character
   myBoolean = true;   // true = constant boolean
   return 0;
}
```



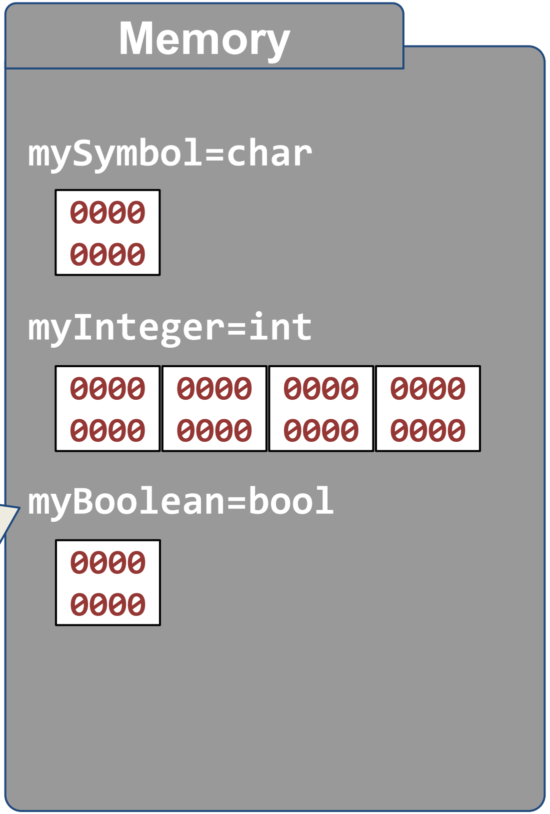
2.3. Data types: A (simplified) Memory View

```
/* reserving variables */
int main() {
  char mySymbol;    // store one character
  int myInteger;    // store an integer
  bool myBoolean;   // store a boolean
  float myFloat;     // store a floating point
  myInteger = 12;   // 12 = constant integer
  myFloat = 12.0f; // 12.0f= constant floating point
  mySymbol = '@';   // '@' = constant character
  myBoolean = true;   // true = constant boolean
  return 0;
}
```



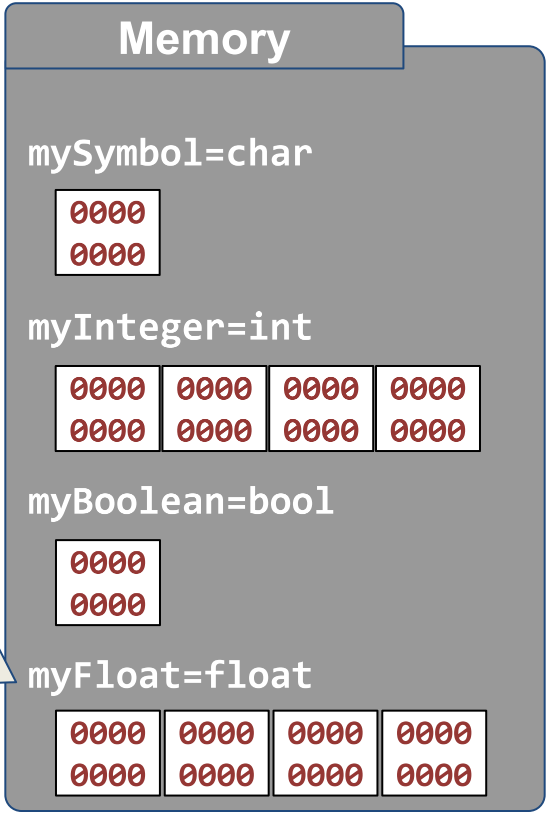
2.3. Data types: A (simplified) Memory View

```
/* reserving variables */
int main() {
  char mySymbol;    // store one character
  int myInteger;    // store an integer
  bool myBoolean;   // store a boolean
  float myFloat;     // store a floating point
  myInteger = 12;  // 12 = constant integer
  myFloat = 12.0f; // 12.0f= constant floating point
  mySymbol = '@';  // '@' = constant character
  myBoolean = true;  // true = constant boolean
  return 0;
}
```



2.3. Data types: A (simplified) Memory View

```
/* reserving variables */
int main() {
  char mySymbol;    // store one character
  int myInteger;    // store an integer
  bool myBoolean;   // store a boolean
  float myFloat;     // store a floating point
  myInteger = 12;   // 12 = constant integer
  myFloat = 12.0f; // 12.0f= constant floating point
  mySymbol = '@';   // '@' = constant character
  myBoolean = true;   // true = constant boolean
  return 0;
}
```



2.3. Data types: A (simplified) Memory View

```
/* reserving variables */
int main() {
  char mySymbol;    // store one character
  int myInteger;    // store an integer
  bool myBoolean;   // store a boolean
  float myFloat;     // store a floating point
  myInteger = 12;   // 12 = constant integer
  myFloat = 12.0f; // 12.0f= constant floating point
  mySymbol = '@';  // '@' = constant character
  myBoolean = true;  // true = constant boolean
  return 0;
}
```

Memory

mySymbol=char

0000 0000

myInteger=int

0000 0000	0000 0000	0000 0000	0000 1100

myBoolean=bool

0000 0000

myFloat=float

0000 0000	0000 0000	0000 0000	0000 0000

2.3. Data types: A (simplified) Memory View

```
/* reserving variables */
int main() {
  char mySymbol;    // store one character
  int myInteger;    // store an integer
  bool myBoolean;   // store a boolean
  float myFloat;     // store a floating point
  myInteger = 12;   // 12 = constant integer
  myFloat = 12.0f;  // 12.0f= constant floating point
  mySymbol = '@';   // '@' = constant character
  myBoolean = true;   // true = constant boolean
  return 0;
}
```

Memory

mySymbol=char

0000 0000

myInteger=int

0000 0000	0000 0000	0000 0000	0000 1100

myBoolean=bool

0000 0000

myFloat=float

0100 0010	0100 0000	0000 0000	0000 0000

2.3. Data types: A (simplified) Memory View

```
/* reserving variables */
int main() {
  char mySymbol;     // store one character
  int myInteger;     // store an integer
  bool myBoolean;    // store a boolean
  float myFloat;      // store a floating point
  myInteger = 12;    // 12 = constant integer
  myFloat = 12.0f;   // 12.0f= constant floating point
  mySymbol = '@';    // '@' = constant character
  myBoolean = true;   // true = constant boolean
  return 0;
}
```

Memory

mySymbol=char

0100 0000

myInteger=int

0000 0000	0000 0000	0000 0000	0000 1100

myBoolean=bool

0000 0000

myFloat=float

0100 0010	0100 0000	0000 0000	0000 0000

2.3. Data types: A (simplified) Memory View

```
/* reserving variables */
int main() {
  char mySymbol;    // store one character
  int myInteger;    // store an integer
  bool myBoolean;   // store a boolean
  float myFloat;     // store a floating point
  myInteger = 12;   // 12 = constant integer
  myFloat = 12.0f; // 12.0f= constant floating point
  mySymbol = '@';   // '@' = constant character
  myBoolean = true;  // true = constant boolean
  return 0;
}
```

Memory

mySymbol=char

0100 0000

myInteger=int

0000 0000	0000 0000	0000 0000	0000 1100

myBoolean=bool

0000 0001

myFloat=float

0100 0010	0100 0000	0000 0000	0000 0000

2.3. Data types: Constants

Constants for these type are visible from how they are written, mostly using a type-dependant ***suffix***:

- `int` whole numbers without suffix, e.g.: `729`, `-3628`, `0`, `-12632832`
- `unsigned int` `u` or `U`, e.g.: `56u`, `7126u`, `0U`
- `long int` `l` or `L`, e.g.: `523374631`, `-363433428L`
- `long unsigned` `ul` or `UL`, e.g.: `39ul`, `3637428UL`
- `long long int` `ll` or `LL`, e.g.: `52ll`, `1634428LL`
- `long long unsigned` `ull` or `ULL`, e.g.: `7ull`, `8428ULL`
- `float` numbers with decimal range and `f`, e.g.: `3.612f`, `5.2f`, `0.001f`
- `double` numbers with decimal range, e.g.: `3.612`, `5.2`, `0.001`
- `char` a character between single quotes, e.g.: `'?'`
- `bool` either `true` or `false`

2.3. Data types: Constants

Constants for C++23 float types use a type-dependant ***suffix***:

- `std::bfloat16_t` `bf16` or `BF16`, `3.21bf16`, `-12.345BF16`
- `std::float16_t` `f16` or `F16`, `3.21f16`, `-12.345F16`
- `std::float32_t` `f32` or `F32`, `3.21f32`, `-12.345F32`
- `std::float64_t` `f64` or `F64`, `3.21f64`, `-12.345F64`
- `std::float128_t` `f128` or `F128`, `3.21f128`, `-12.345F128`

Constants for number types can use a ***prefix*** for different representations:

- Binary (since C++14): `0b`, `0b10101010`
- Octal: `O`, `O308`
- Hexadecimal: `0x` or `0X`, `0xFA77DD`, `0X1A7F`

Since C++14, digit separators `'` can be used: `80'000'000` (=80000000)

2.3. Data types: Impact in memory

```c
/* reserving variables */
int main() {
  int sevenInteger = 7;      // seven as integer
  char sevenChar = '7';      // seven as character
  float sevenFloat = 7.0f;   // seven as float
  double sevenDouble = 7.0;  // seven as double
  return 0;
}
```

Memory

myInteger=int

0000 0000	0000 0000	0000 0000	0000 0111

sevenChar=int

0011 0111

sevenFloat=float

0100 0000	1110 0000	0000 0000	0000 0000

myFloat=double

0100 0000	0011 1000	0000 0000	0000 0000
0000 0000	0000 0000	0000 0000	0000 0000

2.3. Data types and variables

When you need a variable, you need to **declare** it first (and you can give them a value straight away, this is called **initialization**):

```
char mySymbol = '&';    // store one character and set it to '&'
int myNumber;           // store an integer, no initial value
bool areWeDone = false; // stores a boolean, set to false
float heightInMeters = 1.85f; // stores 1.85 as a float
double highPrecision;   // stores a double floating point number
```

Later, you can give variables **new values**:

```
areWeDone = true;    // we have now set this variable to true
highPrecision = 0.0;  // we have now set this variable to 0.0
```

Multiple variables can be declared at once: `int myNum = 3, yourNum = 8;`

2.3. Data types and variables

A first console output example: Initialize a variable to the symbol `a`, print its value in the console, then change its value to a `q` , then print it again.

```
#include <iostream>  // to allow use of std::cout and std::endl
int main() {
  char mySymbol;   // variable to hold a character
  mySymbol = 'a';
  std::cout << mySymbol << "\n";  // prints out "a" and endline
  mySymbol = 'q';
  std::cout << mySymbol << "\n";  // prints out "q" and endline
  return 0;
}
```

2.3. Data types and variables: **auto**

Since C++11, the **auto** keyword can be used to let the compiler *deduce* the type from its initialization:

```
auto mySymbol = '&';             // a char
auto myNumber = 2 + 2;           // sum of two ints -> int
auto areWeDone = 7 < 3;          // false -> bool
auto heightInMeters = 1.85f;     // float
auto highPrecision = 1.2345;     // double
```

The auto keyword can make code more maintainable and hide complexity:

```
for ( auto i = start; i < end; i++ ) …    // i has type of start
```

⚠️ Excessive use of type hiding typically makes code less readable.

2.3. Data types and variables: **decltype**

From C++11 onwards, the **decltype** keyword can be used to inspect the declared type of an entity or expression:

```
auto myNumber = 2 + 2;          // sum of two ints -> int
decltype(myNumber) otherNumber = myNumber + 3;  // -> int
decltype('?') mySymbol = '?';
```

STL-based (see later) ways to query type:

- The **typeid** operator can be used to identify the type:

```
typeid(myNumber).name() == 'i';  // returns true
```

- **std::is_same** can be used to check whether two type are the same:

```
std::is_same<int, std::int32_t>::value;  // returns mostly true
```

2.4. Variables and scope

- A variable only exists within a **scope** (block of code, usually between curly braces `{` and `}`) and is deleted after the scope ends (after a `}`)
- Variable names cannot use names that are already taken

```
{  // from here starts one scope:
    int a = 3, b = 1;
    {          // from here starts another scope:
        c = a + b;     // error: c doesn't exist here yet
        int c = 0;
        c += a;        // this works, c is defined here
    }
    b = c;             // error: c doesn't exist here
    double a = 2.1; // error: a already taken
}
```

2.5. Type conversions

- Variables can sometimes take values from other variables and constants that do not have the same type, **implicitly** converting them
- Sticking to **explicit conversion**, using the type in braces, is clearer:

```
double myDouble = 1.1;
int aNumber = 5;
char c = 'a';
bool b = true;
myDouble = (double)aNumber;  // this works: myDouble == 5.0
myDouble = (double)c;        // this works: myDouble == 97.0
aNumber = (int)myDouble;     // works:      aNumber == 97
c = (char)98.0f;             // works too:  c == 'b'
```

2.5. Type conversions

Example: Find out what the ASCII-codes are for the symbols '?', '&', and '#'

```
#include <iostream>  // to allow use of std::cout and std::endl
int main() {
   char question = '?';    // three variables to hold characters
   char ampersand = '&';    // variable to hold a character
   char hash = '#';    // variable to hold a character
   std::cout << (int)question << "\n";    // print ASCII code for '?'
   std::cout << (int)ampersand << "\n";  // print ASCII code for '&'
   std::cout << (int)hash << "\n";        // print ASCII code for '#'
   return 0;
}
```

3. Basic statements: if, switch, loops

3.1. Statements and assignments:

- Variables are the program's data, statements specify what to do with the data
- Statements *always* end with `;` and are executed sequentially
- Assignments *always* calculate the value of what is right of `=` and return this:

```
int x, y, mult;
bool b;
x = 2;                    // assignment returns 2
mult = 1;                 // assignment returns 1
y = x * 7 / 2;                  // y = ?      (try these out yourself
b = ( (x + y) >= 13 );          // b = ?       after learning about
x += -5;                        // x = ?       operators and
mult *= x - y;                  // mult = ?    priority in the next
b = mult != 1337;               // b = ?       slides)
```

3. Basic statements: if, switch, loops

3.1. Statements and assignments:

Operators that we will use in this course:

Arithmetic operators:	+	-	*	/	%	++	--	
Relational operators:	>	>=	==	!=	<=	<	? :	
Logical operators:	&&	||	!					
Assignment operators:	=	+=	-=	*=	/=	%=		
	<<=	>>=						
Three-way comparison operator:	<=>							

unary operators

binary operators

unary and binary operators

ternary operator

3.1. Statements and assignments:

Arithmetic operators (try out yourself):

```
int x=5, y=9, width, length, multiply, division, remainder;

width     = x + y;               // addition:       14
length    = width - 1;           // subtraction:    13
multiply = x * y;                // multiplication: 45
division = x / y;                // division:       0  (why?)
remainder = x % y;               // modulo (remainder after division): 5
x++;                             // increment: x = x + 1:  6
y--;                             // decrement: x = x - 1:  8
length = width = y;              // an assignment returns a value: 8
multiply = (x = 2) * (y = 5);    // (bad style) : 10
```

3.1. Statements and assignments:

Important to note:

- (Arithmetic) Operators do not exist for each type

  for example, `%` (modulo) does not exist for **`float`** or **`double`**

- Operators might have different behavior between types

```
int x = 7, y = 3, z;
z = x / y;        // note: z will get the value 2, since the
                  // operator '/' operates on two integers !
```

3.1. Statements and assignments:

Relational and logical operators (try out yourself):

```
double f1 = 15.2, f2 = 31.6;
bool b1 = true, b2 = false;
std::cout << (f1 == f1);                     // true  -> 1
std::cout << (b1 != b1);                     // false -> 0
std::cout << (b1 == true);                   // true  -> 1
std::cout << (f1 < f2);                      // true  -> 1
std::cout << (f1 <= f1);                     // true  -> 1
std::cout << ((f1 > f1) && (!b1));           // false -> 0
std::cout << ((b1 != b2) || (f2 >= f1) );    // true  -> 1
std::cout << (b1 || b2);                     // true  -> 1
std::cout << ( (f1 > 10.0) ? 'y' : 'n' );    // y
```

3.1. Statements and assignments:

Assignment operators (try out yourself):

```
float x = 5.1f, y = 9.2f;
int z = 7;

x = y + 1.0f;                  // results in x having the value 10.2f
y -= x;                        // y = y - x
x += 1.0f;                     // x = x + 1.0
y /= 2.0f;                     // y = y / 2.0
z %= 3;                        // z = z % 3 (note z is integer)
```

3.1. Statements and assignments:

Three-way comparison operator or spaceship operator `<=>` (C++20)

returns an *object* that can be directly compared with
a positive, a 0, or negative integer:

```
(3 <=> 5) == 0;        // false
('a' <=> 'a') == 0;    // true
(3 <=> 7) < 0;         // true
(7 <=> 5) < 0;         // false
```

3.1. Statements and assignments:

example 00 (difficulty level: 🌶️🌶️):

```
/**
 Try to figure out what is happening in the code below. Then try to compile the
 code, and then change the commented part so that the code's output becomes 1:
*/
#include <iostream>  // to allow use of std::cout and std::endl
int main() {
  auto i = 7;     // what type is variable i?
  auto j = 9.0;   // what type is variable j?

  bool ret = ( ( i <=> j) == 0 );  // change the '== 0' part so that the output is 1

  std::cout << ret << std::endl;
  return 0;
}
```

3. Basic statements: if, switch, loops

3.2. Priority

- Operators are not always executed from left to right, as statements
- **Use braces** to avoid having to learn the table below by heart:

Prio.	Operators	Associativity
11	`,`	left to right
10	`=`, `+=`, `-=`, `*=`, `/=`, `%=`, `? :`	right to left
9	||	left to right
8	`&&`	left to right
7	`<`, `>`, `<=`, `>=`	left to right
6	`<<`, `>>`	left to right
5	`+`, `-`	left to right
4	`*`, `/`, `%`	left to right
3	prefix `++`, prefix `--`, unary `-`, unary `+`, `!`, (type), **`new`**, **`delete`**	right to left
2	suffix `++`, suffix `--`, `.`, ->,	left to right
1	`::`	left to right

3.2. Priority

- Examples:

```
int a = 2, b = 3, c = 4;
a = b * c + d;        // a = ((b * c) + d)
a /= b * c % d;       // a = a / ((b * c) % d)
a += b = c + d;       // a = a + (b = (c + d))
```

Beware of prefix and postfix increment / decrement operators:

```
a = 10;
b = ++a;
// a = 11, b = 11
```

```
a = 10;
b = a++;
// a = 11, b = 10
```

3.3. Blocks

- Statement sequences are executed one by one, from left to right:
  `length = 10; width = 15; surface = length * width;`
- A block of statements thus implements a function (e.g., between the curly braces `{` and `}` for main) and can be used anywhere where a statement can appear. But beware that variables defined there only 'live' in the block:

```
int main() {
  int length, width, surface;
  {
    length = 10; width = 15;
    surface = length * width;
  }
  return 0;
}
```

```
int main() {
  int length = 10, width = 15;
  {
    int surf = length * width;
  }
  return surf; // error: surf unknown
}
```

3.4. Selection statements: `if` and `switch`

- Depending on a condition, selection statements can either execute the next statement, or not
- Simply for one case:

```
if ( number < 0 )     // if number is negative
  sign = '-';         // then the sign is '-'
```

- For both cases:

```
if ( number < 0 )     // if number is negative
  sign = '-';         //  then the sign is '-'
else                  // otherwise:
  sign = '+';         //  the sign is '+'
```

3.4. Selection statements: **`if`** and **`switch`**

Conditions often use relational operators:

```
int x, y, sum, counter, minimum;

if (x > y) …                // is x bigger than y?
if (x >= y) …               // is x bigger or equal than y?
if (sum == x + y) …         // is sum equal to (x+y) ?
if (x != y) …               // is x different from y?
if (counter < 100) …        // is counter smaller than 100?
if (x + 1 <= 1 - y) …       // is (x+1) smaller or equal to (1-y)?

minimum = (x < y)? x : y;   // minimum gets a new value of ...
                            // if ( x < y ), then x, else y
```

3.4. Selection statements: **`if`** and **`switch`**

Conditions also often use logical operators:

```
int x, y, counter;
bool readFlag, writeFlag;

if (readFlag || writeFlag) …    // logical OR: readFlag or
                                // writeFlag need to be true,
                                // will be skipped if both are false

if ((x != 0) && (y / x < 5)) … // logical AND: both y/x needs to be
                                // smaller than 5 and x shouldn't be 0

if (!(x < 0)) …   // NOT: the same as: if (x >= 0)
```

3.4. Selection statements: `if` and `switch`

Nesting `if` statements:

```
if (number == 0)
  sign = 0;
else
  if (number > 0)
     sign = +1;
  else
     sign = -1;
```

Alternatively, using a nested `( ? : )` operator:

```
sign =  (number == 0) ? 0 : ( (number > 0) ? +1 : -1 );
```

3.4. Selection statements: **`if`** and **`switch`**

Nesting **`if`** statements can be tricky, **use curly braces**

```
if ( x == y )
  if ( y == z )
     allEqual = true;
else
     allEqual = false;
```

```
if ( x == y ) {
  if ( y == z )
     allEqual = true;
}
else
     allEqual = false;
```

```
if ( x == y )
  if ( y == z )
     allEqual = true;
  else
     allEqual = false;
```

```
if ( x == y ) {
  if ( y == z )
     allEqual = true;
  else
     allEqual = false;
}
```

3.4. Selection statements: **`if`** and **`switch`**

Nesting many **`if`** statements:

```
if (menuItem == 1) {
         ...
} else if (menuItem == 2) {
         ...
} else if ((menuItem == 3) ||
           (menuItem == 4)) {
         ...
} else if (menuItem == 5) {
         ...
} else {
         ...
}
```

=

Using one **`switch`** statement:

```
switch (menuItem) {
  case 1:  ...
           break;
  case 2:  ...
           break;
  case 3:
  case 4:  ...
           break;
  case 5:  ...
           break;
  default: ...
           break;
}
```

3.5. Loops

3.5.1. The `while` loop:

```
int i, sum;
sum = 0;
i = 1;
while ( i < 7 ) {   // this block
  sum += i;         // is executed
  i++;              // repeatedly
}
```

result:

sum: `0, 1, 3, 5, 9, 14, 20`

3.5.2. The `do while` loop:

```
int i, sum;
sum = 0;
i = 1;
do {                // this block
  sum += i;         // is executed
  i++;              // repeatedly

} while ( i < 7 );
```

result:

sum: `0, 1, 3, 5, 9, 14, 20`

3.5. Loops

3.5.3. The **for** loop:

```
int i, sum = 0;
for ( i = 0; i <= 10; i++ ) {   // this block
  sum += i;                     // is executed repeatedly
}
```

Template:

initialization of loop variable

do we loop again?

in- or decrease the loop variable

```
for (  statement ; condition ; statement  )
     <statement>  ;
```

statement, or block of statements to repeat

3.5. Loops: **break** and **continue**

- **break** terminates the loop, the rest of the loop body will not be executed:

```
int num = 10;
while (num > 0) {
    if (num == 5) break;  // stop after num has reached 5
    num--;
}
```

- **continue** does not terminate the loop, but just skip the rest of the loop body:

```
int num = 10;
while (num > 0) {
    if (num == 5) continue;  // when num == 5, don't decrement
    num--;
}
```

3.5. Loops: When do we use which loop type?

- `for` loop:
  - for a given range (e.g. ” for all i = 1 … N “)
  - when you automatically need a loop variable
- `while` loop:
  - when the (maximal) number of repetitions is not known beforehand
  - when the repetition conditions are more complex
- `do while` loop:
  - when a block needs to be repeated at least once

3.5. Loops: example 02 (difficulty level: 🌶️)

```
/**
  Write a program that prints out a series of numbers, starting at 120.0 and where
  each next number is seven less than the previous one. Stop once the number is
  smaller than 43.7
  */
#include <iostream>  // to allow use of std::cout and std::endl
int main( ) {




  return 0;
}
```

3.5. Loops: example 03 (difficulty level: 🌶️🌶️)

```
/**
  Write a program that asks the user for a number, and then prints out this number
  in the terminal, followed by the half of the previous number until
  the result is smaller than ten. So for 100 it would give out: 100, 50, 25.5, 12.25
  */
#include <iostream>  // to allow use of std::cout and std::endl
int main( ) {





  return 0;
}
```

3.5. Loops: example 04 (difficulty level: 🌶️🌶️)

```
/**
  Write a program that counts from 131 down till 23, one number per line in the
  the terminal, and prints out "hop", if the number is a multiple of 7.
  */
#include <iostream>  // to allow use of std::cout and std::endl
int main( ) {






  return 0;
}
```

3.5. Loops: example 05 (difficulty level: 🌶️🌶️🌶️)

```
/**
 Write a program that prints in the terminal all prime numbers from 3 till 99.
 Remember: A number is a prime when any division by a smaller number results in
 a remainder that is never zero.
*/
#include <iostream>  // to allow use of std::cout and std::endl
int main( ) {




  return 0;
}
```

3.5. Loops: example 06 (difficulty level: 🌶️🌶️🌶️)

```
/**
 Write a program that draws in the terminal a big X out of the character 'X',
 depending on the variable int size (with size = 3, 4, ..., 20):
 size = 3:     size = 4:      size = 5:      etc.
   X X          X  X           X   X
    X            XX             X X
   X X           XX              X
                X  X            X X
                               X   X      */
#include <iostream>  // to allow use of std::cout and std::endl
int main( ) {



  return 0;
}
```

3.5. Loops: example 07 (difficulty level: 🌶️🌶️🌶️🌶️)

```
/**
 Write a program that draws in the terminal a bigger X out of the character 'X',
 depending on the variable int size (with size = 3, 4, ..., 20):
 size = 3:     size = 4:      size = 5:      etc.
   XX X         XX  X          XX   X
    XX           XXX            XX X
   X XX          XXX             XX
                X  XX           X XX
                               X   XX      */
#include <iostream>  // to allow use of std::cout and std::endl
int main( ) {



  return 0;
}
```