An introduction to C++

day 0

Sandro Andreotti, Chris Bielow

Bioinformatics Solution Center, FU Berlin

slides: Hannes Hauswedell *AG Algorithmische Bioinformatik* et al.

Teachers

Name	E-Mail	Office	Lab
Chris Bielow	chris.bielow@fu-berlin.de	T9/K21	all
Sandro Andreotti	sandro.andreotti@fu-berlin.de	T9/K21	all

In the computer lab, Windows-related questions, you can ask Google/Bing/Copilot/ChatGPT/... (24/7), or Chris (only when awake). Linux questions are easier, and can be answered by everyone (on a good day).

Timeframe

Dates	Content
29.09.2025 - 03.10.2025	Language basics, build process, standard library
06.10.2025 - 10.10.2025	Memory management, OOP, Meta-programming
23.02.2026 - 27.02.2026	Profiling, parallelisation, error handling

Time	Monday – Friday	Rooms
10:00 - 12:00	Lecture	A6 / SR 031
13:00 - 15:00	Computer lab*	T9 / K036 + K038
15:00 - 17:00	Computer lab*	T9 / K036 + K038

*) choose 1; except Fridays (Monday 6.10.2025)

Time	Friday (Monday 6.10.2025)	Room
13:00 - 14:00	Test	T9 / Hörsaal

Day	Content	State
Monday	Variables, Constness, I/O, STL (vector, string, ...)	←
Tuesday	Control flow, functions (overloading), function templates	
Wednesday	Lambdas, Enums, struct/class, class template, compilation	
Thursday	STL: tuples, more data containers, algorithms	
Friday	Holiday	
Monday	Questions, Recap, Debugging, **Test**	

Credits

- **Bioinformatics bachelor** students can get 5 ECTS for this class
- Computer Science or Maths bachelor students *probably* get credits (computer science students did last year)
- need to be signed up to KVV
- need to pass the **test** at the end of every week; the test is T/F and multiple choice
- attendending the lecture and doing the lab work is *highly recommended*

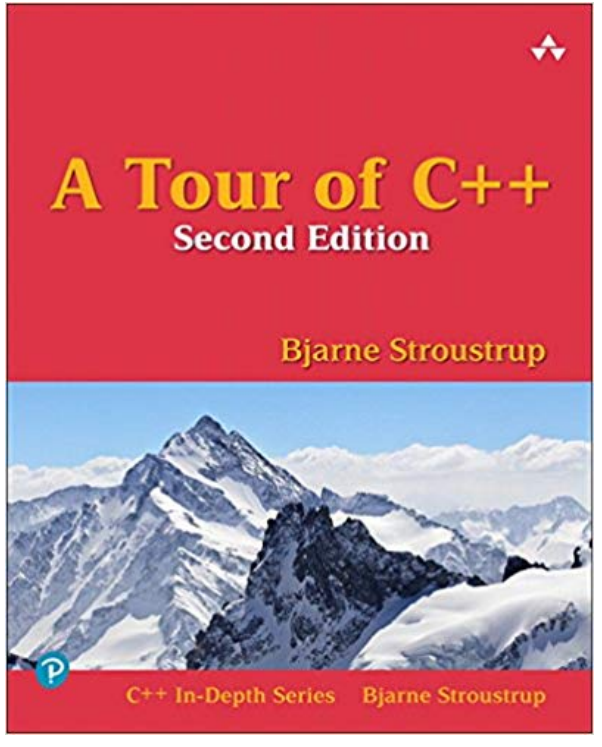
- **Bioinformatics master** students and others may attend, but can't get credits (at least not that we know of)

Recommended reading

- https://isocpp.org/tour
- https://isocpp.org/faq
- https://en.wikipedia.org/wiki/C++

I copy shamelessly from these resources :)

Especially the first part of this class is based strongly on this book by the inventor of C++ →



Scope of this class

"[T]hink of a short sightseeing tour of a city, such as Copenhagen or New York. In just a few hours, you are given a quick peek at the major attractions, told a few background stories, and usually given some suggestions about what to see next. You do not *know the city after such a tour. You do* not *understand all you have seen and heard. To really know a city, you have to live in it, often for years. However, with a bit of luck, you will have gained a bit of an overview, a notion of what might be special about the city, and ideas of what might be of interest to you. After the tour, the real exploration can begin".*

– Bjarne Stroustroup, inventor of C++, https://isocpp.org/tour

About C++

Getting Started

Variables

Arithmetic types

Constants

Simple Input/Output

Arrays, Vectors and Strings

About C++ (from isocpp.org)

C++ is a general-purpose programming language with a bias towards systems programming that

- is a better C
- supports data abstraction (e.g., classes)
- supports object-oriented programming (e.g., inheritance)
- supports generic programming (e.g., reusable generic containers and algorithms)
- supports functional programming (e.g., template metaprogramming, lambda functions, constexpr)

It is defined by an ISO standard, offers stability over decades, and has a large and lively user community.

About C++ (continued)

- a compiled language
  - source code itself is not executable, it has to be translated to machine code
  - the machine code is platform specific (pro: optimised, con: non-portable)
  - certain calculations happen even **before** the program is run
- consists of
  - **core language**: built-in types, loops, control flow…
  - the **standard library**: additional data structures and algorithms (prefixed with `std::`)
- *statically typed*: the type of every entity must be known at compile-time

Evolution of C++

Year	Name	
1985	C++1.0	"C with Classes"
1989	C++2.0	
1998	**C++98**	*first ISO standard*
2003	C++03	*almost no adoption*
2007	C++07/TR1	*almost no adoption*
2011	**C++11/c++0x**	lambdas, `constexpr`, `auto`, range-based-for-loop, `&&`
2014	C++14/c++1y	bugfixes, more `constexpr`, variable templates
2017	C++17/c++1z	many STL additions, `if constexpr`, filesystem
2020	**C++20/c++2a**	Concepts, Ranges, Modules, `<=>`
2023	**C++23/c++2b**	format strings, flat associative containers, import std

Evolution of C++

- C++ was developed as a superset of C. It has diverged greatly, but almost all C code is still valid C++ code.
- C++ itself has changed dramatically in the last 30+ years, but 99% of old C++ code is still valid today.
- This compatibility and stability of the implementation is a major selling point and reason alone C++ will stay an industry standard for the foreseeable future.

ISO Standard

- C++ is not *owned* by a company, it is standardised by the International Organization for Standardization ("ISO") (German chapter is widely known as "DIN")
- members are national body representatives, corporate representatives and independent experts
- there are independent implementations of the standard by the GNU Project, LLVM/Apple, Microsoft, Intel, the Portland Group and others.

About C++

Getting Started

Variables

Arithmetic types

Constants

Simple Input/Output

Arrays, Vectors and Strings

Getting Started

💡 C++ is a compiled language so you need a **compiler**[1]
💡 You need an **editor** to write your code
💡 A **debugger** helps finding bugs[2]

💡 An integrated development environment ("**IDE**") can combine these, but it also obscures core parts of the development process

[1] strictly speaking we also need a linker, but we will get to that later

[2] debuggers will be introduced in the tutorials today

Programming environment

Linux, macOS, unix*

compiler: GNU/g++ or LLVM/Clang
editor: Kate, Atom, VS Code or Sublime
IDE: CLion, QtCreator, Code::Blocks, KDevelop

For this course: any editor and g++>=7

Windows

IDE: Visual Studio

For this course: Visual Studio 2017, Update 8 or later

Hello World!

```
#include <iostream>

int main()                              // I am a code comment!
{
    std::cout << "Hello, World!\n";
}
```

Hello World!

```cpp
#include <iostream>

int main()                              // <-
{
    std::cout << "Hello, World!\n";
}
```

- every C++ program must have a `main()` function; it is the entry-point of the program, all other functions are called from main
- `main()` returns an integer code that signifies success of the program (zero) or failure (non-zero); Operating systems make strong use of this return value; not specifying a return code implies `return 0;` (only for the `main()`-function!)

Hello World!

```cpp
#include <iostream>                 // <-

int main()
{
    std::cout << "Hello, World!\n"; // <-
}
```

- `std::cout` is the standard character output device and its `operator<<` accepts different types, in this case a *string literal*
- `std::cout` is only available, because we included the `iostream` header from the standard library; it is in the standard library's namespace (`std::`)
- a *string literal* is a sequence of characters enclosed in double-quotes; a backslash inside a string literal announces a special character, `\n` is the newline character

About C++

Getting Started

Variables

Arithmetic types

Constants

Simple Input/Output

Arrays, Vectors and Strings

Variables (from "A Tour of C++")

A *declaration* is a statement that introduces a name into the program. It specifies a type for the named entity:

- A *type* defines a set of possible values and a set of operations (for an object).
- An *object* is some memory that holds a value of some type.
- A *value* is a set of bits interpreted according to a type.
- A *variable* is a named object.

Declaration

```
int i;          // i is a variable of integral type, e.g. 1, 77, -3
std::string s;  // s is a variable of string type
```

Variables

Declaration

```
int i;          // i is a variable of integral type, e.g. 1, 77, -3
std::string s;  // s is a variable of string type
```

Initialisation

```
int i;          // value of i is undefined, printing it can kill kittens 🙀
                // C++ Mantra: Only pay for what you use.
int j = 7;      // j is initialised to 7
int k{3};       // k is initialised to 3
int l{};        // l is default-initialised to 0
```

Rule-of-thumb: Always initialise your variables! If unsure, with `{}`.

Variables

Assignment

```
int i{};          // i is initialised to 0
i = 7;            // i is assigned the value 7
```

Beware:

```
int i = 42;       // this is initialisation!
i = 23;           // this is assignment!
```

For `int` there is no difference, but for other types different rules for assignment and initialisation may apply.

Variables

Two kinds of initialisation

```
int j = 7;      // j is initialised to 7
int k{3};       // k is initialised to 3
```

Use "`=`" or "`{}`" to initialise?

- "`=`" is traditional "C-way" of doing it
- "`=`" can be confused with assignment
- not always the same(!), some types can be initialised only via `{}`, e.g. `std::tuple<int, int> p{1, 3};`

→ prefer `{}` unless you have a strong reason to use `()`

Variables

BONUS: parenthesis vs. braces

```
int m(0);   // integer m, initialized to 0
int n = 0;  // integer n, initialized to 0
int o{0};   // integer o, initialized to 0
int p{};    // integer p, initialized to 0

int j(7.9);      // j is initialised to 7
int k{3.2};      // compile error: narrowing

int l{};    // integer l, initialized to 0
int e();    // forward declaration of a function 'e'
            // which takes no arguments and returns an int

vector<int> vec_a{10,20};   // fills the vector with the arguments
vector<int> vec_a = {10,20};// same: fills the vector with the arguments
vector<int> vec_b(10,20);   // uses arguments to parametrize some functionality
```

Variables – Deduced variable types

You can have the compiler *deduce* the type of a variable when *initialising* it:

```
auto i = 7;          // i is of type int
auto d{3.3};         // d is of type double
auto x = foobar();   // x is of whatever type the function foobar() returns
auto& x_ref = x;     // a reference to 'x'
```

- This is still "static typing", the type is fixed at compile-time!
- This is handy when the typename is complex
- 'auto' will never do implicit conversion (avoids accidental conversion!)
- 'auto' will never deduce a '&' (reference) --> must be added explicitly

Further reading/viewing: C++ Weekly - Ep 287 - Understanding `auto`
https://www.youtube.com/watch?v=tn69TCMdYbQ&t=0s

Variables – Scope of variables

```
#include <iostream>

char g = 'G';   // global scope; declare before using it!

void myFunc(char f)
{
  std::cout << f << ' ' << g << '\n';
//    std::cout << b << l << '\n'; // compile-error: 'b' and 'l' not defined
}

int main()
{
    char b = 'B'; // function body scope

    std::cout << g << ' ' << b << '\n';

    { // introduces new local scope
        char l = 'L';
        std::cout << g << ' ' << b << ' ' << l << '\n';
        myFunc(b);
    } // variable l goes "out-of-scope" here

//    std::cout << l << '\n'; // compile-error: 'l' not defined
}
```

About C++

Getting Started

Variables

Arithmetic types

Constants

Simple Input/Output

Arrays, Vectors and Strings

Arithmetic types

type	description	possible values
`bool`	boolean	`true` or `false`
`char`	character	`'a'`, `'\n'`, `32`
`short`, `int`, `long`, `long long`	integral	`1`, `77`, `-3`
`unsigned char` ... `unsigned long long`	unsigned integral	integral >= 0
`float`, `double`	floating point	`2.3`, `1e-10`

- *arithmetic operators*: `+`, `-`, `*`, `/`, `%`
- *arithmetic assignment operators*: `+=`, `-=`, `*=`, `/=`, `%=`
- *increment/decrement*: `++`, `--`
- *comparison operators*: `==`, `!=`, `<`, `<=`, `>`, `>=`

What is '%'? Does it work on floats? Does '++' work on bool?

Arithmetic types

Some examples:

```
int i = 3;
int j{};
j = i + 3;          // j == 6
j += 2;             // j == 8

bool b = (i < j);   // == true
bool c = (i == j);  // == false

double d = 2.3;
bool e = (d < i);   // == true (arith. types are convertible to each other!)

bool f = (i = j);   // ?
```

Arithmetic types

Some examples:

```
int i = 3;
int j{};
j = i + 3;          // j == 6
j += 2;             // j == 8

bool b = (i < j);   // == true
bool c = (i == j);  // == false

double d = 2.3;
bool e = (d < i);   // == true (arith. types are convertible to each other!)

bool f = (i = j);   // BEWARE: you have assigned 8 to i; and 'f' is now true!
```

Arithmetic types

Size of arithmetic types

- All arithmetic types have a fixed precision and a fixed size in memory!
- The size determines the range of values a type can represent.
- Assigning a value that is too large/small for a type causes overflow/underflow (unsigned types), or undefined behavior (signed types) and does not produce a diagnostic!

What are the sizes of the arithmetic types?

Arithmetic types

Size of arithmetic types

- All arithmetic types have a fixed precision and a fixed size in memory!
- The size determines the range of values a type can represent.
- Assigning a value that is too large/small for a type causes overflow/underflow (unsigned types), or undefined behavior (signed types) and does not produce a diagnostic!

What are the sizes of the arithmetic types?

It depends on the platform, i.e. both the CPU architecture and the operating system!

Arithmetic types

type	at least	typically
`bool`	8bit	8bit
`char`	8bit	8bit
`short`	16bit	16bit
`int`	16bit	16bit or 32bit
`long`	32bit	32bit or 64bit
`long long`	64bit	64bit
`float`	32bit	32bit
`double`	64bit	64bit

- `unsigned` types have the same resp. sizes
- you can use `sizeof(type)` to get the size of a type **in bytes**
- it is problematic to use these types in cross-platform code, especially `int` and `long`

Arithmetic types

OS and machine independent fixed-width integers from `<cstdint>`:

- `int8_t`, `int16_t`, `int32_t`, `int64_t`
- `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`

OS independent[1], but machine dependent "large types" from `<cstddef>`:

- `ptrdiff_t` equals `int32_t` or `int64_t`
- `size_t` equals `uint32_t` or `uint64_t`

[1] Unless the OS doesn't fit the machine 😉

Arithmetic types

Summary

- Think about what kind of number range you wish to represent.
- First question: boolean, signed integral, unsigned integral or floating point?
- make integrals unsigned if possible!
- **unsigned integral** and no other information? → `size_t`
- **floating point** and no other information? → `double`
- integral and **known range**? → fixed-width `int*_t` / `uint*_t`
- avoid `int`, `long`, `unsigned int`, `unsigned long`

About C++

Getting Started

Variables

Arithmetic types

Constants

Simple Input/Output

Arrays, Vectors and Strings

Constants (from "A Tour of C++")

C++ supports two notions of immutability:

- `const`: meaning roughly "I promise not to change this value" . This is used primarily to specify interfaces, so that data can be passed to functions without fear of it being modified. The compiler enforces the promise made by `const`.
- `constexpr`: meaning roughly "to be evaluated at compile time". This is used primarily to specify constants, to allow placement of data in memory where it is unlikely to be corrupted, and for performance.

Constants – Examples

```
size_t           i = 5;
size_t const     j = 3;
size_t constexpr k = 4;
i = 6666;                       // ok, since i is not const
// j = 7;                       // no can do, because j is a constant
// k = 7;                       // no can do, because k is a constant

size_t const l = j + 4;         // may be computed at run-time
size_t const m = i + 4;         // computed at run-time

size_t constexpr n = k + 4;     // computed at compile time
// size_t constexpr o = i + 4;  // not possible, because i is not constexpr

size_t const     p = foobar();  // works if foobar() returns size_t
size_t constexpr q = foobar();  // only works if foobar() is constexpr
```

Constants – Notation

"east-const" VS "west-const"

There are two conventions for `const`-notation:

- "east-const": `double const d = 3.3;`
- "west-const": `const double d = 3.3;`

Equivalent in the above example, but for more complex types the rule for "east-const" is easier: `const` applies to what is on its left.

Constants

Summary

- always use constants instead of variables (if possible)
- use `constexpr` instead of `const` (if possible)
- [not all types can be `constexpr`!]
- write `const` / `constexpr` on the right of the type you wish to mark

About C++

Getting Started

Variables

Arithmetic types

Constants

Simple Input/Output

Arrays, Vectors and Strings

Simple Input/Output

```
#include <iostream>
#include <string>

int main()
{
    std::string s{"foo"};
    std::cout << "Welcome to the " << s << " program!\n";
    std::cout << "Enter two floating point numbers followed by [RETURN]\n";

    double d1{};
    double d2{};
    std::cin >> d1 >> d2;

    std::cout << "The sum is: " << d1 + d2 << '\n';
}
```

Simple Input/Output

- Use `std::cout` for normal interaction with the user.
- Use `std::cerr` to print error messages.
- Use `std::cin` to read user input.
- `std::cout` and `std::cerr` are used with output stream operator `<<`.
- `std::cin` is used with the input stream operator `>>`.
- You can "chain" multiple input/output operations by repeatedly invoking the respective stream operator.
- The input stream implictly "splits input" at whitespaces.
- You don't need to understand how the stream operators work at this point.

About C++

Getting Started

Variables

Arithmetic types

Constants

Simple Input/Output

Arrays, Vectors and Strings

Arrays (built-in)

```
// declare built-in array of size 3:
double dd[3];

// can be {} initialised:
double df[3]{3.1, 2.3, 1.1};

// access elements also via []
std::cout << df[0];     // prints 3.1
df[1] = 32.0;           // you can also "assign through" []
df[77] = 32.0;          // memory access violation, no error, kills kittens

size_t constexpr s = 100;
double dg[s];           // only works because s is constexpr
```

- arrays are 0-indexed, size is fixed at compile-time
- accessing elements behind the end is a dangerous problem

Arrays (standard library)

```cpp
#include <array>

// declare standard array of size 3:
std::array<double, 3> dd;

// can be {} initialised:
std::array<double, 3> df{3.1, 2.3, 1.1};

// access elements also via []
std::cout << df[0];     // prints 3.1
df[1] = 32.0;           // you can also "assign through" []
df[77] = 32.0;          // memory access violation, no error, kills kittens
```

- has convenience member functions: `.size()`, `.front()`, `.begin()` ...
- has "safe" random access function: `.at(77)` instead of `[77]`
- preferable to "C-array" in any kind of serious code-project

Vectors

```cpp
#include <vector>

// declare a vector of doubles:
std::vector<double> dd;

// can be {} initialised:
std::vector<double> df{3.1, 2.3, 1.1};

// access elements also via []
std::cout << df[0];     // prints 3.1
std::cout << df.size(); // prints 3

df.push_back(0.5);
std::cout << df.size(); // prints 4
std::cout << df.back(); // prints 0.5
```

- Vectors are "resizable arrays".
- provide many more member functions

String

```cpp
#include <string>

// declare an empty string:
std::string dd;

// can be {} initialised:
std::string df{"FOOBAR"};

// access elements also via []
std::cout << df[0];     // prints 'F'
std::cout << df.size(); // prints 6

df.push_back('a');
std::cout << df.size(); // prints 7
std::cout << df;        // prints "FOOBARa"
```

- `std::string` is similar to `std::vector<char>`
- but has some convenience functions and optimisations.
- It can be printed as a whole!

- `"FOOBAR"` is a string literal (the type is `const char[7]`)
- `'a'` is a character literal (the type is `char`)

Tasks for the computer lab

Setting up the build environment (Linux)

1. Open your editor and create YOUR_FILE.cpp, copy the hello world program into it.
2. Open a terminal and go into into the directory of the file.
3. Run the following to compile the source code:

```
$ g++ -std=c++17 -Wall -Wextra -Werror -pedantic YOUR_FILE.cpp -o hello
```

4. (there should be no errors!)
5. What do the above flags mean? Find out!
6. Then run the program with

```
$ ./hello
```

Setting up the build environment (macOS)

We don't officially support mac in this class.

But the following *should* work:

1. Install homebrew from https://brew.sh
2. Run `brew install gcc`
3. follow the instructions for Linux

or try Visual Studio Community for MacOS or try XCode for MacOS (we cannot help you setting that up)

Setting up the build environment (Windows)

If you are using the Windows desktop computers in T9 basement, skip the first step:

1. Install Visual Studio 2017 or above (make sure to install the "Desktop Development with C++" Workload)
2. Open Visual Studio (VS) and create a new project: File->New->Project->C++ Windows Console App
3. Open Project -> ... Properties and adjust the following values (defaults vary with VS version...)
    - C/C++ -> General
        - Warning Level: `/W3`
        - Treat Warnings As Errors: `Yes`
    - C/C++ -> Language
        - Conformance Mode: `Yes`
        - C++ Standard language: `Latest Draft Standard`
    - C/C++ -> Precompiled Headers
        - Precompiled Headers: `Not using precompiled headers`
4. Change the configuration from `x86` to `x64` (drop-down in main toolbar)
5. Copy the hello world program into your consoleApplication.cpp which VS created for you already
6. Press the green triangle button to build and run the program

P.S.: Windows has a Terminal, too! Give it a try!

Setting up the build environment (quick and dirty)

As a last resort, use an online compiler such as https://www.onlinegdb.com/

--> set to C++ (17)

--> set compile flags to '-std=c++17 -Wall -Wextra -Werror -pedantic'

Switch to a proper IDE, once you've installed one.

Tasks for the computer lab I

Integral types:

1. Write a program that prints for the types `char`, `short`, `int`, `long`, `long long` and the respective `unsigned` versions (e.g. `unsigned int`):
   1. the size in bits
   2. the largest possible value (find the C++ way, not the C way)
   3. the smallest possible value (find the C++ way, not the C way)
2. What happens when you assign a value that is too large/small? Is this *defined* behaviour (according to the C++ standard)?

Tasks for the computer lab II

Floating point types:

1. Adapt the code from the "Simple I/O" page
2. For a single user-given `double`, have your program print the number rounded down, rounded up and it's square root; what is the behaviour for negative numbers?
3. Find out how to change the precision of the output stream so more/less digits are printed.
4. Find out how to change the output format to always show `1e-01` instead of `0.1`.

Tasks for the computer lab III

Debugging:

1. Find an online tutorial on Debuggers for the IDE you are currently using, e.g. https://learn.microsoft.com/de-de/visualstudio/debugger/getting-started-with-the-debugger-cpp?view=vs-2022 for VS2022 or https://doc.qt.io/qtcreator/creator-debugging.html for QtCreator. It should cover the topics of 'Breakpoints', 'Stepping through code' and 'Inspection of variables and their values'
2. Work through that tutorial.

Tasks for the computer lab IV

Out-of-bounds:

1. Write a program that reads a string and then a number from the user keyboard
2. print the i-th character of the string to the user (where i is the number supplied by the user)
3. What happens when the number is larger than the string is big?
4. Did you use `operator[]` or `.at()` to access the string? Try both and compare the behaviour (for the previous subtask)!
5. Use the debugger of your IDE and: 1) set a breakpoint right after reading the user input, 2) run the programm and wait for it to pause at the breakpoint. 3) step through each of the following lines and inspect local variables 4) What does the debugger do when you use an element index which is too big?