C++ Programming

Lecture 0

Secure Software Engineering Group

Philipp Dominik Schubert
C++ is easy.
It’s like riding a bike.
Except the bike is on fire,
and you’re on fire
and everything is on fire
because you’re in hell.
The C++ Programming Language

junior C++ developer

senior C++ developer
Contents

1. Organizational matter
2. Course outline
3. History of C++
4. C++ compilers
5. A “Hello, World!” program
6. Setting up a development environment
7. Basic terms & concepts
Organization

- “Rooms”
  - Lecture: recorded (Panda/YouTube), available on Fridays ~14:00
  - Exercises: livestream (Twitch), Fridays 16:00-18:00

- Instructor
  - Philipp Schubert @home in BI
  - E-Mail philipp.schubert@upb.de
  - Web https://www.hni.uni-paderborn.de/sse/lehre/cppp/

- Prerequisites
  - No programming experience
  - Knowledge on how to use a computer
    - Text editor
    - Operating system (Linux/Windows/Mac)
Organization

- **Benefits**
  - Be confident to take advanced courses that require C++
  - Realize programming projects
  - Will be useful for computational thinking
  - Better understanding on how a computer works
  - Well-paid jobs

- **Studium Generale (SG) EIM-I**
  - Computer science students will not receive credit points
  - Electrical engineering students will not receive credit points
  - When in doubt ask your examination office
  - All (most) other students will receive 4 credit points
  - Everyone obtains a nice certificate for their CV
Organization

- Some of you have not yet registered?
  - Register to this course in Panda
    - [https://panda.uni-paderborn.de/course/view.php?id=22691](https://panda.uni-paderborn.de/course/view.php?id=22691)
    - I will send emails with additional materials
- External students
  - [https://www.hni.uni-paderborn.de/sse/lehre/cppp](https://www.hni.uni-paderborn.de/sse/lehre/cppp)
Contents

1. Organizational matter
2. Course outline: now with even more C++
3. History of the C++ language
4. A “Hello, World!” program
5. C++ compilers
6. Setting up a development environment
7. Basic terms & concepts
Course outline

- Basic introduction
  - History of C & C++
  - Compilers
  - Development environments
  - Basic terms and concepts

- Basic C++ programming
  - Primitive data types, strings, vectors, arrays, pointers
  - Expressions, statements
  - Structures, unions, enumerations
  - Functions, classes
Course outline

- How to organize a project
  - Tooling
  - Namespaces
  - Forward declarations

- C++’ Standard Template Library (STL)
  - IO, containers, generic algorithms
  - Static / dynamic memory
  - Smart pointers

- Advanced techniques
  - Copy control, standard class members
  - Operator overloading
  - Object-oriented programming
  - Templates and generic programming
Course outline

- Useful libraries
  - OpenMP, OpenCV, OpenCL, OpenGL/Vulkan, …
  - Qt
  - Google test
  - Google protobuf
  - Abseil
  - Boost
  - And other useful libraries
- Where to find the desired information you need
- Don’t reinvent the wheel, use libraries
**Literature**

- [5] CppCon, [https://www.youtube.com/user/cppcon/](https://www.youtube.com/user/cppcon/)

Various different input channels are important:

- Lecture
- Exercises
- I’ll try to make links to books and YouTube videos
- Talk to each other and look things up
Exercises

- Weekly exercises
  - Theoretical and practical exercises
- Submissions are graded
  - You need to achieve 50% during semester
- Final project
  - Solve a programming task
- Certificate (+ credit points)
  - Pass exercises + project solved
  - No final exams
- Plagiarism is prohibited (Plage Source Code Copying Detector [https://sourceforge.net/projects/plage/])
- Adhere to the notes on the exercise sheets
- Questions so far?
1. Organizational matter
2. Course outline
3. History of C++
4. C++ compilers
5. A “Hello World” program
6. Setting up a development environment
7. Basic terms & concepts
What is C++?
What is C++?

- An object-oriented programming language
- Generic Programming
- Template meta-programming
- Buffer overflows
- Classes
- Too big
- Host for DSLs
- A hybrid language

- Embedded systems
- Low level
- A random collection of features
- Class hierarchies
- Multi-paradigms
- A failed attempt to build Java
- It's C
- Too complicated

Stroustrup: The Essence of C++
What is C++?

**THE VIRGIN RUST**
- need to use unsafe blocks to make programming exciting
- cannot write a compile-time raytracer yet
- no feeling of reward when dealing with complicated library dependency graphs
- doesn’t let you swap array elements
- you don’t have to creatively implement pattern matching yourself by abusing lambdas

**THE CHAD++**
- multiple compilers, each one with different bugs
- provides several ways of initializing variables to promote diversity
- sometimes segfaults to keep you on your toes
- literally has 3 or 4 accidentally turing-complete languages embedded into it
- allows you to create compile-time tetris
- never removes features to preserve backward compatibility
- provides job security
- fast like sonic
- it’s probably UB
Advice

- Don’t be afraid
- Learning a new language takes time
- Practice, practice, practice
- Read a lot about it (books and C++ forums / as well as code)
- Do the exercises
- Always ask yourself: why does this work?
  - If you are curious about something → use google
    - … and share your knowledge and discuss with friends
- Programming will be fun when understood
History of C++

- All started with BCPL
  - Basic Combined Programming Language
  - Has no data types
- B – a language to implement operating systems
- C – better than B
  - Brian Wilson Kernighan
  - Dennis MacAlister Ritchie
- C with Classes
  - Bjarne Stroustrup
- C++
  - Dynamically evolving
  - C++14/C++17/C++20

[Figure and images taken from images.google.de/ and A Tour of C++, Stroustrup 2013]
History of C++

- But why are we not learning C++20?
- I cannot teach five courses in one
- Adaption needs time
  - Concepts and ideas first
  - Compiler implementations follow
  - // void …
- Industry usually adapts ~ 5-10 years later
  - There are reasons for that
    - Concepts have to be proven as useful
    - Compilers have to mature over time

[Figure from A Tour of C++, Stroustrup 2013]
History of C++

- BCPL, B, C,
  - Why not D after C?
  - C was and is still tremendously successful
  - Lots of existing code was and is still written in C
  - Don’t break compatibility!
  - Be an increment rather than a new language
  - A language called D exists
    - D is no longer compatible with C
  - Be aware: Modern C++ is not C
Contents

1. Organizational matter
2. Course outline
3. History of C++
4. C++ compilers
5. A “Hello, World!” program
6. Setting up a development environment
7. Basic terms & concepts
What is a compiler?

Figure 1.1: A compiler

Figure 1.2: Running the target program
Are there other forms? Interpreter

Figure 1.3: An interpreter
Even more: hybrid compilers

Figure 1.4: A hybrid compiler
C++ compilers

- **Gnu Compiler Collection GCC**
  - Includes C and C++ front-ends
  - Standard on most Linux dists.
  - “Most used C/C++ compiler in the world”
  - Fist stable release was v1.17 (1988)
  - Monolithic design
  - Written by bootstrapping
    - Written by *something else* until its powerful enough to compile itself

- **Clang**
  - Compiler front-end for C-like languages (including C and C++)
  - Used by Google, Apple, Oracle ...
  - Started as a Ph.D. thesis by Chris Lattner
  - Stable version in 2009
  - Part of a reusable compiler infrastructure (LLVM project)
  - Written in C++

There are a lot more: Intel icc, IBM C++, MSVS C++, Oracle +++, Apple C++, Bloodshed Dev-C++, EDG C++

[Figures from gcc.gnu.org and clang.llvm.org]
GCC and Clang are language processing systems

- C++ is (usually) a compiled language
- C++ compilers are language processing systems / compiler tool chains

![Diagram of a language-processing system](image)

Figure 1.5: A language-processing system

Remark on what follows

- “Keep simple things simple, as simple as possible, but not simpler!” (Einstein)
- Problem: where to start when learning a programming language?
  - It all seems like magic
  - In order to be able to start at all we have to …
    1. take certain things for granted
    2. learn the WHY over time
Contents

1. Organizational matter
2. Course outline
3. History of C++
4. C++ compilers
5. A “Hello, World!” program
6. Setting up a development environment
7. Basic terms & concepts
A “Hello, World!” program

- Shortest valid C++ program

#include <iostream>

// This function prints Hello, World!
int main(int argc, char **argv) {
    std::cout << "Hello, World!\n";
    return 0;
}
A “Hello, World!” program

- Tell the compiler to translate ‘hello.cpp’ into executable machine code

  Command:
  - `cc hello.cpp -o hello`
  - You can execute the program ‘hello’ with `../hello`

- Replace `cc` with `g++` or `clang++`

Edit a text file, e.g. ‘hello.cpp’, with the following contents:

```cpp
#include <iostream>

int main(int argc, char **argv){
    std::cout << "Hello, World!\n";
    return 0;
}
```
A “Hello, World!” program

- Some useful compiler flags
  - `-Wall` turns on compiler warning
  - `-Wextra` turns on even more warnings
  - `-g` insert debugging symbols
  - `-Ox` turn on compiler optimization (x is a number: 0,1,2,3)
  - `-o` specify the output file
  - `-std=X` specify the C++ standard
    e.g. `-std=c++17` or `-std=c++20`

- E.g.
  
g++ -Wall -Wextra -std=c++17 hello.cpp -o hello

Edit a text file, e.g. ‘hello.cpp’, with the following contents:

```cpp
#include <iostream>

int main(int argc, char **argv) {
    std::cout << "Hello, World!\n";
    return 0;
}
```
A “Hello, World!” program

- `#`-directives are instructions for the preprocessor
  - Preprocessor runs over the program first
  - Then compiler starts its job
- `#include` directives just perform textual insertion
  - `std::` is a namespace
    - Namespaces hold code
    - Helps to avoid collisions (e.g. variable names, function names, …)

```cpp
#include <iostream>

int main(int argc, char **argv) {
    std::cout << "Hello, World!\n";
    return 0;
}
```

Figure 1.5: A language-processing system

[Figure from Compilers: Principles, Techniques, & Tools, 2007]
A “Hello, World!” program

- Compiler option `-S` shows the assembly code
- `cc hello.cpp -S -o hello.as`

```cpp
#include <iostream>

int main(int argc, char **argv) {
    std::cout << "Hello, World!\n";
    return 0;
}
```

![Figure from Compilers: Principles, Techniques, & Tools, 2007]

```assembly
.LFB971:
... // code still continues
```

Figure 1.5: A language-processing system
A “Hello, World!” program

- Compile to binary directly
- `cc hello.cpp -o hello`
- Content of hello looks like that

```cpp
#include <iostream>

int main(int argc, char **argv) {
    std::cout << "Hello, World!\n";
    return 0;
}
```

Figure 1.5: A language-processing system
Contents

1. Organizational matter
2. Course outline
3. History of C++
4. C++ compilers
5. A “Hello, World!” program
6. Setting up a development environment
7. Basic terms & concepts
Calling the compiler by hand is *wasteful*

- **Makefile, CMake, and friends**
  - Help to organize a software’s source code
  - Text files containing rules that describe how to invoke the compiler
  - Rules are read, identified, and executed on-demand
  - Flexible and powerful
  - Hard to write for complex tasks
    - Start with a template
  - You see what’s going on
    - Nothing is hidden under the carpet

- **Integrated Development Environment (IDE)**
  - Handles the project and corresponding source files for you
  - Handles compiler invocations
  - Easier to use than Makefile, CMake, etc.
  - Will find syntax errors on-the-fly
  - More complex tasks are painful
    - Lack of control
    - Hides complexity

- **I’m using a combination of both!**
Makefile, an example

- Using the compiler ‘by hand’ is fiddly
- Use files describing the compiler commands
  - Makefile
    - Contains executable “targets”
    - Consist of a bunch of declarative rules
    - Processed by `make`
  - Flexible
  - Easy to use
  - Hard to write
    - There are books on `make`

- Project directory: `MyProject/`
  - Makefile
    ```
    PROGNAME := hello
    CC := g+
    FLAGS := -std=c++17
    FLAGS += -Wall
    all: main.cpp
        $(CC) $(FLAGS) *.cpp -o $(PROGNAME)
    clean:
        rm -f $(PROGNAME)
    ```
  - `hello.cpp`:
    ```
    #include <iostream>
    int main() {
        std::cout << "Hello, World!\n";
        return 0;
    }
    ```
Integrated Development Environment (IDE) and other editors

- Visual Studio Code
  - Compact editor
  - Windows / Linux / Mac
- Or use vim, emacs, etc. (hardcore ;-) 
- Use whatever feels best to you
  - Depending on your programming level and experience

[Figures taken from eclipse.org and code.visualstudio.com]
Set up a development environment

- Set up a development environment?
  - I will provide a virtual machine
  - Password: cppp
  - Ubuntu 20.04, ~20 GB (sorry)
  - Ships with everything that is needed

- Remark on compiler errors
  - Errors are the default case
  - Don’t panic and read them
  - Read them carefully
  - Google will help
  - So does stack overflow (a programming forum)

```cpp
#include <iostream>
int main() {
    cout << "Hello, World!\n";
    return 0;
}
```

```
# include <iostream>
int main() {
    cout << "Hello, World!\n";
    return 0;
}
```

```
# include <iostream>
int main() {
    cout << "Hello, World!\n";
    return 0;
}
```

```
# include <iostream>
int main() {
    cout << "Hello, World!\n";
    return 0;
}
```

```
# include <iostream>
int main() {
    cout << "Hello, World!\n";
    return 0;
}
```

```
# include <iostream>
int main() {
    cout << "Hello, World!\n";
    return 0;
}
```

```
# include <iostream>
int main() {
    cout << "Hello, World!\n";
    return 0;
}
```

```
# include <iostream>
int main() {
    cout << "Hello, World!\n";
    return 0;
}
```

```
# include <iostream>
int main() {
    cout << "Hello, World!\n";
    return 0;
}
```

```
# include <iostream>
int main() {
    cout << "Hello, World!\n";
    return 0;
}
```

```
# include <iostream>
int main() {
    cout << "Hello, World!\n";
    return 0;
}
```

```
# include <iostream>
int main() {
    cout << "Hello, World!\n";
    return 0;
}
```
Contents

1. Organizational matter
2. Course outline
3. History of the C++ language
4. C++ compilers
5. A “Hello World” program
6. Setting up a development environment
7. Basic terms & concepts
## Primitive / built-in data types

- **Boolean types**
  - `bool`
  - Can hold `true` or `false`

- **Character types**
  - `char`

- **Integer types**
  - `int`
  - Modifiers and sizes (integer types only)
    - `signed` and `unsigned`
    - `short`, `long`, `long long`

- **Floating point types**
  - `float`
  - `double`
  - `long double`
Integer encoding

- **unsigned char**
  - 1 byte = 8 bit
- Dual number encoding with **unsigned**

```
1 0 1 1 1 0 0 1 1
```

Decimal value: \[1 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0\]

\[= 128 + 32 + 16 + 2 + 1 = 179\]
Integer encoding

- **signed char** or **char**
  - 1 byte = 8 bit
- Two’s complement encoding with **signed** or as default

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
</table>

- Highest bit encodes sign
- Other bits encode value
- Here: sign bit 1, number is negative: take two’s complement (negate and add 1)

<table>
<thead>
<tr>
<th>-</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Take complement

Add one

Decimal value: $1 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 64 + 0 + 0 + 8 + 4 + 0 + 1 = 77 \rightarrow -77$
Floating point number encoding

- IEEE-754 single-precision binary floating-point format

\[
\text{value} = (-1)^{\text{sign}} \times \left( 1 + \sum_{i=1}^{23} b_{23-i} 2^{-i} \right) \times 2^{e-127}
\]

- IEEE-754 double-precision binary floating-point format

\[
(-1)^{\text{sign}} \left( 1 + \sum_{i=1}^{52} b_{52-i} 2^{-i} \right) \times 2^{e-1023}
\]

- Remark
  - Use `double` as default, `float` usually far too imprecise
  - Floating point numbers are not distributed equidistant

[Figures from Wikipedia]
Comments in C++

- Comments tell **other people** what your code does
- Comments tell **yourself** what your code does
  - Or at least what it is supposed to do
- Code can be hard to understand

Examples

- // a single-line comment
- /*
   A multi-line comment
   */
- /* ... */
  ... *
  ... /* this is wrong

Me: Writes some code

Gcc:
Integer literals in C++

- 100 // int decimal
- 123456 // int decimal
- 5L // long, decimal
- 123u // unsigned int, decimal
- 777uL // unsigned long, decimal
- -02O // int, octal
- 0x1fff // int, hexadecimal
- 0x1ffful // unsigned long, hexadecimal
Character literals in C++

- 'A'  // character A
- '*'  // symbol *
- '\0' // end of a string
- '\n' // new line
- '\t' // tabulator
- '\' // apostrophe
- '\\' // backslash

String literals in C++

- "This is a string literal!"  // a string literal
  - More on strings later
Floating-point literals in C++

-9.876 // double
123.456E-7 // double
1e12 // double
.001 // double
1.23f // float
1.23L // long double
Defining variables in C++

- Variables have a
  - Type
  - Name
  - Optional: an initial value

```cpp
int main() {
    // see left side
    return 0;
}
```

- Initialize your variables, unless you know what you are doing!

```cpp
int i = 42;
int j;
int k = 10, l = 42, m;
double d = 1;
double e;
double f = 1.23456;
float g = 12.5f;
float h = 42.13;
char c = 'A';
char c[] = "A string"; // later on
char *c = "Another string"; // later on
char x = -10;
unsigned int ui = 123;
unsigned int huge = -13; // DON'T!!
```
Variables in C++

- `unsigned int huge = -13; // DON'T!!!`
  - Dangerous
  - Integer overflow

- C++ is famous for its undefined behavior
  - C++ standard allows undefined behavior in some situations
    ```cpp
    int i;
    int j = i + 42;
    ```
  - Anything can happen
  - Depends on the compiler's implementation
  - Why?
    - Compilers can produce faster machine code when assuming that certain things cannot happen

[Figure taken from https://www.reddit.com/r/ProgrammerHumor/comments/8p54sk/reporting_errors/]
Variables in C++

- **auto** keyword
  - Automatic type deduction
  - Compiler finds the correct type
  - Always be verbose
    - If type name gets ‘too long’ or type is obvious use `auto`
- What type is x?
  - `auto x = 13L; // long`
  - `auto x = 1.2345; // double`

```cpp
#include <vector>
// C++98 style 😊
std::vector<int> v;
v.push_back(1);
v.push_back(2);
v.push_back(3);
for (std::vector<int>::iterator it = v.begin(); it != v.end(); ++it) {
    std::cout << *it << '
';
}

// using modern C++
std::vector<int> w = {1, 2, 3};
for (auto i : w) {
    std::cout << i << '
';
}
```
Making a point: there are ~50 ways to initialize a simple integer

- `int a = 1;`
- `int b(2);`
- `int c{3};`
- `int d = {4};`
- `auto i = 5;`
- `auto j(6);`
- `auto k{7};`
- `auto l = {8};`
IO streams

- #include <iostream>
- Part of the STL
- Content lives in namespace std
- Use std::
- Important variables
  - cin standard input stream
  - cout standard output stream
  - cerr standard error stream
  - clog general information
  - << and >> are shift operators defined (i.e., overloaded) on the stream variables

Example

```cpp
#include <iostream>

int main() {
    int i = 0;
    std::cout << "Enter an integer: ";
    std::cin >> i;
    std::cout << "The value of i is: " << i << '\n';
    return 0;
}
```
Recap

- Course outline
- What is C++?
- History of C++
- Compilers
- "Hello, World!"
- Built-in types
- Information encoding
- Variables
- IO streams

- Any questions?
And now?

- Quick demo: the development environment and how to write a “Hello, World!” program
  1. Visual Studio Code
  2. How to get a C++ job?
Thank you for your attention

Questions?