ETH zürich

# Programming Techniques for Scientific Simulations I

Michal Sudwoj

# A first C++ program

```cpp
// main.cpp
#include <iostream>
int main() {
    std::cout << "Hello, World!" << std::endl;
}
```

- comment: `// ...` (line), `/* ... */` (block)
- include system header: `#include <...>`
- entrypoint: `int main()`
- block: `{ ... }`
- standard output: `std::cout`
- stream operator: `<<`
- string: `"..."`
- newline and flush: `std::endl`
- in `main()`, `return 0;` is implicit

# A first C++ program

## Compile using:

```
> g++ main.cpp -o main
```

- more on compilation in week 2
- Tip: always use `-std=c++17 -Wall -Wextra -Wpendatic`:

  ```
  > g++ -std=c++17 -Wall -Wextra -Wpedantic main.cpp -o main
  ```

  - `-std=c++17` selects the C++17 standard (without GNU extensions)
  - `-Wall` activates “all” warnings
  - `-Wextra` activates “extra” warnings
  - `-Wpedantic` activates standard conformance

# Contents

# Getting help

- `https://www.cppreference.com`
- available during the exam (albeit without search)

# C++ history

- 1979: developped by Bjarne Stroustrup at AT&T Bell Labs
- 1985: *The C++ Programming Language*
- 1998: First ISO standard
- 2003, 2011, 2014, 2017, 2020: newer standards
- “C with classes”
- legacy $\implies$ things might be weird or complicated, because historic reasons

# Variables & Types

Variables have to be **declared** before they are used:

```
unsigned int N;
bool         is_positive;
float        temperature;
```

C++ is statically typed (type of variables does not change)

```
int x = 1;
x = 2.5; // double cast to int -> x == 2;
x = "three"; // compile-time error
```

# Variables & Types

Fundamental types:

- boolean: `bool`
- integral types: `int`, `signed char`, `short`, `long`, `long long`
- also `unsigned`: `unsigned int`, `unsigned char`, `unsigned short`, ...
- floating-point types: `float`, `double`, `long double`
- character type: `char`
- `void`

# Variables & Types

Cave:

- 1 byte $\leq$ `sizeof(char)`
- 2 bytes $\leq$ `sizeof(short)` $\leq$ `sizeof(int)`
- 4 bytes $\leq$ `sizeof(long)`
- 8 bytes $\leq$ `sizeof(long long)`
- `sizeof(float)` $\leq$ `sizeof(double)` $\leq$ `sizeof(long double)`
- `signed char` $\neq$ `char` $\neq$ `unsigned char`

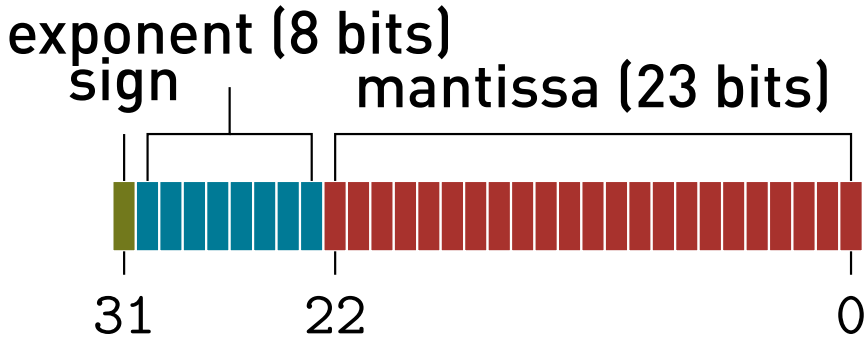
# Variables & Types

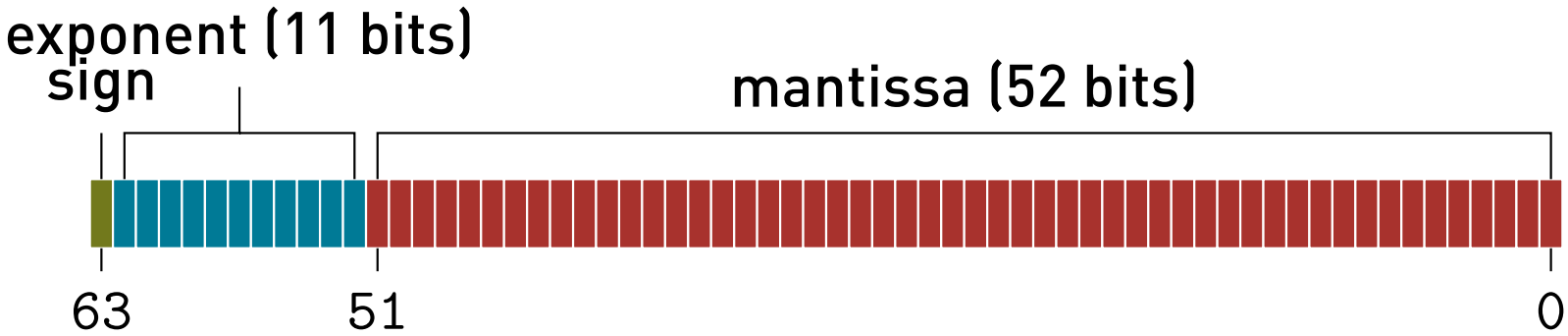
## In practice:

```
sizeof(char)        ==  1;
sizeof(short)       ==  2;
sizeof(int)         ==  4;
sizeof(long)        ==  4; // on Windows
sizeof(long)        ==  8; // on Mac, Linux
sizeof(long long)   ==  8;
sizeof(float)       ==  4;
sizeof(double)      ==  8;
sizeof(long double) ==  8; // on Windows
sizeof(long double) == 10; // on Mac, Linux
```

- if you need a certain length, use fixed-width types from `<cstdint>`: `std::int8_t`, `std::uint64_t`, ...
- especially, never use `long`, `long long` and `long double`

# IEEE-754 floating-point numbers



# IEEE-754 floating-point numbers

| Type | Exponent | Mantissa | Smallest | Largest | Decimal accuracy |
|---|---|---|---|---|---|
| `float` | 8 | 23 | 1.2e-38 | 3.4e38 | 6–9 digits |
| `double` | 11 | 52 | 2.2e-308 | 1.8e308 | 15–17 digits |

- dynamic range, not uniformly distributed
- close together near 0, far-spaced at large magnitudes
  $\Longrightarrow$ truncation and roundoff
- Tip: use `double` by default

# `char` and `void`

- `char`: a single (ASCII) character: `'A'`, `'B'`, `'C'`, ...
- `void`: nothing!

```
void print_int(int x) {
    std::cout << x;
}
```

# Arrays

```
int primes[5] = {2, 3, 5, 7, 11};
x[1] == 3; // true
x[0] = 1; // primes == {1, 3, 5, 7, 11};
```

- contiguous in memory
- **compile-time** sized, not resizeable
- **zero-indexed**
- indexing **does not perform bounds checking**
  $\Longrightarrow$ bugs, `SEGFAULT`s, ...
- $\Longrightarrow$ use `std::vector` from `<vector>`

# Strings

- C-strings: NUL-terminated array of characters:

```
char hello[] = "Hello";
char bye[] = { 'B', 'y', 'e', '\0' };
sizeof(hello) == 6;
```

- use `std::string` from `<string>`

# Enumerators

```
enum class Direction {
    North,
    South,
    East,
    West
};

// usage
Direction dir = Direction::South;
```

- a exhaustive set of options
- mapped to some integer value behind the scenes

# Type aliases

```
using mass_t = double;
typedef int charge_t;

mass_t   m_e = 5.485e-4;
charge_t q_e = -1;
```

- `using` and `typedef` are equivalent
- `using`: modern, clearer syntax
- `typedef`: older, compatible with C
- **neither introduces a new type, only an alias!**

# auto

```
auto my_int           = 1;
auto my_unsigned_long = 1ul;
auto my_float         = 1.0f;
auto my_double        = 1.0;
```

Cool right? Well...

- sometimes we want to use a different type

  ```
  std::string name = "Michal Sudwoj";
  ```

- literal suffixes easy to miss/forget

  ```
  auto my_unsigned_int = 1234; // missing u
  ```

# Operators

- arithmetic: `a + b; a - b; a * b; a / b; a % b; -a;`
- pre/post-increment/decrement: `++a; a++; --a; a--;`
- logical: `a && b; a || b; !a;`
- comparison:

  `a == b; a != b; a < b; a > b; a <= b; a >= b;`
- bitwise: `a & b; a | b; a ^ b; ~a; a << b; a >> b;`
- ternary: `a ? b : c` equivalent to

  `if (a) { b; } else { c; }`
- assignement: `a = b;`
- compound assignement:

  `a += b; a -= b; a *= b; a /= b; ...`

See `https://en.cppreference.com/w/cpp/language/operator_precedence`

# Operators

Cave:

- unsigned integers wrap

```
std::uint8_t a = 1;
std::uint8_t b = 2;
a - b == 255;
```

- signed integer under/overflow is **undefined behaviour**
- integer division

```
int a = 1;
int b = 2;
a / b == 0;
```

- Tip: use parentheses where precedence is not clear
- more on operators in week 4

# Ifs and Loops

- `if`
- `switch`
- `while`
- `do {} while`
- `for`

# if

```
if (condition) {
    do_something();
} else if (other_condition) {
    do_something_else();
} else {
    do_something_different();
}
```

- `else if`, `else` optional

# switch

```
switch (dir) {
    case Direction::North:
        y += dy;
        break;
    case Direction::South:
        y -= dy;
        break;
    case Direction::East:
    case Direction::West:
    default:
        std::cout << "Can't move!\n";
        break;
}
```

- works only on integral types: `int`, `char`, `enum`, ...
- falls-through unless `break`
- `default` catches all; always as last item!

# while

```
while (difference > tolerance) {
    converge(); // modifies difference
}

do {
    at_least_once();
} while (false);
```

- use `break` to exit loop; `continue` to skip to next iteration

```
while (true) {
    std::cout << "Guess a number!" << std::endl;
    std::cin >> guess;
    if (guess == number) {
        break;
    }
}
```

# for

```
for (int i = 0; i < v.size(); ++i) {
    std::cout << v[i] << ' ';
}
// is syntactic sugar for (equivalent to)
{
    int i = 0;
    while (i < v.size()) {
        std::cout << v[i] << ' ';
        ++i;
    }
}
```

- parts can be left out, eg. `for (;;)`

# for

```
for (auto element : vector) { // C++11
    std::cout << element << ' ';
}
```

- just like in Python or Java

# Functions

```
int power_int(int x, unsigned int p = 2) {
    return (p == 0) ? 1 : power_int(x, p - 1);
}

// decaration
void print_array(int array[]);
// definition
void print_array(int array[]) {
    for (auto element : array) {
        std::cout << element << ' ';
    }
    std::cout << '\n';
}
```

- functions are declared with a return type (`void` if none)
- default arguments possible
- **declaration** and **definition** can be split
  - see week 2/ex02

# References and Values

```
int a = 1;
void f(int a) {
    a = 2;
}
f(a);
std::cout << a << '\n';
```

- what is the output?

# References and Values

```
int a = 1;
void f(int a) {
    a = 2;
}
f(a);
std::cout << a << '\n';
```

- what is the output? **1**!

```
void g(int & a) {
    a = 3;
}
g(a);
std::cout << a << '\n';
```

- what is the output?

# References and Values

```
int a = 1;
void f(int a) {
    a = 2;
}
f(a);
std::cout << a << '\n';
```

- what is the output? **1**!

```
void g(int & a) {
    a = 3;
}
g(a);
std::cout << a << '\n';
```

- what is the output? **3**!

# References and Values

- pass by value if you want a copy
- pass by (non-`const`) reference if you want to modify
- integral types: pass by value
- everything else: pass by `const` reference

# `const`-correctness

```
double c = 299792458; // meters per second
double mu_0 = 1.256e-6 // Henry per meter
double eps_0 = 1 / (c * c * mu_0);
double C; // capacitance

double plate_capacitor(double A, double d) {
    return eps_0 * A / d;
}

c = plate_capacitor(1e-4, 1e-3); // oops
```

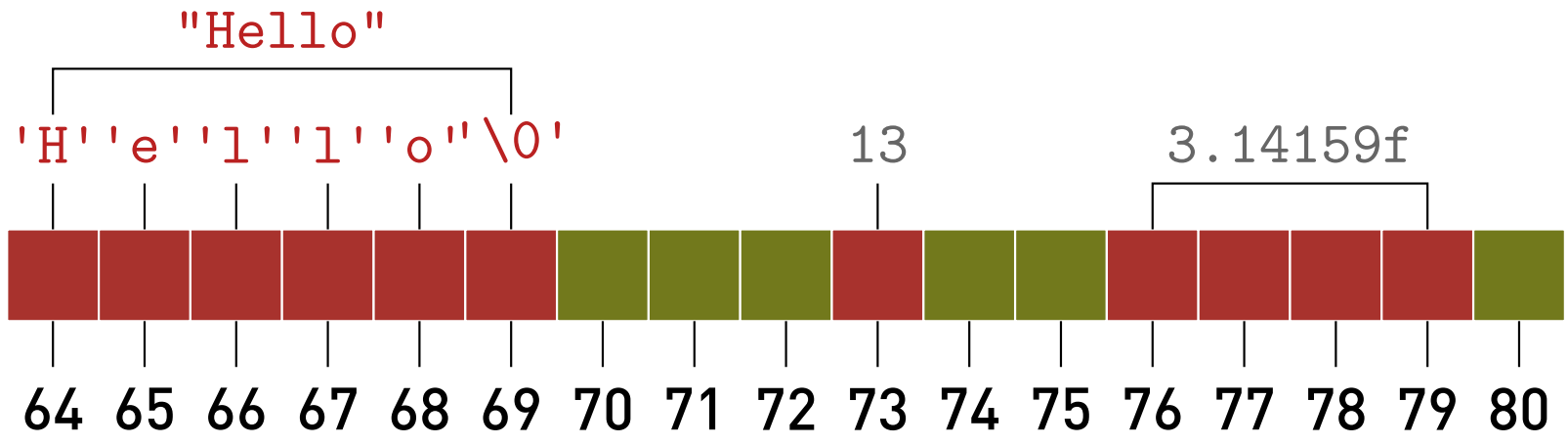
- if you don't modify it, make it `const`!

# Pointers and Memory

- Where do variables reside? In RAM/memory
- Pointer = (virtual) address
- Think of a hotel with rooms
  - single rooms (`sizeof(char) == 1`)
  - double rooms (`sizeof(std::uint16_t) == 2`)
  - quadruple rooms (`sizeof(int) == 4`)
  - ...

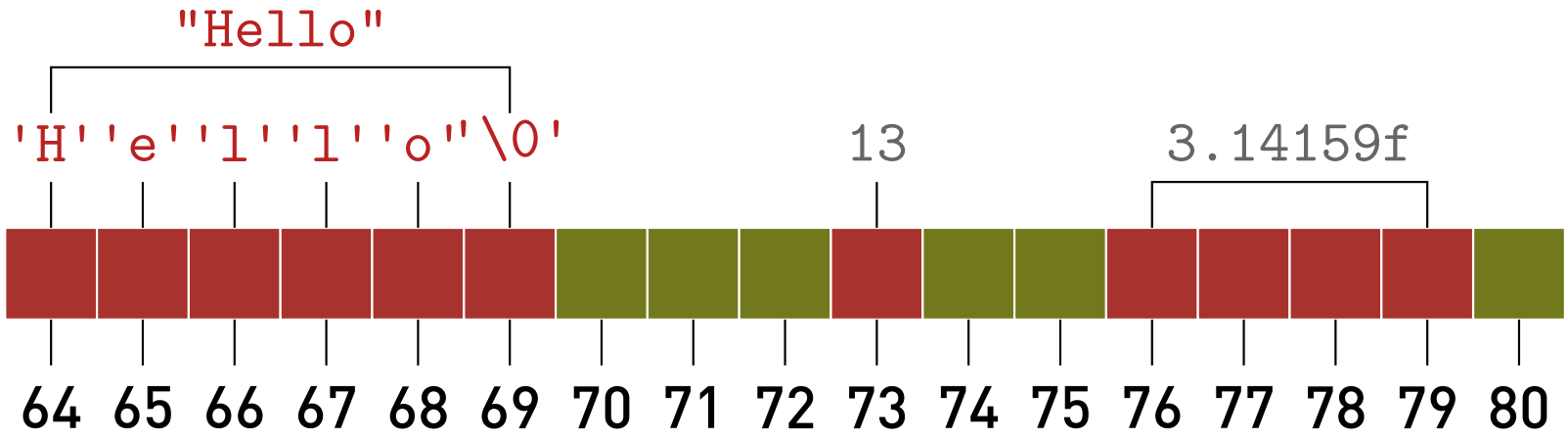
# Pointers and Memory

```
std::uint8_t a = 13;
float pi = 3.14159f;
auto hello = "Hello";
```



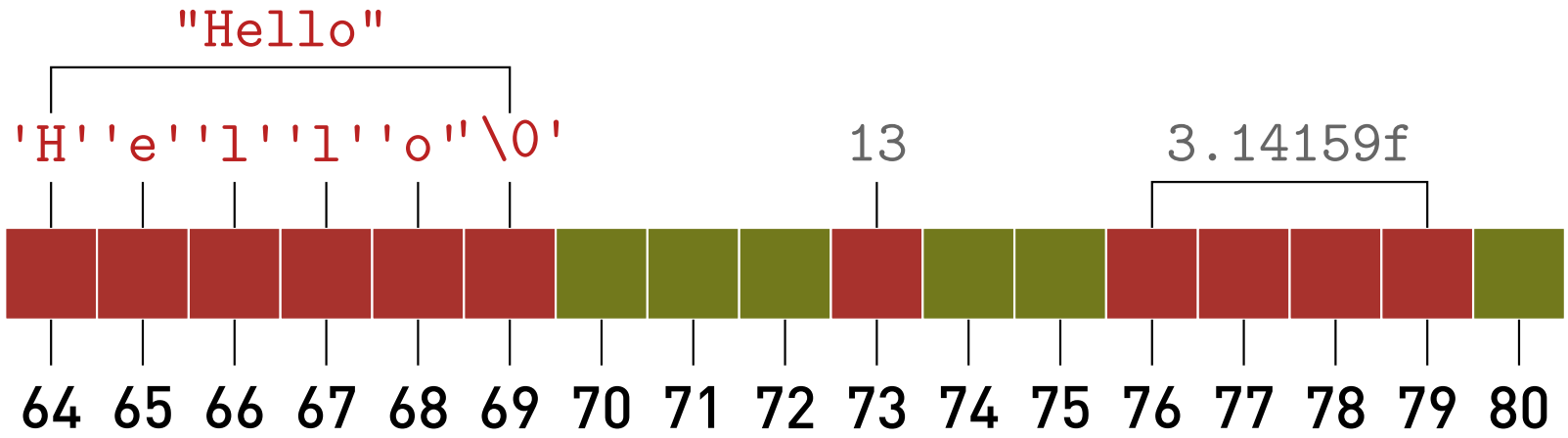

```
// `&a` is the address-of operator
std::uint8_t * a_p  = &a;  a_p   == 73;
float        * pi_p = &pi; pi_p  == 76;
*pi_p == 3.14159f; // `*pi_p` is the dereference operator
```

# Pointers and Memory




```
// C-Strings and arrays are actually pointers
hello == 64;
// `hello[i]` is syntactic sugar for `*(hello + i)`
*hello == 'H';
hello[1] == 'e';
*(hello + 4) == 'o';
```

# Pointers and Memory




Cave:

- compiler can arrage things in memory as it wants to
- arrays are always contiguous
- pointer arithmetic is in units of `sizeof`, eg.

  ```
  pi_p - 1 == 72;
  ```

- no bounds checking or type checking is done $\Longrightarrow$ giberish
- dereferencing uninitialized memory is **undefined behaviour**

# Standard Library

- `#include <iostream>`: input/output using `std::cin`/`std::cout`
- `#include <cmath>`: common math functions
- `#include <cstdint>`: fixed width types
- `#include <string>`: `std::string`
- `#include <vector>`: dynamically sized array `std::vector` (see week 7)
- `#include <algorithm>` and `#include <numeric>`: useful algorithms (find, sort, reduce, ...; see week 7)
- see `https://en.cppreference.com/w/cpp/header`

# Style & Tips

- use type aliases! → easier to refactor
- variables `const` by default
- references `const` by default
- pass by `const &` by default
- comment you code (week 2)
- be consistent in formatting (indentation, brace placement, etc.)
  - eg. use `clang-format`
- **get familiar with** `https://www.cppreference.com`
- Compiler Explorer: `https://godbolt.org` & CppInsights `https://cppinsights.io`

# Questions?

Not covered here:

- compilation pipeline (week 2)
- overloading (week 3)
- templates (week 3)
- operators (week 4)
- standard library (week 7)