Introduction

Why learn C++?

Where does the name come from?

```
C++;
```

Who invented it?

Hello World!

Let's start with the classical "Hello World!"

The code should be written on a file with the extension *cpp*. You may encounter *cxx*, however, we recomend *cpp*

In [ ]:
```
#include <iostream>
int main() { std::cout << "Hello World!" << std::endl; return 0; }
```

Let's compile and run the above code on the console.

Expressions and Statements

Both expression and statements end with a semicolumn:

> Expression;
>
> Statement;

In [ ]:
```
#include <iostream>
std::cout << "Hello World!" << std::endl;
int n = 1;                          // declaration statement
n = n + 1;                          // expression
std::cout << "n = " << n << '\n'; // expression
```

Comments

Comments are not executed, they are ignored by the compiler.

In [ ]:
```
auto a = 1, b= 3;
/* This can be a large block comment
with several lines */
if(a <= b) {
    // one line comment
    std::cout << "a is smaller" << std::endl;
} else {
    std::cout << "b is smaller" << std::endl;
}
```

Variables

A variable is a "named" storage space for data.

- You need to declare it (somewhere in the code).
  For example:
  `int a;`
- You **can** store something already at declaraion time. For example:
  `int a=4;`
- You **can** read or write something into it. For example:
  `a = c + 1; // write`
  `b = a; // read it (to store it again)`
- You can write letters, digits and underscore for this "name"
- Start with a letter
- Don't use reserved words!
  `alignas, alignof, and, and_eq, asm, auto, bitand, bitor, bool, break, case, catch, char, char16_t, char32_t, class, compl, const, constexpr, const_cast, continue, decltype, default, delete, do, double, dynamic_cast, else, enum, explicit, export, extern, false, float, for, friend, goto, if, inline, int, long, mutable, namespace, new, noexcept, not, not_eq, nullptr, operator, or, or_eq, private, protected, public, register, reinterpret_cast, return, short, signed, sizeof, static, static_assert, static_cast, struct, switch, template, this, thread_local, throw, true, try, typedef, typeid, typename, union, unsigned, using, virtual, void, volatile, wchar_t, while, xor, xor_eq`
- Variables are case sensitive.

In [ ]:
```
int numberOfApples = 5;
int b = numberofapples;
bool struct = true;
```

Types

There is a long list of types. Here is a list of the fundamental types. Most used types:

In [ ]:
```
// integral
int cards = 2;

// integral with higher precision
long long raindrops = 345593210987633;

// single precision floating point number, ie. with decimal point
float velocity = 0.4;

// double precision floating point number
double rate = 0.2983460186;

// boolean, takes true or false
bool isCool = true;

// a vector of characters
std::string salutation = "Hello World!/123"
```

Make variables constant by adding the keyword `const` infront:

In [ ]:
```
const double pi=3.1415;

pi= pi + 4.7; //compiler error
```

Arrays

Imagine you have the following data, grades stored in independent variables:

In [ ]:
```
// calculate average
double gEnglish=1.9;
double gMath=1.2;
double gPhilosophy=3.0;
double gBiology=2.3;
```

$\longrightarrow$

In [ ]:
```
double grades[4] = {1.9, 1.2, 3.0, 2.3};
```

Access the value with []. Index starts from zero!

`gEnglish` is in `grades[0]`,

`gMath` in `grades[1]` and so on...

In [ ]:
```
double sum = grades[0] + grades[1] + grades[2] + grades[3];
double average = sum / 4.0;
```

Type std::array

C++11

It works the same way as the built in `[]` array, but supports more functionality (algorithms like sorting, and common operation from containers.

`std::array<TYPE, SIZE>`

In [ ]:
```
#include <array>
std::array<double, 4> grades = {1.9, 1.2, 3.0, 2.3};
```

Converting to array with std::to_array

C++20

Converts the given array or "array-like" object to a `std::array`.

In [ ]:
```
/* Note: this doesnt compile in our notebook at the time of installtion we had up to C++17*/

std::to_array("foo"); // returns `std::array<char, 4>`
std::to_array<int>({1, 2, 3}); // returns `std::array<int, 3>`

std::array<double, 4> mygrades = std::to_array(grades);
```

Auto

C++11

Declare with the word "auto" and the compiler deduces the type for you.

For example:

In [ ]:
```
auto a = 3; /* compiler deduces to int */
auto b = a + 3.0; /* compiler deduces to double */
auto c = true; /* compiler deduces to bool */
```

In [ ]:
```
#include <iostream>
std::cout << typeid(a).name() << std::endl;
std::cout << typeid(b).name() << std::endl;
std::cout << typeid(c).name() << std::endl;
```

Decltype

C++11 C++14

It is similar to `auto` but you have to pass an expression to it. This operator returns the *declared type* of the variable.

```
decltype(expression)
decltype(auto)
```

Examples:

In [ ]:
```
int a1 = 1; // `a1` is declared as type `int`
decltype(a1) b1 = a1; // `decltype(a1)` is `int`
const int c1 = a1; // `c1` is declared as type `const int`
decltype(c1) d = a1; // `decltype(c1)` is `const int`
decltype(123) e = 123; // `decltype(123)` is `int`
```

Comparing `auto` to `decltype` :

In [ ]:
```
auto d; // error! needs initializer
```

In [ ]:
```
decltype(a1) h; // ok, no initialization needed
```

In [ ]:
```
const int x = 0;
auto x1 = x; // int
decltype(auto) x2 = x; // const int
```

Why auto or decltype?

At first sight, it doesn't seem too bad to write int, bool, double, and so on...

But it pays off when you have types from the std library, or own lengthy types.

For instance:

In [ ]:
```
std::vector<int>::iterator it1 = vec.begin();
auto it1 = vec.begin();
```

Scope & namespaces

The curly braces { } will tell us where a variable can be recognized by the compiler. They can be inserted --if syntactically correct-- anywhere inside the code. For example:

In [ ]:
```
{ int a = 3*5; } { auto b = 10 % 2; }
```

Coding style: put a statement per line and use indentations within each scope!

In [ ]:
```
{
    int a = 3*5;
}
{
    auto b = 10 % 2;
}
```

So if `block` is not on the second scope, should we be able to do this (see below)?

✔ ✗

In [ ]:
```
#include <iostream>

{
    int block = 3*5;
}
{
    auto pics = 10 % 2;
    std::cout << block << std::endl;
}
```

Namespaces

A namespace is a named scope!

```
namespace <indentifier> {
  // declarations
}
```

How do we use namespaces?

In [ ]:
```
namespace alpha {
    int a = 3*5;
}
namespace beta {
    auto b = 10 % 2;
}

#include <iostream>
std::cout << beta::b << std::endl;
{
   std::cout << alpha::a << std::endl;
}
```

Using

A convenience feature is to use `using`. We can avoid rewriting every time the namespace!

In [ ]:
```
namespace abc {
    int foo=3;
}
namespace def {
    auto bar = 8;
}

#include <iostream>
using namespace abc;
using namespace def;

std::cout << foo << std::endl;
std::cout << bar << std::endl;
```

You can nest namespaces

In [ ]:
```
namespace A {
  namespace B {
    namespace C {
      int i=2;
    }
  }
}
```

Or nest them like this:

C++17

In [ ]:
```
namespace A::B::C {
  int i=2;
}
```