C++
Introduction

Goals

- Understand the fundamentals concepts of C++
- Study the differences between C++, Java and C

Quick reading
- Read and try to grasp the main ideas

Read
- Read and understand the explained concepts

Study
- Read, understand and remember the concepts, the rules and the principles.

Don't be afraid to try (compile, execute, modify, debug) the proposed examples!

Beginning with C++

```
int main()
{
}
```

Note: like in C, this is also valid:

```
int main(int argc, char *argv[]);
```

```
#include <iostream>

int main()
{
        std::cout << "Hello world" << std::endl;
}
```

Compiling C++ code

```
g++ -std=c++14 -Wall -o out source.cpp
```

C++ 14 standard

```
g++ -std=c++17 -Wall -o out source.cpp
```

C++ 17 standard

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.5)

project(ALF2024 LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

add_executable(ALF2024 main.cpp)

include(GNUInstallDirs)
install(TARGETS ALF2024
    LIBRARY DESTINATION ${CMAKE_INSTALL_LIBDIR}
    RUNTIME DESTINATION ${CMAKE_INSTALL_BINDIR}
)
```

Project name

`set(parameter value)` is used to the value to a parameter

This adds a new target (executable file) based on the given source files

Diving into C++

In C++ header files have no extension

```
#include <iostream>

int main()
{
        std::cout << "Hello world" << std::endl;
}
```

The << operator allows for writing on the standard output

std:: "enters" the namespace where cout and endl are defined

:: is called "scope resolution operator“

Input and output streams

- We can access input and output streams using the following objects:
  - **`cin`**, standard input (keyboard)
  - **`cout`**, standard output (terminal)
  - **`cerr`**, standard error (terminal)

  - Those object are defined within the ***`std`*** namespace, and are declared in ***`iostream`***

You will "explore" streams in the first exercise series

Input and output

- The *cin*, *cout*, e *cerr* define some methods and operators:

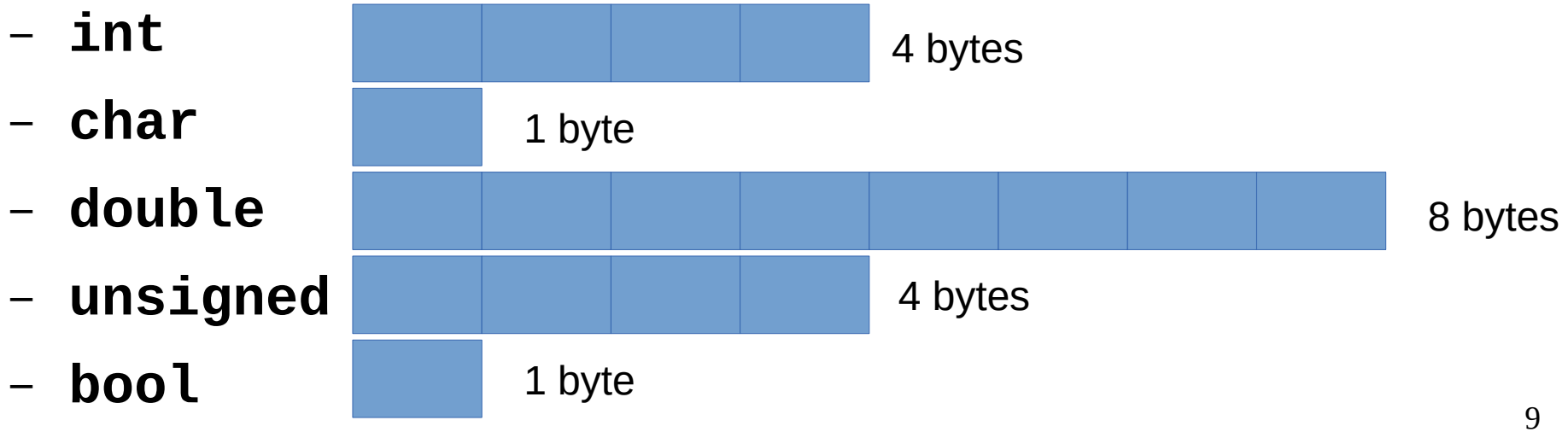
```
std::cout << "Hello world" << std::endl;
std::cin >> name;
```

- C++ supports operator overloading: the shift operators **<<** and **>>** are *overloaded* by the *ostream* classes (those of cout, cerr) and *istream* classes (cin)
  - **endl** is a manipulator that inserts a newline character into the stream and forces a flush
- Concerning binary operators, the equivalent syntax is:

```
obj.operator<<(param);
operator<<(obj, param2);
```

Types and variables

- Fundamental types in C++ are the same as in C (and similar to those of Java)
  - Each type has a specific size (which can be obtained with **`sizeof`**) depending on the architecture of the target machine



Overloading

- Like Java, C++ supports function overloading

```
void write(int x)
{
	cout << "int=" << x << endl;
}

void write(double x)
{
	cout << "double=" << x << endl;
}

void write(int x, int y)
{
	cout << "int=" << x << " int=" << y << endl;
}
```

Name conflicts

```
#include <iostream>


int multiply(int a, int b)
{
    return a*b;
}



int multiply(int a, int b)
{
    return a*b*2;
}


int main()
{
       std::cout << multiply(3,2) << std::endl;
}
```

Redefinition of multiply

Arguments with a default value

```
/* in C */
int sum(int a, int b, int c, int d)
{
        return (a+b+c+d);
}


int x = sum(3,4,0,0);
```

In C (and Java) it is mandatory to pass all the arguments

Arguments with a default value

```
/* in C++ */
int sum(int a, int b=0, int c=0, int d=0)
{
        return (a+b+c+d);
}
int x = sum(3,4);
int y = sum(3);
int z = sum(3,4,5);
int w = sum(3,4,5,6);


int produce_output(double q = 0.0, char* currency="CHF")
{
        cout << q << " " << currency << endl;
}
int produce_output(double q = 0.0, char* currency)
```

We assign a default value to some of the arguments

Error! Cannot declare arguments with default value before arguments with no default

Arguments with a default value

```
int sum(int a, int b=0, int c=0, int d=0);
int sum(int a, int b, int c, int d)
{
        return (a+b+c+d);
}
```

If we have a separate declaration, default values go into the declaration, not the implementation

Ambiguity

```
int foo(int a)
{
    return a;
}



int foo(int a, int b = 0)
{
    return a+b;
}


int main()
{
       std::cout << foo(3,2) << std::endl;
       std::cout << foo(5) << std::endl;
}
```

Ambiguous

Name conflicts (more subtle)

```
// mymath.h
int multiply(int a, int b);

// mymath.cpp
#include "mymath.h"
int multiply(int a, int b) {
    return a*b;
}
```

```
// main.cpp
#include <iostream>
#include "mymath.h"
int multiply(int a, int b) {
    return a*b*2;
}

int main() {
       std::cout << multiply(3,2) << std::endl;
}
```

Namespaces

Namespace definition

```
// mymath.h
namespace supsi {
int multiply(int a, int b);
}

// mymath.cpp
#include "mymath.h"
namespace supsi {
int multiply(int a, int b) {
    return a*b;
}
}
```

```
// main.cpp
#include <iostream>
#include "mymath.h"
int multiply(int a, int b) {
    return a*b*2;
}

int main() {
       std::cout << multiply(3,2)
                 << supsi::multiply(5,6) << std::endl;
}
```

Namespaces

```
#include <iostream>
#include "mymath.h"

namespace supsi {
	int sum(int a, int b)
	{
		return a+b;
	}
}

namespace dti {
	int multiply(int a, int b)
	{
		return a*b;
	}
}

int main()
{
	std::cout << dti::multiply(3,2)
			<< supsi::multiply(3,2)
			<< supsi::sum(7,3)
			<< std::endl;
}
```

Namespaces

```
#include <iostream>

int multiply(int a, int b)
{
     return a*b;
}

namespace supsi {
       int multiply(int a, int b)
       {
              return ::multiply(3,2);
       }
}


int main()
{
       std::cout << multiply(3,2)
                         << supsi::multiply(3,2)
                         << std::endl;
}
```

Call the function defined in the global namespace

Nested namespaces

```
#include <iostream>

namespace supsi {
        namespace dti {
                int multiply(int a, int b)
                {
                        return a*b;
                }
        }
}

int main()
{
        std::cout << supsi::dti::multiply(3,2)
                                << std::endl;
}
```

Namespace alias

```
#include <iostream>

namespace supsi {
        namespace dti {
                int multiply(int a, int b)
                {
                        return a*b;
                }
        }
}

namespace xyz = supsi::dti;

int main()
{
        std::cout << xyz::multiply(3,2)
                                << std::endl;
}
```

Using

```
#include <iostream>

namespace supsi {
        namespace dti {
                int multiply(int a, int b)
                {
                        return a*b;
                }
        }
}

using namespace std;
using namespace supsi::dti;

int main()
{
        cout << multiply(3,2)
                        << endl;
}
```

Makes the names defined in those namespace part of the current scope

Using

```
#include <iostream>

namespace supsi {
        namespace dti {
                int multiply(int a, int b)
                {
                        return a*b;
                }
        }
}


int main()
{
        using namespace std;
        using namespace supsi::dti;
        cout << multiply(3,2)
                        << endl;
}
```

Makes the names defined in those namespace part of the current scope

Variable initialization

```
double pi_a = 3.14;
```
C style

```
double pi_b(3.14);
```
C++ style (pre -11)

```
double pi_c = {3.14};
```
C++-11 style

```
double pi_d {3.14};
```
C++-11 style

Prefer {} to prevent narrowing conversions

- When initializing a variable we can unfortunately lose some information due to **narrowing conversions**
  - If we initialize with { } we get a compiler error

```
int main() {
        int pi_a = 3.14; // Becomes 3!
        int pi_b {3.14}; // Error!
}
```

Automatic type inference

- The compiler can infer the type of a value, hence we can use **auto** instead of an explicit declaration

```
auto pi{3.14}; // double
auto x{42}; // int
auto t{true}; // bool
auto f{false}; // bool
auto k{multiply(4,2)}; // return type
of multiply
```

* from C++14 it is possible to use auto also as return type of a function (→ determined from the return statement)