C++
Introduction

Goals

- Understand the fundamentals concepts of C++
- Study the differences between C++, Java and C
- Study the concept of reference
- Understand how to use strings and streams

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()
{
}
```

```
#include <iostream>

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

Compiling C++ code

```
g++ -std=c++11 -Wall -o out source.cpp
```

C++ 11 standard

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

C++ 14 standard

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

C++ 17 standard

Building with a specific standard

- To compile a project using QMake with a specific language standard, add the following configuration line

```
CONFIG += c++14
```

or

```
CONFIG += c++17
```

On MacOSX you might need:

```
QMAKE_CXXFLAGS += -std=c++14 -stdlib=libc++ -mmacosx-version-min=10.7
LIBS += -stdlib=libc++ -mmacosx-version-min=10.7
```

Or

```
QMAKE_CXXFLAGS += -std=c++17 -stdlib=libc++ -mmacosx-version-min=10.7
LIBS += -stdlib=libc++ -mmacosx-version-min=10.7
```

Building with a specific standard

- To compile a project using CMake with a specific language standard, add the following configuration line

```
set(CMAKE_CXX_STANDARD 17)

set(CMAKE_CXX_STANDARD_REQUIRED ON)
```

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`***

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);
```

Namespaces

```
#include <iostream>

namespace supsi {

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

}

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

Namespace definition

Namespaces

```
#include <iostream>

namespace supsi {
	int multiply(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)
			<< 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

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 global namespace

Functions

- Like in C, functions need to be declared before use:

```
int multiply(int, int);
Data* read();
void reset();
```

In the declaration, only the type of the arguments is necessary

Functions

```
#include <iostream>

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

int main()
{
        std::cout << multiply(3.14,2)
                                << std::endl;
}
```

A separate declaration can be omitted if the function is referenced only later in the file

Functions

```
#include <iostream>

void write()
{
	std::cout << multiply(3.14,2) << std::endl;
}

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

int main()
{
	write();
}
```

Error! Multiply is not yet declared

Functions

```
#include <iostream>

int multiply(int, int);

void write()
{
        std::cout << multiply(3.14,2)
                                << std::endl;
}

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

int main()
{
        write();
}
```

Declaration

Implementation

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;
}
```

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)
```

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

Ambiguities

```
#include <iostream>
#include <string>
using namespace std;

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

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

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

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

int main() {
        write(2.5);
        write(3);
        write(5, 3.14);
        write(23.5, 11);
        write(42, 42);
}
```

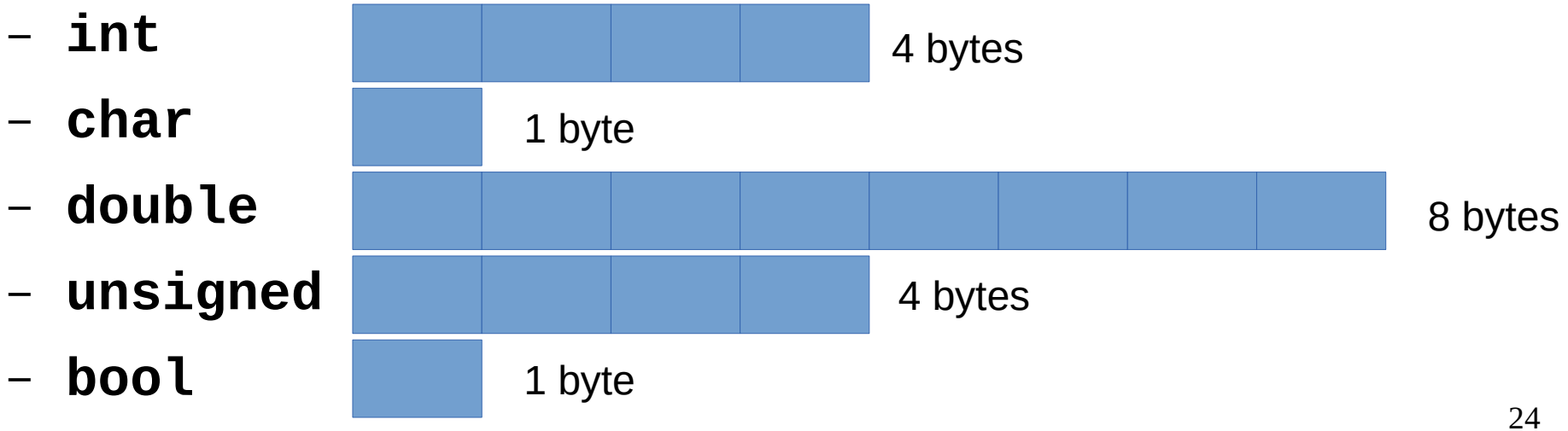
This is ambiguous

To fix, we rewrite as:

```
write( int(42), double(42));
```

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



string

- Partof the C++ standard library (std)
- Manages all the details of memory allocation

```
#include <string>
#include <iostream>
// ref.
http://www.cplusplus.com/reference/string/stri
ng/
using namespace std;

int main(void) {
        string msg{"Hello world!"};
        // string str('x'); // Error
        cout << msg << endl
                << msg.length() << endl
                << msg.empty() << endl;
        string ciao{"Hello"}, mondo{"world"};
        string result;
        result = ciao + " "+ mondo + "!";
        if (result == msg) {
                cout << "Equal" << endl;
        }
        // Character access (by index)
        cout << result[3] << endl;
```

string

- Partof the C++ standard library (std)
- Manages all the details of memory allocation

```
// Inserting a string inside a string
result.insert(5, "large");
cout << result << endl;
// Extracting substrings
cout << result.substr(1,3) << endl;
// Erasing substrings
result.erase(3,6);
cout << result << endl;
// Replacing substrings
result.replace(0,5, "Hello");
cout << result << endl;
// Searching for a substring
cout << result.find("world", 0) << endl;
}
```

Conversions

- When we assign a value, call a function or we perform arithmetical operations the compiler might perform automatic conversions

```
#include <iostream>

using namespace std;

int main() {
        double pi = 3.14;
        int r = 4;
        int c = 2 * r * pi;
        cout << "Circumference: " <<
                    c << endl;
}
```

The mathematical computation is performed with the highest precision, but the result is converted to an int

Initializing a variable

```
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

Preventing loss of information

- When initializing a variable we might already lose some information due to **narrowing conversions**
  - However, if we initialize with { } we get a compiler error

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

auto

- 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 by the return statement)

Array

- An **array** is declared as in C, for example:

```
char c[10];
```

Array of 10 characters

- The size of the array must be a **const** expression
- The contentsof the array can be set by initializing with { } :

```
int myarray[5] { 1, 5, 3, 6, 2};
int myarray[] { 1, 5, 3, 6, 2};
```

Array iteration

```
#include <iostream>

using namespace std;

int main(void) {
        int myarray[] { 1, 5, 3, 6, 2};

        for (auto i=0; i<5; i++) {
                cout << myarray[i] << endl;
        }

        for (auto i : myarray) {
                cout << i << endl;
        }

        for (auto i : { 1, 5, 3, 6, 2}) {
                cout << i << endl;
        }
}
```

“Traditional”

C++-11

C++-11

Memory allocation

- Pointers are typically related to manual memory allocation:
  - In C we use **malloc** and **free** to allocate, respectively deallocate heap memory
  - In C++ we use the following operators:
    - **new**
    - **delete**
- For example:
  - `int* i0{new int};`
  - `char* p0{new char[10]}; // Array`
  - `int* p1{new int[5]}; // Array`
  - `delete i0;`
  - `delete[] p0;`
  - `delete[] p1;`

Example

New type introduced in C++11
(can be converted to boolean)

```
int main(void) {
 int* ip{nullptr};   /* null pointer */
 ip = new int;  /* allocation */
 int* jp{new int{13}};     /* allocation and
initialization */
 //...
 delete ip;
 delete jp;
}
```

Memory management

- In C++ there is no garbage collector, memory allocated by the programmer must be deallocated explicitly *!
  - … and dereferencing an invalid (dangling) pointer can crash the program!
- Instead of allocating memory on the heap we can, whenever possible, employ other strategies
  - Allocated on the stack
  - Pass by reference

* unless we use smart-pointers

Why on the stack

- As a programmer, allocating on the stack is trivial: no explicit allocation and deallocation is required!
  - Objects on the stack are freed when we exit their scope
    - Local scope: end of the function
    - Class scope: when the instance of the class is destroyed
    - Namespace scope: when the program ends