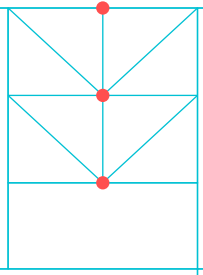
Introduction to C++ and Object-Oriented Programming

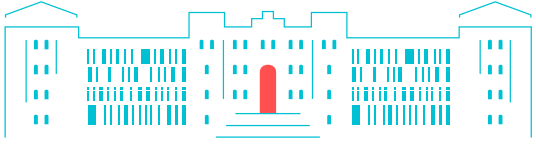
Programmierparadigmen – Objektorientierte Programmierung (OOP)

Week 1/14 – version 26.0

TUHH
Hamburg
University of
Technology

Institute for Autonomous Cyber-Physical Systems





Thibaut Lunet

History and evolution of C++

TUHH

Origin and motivations

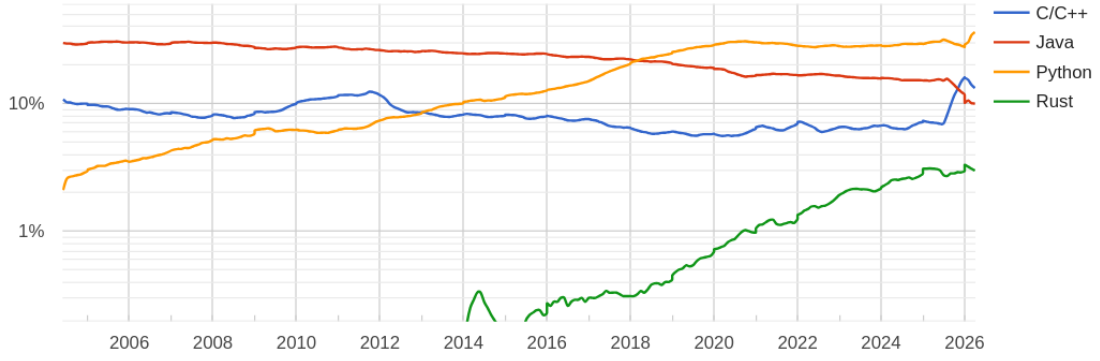
- Created in 1979 by Bjarne Stroustrup (Danish)
- Extension of C to **Object-Oriented Programming**
- Provides **low-level capacities** (close to machine instructions)
  ... and many **high-level features** (Standard Library)

Evolution

- First standardization in 1998 (C++98)
- Genesis of "modern" C++ with C++11
- Constant evolution since (C++14, **C++17**, C++20, C++23, ...)

C++ among others ...

PYPL Popularity of Programming Languages – *how often a tutorial is searched on Google*



⇒ C/C++ and Python are predominant, while Java decreases slowly ... see also TIOBE Index

... ok but for what ?

Use of C++ in real-life applications

Operating Systems

Embedded Systems

Video Games

Softwares and Apps

Web Systems

Basic C++ program

TUHH

C code

```c
// File : program.c
#include <stdio.h>

int main() {
    printf("Hello World!");
    return 0;
}
// Compiled with : gcc program.c -o program.exe
```

C++ code

```cpp
// File : program.cpp
#include <iostream>

int main() {
    std::cout << "Hello World!" << std::endl;
    return 0;
}
// Compiled with : g++ program.cpp -o program.exe
```

- define a `int main()` function that is executed by the **program**
- `return` keyword to end the program / function execution (optional in `main`)
- **Note :** use of the `.exe` extension as a convention to identify the **executable**

From C to C++

- no `#include <stdio.h>` or `printf()` for output, but "streams"
- link to the Standard Library with `std` keyword, but **NEVER** use `using namespace std` !

Base Definitions (1/3)

TUHH

Object

→ memory element of the program that stores a **value** with a given **type**.

Variable = simply a **named object**.

```cpp
#include <iostream>

int main() {
    int var = 1978;     // var is a variable, of type int, storing the value 1978

    // printing in the console displays the value stored by var ...
    std::cout << "Value of var : " << var << std::endl;
}
```

[Output] *Value of var : 1978*

Base Definitions (2/3)

TUHH

Type

→ group of authorized **values** and operations for different **objects**.

Declaration : register one **variable** with a given **type** in program's memory.

```
1  int v1 = 1; int v2 = 2;      // int variable can store integer values
2  int v3 = "Message";          // ... but cannot store strings (compilation error here !)
3
4  char* m1 = "Message";
5  char* m2 = " very important";     // -> strings are stored with char* types
6
7  v1 + v2;    // addition : allowed between two int
8  m1 + m2;    // ... but not allowed between two char* (compilation error here !)
9
10 // Declaration
11 int n;      // -> of an integer variable
12 char* s;    // -> of a string variable
```

Base Definitions (3/3)

TUHH

Value

→ content associated to a **variable** in program's memory.

Assignment : set a **value** for a **variable**.

Note : declaration **always assigns** a value to a variable !

→ **value initialization** : value is explicitly given when declaring the variable

→ **default initialization** : variable is assigned during declaration to a default value ...

```cpp
#include<iostream>

int main() {
    int var = 1;     // declare var, value initialization with 1
    var = 2;         // assignment of the value 2 to var

    int declVar;                          // declaration only, default initialization
    std::cout << declVar << std::endl     // ... it has a value, but can be anything !
    declVar = 0;                          // assignment to a given value
}
```

Output of this code is an **undefined behavior** !

First Examples in C++

TUHH

Initializing variables of the standard types

```cpp
int n = -1000;                                  // Signed integer
size_t i(1);                                    // Unsigned long integer

float x = 1.0;                                  // Single precision floating point number (TO BE AVOIDED !)
double y{3.141592653589793};                    // Double precision floating point number

bool cond(true);                                // Boolean, can also be false
char letter = 'a';                              // Character (not a string !)
```

From C to C++

- equivalencies : `int n{1};` ⇔ `int n = 1;` ⇔ `int n(1);`

  → for now, favor `int n = 1;` for standard types ...

To go further with types ...

Standard arithmetic operators ...

Quick wake-up quiz

Consider this example of variable declarations :

```
1 int i;
2 int j{};
3 int k();
```

What is the most important fact that you should always remember ?

Quick wake-up quiz

Consider this example of variable declarations :

```
1 int i;
2 int j{};
3 int k();
```

What is the most important fact that you should always remember ?

A	C-way of declaring a variable also works in C++

Quick wake-up quiz

TUHH

Consider this example of variable declarations :

```
1 int i;
2 int j{};
3 int k();
```

What is the most important fact that you should always remember ?

A	C-way of declaring a variable also works in C++
B	Variable `k` is not an integer

Quick wake-up quiz

Consider this example of variable declarations :

```
1 int i;
2 int j{};
3 int k();
```

What is the most important fact that you should always remember ?

A	C-way of declaring a variable also works in C++
B	Variable `k` is not an integer
C	Each variable declared here contains a default value

Quick wake-up quiz

Consider this example of variable declarations :

```
1 int i;
2 int j{};
3 int k();
```

What is the most important fact that you should always remember ?

A	C-way of declaring a variable also works in C++	**B**	Variable `k` is not an integer
C	Each variable declared here contains a default value	**D**	This is a hopeless tentative to bring some animation in this lecture ...

Quick wake-up quiz

Consider this example of variable declarations :

```
1 int i;
2 int j{};
3 int k();
```

What is the most important fact that you should always remember ?

Answer B *(but every answers are true !)*

→ `int k();` **declares a function** that takes no argument and returns an `int`

A	C-way of declaring a variable also works in C++	**B**	Variable `k` is not an integer
C	Each variable declared here contains a default value	**D**	This is a hopeless tentative to bring some animation in this lecture ...

Functions

TUHH

Definition

Reusable block of code designed to perform a specific task → self-contained module that :

1. has a name and may accept input data (**parameters** or **arguments**) with given type(s)
2. performs a sequence of defined and / or conditional operations (**instructions**)
3. may return a result (**output**) with a given type

```
1 int myFunction(int a, double b) {        // conditional operations
2     if (b < 0) { return a; } else { return -a + 1; }
3 }
4 int generateNumber() {                   // takes no argument
5     return 2
6 }
7 void printNumber(int a) {                // returns nothing
8     std::cout << "Your number is : " << a << std::endl;
9 }
```

DRY principle : *"Don't Repeat Yourself"*

Function Declaration VS Function Definition

TUHH

Declaring a function means ...

- you define the **signature** of this function → number and types of arguments, return type

```
1 int myFunction(int a, double b);        // takes two arguments (int, double), returns an int
2
3 int myFunction(int, double);            // equivalent declaration, since it does not care about variable names
```

Function Declaration VS Function Definition

TUHH

Declaring a function means ...

- you define the **signature** of this function → number and types of arguments, return type

```
1 int myFunction(int a, double b);        // takes two arguments (int, double), returns an int
2
3 int myFunction(int, double);            // equivalent declaration, since it does not care about variable names
```

Defining a function means ...

- you define the **body** of this function → set of instructions depending on the arguments

```
1 int myFunction(int a, double b) {
2     if (b < 0) { return a; } else { return -a + 1; }
3 }  // no semicolums ; here !
```

⇒ allows to **define functions in two steps** (not mandatory, but useful sometimes ...)

Conditions and Loops

TUHH

Conditional

```cpp
int a = 2;
if ( ((a % 2) == 0) and (a > 0) ) {
    std::cout << "a is even and strictly positive" << std::endl;
} else {
    std::cout << "a is odd or negative" << std::endl;
}
```

Loop

```cpp
for (size_t i = 0; i < 10; i++) {
    std::cout << "square of " << i << " is " << i*i << std::endl;
}
```

From C to C++

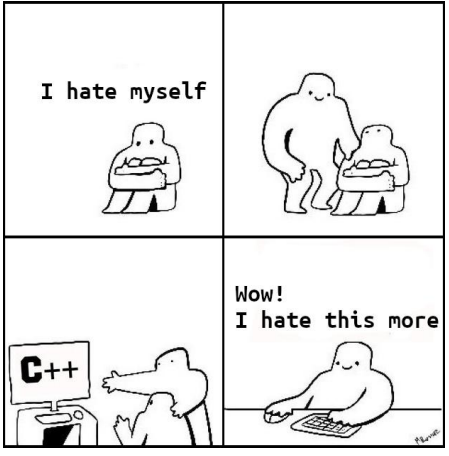
- More English (`and`, `or`, `not`...), even if symbols still work (`&&`, `||`, `!`, ...)

More on logical operators ...

Other statements like **`switch`**, **`while`**, **`break`**, **`continue`**, ...

What brings C++, that really differs from C ?

Besides ...



C++ is more practical than C - Reason 1

TUHH

No need of C-Strings anymore, just use C++ strings from the standard library :

```cpp
#include <string>
using std::string;      // -> avoid writing std::string every time
// ------------------------------------------------------------------------
string s1 = "Hello";
string s2;                              // default value is ""
s2 = "bob";
s2 = "there !";                         // nah, changed my mind !
// ------------------------------------------------------------------------
string s3 = s1 + " " + "there !";       // concatenate strings
bool test = (s3 == "Hola !");           // test their value
// ------------------------------------------------------------------------
char letter = s3[0];                    // extract one character
size_t len = s3.size();                 // get the string size
// ------------------------------------------------------------------------
size_t pos = s3.find("you");            // find a substring
char[] cs3 = s3.c_str();                // if you're nostalgic ...
// ... and so much more ...
```

To go further :

C++ strings VS C-strings docs for `std::string`

C++ is more practical than C – Reason 2

Need an array of values ? Just use vectors from the Standard Template Library (STL) :

```cpp
#include <vector>
using std::vector;      // -> avoid writing std::vector every time
// ---------------------------------------------------------------------------
vector<double> listDouble = {1.2, 2.3, 3.4};       // pre-defined list
vector<string> listStr(5, "Nothing");              // 5 times "Nothing"
// ---------------------------------------------------------------------------
vector<int> listInt1(5);                           // initialize a vector of size 5
for (size_t i = 0; i < listInt1.size(); i++) {
    listInt1[i] = i*2;                             // classical element access (index starts at 0)
}
// ---------------------------------------------------------------------------
vector<int> listInt2;                              // most flexible way for initialization
for (size_t i = 0; i <= 30; i+= 3) {
    listInt2.push_back(i);                         // add elements one by one
}
```

docs for `std::vector`

Fixed size array : `array` → similar as `vector`, without size modifications

docs for `std::array`

C++ is more practical than C - Main Reason

TUHH

Allows Object-Oriented Programming :

→ `std::string` and `std::vector` are actually **objects** of a given **class** !

Some first examples :

```
1 string s = "Hello";
2 size_t len = s.size();
```

⇒ `size` is a function specific to `string` that returns an integer, also called **method**

```
1 vector<double> lDouble = {1.2, 2.3, 3.4};       // pre-defined list
2 vector<string> lStr(5, "Nothing");              // 5 times "Nothing"
3 vector<int> lInt1(5);                           // initialize a vector of size 5
```

⇒ can use different **constructors** to initialize a variable

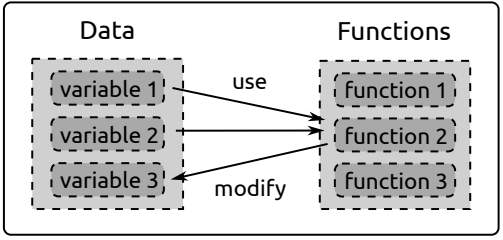
```
1 string s3 = s1 + " " + "there !";       // concatenate strings
2 bool test = (s3 == "Hola !");           // test their value
```

⇒ uses **operator overloading** to facilitate operations and manipulations

Motivation for Object Oriented Programming : Problematic

Starting basis : Procedural Programming

- separate data from functions
- each function uses / modifies global data



Ex : string manipulation (C)

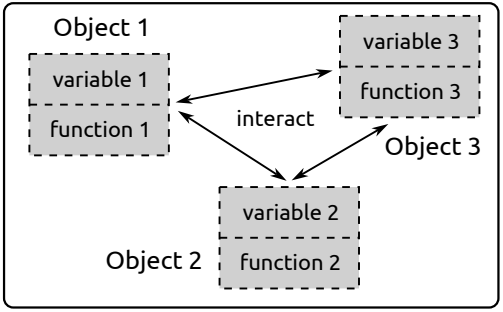
```c
#include <stdio.h>
#include <string.h>

int main () {
    char str1[] = "baba";
    char str2[] = "orum";

    int size = strlen(str1);
    printf("Size of str1: %d\n", size);

    strcat(str1, str2);
    printf("Concat. string: %s\n", str1);
    // str1 is modified, str2 is not !
}
```

⇒ main inconvenient : **increases complexity** of a change for **large programs**

Motivation for Object Oriented Programming : Solution

OOP approach : regroup similar data and associated functions

→ objects are "features" that interact during program execution



Ex : string manipulation (C++)

```cpp
#include <iostream>
#include <string>
using std::cout, std::endl, std::string;

int main () {
    string str1 = "baba";
    string str2 = "orum";

    int size = str1.size();
    cout << "Size of str1: " << size << endl;

    str1 += str2;
    cout << "Concat. string: " << str1 << endl;
}
```

⇒ introduction of a new **programming paradigm**

To conclude

Main takeaways :

1. C++ and C share a **similar syntax**
   → *except for some rare cases, C code works with C++ compiler*
2. C++ can **make your life easier** than C
   → *facilities with standard libraries and modern standards*
3. C++ main utility is its **Object Oriented Programming** capability
   ... while staying a relatively low-level language

Glossary

- program, executable
- `std`, streams
- object, variable
- type, declaration
- value, assignment
- value initialization, default initialization, undefined behavior
- function, argument, output, instructions, DRY
- function declaration, signature, function definition, body
- conditional, loops
- `string`, `vector`, STL