# Introduction to Programming

## Lecture 1 — What's a program?

**Nicolas Gry**

Université de Saint-Étienne | Year 2026/2027

digicrea digital creativity

Université Jean Monnet Saint-Étienne

# 00 Foreword

# Contact information

Nicolas Gry
Located in Lyon, at the CITI laboratory
email: **nicolas.gry@inria.fr** – nicolas.gry@insa-lyon.fr

# A little warning

First time teaching this module

# A little warning

First time teaching this module

If you need me to slow down, shout it out!

# What this course contains...

- 24 hours of lecture and practise

# What this course contains...

- 24 hours of lecture and practise
- The (very) basics of C++

# What this course contains...

- 24 hours of lecture and practise
- The (very) basics of C++
- How to use tools programmed by others

# What this course contains...

- 24 hours of lecture and practise
- The (very) basics of C++
- How to use tools programmed by others
- How to produce small programs that do (almost) exactly what you want

# What this course contains...

- 24 hours of lecture and practise
- The (very) basics of C++
- How to use tools programmed by others
- How to produce small programs that do (almost) exactly what you want
- Some starter knowledge on Javascript

# What this course contains...

- 24 hours of lecture and practise
- The (very) basics of C++
- How to use tools programmed by others
- How to produce small programs that do (almost) exactly what you want
- Some starter knowledge on Javascript

The general guideline is: you should be able to understand small bits of code, and be able to produce code that does what you intended

# . . . and what it doesn't

- Complex C++ concepts

# . . . and what it doesn't

- Complex C++ concepts
- Computer Science engineering

# . . . and what it doesn't

- Complex C++ concepts
- Computer Science engineering

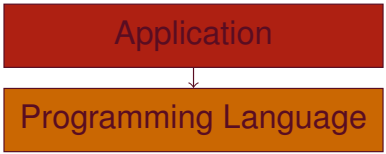
You will not become a developper after this class. However, you should have the tools and methodology to dig deeper if you'd like.

# 01 Introduction

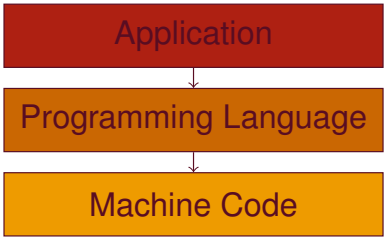
# The different layers of your program

# The different layers of your program

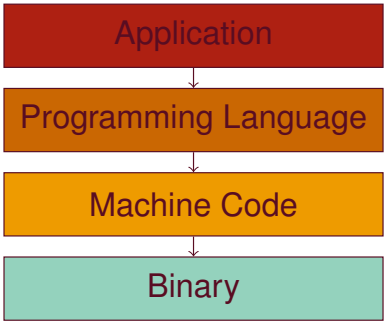
# The different layers of your program



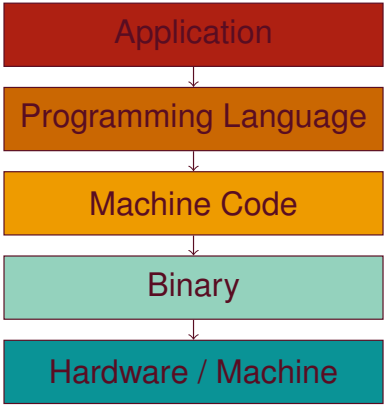
# The different layers of your program



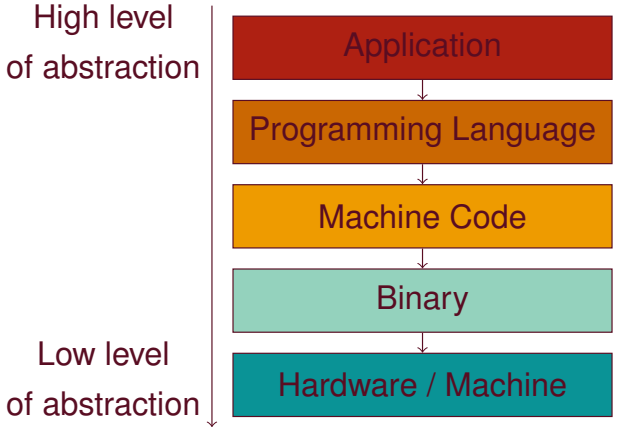
# The different layers of your program



# The different layers of your program



# The different layers of your program



# Writing a program

It is very much like writing a recipe

# Writing a program

It is very much like writing a recipe

- You declare ingredients that will be used

# Writing a program

It is very much like writing a recipe

- You declare ingredients that will be used
- You give instructions on what to do with those ingredients

# Writing a program

It is very much like writing a recipe

- You declare ingredients that will be used
- You give instructions on what to do with those ingredients
- Sometimes, the ingredients will vary but the cooking method will be the same

# Writing a program

It is very much like writing a recipe

- You declare ingredients that will be used
- You give instructions on what to do with those ingredients
- Sometimes, the ingredients will vary but the cooking method will be the same
- If everything goes well, the cook will bake your mix in the oven and after some time, a cake will come out of it

# Writing a program

It is very much like writing a recipe

- You declare **variables** that will be used
- You give instructions on what to do with those ingredients
- Sometimes, the ingredients will vary but the cooking method will be the same
- If everything goes well, the cook will bake your mix in the oven and after some time, a cake will come out of it

# Writing a program

It is very much like writing a recipe

- You declare **variables** that will be used
- You give **instructions** on what to do with those **variables**
- Sometimes, the ingredients will vary but the cooking method will be the same
- If everything goes well, the cook will bake your mix in the oven and after some time, a cake will come out of it

# Writing a program

It is very much like writing a recipe

- You declare **variables** that will be used
- You give **instructions** on what to do with those **variables**
- Sometimes, the **variables** will vary but the **method** will be the same
- If everything goes well, the cook will bake your mix in the oven and after some time, a cake will come out of it

# Writing a program

It is very much like writing a recipe

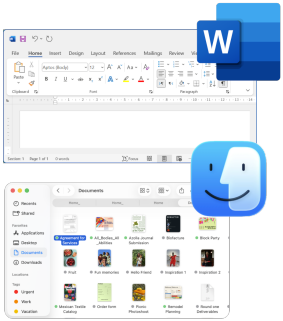
- You declare **variables** that will be used
- You give **instructions** on what to do with those **variables**
- Sometimes, the **variables** will vary but the **method** will be the same
- If everything goes well, the **compiler** will **compile** your **code** and after some time, a **program** will come out of it

# The programs you never see

You are probably used to programs that have **Graphical User Interface (GUI)**.

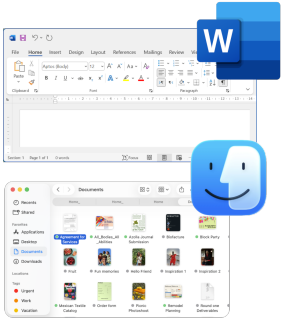
# The programs you never see

You are probably used to programs that have **Graphical User Interface (GUI)**.



# The programs you never see

You are probably used to programs that have **Graphical User Interface (GUI)**.



Some programs don't have a GUI ; they can be executed / run in a **terminal**.

# Your first program

The "Hello World" program !

# The entry point of your program

Every C++ program contains a "function" that is called `main`. It is the starting point of your program:

```
int main() {
    // your program starts here
    return 0;
}
```

# The end of your programm

The `main` function returns an integer value:

```
return 0;
```

Just like someone eating your cake would give feedback on the taste:

# The end of your programm

The `main` function returns an integer value:

```
return 0;
```

Just like someone eating your cake would give feedback on the taste:

- If the program runs well, it returns zero

# The end of your programm

The `main` function returns an integer value:

```
return 0;
```

Just like someone eating your cake would give feedback on the taste:

- If the program runs well, it returns zero — the person will want to have some of your cake again

# The end of your programm

The `main` function returns an integer value:

```
return 0;
```

Just like someone eating your cake would give feedback on the taste:

- If the program runs well, it returns zero — the person will want to have some of your cake again
- Otherwise, the program might:

# The end of your programm

The `main` function returns an integer value:

```
return 0;
```

Just like someone eating your cake would give feedback on the taste:

- If the program runs well, it returns zero — the person will want to have some of your cake again
- Otherwise, the program might:
  - Return a non-zero value

# The end of your programm

The `main` function returns an integer value:

```
return 0;
```

Just like someone eating your cake would give feedback on the taste:

- If the program runs well, it returns zero — the person will want to have some of your cake again
- Otherwise, the program might:
  - Return a non-zero value — The person won't want to eat your cake again

# The end of your programm

The `main` function returns an integer value:

```
return 0;
```

Just like someone eating your cake would give feedback on the taste:

- If the program runs well, it returns zero — the person will want to have some of your cake again
- Otherwise, the program might:
  - Return a non-zero value — The person won't want to eat your cake again
  - Crash

# The end of your programm

The `main` function returns an integer value:

```
return 0;
```

Just like someone eating your cake would give feedback on the taste:

- If the program runs well, it returns zero — the person will want to have some of your cake again
- Otherwise, the program might:
  - Return a non-zero value — The person won't want to eat your cake again
  - Crash — The person who hate your cake will spend a rough night

# Printing things in the terminal

`std::cout` denotes the terminal output.

# Printing things in the terminal

`std::cout` denotes the terminal output.

<< is an operation – just like $\div$ – it takes what's on its right and pushes it to what's on its left.

# Printing things in the terminal

`std::cout` denotes the terminal output.

`<<` is an operation – just like $\div$ – it takes what's on its right and pushes it to what's on its left.

```
std::cout << "Hello, World" << std::endl;
```

# Core Language vs Libraries

C++ contains key words accessible to all programmers

Libraries are pieces of code made accessible to programmers by other developers

To use libraries, you should "include" them in your program with the `#include<>` directive

# 02 Ingredients in your program: Variables

# Using variables

Another example

# Variables as the program memory unit

Variables are a mean of storing things during your program. They use up program memory.

# Variables as the program memory unit

Variables are a mean of storing things during your program. They use up program memory.
In the same way your photos and videos occupy space on your phone or laptop disk, program's variables occupy space on the processor when being run.

# Variable essentials 1/2

Variables have:

# Variable essentials 1/2

Variables have:

- A type: `int`, `float`, `std::string`, ...

# Variable essentials 1/2

Variables have:

- A type: `int`, `float`, `std::string`, ...
- A declaration

```
int a_variable; // declared (and default initialized)
```

# Variable essentials 1/2

Variables have:

- A type: `int`, `float`, `std::string`, ...
- A declaration

```
int a_variable; // declared (and default initialized)
```

- An initialization

```
int a_variable {3}; // declared and initialized to 3
int a_variable = 4; // declared and initialized to 4
```

# Variable essentials 2/2

Variables have:

- A value at a certain point in time

```
int a {3}; // a contains 3
a = 6; // a contains 6
```

# Variable essentials 2/2

Variables have:

- A value at a certain point in time

```
int a {3}; // a contains 3
a = 6; // a contains 6
```

- A scope

```
{
    int a {3};
    // a is accessible in this block
}
// a is not accessible here anymore
a = 5; // invalid statement
```

# Operations on variables 1/2

What does this code print ?

# Operations on variables 1/2

What does this code print ?

```
int a, b;
a = 6;
b = a + 1;
a = 9;
std::cout << b << std::endl;
```

# Operations on variables 1/2

What does this code print ?

```
int a, b;
a = 6;
b = a + 1;
a = 9;
std::cout << b << std::endl;
```

Solution

# Operations on variables 2/2

```
int a, b;
a = 6; // a has value 6
// we look at the value of a at this point in time
// b = 6 + 1 = 7
// b has value 7. It does not have value a + 1.
b = a + 1;
a = 9;
std::cout << b << std::endl; // prints 7
```

# Operations on variables 2/2

```
int a, b;
a = 6; // a has value 6
// we look at the value of a at this point in time
// b = 6 + 1 = 7
// b has value 7. It does not have value a + 1.
b = a + 1;
a = 9;
std::cout << b << std::endl; // prints 7
```

There is a difference between an expression and its value. `a+1` is an expression; it can be evaluated by taking the value of `a` at the moment of the computation.

# Constant variables

Variables can be set as *constant* with the word `const`. Once initialized, their value cannot be altered.

# Constant variables

Variables can be set as *constant* with the word `const`. Once initialized, their value cannot be altered.

```
const int volume = 15;
volume = 3; // won't compile; volume is const
```

# Practise time!

# 03 A special type : booleans

# Boolean values

They're the result of the **evaluation** of a **logical expression**:

```
int a = 6;
bool is_even = a % 2 == 0;
bool is_figure = a < 10;
```

They take one of two values: `true` or `false`.

# Boolean operations 1/2

You can *negate* a boolean – i.e. take its opposite value: `!variable`.

| $p$ | $!p$ |
|---|---|
| $true$ | $false$ |
| $false$ | $true$ |

# Boolean operations 1/2

You can *negate* a boolean – i.e. take its opposite value: `!variable`.

| $p$ | $!p$ |
|---|---|
| $true$ | $false$ |
| $false$ | $true$ |

You can express combination of booleans:

| $p$ | $q$ | $p\&\&q$ |
|---|---|---|
| $true$ | $true$ | $true$ |
| $true$ | $false$ | $false$ |
| $false$ | $true$ | $false$ |
| $false$ | $false$ | $false$ |

| $p$ | $q$ | $p\|\|q$ |
|---|---|---|
| $true$ | $true$ | $true$ |
| $true$ | $false$ | $true$ |
| $false$ | $true$ | $true$ |
| $false$ | $false$ | $false$ |

# Boolean operations 2/2

&& has the priority over || – just like $\times$ has priority over $+$.

# Boolean operations 2/2

`&&` has the priority over `||` – just like $\times$ has priority over $+$.

Thus, `(a && b) || (c && d)` and `a && b || c && d` are equivalent.

# Boolean operations 2/2

`&&` has the priority over `||` – just like $\times$ has priority over $+$.

Thus, `(a && b) || (c && d)` and `a && b || c && d` are equivalent.

However, `(a || b) && (c || d)` and `a || b && c || d` are different.

# Comparison operations

You can store the evaluation of a comparison in a boolean variable :

```
bool b1;
b1 = a > b;
b1 = a >= b;
b1 = a == b; // note the difference between = and ==
b1 = a != b; // a and b are different
b1 = a <= b;
b1 = a < b;
```

# Practise time

# 04 Cooking with variables

# if . . . else . . .

Yet another example

# Conditional statements

Conditional statements makes the code behave differently depending on the value of the given condition. They can be used to apply actions when variables have a specific value.

```
if(condition 1){
    // scope 1
} else if (condition 2) {
    // scope 2
}  else { // all other cases
    // scope 3
}
```

# Practise time!

# The many cases problem

What if we want to handle cases where a variable can take more than two values?

See this example

# Enumerations

`enum class` defines a type that can take a restrained amount of values:

```
enum class Instrument { Piano, Guitar, Drums };
Instrument a_piano = Instrument::Piano;
```

It's a good way to represent a family of related constants.

# Switch-case statements 1/2

`switch-case` statement is a way to write conditional statements on *integral types*[1] or enum types.

It is prefered when possible because it is more readable, and it executes in a *constant time*.

[1] Numerical values and booleans are integral types

# Switch-case statements 2/2

`break` signals the end of case-block: we can handle several cases in the same way by omitting it.

```
switch(my_instrument) {
    case Instrument::Piano:
        // ...
        break; // end of case Piano
    case Instrument::Guitar:
        // This code is applied to Guitar only
    case Instrument::Drums:
        // This code is applied to both Guitar and Drums
        break; // end of cases Guitar and Drums
}
```

# Practise time

# 05 Repeating actions

# While . . .

We still start with an example

# The `while` loop

`while` allows repeating a piece of code as long as a certain condition evalutes to `true`.

# The `while` loop

`while` allows repeating a piece of code as long as a certain condition evalutes to `true`.

```
while(/* condition */ ) {
    // condition is evaluated to true
    // we enter the loop
}
// condition is false, we exit the loop
```

# The `while` loop

`while` allows repeating a piece of code as long as a certain condition evalutes to `true`.

```
while(/* condition */ ) {
    // condition is evaluated to true
    // we enter the loop
}
// condition is false, we exit the loop
```

Notes:

- If the condition is not true when first encountering the loop, we never enter the loop;
- If the condition is never false, we never exit the loop: this is called *infinite* loop.

# On infinite loops

# On infinite loops

- They can be intended (a game-loop, . . . );

# On infinite loops

- They can be intended (a game-loop, . . . );
- They can be a sign of a bug (a condition that should become false but never does, . . . ).

# Known repetition

If you know in advance how many times you want to repeat an action, you can use `for` loops.
See this example

# The `for` loop 1/3

```
for(
  auto count = 0;      // the loop counter
  count < top_value;   // the condition to check at each loop start
  ++count              // the action at each end of loop
){
}
```

Note: `count`'s scope is the `for` loop's scope: it is not usable outside of it

# The `for` loop 2/3

```
for(auto count = 0; count < top_value; ++count){
}
```

# The `for` loop 2/3

```
for(auto count = 0; count < top_value; ++count){
}
```

- `auto` will determine the type of the variable if possible (usable anywhere)

# The `for` loop 2/3

```
for(auto count = 0; count < top_value; ++count){
}
```

- `auto` will determine the type of the variable if possible (usable anywhere)
- `++` is an operation that increments the variable by one

# The `for` loop 2/3

```
for(auto count = 0; count < top_value; ++count){
}
```

- `auto` will determine the type of the variable if possible (usable anywhere)
- `++` is an operation that increments the variable by one

We could have written:

```
for(int count = 0; count < top_value; count += 1){}
```

```
for(int count = 0; count < top_value; count = count + 1){}
```

# The `for` loop 3/3

```
for(auto count = 0; count < top_value; ++count){
}
```

is almost equivalent to:

# The `for` loop 3/3

```
for(auto count = 0; count < top_value; ++count){
}
```

is almost equivalent to:

```
auto count = 0;
while(count < top_value){
    ++count;
}
```

# The `for` loop 3/3

```
for(auto count = 0; count < top_value; ++count){
}
```

is almost equivalent to:

```
auto count = 0;
while(count < top_value){
    ++count;
}
```

*Note: why almost?*

# Practise time!

# See you next time!