Popular languages in our field:

- C: very fast, compiled, low-level (memory management)
- C++ : very fast, object-oriented, compiled
- Python: script, slower than above but easy to interface with C++
- Matlab: non-free, used for off-line data processing
- And many others : Java, Lisp, Labview, ...

My `Code` folder on this computer (does not include past works)

- 735 Matlab script + 1 Simulink file
- 450 Python scripts +1
- 1769 C++ headers + 1704 C++ sources
- Also 20+ Labview files for a specific industrial application
- And this was counted with a Python script

This course is on C++ with a focus on mathematics and algorithms

Advanced Robot Programming

Introduction to C++ programming

Olivier Kermorgant

General concepts underlying C++

- Compiled vs script: from raw code to binary program
- Upper-level compilation tools
- Architecture of a typical program

Programming is...

- Reading: vocabulary and syntax
- Thinking: objects and algorithms
- Writing: good practices for efficient writing
- Recycling: use of external libraries
- Confusing (sometimes): common errors and debugging tools

Robotics imply programming at several levels

- Hardware-related: interface with real robots, sensors, actuators...
- Software engineering: simulators, communications
- Maths: Vision, control algorithms, state estimation...
- Support software: user feedback, logging, analysing results...

Programming is needed... at least to obtain experimental results!

Language knowledge and programming good practices mean:

- Trying ideas faster
- Getting bad results as everybody but...
- Efficient debugging (limits and usual errors of a given language)

Eventually getting meaningfull experimental results, performing exhaustive simulations, comparing with other algorithms, etc.

Syntax examples

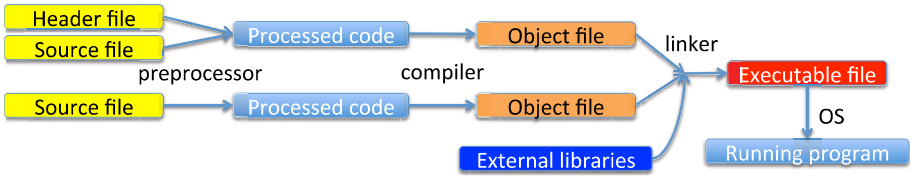
```cpp
#include<iostream>
int main(int argc, char ** argv)
{
    std::cout << "Hello World!" << std::endl;
}
```

```cpp
#include<iostream>
int main(int argc, char ** argv)
{
    std::cout << "Press a key to roll a dice, 'q' to quit"
              << std::endl;
    std::string key = "";
    while(key != "q")
    {
        std::cin >> key;
        int n = std::rand()%6+1;
        std::cout << "Rolled a " << n << std::endl;
    }
}
```

From raw code to a running program

Scripts (Python, Matlab...): online execution (see example)

Compiled languages: several steps between code and execution



- Different steps, different types of error...

Graph from John Marrero

What you should know about at the end

Compiling

- make and cmake
- Using a IDE
- Using several files in a single project

Basic syntax

- Built-in types and basic controls: if, then, for, while, switch
- Standard Template Library's useful types: string, vector
- How to define functions, structures and classes

More advanced syntax and tools

- STL's useful algorithms: find, sort, count
- lambda functions
- Generic programming: templates

Development tools

- Debugging and profiling code

The best ways to learn C++ (and other programming languages)

Find a personal project with increasing complexity

- Maybe one from your hobbies

Create a basic, turn-based text game

- Battleship, Rock-Paper-Scissors, 4-in-a-row, 21 sticks game...
- Then with an artificial intelligence (except for Rock-Paper-Scissors...)

Program an optimization algorithm

- Solving the Tower of Hanoi
- Path planning with A* (used in 15 puzzle game)
- Traveling salesman problem with genetic algorithm

Or just wait for the labs and group projects...

- But C++ will be used intensively in many courses

Architectures

C++ is used to create programs or libraries

Programs

- Have a `main()` function that is the entry point
- May be built from several files
- Use external libraries (never reinvent the wheel!)

Libraries

- Are almost the same... without the `main()` function
- Used to share common tools for several programs
- Very useful to have your own library for custom tools

How to compile a C++ program

Lowest-level: directly call the compiler

```
g++ hello.cpp -o hello
```

- Ok for a few files and no fancy dependencies

Use of Makefiles and `make` command

```
hello:  hello.cpp
```
+ `make`

- Allows more complex projects, handles dependencies

Higher-level: CMake that generates the Makefile

```
add_executable(hello hello.cpp)
```
+ `cmake` + `make`

- Multi-platform, looks for dependencies, allows cleaning, rebuilding...

What's in a C++ code?

6 different tokens can be found in a C++ code

Token type	Description	Examples
Keywords	Words with special meaning to the compiler	`int`, `double`, `for`, `if`, `class`
Identifiers	Words that are not into the C++ language	`std`, `x`, `MyFunction`
Literals	Constant values directly specified in the source code	0, "user", 3.14, "abc"
Operators	Math or logical operations	+, -, &&, %, >>
Punctuation	Defines the structure of the program	{} ( ) , ;
Whitespace	What is removed by the preprocessor	Spaces, tabs, comments...

Table from John Marrero

Numerous ways of editing and compiling code

Basic text editor and console

- Useful to know for remote compilation
- Notepad, vi(m), nano...

Smart editors

- Syntax highlighting, basic code completion
- May already have some compilation shortcuts
- Geany, Notepad++, Kate, SublimeText...

IDE (Integrated Development Editors)

- From assisted edition to compilation
- Notion of project: bunch of files with ad-hoc compilation
- Support custom compilation, going from one file to another...
- Eclipse, QtCreator, Visual Studio...

Structures: bunch of types

Used to create a group of several variables

```
struct MyRobot
{
    double x;
    double y;
    double theta
    std::string name;
};

MyRobot robot;
robot.x = 3;
robot.y = 4;
robot.theta = M_PI/2;
robot.name = "My mobile robot";
```

Another classical example

```
#include<stdio.h>
#include<iostream>
// A comment
int main(int argc, char ** argv)
{
    std::cout << "Hello World!";
    if(argc > 0)
    {
        std::cout << " Here are your arguments:"<< std::endl;
        for(unsigned int i=0;i<argc;i++)
        {
            std::cout << "- arg #" << i+1 << ": "
                << argv[i] << std::endl;
        }
    }
    else
        std::cout << std::endl;
    return 0;
}
```

What do we find here?

The Standard Template Library

Called STL and seen in code with `std::`

Not built into the language but almost always shipped with installation

To be used instead of old C-types

- Strings: `std::string`
- Containers: `std::vector`
- Print to screen: `std::cout`, `std::endl`
- Building a custom string: `std::stringstream`

Many existing algorithms in the STL

Data types

Variables need to be declared to be used: `int x;`

- If not assigned (no given value) then default value
- Possible to assign at declaration: `int x = 4;`

Every variable has a type

- Numbers: `int` (`unsigned`), `double` (`float`), `bool`
- Strings: `char` but also `std::string`
- Possible to create your own types, and more complex objects

Compiler checks for coherence

- Operations like `x+y`, `"abc"+ "def"` or `x/y`

Variables can be put into containers

- Built-in: arrays `int x[4];`
- More useful: vectors (also a type) `std::vector<int> x(4);`

Old style configuration or data files

Raw text

- Not standard, no structure

```
expnb 0
start_pos 0 1 2
end_pos 0 1 3.5
```

CSV (comma separated values)

- More useful for data logging

```
x ; y; z
0 ; 1 ; 2
0 ; 1 ; 3.5
```

Modern files: markup languages

XML (EXtensible Markup Language)

- Widely used, many parsers available

```
<point name="start">
  <x>0</x>
  <y>0</y>
  <z>0</z>
</point>
```

YAML (Yaml ain't a Markup Language)

- Easier to read and hand-write, many parsers too

```
start:
  x: 0
  y: 1
  z: 2
end:
  x: 0
  y: 1
  z: 2.5
```

Basic Input/Output

Because not everything may be hard-coded

```
int main() {
    int x = std::rand() % 100 +1, n=0,count=0;
    while(n != x)
    {
        count++;
        std::cout << "Give a number: ";
        std::cin >> n;
        if(n < x)
            std::cout << "Too small!" << std::endl;
        else if(n > x)
            std::cout << "Too large!" << std::endl;
    }
    std::cout << "Found in " << count << " tries!" << std::endl;
    std::ofstream out("tries.txt", std::ios_base::app);
    out << x << ": " << count << std::endl;
    out.close();}
```

Exercise: program where the computer has to guess

Same program, different parameters

Hard-coded parameters

- Usually non optimal
- Leads to recompiling for each new run

Command-line arguments

- Ok for a few parameters
- Some libraries help automatic parsing

```
myprogram -f file -r -h
```

Best practice: use configuration files

- Load file at startup
- Read it and initialize parameter variables
- Many formats availables for configuration files

Classical for-loops

```
// all combinations of 2 ints from 0 to 100
unsigned int i;
for(i=0; i < 99; i++)
{
    for(unsigned int j=i+1; j < 100; j++)
        std::cout << i << " and " << j << std::endl;
}
```

1. Initialization (declaration or assignment) (`i=0;`)
2. For-loop condition (`i < 99;`)
3. What's executed at the end of each loop (`i++`)

Control, logical and blocks

Blocks are the backbone of a program

- `if` block: do different things depending on a condition
- `for` loops: the loop is executed while changing a given variable
- `while` loops: will loop as long as a given condition is true
- `switch-case` blocks: jump to a given block depending on a variable

The `bool` type and its combinations are essential to define conditions

- Comparing two variables: == (not "="), >, <=, !=...
- Boolean algebra
  - A and B: `A && B`
  - A or B: `A || B`
  - not A: `!A`

For-loops are often used with containers

```
std::vector<int> vec(6);
for(int i=0;i<vec.size();++i)    // explicit indexing
    vec[i] = i*i;                // vec = [0,1,4,9,16,25]

for(auto &x : vec)               // will loop on x as vec's element
    x = 4;                       // all components of vec set to 4

for(auto x : vec)                // here x is a copy of the element
    x = 5;                       // x is changed but not vec

for(auto const &x : vec)         // here x is forced to be constant
{
    std::cout << x << std::endl;// ok
    //x = 5;                     // does not compile
}
```

- `auto` initialization depending on vector type
- Use of `&` and/or `const` shows what is expected from `x`
- Explicit indexing may still be useful
  - when using several indices at the same time `v[i+1]-v[i]`
  - when using several vectors at the same time `u[i]+v[i]`

Useful robot while loop

```
// conditions when I want to stop
const double error_min = 0.01;
const unsigned int iter_max = 1000;

// initialize loop variables
bool external_cancel = false;
double error = 2*error_min;
unsigned int iter = 0;

while((iter < iter_max) && (error > error_min) && !external_cancel)
{
    // update iteration count
    iter++;

    // update error
    error = ...

    // check for external cancel

}
std::cout << "End of loop!"<< std::endl;
```

Typical use of controls: algorithms

Algorithms are like cooking: several small steps to get a desired result

Typical algorithms

- Getting the min value of a list or sorting a list
- Finding the LCD of two `int`'s
- Solving a contrained optimization problem
- Detecting an object in an image

Many basic tools in the STL (Standard Template Library)

- `sort`, `find`, `count` ...
- Based on the use of containers
- And # `include` < `algorithm` >

Many specialized tools in other libraries (OpenCV, ViSP, ROS...)

Example of switch-case block

```
// get the name of the polygon
unsigned int sides = ...
std::string msg = "";
switch(sides)
{
    case 0:
    case 1:
    case 2:
        msg = "This is not a polygon";
        break;
    case 3:
        msg = "This is a triangle";
        break;
    case 4:
        msg = "This is a square";
        break;
    default:
        msg = "This is too much complicated";
}

std::cout << msg << std::endl;
```

Functions

A way to rearrange the code: functions

- Improves readability, maintenability, code reuse

Must be defined with their input / output datatypes

- No output: `void` type

```
int Square(int x)
{
    return x*x;
}
double Square2(double x)
{
    return x*x;
}
```

Function signature: types of its arguments

Overloading: same name, different signature

Possible to overload even operators! +, -, <...

Control blocks and scope

Part of the code where a given variable is defined (and usable)

Its definition block + following included blocks

```
#include <iostream>
using std::cout;using std::endl;
int main()
{
    int i = 2, k = 3;
    cout << i << endl;      // prints 2
    if(i == 2)
    {
        int j = 3;
        int i = 4;
        cout << i << endl; // prints 4, original is shadowed
        cout << j << endl; // prints 3
        k = 5;
    }
    cout << i << endl;      // prints 2
    cout << j << endl;      // j does not exist, does not compile
    cout << k << endl;      // prints 5
}
```

Possible to shadow existing variable in a block: very, very, bad practice

Combining structure (bunch of variables) and functions

```
class MyRobot
{
private:
    double x, y, theta;
    std::string name;
public:
    // constructor function
    MyRobot(const std::string &_name) {name = _name;}
    // Motion function
    void Move(double _dx, double _dy, double _dtheta)
    {
        x += dx;
        y += dy;
        theta += dtheta;
    }
};
```

Inner variables: attributes (`MyRobot` has `x, y, theta`)

Inner functions: methods (`MyRobot` can do `Move`)

Classes can then be used as a type

```
class MyRobot
{...};

void main()
{
    // initialize at (x=0,y=0,theta=0)
    MyRobot robot("Hector");

    // updates x, y and theta
    robot.Move(1,2, 0.4);

    // won't work
    std::cout << "x-position: "<< robot.x << std::endl;
}
```

Last statement will not work because `x` is private

- Attributes can be all public for small projects
- They will be protected or private in most libraries
- Classical use of *setters* and *getters*

Several ways to write the signature

- Pass-by-value: modifications will stay inside

  `Function(int x)`

- Pass-by-reference: modifications will be valid outside

  `Function(int &x)`

- Pass-by-const reference: modifications are not possible

  `Function(const int &x)`


```
int AddVectors(const unsigned int &_size,
                        const std::vector<double> &_v1,
                        const std::vector<double> &_v2,
                        std::vector<double> &_v)
{
    if(_v1.size() != _size)
        return 1;
    if(_v2.size() != _size)
        return 2;

    _v.resize(_size);
    for(unsigned int i = 0;i<_size;++i)
        _v[i] = _v1[i] + _v2[i];
   return 0;
}
```

Passing by reference allows several return values

Usually, returning 0 = no error

Inheritance

Inheritance: a classe can inherit from another one

```
class Vector : Matrix
{...};
```

Abstract class: only to define sub-classes (daughters)

```
class Robot
{
    virtual void MoveEndEffector() = 0;
};

class Baxter : Robot
{
    void MoveEndEffector() {...}
};
```

Setters and getters

Keep a control on what's happening when defining attributes

```
class MyRobot
{
public:
    void Get_x(double &_x) {_x = x;}
    void Get_y(double &_y) {_y = y;}
    void Set_theta(double &_theta)
    {
        if(_theta < -M_PI)
            theta = _theta + 2*M_PI;
        else if(_theta > M_PI)
             theta = _theta - 2*M_PI;
         else
             theta = _theta;
    }
};
```

Methods, attributes, inheritance... a summary

Concept	Verb	Example
Attribute	has	`Robot` *has* joint values
Method	can do	`Matrix` *can do* vector multiplication
Inheritance	is a	`Vector` *is a* kind of `Matrix`

Daughter classes can use public or protected attributes / methods

They can also use overloading to re-define methods

Structures can also have methods...

```
class MyClass{
    double y;
    public:
        double x;
};
struct MyStruct {
    double y;
    private:
        double x;
};
int main() {
    MyClass mc;
    mc.x = 1;     // ok
    mc.y = 1;     // not ok
    MyStruct ms;
    ms.x = 1;     // not ok
    ms.y = 1;     // ok
}
```

Only difference is the default behavior:

- `public` for structures
- `private` for classes

Classes, structures and namespaces

In practice, classical use of classes / structures / namespaces:

What's in	Use	Example
Only attributes	`struct`	"Plain Old Data", grouping variables: Point(x,y)
Only static functions	`namespace`	Group of similar independant functions: ReadFile, WriteFile, etc.
Mix of attributes and methods	`class`	Any other case: Robot(x,y,Move, etc..)

Questions to ask first:

- May two of them exist in the same program? ⇒ class / struct
- Can it actually do things? ⇒ class
- I am just regrouping variables for readability? ⇒ namespace

Classes and structures may also be part of namespaces...

Templates

Writing the same function for different types or classes?

```
int min(int a, int b) {if(a<b) return a; return b;}
```

- Copy-paste using function overloading

```
int min(int a, int b)
   {if(a<b) return a; return b;}

double min(double a, double b)
   {if(a<b) return a; return b;}

unsigned int min(unsigned int a, unsigned int b)
   {if(a<b) return a; return b;}
```

- Not easy to maintain
- Will not work if given type is not prepared
- Bad idea in general

Overloading example with Matrix and Vectors

```
class Matrix
{
    Matrix operator*(const &Matrix _M);
    ColVector operator*(const &ColVector _v);
};

class ColVector : Matrix
{
    Matrix operator*(const &RowVector _v);
};

class RowVector : Matrix
{
    double operator*(const &ColVector _v);
};
```

What happens when doing:

- `ColVector * RowVector`
- `RowVector * ColVector`
- `((Matrix) RowVector) * ColVector`

Static methods and namespaces

A class can have `static` methods and attributes

- Common to all instances
- Static functions may only read/write static attributes
- Difference with namespaces?

```
class MyClass
{
    static double x;
    static void PrintX();
};

namespace MyNamespace
{
    double x;
    void PrintX();
}

int main()
{
    MyClass::x = 1;
    MyClass::PrintX();
    MyNamespace::x = 2;
    MyNamespace::PrintX();
}
```

New from C++11, used to create on-the-fly functions

```
int main
{
auto func = [](int a, int b){return a+b;};
cout << func(1,2) << endl; // prints 3
}
```

General syntax

```
[](int a, int b){return a+b;}
```

- `[]` : this is a lambda function
- `(int a, int b)` : the function arguments (signature)
- `{return a+b;}` : what the function does

Very useful in algorithms and some other cases

Templates are about writing the same code for different types

```
template <class T> T min(T a, T b)
   { if(a < b) return a; return b}
```

- Then the magic happens (or not):

```
min(2,3);           // will assume T = int
min(2.4,2.3);       // will assume T = double
min(2,1.2);         // does not compile, T cannot be guessed
min<int>(2,1.2);    // impose T = int, will return 1
min<float>(2,1.2); // impose T = float, will return 1.2
```

- Very easy to maintain
- Actually performs code generation at compilation

Concept of duck-typing

- Do not care about the actual type as long as it can do what we ask

We often want to sort things or find an element in a `vector`

```
vector<int> v(10);
for(auto &i: v)
  i = rand();
sort(v.begin(),v.end());
```

What if the elements cannot be compared?

```
class MyObj
{
  int value;
  double other_value;
  public:
    MyObj() {value = rand();}
    int value() {return value;}
};
vector<MyObj> v(10);
sort(v.begin(),v.end()); // fails
```


Duck-typing in practice

```
struct Duck {  void quacks() {}   };
struct Dog  {  void barks() {}   };
// a seagull that knows how to quack
struct Seagull {  void quacks() {}   };

template <class T> T CompileIfDuck(T animal)   { animal.quacks();}

int main() {
  Duck duck;  Dog dog; Seagull seagull;
  CompileIfDuck(duck);          // compiles
  CompileIfDuck(dog);           // does not compile
  CompileIfDuck(seagull);       // compiles
}
```

Compilation error at code generation:

```
In instantiation of T CompileIfDuck(T) [with T = Dog]:
  required from here
error: struct Dog has no member named quacks
   { animal.quacks();}
```

Organizing code in several files

Functions, classes... all in one file?

- Very big files if everything in the same
- Makes it difficult to re-use code

Write some parts in other files

- Headers: declaration of classes and functions
- Sources: Definition of methods and functions
- Main file: includes headers

Libraries: pre-compile stable and common parts

- Compile headers and sources to library
- Main program *includes* headers and is *linked* to library

Example for typical class

Headers only declare classes and functions, `matrix.h` :

```
class Matrix
{
public:
    Matrix(const int &_rows, const int &_cols);
    void Set(const int &_i, const int &_j, const double &_x);
protected:
    int rows_, cols_;
};
```

Other sources define what need to be, `matrix.cpp` :

```
#include <matrix.h>    // declares the Matrix class
Matrix::Matrix(const int &_rows, const int &_cols)
{
    ...
}

void Matrix::Set(const int &_i, const int &_j, const double &_x)
{
    ...
}
```

Class, algorithms and lambdas

Define a function to compare

```
bool compare(MyObj &o1,MyObj &o2)
    {return o1.value() < o2.value();}

sort(v.begin(),v.end(),compare); // will use the compare function
```

Or overload the < operator

```
class MyObj
{
friend bool operator<(MyObj &o1,MyObj &o2)
    {return o1.value() < o2.value();}
};
sort(v.begin(),v.end()); // will use the operator<
```

Or use a lambda (on-the-fly function)

```
// will use the given lambda function
sort(v.begin(),v.end(),
     [](auto &o1,auto &o2){return o1.value() < o2.value();});
```

Other useful STL <algorithm> functions

`find(v.begin(),v.end(),value)`

- returns first element that matches `value`

`find_if(v.begin(),v.end(),boolean_function)`

- returns first element where `boolean_function == true`

`count(v.begin(),v.end(),value)`

- counts number of elements equal to `value`

`count_if(v.begin(),v.end(),boolean_function)`

- counts number of elements where `boolean_function == true`

`for_each(v.begin(),v.end(),do_something_function)`

- calls `do_something_function` on each element of v

Debugger

When everything compiles...

- Still some runtime error
- Typically: segmentation fault when accessing outside of a vector/list

Naive approach: find the crash by hand

- Put `std::cout` 's everywhere
- Try to understand what happens between the last printed message and the next one

Improved naive approach: use a global variable to set debug level

- Allows enabling/disabling mentionned `std::cout` 's
- Just in case we need them later

Corresponding code and compilation

Main code, `math_program.cpp` :

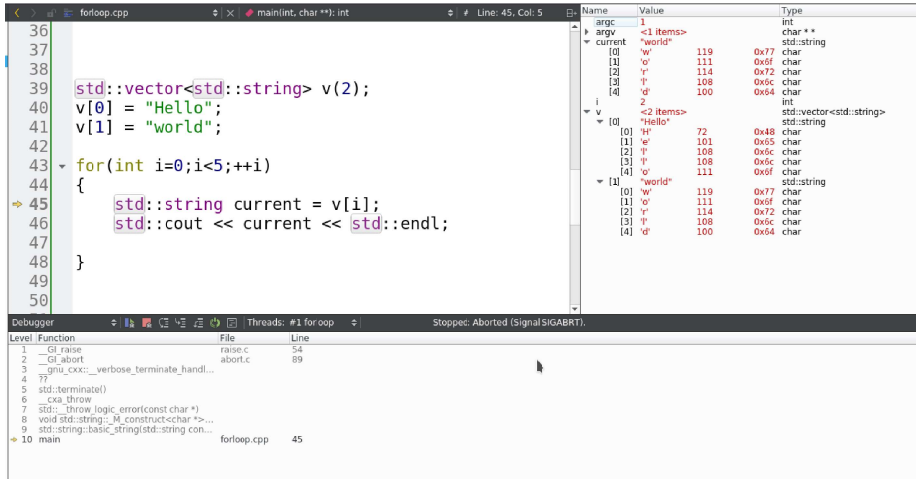
```
#include <matrix.h>    // declares the Matrix class

int main()
{
    Matrix matrix(4,5);
    ...
}
```

Compile everything together, `CMakeLists.txt` :

```
add_executable(math_program math_program.cpp
               matrix.h matrix.cpp)
```

Actal debugging: use a tool

Use of gdb (GNU Debugger)
CMake: `cmake -DCMAKE_BUILD_TYPE=Debug`
Easy to use with IDE's



Developing in practice

Start by a general design of:

- What the program should do
- In which order
- What are the inputs and outputs

Write base functions and file structure

- Helps to visualize the whole thing
- Copy/paste former code that was already doing the job

Try to compile and run often

- It will not compile, or it will not run, or it will do strange things
- Read the console message that try to help you
  - When asked for help during labs, we actually read these messages
  - It works.

IDE's are also a great help to focus on the code

Classical programming is sequential
Using threads is a way to run several function at the same time

- The functions should not rely from the others

```
#include <thread>
// a long duration function that compute the "out" argument
void MyLongFunction(const double &in, double &out)
{...}

int main() {
double out1, out2;

// launches the function with arguments (2, out1)
std::thread t1(MyLongFunction, 2, out1);
// launches the function with arguments (2.6, out2)
std::thread t2(MyLongFunction, 2.6, out2);

t1.join();      // waits for the end of first thread
t2.join();      // same for 2nd thread

// here the two functions have returned
// out1 and out2 can be used
}
```


A priori optimization

- A bad idea if it goes against readability
- Avoid too many variable creation or copy
- Use ad-hoc argument passing (again, no copy)
- Use existing, recognized libraries that do the job
- Especially for math algorithms (vector sort/find, matrix manipulation...)

Optimizing existing code?

- Re-read the code to correct a priori badly written parts
- Use of profiling tools

C++ for robotics

- Language syntax
- Algorithmics
- Knowledge of existing libraries
- Good practice for efficient debug and optimization

Syntax and algorithms:

- `coderbyte.com`
- `www.cppforschool.com` (easier and with solutions)

Personal projects are great (online help from forums)

- Start on paper: structure of the program, definition of algorithms and classes
- Better use configuration files from the beginning
- Build elementary code that works and improve it
- When slow, profile to optimize

Show percentage of time passed in each function

Example with valgrind: `valgrind -tool=callgrind ./prof`

Then use a program (Kcachegrind) to visualize

- Call map (usually hardly readable)
- Call graph