

# Introduction au langage C++

## Option MAREVA 2016

Hassan Bouchiba<sup>1</sup>, Valérie Roy<sup>1</sup>

<sup>1</sup>[prenom.nom@mines-paristech.fr](mailto:prenom.nom@mines-paristech.fr), Mines ParisTech

11/02/2016



# Plan

- ➊ Introduction générale
- ➋ Un premier programme
- ➌ Gestion de la mémoire en C++
- ➍ La POO en C++
- ➎ Outils de développement et bonnes pratiques



## Le C++ en 3 questions

### 1 Introduction générale

Le C++ en 3 questions

Les librairies en C++

### 2 Un premier programme

### 3 Gestion de la mémoire en C++

### 4 La POO en C++

### 5 Outils de développement et bonnes pratiques

# Le C++ qu'est ce que c'est ? (1)

Le C++ c'est un langage de programmation...

- Crée par Bjarne Stroustrup en 1983.
- Compilé.
- Fortement typé.
- Orienté objet.
- Haut niveau.



## Le C++ qu'est ce que c'est ? (2)

Le C++ c'est un langage de programmation défini par un standard international.

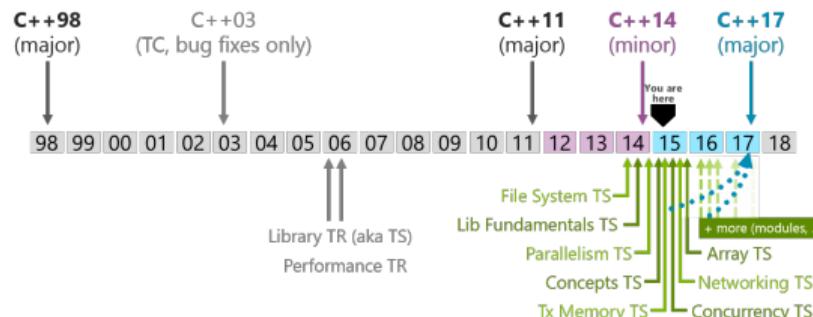


Figure: Évolutions du langage C++ depuis 1998. (TS : Technical Specification)

Le C++ c'est un langage de programmation dérivé du langage C : la librairie standard du C est incluse dedans.

# Le C++ à quoi ça sert ?

On peut utiliser le C++ lorsque l'on désire :

- Écrire des programmes de moyenne ou grosse taille.
- Avoir un contrôle sur ce que fait votre programme.

Il vaut mieux éviter lorsque :

- On veut prototyper des algorithmes simples.
- On a des besoins de programmation spécifiques.

# Comment je fais pour coder en C++ ?

## IDE



Figure: Environnement de développement



## Les librairies en C++

### 1 Introduction générale

Le C++ en 3 questions

Les librairies en C++

### 2 Un premier programme

### 3 Gestion de la mémoire en C++

### 4 La POO en C++

### 5 Outils de développement et bonnes pratiques

## Les librairies

*"You shouldn't reinvent the wheel unless you plan on learning more about wheels : use libraries"*





# STL : la librairie standard

Exemples d'éléments de la librairie standard du C++ (liste non exhaustive) :

- **Conteneurs** : <vector>, <list>, <stack>, <set> ...
- **Chaines de caractères** : <string>
- **Entrées/sorties** : <iostream>, <fstream> ...
- **Fonctionnalités du langage** : <exceptions>, <limites> ...
- **Threads** : <thread>, <mutex> ...

## Comment utiliser un élément de la librairie standard ?

Pour que l'élément soit accessible, il faut l'inclure via la commande **#include**.

Exemple :

```
#include <string>    // librairie contenant les chaines de caractere
#include <iostream>   // librairie pour les entrees/sorties
```

Les éléments de la librairie standard peuvent alors être utilisés suivis du préfixe **std ::**.  
C'est un espace de nom (ou namespace anglais).

Exemple :

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

Dans certains tutoriels vous verrez la commande suivante qui vous permettra de vous passer de ce préfixe.

```
using namespace std; // ne jamais faire ca !!!!!!!!
```

## Les librairies en C++

## Les autres librairies en C++

Exemple de librairies C++ :

- **Generic** : Boost
- **GUI** : GTK+, Qt, VTK
- **Multimedia** : SDL
- **3D** : OpenGL, Direct3D
- **Maths** : Eigen, glm
- **Image processing** : OpenCV



# Déroulement du cours

**Jeudi 11 février :**

- Présentation générale du C++ et des outils de développement
- TPs : applications console

**Vendredi 12 février :**

- Application : réalisation d'une application 3D avec OpenGL



## Syntaxe de base en C++

### 1 Introduction générale

### 2 Un premier programme

Syntaxe de base en C++

TP

Organisation d'un programme

### 3 Gestion de la mémoire en C++

### 4 La POO en C++

### 5 Outils de développement et bonnes pratiques

## Les types de base

Types natifs du C++ :

- **booléens** : bool
- **caractères** :: char
- **entiers** : int, unsigned int, long int, short int.
- **les flottants** : float, double.

```
// déclaration sans initialisation : la valeur initiale de la variable peut
// être n'importe quoi, à éviter.
int ma_variable;

// déclaration avec initialisation : différentes syntaxes.
int ma_variable = 42;
int ma_variable(42);
int ma_variable{10};
```

## Attention au choix de type

	<b>char</b>	<b>short int</b>	<b>int</b>	<b>long int</b>
<b>taille</b>	1	2	4	8
<b>min</b>	-128	-32768	$-2.110^9$	$-9.210^{18}$
<b>max</b>	127	32767	$2.110^9$	$9.210^{18}$

**Table:** Tailles en octets, valeurs minimales et maximales pour certains types entiers



**Figure:** Explosion du vol inaugural d'Ariane 5, 4 juin 1996

# Les fonctions

## Principe

En C++ on distingue la *déclaration* de la *définition*.

**Déclaration** : prototype de la fonction (spécification du type des variables d'entrée et du type de retour)

Exemple : la fonction foo prend en entrée un flottant de type double et retourne un entier.

```
int foo(double valeur);
```

**Définition** : implémentation de la fonction.

```
int foo(double valeur) {  
    return static_cast<int>(valeur);  
}
```

## Syntaxe de base en C++

# Les instructions de base (1)

Boucle "for" :

```
for (int i(0); i<num_iterations; ++i) {
    // on passera num_iterations fois dans cette boucle
}
```

Bloc "if else" :

```
if /*expression logique*/ {
    // a executer si l'expression logique est a true
} else if /*autre expression logique*/ {
    // a executer sinon et si l'autre expression logique a true
} else {
    // a executer sinon
}
```

## Syntaxe de base en C++

# Les instructions de base (2)

Boucle "while" :

```
while(/*expression_logique*/) {  
    // on passera dans cette boucle tant que l'expression logique sera vraie  
}
```

Bloc "switch case" :

```
switch (i) { // i doit etre de type entier  
    case val_1: {  
        // instruction a executer lorsque i=val_1  
        break;  
    }  
    case val_2: {  
        // instruction a executer lorsque i=val_2  
        break;  
    }  
    default: {  
        // instruction a executer par defaut  
    }  
}
```

## 1 Introduction générale

## 2 Un premier programme

## Syntaxe de base en C++

TP

## Organisation d'un programme

3 Gestion de la mémoire en C++

## 4 La POO en C++

## 5 Outils de développement et bonnes pratiques

## Énoncé

Créer un dossier pour votre projet :

```
$$ mkdir
```

Créer un fichier hello\_world.cc avec l'éditeur gedit :

```
$$ gedit hello_world.cc
```

Inclure iostream de la librairie standard :

```
#include <iostream>
```

Créer une fonction main() :

```
int main() /*votre code ici*/
```

Afficher la chaîne de caractères "Hello World!" :

```
std::cout << "Hello World!" << std::endl
```

Compiler votre fichier source :

```
$$ g++ hello_world.cc
```

Exécuter votre programme :

```
$$ ./a.out
```

Tester les différentes options de compilation :

```
$$ g++ hello_world.cc -c  
$$ g++ hello_world.cc -o mon_super_programme
```

# Solution

```
#include <iostream>
#include <string>

int main() {
    // On affiche "Hello World!" dans la sortie standard (la console).
    std::cout << "Hello World!" << std::endl;

    // On lit une chaîne de caractères sur l'entrée standard et on l'affiche
    // ensuite dans la sortie standard.
    std::string input_str("");

    std::cin >> input_str;
    std::cout << input_str << std::endl;

    return 0;
}
```

## Organisation d'un programme

### 1 Introduction générale

### 2 Un premier programme

Syntaxe de base en C++

TP

Organisation d'un programme

### 3 Gestion de la mémoire en C++

### 4 La POO en C++

### 5 Outils de développement et bonnes pratiques



## Organisation d'un programme

# Qu'est ce qu'un programme ?

Tout programme est composé des parties suivantes :

- Parties séparées **logiquement** (fonctions, classes, namespaces)
- Parties séparées **physiquement** (librairies, fichiers source)

L'idée principale est la **modularité** : garder les choses séparées et y accéder via des interfaces publiques.



## Organisation d'un programme

# Organisation du code source (1)

### Fichiers .h

#### Fichier de définition

### Fichiers .cc

#### Fichier d'implémentation

- Interface publique du code
- Contient les différentes définitions (Classes, fonctions ...)
- Inséré dans le code via `#include`
- Autres suffixes possibles (.hpp)

- Contient l'implémentation du code
- C'est ce fichier qui sera compilé et linké pour générer un exécutable (ou une librairie)
- Autres suffixes possibles (.cxx, .cpp)



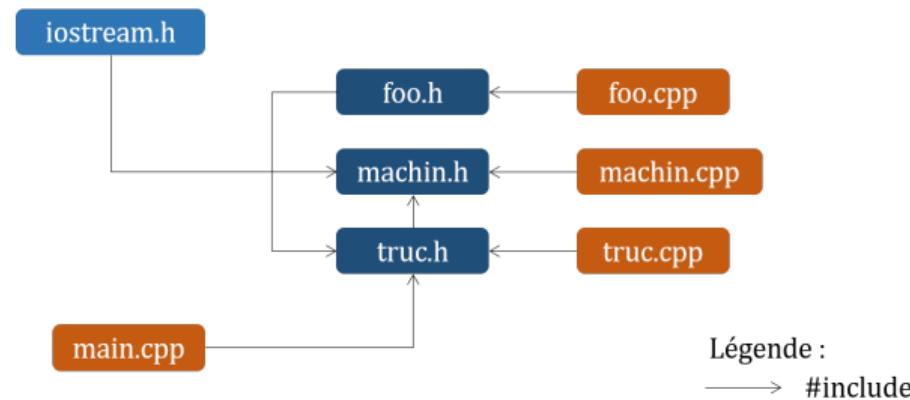
## Organisation du code source (2)

### Avantages de la séparation du code source

- Vitesse de (re)compilation du code
- Lisibilité du code
- Flexibilité (modularité)

## La directive `#include` (1)

- Toutes les commandes commençant par `#` sont des directives de préprocesseur (ex : `#define`, `#include`, `#ifndef`)
- Règle de base : On inclue toujours les `.h` jamais les `.cpp`



## Organisation d'un programme

# La directive #include (2)

Il existe 2 manières d'inclure un .h :

```
// entre crochets quand il s'agit d'une librairie externe
#include <iostream>
#include <Eigen/Dense>
// entre guillemets lorsqu'il s'agit d'un fichier de votre projet
#include "mon_projet/ma_classe.h"
```

## Organisation d'un programme

### La directive #include (3)

Comment gérer les inclusions multiples ?

Exemple : fichier titi/toto.h

```
#ifndef MON_PROJET_TITI_TOTO_H_
#define MON_PROJET_TITI_TOTO_H_ // le compilateur ne passera ici qu'une seule fois.

/* votre code ici */

#endif // MON_PROJET_TITI_TOTO_H_
```

## La fonction main

C'est la fonction de départ, c'est elle que le compilateur va chercher pour commencer votre programme : c'est le **point d'entrée** de votre programme.

- Elle est unique
- Elle ne peut pas être surchargée
- Elle en peut pas être appelée au sein du programme

On peut la définir de deux manières :

```
int main() {  
    /* votre code ici */  
    return 0;  
}
```

```
int main(int argc, char* argv[]) {  
    /* votre code ici */  
    return 0;  
}
```



## Passer des arguments à votre programme

```
int main(int argc, char* argv[]) {
    /* votre code ici */
    return 0;
}
```

- *argc* est la longueur du tableau *argv*.
- *argv[0]* est le nom du programme.
- *argv[i](i > 0)* sont les éventuels arguments passés au programme.

## Organisation d'un programme

# TP

Écrire un programme qui affiche dans la console son nom, ainsi que la liste des arguments qu'on lui a passé en entrée.

Organisation d'un programme

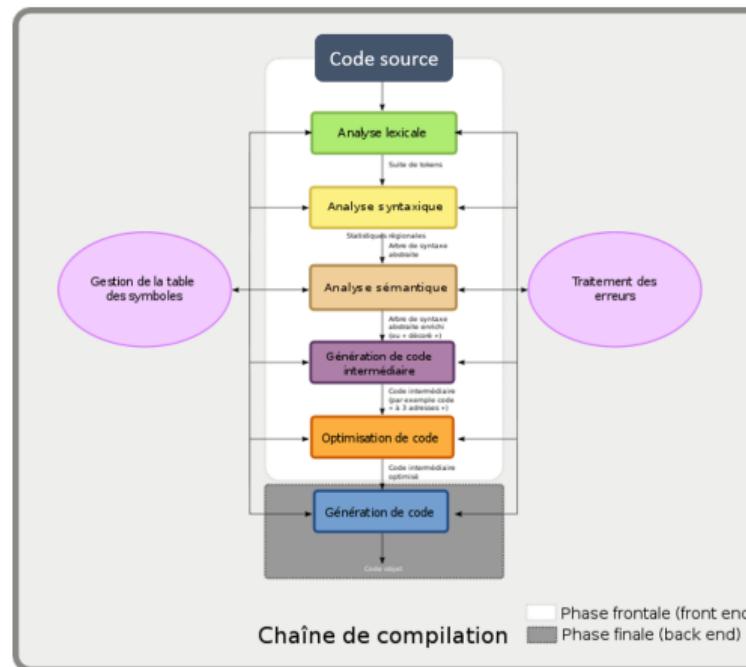
# Correction

```
#include <iostream>

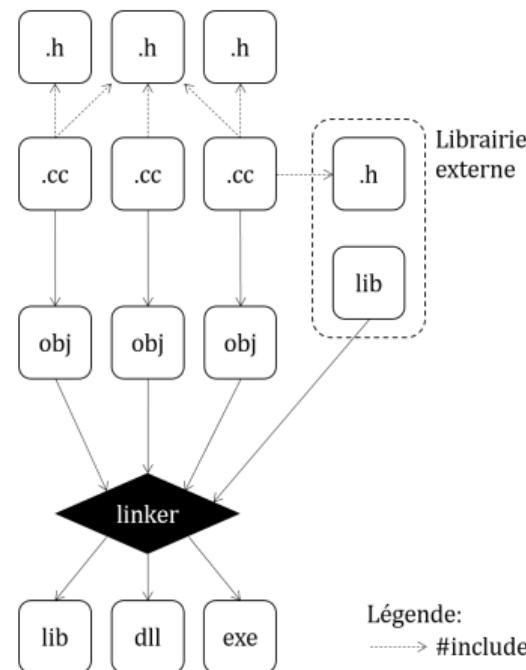
int main(int argc, char* argv[]) {
    std::cout << "nom du programme : " << argv[0] << std::endl;
    std::cout << "nombre d'arguments : " << argc-1 << std::endl;
    std::cout << "liste des arguments :";
    for (int i(1); i<argc; ++i){
        std::cout << " " << argv[i] << std::endl;
    }

    return 0;
}
```

# Compilation (1)



# Compilation (2)



## Les pointeurs

### 1 Introduction générale

### 2 Un premier programme

### 3 Gestion de la mémoire en C++

Les pointeurs

Les zones mémoire

Les références

### 4 La POO en C++

### 5 Outils de développement et bonnes pratiques

# Adresse mémoire

## Définition

Durant l'exécution d'un programme chaque objet du programme (une variable, une fonction, un tableau) est situé quelque part dans une zone mémoire l'endroit où est situé un objet en mémoire s'appelle **une adresse**.

En C++, on peut obtenir l'adresse d'un objet.





## Les pointeurs

# Qu'est ce qu'un pointeur ?

- Un pointeur est une variable contenant l'adresse d'un objet
- Un pointeur doit être initialisé à `nullptr`
- Pour un type T, le pointeur associé est de type T\*. Par exemple : un pointeur vers un int sera un int\*

Dans une expression :

- & est l'opérateur "adresse de..."
- \* est l'opérateur "est pointé par..."

## Pourquoi un pointeur doit être initialisé à nullptr ? (1)

```
#include <iostream>
#include <string>

int main() {
    float* un_pointeur;
    std::cout << *un_pointeur << std::endl;

    return 0;
}
```

Ici le pointeur pf n'est pas initialisé, sa valeur peut être n'importe quoi.

## Les zones mémoire

## 1 Introduction générale

## 2 Un premier programme

## 3 Gestion de la mémoire en C++

Les pointeurs

Les zones mémoire

Les références

## 4 La POO en C++

## 5 Outils de développement et bonnes pratiques

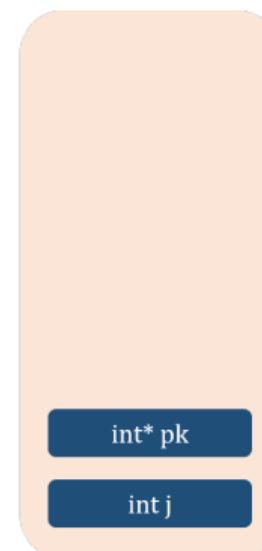
## Les zones mémoire

## Les deux zones mémoire

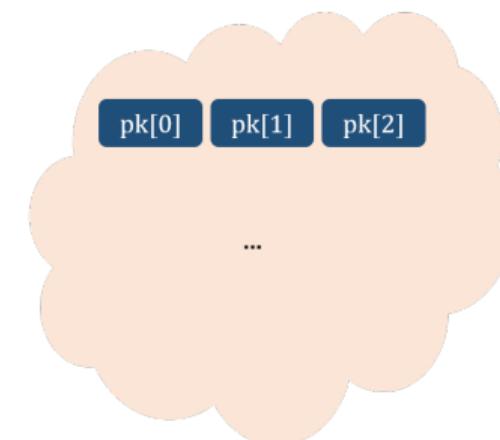
```
int main() {
    int j(42);
// entier alloue statiquement
    int* j = new int(42)
// entier alloue dynamiquement

    delete j;

    return 0;
}
```



## pile d'exécution (stack)



tas (heap)



## Les zones mémoire

# Durée de stockage

## Définition

Durée de stockage : propriété d'un objet qui définit la durée de vie de l'emplacement mémoire qui contient l'objet.

La durée de stockage est déterminée par le type d'allocation choisie pour créer l'objet.  
Elle peut être :

- **automatique** : pile d'exécution (allocation statique) **Vous n'avez pas à vous en soucier**
- **dynamique** : le tas (allocation dynamique) **C'est à vous de la gérer !**

## Les zones mémoire

# La pile d'exécution (stack)

- **Création d'un objet** : à la déclaration.
- **Destruction** : à la fin du bloc englobant {}.

```
// programme                                // etat de la pile d'execution
int main() {                                //
    int i(2);                                // [i, 2]
    {                                         // [i, 2]
        int j=i;                            // [i, 2] - [j, 2]
    }                                         // [i, 2]
    int k;                                    // [i, 2] - [k, -254782]
}
```

## Les zones mémoire

# Le tas (heap)

- **Création d'un objet** : opérateur **new**.
- **Destruction** : opérateur **delete** ou **delete[]** pour les tableaux.

```
#include <iostream>

int main() {
    float* pf = new float(0.17f);
    *pf = *pf + 1.12f;
    std::cout << *pf << std::endl;
    delete pf;

    float* tab= new float[31];
    int* vec = new int[31];
    delete[] tab;
    delete[] vec;
}
```

## Les références

### 1 Introduction générale

### 2 Un premier programme

### 3 Gestion de la mémoire en C++

Les pointeurs

Les zones mémoire

Les références

### 4 La POO en C++

### 5 Outils de développement et bonnes pratiques

## Les références

# Problème

Je veux écrire une fonction qui échange les valeurs de deux entiers.

```
void SwapPtr(int* nombre_gauche, int* nombre_droite) {
    int nombre_tmp = *nombre_gauche;

    *nombre_gauche = *nombre_droite;
    *nombre_droite = nombre_tmp;
}
```

```
int main() {
    int n1(42);
    int n2(37);

    SwapPtr (&n1, &n2);

    return 0;
}
```

## Les références

# Solution utilisant les références : à privilégier au maximum

```
void SwapRef(int& nombre_gauche, int& nombre_droite) {  
    int nombre_tmp = nombre_gauche;  
  
    nombre_gauche = nombre_droite;  
    nombre_droite = nombre_tmp;  
}
```

```
int main() {  
    int n1(42);  
    int n2(37);  
  
    SwapRef(n1, n2);  
  
    return 0;  
}
```



## Les références

# Les références (1)

## Définition

Une référence peut être vue comme un **alias d'une variable**. C'est à dire qu'utiliser la variable, ou une référence à cette variable est équivalent. Ce qui signifie que l'on peut modifier le contenu de la variable en utilisant une référence.

## Propriété

Une référence ne peut être initialisée qu'une seule fois : à la déclaration. Toute autre affectation modifie en fait la variable référencée. **Une référence ne peut donc référencer qu'une seule variable tout au long de sa durée de vie.**



## Les références

# Les références (2)

Les références...

- Sont à préférer aux pointeurs lorsque c'est possible car elles sont plus sûres
- Permettent d'avoir un code plus lisible
- Sont encore plus pratiques lorsqu'on vient à manipuler des classes
- Permettent d'utiliser moins de mémoire
- Permettent de modifier des arguments d'une fonction et de cette manière, avoir plusieurs valeurs de sortie différentes.

## Les classes en C++

- 1 Introduction générale
- 2 Un premier programme
- 3 Gestion de la mémoire en C++
- 4 La POO en C++  
    Les classes en C++
- 5 Outils de développement et bonnes pratiques

# Comment écrire un classe en C++ ? (1)

Déclaration d'une classe en C++ : exemple fichier ma\_classe.h :

```
#ifndef MON_PROJET_MA_CLASSE_H_
#define MON_PROJET_MA_CLASSE_H_


namespace mareva {

class MaClasse {
public:
    // Constructeur par défaut. Permet d'instancier
    // des tableaux de MaClasse.
    MaClasse();
    // Autre constructeur.
    MaClasse(int num);
    // Constructeur par recopie.
    MaClasse(const MaClasse& autre_instance);
    // Destructeur
}
```

```
~MaClasse();

// Surcharge de l'opérateur assignement.
MaClasse& operator=(const MaClasse& autre_instance);

// Méthode de classe.
void foo();

private:
    int num_points_;
    float* tab_;
}; // attention au point virgule ici!

} // namespace mareva

#endif // MON_PROJET_MA_CLASSE_H_
```

## Les classes en C++

# Comment écrire un classe en C++ ? (2)

Définition d'une classe en C++ : exemple fichier `ma_classe.cc` :

```

#include <iostream>
#include "ma_classe.h"

mareva::MaClasse::MaClasse()
: num_points(0),
  tab_(nullptr) {}

mareva::MaClasse::MaClasse(int num)
: num_points(num),
  tab_(nullptr) {
  tab_ = new float[num];
}

mareva::MaClasse::MaClasse(const MaClasse&
    autre_instance) {
  num_points_ = autre_instance.num_points_;

  delete[] tab_;
  tab_ = nullptr;
  tab_ = new float[num_points_];

  for (int i(0); i<num_points_; ++i) {
    tab_[i] = autre_instance[i];
  }
}
  
```

```

mareva::MaClasse::~MaClasse() {
  delete[] tab_;
}

mareva::MaClasse& mareva::MaClasse::operator=(const
    MaClass& autre_instance) {
  num_points_ = autre_instance.num_points_;

  delete[] tab_;
  tab_ = nullptr;
  tab_ = new float[num_points_];

  for (int i(0); i<num_points_; ++i) {
    tab_[i] = autre_instance[i];
  }
}

void mareva::MaClasse::foo() {
  std::cout << "foo !" << std::endl;
}
  
```

# Comment utiliser un classe en C++ ?

Exemple fichier utilisation\_ma\_classe.cc :

```
#include "ma_classe.h"

int main(int argc, char* argv[]) {
    // instanciation avec le constructeur par defaut
    mareva::MaClasse une_instance;
    // instanciation avec le deuxième constructeur
    mareva::MaClasse une_autre_instance(3);
    // appel au constructeur par defaut
    mareva::MaClasse* un_tableau = new mareva::MaClasse[255];

    une_instance.foo();

    mareva::MaClasse* une_instance_dynamique = new mareva::MaClasse(8);
    une_instance_dynamique->foo();

    delete une_instance_dynamique;
    delete[] un_tableau;
}
```

- 1 Introduction générale
- 2 Un premier programme
- 3 Gestion de la mémoire en C++
- 4 La POO en C++
- 5 Outils de développement et bonnes pratiques
  - Git
  - CMake

# Git

## Hébergement :

- Github
- Gitlab
- Bitbucket

## Outils :

- Source tree (gui windows)
- Git cola (gui linux)

- 1 Introduction générale
- 2 Un premier programme
- 3 Gestion de la mémoire en C++
- 4 La POO en C++
- 5 Outils de développement et bonnes pratiques
  - Git
  - CMake

## Rappel : compilation d'un programme

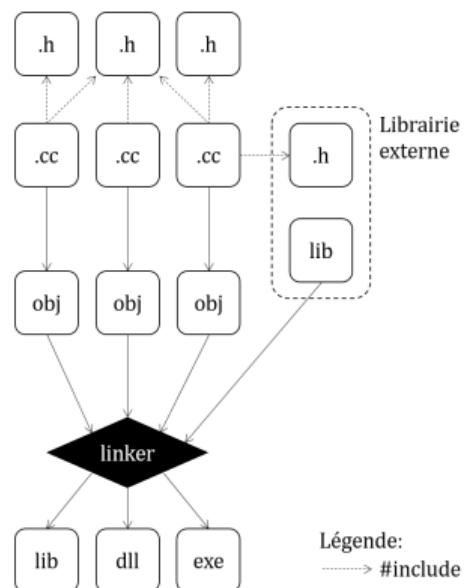


Figure: Étapes de l'édition des liens d'un programme en C++

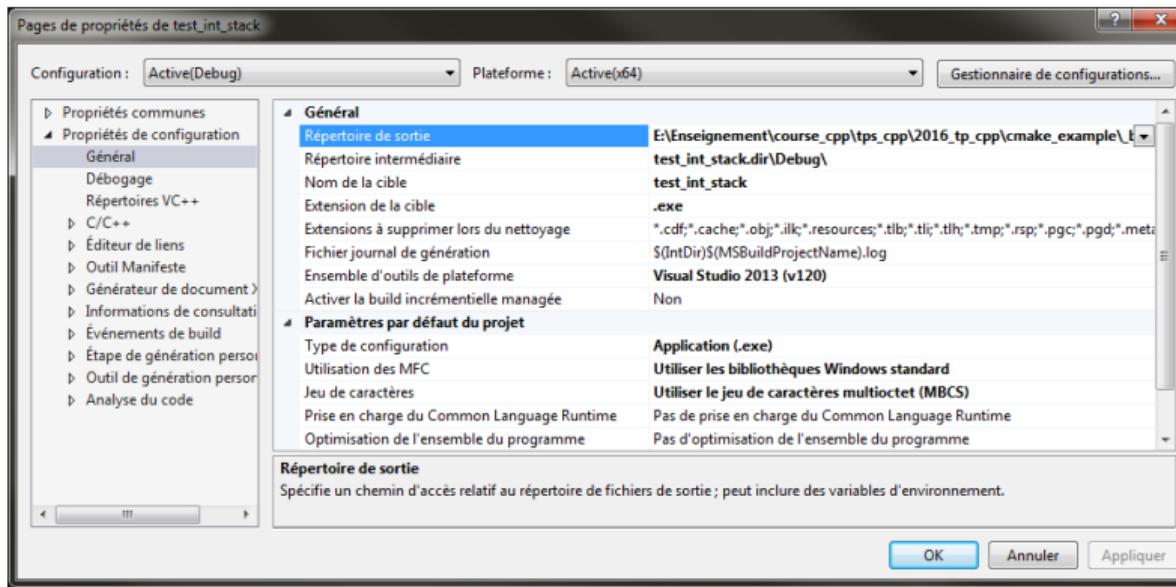


Figure: Interface de configuration d'un projet sous Visual Studio 2013

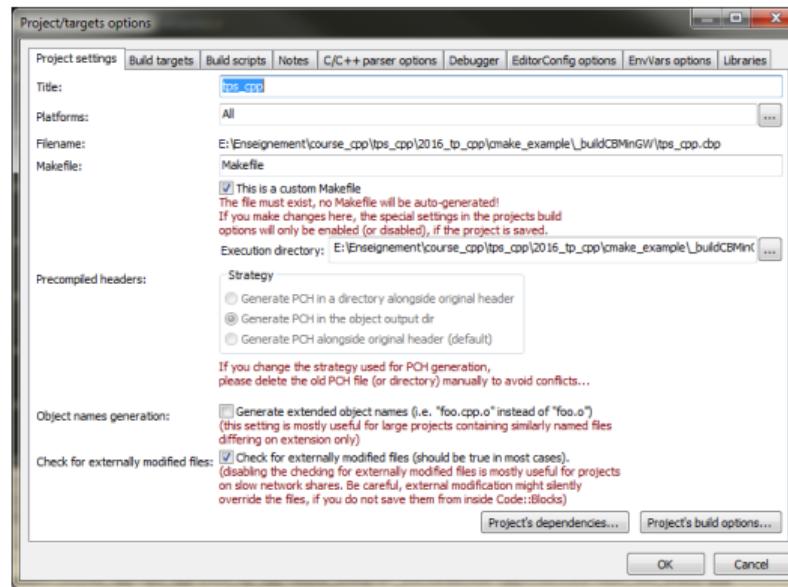
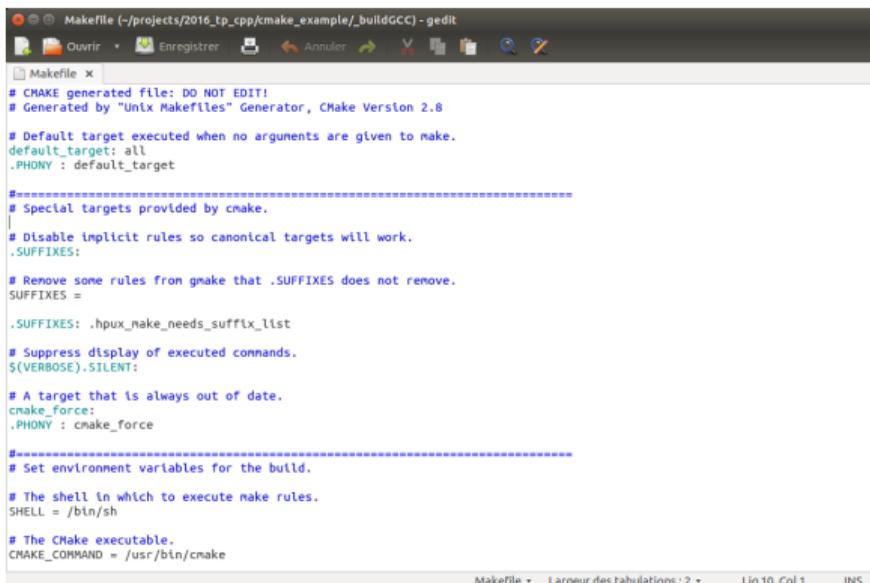


Figure: Interface de configuration du projet sous CodeBlocks

## CMake



The screenshot shows a Gedit text editor window with a single tab labeled "Makefile". The file content is a standard CMake-generated Makefile. The code includes comments explaining various CMake commands and their purposes, such as setting default targets, disabling implicit rules, and defining suffixes. It also specifies environment variables like SHELL and CMAKE\_COMMAND, and defines targets like .PHONY and .SILENT.

```
# CMAKE generated file: DO NOT EDIT!
# Generated by "Unix Makefiles" Generator, CMake Version 2.8

# Default target executed when no arguments are given to make.
default_target: all
.PHONY : default_target

#-----
# Special targets provided by cmake.

# Disable implicit rules so canonical targets will work.
.SUFFIXES:

# Remove some rules from gmake that .SUFFIXES does not remove.
SUFFIXES =

.SUFFIXES: .hpx_nake_needs_suffix_list

# Suppress display of executed commands.
$(VERBOSE).SILENT:

# A target that is always out of date.
.cmake_force:
.PHONY : cmake_force

#-----
# Set environment variables for the build.

# The shell in which to execute make rules.
SHELL = /bin/sh

# The CMake executable.
CMAKE_COMMAND = /usr/bin/cmake
```

Figure: Makefile du projet



# Problème de l'environnement de développement

## Problème 1

Comment s'y retrouver parmi tout ça ?

## Problème 2

Comment travailler (efficacement) avec des gens qui n'utilisent pas les mêmes outils de développement ?

## Problème 3

Comment travailler (efficacement) avec des gens qui ne travaillent pas sur les mêmes plateformes ?

# La solution

## Solution

Utiliser CMake !



- CMake est un moteur de production multi-plateforme.
- Il permet aussi de gérer plusieurs options de compilation...
- ... et de faire plein de truc cools qui dépassent le cadre de ce cours.

# Bien organiser son code

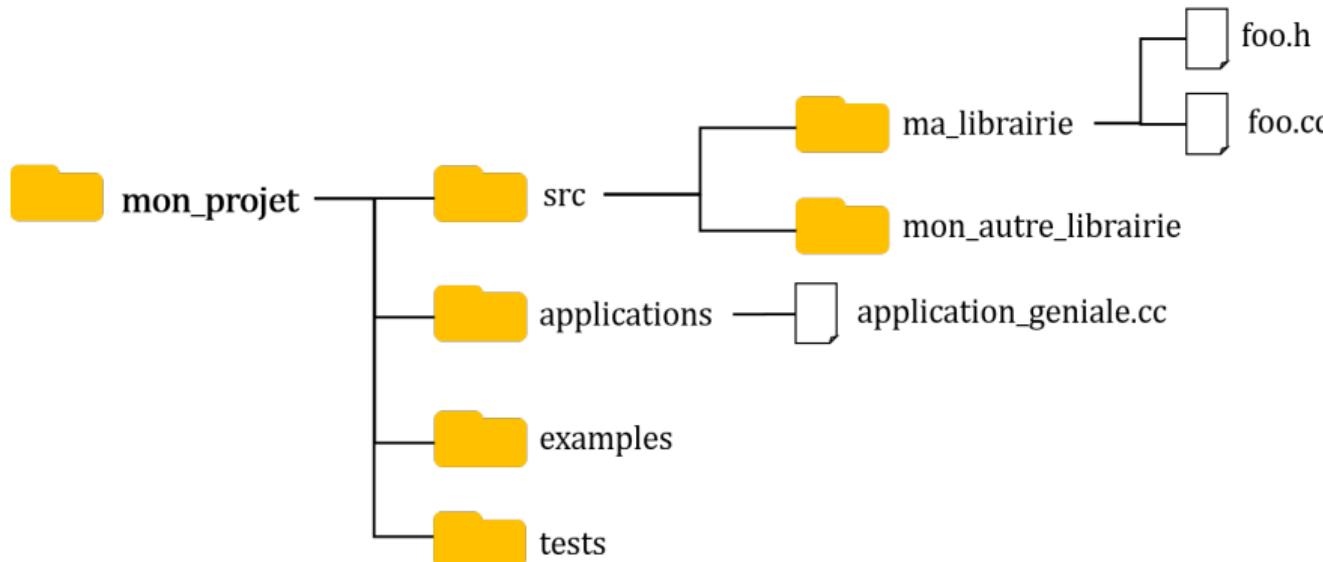


Figure: Arborescence classique d'un projet en C++

## Les fichiers CMakeLists.txt (1)

**Exemple :** premier fichier CMakeLists.txt contenu dans le dossier principal du projet "mon\_projet".

```
cmake_minimum_required (VERSION 2.6)
# définition du nom du projet.
project (mon_projet)

# options de compilation spécifiques lorsque le compilateur est gcc.
if(CMAKE_COMPILER_IS_GNUCXX)
    add_definitions(-std=c++11)
    add_definitions(-ffriend-injection)
endif()

# options pour Visual Studio (pas nécessaire).
set_property(GLOBAL PROPERTY USE_FOLDERS ON)
set_property(GLOBAL PROPERTY PREDEFINED_TARGETS_FOLDER "_CMakePredefinedTargets")

# on spécifie les répertoires d'include de manière globale
include_directories(src)

# avec la commande add_subdirectory, on va chercher les fichiers CMakeLists.txt
# dans les sous-dossiers spécifiés.
add_subdirectory(src)
add_subdirectory(applications)
```



## Les fichiers CMakesLists.txt (2)

Exemple : fichier CMakeLists.txt contenu dans le dossier "mon\_projet/src".

```
# commande très utile, permet d'aller chercher tous les fichiers qui collent
# à un pattern, ici récursivement tous les fichiers .cc et .h dans le dossier
# ma_libreria/. Ces fichiers sont stockés dans la variable ma_libreria_SRC.
file(GLOB_RECURSE ma_libreria_SRC "ma_libreria/*.h"
                               "ma_libreria/*.cc")
# on crée une librairie statique à partir des sources collectées.
add_library(ma_libreria_s STATIC ${ma_libreria_SRC})
# option sous Visual Studio (pas nécessaire)
set_target_properties(ma_libreria_s PROPERTIES FOLDER "libraries")
```

## Les fichiers CMakesLists.txt (3)

Exemple : fichier CMakeLists.txt contenu dans le dossier "mon\_projet/applications".

```
# on cree un executable "application_geniale"
add_executable(application_geniale application_geniale.cc )
# on specifie que cet executable doit inclure la librairie ma_librarie_s
# generee egalement par ce projet.
# cmake s'occupe automatiquement d'ajouter les chemins permettant de trouver
# les fichiers d'include lies a cette librairie.
target_link_libraries(application_geniale ma_librarie_s)
# option sous Visual Studio (pas necessaire)
set_target_properties(application_geniale PROPERTIES FOLDER "applications")
```

## Executer CMake

Une fois les fichiers CMakeLists.txt ont été écrits, on peut lancer CMake qui va s'occuper de générer le projet lié à notre environnement de développement.  
On crée le dossier de build :

```
$$ mkdir _buildGCC  
$$ cd _buildGCC
```

On lance CMake :

```
$$ cmake .. -G "Unix Makefiles"
```

On peut maintenant lancer make :

```
$$ make
```