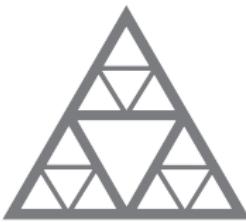


# PRALG: rappels de C++

Pascal Monasse  
[pascal.monasse@enpc.fr](mailto:pascal.monasse@enpc.fr)

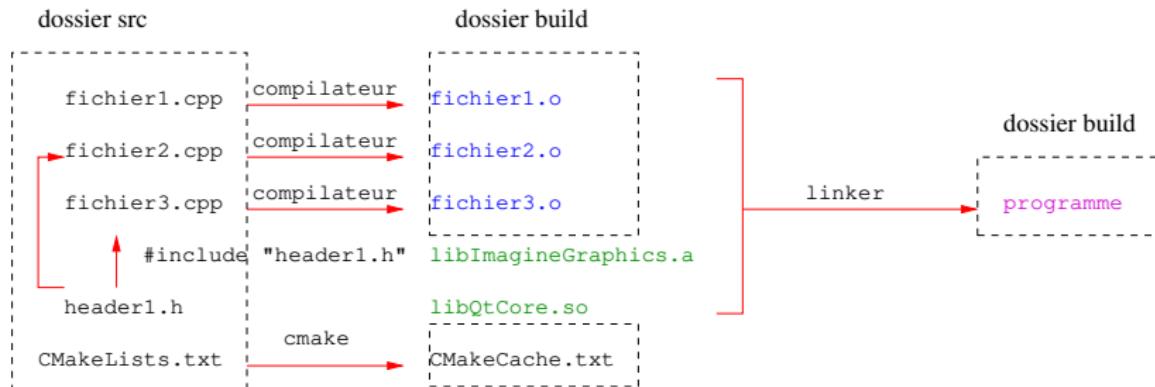


ÉCOLE NATIONALE DES  
**PONTS**  
ET CHAUSSÉES



**IP PARIS**

# Build



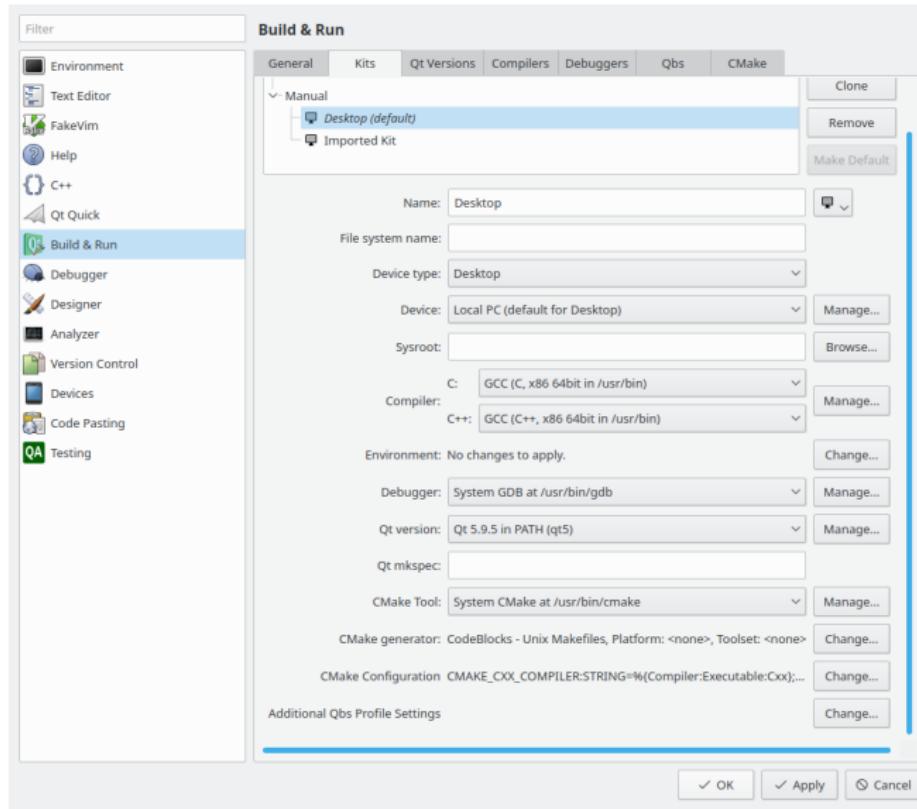
## CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10)
project(programme)
find_package(Imagine REQUIRED COMPONENTS
Graphics)
add_executable(programme fichier1.cpp fichier2.cpp
fichier3.cpp header1.h)
target_link_libraries(programme Imagine::Graphics)
```

## Compilateurs :

- ▶ gcc (GNU compiler collection)
- ▶ mingw32 (Minimal GNU for Windows)
- ▶ clang

# Menu “Tools/Options” de QtCreator



Les outils  
(cmake,  
compilateur,  
version de  
Qt) sont  
rassemblés  
dans un kit  
de  
QtCreator

# Problèmes divers

The screenshot shows a software interface for managing CMake projects. On the left, the 'Projects' panel displays a project named 'Project' containing files like 'CMakeLists.txt', 'fichier1.cpp', 'fichier2.cpp', 'fichier3.cpp', and 'header1.h'. The 'CMake Modules' section is also visible. The main workspace consists of four code editors arranged in a 2x2 grid. The top-left editor shows 'CMakeLists.txt' with the following content:

```
cmake_minimum_required(VERSION 2.8)
find_package(Imagine)
add_executable(programme fichier1.cpp
               fichier2.cpp
               fichier3.cpp
               header1.h)
ImagineModules(programme Graphics)
```

The other three editors show parts of 'header1.h' and 'fichier2.cpp', 'fichier3.cpp', respectively, which include the 'header1.h' file and define functions 'f1()', 'f2()', and 'f3()'.

At the bottom, the 'Compile Output' tab shows the build log:

```
12:33:50: Running steps for project Project...
12:33:50: Persisting CMake state...
12:33:50: Starting: "/usr/bin/cmake" --build . --target all
Scanning dependencies of target programme
[ 25%] Building CXX object CMakeFiles/programme.dir/fichier1.cpp.o
[ 50%] Building CXX object CMakeFiles/programme.dir/fichier2.cpp.o
[ 75%] Building CXX object CMakeFiles/programme.dir/fichier3.cpp.o
[100%] Linking CXX executable programme
[100%] Built target programme
12:33:51: The process "/usr/bin/cmake" exited normally.
12:33:51: Elapsed time: 00:01.
```

The status bar at the bottom includes icons for file operations and a search bar.

Pas de  
problème

# Problèmes divers

The screenshot shows a development environment with the following interface elements:

- File, Edit, Build, Debug, Analyze, Tools, Window, Help**: Standard menu bar.
- Projects**: Left sidebar with icons for Welcome, Editor, Design, Projects, and Help.
- Project Tree**: Shows a project named "Project" containing a "CMakeLists.txt" file and a "programme" directory with files "fichier1.cpp", "fichier2.cpp", "fichier3.cpp", and "header1.h".
- Code Editors**: Four code editors showing the contents of CMakeLists.txt, header1.h, fichier2.cpp, and fichier3.cpp.
- Open Documents**: List of open files: CMakeLists.txt, fichier1.cpp, fichier2.cpp, fichier3.cpp, and header1.h. "fichier3.cpp" is currently selected.
- Compile Output**: Terminal window showing the compilation process and errors. The output includes:

```
12:48:30: Running steps for project Project...
12:48:30: Starting: "/usr/bin/cmake" --build . --target all
Scanning dependencies of target programme
[ 25%] Building CXX object CMakeFiles/programme.dir/fichier1.cpp.o
[ 25%] Building CXX object CMakeFiles/programme.dir/fichier2.cpp.o
[ 25%] Building CXX object CMakeFiles/programme.dir/fichier3.cpp.o
[ 25%] Linking CXX executable programme
[ 25%] Built target programme
fichier3.cpp:8:5: error: 'f2' was not declared in this scope
    f2();
      ^
fichier3.cpp:8:5: note: suggested alternative: 'f3'
    f2();
      ^
    f3
fichier3.cpp:8:5: note: suggested alternative: 'f2'
    f2();
      ^
fichier3.cpp:8:5: note: suggested alternative: 'f2'
make[2]: *** [CMakeFiles/programme.dir/fichier3.cpp.o] Error 1
CMakeFiles/programme.dir/build.make:110: recipe for target 'CMakeFiles/programme.dir/fichier3.cpp.o' failed
```

**Status Bar**: Type to locate (Ctrl+K) and navigation icons.

Problème  
compilation :  
f2 non  
connue dans  
fichier3.cpp  
(manque  
#include)

# Problèmes divers

The screenshot shows a C++ development environment with the following details:

- Project Structure:** A tree view shows a project named "Project" containing a "programme" folder with files "fichier1.cpp", "fichier2.cpp", "fichier3.cpp", and "header1.h".
- Code Editors:** There are four code editors visible:
  - CMakeLists.txt:** Contains CMake configuration.
  - fichier1.cpp:** Contains a function f1() returning 1.
  - header1.h:** Contains a function f2() returning 2.
  - fichier3.cpp:** Contains a function f3() returning 3 and a main() function calling f2() and f3().
- Compile Output:** Shows the following build logs:

```
12:58:13: Running steps for project Project...
12:58:13: [ 5%] Building CXX object CMakeFiles/programme.dir/fichier3.cpp.o
[ 50%] Building CXX object CMakeFiles/programme.dir/fichier3.cpp.o
[ 75%] Linking CXX executable programme
CMakeFiles/programme.dir/fichier3.cpp.o: In function `f2()':
fichier3.cpp:(.text+0x0): multiple definition of `f2()'
CMakeFiles/programme.dir/fichier2.cpp.o:fichier2.cpp:(.text+0x0): first defined here
CMakeFiles/programme.dir/build.make:152: recipe for target 'programme' failed
make: [programme] Error 2
make[3]: recipe for target 'all' failed
make[2]: *** [programme] Error 1
```
- Open Documents:** Lists "CMakeLists.txt", "fichier1.cpp", "fichier2.cpp", "fichier3.cpp", and "header1.h".
- Bottom Bar:** Includes tabs for Issues, Search Res., Application, Compile O., Debugger, General M., Test Results, and a search bar.

Problème de linker :  
f2 compilée deux fois :  
par  
fichier2.cpp  
et  
fichier3.cpp  
⇒ Pas de *définition* dans header,  
seulement dans cpp

# Problèmes divers

The screenshot shows a development environment with multiple windows:

- Projects:** Shows a project named "Project" with files: CMakeLists.txt, programme (containing fichier1.cpp, fichier2.cpp, fichier3.cpp), and header1.h.
- Code Editors:** Three panes showing code:
  - CMakeLists.txt (top left):

```
cmake_minimum_required(VERSION 2.8)
find_package(Imagine)
add_executable(programme fichier1.cpp
fichier2.cpp fichier3.cpp header1.h)
Imagine${semModules}(programme Graphics)
```
  - header1.h (top right):

```
#pragma once
int f2();
int f3();
```
  - fichier2.cpp (bottom left):

```
#include "header1.h"
int f2() {
    return 2;
}
```
  - fichier3.cpp (bottom right):

```
#include "header1.h"
int f3() {
    return 3;
}
//f2()
//f2();
//f3();
//f3();
```
- Compile Output:** Shows the build process:

```
13:51:38: Running steps for project Project...
13:51:38: Scanning dependencies of target programme
[ 25%] Building CXX object CMakeFiles/programme.dir/fichier3.cpp.o
[ 50%] Building CXX object CMakeFiles/programme.dir/fichier3.cpp.o
[ 75%] Linking CXX executable programme
/usr/lib/gcc/x86_64-linux-gnu/7/../../../../x86_64-linux-gnu/Scrt1.o: In function `__start':
(.text+0x20): undefined reference to `main'
collect2: error: ld returned 1 exit status
CMakeFiles/programme.dir/build.make:152: recipe for target 'programme' failed
CMakeFiles/programme.dir/build.make:152: recipe for target 'CMakeFiles/programme.dir/all' failed
make[2]: recipe for target 'all' failed
make[1]: *** [programme] Error 1
make: *** [programme] Error 1
make[1]: *** [CMakeFiles/programme.dir/all] Error 2
```
- Status Bar:** Includes links for Issues, Search Results, Application, Compile, Debugger, General M..., Test Results, and a search field.

Problème  
de linker :  
fonction  
main non  
trouvée

# Problèmes divers

The screenshot shows a development environment with multiple windows:

- Projects:** Shows a project named "Project" with files: CMakeLists.txt, programme (containing fichier1.cpp, fichier2.cpp, fichier3.cpp), and header1.h.
- Code Editors:** Five windows showing code:
  - CMakeLists.txt: cmake\_minimum\_required(VERSION 2.8) find\_package(Imagine) add\_executable(programme fichier1.cpp fichier3.cpp header1.h) ImagineUseModules(programme Graphics)
  - header1.h: #pragma once int f2();
  - fichier1.cpp: (empty)
  - fichier2.cpp: #include "header1.h" int f2() { return 2; }
  - fichier3.cpp: #include "header1.h" int f3() { return 3; } int main() { f2(); f3(); return 0; }
- Compile Output:** Shows the build process and an error:

```
13:06:36: Running steps for project Project...
13:06:36: Running CMake in preparation to build...
13:06:37: Starting: "/usr/bin/cmake" --build . --target all
Scanning dependencies of target programme
[ 33%] Building CXX object CMakeFiles/programme.dir/fichier3.cpp.o
60% Linking CXX executable programme
CMakeFiles/programme.dir/fichier3.cpp.o: In function `main':
fichier3.cpp(.text+0x10): undefined reference to `f2()'
collect2: error: ld returned 1 exit status
CMakeFiles/programme.dir/build.make:126: recipe for target 'programme' failed
CMakeFiles/Makefile2:67: recipe for target 'CMakeFiles/programme.dir/all' failed
Makefile:83: recipe for target 'all' failed
make[2]: *** [programme] Error 1
make[1]: *** [CMakeFiles/programme.dir/all] Error 2
```
- Open Documents:** Shows files: CMakeLists.txt, fichier1.cpp, fichier2.cpp, fichier3.cpp, and header1.h.
- Bottom Bar:** Includes tabs for Issues, Search Results, Application, Compile Output, Debugger, General M..., Test Results, and icons for Help, Undo, Redo, and Find.

Problème de linker :  
f2 non compilée car fichier2.cpp n'est pas dans le CMakeLists

# Fonctions

```
type_de_retour nom_fonction()
type_de_retour nom_fonction([const] type1[&] arg1)
type_de_retour nom_fonction([const] type1[&] arg1, [const]
                           type2[&] arg2)
```

- ▶ **nom\_fonction** : combinaison de a–z, A–Z, 0–9 et \_ sauf prefixes 0–9 (interdits), \_\_ et \_[A–Z] (réservés au compilateur)
- ▶ **type\_de\_retour** : **void**, un des types de base (**bool**, (**unsigned**) **char**, (**unsigned**) **short** (**int**), (**unsigned**) **int**, (**unsigned**) **long** (**int**), (**unsigned**) **long long** (**int**), **float**, **double**, **long double**), un objet de la bibliothèque standard (**std :: string**, **std :: pair<type1,type2>**) ou non (défini par **class** ou **struct**).
- ▶ Passage par référence & pour modifier la variable

```
void somme(double& x, double y) { x += y; }
int main() {
    double f=3.14, g=-1.0;
    somme(f,g); //OK, modifie f
    somme(f,1.0); //OK
    somme(3.14,g); //Erreur, le premier argument n'est pas une variable
}
```

# Tableaux

## Statique

```
int m=10;
int tab[m]; //KO, m var
const int n=10;
int tab[n]; //OK, n const
tab[0]=0; //1er element
for(int i=1; i<n; i++)
    tab[i] = tab[i-1]+i;
tab[n] = 0; //BUG indice
```

## Arg de fonction sans réf.

```
void init(int tab[10]) {
    for(int i=0; i<10; i++)
        tab[i]=0;
int* t1=new int[10]; init(t1);
delete [] t1;
int t2[20]; init(t2);
int t3[5]; init(t3); //KO
```

## Dynamique

```
int m=10;
int* tab=new int[m]; //Alloc
tab[0]=0; //1er element
for(int i=1; i<m; i++)
    tab[i] = tab[i-1]+i;
cout << tab[m-1] << endl;
delete [] tab; //Desalloc
```

## Linéarisation tableau 2D

```
int m=1000, n=500;
float* mat=new float[m*n];
for(int j=0; j<n; j++)
    for(int i=0; i<m; i++)
        mat[j*m+i]=0;
delete [] mat;
```

# Embranchements et boucles

## Embranchements

```
if(condition)           if(condition)      if(condition1)
    instructionVrai;   instructionVrai;  instructionVrai1;
else                           else           instructionVrai2;
    instructionFaux;   instructionFaux;  instructionFaux2;
else
```

## Boucles

```
for(inst1; cond; inst2){      while(cond){      do{
    inst;                      inst;          inst;
}                                }                }
{ //nouveau scope, confine une decl. dans inst1
  inst1;
  while(cond) {
    inst;
    inst2;
  }
}
```

for  $\Leftrightarrow$

# Classes

- ▶ Alias d'un type existant (les deux deviennent équivalents) :

```
typedef unsigned char byte;
```

- ▶ Nouveau type :

```
struct Matrice {  
    int m, n;  
    double* tab;  
}; //Ne pas oublier le ; apres cette accolade  
void init(Matrice& M) {  
    for(int i=0; i<M.m*M.n; i++)  
        M.tab[i]=0;  
}  
Matrice cree(int nlig, int ncol) {  
    Matrice M = {nlig, ncol, new double[nlig*ncol]};  
    init(M);  
    return M;  
}  
int main() {  
    Matrice M = cree(10,2000);  
    M.tab[2000*3+1416] = 2.718;  
    delete [] M.tab; //Ne pas oublier !  
}
```

# Méthodes

On ajoute la méthode (=fonction de classe) `void Matrice:: init ()`.

```
class Matrice { //Dans .h
public:
    int m, n;
    double* tab;
    void init() { // def "inline"
        for(int i=0; i<m*n; i++)
            tab[i]=0;
    }
}; //Ne pas oublier le ; apres cette accolade
Matrice cree(int nlig, int ncol) { // dans .cpp
    Matrice M;
    M.m=nlig; M.n=ncol; M.tab=new double[nlig*ncol];
    M.init();
    return M;
}
```

# Méthodes

Idem avec définition non inline :

```
class Matrice { // dans .h
public:
    int m, n;
    double* tab;
    void init(); //declaration
}; //Ne pas oublier le ; apres cette accolade
Matrice cree(int nlig, int ncol) { // dans .cpp
    Matrice M;
    M.m=nlig; M.n=ncol; M.tab=new double[nlig*ncol];
    M.init();
    return M;
}
void Matrice::init() { // definition dans .cpp
    for(int i=0; i<m*n; i++)
        tab[i]=0;
}
```

# Opérateurs

- ▶ Les types numériques comprennent les opérateurs suivants :  
+ - \*/ (et % pour types entiers), et les affectations du genre  
+= ainsi que les comparaisons <, >, <=, >=, ==, !=  
pour les types numériques. Pour les **bool** : ==, !=, !  
(négation unaire, i.e. 1 seul argument), &&, ||.
- ▶ Il faut définir soi-même ceux dont on veut :

```
bool operator==(Matrice m1, Matrice m2) {  
    if (m1.m!=m2.m || m1.n!=m2.n) return false;  
    for (int i=0; i<m1.m*m1.n; i++)  
        if (m1.tab[i]!=m2.tab[i])  
            return false;  
    return true;  
}  
Matrice m1=cree(2,2), m2=cree(2,2);  
cout << m1==m2 << endl; //=> operator==(m1,m2)  
cout << m1!=m2 << endl; //OK, même si pas explicite
```

La définition de **operator==** provoque celle de **operator!=**.

# Opérateurs

- ▶ Restriction : les deux types d'un opérateur binaire peuvent différer, mais l'un d'eux doit être un objet.
- ▶ Encapsulation de l'opérateur dans la classe :

```
bool operator==(matrice m1, matrice m2);  
bool Matrice::operator==(matrice m2);
```

- ▶ On peut alors ajouter `operator()` et `operator[]`. `operator()` est spécial car on peut le définir avec un nombre arbitraire d'arguments (y compris 0). `operator[]` n'a qu'un argument.

```
class Matrice { ...  
    double operator()(int i, int j) const;  
    double& operator()(int i, int j);  
};  
cout << M(0,2) << endl; M(1,1) = 0;
```

Notez qu'on a les versions `const` et non-`const` (non appellable pour les matrices non-`const` et permettant de modifier le coefficient).

# Constructeurs/destructeur

- ▶ On cache les champs de Matrice en `private`, seules les méthodes peuvent y accéder. Il est bon d'avoir un ou des constructeurs :

```
class Matrice {  
    double* tab; int m,n;  
public:  
    Matrice(int nligcol); //matrice carree  
    Matrice(int nlig, int ncol);  
    ~Matrice(); // destructeur  
}  
Matrice:: Matrice(int nligcol) {  
    m=n=nligcol;  
    tab = new double[m*n];  
}  
Matrice::~ Matrice() {  
    delete [] tab;  
}  
Matrice M(3);
```

- ▶ Si aucun constructeur n'est défini, on peut quand même créer un objet (constr. sans argument implicite). Il faut un constr. sans arg. pour créer un tableau d'objets.
- ▶ **On n'appelle jamais explicitement le destructeur**, c'est automatique quand l'objet sort de sa portée.

## Constructeurs/destructeur

- ▶ Une fonction (ou méthode) qui prend un objet en arg. sans référence crée une variable locale, construite avec le constructeur par copie :

```
class Matrice {  
public:  
    Matrice(const Matrice& M);  
    void operator=(const Matrice& M);  
}  
Matrice::Matrice(const Matrice& M) {  
    m=M.m; n=M.n;  
    tab = M.tab; // TRES DANGEREUX  
}
```

Le fait de faire `tab=M.tab` provoquera un crash lors de l'appel du destructeur de l'objet copié car on aura déjà libéré son champ `tab` lors du destructeur de la copie. Il faut allouer la propre mémoire pour la matrice (cf TP pour alternative)

- ▶ `operator=` est le plus compliqué : on doit faire l'équivalent du destructeur de l'objet avant de faire la copie de `M`.
- ▶ Inutile de faire de destructeur, const. par copie et `operator=` si votre classe ne fait pas de `new`. Dans le cas contraire, ils sont nécessaires.

# Variables statiques

- Dans une fonction :

```
int my_random() {
    static bool seeded=false;
    if (! seeded) {
        seeded=true;
        srand((unsigned int)time(0));
    }
    return rand();
}
```

- Dans une classe :

```
struct Tool {
    static int nb; int serialNumber;
    Tool() { serialNumber=nb++; }
};
int Tool::nb=0; //init , dans cpp
void func() {
    Tool a,b;
    cout << Tool::nb; //OK
    cout << a.nb << '-' << b.nb; } //=> Tool::nb
```

# Divers

- ▶ Utiliser assert (`#include <cassert>`).
- ▶ `std :: cin` et `std :: cout` (`#include <iostream>`) peuvent aussi être utilisés pour un objet :

```
ostream& operator<<(ostream& str, const Matrice& M) {
    str << M.lignes() << 'x' << M.colonnes();
    return str;
}
istream& operator>>(istream& str, Matrice& M) {
    int nlig, ncol;
    str >> nlig >> ncol;
    M.reset_dim(nlig, ncol);
    return str;
}
cout << M << endl;
cout << "Entrez les nouvelles dimensions de M: -";
cin >> M;
```

- ▶ Le fait de retourner le flot permet d'enchaîner les `<<` et `>>`.
- ▶ Les instructions `continue` et `break` (dans une boucle) permettent de passer à l'itération suivante (en vérifiant la condition) ou de sortir directement.