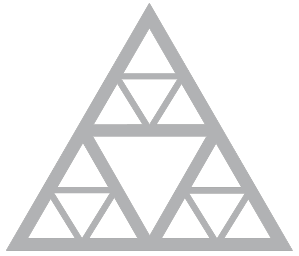
Introduction à la programmation en C++

Fichiers séparés - Opérateurs

Nicolas Audebert

Vendredi 12 octobre 2018



École des Ponts
ParisTech

Avant toute chose

Rendus de TP et des exercices

Les rendus se font sur Educnet.

1. Le code rendu **doit compiler**.
2. Le code rendu doit **être propre** (indentation, noms de variables clairs).
3. Le code rendu doit **être commenté** (réponses aux questions, fonctionnement du code).
4. Rassembler le code dans une seule archive (`.zip`, `.rar`, `.tar.gz`, etc.).

Plan de la séance

Les structures

Définition

Les **structures** C++ permettent de regrouper des variables hétérogènes dans un ensemble cohérent. Une structure définit un nouveau **type**.

```
struct Point{
    double x,y;
};
Point p1={1,2}, p2;
p2 = p1;

struct Cercle{
    Point centre;
    double rayon;
    Color couleur;
};
Cercle c;
c.centre.x = 0.5;
c.couleur = RED;
```

```
Point pt;
pt.x = pt.y = 5.5;
c.centre = pt;

// Tableau de cercles
Cercle liste_cercles[10];
for(int i=0; i<10; i++)
    liste_cercle[i] = c;

Cercle cree_cercle(int rayon, Color col){
    Point origine = {0, 0};
    Cercle res = {origine, rayon, col};
    return res;
}
```

Plan de la séance

Fichiers séparés…

Jusqu'ici, tout le code est organisé un seul fichier qui contient :

- une fonction `int main()` (le point d'entrée du programme),
- des définitions de fonction,
- des définitions de structure.

En pratique

Structurer son code dans plusieurs fichiers permet de :

- mieux organiser le programme (plus lisible, regroupements par modules),
- partager ses fonctions et de les réutiliser dans plusieurs projets,
- accélérer la compilation des gros projets.

Multiples fichiers sources

Ce qu'on aimerait faire, c'est pouvoir définir une fonction dans un fichier et la réutiliser dans un autre.

```cpp
// fichier1.cpp

int ma_fonction(int var){
    ...
    autre(var);
    // ERREUR : autre est
 ↪   inconnue !
    ...
}
```

```cpp
// fichier2.cpp

void autre(int arg){
    ...
    ...
    ...
    ...
    ...
}
```

Attention

Pour pouvoir utiliser une fonction, il faut qu'elle soit connue dans le fichier où on l'utilise.

Multiples fichiers sources

```cpp
// fichier1.cpp

// Signature de autre_fonction
void autre(int arg);

int ma_fonction(int var){
    ...
    autre(var); // OK
    ...
}
```

```cpp
// fichier2.cpp

void autre(int arg){
    ...
}
```

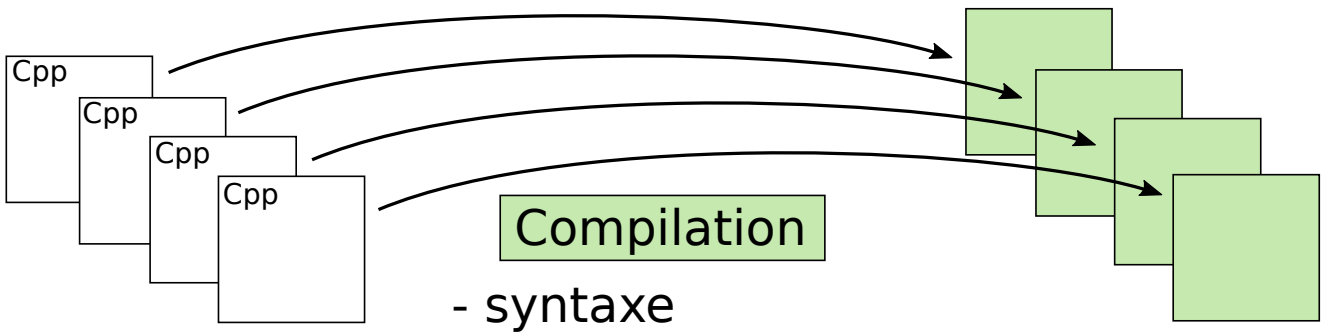
Solution

On **déclare** la **signature** de la fonction dans le fichier qui souhaite l'utiliser pour signifier au compilateur que la fonction existe.

Pourquoi ?

La production d'un exécutable à partir du code source C++ se réalise en deux étapes :

1. La compilation (transforme le code en fichiers objets)
2. L'édition des liens (transformes les fichiers objets en exécutable)



Les sources

Compilation

- syntaxe
- compatibilité des types
- déclaration des fonctions

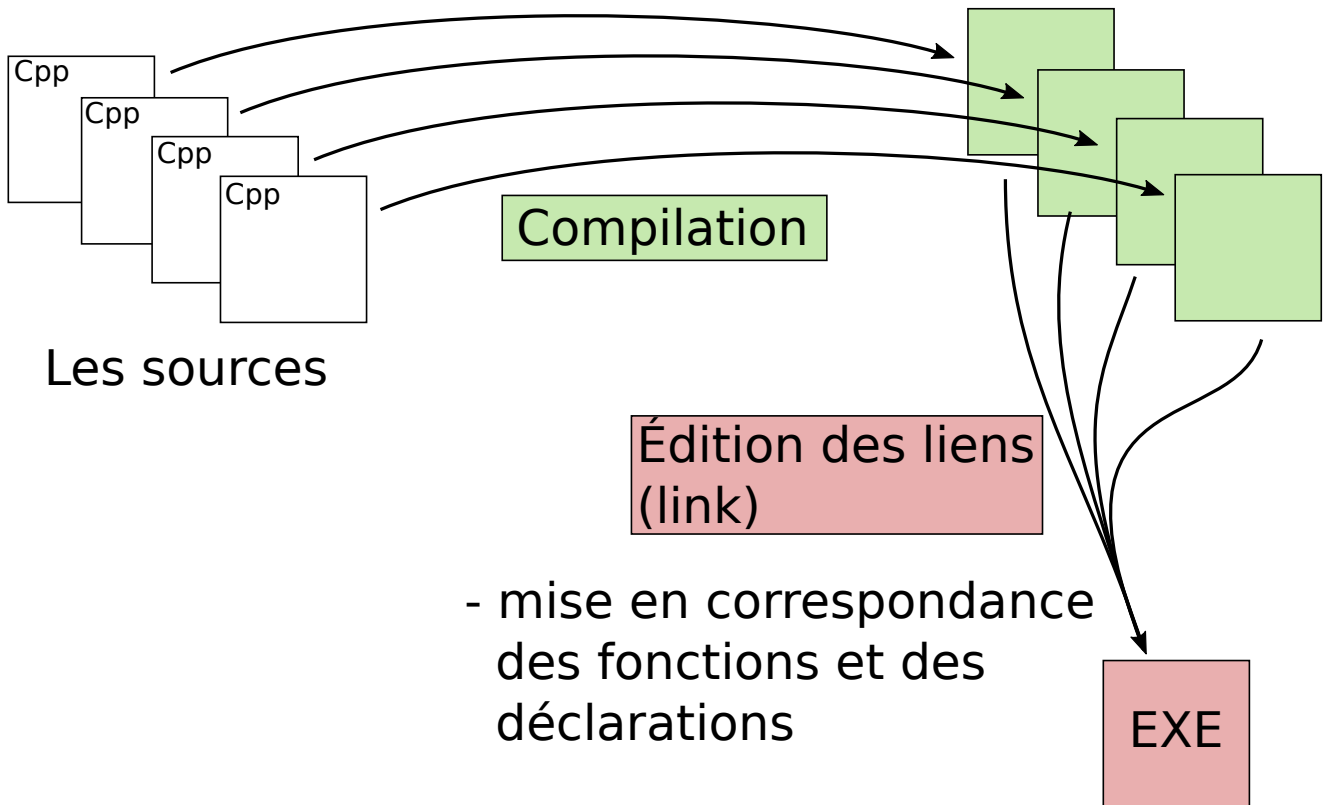
...

Un produit par .cpp

Pourquoi ?

La production d'un exécutable à partir du code source C++ se réalise en deux étapes :

1. La compilation (transforme le code en fichiers objets)
2. L'édition des liens (transformes les fichiers objets en exécutable)



Oui mais...pourquoi ?!

Ce fonctionnement présente deux intérêts :

- La compilation est plus rapide : on ne recompile que les fichiers `.cpp` qui ont été modifiés.
- On peut créer et utiliser des librairies : des fichiers précompilés et réutilisés dans notre propre code (exemple : Imagine++).

CMake

CMake contrôle le compilateur et l'éditeur de liens.

```
cmake_minimum_required(VERSION 2.6)
# On indique à l'éditeur de liens où trouver Imagine++
file(TO_CMAKE_PATH "$ENV{IMAGINEPP_ROOT}/CMake" p)
# On indique que Imagine++ est REQUIS pour compiler
find_package(Imagine REQUIRED)

# On nomme le projet
project(Mon_projet_cpp)

# On indique quels sont les fichiers sources à compiler
# On peut compiler plusieurs programmes différents dans un
# même projet ! (cf. TP Tennis, Mastermind, ...)
add_executable(Mon_executable
 fichier1.cpp fichier2.cpp fichier3.cpp ...
)
# On indique quels sont les modules Imagine++ utilisés par
# les différents exécutables
ImagineUseModules(Mon_executable Graphics)
```

Ajouter un fichier dans un projet existant

Solution pour tous les IDEs

1. créer le fichier dans le même dossier que les autres,
2. modifier le CMakeLists.txt avec un éditeur de texte : ajouter le nom du fichier
3. recompiler le programme dans l'IDE.

Ajouter un fichier dans un projet existant

Solution pour QtCreator

Dans QtCreator :

1. ouvrir le menu `File/New File or Project` ou faire `Ctrl+N`, choisir *C++ Source File*. **Attention : mettre le fichier dans le dossier des sources**,
2. rajouter ce fichier dans le `CMakeLists.txt`,
3. recompiler le programme dans QtCreator.

Les fichiers d'en-tête

Constat

- Pénible de recopier toutes les déclarations dans tous les fichiers **.cpp**,
- Pas de partage des structures de cette façon.

Solution

Mettre toutes les **déclarations** et les **structures** dans des **fichiers d'en-tête** (*header*) repérés par l'extension **.h**

Mise en œuvre

```cpp
// source.cpp

#include "auxiliaire.h"
// similaire au import Python

int ma_fonction(int var){
    ...
    autre(var);
    ...
}
```

```cpp
// auxiliaire.h
// Signature
void autre(int var);

struct Vect{ ... };

// auxiliaire.cpp
#include "auxiliaire.h"

// Définition
void autre(int var){...}
```

Note

Jusqu'à présent, les fonctions externes au projet étaient incluses grâce à la directive `#include<...>`. Il s'agit en fait de headers externes pour différents librairies (librairie standard `std`, librairie Imagine++, ...).

Ajouter un fichier d'en-tête dans un projet

Méthode générique

Exactement comme pour les `.cpp`.

QtCreator

Idem, mais il faut créer un *C++ Header File*.

Inclusions mutuelles

En pratique, la directive **`include`** ne fait que copier et coller l'en-tête dans le fichier source et rien n'empêche les inclusions mutuelles :

```c
// fichier1.h

#include "fichier2.h"

int function(int var);
```

```c
// fichier2.h

#include "fichier1.h"

void aux(double arg);
```

Attention

Boucle dans les inclusions → compilation impossible et plantage.

Se protéger des inclusions mutuelles

La version classique :

```
// fichier1.h
#ifndef NOM_UNIQUE
#define NOM_UNIQUE

#include "fichier2.h"

int function(int var);

#endif
```

La version moderne :

```
// fichier1.h
#pragma once

#include "fichier2.h"

int function(int var);
```

Intérêt

Dans les deux cas, les directives du pré-processeur (préfixées par #) permettent de s'assurer qu'un fichier est inclus au plus une fois.

Plan de la séance

Opérateurs

Les **opérateurs** définissent le comportement de certains signes de ponctuation ou mathématiques :

- `+` `-` `/` `*` `=` ...

Il est possible de redéfinir ces opérateurs pour les utiliser avec les structures que l'on a créé.

Exemple

```
struct Point {
    double x, y;
}
Point v1 = {0,0}, v2 = {1,5};
```

Pour l'instant, on doit écrire :

```
// Addition
Point v3 = {v1.x+v2.x, v1.y+v2.y};
// Produit scalaire
double s = v1.x*v2.x + v1.y*v2.y;
```

Mais on aimerait :

```
Point v3 = v1 + v2;
double s = v1 * v2;
```

Implémentation

```
// opérateur + sur des vecteurs
Point operator+(Point vA, Point vB){
    Vect v = {vA.x+vB.x, vA.y+vB.y};
    return v;
}

// opérateur * sur des vecteurs
double operator*(Point vA, Point vB){
    return vA.x*vB.x + vA.y*vB.y;
}

Point v1 = {1, 2}, v2 = {5, 5};
// addition de deux vecteurs
Point v3 = v1+v2;
// produit scalaire
double s = v1*v2;
```

Surcharge des opérateurs

```
// opérateur * pour deux vecteurs
double operator*(Point vA, Point vB){
    return vA.x*vB.x + vA.y*vB.y;
}

// opérateur * vecteur et réel
Point operator*(Vect vA, double alpha){
    Point v = {alpha*v.x, alpha*v.y};
    return v;
}

Point v1 = {1, 2}, v2 = {5, 5};

// produit scalaire
double s = v1*v2;

// multiplication par un réel
double m = 5.5;
Point v3 = v1 * m;
```

Surcharge des opérateurs

Attention

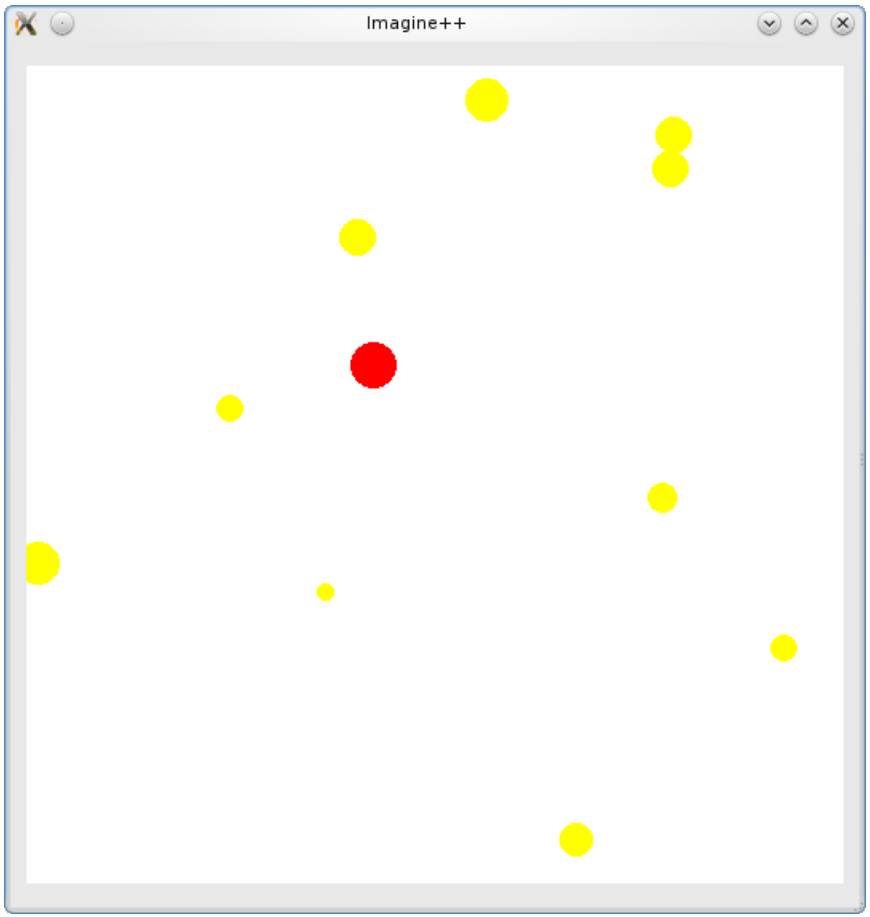
L'ordre des arguments et important : `v1 * m` est différent de `m * v1`. La commutativité doit être explicitement définie.

```
// opérateur * vecteur et réel
Point operator*(Point vA, double alpha){
    Point v = {alpha*v.x, alpha*v.y};
    return v;
}

// opérateur * vecteur et réel
Point operator*(double alpha, Point vA){
    return v*alpha;
}
```

```
Point v1, v2;
...

// multiplication par un
 ↪  réel
double m = 5.5;

Point v3 = v1*m;

Point v4 = m*v2;
```

Plan de la séance

Le TP du jour



On continue le TP ”Gravitation”.

1. Finir le TP de la séance précédente,
2. Organiser son code dans plusieurs fichiers,
3. Utiliser des opérateurs personnalisés pour les calculs.