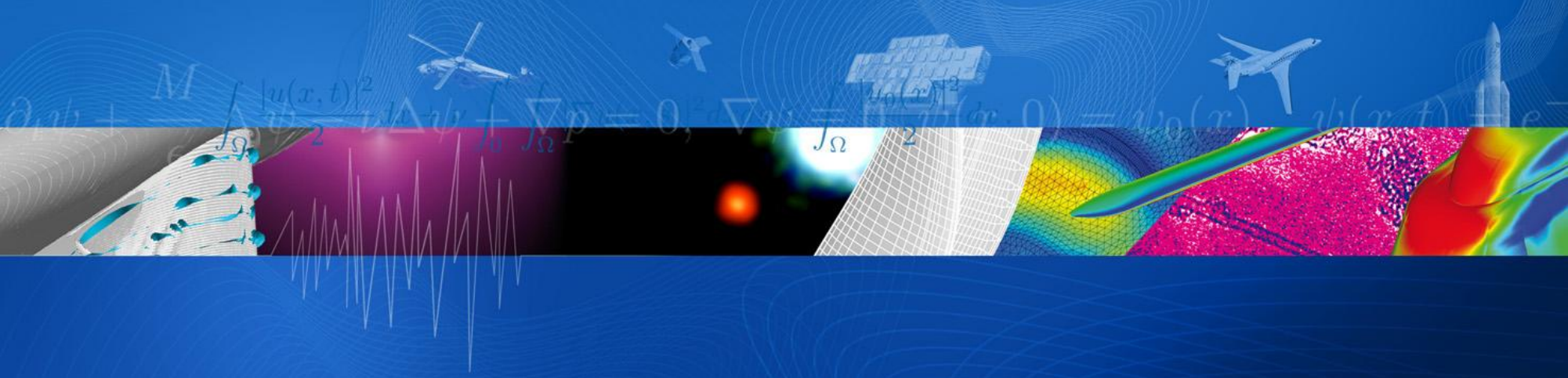


Introduction à la programmation C++

Fichiers séparés / opérateurs

BOULCH Alexandre

ONERA
THE FRENCH AEROSPACE LAB

retour sur innovation

Plan de la séance

Jusqu'à maintenant ...

- un `main`
- des fonctions
- des structures

... mais tout dans le même fichier.

Plusieurs fichiers pour ...

- organiser mieux le code (plus lisible, regrouper par modules)
- partager le code entre plusieurs projets
- accélérer la compilation (sensible pour les gros projets)

Plusieurs fichiers sources

Fonctions dans deux fichiers :

```
// fichier1.cpp


A f1(B b){
    ...
    g2(var1); // Erreur
    ...
}
```

```
// fichier2.cpp


void g2(C c){

    ...

}
```

Attention

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

Plusieurs fichiers sources 2

Fonctions dans deux fichiers :

```
// fichier1.cpp

void g2(C c);

A f1(B b){
     ...
     g2(var1); // OK
     ...
}
```

```
// fichier2.cpp


void g2(C c){

     ...

}
```

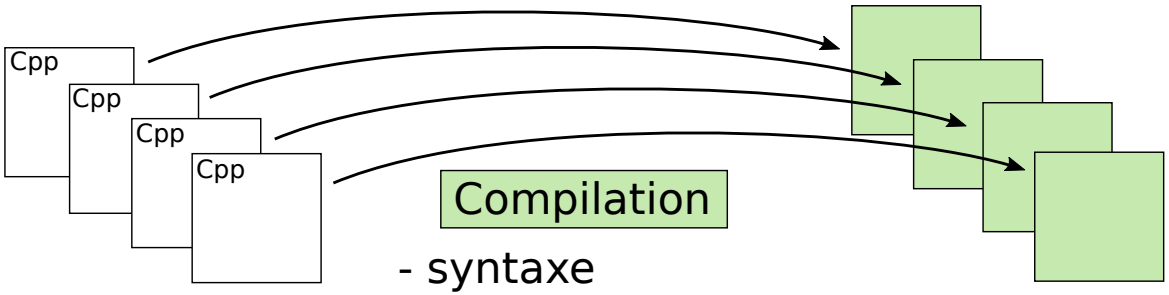
Solution

On déclare la fonction dans le fichier qui utilise la fonction pour dire au compilateur que la fonction existe.

Pourquoi ?

Il y a deux étapes dans pour la production de l'executable.

- La compilation (fichiers objets)



Compilation

- syntaxe
- compatibilité des types
- déclaration des fonctions

...

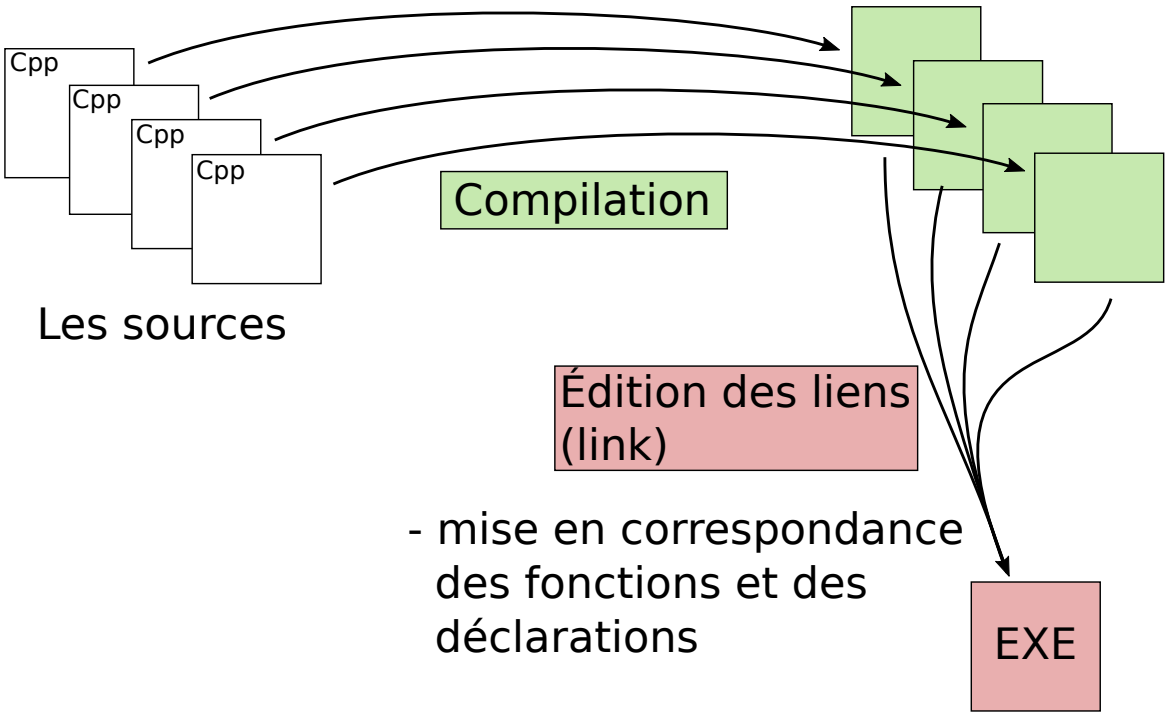
Un produit par .cpp

Les sources

Pourquoi ?

Il y a deux étapes dans pour la production de l'executable.

- La compilation (fichiers objets)
- L'édition des liens (executable)



Pourquoi ? 2

Un fonctionnement en deux étapes pour :

- compiler plus rapidement : on ne recompile que les fichiers modifiés
- possibilité de faire des librairies : fichiers précompilés et appelé dans le code (exemple Imagine++)

CMake

CMakeList.txt

```
CMAKE_MINIMUM_REQUIRED(VERSION 2.6)
list(APPEND CMAKE_MODULE_PATH $ENV{IMAGINEPP_ROOT}/CMake
find_package(Imagine REQUIRED)

PROJECT(Exemple_Cours)

add_executable(Exemple_exec
 fichier1.cpp fichier2.cpp fichier3.cpp ...
 )
ImagineUseModules(Exemple_exec Graphics)
```

Ajouter un fichier dans un projet existant

Solution pour tous les IDEs

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

Ajouter un fichier dans un projet existant

Solution pour QtCreator

- dans QtCreator, 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**.
- rajouter ce fichier dans le CMakeLists.txt.
- recompiler le programme dans QtCreator

Les fichiers d'entête

Constat

- Lourd de recopier toutes les déclarations
- Pas de partage des structures

Solution

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

Implémentation

```cpp
// fichier1.cpp

#include "fichier2.h"

A f1(B b){
    ...
    g2(var1);
    ...
}
```

```cpp
// fichier2.h
void g2(C c);

struct Vect{
    ...
};
```

```cpp
// fichier2.cpp

//habitude a prendre
#include "fichier2.h"

void g2(C c){...}
```

Note

Jusqu'à présent on a utilisé des `#include<...>`. C'est pour les headers externes au projet.

En pratique

Méthode générique

Exactement comme pour les .cpp

QtCreator

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

Inclusions mutuelles

Faire un `include` fait un copier/coller du header dans le fichier source.
Rien n'interdit les inclusion mutuelles :

```
// fichier1.h

#include "fichier2.h"

A f1(B b);
```

```
// fichier2.h

#include "fichier1.h"

void g2(C c);
```

Boucle dans les inclusions $\rightarrow$ crash.

Inclusions mutuelles 2

Pour tous les OS :

```
// fichier1.h

#ifndef NOM_UNIQUE
#define NOM_UNIQUE

#include "fichier2.h"

A f1(B b);

#endif
```

Plus récent :

```
// fichier1.h

#pragma once

#include "fichier2.h"

A f1(B b);
```

Plan de la séance

Operateurs

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 créées.

Exemple

```
struct Vect{
   double x,y;
};
```

Ce qu'on voudrait :

```
Vect v1, v2;
...
// additionner deux vecteurs
Vect v3 = v1+v2;
// produit scalaire
double s = v1*v2;
```

```
Vect v1, v2;
...
// additionner deux vecteurs
Vect v3 = {v1.x+v2.x, v1.y+v2.y};
Vect v4;
v4.x = v1.x+v2.x;
v4.y = v1.y+v2.y;
// produit scalaire
double s = v1.x*v2.x + v1.y*v2.y;
```

Implémentation

```
struct Vect{
   double x,y;
};

// operateur +
Vect operator+(Vect vA, Vect vB){
    Vect v = {vA.x+vB.x, vA.y+vB.y};
    return v;
}

// operateur *
double operator*(Vect vA, Vect vB){
    return vA.x*vB.x + vA.y*vB.y;
}
```

```
Vect v1, v2;
...

// additionner deux vecteurs
Vect v3 = v1+v2;

// produit scalaire
double s = v1*v2;
```

Surcharge des opérateurs

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

// operateur * vecteur et reel
Vect operator*(Vect vA, double alpha){
    Vect v = {alpha*v.x, alpha*v.y};
    return v;
}
```

```
Vect v1, v2;
...

// produit scalaire
double s = v1*v2;

// multiplication par un reel
double m = 5.5;
Vect v3 = v1*m;
```

Surcharge des opérateurs 2

Attention

L'ordre des arguments et important :

`v1 * m` est différent de `m * v1`

```
// operateur * vecteur et reel
Vect operator*(Vect vA, double alpha){
    Vect v = {alpha*v.x, alpha*v.y};
    return v;
}

// operateur * vecteur et reel
Vect operator*(double alpha, Vect vA){
    return v*alpha;
}
```

```
Vect v1, v2;
...

// multiplication par un reel
double m = 5.5;

Vect v3 = v1*m;

Vect v4 = m*v2;
```

Plan de la séance

Le TP du jour

On prend le même et on continue :

- Finir le TP de précédent
- Incrémenter en utilisant plusieurs fichiers
- Utiliser des opérateurs pour les calculs