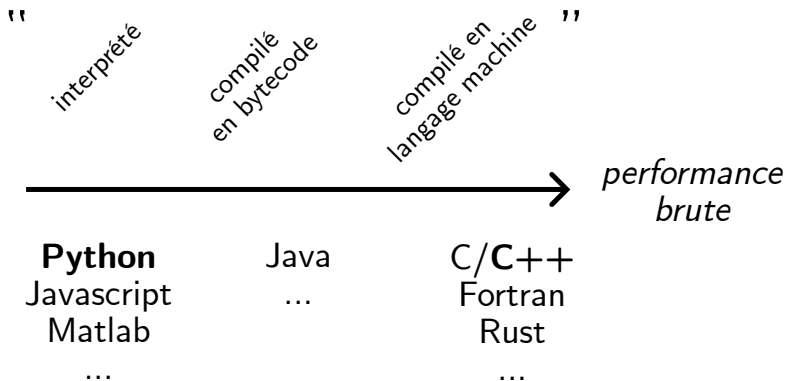


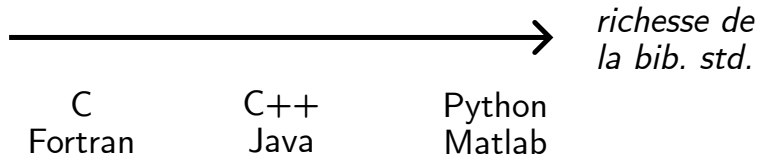
Cours C++

Compilation et directives de préprocesseur

Comparaison de quelques langages



Comparaison de quelques langages



Mécanismes de compilation en C/C++

1. **Précompilation** : consiste à traiter, à *parser* les fichiers sources avant compilation (résolution des directives de préprocesseur *i.e.* remplacement de macros, suppression de texte, inclusion de fichiers...)

```
$ g++ -E fichier_source.cpp
```

On ne le fait que très rarement soi-même, l'appel du compilateur effectue automatiquement la précompilation.

Mécanismes de compilation en C/C++

1. Précompilation
2. Compilation séparée : traduction du code en langage machine.

```
$ g++ -c fichier_source.cpp
```

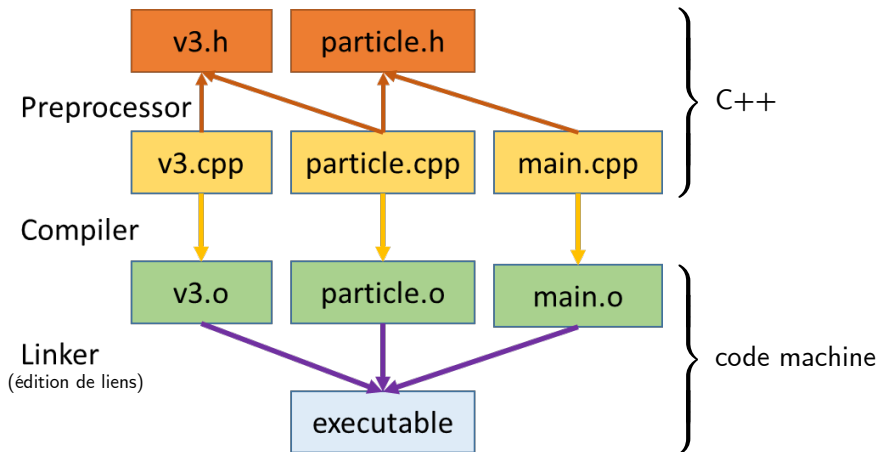
Cette opération conduit à la création d'un **fichier objet** d'extension **.o** (ici **fichier_source.o**).

Mécanismes de compilation en C/C++

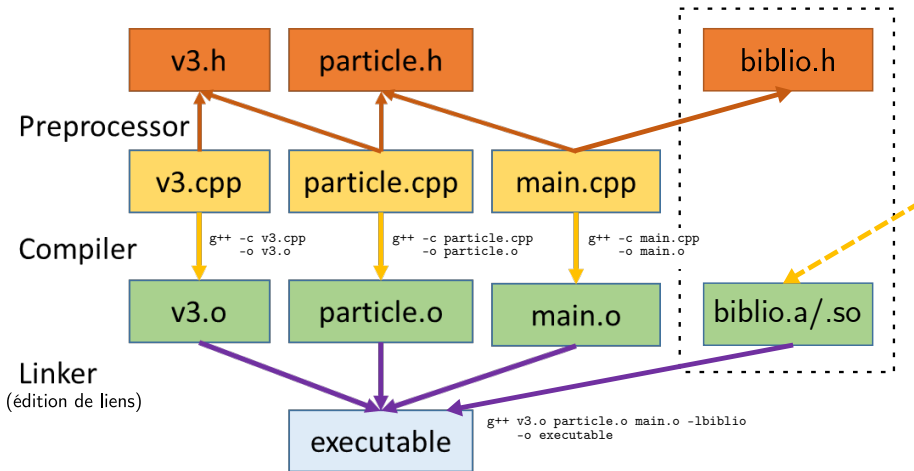
1. Précompilation
2. Compilation séparée
3. **Édition de liens** : consiste à regrouper la totalité des données de même que les fichiers objets et les bibliothèques (fonctions de la bibliothèque standard et des autres bibliothèques externes) ainsi qu'à résoudre les références inter-fichiers.

```
$ g++ fichier_source1.o fichier_source2.o...  
    -o fichier_executable
```

Mécanismes de compilation en C/C++



Mécanismes de compilation en C/C++



Mécanismes de compilation en C/C++

- ▶ Les trois précédentes étapes peuvent être réalisées en une seule opération via la commande

```
$ g++ fichier_source1.cpp fichier_source2.cpp...  
-o fichier_executable
```

☞ Les fichiers d'en-tête ne sont jamais compilés !

- ▶ Le binaire ainsi généré fichier_executable peut alors être exécuté depuis le répertoire local

```
$ ./fichier_executable
```

- ▶ À partir du deuxième TD, on proposera aussi l'utilisation d'un *Makefile*

Options de compilation

- ▶ l'option `-Ichemin` indique au compilateur où sont localisés les fichiers d'en-têtes nécessaires à la compilation séparée du programme. Par défaut, le compilateur recherche ces fichiers localement et dans le répertoire `/usr/include`,
- ▶ l'option `-W` permet l'activation des messages de mise en garde. Différents niveaux sont accessibles, le plus complet étant `-Wall -Wextra`,

Options de compilation

- ▶ l'option `-Ichemin` indique au compilateur où sont localisés les fichiers d'en-têtes nécessaires à la compilation séparée du programme. Par défaut, le compilateur recherche ces fichiers localement et dans le répertoire `/usr/include`,
- ▶ l'option `-W` permet l'activation des messages de mise en garde. Différents niveaux sont accessibles, le plus complet étant `-Wall -Wextra`,

Démystifier la command ccc (cf L3 Physique fondamentale, Orsay)

Directives de préprocesseur

Le préprocesseur recherche des directives de compilation repérées en début de ligne par le symbole # et se terminant avec la fin de la ligne

```
#directive [paramètre]
```

On utilise des directives d'inclusion, de définition et de condition. Par exemple,

```
#include <iostream>
#define PI 3.141592
#if, #ifdef, #ifndef .... #endif
```

Les directives de condition permettent de contrôler ce qui sera compilé effectivement ou non.

Utilisation de directives de condition

Il est possible de contrôler ce qui sera compilé effectivement ou non, avec les clauses de condition :

```
#if condition  
...  
#endif
```

Le code compris dans la séquence `#if -- #endif` est considéré par le compilateur seulement si la condition est vraie (non nulle).

Exemple

► test_debug.cpp

```
#include <iostream>
using namespace std;

int main()
{
    #if (DEBUG == 1)
        cout << "DEBUG: "
              << "Mode debug du programme" << endl;
    #else
        cout << "NOTICE: "
              << "Mode normal du programme" << endl;
    #endif
}
```

Compilation :

```
$ g++ -DDEBUG=1 test_debug.cpp -o test_debug
```

Compilation séparée

► dummy.h

```
#ifndef _DUMMY_H_
#define _DUMMY_H_
void dummy();
#endif
```

► dummy.cpp

```
#include "dummy.h"
#include <iostream>
void dummy() { std::cout << "coucou !" << std::endl; }
```

► test_dummy.cpp

```
#include "dummy.h"
int main() {
    dummy();
}
```

Compilation :

```
$ g++ dummy.cpp test_dummy.cpp -o test_dummy
```


Compilation séparée

► dummy.h

```
#ifndef _DUMMY_H_
#define _DUMMY_H_
void dummy();
#endif
```

► dummy.cpp

```
#include "dummy.h"
#include <iostream>
void dummy() { std::cout << "coucou !" << std::endl; }
```

► test_dummy.cpp

```
#include "dummy.h"
int main() {
    dummy();
}
```

Compilation :

```
$ g++ dummy.cpp test_dummy.cpp -o test_dummy
```

Compilation séparée

► dummy.h

```
#ifndef _DUMMY_H_
#define _DUMMY_H_
void dummy();
#endif
```

► dummy.cpp

```
#include "dummy.h"
#include <iostream>
void dummy() { std::cout << "coucou !" << std::endl; }
```

► test_dummy.cpp

```
#include "dummy.h"
int main() {
    dummy();
}
```

Compilation :

```
$ g++ dummy.cpp test_dummy.cpp -o test_dummy
```

Compilation séparée

► dummy.h

```
#ifndef _DUMMY_H_
#define _DUMMY_H_
void dummy();
#endif
```

► dummy.cpp

```
#include "dummy.h"
#include <iostream>
void dummy() { std::cout << "coucou !" << std::endl; }
```

► test_dummy.cpp

```
#include "dummy.h"
int main() {
    dummy();
}
```

Compilation :

```
$ g++ dummy.cpp test_dummy.cpp -o test_dummy
```

Compilation séparée

► dummy.h

```
#ifndef _DUMMY_H_
#define _DUMMY_H_
void dummy();
#endif
```

► dummy.cpp

```
#include "dummy.h"
#include <iostream>
void dummy() { std::cout << "coucou !" << std::endl; }
```

► test_dummy.cpp

```
#include "dummy.h"
int main() {
    dummy();
}
```

Compilation :

```
$ g++ dummy.cpp test_dummy.cpp -o test_dummy
```

Utilité des fichiers séparés ?

1. **Protection du code source** : un utilisateur λ n'a besoin que de la déclaration de la fonction *i.e.* le fichier `dummy.h` et du fichier objet associé *i.e.* `dummy.o` (comme dans une bibliothèque).
2. **Temps de compilation** : seuls les fichiers sources *i.e.* `*.cpp` modifiés, doivent être recompilés.
3. **Structure/Organisation du code** : à terme chaque structure/classe nouvellement créée se verra associer deux fichiers, sa déclaration et sa définition, ainsi qu'un programme test.

Utilité des fichiers séparés ?

1. **Protection du code source** : un utilisateur λ n'a besoin que de la déclaration de la fonction *i.e.* le fichier `dummy.h` et du fichier objet associé *i.e.* `dummy.o` (comme dans une bibliothèque).
2. **Temps de compilation** : seuls les fichiers sources *i.e.* `*.cpp` modifiés, doivent être recompilés.
3. **Structure/Organisation du code** : à terme chaque structure/classe nouvellement créée se verra associer deux fichiers, sa déclaration et sa définition, ainsi qu'un programme test.

Utilité des fichiers séparés ?

1. **Protection du code source** : un utilisateur λ n'a besoin que de la déclaration de la fonction *i.e.* le fichier `dummy.h` et du fichier objet associé *i.e.* `dummy.o` (comme dans une bibliothèque).
2. **Temps de compilation** : seuls les fichiers sources *i.e.* `*.cpp` modifiés, doivent être recompilés.
3. **Structure/Organisation du code** : à terme chaque structure/classe nouvellement créée se verra associer deux fichiers, sa déclaration et sa définition, ainsi qu'un programme test.