

Le préprocesseur

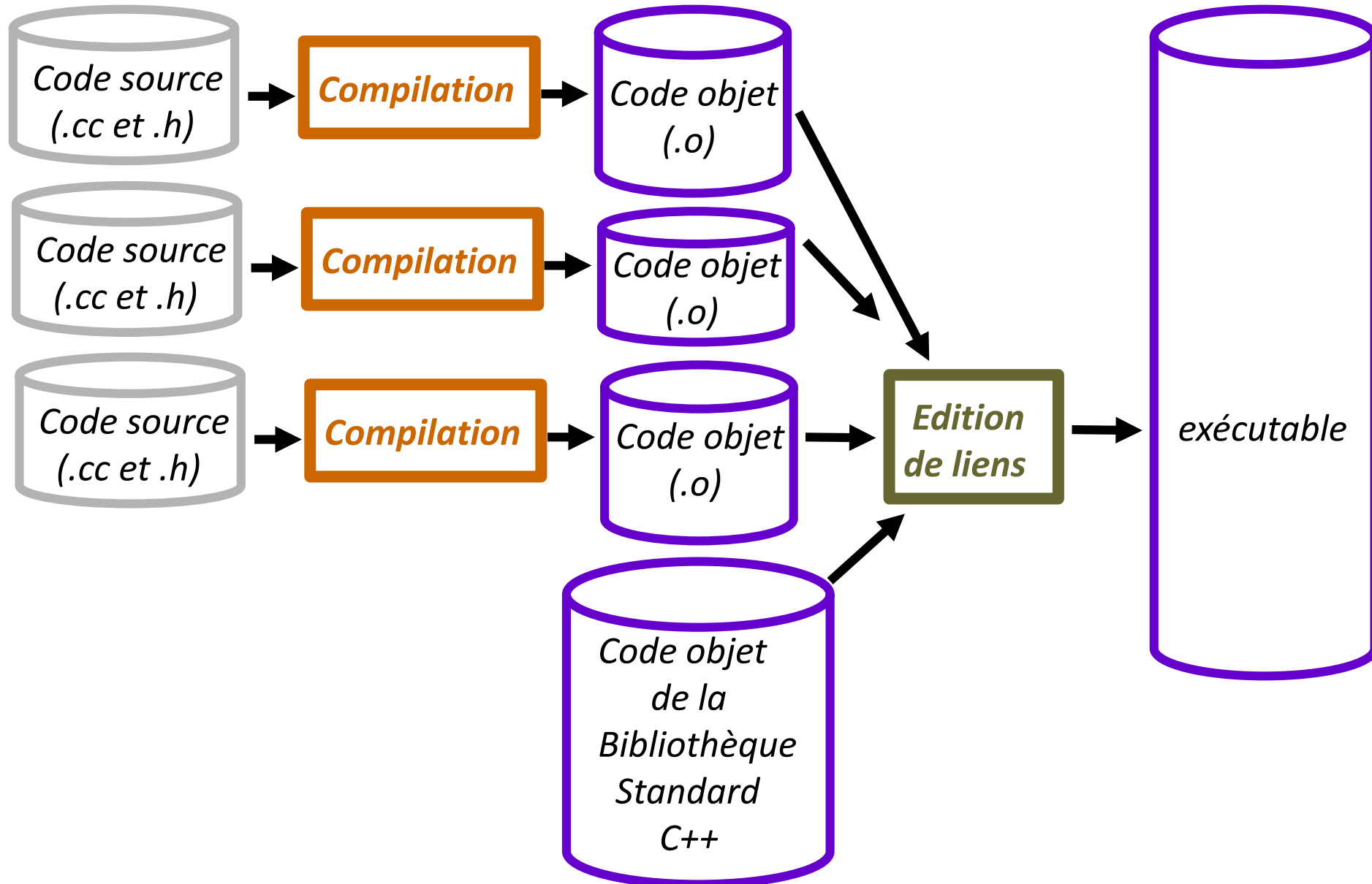
Objectifs:

- Savoir quoi mettre dans l'interface d'un module
- Gérer les constantes en programmation modulaire
- Découvrir quelques outils de mise au point

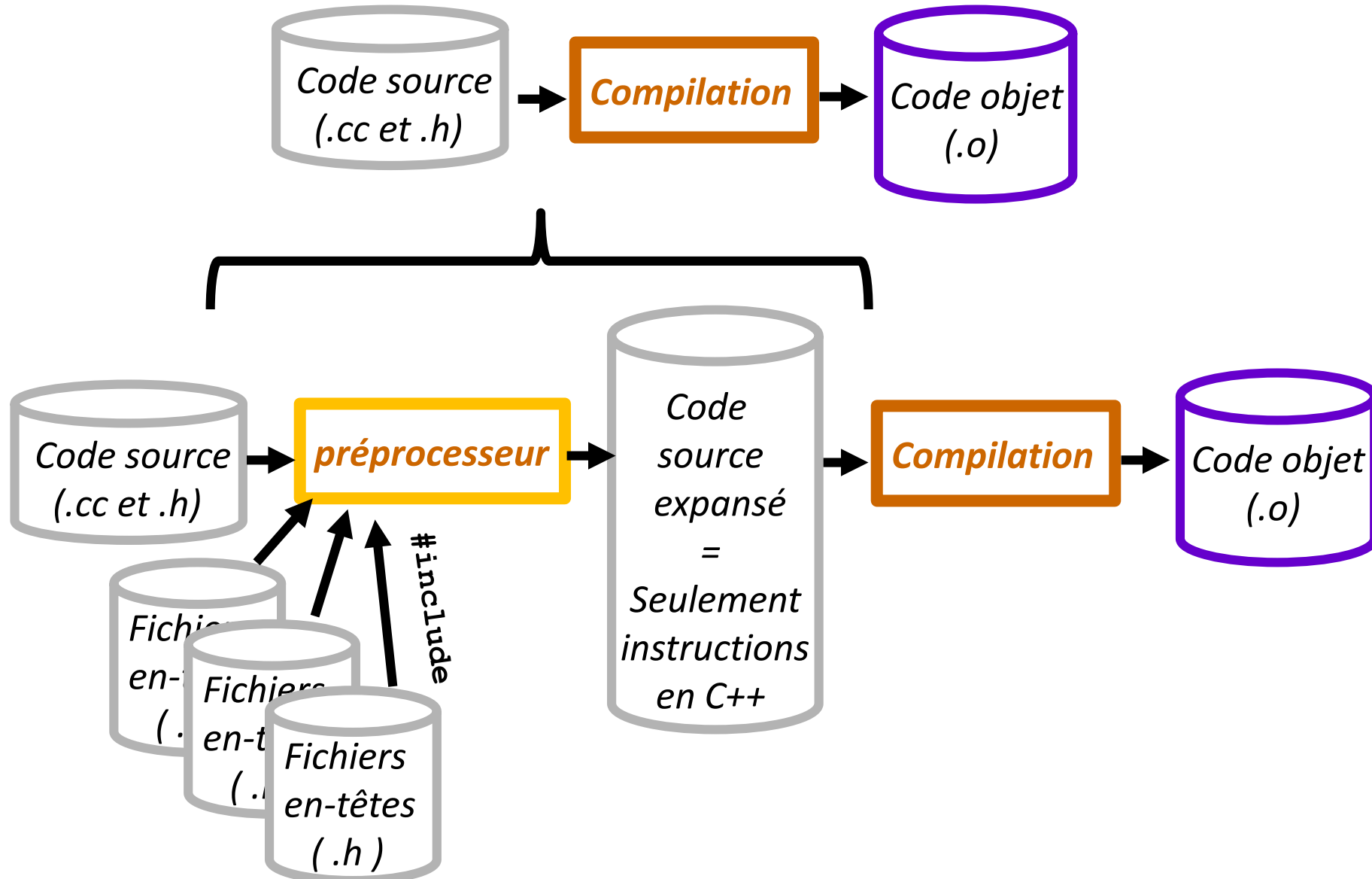
Plan:

- Où intervient le préprocesseur ?
- **include** et fichier en-tête
- Les constantes et l'usage de **define**
- Compilation conditionnelle
- Traiter le problème de l'inclusion multiple

Mais où intervient le préprocesseur ?

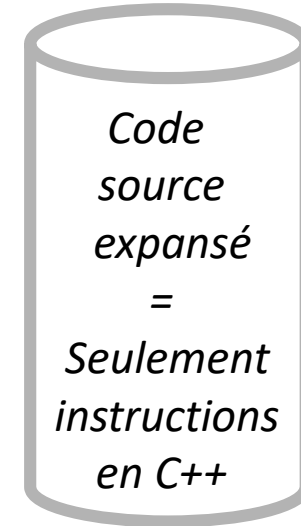


Le préprocesseur prépare la compilation



Vue d'ensemble

Un petit nombre de **# directives** est traduit par le préprocesseur qui produit le code source expansé



Intérêts:

- intègre l'interface des modules avec **#include**
- Empêcher les doubles définitions avec **#ifndef, #endif** et **#define**
- gestion de versions avec **#ifdef, #endif** etc...
- outil de mise au point. Ex: **NDEBUG**

La directive `#include`

Deux types de fichiers en-tête inclus :

- Standard C++ => prédéfini :

```
#include <iostream>
```

- interface d'un module => écrit par l'auteur d'une application. Par défaut, le fichier en-tête doit se trouver dans le même répertoire que le code source

```
#include "mon_fichier.h"
```

Architecture modulaire

*Que faut-il mettre au juste dans l'**interface** d'un module ?*

Réponse: ***le moins possible***

car le principe d'encapsulation recommande de *cacher au maximum l'implémentation d'un module pour réduire les dépendances et les collisions de noms.*

- Mettre le strict nécessaire = ce qui doit être dans l'interface pour utiliser ce qui est exporté: *en particulier les types utilisés dans les prototypes. Ex: string, etc...*
- Comment savoir si un élément DOIT être dans l'interface ?
⇒ *Si le module qui inclut l'interface ne compile pas quand cet élément est absent, c'est donc qu'il était nécessaire*
- Une interface ne doit PAS importer d'espace de nom avec l'instruction *using namespace*

La directive **#define**

Les deux conséquences d'une directive **#define** :

- **toujours**: elle crée un symbole, encore appelée une « *variable du préprocesseur* ». D'autres directives permettent de tester si un symbole existe
- **optionnel**: elle associe un texte au symbole. Ce texte remplacera le symbole partout où le préprocesseur rencontrera le symbole dans le code source (sauf dans "**les chaînes de char**")

```
#define VITESSE_ROBOT          2.5  
#define SYMBOLE_SANS_TEXTE
```

Les directives **#define** ont beaucoup été utilisées pour remplacer les *magic number*
=> Actuellement on utilise plutôt **const**, **constexpr** et **enum** ([slides 11-12, Topic2 sem1](#))

En programmation modulaire, il faut privilégier l'usage de **constexpr** et **enum**.

⇒ **le compilateur** définit toujours toute variable globale déclarée avec **constexpr**
dans l'espace de nom **non-nommé** du module

Définir un symbole à la compilation

Un symbole peut être créé sur la ligne de compilation avec l'option de compilation **-D**

```
g++ -D NDEBUG -std=c++11 -c projet.cc
```

Est équivalent à écrire la directive ci-dessous dans projet.cc

```
#define NDEBUG
```

Conséquences:

- Le symbole **NDEBUG** existe.
- D'autres directives peuvent tester son existence (compilation conditionnelle)

La compilation conditionnelle

Plusieurs directives peuvent **inclure/exclure des morceaux de code** à l'étape de la **compilation** si un symbole est défini ou pas.

Intérêts:

Centraliser plusieurs versions d'un code source dans un même fichier source MAIS l'exécutable contient seulement le code compilé qui correspond à des versions spécialisée telles que :

- » versions "mise au point" / "rendu" / "commerciale"
- » versions dépendant de la plateforme cible
- » versions dépendant d'options marketing

Test d'existence d'un symbole avec les directives **#ifdef** et **#endif**

Le code source entre **#ifdef** et **#endif** est inclus dans le fichier source expansé seulement si le symbole qui suit **#ifdef** existe.

Exemple: le code entre **#ifdef** et **#endif** est compilé car **MYDEBUG** existe:

```
#define MYDEBUG
```

```
...
```

```
#ifdef MYDEBUG
```

```
    // ici du code supplémentaires, par ex
```

```
    // pour l'affichage de messages non
```

```
    // destinés au rendu officiel
```

```
    // voir liste des symboles prédéfinis slide suivant
```

```
    // pour documenter efficacement vos messages
```

```
#endif
```

Outils de mise au point : assert(), NDEBUG et autres symboles

#include <cassert>

Bonne pratique: une **assertion** est une **condition** qui doit toujours être vraie pour le bon fonctionnement du programme. Elle est mise en œuvre avec **assert (condition) ;**

Si elle est fausse, l'exécution s'arrête en indiquant la condition, le fichier et la ligne du code source

A savoir : le symbole **NDEBUG** veut dire «no debug» ; s'il est défini avant **#include <cassert>** alors le préprocesseur désactive les asserts (pour la version finale sans bug)

Autres symboles pour documenter la mise au point :

func

FILE

LINE

TIME

DATE

nom de la fonction où se trouve le symbole
nom du fichier source où se trouve le symbole
numéro de ligne où se trouve le symbole
heure de la dernière compilation du fichier source
date de la dernière compilation du fichier source

Architecture modulaire et pathologie de l'inclusion multiple...



Un projet organisé avec de nombreux modules peut conduire à des cas d'inclusions multiples de certains fichiers en-tête, donc à des définitions multiples.

Problème: la définition multiple d'un symbole ou d'un nom de type ou de fonction conduit à une erreur syntaxique.

Solution: systématiquement mettre en place un *Header guard* pour chaque fichier en-tête de façon à n'autoriser qu'une seule inclusion de son contenu dans le code source à compiler.

Exemple détaillé en Série0

Eviter l'inclusion multiple avec un **header guard**

Le code entre **#ifndef** et **#endif** est inclus dans le code source expansé seulement si le symbole qui suit **#ifndef** n'existe PAS.

Ex: l'interface **nom.h** doit être structurée comme suit:

```
#ifndef NOM_H  
#define NOM_H  
  
    ici le contenu habituel de nom.h  
    avec les prototypes des fonctions,  
    etc...  
  
#endif
```

Danger de collision de nom! Le symbole utilisé comme «garde» doit être unique sur l'ensemble du programme => MAJUSCULES

Résumé

- *Le préprocesseur prepare le code source expansé pour la compilation.*
- *le **principe d'encapsulation** recommande de montrer seulement le strict nécessaire dans l'interface d'un module*
- *La directive **define** permet de créer des symboles dont l'existence peut être testée avec d'autres directives*
- *les directives permettent de gérer plusieurs versions du code dans un même fichier à l'aide de la **compilation conditionnelle***
- *assert() est un outil de mise au point recommandé (version debug) ; il est désactivé si NDEBUG est défini avant l'include <cassert>.*
- *Pour **éviter les double définitions**, mettre en oeuvre un header guard pour tous les fichiers en-tête.*