


chapitre 2.- Modularité et Organisation du code



Ressource R1.01 : Initiation au développement - Partie 2

Bibliographie : cours DUT M1102 de même nom, de P.Etcheverry (source principale)
cours DUT AP2 de même nom, de P.Dagorret (source complémentaire)

Institut Universitaire de Technologie de Bayonne – Pays Basque
BUT Informatique – Semestre 1 - P. Dagorret

Plan

1.	<u>Intérêt des modules</u>	3
2.	<u>Notions de Module et de Modularité</u>	4
3.	<u>Utiliser un module C++</u>	6
	– Directive d'inclusion #include	
	– Utiliser un module dans un programme / module	
	– 3 possibilités d'inclusion d'un module	
4.	<u>Création d'un module C++</u>	11
	– <u>Interface du module</u>	12
	• Contenu	
	• Exemple	
	• Mécanisme des gardes d'inclusion	
	– <u>Corps du module</u>	17
	• Contenu	
	• Exemple	
5.	<u>Notion de Portée</u>	19
	– <u>Contexte</u>	19
	– <u>Définition</u>	20
	– <u>Règles de Portée</u>	21
	<u>Annexe 1</u> : Mécanisme des gardes d'inclusion	31
	Problème lors de la double compilation d'un fichier	



1.- Intérêt des Modules

- ❑ Un programme peut être constitué de plusieurs millions de lignes de code*, et de centaines de sous-programmes
- ❑ Dès qu'un programme comporte plus d'une dizaine de sous-programmes, il devient essentiel d'en organiser le code pour pouvoir en garder le contrôle
- ❑ L'organisation passe notamment par le regroupement de certains sous-programmes dans un même ensemble, appelé **module**
- ❑ L'application finale sera alors composée de plusieurs parties relativement indépendantes : le programme principal (`main.cpp`) et les modules qu'il utilise

* Windows Vista 2007 : 50 millions de lignes de code

2.- Notions de Module et Modularité

❑ Un module

- est un regroupement de types et/ou de constantes et/ou de sous-programmes et/ou de variables, relevant d'un **même thème** ou participant à un **même objectif**
- forme un 'tout' signifiant, c'est-à-dire qui a du sens : une signification, un objectif identifié
- doit, *dans la mesure du possible*, être suffisamment **complet et réutilisable**, pour être utilisé par des programmes différents dans des contextes de développement différents

❑ Modularité

Concevoir une application de manière **modulaire** consiste à l'organiser en identifiant les thèmes ou critères qui permettront de créer et de regrouper ses sous-programmes en modules

2.- Notions de Module et Modularité

❑ Exemples de modules

- module C++ regroupant toutes les opérations d'entrée/sortie de valeurs de types simples du langage (`iostream`)
- module contenant les opérations mathématiques les plus fréquentes
- module contenant des sous-programmes d'une même famille, par exemple des algorithmes de tri : tri par sélection, tri de la bulle quicksort, ...
- module spécifique à l'utilisation des fractions. Il doit proposer :
 - Un **type**, par exemple : `Fraction`
 - Un ensemble d'opérations dédiées aux fractions : additionner, soustraire, multiplier, comparer,... sans oublier les entrées/sorties

En programmation objet (cf. R2.01), un module sera, par exemple, implémenté sous la forme d'une *classe*

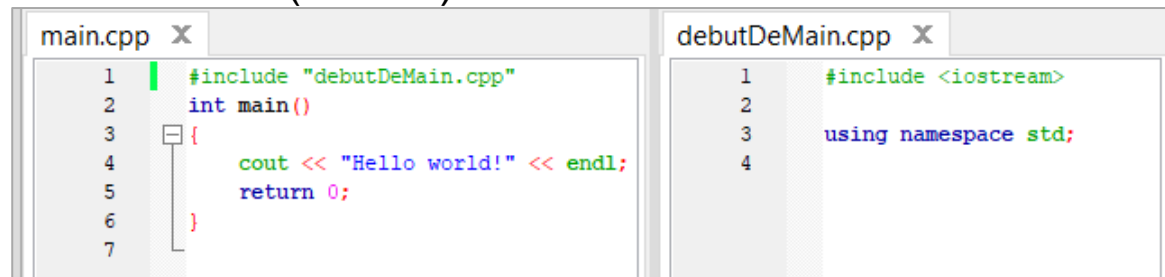
3.- Utiliser un module en C++

❑ Directive `#include nomDeFichier`

– Effet

Elle indique au préprocesseur* d'inclure, avant la compilation, le contenu du fichier `nomDeFichier` à l'endroit où la directive apparaît

– Illustration (triviale)



The screenshot shows two code editors side-by-side. The left editor, titled 'main.cpp', contains the following code:

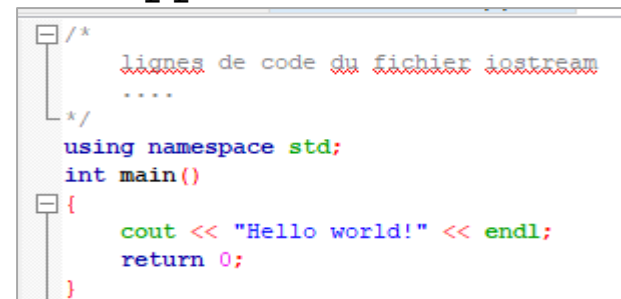
```
1 #include "debutDeMain.cpp"
2 int main()
3 {
4     cout << "Hello world!" << endl;
5     return 0;
6 }
7
```

The right editor, titled 'debutDeMain.cpp', contains the following code:

```
1 #include <iostream>
2
3 using namespace std;
4
```

– Le préprocesseur inclut le contenu du fichier `debutDeMain.cpp` à la ligne 1 du fichier `main.cpp`

– Puis le compilateur compile les lignes de code source dans l'ordre suivant :



The screenshot shows the code after preprocessing. The `#include "debutDeMain.cpp"` line has been replaced by the actual content of `debutDeMain.cpp`, which includes the `using namespace std;` line and the `int main()` function. The code is as follows:

```
/*
   lignes de code du fichier iostream
   ....
*/
using namespace std;
int main()
{
    cout << "Hello world!" << endl;
    return 0;
}
```

* Préprocesseur : Assure une phase préliminaire à la compilation, notamment l'inclusion de segments de code source disponibles dans d'autres fichiers (d'en-tête), ou encore la compilation conditionnelle

3.- Utiliser un module en C++

❑ Utiliser un module dans un programme / module

- Pour qu'un programme (`main`) ou un module `X`, utilise les services d'un module `monModule`, ce programme / module `X` doit **inclure** dans son code source le fichier d'en-tête* de `monModule`
- L'inclusion se fait via la directive de pré-compilation **#include**
`#include <nomFichierDeMonModule>`
`#include "nomFichierDeMonModule"`
- Les symboles `< >` et `" "` indiquent au préprocesseur la *manière* de trouver les fichiers à inclure
`#include <nomFichierDeMonModule>` : cherche le fichier d'abord dans les chemins pré-configurés du compilateur, puis dans le même répertoire que le fichier source incluant le module
`#include "nomFichierDeMonModule"` : cherche le fichier d'abord dans le même répertoire que le fichier source incluant le module, puis dans les chemins pré-configurés du compilateur

* Fichier d'en-tête : fichier contenant les définitions de types et/ou de constantes et/ou de variables et/ou les **déclarations** des sous-programmes du module

3.- Utiliser un module en C++

❑ 3 Possibilités d'inclusion de modules

1.- Inclusion de modules **de la bibliothèque standard de C++**

Le nom du fichier d'en-tête est entouré des symboles < et >

```
#include <iostream>
```

```
#include <cmath>
```

2.- Inclusion de modules **de la bibliothèque standard de C**

Le nom du fichier d'en-tête est entouré des guillemets "

et porte l'extension .h

```
#include "math.h"
```

3.- Inclusion de modules **élaborés par le (ou un autre) programmeur** et dont le code source est disponible

...

3.- Utiliser un module en C++

❑ 3 Possibilités d'inclusion de modules

3.- Inclusion de modules **élaborés par le (ou un autre) programmeur** et dont le code source est disponible

- Le nom du fichier d'en-tête est entouré des guillemets " et porte l'extension `.h`
`#include "game-tools.h"`
- Le préprocesseur cherche le fichier d'abord dans le même répertoire que le fichier source incluant le module, puis dans les chemins pré-configurés du compilateur
- **Mais** le programmeur peut organiser **sa propre arborescence** pour classer les modules qu'il utilise.

Dans ce cas, le nom du fichier du module contient le chemin d'accès au fichier du module :

```
#include "../utilitairesJeux/game-tools.h"
```

3.- Utiliser un module en C++

❑ Récapitulatif des possibilités d'inclusion d'un module

Par exemple dans un programme principal :

```
main.cpp X
1  /** Récapitulatif des modalités d'inclusion d'un module dans une
2      unité de programme (ici, un programme principal) */
3
4      /* Inclusion d'un module de la bibliothèque standard de C++ */
5      #include <iostream>
6
7      /* Inclusion d'un module de la bibliothèque standard de C */
8      #include "math.h"
9
10     /* Inclusion du module permettant l'utilisation de fractions,
11         rangé dans le répertoire courant (cad du répertoire du main.cpp) */
12     #include "fractions.h"
13
14     /* Inclusion du module game-tools, rangé dans un sous-répertoire
15         du répertoire courant (cad du répertoire du main.cpp) */
16     #include "../utilitairesJeux/game-tools.h"
17
18     int main()
19     {
20         ...
21         return 0;
22     }
```

4.- Création d'un module en C++

En C++, un module est implémenté à l'aide de deux fichiers :

□ Un fichier `nomFichierDeMonModule.h`

- Il est appelé **Interface** du module, ou encore **fichier d'en-tête** du module
- Il présente les **services offerts** par le module, mais pas leur implémentation
- Il s'agit du 'mode d'emploi' du module à destination du programmeur souhaitant **utiliser** ce module

□ Un fichier `nomFichierDeMonModule.cpp`

- Il est appelé **corps** du module
- Il contient le code implémentant (mettant en œuvre) les services rendus par le module.

4.- Création d'un module en C++

Interface du module (fichier .h)

L'interface d'un module contient :

- Des **gardes d'inclusion** précisant au compilateur de ne compiler le code du module qu'une seule fois*, même si la directive `#include "nomFichierDeMonModule.h"` apparaît plusieurs fois dans l'application, par exemple dans le programme principal `main.cpp` ainsi que dans d'autres modules de l'application
- Les **directives d'inclusion** des modules nécessaires à l'Interface
- Les définitions des **types** que le module met à disposition du programme ou module qui l'utilise
Par exemple, définition du type `Fraction`
- Les déclarations et initialisations des **constantes** et les déclarations des **variables** que le module met à disposition du programme ou module qui l'utilise. Par exemple, la définition de la constante `FRACTION_NULLE`
- Les déclarations des **sous-programmes** que le module met à disposition du programme ou module qui l'utilise
Par exemple, la déclaration suivante :

```
Fraction reduire(Fraction frac);  
// Retourne la fraction irréductible associée à la fraction  
// passée en paramètre
```

* Afin d'éviter les erreurs `'error: redefinition of ...'`

Illustration problème double compilation : [cf annexe 1](#)

4.- Création d'un module en C++ Interface du module (fichier .h)

❑ Exemple - Interface du module Fractions (fichier fractions.h)

```
fractions.h X
1  #ifndef FRACTIONS_H
2  #define FRACTIONS_H
3
4  // définition du type Fraction mis à disposition
5  struct Fraction
6  {
7      int num;           // numérateur, porte le signe de la fraction
8      unsigned int den ; // dénominateur, > 0
9  };
10
11 // Constante zero
12 const Fraction FRACTIONNULLE = {0, 1};
13
14 /* Déclaration des sous-programmes proposés dans l'interface */
15 void afficher (Fraction frac);
16 // affiche à l'écran le contenu du paramètre fraction frac
17
18 Fraction additionner (Fraction frac1, Fraction frac2);
19 // Retourne la fraction irréductible de frac1 + frac2
20
21 Fraction reduire (Fraction frac);
22 // retourne la fraction irréductible associée au paramètre frac
23 // Opérateurs d'entrée / sortie
24
25
26 ...
27 ...
28
29 #endif // FRACTIONS_H
```

Garde d'inclusion
(Pas de directive d'inclusion)

Définition d'un type

Définition d'une constante

Déclarations
des sous-programmes
proposés par le module

4.- Création d'un module en C++

Remarque : Mécanisme des gardes d'inclusion

❑ Définition

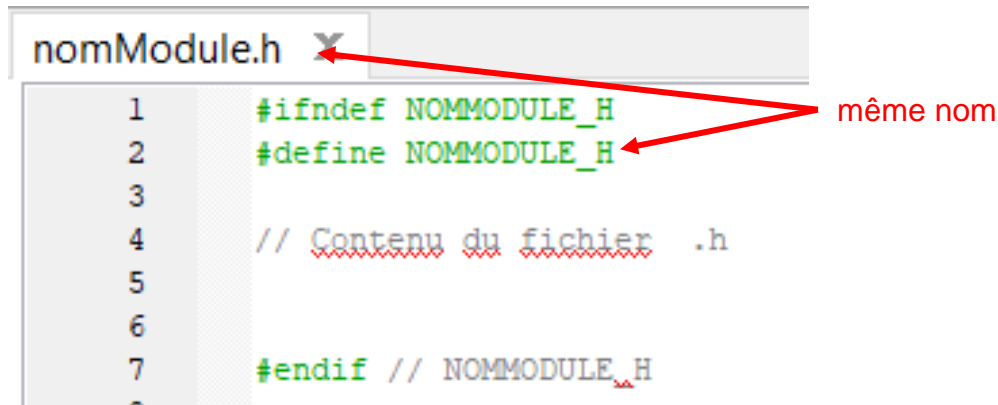
- Les **gardes d'inclusion** sont des directives adressées au préprocesseur
- Elles évitent que le code **d'un fichier d'en-tête** soit compilé plusieurs fois lorsque qu'une même directive `#include` de ce fichier apparaît plusieurs fois, dans plusieurs modules du programme.
Illustration du problème lors d'une double compilation : [cf annexe 1](#)
- Elles commencent par le symbole `#` et ont la structure suivante

```
nomModule.h X
1      #ifndef NOMMODULE_H
2      #define NOMMODULE_H
3
4      // Contenu du fichier .h
5
6
7      #endif // NOMMODULE_H
~
```

4.- Création d'un module en C++

Remarque : Mécanisme des gardes d'inclusion

❑ Conventions de nommage d'une garde d'inclusion*



```
nomModule.h
1  #ifndef NOMMODULE_H
2  #define NOMMODULE_H
3
4  // Contenu du fichier .h
5
6
7  #endif // NOMMODULE_H
```

- Les noms des constantes définies par les gardes d'inclusion portent le même nom que les fichiers .h
- En tant que constantes, elles doivent être écrites en majuscules
- Si le nom du fichier comporte des caractères illégaux pour un nom de constante (par exemple, le . de l'extension), ces caractères sont remplacés par un trait bas (underscore)

* Henricson M., Nyquist E, and Utvecklings E.- Industrial Strength C++ - Prentice Hall PTR 1997

4.- Création d'un module en C++

Remarque : Mécanisme des gardes d'inclusion

❑ Fonctionnement d'une garde d'inclusion

```
nomModule.h X
1  #ifndef NOMMODULE_H
2  #define NOMMODULE_H
3
4  // Contenu du fichier .h
5
6
7  #endif // NOMMODULE_H
```

1. Une garde d'inclusion définit une constante pour le préprocesseur
2. La directive `ifndef` signifie `if not defined`
3. Lors du lancement de la première compilation du fichier d'en-tête, le préprocesseur applique la directive suivante :
Si la constante `nommodule_h` n'est pas définie, `#ifndef nommodule_h`
alors la définir (=la créer) `#define nommodule_h`
Et le contenu du module (. h) sera compilé
4. Si le module est inclus dans d'autres modules, le préprocesseur verra à nouveau la même directive.
Mais cette fois, la constante `nommodule_h` existe ; la condition `ifndef` n'est donc pas valide, et le contenu du module (. h) ne sera pas (re)compilé.

4.- Création d'un module en C++

Corps du module (fichier .cpp)

Le corps d'un module contient :

- Une directive d'inclusion de l'Interface du module :
`#include "nomModule.h"`
- Les directives d'inclusion d'autres modules nécessaires au corps du module
- Les définitions de tous les sous-programmes déclarés dans l'Interface
- Les types, constantes, variables utilisées uniquement dans le corps du module pour la mise en œuvre des sous-programmes de l'interface
- Les déclarations et définitions des sous-programmes utilisés uniquement dans le corps du module pour la mise en œuvre des sous-programmes de l'interface

4.- Création d'un module en C++

Corps du module (fichier .cpp)

❑ Exemple - Corps du module Fractions (fichier fractions.cpp)

fractions.cpp X

```
1  #include "fractions.h"
2
3  #include <iostream>
4  using namespace std;
5
6  const Fraction RACINEDEDEUX = {99, 70}; // à 0,000072 près
7
8  // Corps des sous-programmes proposés par le module
9  void afficher (Fraction frac)
10 {
11     cout << frac.num << "/" << frac.den << endl;
12 }
13
14 Fraction additionner (Fraction frac1, Fraction frac2)
15 // Retourne la fraction irréductible de frac1 + frac2
16 {
17     ...
18 }
```

Garde d'inclusion de l'interface

Directive d'inclusion pour utiliser cout dans la procédure afficher

Définition d'une constante

Définition des sous-programmes déclarés dans l'Interface

5.- Notion de Portée

□ Contexte

La répartition du code sur plusieurs fichiers entraîne immédiatement le besoin de préciser deux informations complémentaires :

- Pour chaque élément ou sous-programme, quelle est la région (ou portion) de code où cet élément / action est accessible et utilisable ?
- Dans chaque région (ou portion) de code, quels sont les éléments / sous-programmes qui sont accessibles et utilisables ?

5.- Notion de Portée

□ Définition

La **portée d'une entité** est la région de code dans laquelle cette entité est utilisable.

La portée d'une entité dépend de l'endroit où elle a été déclarée.

Une **entité** peut être :

- Un type
- Une variable
- Une constante
- Un sous-programme
- Un module
- ...

5.- Notion de portée

Règles de portée concernant les modules

Considérons un programme principal, enregistré dans un fichier `main.cpp`, et utilisant un module dont l'interface et le corps sont enregistrés dans les fichiers `module.h` et `module.cpp`.

Les règles de portée concernant ces unités de programme sont :

1. Une entité déclarée dans `main.cpp` est accessible à partir de la ligne où elle a été déclarée et jusqu'à la fin du bloc où elle a été déclarée.
→ elle est accessible **uniquement** dans `main.cpp`
2. Une entité déclarée dans `module.cpp` est accessible à partir de la ligne où elle a été déclarée et jusqu'à la fin du bloc où elle a été déclarée.
3. Une entité déclarée dans l'Interface `module.h` est accessible :
 - Dans `module.h` à partir de la ligne où elle a été déclarée et jusqu'à la fin du bloc où elle a été déclarée
 - Dans le corps du module
→ dans `module.cpp`
 - Dans tout programme ou module utilisant ce module, à partir de la ligne où est située la directive d'inclusion `#include "module.h"` et jusqu'à la fin du bloc
→ dans `main.cpp`, aux conditions décrites ci-dessus

5.- Notion de portée

Règles de portée concernant les modules

main.cpp X

```
1  #include "fractions.h"
2
3  int main()
4  {
5      Fraction fracOne = {1, 1};
6      Fraction fracSomme ; // somme de fractions, accumulateur
7
8      fracSomme = FRACTIONNULLE;
9      for (unsigned short int i = 1; i <=5; i++)
10     {
11         fracSomme = additionner (fracSomme, fracOne);
12     }
13     afficher (fracSomme);
14     return 0;
15 }
```

fractions.h X

```
1  #ifndef FRACTIONS_H
2  #define FRACTIONS_H
3
4  // définition du type Fraction mis à disposition
5  struct Fraction
6  {
7      int num;           // numérateur, porte le signe de la fraction
8      unsigned int den ; // dénominateur, > 0
9  };
10
11  // Constante zero
12  const Fraction FRACTIONNULLE = {0, 1};
13
14  /* Déclaration des sous-programmes proposés dans l'interface */
15  void afficher (Fraction frac);
16  // affiche à l'écran le contenu du paramètre fraction frac
17
18  Fraction additionner (Fraction frac1, Fraction frac2);
19  // Retourne la fraction irréductible de frac1 + frac2
20
21  ...
22
23  #endif // FRACTIONS_H
```

Illustration :

- Un programme principal **main.cpp**
- Un module Fractions implémenté avec 2 fichiers :
 - **fractions.h** (interface)
 - **fractions.cpp** (corps)

fractions.cpp X

```
1  #include "fractions.h"
2
3  #include <iostream>
4  using namespace std;
5
6  const Fraction RACINEDEDEUX = {99, 70}; // à 0,000072 près
7
8  // Corps des sous-programmes proposés par le module
9  void afficher (Fraction frac)
10 {
11     cout << frac.num << "/" << frac.den << endl;
12 }
13
14 Fraction additionner (Fraction frac1, Fraction frac2)
15 // Retourne la fraction irréductible de frac1 + frac2
16 {
17     ...
18 }
```

5.- Notion de portée

Règles de portée concernant les modules

main.cpp X

```
1  #include "fractions.h"
2
3  int main()
4  {
5      Fraction fracOne = {1, 1};          Portée de fracSomme
6      Fraction fracSomme ; // somme de fractions, accumulateur
7
8      fracSomme = FRACTIONNULLE;
9      for (unsigned short int i = 1; i <=5; i++)
10     {
11         fracSomme = additionner (fracSomme, fracOne);
12     }
13     afficher (fracSomme);
14     return 0;
15 }
```

fractions.h X

```
1  #ifndef FRACTIONS_H
2  #define FRACTIONS_H
3
4  // définition du type Fraction mis à disposition
5  struct Fraction
6  {
7      int num;          // numérateur, porte le signe de la fraction
8      unsigned int den ; // dénominateur, > 0
9  };
10
11  // Constante zero
12  const Fraction FRACTIONNULLE = {0, 1};
13
14  /* Déclaration des sous-programmes proposés dans l'interface */
15  void afficher (Fraction frac);
16  // affiche à l'écran le contenu du paramètre fraction frac
17
18  Fraction additionner (Fraction frac1, Fraction frac2);
19  // Retourne la fraction irréductible de frac1 + frac2
20
21  ...
22
23  #endif // FRACTIONS_H
```

Illustration : Règle 1

Portée de **fracSomme** :

- Dans main.cpp
- A partir de la ligne où elle a été déclarée
- Jusqu'à la fin du bloc (main.cpp) où elle a été déclarée

fractions.cpp X

```
1  #include "fractions.h"
2
3  #include <iostream>
4  using namespace std;
5
6  const Fraction RACINEDEDEUX = {99, 70}; // à 0,000072 près
7
8  // Corps des sous-programmes proposés par le module
9  void afficher (Fraction frac)
10 {
11     cout << frac.num << "/" << frac.den << endl;
12 }
13
14 Fraction additionner (Fraction frac1, Fraction frac2)
15 // Retourne la fraction irréductible de frac1 + frac2
16 {
17     ...
18 }
```

5.- Notion de portée

Règles de portée concernant les modules

main.cpp X

```
1  #include "fractions.h"
2
3  int main()
4  {
5      Fraction fracOne = {1, 1};
6      Fraction fracSomme ; // somme de fractions, accumulateur
7
8      fracSomme = FRACTIONNULLE;
9      for (unsigned short int i = 1; i <=5; i++)
10     {
11         fracSomme = additionner (fracSomme, fracOne);
12     }
13     afficher (fracSomme);
14     return 0;
15 }
```

Portée de i

Illustration : Règle 1

Portée de **i** :

- Dans le bloc **for** de `main.cpp`
- A partir de la ligne où elle a été déclarée
- Jusqu'à la fin du bloc **for** où elle a été déclarée

fractions.h X

```
1  #ifndef FRACTIONS_H
2  #define FRACTIONS_H
3
4  // définition du type Fraction mis à disposition
5  struct Fraction
6  {
7      int num; // numérateur, porte le signe de la fraction
8      unsigned int den ; // dénominateur, > 0
9  };
10
11  // Constante zero
12  const Fraction FRACTIONNULLE = {0, 1};
13
14  /* Déclaration des sous-programmes proposés dans l'interface */
15  void afficher (Fraction frac);
16  // affiche à l'écran le contenu du paramètre fraction frac
17
18  Fraction additionner (Fraction frac1, Fraction frac2);
19  // Retourne la fraction irréductible de frac1 + frac2
20
21  ...
22
23  #endif // FRACTIONS_H
```

fractions.cpp X

```
1  #include "fractions.h"
2
3  #include <iostream>
4  using namespace std;
5
6  const Fraction RACINEDEDEUX = {99, 70}; // à 0,000072 près
7
8  // Corps des sous-programmes proposés par le module
9  void afficher (Fraction frac)
10 {
11     cout << frac.num << "/" << frac.den << endl;
12 }
13
14 Fraction additionner (Fraction frac1, Fraction frac2)
15 // Retourne la fraction irréductible de frac1 + frac2
16 {
17     ...
18 }
```


5.- Notion de portée

Règles de portée concernant les modules

Illustration : Règle 2

Portée des fonctionnalités du module **iostream** :

- Dans fractions.cpp
- A partir de la ligne où il a été inclus
- Jusqu'à la fin du bloc fractions.cpp

main.cpp X

```
1  #include "fractions.h"
2
3  int main()
4  {
5      Fraction fracOne = {1, 1};
6      Fraction fracSomme ; // somme de fractions, accumulateur
7
8      fracSomme = FRACTIONNULLE;
9      for (unsigned short int i = 1; i <=5; i++)
10     {
11         fracSomme = additionner (fracSomme, fracOne);
12     }
13     afficher (fracSomme);
14     return 0;
15 }
```

fractions.h X

```
1  #ifndef FRACTIONS_H
2  #define FRACTIONS_H
3
4  // définition du type Fraction mis à disposition
5  struct Fraction
6  {
7      int num;           // numérateur, porte le signe de la fraction
8      unsigned int den ; // dénominateur, > 0
9  };
10
11  // Constante zero
12  const Fraction FRACTIONNULLE = {0, 1};
13
14  /* Déclaration des sous-programmes proposés dans l'interface */
15  void afficher (Fraction frac);
16  // affiche à l'écran le contenu du paramètre fraction frac
17
18  Fraction additionner (Fraction frac1, Fraction frac2);
19  // Retourne la fraction irréductible de frac1 + frac2
20
21  ...
22
23  #endif // FRACTIONS_H
```

fractions.cpp X

```
1  #include "fractions.h"
2
3  #include <iostream>
4  using namespace std;
5
6  const Fraction RACINEDEDEUX = {99, 70}; // à 0,000072 près
7
8  // Corps des sous-programmes proposés par le module
9  void afficher (Fraction frac)
10 {
11     cout << frac.num << "/" << frac.den << endl;
12 }
13
14 Fraction additionner (Fraction frac1, Fraction frac2)
15 // Retourne la fraction irréductible de frac1 + frac2
16 {
17     ...
18 }
```

Portée des
fonctionnalités de iostream

5.- Notion de portée

Règles de portée concernant les modules

Illustration : Règle 2

Portée de **RACINEDEDEUX** :

- Dans fractions.cpp
- A partir de la ligne où elle a été déclarée
- Jusqu'à la fin du bloc fractions.cpp où elle a été définie

main.cpp X

```
1  #include "fractions.h"
2
3  int main()
4  {
5      Fraction fracOne = {1, 1};
6      Fraction fracSomme ; // somme de fractions, accumulateur
7
8      fracSomme = FRACTIONNULLE;
9      for (unsigned short int i = 1; i <=5; i++)
10     {
11         fracSomme = additionner (fracSomme, fracOne);
12     }
13     afficher (fracSomme);
14     return 0;
15 }
```

fractions.h X

```
1  #ifndef FRACTIONS_H
2  #define FRACTIONS_H
3
4  // définition du type Fraction mis à disposition
5  struct Fraction
6  {
7      int num;           // numérateur, porte le signe de la fraction
8      unsigned int den ; // dénominateur, > 0
9  };
10
11  // Constante zero
12  const Fraction FRACTIONNULLE = {0, 1};
13
14  /* Déclaration des sous-programmes proposés dans l'interface */
15  void afficher (Fraction frac);
16  // affiche à l'écran le contenu du paramètre fraction frac
17
18  Fraction additionner (Fraction frac1, Fraction frac2);
19  // Retourne la fraction irréductible de frac1 + frac2
20
21  ...
22
23  #endif // FRACTIONS_H
```

fractions.cpp X

```
1  #include "fractions.h"
2
3  #include <iostream>
4  using namespace std;
5
6  const Fraction RACINEDEDEUX = {99, 70}; // à 0,000072 près
7
8  // Corps des sous-programmes proposés par le module
9  void afficher (Fraction frac)
10 {
11     cout << frac.num << "/" << frac.den << endl;
12 }
13
14 Fraction additionner (Fraction frac1, Fraction frac2)
15 // Retourne la fraction irréductible de frac1 + frac2
16 {
17     ...
18 }
```

5.- Notion de portée

Règles de portée concernant les modules

Illustration : Règle 2

Portée du paramètre **frac** :

- Dans fractions.cpp
- A partir de la ligne où il a été déclaré
- Jusqu'à la fin du bloc où il a été déclaré, à savoir jusqu'à la fin du corps de la procédure afficher

main.cpp X

```
1  #include "fractions.h"
2
3  int main()
4  {
5      Fraction fracOne = {1, 1};
6      Fraction fracSomme ; // somme de fractions, accumulateur
7
8      fracSomme = FRACTIONNULLE;
9      for (unsigned short int i = 1; i <=5; i++)
10     {
11         fracSomme = additionner (fracSomme, fracOne);
12     }
13     afficher (fracSomme);
14     return 0;
15 }
```

fractions.h X

```
1  #ifndef FRACTIONS_H
2  #define FRACTIONS_H
3
4  // définition du type Fraction mis à disposition
5  struct Fraction
6  {
7      int num;           // numérateur, porte le signe de la fraction
8      unsigned int den ; // dénominateur, > 0
9  };
10
11  // Constante zero
12  const Fraction FRACTIONNULLE = {0, 1};
13
14  /* Déclaration des sous-programmes proposés dans l'interface */
15  void afficher (Fraction frac);
16  // affiche à l'écran le contenu du paramètre fraction frac
17
18  Fraction additionner (Fraction frac1, Fraction frac2);
19  // Retourne la fraction irréductible de frac1 + frac2
20
21  ...
22
23  #endif // FRACTIONS_H
```

fractions.cpp X

```
1  #include "fractions.h"
2
3  #include <iostream>
4  using namespace std;
5
6  const Fraction RACINEDEDEUX = {99, 70}; // à 0,000072 près
7
8  // Corps des sous-programmes proposés par le module
9  void afficher (Fraction frac)
10 {
11     cout << frac.num << "/" << frac.den << endl;
12 }
13
14 Fraction additionner (Fraction frac1, Fraction frac2)
15 // Retourne la fraction irréductible de frac1 + frac2
16 {
17     ...
18 }
```

Portée de frac

5.- Notion de portée

Règles de portée concernant les modules

main.cpp X

```
1  #include "fractions.h"
2
3  int main()
4  {
5      Fraction fracOne = {1, 1};
6      Fraction fracSomme ; // somme de fractions, accumulateur
7
8      fracSomme = FRACTIONNULLE;
9      for (unsigned short int i = 1; i <=5; i++)
10     {
11         fracSomme = additionner (fracSomme, fracOne);
12     }
13     afficher (fracSomme);
14     return 0;
15 }
```

fractions.h X

```
1  #ifndef FRACTIONS_H
2  #define FRACTIONS_H
3
4  // définition du type Fraction mis à disposition
5  struct Fraction
6  {
7      int num; // numérateur, porte le signe de la fraction
8      unsigned int den ; // dénominateur, > 0
9  };
10
11  // Constante zero
12  const Fraction FRACTIONNULLE = {0, 1};
13
14  /* Déclaration des sous-programmes proposés dans l'interface */
15  void afficher (Fraction frac);
16  // affiche à l'écran le contenu du paramètre fraction frac
17
18  Fraction additionner (Fraction frac1, Fraction frac2);
19  // Retourne la fraction irréductible de frac1 + frac2
20
21  ...
22
23  #endif // FRACTIONS_H
```

Portée du type Fraction

Illustration : Règle 3

Portée du type **Fraction** :

- Dans fractions.h
- A partir de la ligne où il a été défini
- Jusqu'à la fin du bloc (fractions.h) où il a été défini
- Dans fractions.cpp
- Dans main.cpp

fractions.cpp X

```
1  #include "fractions.h"
2
3  #include <iostream>
4  using namespace std;
5
6  const Fraction RACINEDEDEUX = {99, 70}; // à 0,000072 près
7
8  // Corps des sous-programmes proposés par le module
9  void afficher (Fraction frac)
10 {
11     cout << frac.num << "/" << frac.den << endl;
12 }
13
14 Fraction additionner (Fraction frac1, Fraction frac2)
15 // Retourne la fraction irréductible de frac1 + frac2
16 {
17     ...
18 }
```

5.- Notion de portée

Règles de portée concernant les modules

Illustration : Règle 3

Portée de **FRACTIONNULLE** :

- Dans fractions.h
- A partir de la ligne où elle a été déclarée
- Jusqu'à la fin du bloc fractions.h
- Dans fractions.cpp
- Dans main.cpp

main.cpp X

```
1  #include "fractions.h"
2
3  int main()
4  {
5      Fraction fracOne = {1, 1};
6      Fraction fracSomme ; // somme de fractions, accumulateur
7
8      fracSomme = FRACTIONNULLE;
9      for (unsigned short int i = 1; i <=5; i++)
10     {
11         fracSomme = additionner (fracSomme, fracOne);
12     }
13     afficher (fracSomme);
14     return 0;
15 }
```

fractions.h X

```
1  #ifndef FRACTIONS_H
2  #define FRACTIONS_H
3
4  // définition du type Fraction mis à disposition
5  struct Fraction
6  {
7      int num;           // numérateur, porte le signe de la fraction
8      unsigned int den ; // dénominateur, > 0
9  };
10
11  // Constante zero
12  const Fraction FRACTIONNULLE = {0, 1};
13
14  /* Déclaration des sous-programmes proposés dans l'interface */
15  void afficher (Fraction frac);
16  // affiche à l'écran le contenu du paramètre fraction frac
17
18  Fraction additionner (Fraction frac1, Fraction frac2);
19  // Retourne la fraction irréductible de frac1 + frac2
20
21  ...
22
23  #endif // FRACTIONS_H
```

Portée de FRACTIONNULLE

fractions.cpp X

```
1  #include "fractions.h"
2
3  #include <iostream>
4  using namespace std;
5
6  const Fraction RACINEDEDEUX = {99, 70}; // à 0,000072 près
7
8  // Corps des sous-programmes proposés par le module
9  void afficher (Fraction frac)
10 {
11     cout << frac.num << "/" << frac.den << endl;
12 }
13
14 Fraction additionner (Fraction frac1, Fraction frac2)
15 // Retourne la fraction irréductible de frac1 + frac2
16 {
17     ...
18 }
```

5.- Notion de portée

Règles de portée concernant les modules

Illustration : Règle 3

Portée de la fonction **additionner** :

- Dans fractions.h
- A partir de la ligne où elle a été déclarée
- Jusqu'à la fin du bloc fractions.h où elle a été déclarée
- Dans fractions.cpp
- Dans main.cpp

main.cpp X

```
1  #include "fractions.h"
2
3  int main()
4  {
5      Fraction fracOne = {1, 1};
6      Fraction fracSomme ; // somme de fractions, accumulateur
7
8      fracSomme = FRACTIONNULLE;
9      for (unsigned short int i = 1; i <=5; i++)
10     {
11         fracSomme = additionner (fracSomme, fracOne);
12     }
13     afficher (fracSomme);
14     return 0;
15 }
```

fractions.h X

```
1  #ifndef FRACTIONS_H
2  #define FRACTIONS_H
3
4  // définition du type Fraction mis à disposition
5  struct Fraction
6  {
7      int num;           // numérateur, porte le signe de la fraction
8      unsigned int den ; // dénominateur, > 0
9  };
10
11 // Constante zero
12 const Fraction FRACTIONNULLE = {0, 1};
13
14 /* Déclaration des sous-programmes proposés dans l'interface */
15 void afficher (Fraction frac);
16 // affiche à l'écran le contenu du paramètre fraction frac
17
18 Fraction additionner (Fraction frac1, Fraction frac2);
19 // Retourne la fraction irréductible de frac1 + frac2
20
21 ...
22
23 #endif // FRACTIONS_H
```

Portée de la fonction **additionner**

fractions.cpp X

```
1  #include "fractions.h"
2
3  #include <iostream>
4  using namespace std;
5
6  const Fraction RACINEDEDEUX = {99, 70}; // à 0,000072 près
7
8  // Corps des sous-programmes proposés par le module
9  void afficher (Fraction frac)
10 {
11     cout << frac.num << "/" << frac.den << endl;
12 }
13
14 Fraction additionner (Fraction frac1, Fraction frac2)
15 // Retourne la fraction irréductible de frac1 + frac2
16 {
17     ...
18 }
```


Annexe 1 - Mécanisme des gardes d'inclusion

Problème lors de la double compilation d'un fichier

❑ Illustration - sans garde d'inclusion

- Double compilation du fichier `zero.h`

```
main.cpp X
1 #include <iostream>
2 using namespace std;
3
4 #include "zero.h"
5 #include "unEtSomme.h"
6
7 int main()
8 {
9     cout << ZERO << " + " << UN << " = " << somme ;
10    return 0;
11 }
```

2 #include imbriqués :
main inclut unEtSomme
qui inclut zero

```
zero.h X
1 const unsigned int ZERO = 0;
2
```

```
unEtSomme.h X
1 #ifndef UNETSOMME_H
2 #define UNETSOMME_H
3
4 #include "zero.h"
5
6 const unsigned int UN = 1;
7 int somme = ZERO + UN;
8
9 #endif // UNETSOMME_H
```

- Le préprocesseur compile les instructions de `main.cpp` dans l'ordre suivant :
- Compilation **NOK**

```
/*
   lignes de code du fichier iostream
*/
using namespace std;

const unsigned int ZERO = 0;
const unsigned int ZERO = 0;
const unsigned int UN = 1;
int somme = ZERO + UN;

int main()
{
    cout << ZERO << " + " << UN << " = " << somme << endl;
    return 0;
}
```

de zero.h
de unEtSomme.h

```
Message
=== Build: Debug in testInclude-4 (compiler: GNU GCC Compiler) ===
error: redefinition of 'const unsigned int ZERO'
note: 'const unsigned int ZERO' previously defined here
```

Annexe 1 - Mécanisme des gardes d'inclusion

Problème lors de la double compilation d'un fichier

❑ Illustration - avec garde d'inclusion

- Une seule compilation du fichier `zero.h`
car gardes d'inclusion

```
main.cpp X
1  #include <iostream>
2  using namespace std;
3
4  #include "zero.h"
5  #include "unEtSomme.h"
6
7  int main()
8  {
9      cout << ZERO << " + " << UN << " = " << somme ;
10     return 0;
11 }

zero.h X
1  #ifndef ZERO_H
2  #define ZERO_H
3
4  const unsigned int ZERO = 0;
5
6  #endif // ZERO_H

unEtSomme.h X
1  #ifndef UNETSOMME_H
2  #define UNETSOMME_H
3
4  #include "zero.h"
5
6  const unsigned int UN = 1;
7  int somme = ZERO + UN;
8
9  #endif // UNETSOMME_H
```

- Compilation OK

```
Process terminated with status 0 (0 minute(s), 1 second(s))
0 error(s), 0 warning(s) (0 minute(s), 1 second(s))
```

- Exécution OK

```
"C:\Users\Pantxika\OneDrive - IUT de Bayonne\but\r
```

```
0 + 1 = 1
```

```
Process returned 0 (0x0)   execution time : 6.103 s
Press any key to continue.
```


chapitre 2.- Modularité et Organisation du code

Ressource R1.01 : Initiation au développement - Partie 2

Merci pour votre attention !