

R1.01 : Initiation au développement (partie 2) Feuille TP n° 1

version 2

Modularité

OBJECTIFS PEDAGOGIQUES :

- 1.- Codage d'algorithmes sous forme modulaire : création et utilisation de sous-programmes et de modules, séparation du code dans fichiers de spécification et d'implémentation
- 2.- S'exercer à l'écriture progressive de programmes.
- 3.- Réaliser et consigner le test fonctionnel d'un programme.

OBJECTIF PRATIQUE :

- Implémenter le module Fractions vu dans le TD n°2 à l'aide de 2 fichiers : **fractions.h** et **fractions.cpp**
- Tester individuellement chaque sous-programme à partir d'un programme de test spécifique également à développer

RESSOURCES A VOTRE DISPOSITION POUR REALISER CE TP :

- **workspace.zip**
- **ressourcesTP1.zip** : une archive composée de
 - De deux fichiers **.txt** contenant du code à intégrer dans votre module.
 - **feuilleTests_tp1.xls** : une feuille de tests qui vous permettra de consigner les résultats des tests fonctionnels réalisés sur les sous-programmes développés

PREPARATION AU TRAVAIL

1. Dans votre espace de travail, à côté du répertoire **r101_partie1**, créer un répertoire **r101_partie2** pour accueillir tous les TPs qui seront réalisés dans le cadre des activités de la ressource R1.01-partie2.
2. Sur eLearn, récupérer l'ensemble des ressources associées à ce TP.
3. Décompresser l'archive **workspace.zip**, puis copier **son contenu** dans le dossier **r101_partie2** qui vient d'être créé. Votre dossier **r101_partie2** a maintenant l'allure suivante :

Remarque

Le dossier **tp1** contient déjà tous les fichiers dont vous aurez besoin pour faire ce TP, mais vides.

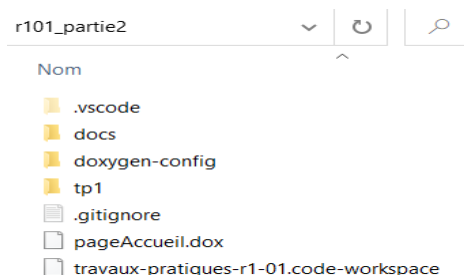


Figure 1 : contenu du workspace pour R1.01 - partie 2

4. Supprimer l'archive .zip et le dossier workspace décompressé
5. Dézipper l'archive **ressourcesTP1.zip**
6. Déplacer le fichier **feuilleTests_tp1.xls** dans le dossier **tp1**

DIRECTIVES PROPRES AU TP

La méthodologie proposée vous permet de créer et de mettre au point un module, par étapes, en validant chaque étape.

Création du module Fractions (minimal) et Validation du lien avec le programme principal

Les fichiers `fractions.h` et `fractions.cpp` sont déjà dans le workspace, mais vides et donc à compléter.

7. Compléter l'Interface du module avec

- Les gardes d'inclusion
- La définition du type `Fraction`
- La déclaration de la constante `FRACTIONNULLE`

8. Compléter le corps du module avec la directive `#include` nécessaire pour le relier à son Interface

9. Compléter le corps du `main.cpp` avec :

- La directive `#include` appropriée
- La déclaration suivante :
`Fraction maFraction = FRACTIONNULLE ;`

10. Compiler

Si la compilation est OK, cela veut dire que votre application, composée d'un programme principal et d'un module Fractions, est sans erreur (notamment, le contenu du module est bien utilisable par le programme) et prêt à être complété.

Si ce n'est pas le cas, corriger la/les erreur(s) avant de passer à l'étape suivante.

Conclusion :

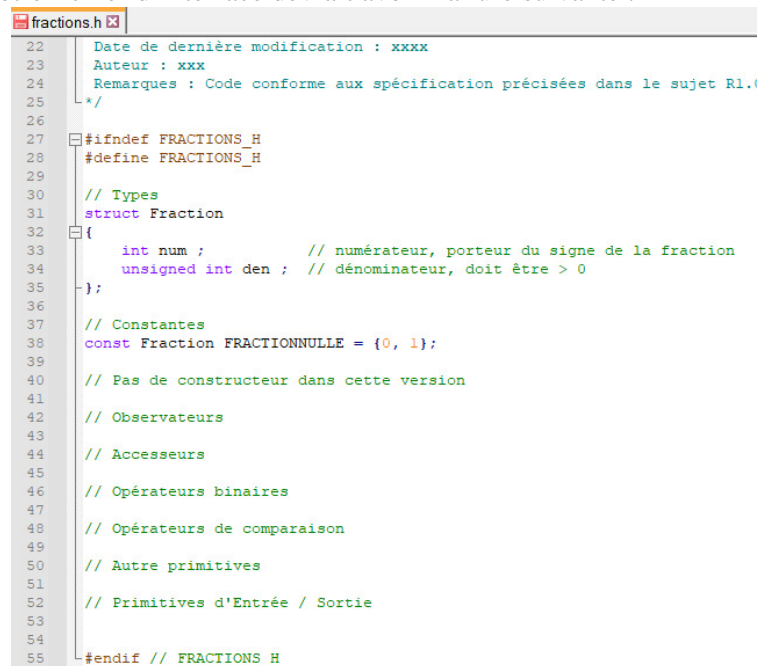
Cette première étape est fondamentale. Elle garantit que le 'cadre' de travail (développement) est stable. On peut donc s'appuyer dessus pour continuer le développement.

Documenter votre module

11. Documenter l'interface

Compléter le fichier `.h` avec les rubriques composant habituellement l'interface d'un module.

Après cette étape, votre fichier d'interface devrait avoir l'allure suivante :



```
22  Date de dernière modification : xxxx
23  Auteur : xxx
24  Remarques : Code conforme aux spécifications précisées dans le sujet R1.
25  */
26
27  #ifndef FRACTIONS_H
28  #define FRACTIONS_H
29
30  // Types
31  struct Fraction
32  {
33      int num ;           // numérateur, porteur du signe de la fraction
34      unsigned int den ; // dénominateur, doit être > 0
35  };
36
37  // Constantes
38  const Fraction FRACTIONNULLE = {0, 1};
39
40  // Pas de constructeur dans cette version
41
42  // Observateurs
43
44  // Accesseurs
45
46  // Opérateurs binaires
47
48  // Opérateurs de comparaison
49
50  // Autres primitives
51
52  // Primitives d'Entrée / Sortie
53
54
55  #endif // FRACTIONS_H
```

Figure 2 : Documentation de `fraction.h`

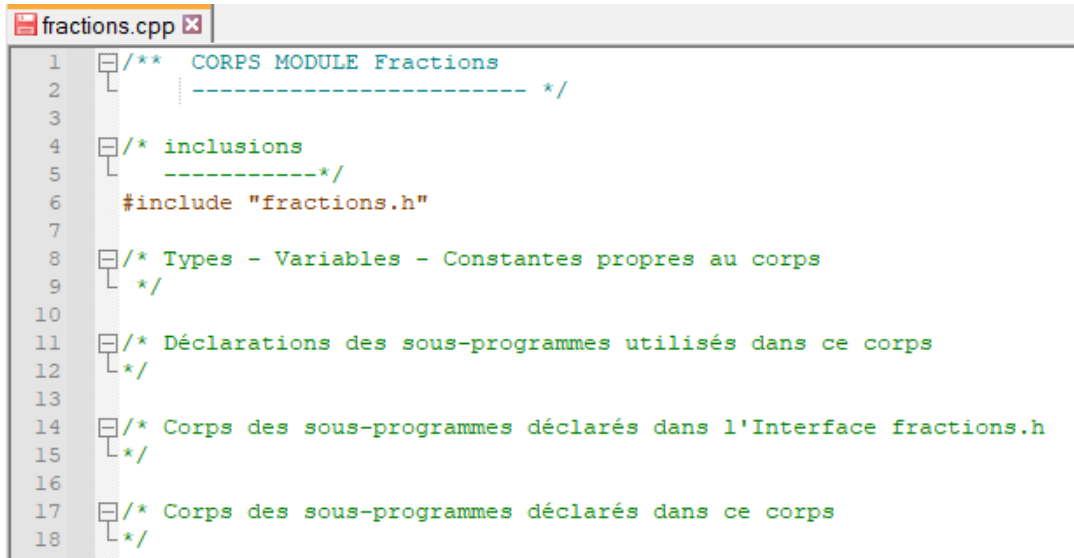
12. Compiler pour éliminer les éventuelles coquilles.

13. Documenter le corps.

Compléter le fichier .cpp avec les rubriques composant habituellement le corps d'un module

- 1 : Rubrique des types, constantes et variables **propres au corps du module**
- 2 : Rubrique des déclarations des sous-programmes utilisés **dans le corps du module**
- 3 : Rubrique des corps des sous-programmes déclarés **dans l'Interface du module**
- 4 : Rubrique des corps des sous-programmes déclarés dans le corps du module (dans rubrique 2)

Après cette étape, votre fichier .cpp devrait avoir l'allure suivante :



```
1  /** CORPS MODULE Fractions
2      ----- */
3
4  /** inclusions
5      -----*/
6      #include "fractions.h"
7
8  /** Types - Variables - Constantes propres au corps
9      */
10
11 /** Déclarations des sous-programmes utilisés dans ce corps
12     */
13
14 /** Corps des sous-programmes déclarés dans l'Interface fractions.h
15     */
16
17 /** Corps des sous-programmes déclarés dans ce corps
18     */
```

Figure 3 : Documentation de fractions.cpp

Ajouter une opération d'entrée-sortie : afficher()

14. Copier les codes qui vous ont été fournis dans le fichier de ressources `codeAfficherAIntegrer.txt` décompressé et coller chaque portion dans le fichier (`fractions.h`, `fractions.cpp`) adéquat.
15. Compiler. Corriger corriger la/les erreur(s) avant de passer à l'étape suivante.

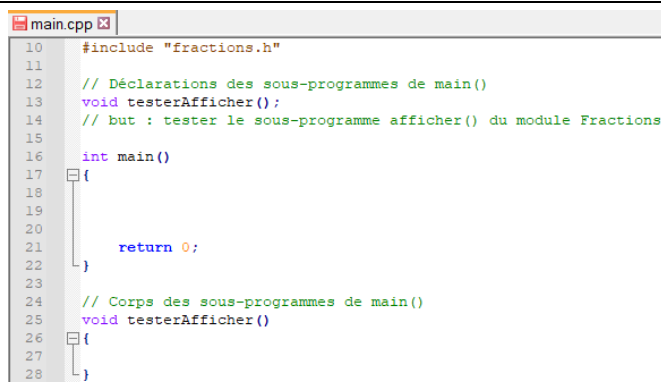
Tester l'opération d'entrée-sortie : afficher()

16. Créer un sous-programme de test de cette opération.

Dans le fichier main.cpp, créer un sous-programme (procédure) nommé `testAfficher()`, sans paramètres, dont le but est de tester la procédure `afficher()` qui vient d'être ajoutée au module.

Pour ce faire :

- Écrire sa déclaration
- Créer un corps minimal.



```
10 #include "fractions.h"
11
12 // Déclarations des sous-programmes de main()
13 void testerAfficher();
14 // but : tester le sous-programme afficher() du module Fractions
15
16 int main()
17 {
18
19
20
21     return 0;
22 }
23
24 // Corps des sous-programmes de main()
25 void testerAfficher()
26 {
27
28 }
```

Figure 4 : Création du sous-programme testerAfficher() dans le fichier main.cpp

17. Compléter le corps du sous-programme de test avec le jeu d'essai prévu.

Il est prévu de tester l'affichage :

- D'une fraction positive, exemple : 1/3
- D'une fraction négative, exemple -1/3
- De la fraction nulle : 0/1 ou **FRACTIONNULLE**

Vous pouvez aussi tester l'affichage d'un nombre entier, exemple 1/1.

Le corps du sous-programme de test sera alors :

```
29 void testerAfficher()
30 {
31     afficher({1, 3}) ; cout << endl ;
32     afficher({-1, 3}) ; cout << endl ;
33     afficher({0, 1}) ; cout << endl ;
34     afficher(FRACTIONNULLE) ; cout << endl ;
35     afficher ({1,1}) ; cout << endl ;
36 }
```

Figure 5 : Corps du sous-programme testerAfficher()

Si des éléments (par exemple des variables) étaient nécessaires pour écrire le sous-programme de tests, ils seraient déclarés dans le sous-programme de test.

18. Compléter le programme principal **main()** avec l'appel du sous-programme de **testerAfficher()** .

Le corps de **main()** sera alors :

```
14 // Déclarations des sous-programmes de main()
15 void testerAfficher();
16 // but : tester le sous-programme afficher() du module Fractions
17
18 int main()
19 {
20
21
22
23     testerAfficher();
24
25     return 0;
26 }
27
28 // Corps des sous-programmes de main()
29 void testerAfficher()
30 {
31     afficher({1, 3}) ; cout << endl ;
32     afficher({-1, 3}) ; cout << endl ;
33     afficher({0, 1}) ; cout << endl ;
34     afficher(FRACTIONNULLE) ; cout << endl ;
35     afficher ({1,1}) ; cout << endl ;
36 }
```

Figure 6 : Fichier main.cpp lors du test de afficher()

19. Compiler et corriger la/les erreur(s)

20. **Passer le test.** Cela correspond à :

- exécuter le programme **main()**,
- noter les valeurs obtenues sur le fichier de test excel fourni,
- et comparer les **valeurs obtenues** (fournies par l'exécution du programme) avec les **valeurs attendues** (celles figurant dans la colonne 'valeurs attendues' du fichier excel de tests).

Si les résultats obtenus sont conformes aux valeurs attendues, le développement et test du sous-programme **afficher()** est terminé.

Sinon, il faut modifier le sous-programme **afficher()** pour qu'il soit conformes aux attentes.

Ajouter une opération d'entrée-sortie : `saisir()`

21. Copier les codes qui vous ont été fournis dans le fichier ressource `codeSaisirAIntegrer.txt` décompressé et coller chaque portion dans le fichier (`fractions.h` - `fractions.cpp`) adéquat.
22. Compiler. Corriger la/les erreur(s) avant de passer à l'étape suivante.
23. Analyser le sous-programme ajouté.
Le code du sous-programme `saisir()` n'est pas complet (pour l'instant), car son algorithme fait appel à un sous-programme, `reduire()`, que vous n'avez pas encore développé. Vous aurez cette ressource lorsque vous aurez fait l'algorithme et développé le sous-programme `reduire()`.
Mais cela ne nous empêche pas d'ajouter le sous-programme `saisir()` et de compiler le module sans erreur.
 - Pourquoi ?
 - Quelles sont les instructions composant le corps minimal d'une fonction ?
 - Quelles sont les instructions composant le corps minimal d'une procédure ?

Conclusion : Coder et tester les autres sous-programmes du module Fractions

Vous avez maintenant la méthode pour continuer à compléter le module Fractions.

L'ajout se fera sous-programme par sous-programme, selon les étapes vues précédemment :

- Ajouter dans la déclaration du sous-programme (dans l'Interface du module)
- Ajouter un corps minimal (dans le corps du module)
- Compiler
- Compléter le corps minimal
- Tester le sous-programme (en l'appelant dans le `main.cpp`)

Vous coderez et testerez des sous-programmes de chacune des familles suivantes :

- Accesseurs, 2 : `numérateur()` et `denominateur()` facultatif mais recommandé
- Opérateurs binaires : 2 : `additionner()` et `diviser()`
- Opérateurs de comparaison, 1 : `estEgal()`
- Autres primitives, 1 : `reduire()`

Rappels

Avant de coder :

- Faire un algorithme
- Préparer le jeu d'essai servant à écrire le sous-programme de test
 - o Ajouter, dans le fichier de tests excel fourni, un onglet spécifique au sous-programme
 - o Créer le tableau avec les valeurs qui seront fournies au sous-programme testé, et les valeurs attendues

Lors du codage : appliquer toutes les recommandations vues dans la première partie de R1.01-Initiation au développement (partie 1).