LO02

Principes et pratique de la Programmation Orientée Objet

Exercices corrigés

Antoine Jouglet antoine.jouglet@utc.fr

Avant-propos

Ce document est composé d'exercices et de leur correction. Ces exercices sont destinés à vous aider dans la compréhension des notions vues en cours.
Le document est découpé en chapitres thématiques. Le dernier chapitre regroupe des problèmes issus de sujets d'examen. Chaque chapitre contient les énoncés de plusieurs exercices. Une correction détaillée de ces exercices, avec souvent quelques rappels de points de cours, est fournie à la fin de chaque chapitre.

Remarque

Il est très probable qu'il subsiste des fautes d'orthographe et des erreurs de correction. Il existe aussi plusieurs notions du cours qui ne sont pas encore vues dans ce recueil mais qui mériteraient un exercice d'illustration.
Vous pouvez aider à améliorer ce document en détectant les erreurs et les envoyant à **`antoine.jouglet@utc.fr`**. Toute proposition de thème d'exercice est aussi la bienvenue.
Merci pour votre aide !

Table des matières

Éléments de base en C++

Exercice 1 - C++ : programme le plus court

Écrire le programme le plus court en C++.

Exercice 2 - Possibilités d'E/S du C++

Écrire un programme qui demande à l'utilisateur son prénom et qui lui dit "bonjour" en réindiquant son prénom.

Exercice 3 - Définition - Initialisation - Affectation

Définir une variable `x` de type **`double`** en l'initialisant avec la valeur `3.14` et afficher sa valeur. Définir une variable `y` de type **`double`** en l'affectant avec la valeur `3.14` et afficher sa valeur avant et après affectation.

Exercice 4 - Objets constants

Définir une constante `pi` de type **`double`**, lui donner la valeur `3.14`, et afficher sa valeur.

Exercice 5 - Adressage indirect

Définir une variable `x` de type **`double`** en l'initialisant avec la valeur `3.14` et afficher sa valeur. Définir une variable `pt` de type pointeur de **`double`** qui sera initialisée avec l'adresse de la variable `x`. Modifier la valeur de `x` à partir du pointeur `pt`. Afficher la nouvelle valeur de `x`, une fois à partir de l'identificateur `x`, une fois à partir du pointeur `pt`. Afficher l'adresse de `x` en utilisant le pointeur `pt`. Définir une variable référence `ref` qui sera initialisée avec l'adresse de la variable `x`. Modifier la valeur de `x` à partir de la référence `ref`. Afficher la nouvelle valeur de `x`, une fois à partir de l'identificateur `x`, une fois à partir de la référence `ref`. Afficher l'adresse de `x` en utilisant la référence `ref`.

Exercice 6 - Pointeurs - Pointeurs `const`

Parmi les instructions suivantes, indiquer celles qui sont valides et celles qui sont ne sont pas valides en expliquant pourquoi.

```
1  double* pt1=4096;
2  double* pt2=(double*)4096;
3  void* pt3=pt1;
4  pt1=pt3;
5  pt1=(double*)pt3;
6  double d1=36;
7  const double d2=36;
8  double* pt4=&d1;
9  const double* pt5=&d1;
10 *pt4=2.1;
11 *pt5=2.1;
12 pt4=&d2;
13 pt5=&d2;
14 double* const pt6=&d1;
15 pt6=&d1;
16 *pt6=2.78;
17 double* const pt6b=&d2;
18 const double* const pt7=&d1;
19 pt7=&d1;
```

```
20 *pt7=2.78;
21 double const* pt8=&d1;
22 pt8=&d2;
23 pt8=&d1;
24 *pt8=3.14;
```

dans une unité de compilation .cpp

Exercice 7 - Référence - Références `const`

Parmi les instructions suivantes, indiquer celles qui sont ne sont pas valides et expliquer pourquoi.

```
1  double& d1=36;
2  double d2=36;
3  double& ref=d2;
4  ref=4;
5  const double d3=36;
6  const double& d4=36;
7  const double& d5=d2;
8  d5=21;
9  const double& d6=d3;
10 double& ref2=d6;
11 int i=4;
12 double& d7=i;
13 const double& d8=i;
14 d8=3;
```

dans une unité de compilation .cpp

Exercice 8 - Allocation dynamique d'une variable

Allouer dynamiquement un entier, stocker son adresse dans un pointeur et affecter cet entier avec la valeur 2. Allouer dynamiquement un entier en l'initialisant avec la valeur 4 et stocker son adresse dans un pointeur. Désallouer les 2 entiers alloués dynamiquement.

Exercice 9 - Allocation dynamique d'un tableau de variables

Allouer dynamiquement un tableau de 20 **`double`** et stocker son adresse dans un pointeur. Affecter chaque cellule du tableau avec la valeur 3.14. Désallouer ce tableau.

Exercice 10 - Fonctions et passage de paramètre par valeur, par adresse et par référence

Soit la structure

```
1 struct Date {
2  unsigned int jour, mois, annee;
3 };
```

date.h

Ecrire une fonction qui modifie le jour, le mois ou l'année d'une structure `Date` donnée. Il y a trois solutions à ce problème (passage par valeur, par adresse et par référence). Les mettre en œuvre et expliquer les différences.

Exercice 11 - Structure et adressage indirect

Ecrire l'entête d'une fonction qui reçoit une structure de grande taille en mémoire sachant que l'on ne veut ni perdre de place mémoire, ni offrir à la fonction la possibilité d'en modifier la valeur.

Exercice 12 - Retour de valeur par référence

Dans une fonction, que se passe-t-il lorsque l'on retourne une variable locale et que la fonction a pour type de retour une référence ? Que faudrait-il faire pour solutionner ce problème ? Donner un exemple.

```
1 objet& fonction(){
2  objet the_object_to_return; // variable locale
3  // ...
4  return the_object_to_return;
5 }
```

dans une unité de compilation .cpp

Exercice 13 - Références et initialisation

Soit les fonctions `f`, `g`, `h` de prototypes :

- `void f(string);`
- `void g(string&);`
- `void h(const string&);`

Parmi les instructions suivantes, distinguer celles qui sont invalides. Expliquer.

```
1  string s1="bonjour";
2  f(s1);
3  g(s1);
4  h(s1);
5  const string s2="salut";
6  f(s2);
7  g(s2);
8  h(s2);
9  const string& ref1=s2;
10 string& ref2=s2;
11 string s3=s2;
12 f("coucou");
13 g("coucou");
14 h("coucou");
```

dans une unité de compilation .cpp

Exercice 14 - Fonctions et arguments par défaut

Soit la fonction `somme` suivante qui permet de trouver la somme de 5 nombres entiers passés en paramètres.

```
1 int somme(int a, int b, int c, int d, int e);
```

exemple.h

```
1 int somme(int a, int b, int c, int d, int e){
2  return a+b+c+d+e;
3 }
```

exemple.cpp

Ajouter des arguments par défaut à cette fonction de façon à ce que les sommes de 2, 3 ou 4 nombres passés en argument soient aussi permises.

Exercice 15 - Fonctions et arguments par défaut

Soit la fonction `minimum` suivante qui permet de trouver le minimum de 5 nombres entiers passés en paramètres.

```
1 unsigned int minimum(unsigned int a, unsigned int b, unsigned int c, unsigned
     int d, unsigned int e);
```

exemple.h

```
1 unsigned int minimum(unsigned int a, unsigned int b, unsigned int c, unsigned
     int d, unsigned int e){
2  unsigned int min=a;
3  if (b<min) min=b;
4  if (c<min) min=c;
5  if (d<min) min=d;
6  if (e<min) min=e;
7 }
```

exemple.cpp

Ajouter des arguments par défaut de façon à ce que cette fonction permette aussi la recherche du minimum de 2, 3 ou 4 nombres passés en argument.

Exercice 16 - Surcharge de fonction

Ecrire une fonction `incrementer` qui permet d'ajouter 1 à la valeur d'une variable de type **`int`** passée par référence. Surcharger cette fonction de manière à pouvoir faire le même traitement sur une variable de type **`char`** puis sur une variable de type **`double`**. Surcharger cette fonction de manière à pouvoir incrémenter toutes les valeurs d'un tableau de variables de type **`int`**. Le tableau sera transmis par adresse en même temps qu'une valeur de type **`unsigned int`** (transmise par valeur) indiquant le nombre d'éléments du tableau. Surcharger cette fonction de manière à pouvoir faire le même traitement sur un tableau de variables de type **`char`** puis sur un tableau de variables de type **`double`**.

Exercice 17 - Fonctions `inline`

Reprendre l'Exercice 16 en transformant les trois premières fonctions `incrementer` en fonctions **`inline`**.

Exercice 18 - Espace de noms (`namespace`)

Reprendre l'Exercice 16 en englobant les trois premières fonctions `incrementer` dans le namespace `FX`. Ecrire une fonction `main` utilisant ces fonctions. Utiliser au moins une fois l'opérateur de résolution de portée `::`, une **`using`**-déclaration et une **`using`**-directive.

Solution de l'exercice 1

```
1 int main() {
2  return 0;
3 };
```

Un programme en C++ doit toujours au moins comporter la fonction `main` qui est le point d'entrée du programme. La première instruction de cette fonction sera la première exécutée. La fonction `main` doit normalement obligatoirement renvoyer un entier (à l'aide du mot clé **`return`**). Cette valeur est en général 0 pour indiquer au système qui a lancé l'exécution du programme, qu'il n'y a pas eu d'erreur lors de l'exécution du programme. Toute valeur différente de 0 aurait indiqué un problème.

Solution de l'exercice 2

```
1  #include<iostream>//déclarations des fonctions d'E/S
2  #include<string>
3  using namespace std;
4  int main() {
5   cout<<"Entrez votre prénom :";
6   string prenom;
7   cin>>prenom;
8   cout<<"Bonjour "<<prenom<<"\n";
9   return 0;
10 };
```

dans une unité de compilation .cpp

Pour faire des entrées/sorties avec le C++, il faut utiliser la bibliothèque `iostream`. Pour cela, l'instruction au préprocesseur **`#include`** permet d'inclure le fichier d'entête de la bibliothèque concernée avant la compilation. Ce fichier d'entête contient les déclarations de tous les éléments de cette bibliothèque. Ces déclarations sont nécessaires pour compiler l'unité de compilation contenant le programme.
Toutes ces déclarations étant placées dans l'espace de nom `std`, il faut utiliser l'instruction **`using namespace`** `std;` afin de les rendre visibles dans la portée globale.
Pour écrire sur la sortie standard (en général un écran), il faut utiliser l'opérateur `<<` sur l'objet `cout` de la classe `ostream` (qui représente la sortie standard) en lui transmettant l'élément à afficher. Cet opérateur (qui est une fonction) renvoie une référence sur l'objet `ostream` auquel il s'est appliqué (ici `cout`). L'opérateur `<<` peut alors être réutilisé directement sur cette référence pour une autre écriture sur le flux (voir ligne 8).
Pour lire sur l'entrée standard (en général un clavier), il faut utiliser l'opérateur `>>` avec en premier paramètre l'objet `cin` de la classe `istream` (qui représente l'entrée standard), et en second paramètre l'objet qui sera affecté par la valeur lue. Cette fonction renvoie une référence sur l'objet `istream` auquel elle s'est appliquée (ici `cin`). L'opérateur `>>` peut alors être réutilisé directement sur cette référence pour une autre lecture sur le flux.

Solution de l'exercice 3

```
1 #include<iostream> using namespace std;
2 int main() {
3  double x=3.14; // c'est une initialisation
4  cout<<"x="<<x<<"\n";
5  double y;
6  cout<<"y="<<y<<"\n";
```

```
 y=3.14; // c'est une affectation
 cout<<"y="<<y<<"\n";
 return 0;
};
```

dans une unité de compilation .cpp

Une *initialisation* de variable signifie qu'une valeur doit être donnée au moment de la définition de la variable. Une *affectation* correspond à l'utilisation de l'opérateur = en dehors d'une initialisation. Si une variable n'est pas initialisée, sa valeur n'est pas connue jusqu'à sa première affectation.
Il y a une différence profonde entre l'affectation et l'initialisation. Cette différence est beaucoup plus visible lorsque des objets sont manipulés. Notamment, lors de l'initialisation d'un objet, c'est un constructeur de la classe de cet objet qui est appelé, alors que c'est la surcharge de l'opérateur = qui est appelé lors d'une affectation.

Solution de l'exercice 4

```
#include<iostream> using namespace std;
int main() {
 const double pi=3.14; // c'est une initialisation
 cout<<"pi="<<pi<<"\n";
 return 0;
};
```

dans une unité de compilation .cpp

L'utilisation du mot clé **const** à la définition d'une variable permet de faire en sorte que la valeur de la variable ne puisse pas changer au cours de sa vie. Ainsi, à partir de la définition de cette variable, l'opérateur d'affectation = ne pourra plus s'appliquer dessus. C'est pourquoi cette variable *constante* doit **obligatoirement** être initialisée ; dans le cas contraire cela provoquera une erreur de compilation.

Solution de l'exercice 5

```
#include<iostream> using namespace std;
int main() {
 double x=3.14;
 cout<<"x="<<x<<"\n";

 double* pt=&x;
 *pt=2.71828;
 cout<<"x="<<x<<" "<<"x="<<*pt<<"\n";
 cout<<"adresse="<<pt<<"\n";

 double& ref=x;
 ref=2.71828;
 cout<<"x="<<x<<" "<<"x="<<ref<<"\n";
 cout<<"adresse="<<&ref<<"\n";

 return 0;
};
```

dans une unité de compilation .cpp

L'adresse de `x` peut être stockée dans une variable de type **`double`**`*` (qui peut être lu "pointeur sur **`double`**"). L'adresse de `x` peut être obtenue en lui appliquant l'opérateur unaire préfixe `&`. L'opérateur d'*indirection* (ou de *déréférencement*) `*` permet de renvoyer une lvalue correspondant à l'espace

mémoire pointé par le pointeur pour y lire la valeur ou y écrire (si le pointeur n'est pas **const**) une nouvelle valeur. La variable pt contient une valeur qui est égale à l'adresse de la variable x et qui peut être affichée sur la sortie standard.
L'adresse de x peut aussi être stockée dans une variable de type **double**& (qui peut être lu "lvalue-référence sur **double**"). Pour cela la lvalue x est directement utilisée pour initialiser la référence sans utiliser l'opérateur &; c'est le compilateur qui prend en charge la récupération de l'adresse. La référence permet d'accéder directement (sans utiliser l'opérateur d'indirection) à l'espace mémoire pointé par la référence pour y lire la valeur ou y écrire (si la référence n'est pas **const**) une nouvelle valeur ; c'est le compilateur qui prend en charge le déréférencement à partir de l'adresse stockée dans la référence. On accède à l'adresse contenue dans une référence en lui appliquant l'opérateur unaire préfixe &.

Solution de l'exercice 6

```
1  double* pt1=4096; /* erreur, pas de conversion implicite de int vers double* */
2  double* pt2=(double*)4096; /* ok, conversion explicite autorisée; dangereux à
       utiliser (est ce que cette adresse est valide ? y a t-il une valeur de type
       double à cette adresse); de plus double * pt2=reinterpret_cast<double *
       >(4096); est normalement la bonne manière en C++ d'écrire  ce type d'
       instruction */
3  void* pt3=pt1; // ok, conversion implicite de double* en void* autorisée
4  pt1=pt3; // erreur, conversion implicite de void* en double* interdite
5  pt1=(double*)pt3; /* ok, conversion explicite autorisée.
6  Cependant, en C++, pour obtenir un code sûr et portable, il faudrait utiliser l'
       instruction suivante:
7  pt1=static_cast<double*>(pt3);*/
8  double d1=36; // ok
9  const double d2=36; // ok
10 double* pt4=&d1; // ok
11 const double* pt5=&d1; // ok, conversion implicite de double* vers const double*
12 *pt4=2.1; // ok, modification de la valeur pointée possible
13 *pt5=2.1; /* erreur, pas de modification de la valeur pointée possible à partir
       d'un pointeur const :
14 la lvalue renvoyée par l'indirection est constante */
15 pt4=&d2; /* erreur, pas de conversion implicite posssible de const double* vers
       double* */
16 pt5=&d2; // ok
17 double* const pt6=&d1; // ok
18 pt6=&d1; /* erreur, on ne peut pas changer la valeur d'un pointeur constant */
19 *pt6=2.78; /* ok, pt6 est un pointeur constant, mais pas un pointeur const; on
       peut donc modifier la valeur pointée par pt6, mais on ne peut pas modifier la
        valeur de pt6 */
20 double* const pt6b=&d2; /* erreur, pas de conversion de const double* vers
       double* const */
21 const double* const pt7=&d1; // ok
22 pt7=&d1; /* erreur, on ne peut pas changer la valeur d'un pointeur constant */
23 *pt7=2.78; /* erreur, pas de modification possible de la valeur pointée à partir
        d'un pointeur const */
24 double const* pt8=&d1; /* ok, attention équivalent à const double* pt8=&d1; */
25 pt8=&d2; // ok
26 pt8=&d1; /* ok, conversion implicite de double* vers const double* */
```

```
27 *pt8=3.14; /* erreur, pas de modification possible de la valeur pointée à partir
       d'un pointeur const */
```

dans une unité de compilation .cpp

Solution de l'exercice 7

```
1  double& d1=36; // erreur, pas d'initialisation possible d'une lvalue-référence
       avec une rvalue
2  double d2=36; // ok
3  double& ref=d2; // ok, modification possible de la valeur référencée
4  ref=4; // ok, même effet que d2=4;
5  const double d3=36; // ok
6  const double& d4=36; // ok, initialisation possible d'une référence const avec
       une rvalue
7  const double& d5=d2; // ok
8  d5=21; // erreur, pas de modification possible de la valeur référencée à partir
       d'une référence const
9  const double& d6=d3; // ok
10 double& ref2=d6; // erreur, initialisation impossible d'une référence non-const
       avec une référence const
11 int i=4;
12 double& d7=i; // erreur, une référence de type A& doit être initialisée avec une
       lvalue type A
13 const double& d8=i; //ok, conversion possible
14 d8=3; // erreur, pas de modification possible de la valeur référencée à partir d
       'une référence const
```

dans une unité de compilation .cpp

Remarquons que la ligne 12 provoque une erreur au contraire de la ligne 13. En utilisant une référence **const**, s'il existe une conversion acceptable du type de la donnée à référencer et le type de la référence, le compilateur accepte de créer cette référence. Cependant, la donnée n'est pas vraiment référencée ; c'est une donnée temporaire résultante de la conversion qui l'est. Cela est transparent pour l'utilisateur car il ne peut pas modifier la donnée réellement référencée (ce qui ne serait pas le cas avec une référence non-**const**, d'où l'erreur ligne 12).

Solution de l'exercice 8

```
1 int* pt1=new int; // ou new int();
2 *pt1=2;
3 int* pt2=new int(4);
4 delete pt1;
5 delete pt2;
```

dans une unité de compilation .cpp

Pour initialiser un entier lors de son allocation dynamique, il suffit de transmettre entre parenthèses, après **int**, la valeur qui doit servir à l'initialisation. Dans l'instruction `int* pt1=new int;`, la variable pointée par `pt1` n'est pas initialisée (sa valeur reste donc inconnue jusqu'à sa première affectation. Si par contre, à sa place, on utilise l'instruction `int* pt1=new int();`, la variable pointée par `pt1` est initialisée avec la *valeur par défaut* d'un entier, c'est à dire 0. Ce faisant, on utilise le constructeur sans argument du type **int** qui est généré par le compilateur pour simuler le comportement des variables de type objet (qui possèdent toutes des constructeurs). L'opérateur **delete** unaire permet de libérer la zone mémoire pointée par l'adresse qui lui est fournie.

Solution de l'exercice 9

```
1 double* pt1=new double[20];
2 for(unsigned int i=0; i<20; i++) pt1[i]=3.14;
3 delete[] pt1;
```

dans une unité de compilation .cpp

Pour allouer dynamiquement un tableau, l'opérateur **new[]** est utilisée à la place de **new**. Il n'est pas possible d'initialiser les cellules du tableau avec cet opérateur. L'opérateur **delete[]** doit être utilisé pour désallouer le tableau en entier. Si l'opérateur **delete** est utilisé à sa place, seule la première cellule du tableau sera désallouée.

Solution de l'exercice 10

Question - Version 1 : Passage de l'objet `Date` par valeur

La variable `d` est une variable locale à la fonction qui sera initialisée par recopie de la valeur passée en paramètre. La date résultat est également retournée par recopie afin de permettre à son utilisateur de mettre à jour son objet `Date` (qui se trouve dans un autre contexte).

```
1 #include "date.h"
2 Date UpdateDateVal(Date d, unsigned int j, unsigned int m, unsigned int a){
3  d.jour=j; d.mois=m; d.annee=a ; return d;
4 }
```

date.cpp

Question - Version 2 : Passage de l'objet `Date` par adresse (pointeur)

La modification porte sur l'objet au travers de son adresse transmise par valeur qui sert à initialiser le pointeur.

```
1 ...
2 void UpdateDatePt(Date* pt, unsigned int j, unsigned int m, unsigned int a){
3  pt->jour=j; pt->mois=m; pt->annee=a ;
4 }
```

date.cpp

Question - Version 3 : Passage de l'objet `Date` par référence

La modification porte sur l'objet au travers de son adresse transmise par valeur qui sert à initialiser la référence.

```
1 ...
2 void UpdateDateRef(Date& ref, unsigned int j, unsigned int m, unsigned int a){
3  ref.jour=j; ref.mois=m; ref.annee=a ;
4 }
```

date.cpp

Question - Appel des 3 versions

```
1 struct Date {
2  unsigned int jour, mois, annee;
3 };
```

```
4 Date UpdateDateVal(Date d, unsigned int j, unsigned int m, unsigned int a);
5 void UpdateDatePt(Date* pt, unsigned int j, unsigned int m, unsigned int a);
6 void UpdateDateRef(Date& ref, unsigned int j, unsigned int m, unsigned int a);
```

date.h

```
1 #include "date.h"
2 void main(){
3  Date d;
4  d=UpdateDate(d,3,2,2020);
5  UpdateDatePt(&d,3,2,2020);
6  UpdateDateRef(d,3,2,2020);
7 }
```

dans une unité de compilation .cpp

Question - Discussion

- La version 1 provoque la recopie de 2 valeurs `Date` lors de la transmission des arguments, ainsi que la recopie d'une valeur lors du retour du résultat. Chaque recopie est de la taille de **`sizeof`** `(Date)` unités de mémoire. Pour des raisons d'efficacité, cette version est donc à éviter. De plus l'utilisateur peut très bien ne pas utiliser la valeur de retour pour affecter la date passée en argument.
- La version 2 semble alors plus appropriée. Mais l'utilisation en est plus compliquée puisqu'il faut utiliser l'opérateur & pour fournir l'adresse de la variable `Date` à manipuler. De plus, la manipulation de pointeurs reste une opération délicate : l'utilisateur peut toujours fournir des adresses qui ne correspondent pas à des objets `Date`.
- Dans ce contexte (où il n'est pas nécessaire de changer la valeur des pointeurs durant leur utilisation), il est donc préférable d'utiliser la version 3 qui présente tous les avantages : lisibilité, efficacité (seules des références d'objet sont créées) et sécurité (les références étant normalement valides sauf manipulation intentionnelle de l'utilisateur de la fonction).

Solution de l'exercice 11

```
1 struct G_Struct {
2  // ...
3  char large_array[100000];
4  // ...
5 };
6
7 void read_only_procedure(const G_Struct& the_struct);
```

dans un fichier d'entête .h

Il faut utiliser le passage par référence pour éviter les recopies. De plus, il faut que cette référence soit **`const`** pour s'assurer qu'il n'y aura pas de modification possible pour la structure référencée.

Solution de l'exercice 12

A la sortie du bloc la variable `the_object_to_return` n'existe plus car c'est une variable automatique. Par conséquent, puisque c'est une variable locale à la fonction, et même si l'emplacement mémoire associé peut encore contenir la bonne valeur, cet emplacement est libéré à partir de la fin de la fonction. Attention, si en appelant une telle procédure tout se passe normalement, ce n'est que le

fruit du hasard et de la chance. C'est que l'emplacement mémoire n'a pas encore été réutilisé et qu'il contient toujours la bonne valeur, mais il s'agit bien d'une erreur.
Plusieurs solutions sont envisageables pour remédier à ce genre de situation :

— Retourner la valeur par recopie : peu efficace dans le cas où le type de retour est un type lourd en mémoire.

```
object solution1_function(){
 object the_object_to_return;
  ...
 return the_object_to_return;
}
```

dans une unité de compilation .cpp

— Créer un objet dynamique et le retourner.

```
object& solution2_function(){
 object* the_object_to_return = new object;
  ...
 return *the_object_to_return;
}
```

dans une unité de compilation .cpp

Cependant, cette solution est dangereuse car elle responsabilise l'appelant. En effet, c'est à lui de supprimer par un **delete** l'objet retourné.

— L'appelant donne à l'appelé l'objet sur lequel il doit travailler et qu'il doit retourner.

```
object& solution3_function(object& the_object_to_return){
  ...
 return the_object_to_return;
}
```

dans une unité de compilation .cpp

Notons que dans ce cas, il n'y a plus beaucoup d'intérêt à renvoyer une référence.

Solution de l'exercice 13

```
string s1="bonjour";
f(s1);
g(s1);
h(s1);
const string s2="salut";
f(s2);
g(s2); // erreur : référence non const initialisée avec une valeur constante
h(s2);
const string& ref1=s2;
string& ref2=s2; // erreur : référence non const initialisée avec une valeur
    constante
string s3=s2;
f("coucou");
g("coucou"); // erreur, initialisation d'une référence string avec une valeur de
    type const char* (types incompatibles)
```

```
14 h("coucou"); // ok, initialisation possible d'une référence const de string avec
       le résultat de la conversion de const char* en string
```

dans une unité de compilation .cpp

Solution de l'exercice 14

```
1 int somme(int a, int b, int c=0, int d=0, int e=0);
```

exemple.h

Les valeurs par défaut sont toujours placées dans un fichier d'entête. On ne doit pas répéter ces valeurs dans l'unité de compilation. Remarquons que dans notre cas, la définition de la fonction `somme` reste inchangée (des zéros sont ajouter à la somme sans incidence sur le résultat final).

Solution de l'exercice 15

```
1 #include <limits>
2 unsigned int minimum(
3       unsigned int a, unsigned int b,
4       unsigned int c=std::numeric_limits<unsigned int>::max(),
5       unsigned int d=std::numeric_limits<unsigned int>::max(),
6       unsigned int e=std::numeric_limits<unsigned int>::max());
```

exemple.h

La valeur maximum que peut prendre une variable de type **`unsigned int`** est une bonne candidate pour servir de valeur par défaut. On peut connaitre cette valeur limite en utilisant la fonction `std ::numeric_limits<unsigned int>::max()` dont la déclaration est fournie dans le fichier d'entête `limits` de la bibliothèque standard (namespace `std`). Remarquons alors que la valeur par défaut peut être le résultat de l'appel d'une fonction. Ces fonctions sont ici des méthodes **`static`** de la classe patron standard `numeric_limits`.

Solution de l'exercice 16

```
1 void incrementer(int& x);
2 void incrementer(double& x);
3 void incrementer(char& x);
4 void incrementer(int* tab, unsigned int n);
5 void incrementer(double* tab, unsigned int n);
6 void incrementer(char* tab, unsigned int n);
```

fonctions_increment.h

```
1  #include "fonctions_increment.h"
2  void incrementer(int& x){ x++; }
3
4  void incrementer(double& x){ x++; }
5
6  void incrementer(char& x){ x++; }
7
8  void incrementer(int* tab, unsigned int n){
9   for(unsigned int i=0; i<n; i++) incrementer(tab[i]);
10 }
11
```

```
12 void incrementer(double* tab, unsigned int n){
13  for(unsigned int i=0; i<n; i++) incrementer(tab[i]);
14 }
15
16 void incrementer(char* tab, unsigned int n){
17  for(unsigned int i=0; i<n; i++) incrementer(tab[i]);
18 }
```

fonctions_increment.cpp

Solution de l'exercice 17

```
1 inline void incrementer(int& x){ x++; }
2 inline void incrementer(double& x){ x++; }
3 inline void incrementer(char& x){ x++; }
```

fonctions_increment.h

```
1 // plus de définition ici
```

fonctions_increment.cpp

Pour transformer ces fonctions en fonctions **inline**, il suffit d'ajouter le mot clé **inline** devant l'entête de la fonction. Notons que le corps de la fonction doit alors obligatoirement se trouver dans un fichier d'entête (d'extension `.h`) qui sera inclus par toute unité de compilation utilisant ces fonctions : le code source de ces fonctions doit être disponible pour que le compilateur puisse remplacer les appels aux fonctions par leurs instructions.

Solution de l'exercice 18

```
1  #include<iostream>
2  using namespace std;
3  namespace FX {
4   void incrementer(int& x){
5    x++;
6   }
7
8   void incrementer(double& x){
9    x++;
10  }
11
12  void incrementer(char& x){
13   x++;
14  }
15 }
16
17 int main() {
18  int a=6;
19  char b=8;
20  double c=34.2;
21  FX::incrementer(a); // utilisation de l'opérateur de résolution de portée
22  using FX::incrementer; // using declaration
23  incrementer(b);
```

```
24  using namespace FX; // using directive, inutile ici
25  incrementer(c);
26  return 0;
27 }
```

Classes

Exercice 19 - Définition d'une classe

Écrire une classe `Personne` permettant de décrire complètement un individu, sachant que l'on souhaite avoir autant d'informations que dans la phrase suivante : "M. Holly Pierre est né en 1965, il est célibataire".

Exercice 20 - Définition de méthodes

Ajouter à la classe `Personne` de l'Exercice 19 des accesseurs en lecture pour tous les attributs. Ces méthodes seront **`inline`** . Définir aussi une méthode `age` qui renvoie l'âge de l'individu en fonction d'une année fournie en paramètre de méthode et de type **`unsigned int`**. Définir cette méthode afin qu'elle ne soit pas **`inline`**.

Exercice 21 - Définition d'un constructeur

Ajouter un constructeur à la classe `Personne` qui permet d'initialiser tous les attributs d'un objet `Personne`.

Exercice 22 - Allocation dynamique d'un objet ou d'un tableau d'objets

1. Écrire une fonction dans laquelle un objet `Personne` (voir Exercices 19, 20 et 21) est alloué dynamiquement puis détruit.
2. Peut-on allouer dynamiquement un tableau d'objets `Personne` ? Expliquer.

Exercice 23 - Diagramme de classe

Dessiner un modèle UML de la classe `Personne` (voir Exercices 19, 20 et 21) en faisant apparaître tous les attributs et les méthodes avec leur visibilité, ainsi que tous les détails qui semblent utiles dans un diagramme de classe.

Exercice 24 - Définition d'une méthode et utilisation de la classe standard **`stringstream`**

Ajouter à la classe `Personne` (voir Exercices 19, 20 et 21) une méthode retournant une chaîne de caractères similaire à la phrase donnée dans l'énoncé de l'exercice 19.

Exercice 25 - Relations entre objets

1. Ajouter à la classe `Personne` (voir Exercices 19, 20 et 21) un attribut représentant un éventuel conjoint et examiner les conséquences que cela peut avoir sur l'ensemble du code. Introduire des méthodes qui permettent le mariage et le divorce entre 2 personnes. On supposera qu'une personne ne peut avoir que 0 ou 1 seul conjoint.
2. Représenter cette relation et ces méthodes en complétant le diagramme UML de la classe `Personne` (*cf.* Exercice 23).
3. Cette relation peut-elle être représentée par une agrégation ? une composition ? Expliquer.

Exercice 26 - Objet composite et allocation dynamique

Définir une classe `vecteur` qui compose un tableau de **`double`**. Ce tableau est représenté par un attribut de type pointeur sur **`double`**. Un attribut supplémentaire `dim`, de type **`unsigned int`**, représente la dimension du vecteur. Ecrire un constructeur prenant en argument la dimension du vecteur à construire, qui initialise l'attribut `dim` avec cette dimension, qui alloue dynamiquement un tableau de **`double`** de taille `dim` et qui initialise ce tableau avec des zéros.

Exercice 27 - Le mot clé `explicit`

Dans l'exercice précédent, est-il intéressant d'utiliser le mot clé **`explicit`** devant ce constructeur ? Expliquer.

Exercice 28 - Objet composite et destruction

Ajouter un destructeur à la classe `vecteur` (voir Exercice 26) si nécéssaire.

Exercice 29 - Surcharge du constructeur

Reprendre l'Exercice 26 en ajoutant un constructeur qui prend en argument la dimension du vecteur et un tableau de valeurs de type **`double`** qui servira à l'initialisation d'un objet `vecteur`.

Exercice 30 - Constructeur et valeurs par défaut

Remplacer les deux constructeurs définis dans l'Exercice 26 et l'Exercice 29 en les remplaçant par un seul constructeur qui utilise un argument par défaut.

Exercice 31 - Surcharge du constructeur de recopie

Surcharger le constructeur de recopie de la classe `vecteur` (voir Exercices 26 et 28) si nécessaire. Expliquer.

Exercice 32 - Surcharge d'opérateurs

Compléter la classe `vecteur` (voir Exercices 26 et 28) en ajoutant la surcharge des opérateurs `+` et `*` qui permettent respectivement d'effectuer l'addition et le produit scalaire de 2 vecteurs. Surcharger l'opérateur `+` en tant que méthode de la classe `vecteur` et l'opérateur `*` en tant que fonction non-membre de la classe.

Exercice 33 - Surcharge de l'opérateur d'affectation

Surcharger l'opérateur d'affectation `=` de la classe `vecteur` (voir Exercices 26 et 28) si nécessaire. Expliquer.

Exercice 34 - Surcharge de l'opérateur `[]`

Compléter la classe `vecteur` (voir Exercices 26 et 28) en ajoutant la surcharge de l'opérateur `[]` qui permet d'accéder en lecture et en écriture à la valeur de la dimension `i` d'un objet `vecteur`.

Exercice 35 - Membres `static`

Ajouter à la classe `vecteur` (voir Exercices 26-34) un attribut **`static`** qui permet de compter le nombre d'objets `vecteur` présents en mémoire. Ajouter également une méthode **`static`** qui permet d'accéder en lecture à ce nombre. Modifier si nécessaire des méthodes de la classe `vecteur`.

Solution de l'exercice 19

Dans la phrase "M. Holly Pierre est né en 1965, il est célibataire", on trouve : le sexe (M., né, il), le nom, le prénom, l'année de naissance et la situation familiale. Les informations de situation familiale "marié" et "veuf" doivent être adaptées en fonction du sexe lors d'un éventuel affichage.

```
1  #include<string>
2  using namespace std;
3  enum statut { celibataire, marie, veuf };
4  enum sexe_enum { homme, femme };
5  class Personne {
6   string nom; // le nom
7   string prenom; // le prénom
8   unsigned int annee_naissance; // année de naissance
9   sexe_enum sexe; // sexe
10  statut situation_familiale; // situation familiale
11 };
```

Personne.h

Deux énumérations sont définies avec le mot clé **enum** afin d'établir les différentes possibilités de situation familiale et de sexe.
Pour définir une classe, on utilise le mot clé **class** puis les informations sont définies entre "`{`" et "`};`". Les différentes informations sont réprésentées par des attributs (ou données membres). Un attribut est défini comme une variable (son type, suivi de son indentificateur et d'un point-virgule).
La plupart du temps, une classe est définie dans un fichier d'entête (fichier `.h`) afin de la rendre utilisable dans plusieurs unités de compilation.
La représentation en mémoire d'un objet `Personne` est exactement la même que pour une variable structure : en effet, on peut remplacer **class** par **struct** de manière équivalente, mais à quelques nuances prêts. En effet, contrairement à une variable de type **struct**, les attributs ne sont pas, ici, accessibles par un utilisateur de la classe. Ainsi, les instructions suivantes constituent une erreur :

```
1 Personne p;
2 p.annee_naissance=1991; // erreur, pas d'accès à un membre privé
```

instructions à mettre dans une fonction

Cela est dû au fait que ces attributs sont placés *implicitement* dans la partie privée de la classe. On peut marquer ce fait de façon *explicite* en utilisant le mot clé **private**. La définition de la classe `Personne` suivante serait alors équivalente :

```
1  #include<string>
2  using namespace std;
3  enum statut { celibataire, marie, veuf };
4  enum sexe_enum { homme, femme };
5  class Personne {
6  private: \\début de la partie privée
7   string nom; // le nom
8   string prenom; // le prénom
9   unsigned int annee_naissance; // année de naissance
10  sexe_enum sexe; // sexe
11  statut situation_familiale; // situation familiale
12 };
```

Personne.h

La différence entre **struct** et **class** en C++ est : tout est public par défaut dans une structure, alors que tout est privé par défaut dans une classe. On peut utiliser le mot clé **private** dans une structure

pour y ajouter des parties privées. De même, on peut utiliser le mot clé **public** pour ajouter des parties publiques dans une classe. Ce mot clé sert alors à indiquer les attributs et les méthodes qui seront accessibles à l'utilisateur de la classe. Ainsi, si dans le code précédent, le mot clé **private** est remplacé par le mot clé **public**, les attributs de la classe `Personne` deviennent accessibles à un utilisateur de la classe et les instructions suivantes deviendraient correctes :

```
1 Personne p;
2 p.annee_naissance=1991; // ok, maintenant annee_naissance est un attribut public
```

instructions à mettre dans une fonction

Mettre les attributs dans la partie publique de la classe est cependant à éviter. En effet, cela constitue une entorse au *principe d'encapsulation* des données qui veut qu'un attribut ne soit accessible que par l'intermédiaire de méthodes afin de contrôler son accès et de masquer la structure de données de la classe que l'utilisateur est censé ignorer.
Enfin, notons l'utilisation du type `string` qui permet de représenter des chaines de caractères en C++ de manière plus sécurisée et plus simple que les chaînes de type C (tableaux de **char**).

Solution de l'exercice 20

Puisque les attributs sont dans la partie privée de la classe, il ne sont pour l'instant pas accessibles par l'utilisateur (voir correction de l'Exercice 19). On peut cependant définir des méthodes dans la partie publique pour pouvoir y accéder :

```
1   ...
2 class Personne {
3  ...
4 public:
5   ...
6  string get_nom() const { return nom; }
7  string get_prenom() const { return prenom; }
8  unsigned int get_annee_naissance() const  { return annee_naissance; }
9  sexe_enum get_sexe() const { return sexe; }
10 statut get_situation_familiale() const { return situation_familiale; }
11 unsigned int age(unsigned int anneeEnCours) const;
12 };
```

Personne.h

```
1 #include "Personne.h"
2 unsigned int Personne::age(unsigned int anneeEnCours) const { return
     anneeEnCours-annee_naissance; }
```

Personne.cpp

Une méthode (ou fonction membre) est une fonction déclarée à l'intérieure de la définition de la classe (au même endroit que les attributs) et qui s'applique à un objet.
Comme n'importe quelle fonction, elle possède un type de retour et des paramètres. La différence est qu'une méthode possède en plus un argument implicite : l'objet qui appelle la méthode. Cet objet est accessible par l'intermédiaire du mot clé **this** : dans le corps d'une méthode, **this** est un pointeur qui contient l'adresse de l'objet qui l'appelle. Ainsi, on peut accéder à un attribut de l'objet au travers de ce pointeur, *e.g.* **this**->nom. Cependant, comme une méthode s'applique toujours à un objet, il n'est pas obligatoire de préciser **this**-> : le nom d'un attribut se rapporte toujours par défaut à l'argument implicite.
Quand le corps de la méthode est écrit dans la déclaration de la classe (comme la méthode `get_nom()` , la méthode est **inline** (voir correction de l'Exercice 17). Tout appel à cette méthode sera alors peut être (dans les contextes où c'est possible et intéressant) remplacé par les instructions de la méthode

au moment de la compilation. En général, toutes les méthodes courtes sont définies **inline** pour éviter le coût (en temps et en mémoire) généré par un appel de fonction. C'est souvent le cas des *accesseurs*. "Accesseur" est le nom que l'on donne à une méthode qui permet d'accéder à un attribut. Un accesseur qui sert à seulement connaître (et non modifier) la valeur d'un attribut est dit *accesseur en lecture*. Un accesseur qui sert à modifier un attribut (il n'y en a pas dans cet exercice) est dit *accesseur en écriture*.
Les fonctions qui ont un certain nombre d'instructions ne sont, en général, pas **inline**. Même si la méthode `age(unsigned int anneeEnCours) const` aurait pu être **inline** (elle est très courte) nous la rendons non **inline** pour l'exemple. Dans ce cas, la méthode est seulement déclarée dans la définition de la classe. La définition de la méthode est alors fournie en dehors de la classe. Cette définition doit être placée dans une unité de compilation (et non dans le même fichier d'entête que la classe) afin qu'il n'y ait pas plusieurs définitions de la méthode lors des multiples inclusions du fichier d'entête de la classe dans les différentes unités de compilation où la classe pourrait être utilisée. Afin de préciser au compilateur qu'une définition de fonction est en fait la définition d'une méthode non-**inline** de la classe `Personne`, l'opérateur de résolution de portée `Personne::` est utilisé devant le nom de la méthode dans son entête.
Remarquons que les méthodes accèdent aux attributs, donc à la partie privée de la classe : seules les méthodes de la classe ont ce privilège.
Puisque ces méthodes ne sont pas censées modifier la valeur d'un attribut, elles devraient être toutes **const**. Ce **const** s'applique en fait à l'argument implicite de la méthode : il indique que les attributs de l'objet de la méthode ne peuvent pas être modifiés dans le corps de la méthode. En plus de sécuriser le code au moment de son écriture, ce mot clé indique quelles sont les méthodes qui peuvent être appelées par les objets constants ou les références **const** d'objet.

Solution de l'exercice 21

Un constructeur est une méthode qui porte le même nom que la classe. Un constructeur n'a jamais de type de retour (même pas **void**). Les paramètres transmis à cette méthode serviront normalement à initialiser les attributs de l'objet. À chaque fois qu'un objet est défini, un constructeur est appelé.
Alors qu'aucun constructeur n'a encore été défini, le code suivant est pourtant correct et déclenche l'appel d'un constructeur :

```
1 Personne p;
```

instruction à mettre dans une fonction

Cette instruction déclenche en effet l'appel d'un constructeur généré automatiquement par le compilateur. Ce constructeur est sans argument, ce qui explique qu'il n'y a pas besoin de lui transmettre de l'information lors de cette instruction. C'est en quelque sorte une méthode qui permet d'initialiser avec une valeur par défaut un objet `Personne`. Cependant, le constructeur sans argument généré par le compilateur ne fait qu'appeler le processus d'initialisation par défaut de chacun des attributs. Pour les attributs de type `string`, cela correspond à une initialisation avec la chaîne de caractères vide. Pour les attributs de type primitif (comme **unsigned int**), la valeur reste malheureusement inconnue jusqu'à la prochaine affectation. Pour redéfinir ce comportement, il suffit de proposer une définition du constructeur sans argument au compilateur.
Le constructeur que nous définissons ici prend en argument les différents éléments qui servent à donner des valeurs aux différents attributs :

```
1   ...
2 class Personne {
3  ...
4 public:
5  Personne(const string& n, const string& p,
6   unsigned int a, sexe_enum s, statut  sit){
7   nom=n;
8   prenom=p;
9   annee_naissance=a;
```

```
10   sex=s;
11   situation_familiale=sit;
12  }
13 };
```

personne.h

Remarquons que ce constructeur doit être placé dans la partie publique de la classe pour qu'il puisse être accessible par un de ses utilisateurs. Notons aussi que l'instruction suivante devient maintenant incorrecte :

```
1 Personne p; // erreur, il n'y a plus de constructeur sans argument
```

instruction à mettre dans une fonction

En effet, à partir du moment où un constructeur est défini par l'utilisateur, le compilateur ne génére plus le constructeur sans argument par défaut. La seule manière possible pour définir un objet `Personne` est maintenant de fournir les informations requises comme dans l'instruction suivante :

```
1 Personne p("Holly","Pierre",1965,homme,celibataire); // ok
```

instructions à mettre dans une fonction

Si l'on souhaite pouvoir maintenant construire un objet `Personne` sans fournir d'information, il faut définir un constructeur sans argument.
Si on s'intéresse maintenant aux instructions du constructeur, on peut remarquer que toutes les instructions (comme `prenom=pre;`) sont en fait des affectations, *i.e.* les attributs sont déjà initialisés au moment de l'ouverture de l'accolade marquant le début des instructions. Il est cependant possible de vraiment initialiser les attributs (et non plus de faire une affectation) en utilisant la notation avec `":"` entre la déclaration des paramètres et le début des instructions :

```
1   ...
2 class Personne {
3  ...
4 public:
5  Personne(const string& n, const string& p,
6   unsigned int a, sexe_enum s, statut  sit):
7   nom(n), prenom(p), annee_naissance(a)
8   sexe(s), situation_familiale(sit){}
9 };
```

personne.h

Notons que cette notation, qui est la seule possible pour initialiser des attributs, devient nécessaire pour toute donnée membre qu'il est nécessaire habituellement d'initialiser comme les constantes ou les références.

Solution de l'exercice 22

1. Pour créer dynamiquement un objet `Personne`, il faut utiliser l'opérateur **`new`** suivi de l'appel d'un constructeur auquel on transmet les informations nécessaires à son initialisation :

```
1 #include "Personne.h";
2 int f() {
3  Personne* pt=new Personne("Holly","Pierre",1965,homme,celibataire);
4  delete pt;
5  return 0;
6 }
```

une unité de compilation .cpp

Si la classe possédait un constructeur sans argument (soit généré par le compilateur, soit défini par le créateur de la classe), on pourrait utiliser l'instruction `Personne* pt=new Personne;` (remarquons que ce n'est pas le cas ici).
L'opérateur **`delete`** permet de désallouer un objet alloué dynamiquement. Notons que cet opérateur fait systématiquement appel au destructeur de l'objet avant de libérer son emplacement mémoire.

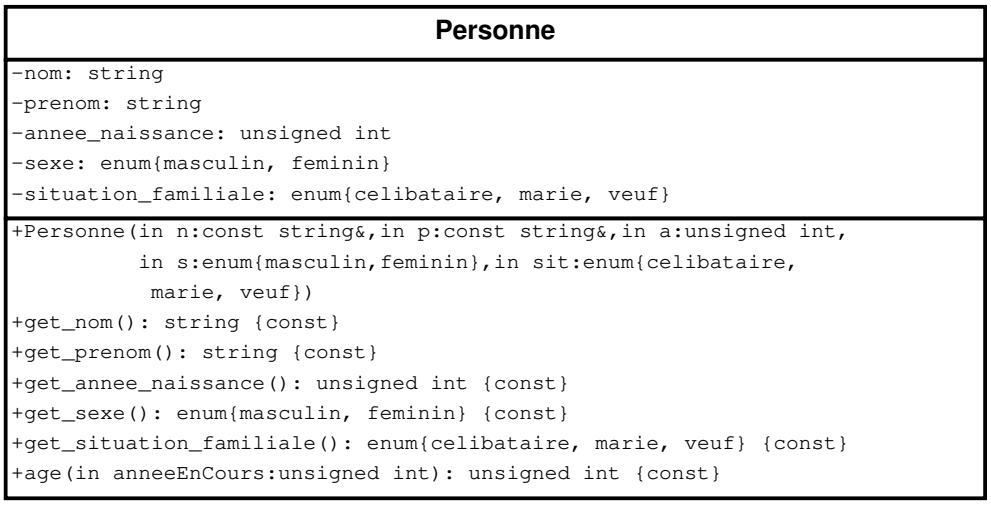
2. Lorsqu'on utilise l'opérateur **`new[]`**, il n'est pas possible de lui transmettre des informations pour initialiser les différents objets d'un tableau d'objet. Aussi seule la syntaxe suivante est correcte :

```
1 #include "Personne.h";
2 int f() {
3  Personne* tab=new Personne[10]; // correct seulement si Personne::Personne
       () est définie
4  delete[] tab;
5  return 0;
6 }
```

une unité de compilation .cpp

Cependant, dans ce contexte, cette instruction est tout de même invalide car elle suppose qu'un constructeur sans argument de la classe `Personne` va être appelé. Or ce constructeur n'existe pas pour la classe `Personne`. Il faut donc retenir que l'on peut construire un tableau d'objets seulement si la classe de ces objets possède un constructeur sans argument. Il reste cependant la solution d'allouer dynamiquement, un par un, chaque objet du tableau d'objets. S'il avait été possible d'allouer dynamiquement un tableau d'objets `Personne`, l'opérateur **`delete[]`** aurait permis de le désallouer. Notons que cet opérateur fait systématiquement appel au destructeur de chacun des objets du tableau avant de libérer son emplacement mémoire.

Solution de l'exercice 23



Pour dessiner le diagramme UML de la classe `Personne`, un rectangle divisé en 3 parties est dessiné. Dans la première partie, le nom de la classe est inscrit. Les attributs sont placés dans la deuxième partie et les méthodes dans la troisième. Les membres privés sont précédés du signe – alors que les membres publics sont précédés du signe +. On indique toujours le type d'un élément après le signe "**:**". Des commentaires (comme **`const`**) peuvent être inscrits entre accolades.

Solution de l'exercice 24

```
1 #include<string>
2 #include<sstream>
```

```
3 using namespace std;
4 class Personne {
5  ...
6 public:
7  string retourneInfos() const {
8   stringstream sb;
9   sb<< sexe==homme?"M. ":"Mme ")
10    <<nom<<' '<<prenom
11    <<" est "<< (sexe==homme?"né ":"née ")
12    <<"en "<<annee_naissance
13    <<", "<<(sexe==homme?"il ":"elle ")<<"est ";
14   switch (situation_familiale) {
15   case celibataire: sb<<" celibataire  "; break;
16   case marie: sb<<(sexe==homme?" marié ":" mariée "); break;
17   case veuf: sb<<(sexe==homme?" veuf ":" veuve ");
18   }
19   return sb.str();
20  }
21 };
```

personne.h

Un objet de la classe `stringstream` se comporte comme un objet de la classe `ostream`. On construit une chaîne de caractère de la même manière que l'on ferait un affichage en utilisant l'objet `cout` avec l'opérateur **`operator`**<<. La méthode `str()` permet de récupérer la chaîne de caractères construite sous forme d'objet de la classe `string`. Notons que la méthode `retourneInfos` est **`const`** puisqu'elle ne modifie aucun des attributs de la classe.

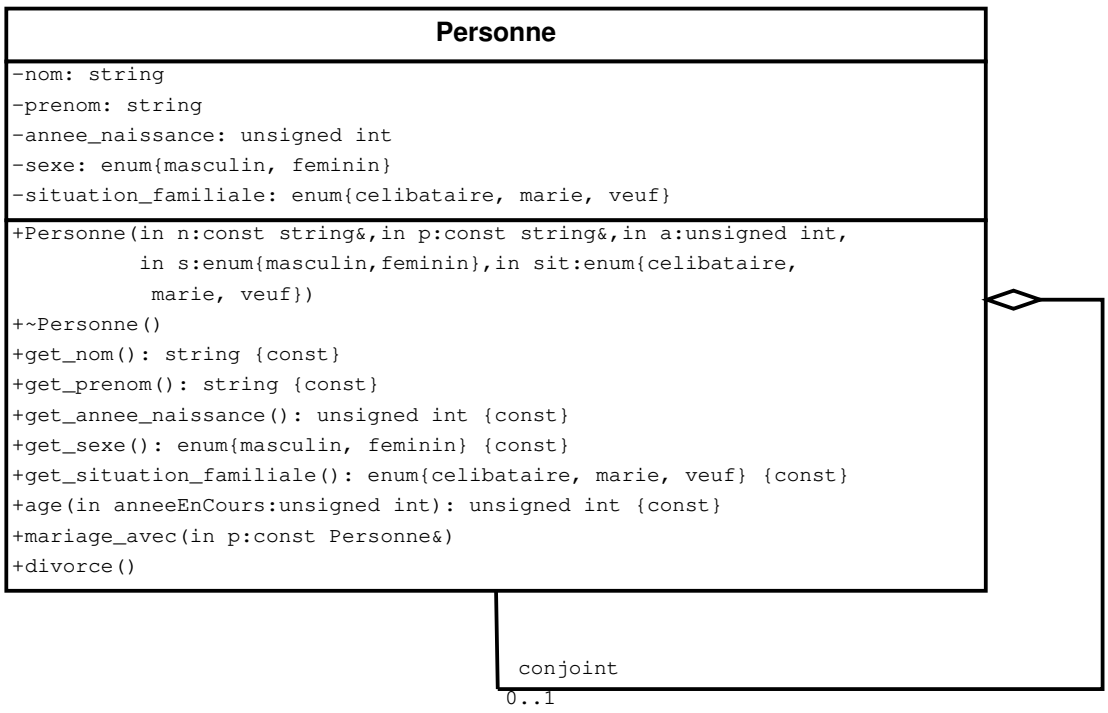
Solution de l'exercice 25

```
1 #include<iostream>
2 using namespace std;
3 ...
4 class Personne {
5  ...
6 private:
7  Personne* conjoint;
8 public:
9  Personne(const string& n, const string& p,
10   unsigned int a, sexe_enum s, statut  sit):
11   nom(no), prenom(pre), annee_naissance(a),
12   sexe(s), situation_familiale(sit),conjoint(nullptr){}
13  ~Personne(){
14   if (conjoint!=nullptr) {
15    conjoint->conjoint=nullptr;
16    conjoint->situation_familiale=veuf;
17   }
18  }
19  void mariage_avec(Personne& p){
20   if (conjoint==nullptr && p.conjoint==nullptr){
21    conjoint=&p;
22    situation_familiale=marie;
23    p.conjoint=this;
```

```
24    p.situation_familiale=marie;
25   }else throw "mariage impossible\n";
26  }
27  void divorce(){
28   if (conjoint!=nullptr){
29    conjoint->conjoint=nullptr;
30    conjoint->situation_familial=celibataire;
31    conjoint=nullptr;
32    situation_familiale=celibataire;
33   }else throw "erreur divorce: conjoint inexistant\n";
34  }
35 };
```

personne.h

L'existence du conjoint implique une liaison entre deux objets `Personne`. Celle-ci se fait par un cycle de pointeurs : un objet `Personne` pointera vers son conjoint et inversement. Au niveau du constructeur il faut initialiser l'attribut pointeur `conjoint` au pointeur nul `nullptr`, signifiant que l'objet ne possède pas (pour l'instant) de conjoint. Les méthodes `mariage_avec` et `divorce` induisant une modification de l'attribut `conjoint`, elles ne peuvent pas être **`const`** .
Une modification de statut pour l'un, induira une modification du statut de l'autre. En cas de disparition de l'un par séparation ou par décès, il faudra modifier le statut de l'autre et le pointeur de celui-ci, si l'objet `Personne` est supprimé.
Il faut donc introduire un destructeur pour la classe `Personne`. Le destructeur est une méthode qui est appelée automatiquement avant la destruction de l'objet. Il porte le nom de la classe précédé du signe "~". Il n'a pas de type de retour (comme le constructeur). Il sert à effectuer des traitements que l'on juge nécessaires avant la disparition de l'objet de la mémoire : ici il s'agit de la modification du statut d'un éventuel conjoint (qui devient veuf à la suite du décès de l'objet). On remarquera que la disparition d'un conjoint n'entraîne pas nécessairement la disparition de l'autre. La relation conjoint est donc une agrégation simple et non une composition. Une composition signifierait en effet que si un objet `Personne` est détruit (par ex. à la suite du décès de la personne qu'il représente), son conjoint doit l'être aussi. La relation `conjoint` peut donc être représentée sur le modèle UML suivant :



Solution de l'exercice 26

```
1 class vecteur {
2 public:
3  vecteur(unsigned int d);
4 private:
5  unsigned int dim;
6  double* tab;
7 };
```

vecteur.h

```
1 #include "vecteur.h"
2 vecteur::vecteur(unsigned int d):tab(new double[d]), dim(d){
3  for(unsigned int i=0; i<dim; i++) tab[i]=0;
4 }
```

vecteur.cpp

Solution de l'exercice 27

```
1 class vecteur {
2 public:
3  explicit vecteur(unsigned int d);
4 private:
5  unsigned int dim;
6  double* tab;
7 };
```

vecteur.h

Il est utile ici d'utiliser le mot clé **explicit** devant ce constructeur à un argument. Ce mot clé empêche la conversion *implicite* d'**unsigned int** en `vecteur` qui pourrait entrainer des erreurs de la part de l'utilisateur. En effet, rappelons que si une classe `A` implémente un constructeur prenant un seul objet de type `B` en paramètre, *i.e.* `A::A(B)` ou `A::A(B&)` ou `A::A(const B&)`, alors le compilateur met en place la conversion *implicite* de `B` vers `A`. Les opérations qui suivent sont alors possibles :

```
1 void f(A); // déclaration d'une fonction f
2 B b;
3 A a=b;
4 f(b);
```

En particulier, la fonction `f` peut prendre indifféremment un objet de type `A` ou de type `B` en argument grâce à la conversion de `B` vers `A`. Dans le cas de la classe `vecteur` et de son constructeur prenant un **unsigned int** en paramètre, cela signifie que toute fonction qui attend un objet de type `vecteur` peut recevoir à la place une valeur de type **unsigned int** puisque c'est une opération valide. Dans ce cas, un objet `vecteur` temporaire de la dimension indiquée par l'**unsigned int** est créé et sert à l'initialisation du paramètre de la fonction. Si l'on souhaite éviter cet effet, qui peut être ici pénalisant (est ce vraiment de que l'on souhaite ?), le mot clé **explicit** permet d'empêcher cette conversion implicite. Si besoin, cette conversion devra alors être explicite.
Ainsi si une classe `A` implémente un constructeur prenant un seul objet de type `B` en paramètre mais en utilisant le mot clé **explicit**, *e.g.* **explicit** `A(B);`, alors le compilateur empêche la conversion *implicit* de `B` vers `A`. Les opérations qui suivent sont alors impossibles :

```
1 void f(A); // déclaration d'une fonction f
2 B b;
3 A a=b; // erreur
4 f(b); // erreur
```

L'utilisateur est obligé d'utiliser explicitement le constructeur :

```
1 void f(A); // déclaration d'une fonction f
2 B b;
3 A a(b); // ou A a=A(b);
4 f(A(b));
```

Solution de l'exercice 28

```
1 class vecteur {
2 public:
3  ...
4  ~vecteur(){ delete[] tab; }
5 };
```

vecteur.h

Puisque l'objet `vecteur` est responsable du tableau (alloué dynamiquement) qu'il compose, il est responsable de sa destruction. Ceci se fait normalement à la fin de la vie de cet objet. C'est pourquoi la désallocation dynamique du tableau a lieu dans le destructeur qui est appelé juste avant la libération mémoire de l'objet `vecteur`.

Solution de l'exercice 29

```
1 class vecteur {
2 public:
3  ...
4  vecteur(unsigned int d, const double* v);
5  ...
6 };
```

vecteur.h

```
1 vecteur::vecteur(unsigned int d, const double* v):tab(new double [d]), dim(d){
2  for(unsigned int i=0; i<dim; i++) tab[i]=v[i];
3 }
```

vecteur.cpp

Solution de l'exercice 30

```
1 class vecteur {
2 public:
3  ...
4  explicit vecteur(unsigned int d, const double* v=nullptr);
```

```
5  ...
6 };
```

vecteur.h

```
1 vecteur::vecteur(unsigned int d, const double* v):tab(new double [d]), dim(d){
2  if (v)
3   for(unsigned int i=0; i<dim; i++) tab[i]=v[i];
4  else
5   for(unsigned int i=0; i<dim; i++) tab[i]=0;
6 }
```

vecteur.cpp

La valeur par défaut `nullptr` pour le paramètre `v` permet de ne pas définir le constructeur de l'Exercice 26. Si l'utilisateur ne propose pas de 2^{eme} argument, la valeur de `v` sera alors `nullptr`. Un simple test dans le constructeur permet alors de savoir si `v` doit être pris en compte ou non. Notons la présence du mot clé **`explicit`** qui interdit la conversion d'**`unsigned int`** en `vecteur` si l'utilisateur ne fournit qu'un seul argument.

Solution de l'exercice 31

Rappelons que le compilateur génère par défaut un constructeur pour la recopie d'objet. Celui-ci permet de construire un objet à partir d'un objet de la même classe. Cela permet notamment de généraliser l'initialisation d'une variable avec une autre variable du même type. Dans le fonctionnement par défaut, chaque attribut de l'objet à créer est initialisé avec l'attribut correspondant de l'objet qui sert à l'initialisation.
Si cet attribut est un type primitif, cette initialisation est classique : la valeur de l'attribut de l'objet initialisateur est utilisée telle quelle pour initialiser l'attribut de l'objet à initialiser.
Si cet attribut est lui-même un objet, alors le constructeur de recopie (généré par défaut par le compilateur ou créé par le concepteur de la classe) de cet objet est utilisé avec comme argument l'objet-attribut de l'objet initialisateur.
Ce comportement par défaut ne convient pas pour la classe vecteur. En effet, un pointeur est un type primitif, et donc, dans le constructeur de recopie généré par défaut, la valeur de l'attribut `tab` de l'objet `vecteur` initialisateur sert à initialiser l'attribut `tab` de l'objet `vecteur` à initialiser. Cela signifie qu'après l'appel de ce constructeur, deux objets `vecteur` partagent le même tableau de **`double`**. Or chaque objet `vecteur` **compose** son tableau de **`double`**. Cela signifie que chaque objet `vecteur` est propriétaire de son tableau de **`double`** et gère sa construction, sa modification et sa destruction. Un tableau de **`double`** *partagé* ne convient donc pas : si un des deux vecteurs modifie (détruit) son tableau, l'autre vecteur est également modifié (détruit). En particulier, la désallocation répétitive d'un tableau conduit à une erreur d'exécution.
Pour remédier à cette situation, il suffit de proposer une autre définition du constructeur de recopie :

```
1 class vecteur {
2 public:
3  ...
4  vecteur(const vecteur& v);
5 };
```

vecteur.h

```
1 vecteur::vecteur(const vecteur& v):
2  tab(new double [v.dim]),dim(v.dim){
3  for(unsigned int i=0; i<dim; i++) tab[i]=v.tab[i];
4 }
```

vecteur.cpp

Remarquons que l'objet `vecteur` initialisateur est toujours transmis par référence. Une tentative d'omettre le signe indiquant la référence est refusée par le compilateur. En effet, un passage par valeur, qui correspond à une initialisation d'un paramètre avec la valeur passée en argument provoque l'appel du constructeur de recopie du type du paramètre. Or ; dans le constructeur que nous sommes en train de définir, le type du paramètre est `vecteur`. Un passage par valeur du `vecteur` argument provoquerait donc un appel récursif infini puisqu'il appelle le constructeur par recopie qui est en train d'être défini.
Ici, le modificateur **`const`** est utilisé avec le paramètre pour empêcher la modification de l'argument dans les instructions du constructeur. Bien que cela n'est pas obligatoire, il est évidemment utile de le faire dans ce contexte.

Solution de l'exercice 32

```
1 class vecteur {
2 public:
3  ...
4  vecteur operator+(const vecteur& v) const;
5  ...
6  friend double operator*(const vecteur&, const vecteur&);
7 };
8
9 double operator*(const vecteur& v1, const vecteur& v2);
```

vecteur.h

```
1 vecteur vecteur::operator+(const vecteur& v) const{
2  if (dim!=v.dim)
3   throw "erreur : addition de vecteurs de dimensions différentes";
4  vecteur res=*this;
5  for(unsigned int i=0; i<dim; i++) res.tab[i]+=v.tab[i];
6  return res;
7 }
8
9 double operator*(const vecteur& v1, const vecteur& v2){
10  if (v1.dim!=v2.dim)
11   throw "erreur : produit scalaire de vecteurs de dimensions différentes";
12  double res=0;
13  for(unsigned int i=0; i<v1.dim; i++) res+=v1.tab[i]*v2.tab[i];
14  return res;
15 }
```

vecteur.cpp

Notons que le retour de ces opérateurs doivent se faire par valeur. Le retour par référence d'une variable locale à la fonction pourrait être source d'erreur d'exécution si l'utilisateur d'une de ces fonctions utilisait cette référence (invalide, puisque pointant vers un espace mémoire libéré à la fin de la fonction).
Notons l'utilisation d'une déclaration d'amitié de la classe `vecteur` envers la surcharge de l'opérateur `+`. En effet, cette fonction n'étant pas une méthode de la classe `vecteur`, l'accès à la partie privée de `vecteur`, et en particulier à l'attribut `tab`, ne lui est normalement pas permis. L'absence d'accesseur à cette donnée (pour le moment) nous oblige à palier à ce problème.

Solution de l'exercice 33

Notons tout d'abord qu'une affectation n'est pas une situation de construction : l'objet à gauche du = est déjà construit avant l'appel de l'opération d'affectation.

Rappelons que le compilateur génère par défaut un opérateur d'affectation pour l'affectation d'un objet avec un objet de la même classe. Dans le fonctionnement par défaut, chaque attribut de l'objet à gauche de l'opérateur = est affecté avec l'attribut correspondant de l'objet à droite de l'opérateur d'affectation.
Si cet attribut est un type primitif, cette affectation est classique : la valeur de l'attribut de l'objet à droite de = est utilisée telle quelle pour affecter l'attribut de l'objet à gauche de =.
Si cet attribut est lui-même un objet, alors l'opérateur d'affectation (généré par défaut par le compilateur ou créé par le concepteur de la classe) de cet objet est utilisé avec, comme argument, l'objet-attribut de l'objet à droite.
Ce comportement par défaut ne convient pas pour la classe `vecteur`. En effet, un pointeur est un type primitif et donc, tout comme dans le constructeur de recopie généré par défaut, la valeur de l'attribut `tab` de l'objet `vecteur` à droite de = sert à affecter l'attribut `tab` de l'objet `vecteur` à gauche de =. Cela signifie qu'après l'appel de cet opérateur, deux objets `vecteur` partagent le même tableau de **`double`**. Pour les mêmes raisons que pour le constructeur de recopie (voir Exercice 31), il faut alors proposer une autre définition de l'opérateur d'affectation :

```
1 class vecteur {
2 public:
3  ...
4  vecteur& operator=(const vecteur& v);
5 };
```

vecteur.h

```
1 vecteur& vecteur::operator=(const vecteur& v){
2  if (this!=&v){
3   delete[] tab;
4   dim=v.dim;
5   tab=new double[dim];
6   for(unsigned int i=0; i<dim; i++) tab[i]=v.tab[i];
7  }
8  return *this;
9 }
```

vecteur.cpp

Remarquons que l'objet `vecteur` initialisateur est transmis par référence. Contrairement au constructeur de recopie, ce mode de transmission n'est pas obligatoire. Il est cependant conseillé car un passage par valeur engendrerait, par la même occasion, l'appel du constructeur de recopie de `vecteur` pour initialiser le paramètre de la méthode = : ici, cette recopie est inutile et consommatrice de temps et de mémoire.
Le modificateur **`const`** est utilisé avec le paramètre pour empêcher la modification de l'argument dans les instructions de l'opérateur. Bien que cela n'est pas obligatoire, il évidemment utile de le faire dans ce contexte.
Le retour par référence utilisé ici n'est pas obligatoire : ce pourrait être un passage par valeur, un autre type que `vecteur`, ou tout simplement **`void`**. Ici, on préfère utiliser une transmission par référence de l'objet lui même pour permettre des affectations en cascade du type `v1=v2=v3=v4;`. Notons que dans ce cas, l'opérateur = est associatif à droite, *i.e.* les affectations sont réalisées en commençant par l'affectation la plus à droite (ici, `v3=v4`).
Notons aussi que dans l'opérateur d'affectation, il est souvent nécessaire de vérifier qu'il ne s'agit pas d'un cas d'auto-affectation en utilisant une instruction du type **`if`** `(this!=&v)...` (l'adresse de l'objet sur lequel s'applique l'opérateur est comparée à l'adresse de l'objet passé en argument). Dans le code précédent, si cette vérification est omise, le vecteur désallouera son propre tableau de **`double`** et mènera à une perte de ses informations et probablement à une erreur d'exécution lors de la lecture des données désallouées en ligne 16.

Solution de l'exercice 34

```
1 class vecteur {
2 public:
3  ...
4  double& operator[](unsigned int i){ return tab[i]; }
5  const double& operator[](unsigned int i) const { return tab[i]; }
6  ...
7 };
```

vecteur.h

L'opérateur `[]` ne peut être surchargé qu'en tant que méthode de classe. Tenter d'en faire une fonction non-membre provoquerait une erreur à la compilation. C'est un opérateur binaire. En tant que fonction membre (méthode), il doit donc prendre un argument (l'autre étant l'argument implicite, c'est à dire l'objet qui appelle la méthode). Ici deux surcharges de l'opérateur sont proposées : une méthode non **const** qui sera utilisée par les objets non constants (ou à partir de références non **const**), et une méthode **const** qui sera utilisée par les objets constants (ou à partir des références **const** d'objet).
Pour la première surcharge, notons la transmission de la valeur de retour avec une référence non-**const** . Ceci permet la modification d'une dimension d'un vecteur par l'utilisateur de la méthode. Pour la deuxième surcharge, la valeur de retour est une référence **const**. Cette référence ne peut donc pas être utilisée pour modifier la valeur de la dimension concernée. Ce retour aurait également pu être fait par valeur.

Solution de l'exercice 35

```
1 class vecteur {
2 public:
3  vecteur(unsigned int d);
4   vecteur(unsigned int d, const double* v);
5  vecteur(const vecteur& v);
6  vecteur& operator=(const vecteur& v);
7  ~vecteur(){ delete[] tab; nb--;}
8  ...
9  static unsigned int get_nb_vecteurs() { return nb; }
10 private:
11  ...
12  static unsigned int nb;
13 };
```

vecteur.h

```
1 unsigned int vecteur::nb=0;
2
3 vecteur::vecteur(unsigned int d):
4  tab(new double[d]), dim(d)
5 {
6  for(unsigned int i=0; i<dim; i++) tab[i]=0;
7  nb++;
8 }
9 vecteur::vecteur(unsigned int d, const double* v):
10  tab(new double [d]), dim(d)
11 {
```

```
12  for(unsigned int i=0; i<dim; i++) tab[i]=v[i];
13  nb++;
14 }
15
16 vecteur::vecteur(const vecteur& v):
17  tab(new double [v.dim]),dim(v.dim)
18 {
19  for(unsigned int i=0; i<dim; i++) tab[i]=v.tab[i];
20  nb++;
21 }
```

vecteur.cpp

```
1  #include<iostream> using namespace std;
2  #include "vecteur.h"
3  int main() {
4    cout<<"nombre de vecteurs="<<vecteur::get_nb_vecteurs()<<"\n";
5   vecteur v1(3);
6   double tab[3]={ 0.11, 3.2, 6.7 };
7   vecteur v2(3,tab);
8   cout<<"nombre de vecteurs="<<vecteur::get_nb_vecteurs()<<"\n";
9   vecteur v3(v2);
10  vecteur* v4= new vecteur(3);
11  cout<<"nombre de vecteurs="<<vecteur::get_nb_vecteurs()<<"\n";
12  delete v4;
13  cout<<"nombre de vecteurs="<<vecteur::get_nb_vecteurs()<<"\n";
14  cout<<"nombre de vecteurs="<<v2.get_nb_vecteurs()<<"\n";
15  return 0;
16 }
```

dans une unité de compilation .cpp

Pour que le nombre de vecteurs `nb` soit toujours à jour, il est nécessaire que tout constructeur (y compris le constructeur de recopie) incrémente (`nb++`) cet attribut **`static`**, et que le destructeur le décrémente (`nb--`).

Notons qu'un membre **`static`** public est accessible en utilisant l'opérateur de résolution de portée, *e.g.* `vecteur::get_nb_vecteurs()`, et ceci même si aucun objet de la classe existe (comme dans la première instruction de la fonction `main` de l'exemple). On peut aussi y accéder en utilisant un objet, *e.g.* `v2.get_nb_vecteurs()`.

Puisque la méthode `get_nb_vecteurs()` ne modifie pas d'attribut de la classe, on peut être tenté d'ajouter le qualificatif **`const`** dans l'entête de la méthode : **`static unsigned int`** `get_nb_vecteurs()`**`const`**. Cependant, ceci provoquerait une erreur à la compilation. En effet, dans ce contexte, le qualificatif **`const`** s'applique à l'argument implicite de la méthode, c'est à dire l'objet sur lequel s'applique la méthode. Or, une méthode **`static`** pouvant s'exécuter sans objet, elle ne possède pas d'argument implicite. Pour ces mêmes raisons, une méthode **`static`** **n'a pas accès aux membres (attributs et méthodes) non-`static`** de la classe, puisque ces derniers appartiennent à un objet en particulier (attributs) ou nécessitent un objet et ses attributs (méthodes).

Héritage

Exercice 36 - Sous-classement d'une classe

Soit la classe `tache` définie comme suit :

```
1  // fichier tache.h
2  #include <iostream>
3  #include <string>
4  using namespace std;
5  class tache {
6  public:
7   tache(const string& n, unsigned int d=0):nom(n),duree(d){}
8   void afficher(ostream& f=cout) const {
9    f<<"tache "<<nom<<": duree="<<duree;
10  }
11  const string& get_nom() const { return nom; }
12  unsigned int get_duree() const { return duree; }
13 private:
14  string nom;
15  unsigned int duree;
16 };
```

Un objet `tache` se caractérise par une durée et un nom. Les méthodes fournies permettent de construire un objet `tache` en fournissant son nom et éventuellement sa durée (nulle par défaut), d'accéder en lecture à chacun des attributs, et d'écrire les caractéristiques d'un objet sur un flux `ostream`.

1. Ecrire une classe `mission` sachant qu'un objet `mission` est une `tache` munie d'une priorité se caractérisant par un entier positif.
2. Définir toute fonction nécessaire à la *création* d'un objet mission. Compléter la classe `mission` en ajoutant un accesseur en lecture pour l'attribut représentant la priorité de la mission.
3. Assurez vous qu'un objet de la classe `mission` peut utiliser les membres **`public`** de la classe `tache`.

Exercice 37 - Héritage en UML

Dessiner en UML la hiérarchie de classe correspondante à l'Exercice 36 en faisant apparaître les attributs et les méthodes.

Exercice 38 - Redéfinition de méthode

1. Redéfinir la méthode `afficher` dans la classe `mission` de façon à afficher aussi la priorité d'un objet `mission` lorsque cette méthode est appelée.
2. Modifier le modèle UML de l'Exercice 37 en conséquence pour tenir compte du changement apporté.
3. Le mot clé **`virtual`** est-il nécessaire pour que la méthode `afficher` puisse être appelée à partir d'un objet `mission` ?
4. S'assurer que les membres privés de la classe `tache` ne sont pas accessibles dans le corps de la méthode `mission::afficher`.

Exercice 39 - Membres `protected`

1. Faire en sorte que les attributs de la classe `tache` de l'Exercice 36 soient accessibles dans le corps des méthodes de la classe `mission` mais pas par les utilisateurs de la classe mission.
2. Modifier le modèle UML de l'Exercice 38 en conséquence.

Exercice 40 - Substitution - Upcasting

Parmi les instructions suivantes, quelles sont celles qui sont valides et celles qui ne le sont pas ?

```
1  tache t("tache 1",5);
2  mission m("mission 1",3,99);
3  t=m;
4  m=t;
5  tache t2(m);
6  mission m2(t);
7  tache* ptt;
8  ptt=&t;
9  ptt=&m;
10 mission* ptm;
11 ptm=&t;
12 ptm=&m;
13 tache& ref1=t;
14 tache& ref2=m;
15 mission& ref3=t;
16 mission& ref4=m;
```

Exercice 41 - Couplage statique

Dans les instructions suivantes, à chaque fois qu'une méthode `afficher` est appelée, indiquer s'il s'agit de celle de `tache` ou de celle de `mission` :

```
1  tache t("tache 1",5);
2  mission m("tache 2",3,99);
3  t.afficher();
4  m.afficher();
5  tache* ptt;
6  ptt=&t;
7  ptt->afficher();
8  ptt=&m;
9  ptt->afficher();
10 mission* ptm=&m;
11 ptm->afficher();
12 tache& reft1=t;
13 reft1.afficher();
14 tache& reft2=m;
15 reft2.afficher();
16 mission& refm=m;
17 refm.afficher();
```

Exercice 42 - Couplage dynamique- Polymorphisme

1. Transformer le code des exercices précédents de façon à ce que, lorque la méthode `afficher` est appelée à partir d'un pointeur ou de référence de `tache`, et que ce pointeur ou cette référence pointe en réalité sur un objet `mission`, ce soit la méthode `mission::afficher()` qui soit appelée.
2. Modifier le modèle UML de l'Exercice 39 en conséquence.

Exercice 43 - Classe abstraite

1. Définir une classe abstraite `affichable` qui désigne toute classe implémentant une méthode **`public`** dont l'entête est **`void`** `afficher(ostream&)` **`const`**.
2. Définir une classe `truc` composant un attribut de type **`int`**, qui hérite de la classe `affichable`, et dont il est possible d'instancier des objets.
3. Modifier votre code de façon à ce que la classe `tache` hérite de la classe `affichable`
4. Modifier le modèle UML de l'Exercice 42 en introduisant les 2 nouvelles classes `affichable` et `truc`.

Exercice 44 - Design Pattern Template-Method

Implémenter une fonction qui permet d'afficher n'importe quel objet `affichable` sur un flux `ostream` en utilisant l'opérateur <<.

Exercice 45 - Reconnaissance de type à l'exécution

1. Définir une méthode pour la classe `affichable` dont l'entête est **`void`** `afficher_type(ostream& f=cout)` **`const`**. Cette méthode affiche sur un flux `ostream` le type réel de l'objet qui appelle la méthode.
2. Définir la méthode d'entête **`bool`** `est_du_meme_type_que(`**`const`** `affichable&)` **`const`** pour la classe `affichable`. Cette méthode renvoie la valeur **`true`** si l'objet passé en paramètre de la méthode est du même type (réel) que l'objet qui appelle la méthode, et la valeur **`false`** sinon.

Exercice 46 - Héritage multiple - Downcasting

1. Définir un modèle de classe `comparable` qui représente toute classe implémentant la méthode **`bool operator`**`==(`**`const`** `comparable& c)` **`const`**. Cette méthode (virtuelle pure) renverra **`true`** si l'objet qui est passé en paramètre de la méthode est du même type et a les mêmes valeurs d'attributs que l'objet qui appelle la méthode.
2. Modifier le code de façon à ce que les classes `tache` et `truc` héritent à la fois de la classe `affichable` et de la classe `comparable`. Définir la méthode **`operator`**== pour chacune de ces classes.
3. Modifier le modèle UML de l'Exercice 43 en introduisant la classe `comparable`.

Exercice 47 - Héritage multiple en losange

1. Définir une classe `commande` qui hérite de la classe `tache` et qui compose en plus un entier positif désignant un numéro de commande. Définir un constructeur et un accesseur pour son attribut. Redéfinir la méthode `afficher`.
2. Définir une classe `commande_prioritaire` qui hérite à la fois de la classe `commande` et de la classe `mission`. Définir un constructeur. Redéfinir la méthode `afficher`. Faire cette question sans utiliser l'héritage **`virtual`**.
3. Refaire la question précédente en utilisant l'héritage **`virtual`**. Modifier le code si nécessaire.
4. Quelles sont les limites de l'héritage en losange et de l'héritage **`virtual`** ?

Solution de l'exercice 36

1. Pour qu'une classe `B` hérite d'une classe `A`, il suffit de l'indiquer en ajoutant une instruction du type `:` *"mode"* `A` entre le nom de la classe `B` et le signe `}`. Le *mode* peut être **`public`**, **`protected`** , **`private`** permet de préciser des nuances d'accès pour les utilisateurs et les méthodes de la classe `B` par rapport aux membres **`public`** et **`protected`** de la classe `A`. Le mode **`public`** indique que tous les membres **`public`** de la classe `tache` sont accessibles aussi bien par les méthodes de la classe `mission` que par les utilisateurs de la classe `mission`. Les membres **`private`** de la classe `tache` restent inaccessibles en dehors des méthodes de la classe `tache`.

```
1 class mission : public tache {
2 private:
3  unsigned int priorite;
4 };
```

2. Pour l'instant, il n'est pas possible de construire un objet `mission` car le compilateur ne peut pas générer automatiquement de constructeur sans argument car il ne sait pas comment construire la partie `tache` de mission. La règle est que tout constructeur d'une classe dérivée doit obligatoirement faire appel à un constructeur de sa classe de base (afin d'initialiser la partie correspondant à la classe de base), sauf si la classe de base dispose d'un constructeur sans argument (la partie de la classe de base peut alors être initialisée sans information supplémentaire). La classe `tache` ne disposant pas de constructeur sans argument, il faut obligatoirement définir un constructeur pour la classe mission qui peut être par exemple :

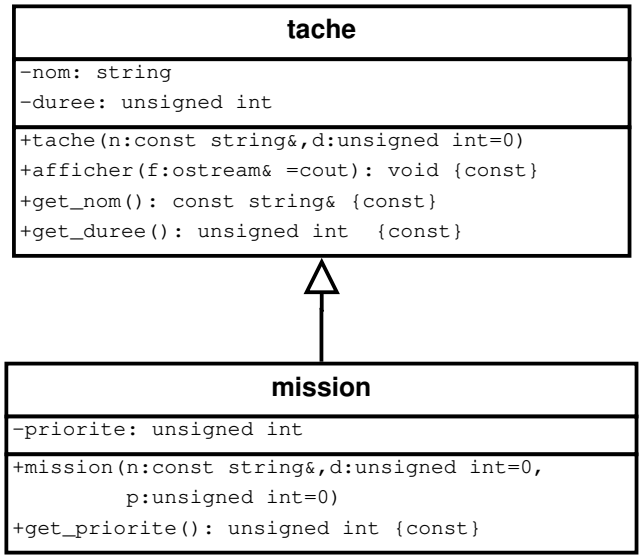
```
1 class mission : public tache {
2 public:
3  mission(const string& n, unsigned int d=0, unsigned int p=0):
4             tache(n,d),priorite(p){}
5   unsigned int get_priorite() const { return priorite; }
6 private:
7  unsigned int priorite;
8 };
```

   Remarquons alors, l'appel au constructeur de `tache` (le seul disponible) juste après les arguments du constructeur en utilisant la notation avec `":"`.

3. L'exécution des instructions suivantes montre que les méthodes `afficher(),get_duree()`, et `get_nom()` de la classe `tache` sont utilisables à partir d'un objet `mission` de la même manière que la méthode `get_priorite()` :

```
1 mission m("une mission",4,78);
2 m.afficher();
3 cout<<m.get_priorite()<<"\n";
4 cout<<m.get_duree()<<"\n"; //ok
5 cout<<m.get_nom()<<"\n"; // ok
```

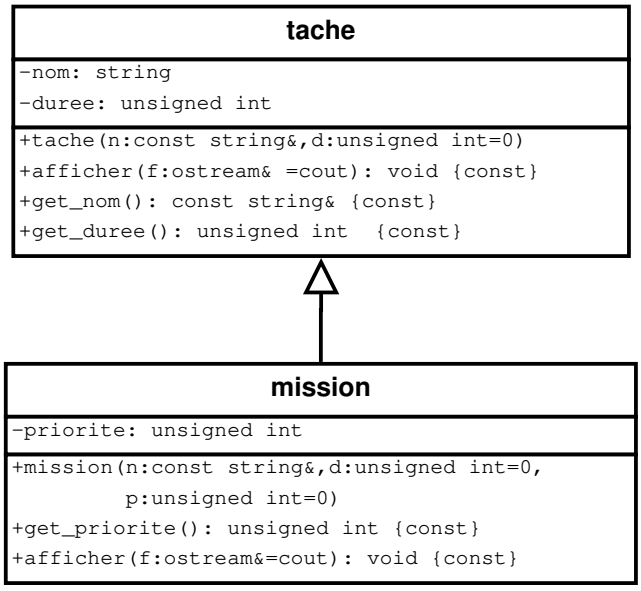
Solution de l'exercice 37



Solution de l'exercice 38

```
1  class mission : public tache {
2  public:
3   ...
4   void afficher(ostream& f=cout) const {
5    tache::afficher(f);
6    f<<", priorite="<<priorite;
7    // ou f<<"tache "<<get_nom()<<": duree="<<get_duree()<<", priorite="<<
         get_priorite();
8   }
9   ...
10 };
```

1. Pour redéfinir la méthode `afficher`, il suffit d'en fournir les nouvelles instructions. La méthode `afficher` de la classe `tache` est toujours accessible en utilisant l'opérateur de résolution de portée (voir ligne 5). Attention, dans ce contexte, si l'opérateur de résolution de portée était omis, il s'agirait d'un appel récursif à la méthode `afficher` de la classe `mission`. Notons, qu'ici, il n'était pas obligatoire d'utiliser la méthode `afficher` de la classe `tache`. En effet, on aurait pu exécuter, à la place, les instructions de la ligne 7 en utilisant les accesseurs de la classe `tache`.
2. Puisque que la méthode `afficher` est maintenant redéfinie dans la classe `mission`, on la fait apparaître sur son diagramme dans le modèle UML :



3. Le mot clé **`virtual`** n'a pas besoin d'être présent pour que la méthode `afficher` puisse être appelée à partir d'un objet `mission` comme le montrerait l'exécution des instructions suivantes :

```
1 tache t("tache 1",5);
2 mission m("tache 2",3,99);
3 t.afficher(); // appel de tache::afficher
4 m.afficher(); // appel de mission::afficher
```

4. Les attributs de la classe `tache` sont **`private`**. Ils ne sont donc pas accessibles dans le corps des méthodes de `mission`. Ainsi, le compilateur indique des erreurs, si on tente la définition de `mission::afficher()` avec les instructions suivantes :

```
1 class mission : public tache {
2 public:
3  ...
4  void afficher(ostream& f=cout) const {
5   f<<"tache "<<nom<<": duree="<<duree<<", priorite="<<priorite;
6  }
7  ...
8 };
```

Solution de l'exercice 39

1. Pour cela, il suffit de placer les attributs de la classe `tache` dans la partie **`protected`** de la classe `tache` :

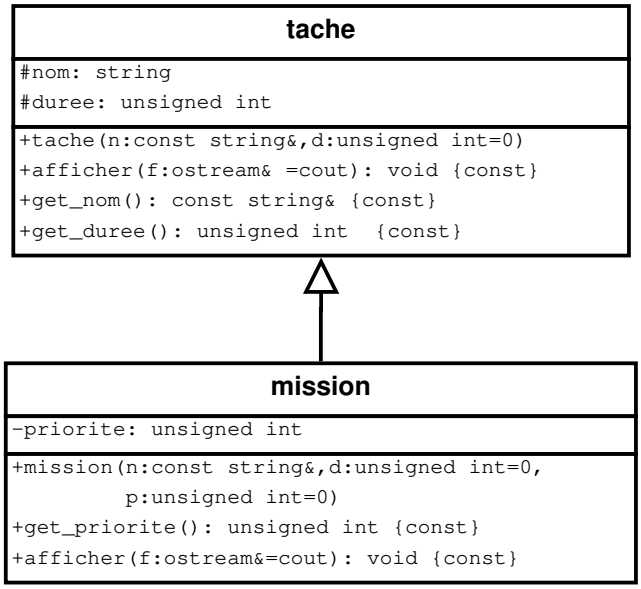
```
1 class tache {
2 public:
3  ...
4 protected:
5  string nom;
6  unsigned int duree;
7 };
```

Remarquons alors que cela donne le droit aux méthodes de la sous-classe `mission` d'accéder à ces attributs. La méthode `mission::afficher` pourrait être réécrite :

```
1 class mission : public tache {
2 public:
3  ...
4  void afficher(ostream& f=cout) const {
5   f<<"tache "<<nom<<": duree="<<duree<<", priorite="<<priorite;
6  }
7  ...
8 };
```

Les attributs **`protected`** de la classe `tache` restent néanmoins inaccessibles pour les utilisateurs de la classe `mission`.

2. En remplaçant les – (indiquant des attributs **`private`**) par des **`#`** (indiquant des attributs **`protected`**), le modèle UML devient :



Solution de l'exercice 40

```
1 tache t("tache 1",5); // ok
2 mission m("mission 1",3,99); // ok
3 t=m; // ok, affectation de t avec la partie tache de l'objet m.
4 m=t; // erreur, pas d'affectation possible d'un objet d'une classe vers un objet
      d'une de ses sous-classes
5 tache t2(m); // ok, initialisation de t2 avec la partie tache de l'objet m.
6 mission m2(t); // erreur, pas de construction possible d'un objet d'une sous-
     classe à partir d'un objet de sa classe de base
7 tache* ptt; // ok
8 ptt=&t; // ok
9 ptt=&m; // ok, un pointeur de tache peut pointer vers un objet d'une sous-classe
      de tache
10 mission* ptm; // ok
11 ptm=&t; // erreur, un pointeur de mission ne peut pas pointer vers un objet de
     sa classe de base.
12 ptm=&m; // ok
```

```
13 tache& ref1=t; // ok
14 tache& ref2=m; // ok, une référence de tache peut référencer un objet d'une sous
       -classe de tache
15 mission& ref3=t; // erreur, une référence de mission ne peut pas référencer un
       objet de sa classe de base.
16 mission& ref4=m; // ok
```

Remarquons que pour une instruction du type `ptt=&m`, une conversion de `mission*` en `tache*` est exécutée, *i.e.* une conversion d'une classe vers une classe ascendante. On parle alors d'*up-casting* (conversion vers le haut de l'arborescence).

Solution de l'exercice 41

```
1  tache t("tache 1",5);
2  mission m("tache 2",3,99);
3  t.afficher(); //tache::afficher()
4  m.afficher(); //mission::afficher()
5  tache* ptt;
6  ptt=&t;
7  ptt->afficher(); //tache::afficher()
8  ptt=&m;
9  ptt->afficher(); //tache::afficher()
10 mission* ptm=&m;
11 ptm->afficher(); //mission::afficher()
12 tache& reft1=t;
13 reft1.afficher(); //tache::afficher()
14 tache& reft2=m;
15 reft2.afficher(); //tache::afficher()
16 mission& refm=m;
17 refm.afficher(); //mission::afficher()
```

Le couplage entre un objet et une méthode est ici décidé à la compilation en fonction du type de la variable de la manière suivante :

- si la méthode `afficher` est appelée à partir d'une variable `tache`, d'un pointeur de `tache` ou d'une référence de `tache`, c'est la méthode `tache::afficher()` qui est appelée ;
- si la méthode `afficher` est appelée à partir d'une variable `mission`, d'un pointeur de `mission` ou d'une référence de `mission`, c'est la méthode `mission::afficher()` qui est appelée.

Ce couplage variable-méthode n'est pas remis en cause lors de l'exécution : même si un pointeur de `tache` pointe sur un objet `mission` (comme sur les lignes 4 et 12), c'est `tache::afficher()` qui est appelée. C'est pourquoi on parle de *couplage statique*.

Solution de l'exercice 42

1. Pour que le couplage variable-méthode soit *dynamique*, *i.e.* décidé au moment de l'exécution et non au moment de la compilation, il suffit d'ajouter le mot clé **`virtual`** devant l'entête de la méthode dans la définition de la classe de base :

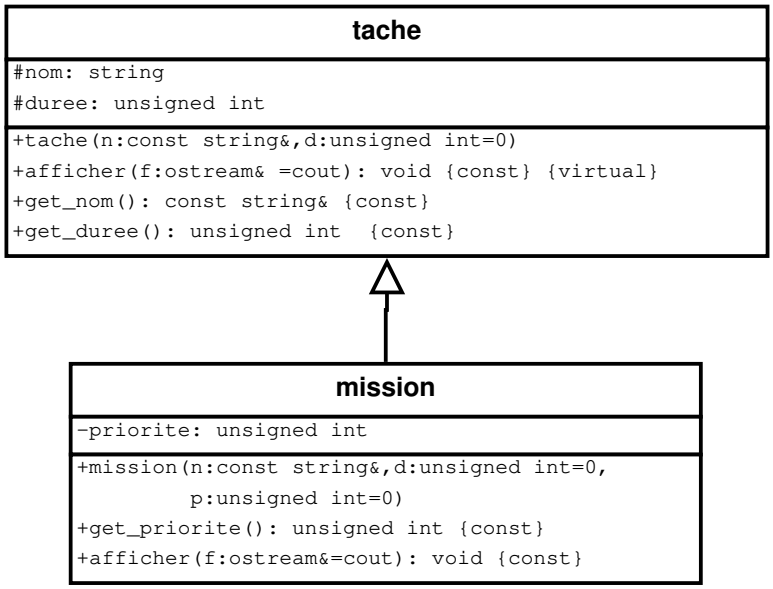
```
1 // fichier tache.h
2 class tache {
3 public:
4  ...
```

```
 virtual void afficher(ostream& f=cout) const;
 ...
};

class mission : public tache {
public:
 ...
 virtual void afficher(ostream& f=cout) const;
 ...
};

// fichier tache.cpp
void tache::afficher(ostream& f) const {
 f<<"tache "<<nom<<": duree="<<duree;
}

void mission::afficher(ostream& f) const {
 tache::afficher(f);
 f<<", priorite="<<priorite;
}
```

Même si cela ne constitue pas une erreur, le mot clé **virtual** n'a pas besoin d'être ajouté devant l'entête de la méthode dans la définition d'une classe dont `tache` serait un ancêtre (en particulier la classe `mission` : le couplage variable-méthode `afficher` est maintenant dynamique pour tous les objets d'une classe descendante de la classe `tache`.
Remarquons que la méthode `afficher` ne peut plus être **inline** car cette fonction a besoin d'une adresse pour être choisie dynamiquement : les appels à cette méthode ne peuvent plus être remplacés au cours de la compilation puisque c'est au moment de l'exécution que la méthode `afficher` de la classe `tache` ou `mission` sera choisie. En conséquence, le corps de ces méthodes doit maintenant obligatoirement se trouver dans une unité de compilation.
Maintenant, si on exécute les instructions de l'Exercice 41, le couplage est décidé au moment de l'exécution :

```
tache t("tache 1",5);
mission m("tache 2",3,99);
t.afficher(); //tache::afficher()
m.afficher(); //mission::afficher()
tache* ptt;
ptt=&t;
ptt->afficher(); //tache::afficher()
ptt=&m;
ptt->afficher(); //mission::afficher() (et non plus tache::afficher())
mission* ptm=&m;
ptm->afficher(); //mission::afficher()
tache& reft1=t;
reft1.afficher(); //tache::afficher()
tache& reft2=m;
reft2.afficher(); //mission::afficher() (et non plus tache::afficher())
mission& refm=m;
refm.afficher(); //mission::afficher()
```

2. En ajoutant le commentaire **virtual** après la méthode `afficher` sur le **modèle UML**, on indique que la méthode est polymorphe :



Solution de l'exercice 43

1. Une classe *abstraite* est une classe qui contient au moins une méthode *virtuelle pure*, c'est à dire polymorphe, mais non encore définie. Dans cet exercice, il s'agit de la méthode `afficher` . On indique qu'elle est virtuelle pure en faisant précéder son entête par **virtual** (elle est polymorphe) et en faisant suivre son entête par `=0` (son corps est nul : la méthode n'est pas définie) :

```
1 class affichable {
2 public:
3  virtual void afficher(ostream& f=cout) const=0;
4 };
```

tache.h

2. Pour que la classe `truc` soit instanciable, elle doit définir toute méthode virtuelle pure d'une classe abstraite dont elle hériterait (ici `afficher`) :

```
1 class truc : public affichable{
2  int x;
3 public:
4  void afficher(ostream& f=cout) const;
5 };
```

tache.h

```
1 void truc::afficher(ostream& f) const {
2  f<<x;
3 }
```

tache.cpp

Remarquons que si on n'exige pas d'initialiser l'attribut `x`, il n'est pas obligatoire de définir un constructeur. Le compilateur génère un constructeur sans argument qui se charge d'appeler le constructeur sans argument de la classe `affichable`. Le constructeur sans argument de la classe `affichable` existe aussi puisqu'aucun constructeur n'a été défini pour cette classe : le compilateur en a donc généré un.

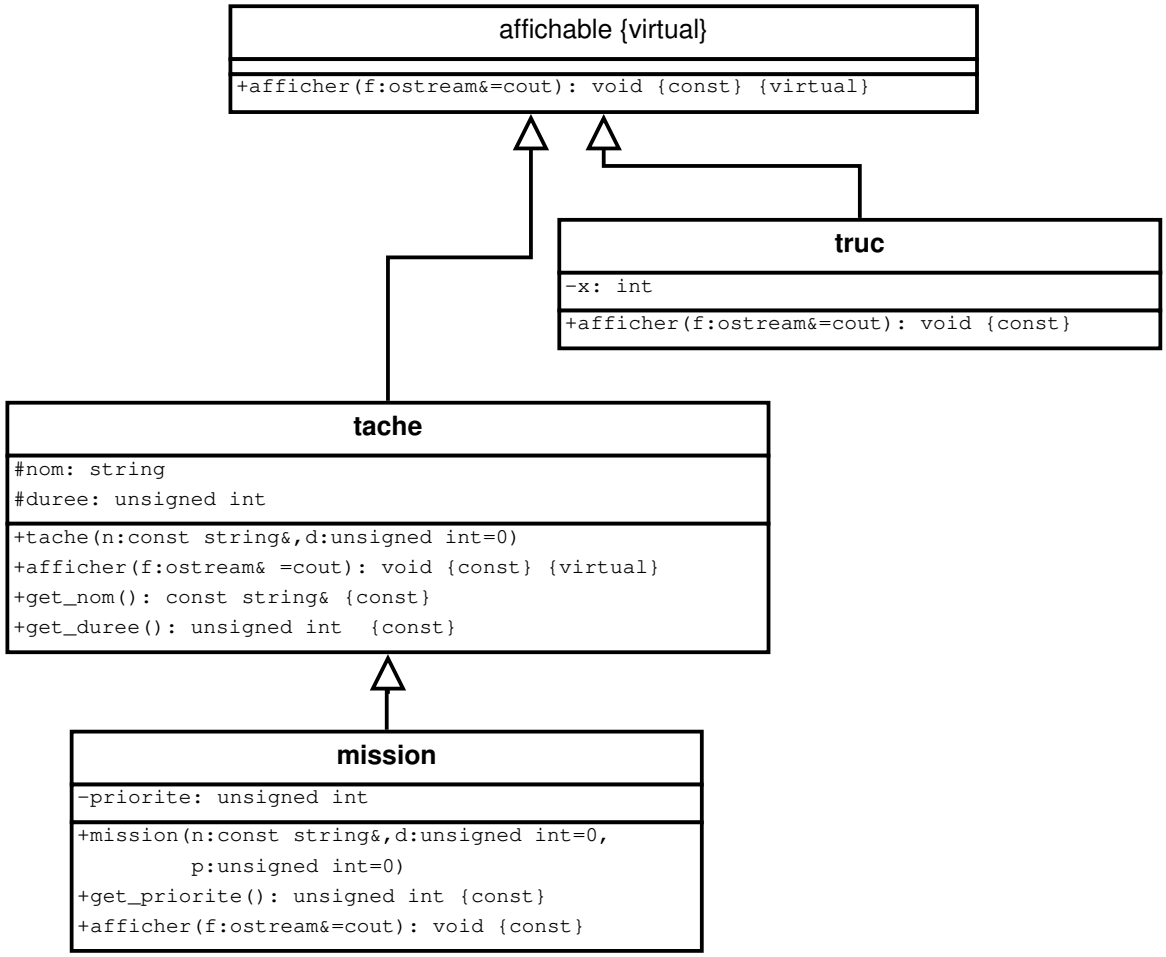
3. Pour que la classe `tache` hérite de `affichable`, il suffit de l'indiquer :

```
1 class affichable; // au moins une déclaration avant la classe tache
2 ...
3 class tache : public affichable {
4 ...
5 public:
6  void afficher(ostream& f=cout) const;
7 ...
8 };
```

tache.h

Remarquons que la classe `tache` implémente déjà la méthode `afficher`, elle reste donc instanciable (*i.e.* non abstraite). Notons aussi que le mot clé **`virtual`** qui se trouvait auparavant devant l'entête de la méthode devient redondant (la méthode `afficher` est désormais polymorphe depuis la classe `affichable` jusqu'à tous ses descendants.
Remarquons aussi que puisque le compilateur a généré un constructeur sans argument pour la classe `affichable`, le constructeur de la classe `tache` n'a pas besoin d'y faire appel explicitement. C'est pourquoi il n'est pas modifié dans ce cas.

4. En UML, le nom des classes abstraites et de leurs méthodes virtuelles pures apparaissent en italique. On peut également les faire suivre par l'indication **`virtual`** pour plus de visibilité. Le modèle UML devient alors :



Solution de l'exercice 44

Notons que la classe `affichable` n'est pas instanciable. Cependant, on peut tout à fait créer des pointeurs ou des références d'`affichable` qui pointent en réalité sur des objets de classes concrètes descendantes de la classe `affichable`. Toutes ces classes implémentent toutes la méthode `afficher`

sinon elles ne sont pas concrètes (donc abstraites) et non instanciables. Le compilateur autorise donc un appel de la méthode `afficher` à partir de pointeurs ou de références de `affichable`. Il s'agit d'une application directe du *design-pattern Template Method* où l'on utilise l'algorithme `afficher` avec un objet `affichable` dans une fonction (ici **`operator`**<<) alors que cet algorithme ne sera réellement implémenté que dans les sous-classes de la classe `affichable`.

```
1 ...
2 ostream& operator<<(ostream& f, const affichable& a);
3 ...
```

tache.h

```
1 ...
2 ostream& operator<<(ostream& f, const affichable& a){
3  a.afficher(f);
4  return f;
5 }
6 ...
```

tache.cpp

Solution de l'exercice 45

1. Pour connaître le type d'un objet, il suffit d'utiliser l'opérateur **`typeid`** sur cet objet. Il faut préalablement utiliser le fichier d'entête `typeinfo`. L'opérateur renvoie alors une valeur de type **`type_info`** sur laquelle on peut appliquer la méthode `name()` qui renvoie une chaîne de caractères représentant le type de l'objet. Notons que la valeur renvoyée par **`typeid`** ne peut pas être copiée dans une autre variable **`type_info`** : il faut donc utiliser la méthode `name()` directement sur la valeur retournée.

```
1 #include<typeinfo>
2 class affichable {
3 public:
4  ...
5  void afficher_type(ostream& f=cout) const { f<<typeid(*this).name(); }
6  ...
7 };
```

tache.h

On peut alors exécuter les instructions suivantes qui afficheront des valeurs différentes pour chaque objet `affichable` dont on veut connaître le type :

```
1 truc z;
2 tache t("x",2);
3 mission m("y",3,99);
4 affichable &a =z;
5 affichable &b =t;
6 affichable &c =m;
7 a.afficher_type(); std::cout<<"\n"; // class truc
8 b.afficher_type(); std::cout<<"\n"; // class tache
9 c.afficher_type(); std::cout<<"\n"; // class mission
```

2. On peut directement comparer avec les opérateurs == et != les valeurs de type **`type_info`** renvoyées par les appels de **`typeid`** sur les objets dont on veut comparer le type :

```
1 #include<typeinfo>
2 class affichable {
3 public:
4  ...
5  bool est_du_meme_type_que(const affichable& a) const { return typeid(*this
      )==typeid(a); }
6 };
```

tache.h

Remarquons que les méthodes `afficher_type` et `est_du_meme_type_que` ne sont pas virtuelles (ni virtuelles pures) puisque ces méthodes n'auront a priori pas besoin d'être reféfinies dans les classes descendantes de la classe `affichable`. Notons aussi alors qu'une classe abstraite peut aussi implémenter des méthodes concrètes : il suffit d'une seule méthode virtuelle pure pour en faire une classe abstraite.

Solution de l'exercice 46

1. On écrit une classe abstraite `comparable` avec la méthode virtuelle pure **`operator`**`==` :

```
1 ...
2 class comparable {
3 public:
4  virtual bool operator==(const comparable&) const=0;
5 };
6 ...
```

tache.h

La méthode ne pourra être implémentée que dans les sous-classes, où l'on connaîtra le nombre et la nature des attibuts qui sont composés par chacune de ces sous-classes.

2. Les classes `truc` et `tache` héritent maintenant aussi de la classe `comparable` en plus de la classe `affichable` . Pour rester concrètes, ces classes doivent implémenter la méthode **`operator`**`==`. La classe `mission` hérite aussi de la classe `comparable` mais par l'intermédiaire de la classe `tache` : il ne faut donc pas réécrire ce lien d'héritage. Par contre la classe `mission` doit aussi implémenter la méthode **`operator`**`==` puisqu'elle est la seule à pouvoir implémenter une méthode qui permet de comparer la valeur des attributs d'objets `mission`. On remarque que le type des paramètres ne changent pas dans l'entête de chacune de ces méthodes : il reste `comparable&`. Par exemple dans `truc::`**`operator`**`==` on ne peut pas changer le type **`const`** `comparable&` par **`const`** `truc&`. En effet, il peut s'agir d'un objet `comparable` qui n'est pas un objet `truc` (*e.g.* un objet `tache`). D'autre part, dans ce cas, il s'agirait d'une surcharge de l'opérateur == qui, de plus, n'est pas soumise au polymorphisme lorsqu'elle est appelée à partir d'une référence (ou d'un pointeur) d'objet `affichable` pointant réellement sur un objet `truc`.

```
1 ...
2 class truc : public affichable, public comparable{
3 ...
4 public:
5  bool operator==(const comparable&) const;
6 ...
7 };
8
9 class tache : public affichable, public comparable {
10 public:
11 ...
```

```
12  bool operator==(const comparable&) const;
13 ...
14 };
15
16 class mission : public tache {
17 public:
18 ...
19  bool operator==(const comparable&) const;
20 ...
21 };
```

tache.h

Il y a au moins 3 manières différentes d'implémenter la méthode **operator**== que nous illustrons dans les 3 implémentations suivantes :

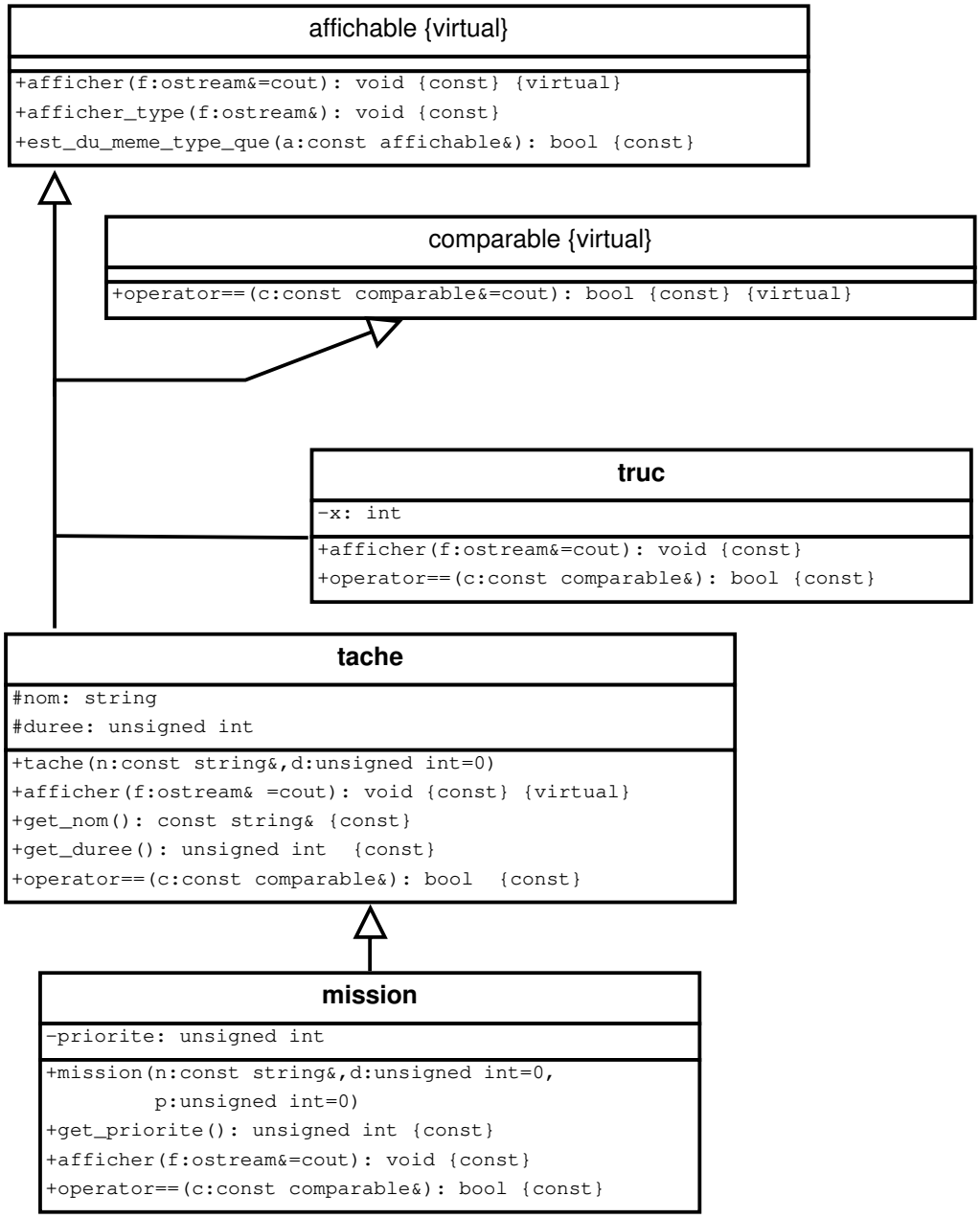
```
1  ...
2  bool tache::operator==(const comparable& c) const {
3   if (typeid(*this)!=typeid(c)) return false;
4   const tache& tmp=dynamic_cast<const tache&>(c);
5   return duree==tmp.duree && nom==tmp.nom;
6  }
7
8  bool truc::operator==(const comparable& c) const {
9   try {
10   const truc& tmp=dynamic_cast<const truc&>(c);
11   return x==tmp.x;
12  }catch(bad_cast e){
13   // rien à faire
14  }
15  return false;
16 }
17
18 bool mission::operator==(const comparable& c) const {
19  const mission* pt=dynamic_cast<const mission*>(&c);
20  if (pt==nullptr) return false;
21  return duree==pt->duree && nom==pt->nom && priorite== pt->priorite;
22 }
```

tache.cpp

Dans l'implémentation de `tache::operator==`, un test est effectué pour savoir si l'objet passé en paramètre de la méthode est aussi du type `tache` en utilisant l'opérateur == de la classe **type_info** (comme dans l'Exercice 45). En effet, dans cette méthode, l'objet ***this** est toujours du type `tache` grâce au polymorphisme. Une fois qu'il est établi que les objets sont tous les deux du type `tache`, il faut comparer la valeur de leurs attributs. Néanmoins, il est impossible d'accéder à ces attributs tant que l'objet passé en paramètre est du type **const** `comparable &`. Une conversion de **const** `comparable&` en **const** `tache&` de l'objet passé en paramètre est alors effectuée en utilisant l'opérateur **dynamic_cast**<>. Il est alors possible d'accéder aux attributs par l'intermédiaire de la référence créée.
Dans l'implémentation de `truc::operator==`, on procède légèrement différemment en tentant directement un cast de **const** `comparable&` en **const** `tache&` de l'objet passé en paramètre sans savoir si cet objet est vraiment du type `truc`. Si cet objet est vraiment du type `truc`, cela ne pose pas de problème et l'on procède comme dans l'implémentation précédente

en comparant l'attribut des deux objets. Si par contre l'objet `c` n'est pas du type `truc` alors l'exécution de cette instruction déclenche une exception de type **`bad_cast`** qui est ici directement récupérée dans la méthode. Bien que l'utilisateur de cette méthode ne se rendra pas compte du déclenchement de cette exception, le procédé est un peu lourd.
On préfèrera dans ce cas la troisième implémentation à la deuxième implémentation où, cette fois, une conversion de **`const`** `comparable*` en **`const`** `mission*` de l'adresse de l'objet passé en paramètre est tentée. Si l'objet passé en paramètre n'est pas un objet `mission` alors, l'opérateur **`dynamic_cast`**<> renvoie le pointeur nul pour indiquer que ce n'est pas possible. Il suffit alors de vérifier le résultat de la conversion avant de vérifier la valeur des attributs des objets à comparer.
Dans ces trois implémentations, une conversion d'une classe vers une classe descendante est utilisée. Dans ce cas le terme de *downcasting* (conversion vers le bas de l'arborescence) est utilisé.

3. En introduisant la classe `comparable`, le diagramme UML devient :



Solution de l'exercice 47

1. Voici une définition possible de la classe `commande` :

```
1 ...
```

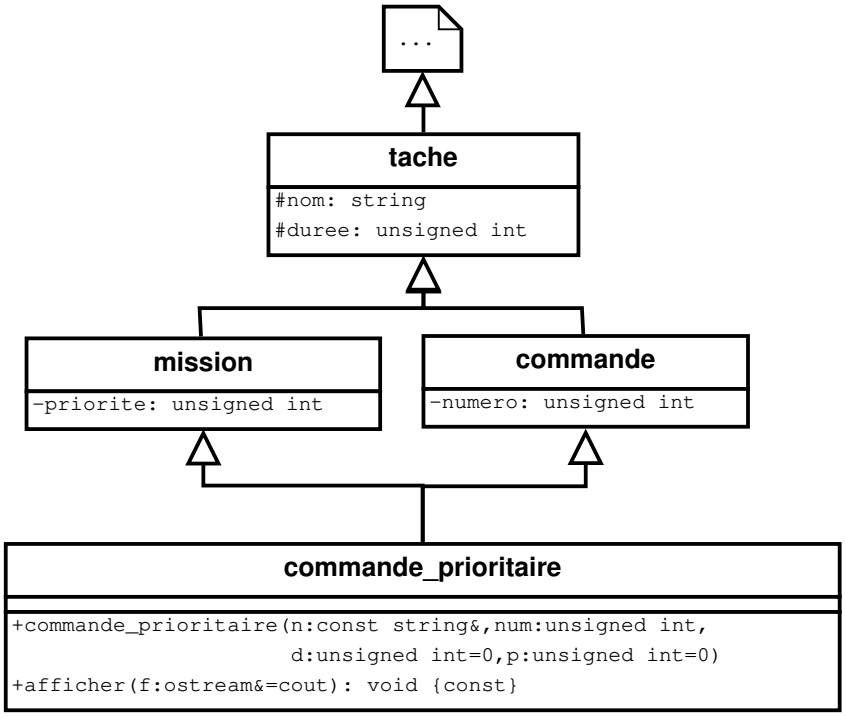
```
2  class commande : public tache {
3  public:
4   commande(const string& n, unsigned int num, unsigned int d=0):
5              tache(n,d),numero(num){}
6   unsigned int get_numero() const { return numero; }
7   void afficher(ostream& f=cout) const;
8  protected:
9   unsigned int numero;
10 };
11 ...
```

tache.h

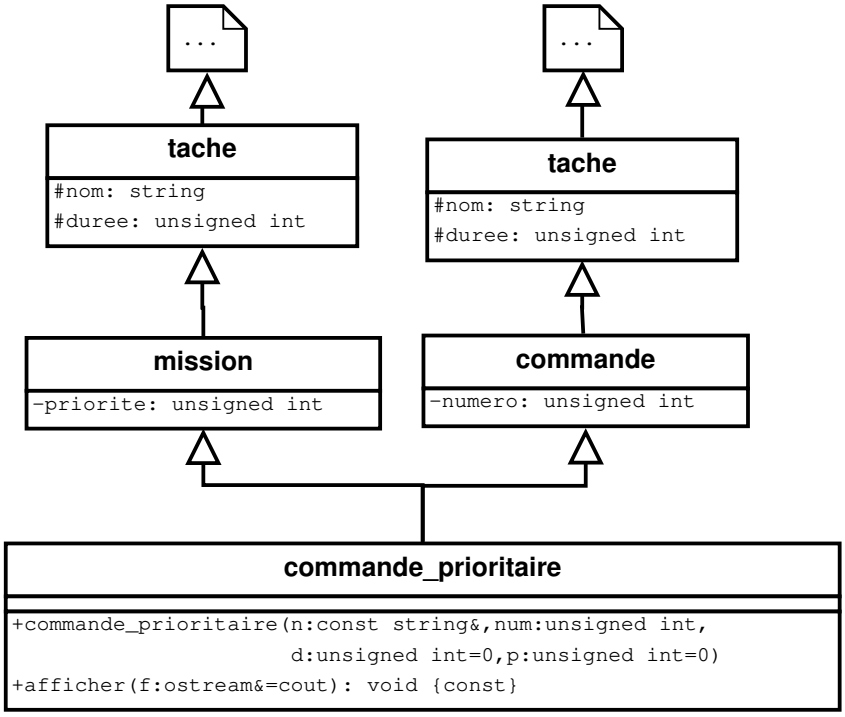
```
1  ...
2  void commande::afficher(ostream& f) const {
3   f<<"commande "<<numero<<"("<<nom<<") : duree="<<duree;
4  }
5  ...
```

tache.cpp

2. Comme l'indique le diagramme UML suivant, remarquons tout d'abord que la classe `commande_prioritaire` va hériter deux fois de la classe `tache` ; une fois par l'intermédiaire de la classe `mission`, une fois par l'intermédiaire de la classe `commande` :



En réalité, au niveau des attributs, le compilateur interprète plutôt ce double héritage de la façon suivante :



En fait, les parties `mission` et `commande` de la classe `commande_prioritaire` possèdent chacune une partie `tache`, *i.e.* les attributs de `tache` (*i.e.* `duree` et `nom`) sont dupliqués. Chacune de ces parties est accessible en utilisant l'opérateur de résolution de portée pour les distinguer. Par exemple, `mission::duree` désigne l'attribut `duree` de `tache` dans la partie `mission` de la classe `commande_prioritaire` alors que `commande::duree` désigne l'attribut `duree` de `tache` dans la partie `commande` de la classe `commande_prioritaire`.
La classe `commande_prioritaire` peut être définie comme suit :

```
1 ...
2 class commande_prioritaire : public mission, public commande {
3 public:
4  commande_prioritaire(const string& n, unsigned int num, unsigned int d=0,
       unsigned int p=0):
5              mission(n,d,p),commande(n,num,d){}
6  void afficher(ostream& f=cout) const;
7 };
8 ...
```

tache.h

```
1 ...
2 void commande_prioritaire::afficher(ostream& f) const {
3  commande::afficher(f); f<<" priorite="<<get_priorite();
4 }
5 ...
```

tache.cpp

Remarquons que le constructeur doit faire appel à un constructeur de chacune des classes dont la classe `commande_prioritaire` descend, *i.e.* `mission` et `commande`. Ces constructeurs s'occuperont de l'initilialisation de chacune des deux parties `tache` de la classe. Notons aussi dans la définition de la méthode `afficher`, l'utilisation de l'opérateur de résolution de portée `commande::` pour indiquer quelle méthode `afficher` des classes ascendantes est utilisée (si cet opérateur est omis, le compilateur indiquera une ambiguïté).
La duplication des attributs de `tache` pourrait poser des problèmes, si des accesseurs en écriture permettaient de modifier ces attributs. L'appel de telles méthodes à partir d'un objet

commande_prioritaire pourrait alors poser problème au niveau de la mise à jour et de la cohérence des informations. En effet, si un utilisateur modifie l'attribut duree de la partie mission d'un objet commande_prioritaire, on peut supposer qu'il est souhaitable dans ce contexte que l'attribut duree de la partie commande de l'objet soit aussi modifié pour être identique... ce qui n'est pas fait automatiquement. On pourrait alors ajouter une redéfinition de cet accesseur en écriture dans la classe commande_prioritaire de manière à toujours mettre à jour avec la même valeur ces deux attributs.
Remarquons qu'il n'est plus possible de créer une référence de tache directement à partir d'un objet commande_prioritaire :

```
1 commande_prioritaire cp("cp",1,23,2);
2 tache& ref_t_cp=cp; // erreur
```

En effet, cela crée une ambiguïté, puisque l'on ne peut pas savoir si cette référence doit référencer la partie tache qui se trouve dans la partie mission ou dans la partie commande de l'objet. Par contre, il est tout à fait correcte de créer une référence de mission qui référence la partie mission d'un objet commande_prioritaire et d'utiliser ensuite une référence de tache pour pointer sur cette partie.

```
1 commande_prioritaire cp("cp",1,23,2);
2 mission& refm_cp=cp;
3 tache& reft_cp=refm_cp;
4 refm_cp.afficher();
```

Notons que, grâce au polymorphisme, ce sera bien la méthode commande_prioritaire::afficher qui sera appelée dans cette dernière instruction.

3. Pour éviter la duplication des attributs de tache dans la classe commande_prioritaire, on peut utiliser l'héritage **virtual** en utilisant le mot clé **virtual** au niveau de l'héritage de mission et de commande par rapport à tache :

```
1  ...
2
3  class mission : public virtual tache {
4  ...
5  };
6
7  class commande : public virtual tache {
8  ...
9  };
10
11 class commande_prioritaire : public mission, public commande {
12 ...
13 };
```

tache.h

Notons qu'il n'y a aucune modification au niveau de l'héritage de la classe commande_prioritaire par rapport aux classes mission et commande. Par contre, si on regarde la définition du constructeur de commande_prioritaire, on peut se demander lequel des deux appels mission(n,d,p) et commande(n,num,d), qui appellent chacun le constructeur de tache, va réellement initialiser la (seule) partie tache de l'objet. La réponse est aucun des deux .... Il est en fait maintenant obligatoire pour ce constructeur de faire appel directement au constructeur de tache pour résoudre ce conflit :

```
1 ...
2 class commande_prioritaire : public mission, public commande {
3 public:
4 ...
5  commande_prioritaire(const string& n, unsigned int num, unsigned int d=0,
      unsigned int p=0):
6              tache(n,d),mission("",0,p),commande("",num,0){}
7 };
```

tache.h

Les appels du constructeur de `tache` qui ont lieu dans les constructeurs de `mission` et `commande` sont alors ignorés par le compilateur. Peu importe, donc, les valeurs qui seront transmises à ces constructeurs pour initialiser les attributs `nom` et `duree` de la classe `tache`. Ici, les valeurs `""` et `0` ont respectivement été utilisées pour illustrer ce propos.

4. Comme on peut le constater, l'héritage en losange peut poser des problèmes si la classe doublement héritée possède des attributs. Si la duplication des attributs correspond à un fait réel qui doit être pris en compte dans le modèle, le résultat est conforme à ce qui est attendu. Si par contre, les attributs ne doivent pas être dupliqués, trois solutions sont possibles :
    1. La duplication des attributs est conservée en introduisant des mécanismes qui permettent la mise à jour correcte des 2 parties. Rappelons que cette méthode n'est pas tout à fait sécurisée (voir ci-avant). De plus, si les attributs dupliqués sont gourmands en mémoire, l'application qui utilise ces classes pourrait être fortement pénalisée par cette duplication inutile.
    2. L'héritage **`virtual`** peut être utilisé. Cependant, cela demande une gestion alternative des appels de constructeurs. En effet un constructeur de la classe doublement sous-classée doit être appelé par les constructeurs de toutes les classes descendantes de cette classe. Ainsi, si la classe `commande_prioritaire` est sous-classée, le constructeur de cette nouvelle classe devra aussi faire appel au constructeur de `tache`. Cette réorganisation des appels de constructeur semble assez rebutante pour tenter d'éviter ce cas au moment de la conception.
    3. Une troisième solution qui paraît préférable est de réorganiser la hiérarchie de classe de manière à éviter ce cas. Cela facilitera l'extension future de la hiérarchie de classes et leur réutilisation.

Exceptions

Exercice 48 - Déclenchement d'une exception

Soit la classe `erreur` ci-dessous :

```
1 #include <string>
2 class erreur {
3 public:
4  erreur(const std::string& x):info(x){}
5  std::string get_info() const { return info; }
6 private:
7  std::string info;
8 };
```

Écrire une fonction `f()` qui déclenche une exception de type `erreur`.

Exercice 49 - Traitement d'une exception

Écrire une fonction `g()` qui appelle la fonction `f()` de l'Exercice 48 et qui capture l'exception déclenchée par cette fonction. Afficher le message de l'exception capturée.

Exercice 50 - Exceptions standard

Modifier la classe `erreur` de l'Exercice 48 pour la faire hériter de la classe standard `exception`. Modifier la fonction `g()` de l'Exercice 49 pour traiter l'exception déclenchée comme une exception standard.

Exercice 51 - Exceptions

Reprendre les méthodes qui permettent le mariage et le divorce entre 2 personnes dans l'exercice 25 en traitant les erreurs par déclenchement d'exceptions. Dans une fonction `main` utiliser le mécanisme de traitement des exceptions pour les capturer.

Solution de l'exercice 48

```
1 void f(){
2  throw erreur("declenchement d'une exception");
3 }
```

Une exception se déclenche en utilisant le mot clé **throw** suivi d'une valeur de n'importe quel type (ici `erreur`).

Solution de l'exercice 49

```
1 void g(){
2  try {
3   f();
4  }catch(erreur e){
5   std::cout<<"message="<<e.get_info()<<"\n";
6  }
7 }
```

Les instructions qui sont susceptibles de déclencher une exception (ici `f();`) doivent être placées dans un bloc précédé du mot clé **try**. Les exceptions déclenchées sont alors récupérées par des blocs **catch**. Chaque bloc **catch** permet de récupérer les exceptions d'un certain type et de tous ses types descendants. Ici le bloc **catch** permet de récupérer les exceptions de type `erreur`. Une fois le traitement de l'exception effectué (ici, il s'agit juste de l'affichage de l'information), l'exécution reprend après les blocs **catch**.

Solution de l'exercice 50

```
1 #include<stdexcept>
2 #include<string>
3 class erreur : public std::exception {
4 public:
5  erreur(const string& x="") throw () :info(x){}
6  ~erreur() throw() {}
7  std::string get_info() const { return info; }
8  const char* what() const throw() { return info.c_str(); }
9 private:
10  std::string info;
11 };
12
13 void f(){
14  throw erreur("declenchement d'une exception");
15 }
16
17 void g(){
18  try {
19   f();
20  }catch(std::exception& e){
21   std::cout<<"message="<<e.what()<<"\n";
22  }
```

```
23 }
```

On peut utiliser les exceptions standards, déclarées dans le fichier d'entête `stdexcept`, en faisant hériter la classe `erreur` de la classe `exception`. Il faut pour cela redéfinir la méthode **`const char*`** `what()` **`const throw()`** qui est censée fournir une chaîne de caractères indiquant la nature de l'exception (notons que **`throw()`** indique que cette méthode ne doit pas déclencher d'exception). Un bloc **`catch`** qui récupère une exception de type `std::exception` récupère aussi toutes les exceptions du type d'une classe descendante (ici la classe `erreur`). On peut alors appliquer la méthode `what()` à l'objet `exception` obtenu. Grâce au polymorphisme, c'est la méthode `erreur::what()` qui sera appelée ici.

Solution de l'exercice 51

```
1  ...
2  class Personne {
3   ...
4   void mariage_avec(const Personne& p){
5    if (conjoint==nullptr && p->conjoint==nullptr){
6     conjoint=&p;
7     statut=marie;
8     p->conjoint=this;
9     p->statut=marie;
10   }else throw<<"mariage impossible";
11  }
12  void divorce(){
13   if (conjoint!=nullptr){
14    conjoint->conjoint=nullptr;
15    conjoint->situation_familial=celibataire;
16    conjoint=nullptr;
17    situation_familiale=celibataire;
18   }else throw<<"erreur divorce: conjoint inexistant";
19 };
```

personne.h

```
1  void main(){
2   Personne* p1=new Personne("Einstein","Albert",1879,homme,celibataire);
3   Personne* p2=new Personne("Monroe","Marilyne",1926,femme,celibataire);
4   Personne* p2=new Personne("Miller","Arthur",1915,homme,celibataire);
5   try{
6    p2->divorce();
7   }catch(const char* e){
8    std::cout<<e<<"\n";
9   }
10  p1->mariage(*p2);
11  try{
12   p2->mariage(p3);
13  }catch(const char* e){
14   std::cout<<e<<"\n";
15  }
16  delete p1;
```

```
17  try{
18   p2->divorce();
19  }catch(const char* e){
20   std::cout<<e<<"\n";
21  }
22  try{
23   p2->mariage(p3);
24  }catch(const char* e){
25   std::cout<<e<<"\n";
26  }
27  delete p2;
28  delete p3;
29 }
```

dans une unité de compilation .cpp

Programmation générique

Exercice 52 - Patron de fonction

1. Écrire un patron de fonction qui permet de faire la somme de 3 variables du même type numérique. On supposera pour cela que, pour ce type, le constructeur de recopie est accessible et que l'opérateur binaire + a été défini.
2. Définir une fonction qui utilise ce patron.

Exercice 53 - Patron de fonction avec plusieurs paramètres de type

1. Refaire l'exercice précédent de manière à ce que le patron de fonction puisse permettre de faire la somme de 3 variables de types numériques différents. On supposera pour cela que le constructeur de recopie et l'opérateur binaire + ont été définis pour ces types. On supposera aussi qu'il existe une conversion possible entre chacun de ces types. Le résultat renvoyé sera du type du premier paramètre de la fonction (induisant une éventuelle perte).
2. Définir une fonction qui utilise ce patron.

Exercice 54 - Patron de classe

Définir un patron de classe `couple` contenant deux attributs de deux types, éventuellement différents, inconnus au moment de l'écriture du patron. L'un des attributs s'appellera `premier` et l'autre `second`.

Exercice 55 - Constructeur d'un patron de classe

1. Définir un constructeur **`inline`** pour le patron de classe `couple` de l'Exercice 54. Ce patron aura deux paramètres correspondant aux valeurs qui serviront à initialiser les attributs.
2. Refaire la question précédente mais en définissant le constructeur en dehors de la définition de la classe.
3. Créer une fonction dans laquelle un objet `couple` est créé avec une valeur de type `string` et une valeur de type **`double`**.

Exercice 56 - Méthode de patron de classe

Définir des accesseurs en lecture **`inline`** pour chacun des attributs de la classe `couple` des Exercices 54 et 55. Déclarer des accesseurs en écriture pour chacun des attributs et les définir en dehors de la définition de la classe.

Exercice 57 - Diagramme UML d'un patron de classe

Dessiner le diagramme UML de classe `couple` des Exercices 54, 55 et 56.

Exercice 58 - Fonction utilisant un objet d'une classe paramétrée

Définir une fonction qui permet d'afficher un objet `couple` (paramétré avec n'importe quels types) sur un flux `ostream` en utilisant l'opérateur <<. On supposera pour cela que, pour chacun des types utilisés pour paramétrer le patron, l'opérateur << a été surchargé de façon à pouvoir afficher une valeur de ce type sur un flux `ostream`.

Exercice 59 - Argument par défaut pour une méthode de patron de classe

Ajouter des arguments par défaut au constructeur de `couple` défini dans l'Exercice 55, de façon à pouvoir construire un objet `couple` avec 1 ou 0 argument.

Exercice 60 - Valeur de retour paramétrée pour une méthode de patron de classe

Définir une méthode `get_valeur` qui renvoie une copie de l'objet qui appelle la méthode.

Exercice 61 - Patron de méthode de classe patron

1. Définir une méthode **`operator`**== **`inline`** qui permet de comparer les valeurs d'attributs d'un objet `couple` avec un autre objet `couple` qui a éventuellement des paramètres de type différents. On supposera qu'il existe une conversion entre les types différents utilisés.
2. Refaire la question précédente mais en définissant la méthode en dehors de la classe (méthode non-**`inline`**).

Solution de l'exercice 52

1. Le patron `somme` peut s'écrire de la manière suivante dans un fichier d'entête :

```
1 template<class T>
2 T somme(const T& x, const T& y, const T& z){
3  cout<<"type="<<typeid(T).name()<<endl;;
4  T res=x+y+z;
5  return res;
6 }
```

patron.h

On indique la création d'un patron en utilisant le mot clé **`template`**. Les différents paramètres du patron (c'est à dire les types qui ne sont pas encore connus au moment de l'écriture du patron) sont listés entre chevrons (<>). Le mot clé **`class`** ou **`typename`** est utilisé devant chaque paramètre de type. Le seul paramètre de type de cette fonction s'appelle `T`.
Une fois que `T` est déclaré comme étant un paramètre de type, il peut être utilisé dans n'importe quelle situation où l'on peut utiliser habituellement un type (définition de variable, argument de **`sizeof`** ou de **`typeid`**, ...). Il est ici utilisé pour définir les paramètres de la fonction et une variable `res` qui sert à stocker le résultat de la somme des arguments qui seront passés à la fonction.
Notons que `T` ne peut désigner qu'un type pour lequel l'opérateur binaire `+` a été défini et pour lequel le constructeur de recopie est accessible. Dans le cas contraire, la ligne 4 du code précédent ne pourrait pas être compilée.
Notons aussi qu'un patron de fonction (ou de classe) doit s'écrire dans un fichier d'entête si on veut le rendre disponible dans plusieurs unités de compilation. En effet, tout comme une fonction **`inline`**, un patron de fonction ne peut pas être précompilé : le compilateur doit pouvoir fabriquer une instance du patron, *i.e.* une fonction réelle, pour chaque argument de type qui est fourni au patron.

2. Le patron ayant été défini, on peut l'instancier en l'utilisant directement :

```
1 #include <iostream>
2 #include "patron.h"
3 using namespace std;
4 void exemple_patron(){
5  int a=3, b=5, c=-7;
6  cout<<somme(a,b,c)<<"\n";
7 }
```

patron.cpp

Remarquons que le paramètre de type `T` est déduit directement par le compilateur grâce au type des variables `a`, `b` et `c` qui sont passées en argument. Il doit y avoir exacte correspondance entre tous les arguments : il n'y a pas de conversion possible. Ici, la définition du patron exige que le type soit le même pour tous les arguments passés à la fonction (ici **`int`**). Si une des trois variables n'est pas du même type que les deux autres, le compilateur refusera de compiler le code :

```
1 ...
2 void exemple_patron(){
3  int a=3, b=5;
4  double c=4;
5  cout<<somme(a,b,c)<<"\n"; //erreur
6 }
```

patron.cpp

On peut néanmoins contourner ce problème en forçant l'instanciation du paramètre de type avec un type fourni par l'utilisateur entre chevrons :

```
1 ...
2 void exemple_patron(){
3  int a=3, b=5;
4  double c=4;
5  cout<<somme<int>(a,b,c)<<"\n"; //ok, la valeur de c sera convertie en int
       pour être stockée dans le paramètre z (de type int) de la fonction.
6  cout<<somme<double>(a,b,c)<<"\n"; //ok, les valeurs de a et b seront
       converties en double pour être stockées dans les paramètres x et y (de
        type double) de la fonction.
7 }
```

patron.cpp

Notons aussi que, puisqu'une constante entière est par défaut de type **int**, le code suivant est aussi correct :

```
1 ...
2 void exemple_patron(){
3  cout<<somme(3,-2,6)<<"\n"; //ok, T sera instancié avec int
4 }
```

patron.cpp

De même, puisqu'une constante flottante est par défaut du type **double**, le code suivant pourra aussi être compilé :

```
1 ...
2 void exemple_patron(){
3  cout<<somme(3.2,-2.5,6.)<<"\n"; //ok, T sera instancié avec double
4 }
```

patron.cpp

Notons néanmoins que l'on doit écrire au moins "`6.`" (ou "`6.0`") pour que le code soit correct, car `6` serait considérée comme une constante de type **int**.

Solution de l'exercice 53

1. Pour que l'on puisse traiter des types numériques différents, le patron `somme` peut s'écrire de la manière suivante :

```
1 template<class T1, class T2, class T3>
2 T1 somme(const T1& x, const T2& y, const T3& z){
3  T1 res=x+y+z;
4  // code optionnel pour connaître T1, T2 et T3 :
5  cout<<"T1="<<typeid(T1).name()<<"\n";
6  cout<<"T2="<<typeid(T2).name()<<"\n";
7  cout<<"T3="<<typeid(T3).name()<<"\n";
8  return res;
9 }
```

patron.h

Pour cela on indique un paramètre de type pour chacun des paramètres de la fonction. Afin de connaître avec quels types `T1`, `T2` et `T3` seront instanciés, nous avons ajouté l'utilisation de l'opérateur **typeid**.

2. Le patron ayant été défini, on peut l'instancier en l'utilisant directement comme avant :

```
#include <iostream>
#include "patron.h"
using namespace std;
void exemple_patron(){
 int a=3, b=5, c=-7;
 cout<<somme(a,b,c)<<"\n";
}
```

patron.cpp

L'affichage montrera que tous les paramètres de type (*i.e.* `T1`, `T2` et `T3`) ont été instanciés avec **`int`**. Maintenant, si on utilise des types différents, cela ne pose plus de problème (contrairement à l'Exercice 52) :

```
...
void exemple_patron(){
 int a=3, b=5;
 double c=4;
 cout<<somme(a,b,c)<<"\n"; // ok, T1=int, T2=int, T3=double
}
```

patron.cpp

Solution de l'exercice 54

Tout comme un patron de fonction (voir Exercice 52), on utilise le mot clé **`template`** suivi entre chevrons de la liste des paramètres de la classe (ici `X` et `Y`). On peut alors utiliser ces paramètres dans la classe comme n'importe quel type existant. Ici, les paramètres sont utilisés pour définir les deux attributs `premier` et `second`.

```
template<class X, class Y>
class couple {
private:
 X premier;
 Y second;
};
```

couple.h

Solution de l'exercice 55

1. Les paramètres de la classe peuvent aussi être utilisés comme type des paramètres d'une méthode. Les arguments du constructeur sont ici utilisés pour initialiser les attributs :

```
template<class X, class Y>
class couple {
public:
 couple(const X& x, const Y& y):premier(x),second(y){}
private:
 X premier;
 Y second;
};
```

couple.h

2. Si le constructeur est défini en dehors de la définition de la classe, il faut indiquer au compilateur que la fonction est une méthode de la classe `couple` paramétrée avec les types `X` et `Y` en utilisant l'opérateur de résolution de portée `couple<X,Y>::`. Il faut préalablement redéclarer les paramètres de type avec le mot clé **`template`**.

```
1  template<class X, class Y>
2  class couple {
3  public:
4   couple(const X& x, const Y& y);
5  private:
6   X premier;
7   Y second;
8  };
9
10 template<class X, class Y>
11 couple<X,Y>::couple(const X& x, const Y& y):premier(x),second(y){}
```

couple.h

Il n'est pas obligatoire (bien que conseillé pour des questions de lisibilité) d'utiliser les mêmes noms de paramètre de type. Ainsi le code suivant serait aussi tout à fait correct :

```
1  template<class X, class Y>
2  class couple {
3  public:
4   couple(const X& x, const Y& y);
5  private:
6   X premier;
7   Y second;
8  };
9
10 template<class A, class B> // possible mais moins lisible
11 couple<A,B>::couple(const A& x, const B& y):premier(x),second(y){}
```

couple.h

3. Dans la fonction suivante on crée un objet `couple` dans lequel le paramètre `X` est instancié avec le type `string` et le paramètre `Y` avec le type **`double`** :

```
1  #include "couple.h"
2  void exemple(){
3   couple<string,double> c1("pi",3.14159);
4  }
```

couple.cpp

Notons qu'il est obligatoire de préciser les paramètres de type (entre chevrons juste après `couple`). Contrairement aux patrons de fonction, le compilateur n'est pas en mesure, la plupart du temps, de déduire le type des paramètres.

Solution de l'exercice 56

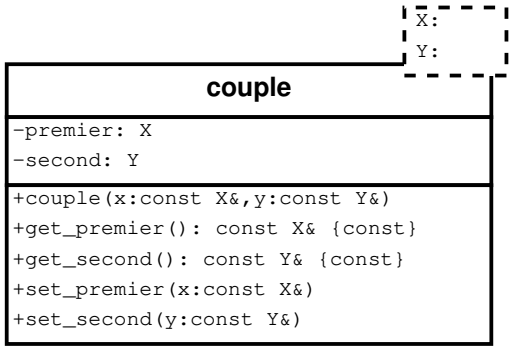
Des méthodes (**`inline`** ou non) peuvent être définies de la même manière que le constructeur. En cas de définition en dehors de la classe, il faut aussi utiliser la redéclaration des paramètres de type et l'opérateur de résolution de portée (voir Exercice 55).

```
1  template<class X, class Y>
2  class couple {
3  public:
4   couple(const X& x, const Y& y):premier(x),second(y){}
5   const X& get_premier() const { return premier; }
6   const Y& get_second() const { return second; }
7   void set_premier(const X& x);
8   void set_second(const Y& y);
9  private:
10  X premier;
11  Y second;
12 };
13
14 template<class X, class Y>
15 void couple<X,Y>::set_premier(const X& x){ premier=x; }
16
17 template<class X, class Y>
18 void couple<X,Y>::set_second(const Y& y){ second=y; }
```

couple.h

Solution de l'exercice 57

Un patron de classe en UML, se dessine comme une classe classique. Les noms des paramètres doivent cependant être indiqués en haut à droite de la classe dans un rectangle en traits d'union :



Solution de l'exercice 58

Il est tout à fait possible d'utiliser dans une fonction un objet d'une classe paramétrée, à condition que la fonction soit elle-même une fonction paramétrée avec au moins les paramètres nécessaires pour instancier l'objet. Ici la fonction **operator**<< est aussi paramétrée avec les types `X` et `Y` (ils peuvent tout à fait être renommés) qui servent à paramétrer l'objet `couple` en deuxième paramètre de la fonction.

```
1  template<class X, class Y> class couple { ... };
2  ...
3  template<class X, class Y>
4  ostream& operator<<(ostream& f, const couple<X,Y>& c) {
5   f<<"["<<c.get_premier()<<";"<<c.get_second()<<"]";
6   return f;
7  }
```

couple.h

```
#include "couple.h"
void exemple(){
 couple<string,double> c1("pi",3.14159);
 cout<<"c1="<<c1<<"\n";
}
```

couple.cpp

Notons que, puisque **operator**<< est une fonction, c'est le compilateur qui déduit dans l'exemple qu'il faut paramétrer X avec string et Y avec **double**.

Solution de l'exercice 59

Si l'utilisateur de la classe ne fournit pas de valeur pour définir un objet couple, on peut supposer qu'il existe des constructeurs sans argument qui permettent de construire des objets de types correspondants aux paramètres X et Y. Ainsi la valeur X(), par exemple, représente un objet X que l'on peut construire sans argument. Les objets X() et Y() peuvent alors servir d'arguments par défaut pour le constructeur :

```
template<class X, class Y> class couple {
public:
 couple(const X& x=X(), const Y& y=Y());
...
};
...
template<class X, class Y>
couple<X,Y>::couple(const X& x, const Y& y):premier(x),second(y){}
```

couple.h

Il est alors possible de construire des objets couple avec 0, 1 ou 2 arguments :

```
#include "couple.h"
void exemple(){
 couple<string,double> c1("pi",3.14159);
 couple<string,double> c2("hello");
 couple<string,double> c3;
 cout<<"c1="<<c1<<"\n"<<"c2="<<c2<<"\n"<<"c3="<<c3<<"\n";
}
```

couple.cpp

Remarquons que dans la construction de l'objet c3, le compilateur serait incapable d'instancier tout seul les paramètres X et Y s'ils n"étaient pas indiqués explicitement. Cela explique pourquoi les paramètres doivent toujours être précisés lors de l'instanciation d'un patron de classe.

Solution de l'exercice 60

```
template<class X, class Y>
class couple {
public:
...
 couple<X,Y> get_valeur() const { return *this; }
};
...
```

couple.h

Solution de l'exercice 61

1. Il est tout à fait possible de définir un patron de méthode à l'intérieur d'un patron de classe. Cela se fait de la même manière qu'une fonction avec le mot clé **template** :

```
1  template<class X, class Y>
2  class couple {
3  public:
4  ...
5   template<class T1, class T2> bool operator==(const couple<T1,T2>& c) const
       {
6    return premier==c.get_premier()&& second==c.get_second();
7   }
8  ...
9  };
10 ...
```

couple.h

2. Si la définition de cette méthode se fait à l'extérieur de la classe, il faut indiquer d'abord les paramètres de la classe (qui serviront à l'opérateur de résolution de portée), puis réutiliser **template** pour déclarer les paramètres supplémentaires spécifiques à la méthode :

```
1  template<class X, class Y>
2  class couple {
3  public:
4  ...
5   template<class T1, class T2> bool operator==(const couple<T1,T2>& c) const
      ;
6  ...
7  };
8  ...
9  template<class X, class Y> template<class T1, class T2>
10 bool couple<X,Y>::operator==(const couple<T1,T2>& c) const{
11  return premier==c.get_premier()&& second==c.get_second();
12 }
```

couple.h

Patrons de conception (design patterns)

Exercice 62 - Design pattern “Adapter”

On dipose de la classe `AleatoireIntGenerateur` définie dans le fichier suivant :

```
#include <string>
#include <cmath>
class AleatoireIntGenerateur{
 double random() const {
  static unsigned int rootA=27134;
  static unsigned int rootB=9213;
  static unsigned int rootC=17773;
  rootA= (rootA*171)%30269;
  rootB= (rootB*172)%30307;
  rootC= (rootC*170)%30323;
  double x= rootA/30269.0 + rootB/30307.0 + rootC/30323.0;
  return x-floor(x); // retourne la partie décimale (entre 0 et 1)
 }
 std::string name;
public:
 AleatoireIntGenerateur(const std::string& n=""):name(n) { }
 unsigned int GetInt(unsigned int max) const {
  double x=random()*max;
  if (x-floor(x)>0.5) return static_cast<int>(ceil(x));
  else return static_cast<int>(floor(x));
 }
};
```

AleatoireIntGenerateur.h

La méthode `random()` utilise 3 entiers **`static`** `rootA`, `rootB` et `rootC`. Chaque fois que cette méthode est appelée, ces trois valeurs sont modifiées avec un calcul spécifique. Un réel de type **`double`** de valeur comprise entre 0 et 1 est alors généré en extrayant la partie décimale d'un nombre obtenu à partir de `rootA`, `rootB` et `rootC`. La méthode `GetInt(max)`, qui utilise la méthode `random()`, permet alors de générer un entier entre `0` et `max`, où `max` est une valeur de type **`unsigned int`** donnée. Notons que :

— Les fonctions `floor` et `ceil` permettent respectivement d'obtenir le plus grand entier inférieur et le plus petit entier supérieur à un nombre réel donné.

— Il n'est absolument pas nécessaire de comprendre ce calcul pour faire l'exercice.

Soit la classe abstraite suivante :

```
class RandomGenerator {
protected:
 unsigned int max;
public:
 unsigned int GetMax() const { return max; }
 RandomGenerator(int m):max(m){}
 // renvoie un nombre aléatoire de {0,...,GetMax()}
 virtual unsigned int GetNumber() const=0;
```

```
};
```

RandomGenerator.h

Le constructeur de cette classe prend en argument une valeur `m` qui indique que le générateur permettra de générer des nombres entre `0` et `m`.
Vous développez le nouveau logiciel de simulation **SIMULA3000** dans lequel vous utilisez un objet générateur de nombres pseudo-aléatoires représenté par une variable de type pointeur sur la classe abstraite `RandomGenerator` :

```
class Simula3000 {
 RandomGenerator* gen;
public:
 Simula3000():gen(nullptr){}
 void set_generateur(RandomGenerator& G) { gen=&G; }
};
```

SIMULA3000.h

En utilisant le design pattern "Adapter", développer une nouvelle classe appelée `RandomIntGenerator` qui adapte la classe `AleatoireIntGenerateur` de telle sorte qu'un objet instanciant cette classe puisse être utilisé comme générateur de nombres pseudo-aléatoires dans `SIMULA3000` (en utilisant la méthode `set_generateur`).
Faire cette question deux fois. Une fois en utilisant un "**adaptateur de classe**", une fois en utilisant un "**adaptateur d'objet**".

Exercice 63 - Design pattern "Singleton"

Transformer la classe `AleatoireIntGenerateur` de l'exercice précédent pour en faire une classe "singleton".

Exercice 64 - Design pattern "Iterator"

1. Compléter la classe `vecteur` des exercices 26-35 en implémentant une classe d'itérateur `vecteur::iterator` qui permet de parcourir séquentiellement les différents éléments d'un objet `vecteur` avec le code suivant :

```
vecteur v(10);
/* ... */
for(vecteur::iterator it=v.get_iterator(); !it.is_done() ; it.next())
   std::cout<<it.current_item()<<"\n";
```

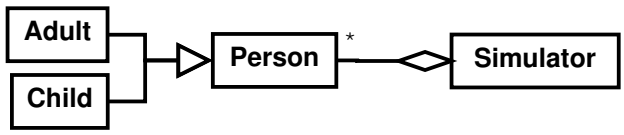
2. Implémenter une classe d'itérateur `vecteur::const_iterator` qui permet de parcourir séquentiellement les différents éléments d'un objet `vecteur` mais qui offre un accès uniquement en lecture aux éléments.
3. Refaire les deux questions précédentes en proposant une interface d'itérateur similaire à celle utilisée par les conteneurs standards du C++ (STL), *i.e.* qui permet de parcourir séquentiellement les différents éléments d'un objet `vecteur` avec le code suivant :

```
vecteur v(10);
/* ... */
for(vecteur::iterator it=v.begin(); it!=v.end() ; ++it){
   *it=*it+1;
   std::cout<<*it<<"\n";
}
for(vecteur::const iterator it=v.begin(); it!=v.end() ; ++it){
   std::cout<<*it<<"\n";
```

```
9 }
```

Exercice 65 - Les Sims

On travaille sur un simulateur de population déjà développé et que l'on souhaite améliorer. Le simulateur peut simuler deux types de personnages : adultes (`Adult`) et enfants (`Child`). Les classes du diagramme ci-contre sont déjà implémentées.



Par ailleurs, voici un extrait du code de la classe Simulator :

```
1  class Simulator {
2    std::vector<Person*> population;
3  public :
4   Simulator(const string& type, const unsigned int& nb){
5    if(type=="Adult")
6     for(unsigned int i=0; i< nb ;i++) population.push_back(new Adult(i));
7    if(type=="Child")
8     for(unsigned int i=0; i< nb ;i++) population.push_back(new Child(i));
9   }
10 };
11 int main() { Simulator  *s = new Simulator("Adult",100); return 0; }
```

On souhaite vendre ce simulateur et permettre aux utilisateurs de programmer leurs personnages spécifiques. Après modification, les utilisateurs de la classe `Simulator` n'auront plus accès au code source de cette classe.

1. En conception orientée objet, quel est le nom du design pattern qui pourrait être utilisé pour permettre cette fonctionnalité ? Proposer une solution sous la forme d'un diagramme de classe.

2. Implémenter les nouvelles classes ajoutées et donner la nouvelle définition de la classe Simulator.

Solution de l'exercice 62

```
1  #include "SIMULA3000"
2  namespace OBJECT_ADAPTER{
3   class RandomIntGenerator : public RandomGenerator{
4   private :
5    AleatoireIntGenerateur G;
6   public:
7    RandomIntGenerator(unsigned int m):RandomGenerator(m),G("Simula3000"){}
8    unsigned int GetNumber() const { return G.GetInt(max); }
9   };
10 }
11 namespace CLASS_ADAPTER{
12  class RandomIntGenerator : public RandomGenerator, private
       AleatoireIntGenerateur{
13  public:
14   RandomIntGenerator(unsigned int m):RandomGenerator(m),AleatoireIntGenerateur("
       Simula3000"){}
15   unsigned int GetNumber() const { return GetInt(max); }
16  };
17 }
```

RandomIntGenerator.h

Dans l'adaptateur d'objet, un attribut `G` de type `AleatoireIntGenerateur` est encapsulé dans la classe `RandomIntGenerator`. La méthode `GetNumber()` fait alors appel à la méthode `GetInt()` sur `G`. Dans l'adaptateur de classe, la classe `RandomIntGenerator` hérite à la fois de `RandomGenerator` (décrivant l'interface "objectif") et de la classe `AleatoireIntGenerateur` (la classe à adapter). Remarquons alors, l'utilisation de l'héritage **`private`** pour que les utilisateurs de la classe `RandomIntGenerator` n'aient plus accès à l'interface de la classe `AleatoireIntGenerateur` qui doit être masquée.

Solution de l'exercice 63

```
1  #include <string>
2  #include <cmath>
3  class AleatoireIntGenerateur{
4  private:
5   double random() const { /* ... */ }
6   std::string name;
7   /* pointeur sur l'unique instance la classe */
8   static AleatoireIntGenerateur* instance;
9   /* les constructeurs doivent être maintenant privés afin d'empêcher l'
       instantiation de la classe */
10  AleatoireIntGenerateur(const std::string& n=""):name(n){}
11  /* cela vaut aussi pour le constructeur de recopie qui serait sinon généré
       automatiquement par le compilateur dans la partie publique de la classe */
12  AleatoireIntGenerateur(const AleatoireIntGenerateur&){}
13  /* le destructeur devrait aussi être private pour empêcher son appel autrement
       que par l'intermédiaire de liberer_instance() */
14  ~AleatoireIntGenerateur(){}
15 public:
16  static AleatoireIntGenerateur& get_instance() {
```

```
17   if (!instance) instance=new AleatoireIntGenerateur("Instance unique");
18   return *instance;
19  }
20  static void liberer_instance() { if (instance) delete instance; }
21  unsigned int GetInt(unsigned int max) const { /* ...*/ }
22 };
```

AleatoireIntGenerateur.h

```
1 #include "AleatoireIntGenerateur.h"
2 /* tout membre static d'une classe doit être définie dans un fichier source et
     initialisé */
3 AleatoireIntGenerateur* AleatoireIntGenerateur::instance=nullptr;
```

AleatoireIntGenerateur.cpp

Cette classe utilise un membre **static** de type pointeur sur cette même classe. Ce pointeur sera chargé de contenir l'adresse de l'unique instance de la classe. Cette instance sera créée dynamiquement lors du premier appel à la méthode `get_instance()`. Les méthodes `get_instance()` et `liberer_instance` doivent être **static** afin de pouvoir les utiliser sans instanciation d'objet (qui n'est ici pas possible). Le code suivant montre comment avoir accès à l'unique instance :

```
1 #include "AleatoireIntGenerateur.h"
2 void f(){
3 AleatoireIntGenerateur& G=AleatoireIntGenerateur::get_instance();
4 /* ... */
5 AleatoireIntGenerateur::liberer_instance()
6 }
```

AleatoireIntGenerateur.cpp

Solution de l'exercice 64

1.

```
1  /* ... */
2  class vecteur {
3  public:
4  /* ... */
5   class iterator {
6    friend class vecteur;
7    double* courant;
8    double* fin;
9    iterator(double* deb, double* f):courant(deb), fin(f){}
10  public:
11   void next() {
12    if (courant==fin) throw "erreur : debordement d'iterator";
13    ++courant;
14   }
15   bool is_done() { return courant==fin; }
16   double& current_item() { return *courant; }
17  };
18  iterator get_iterator() { return iterator(tab, tab+dim); }
19 };
```

vecteur.h

Pour que le type `vecteur::iterator` puisse exister, il faut que la classe `iterator` soit définie à l'intérieur de la définition de la classe `vecteur`.
Ici, le constructeur a été placé dans la partie privée et la classe `vecteur` a été déclarée amie de la classe `vecteur::iterator`. Ainsi, seule la méthode `vecteur::get_iterator()` permettra d'obtenir un itérateur.
La classe `iterator` maintient deux attributs de type **`double*`**. L'attribut `courant` pointe sur l'élément du vecteur désigné par l'itérateur. L'attribut `fin` pointe sur l'adresse mémoire qui suit le dernier élément du vecteur. Lorsqu'un objet `vecteur::iterator` a parcouru tous les éléments d'un objet `vecteur`, on a alors `courant==fin`. Cela peut être testé avec la méthode `is_done()`. La méthode `next()` permet de se déplacer sur l'élément suivant (en incrémentant le pointeur `courant`. Enfin, la méthode `current_item()` renvoie une référence sur l'objet que désigne l'itérateur. Notons que cette référence peut être utilisée en lecture et en écriture pour modifier l'élément.

2. Si on souhaite offrir un accès séquentiel aux différents éléments d'un vecteur mais uniquement en lecture, on peut implémenter en plus, la classe `const_iterator` :

```
1  /* ... */
2  class vecteur {
3  public:
4  /* ... */
5   class iterator {
6    /* ...*/
7   };
8   iterator get_iterator() { return iterator(tab, tab+nb); }
9
10  class const_iterator {
11   friend class vecteur;
12   const double* courant;
13   const double* fin;
14   const_iterator(const double* deb, const double* f):
15     courant(deb), fin(f){}
16  public:
17   void next() {
18    if (courant==fin) throw "erreur : debordement d'iterator";
19    ++courant;
20   }
21   bool is_done() { return courant==fin; }
22   const double& current_item() { return *courant; }
23  };
24  const_iterator get_iterator() const {
25   return const_iterator(tab, tab+dim);
26  }
27 };
```

vecteur.h

La classe `const_iterator` est très similaire à la classe `iterator` si ce n'est que les attributs sont de type **`const double*`** qui n'autorise pas la modification des éléments pointés par ces pointeurs. Dès lors, la méthode `current_item()` doit renvoyer une référence **`const`**. Notons que la méthode `vecteur::get_iterator()` **`const`** renvoie un objet de type `const_iterator`. Ainsi, à partir d'un objet `vecteur` constant ou d'une référence **`const`** de `vecteur`, l'utilisateur n'a accès qu'à un parcours séquentiel en lecture.

3. Voici une manière d'implémenter le design pattern iterator qui utilise la même interface que les itérateurs des classes standards du C++ :

```
1  /* ... */
2  class vecteur {
3  public:
4  /* ... */
5  class iterator {
6    friend class vecteur;
7    double* courant;
8    iterator(double* deb):courant(deb){}
9   public:
10   iterator():courant(nullptr){}
11   double& operator*() const { return *courant; }
12   iterator& operator++() { ++courant; return *this; }
13   iterator operator++(int) {
14    iterator old=*this; ++courant; return old;
15   }
16   bool operator==(iterator it) const {
17    return courant==it.courant;
18   }
19   bool operator!=(iterator it) const {
20    return courant!=it.courant;
21   }
22  };
23  iterator begin() { return iterator(tab); }
24  iterator end() { return iterator(tab+dim); }
25
26  class const_iterator {
27   friend class vecteur;
28   const double* courant;
29   const_iterator(const double* deb):courant(deb){}
30  public:
31   const_iterator():courant(nullptr){}
32   const double& operator*() const { return *courant; }
33   const_iterator& operator++() { ++courant; return *this; }
34   const_iterator operator++(int) {
35    const_iterator old=*this; ++courant; return old;
36   }
37   bool operator==(const_iterator it) const {
38    return courant==it.courant;
39   }
40   bool operator!=(const_iterator it) const {
41    return courant!=it.courant;
42   }
43  };
44  const_iterator begin() const { return const_iterator(tab); }
45  const_iterator end() const { return const_iterator(tab+dim); }
```

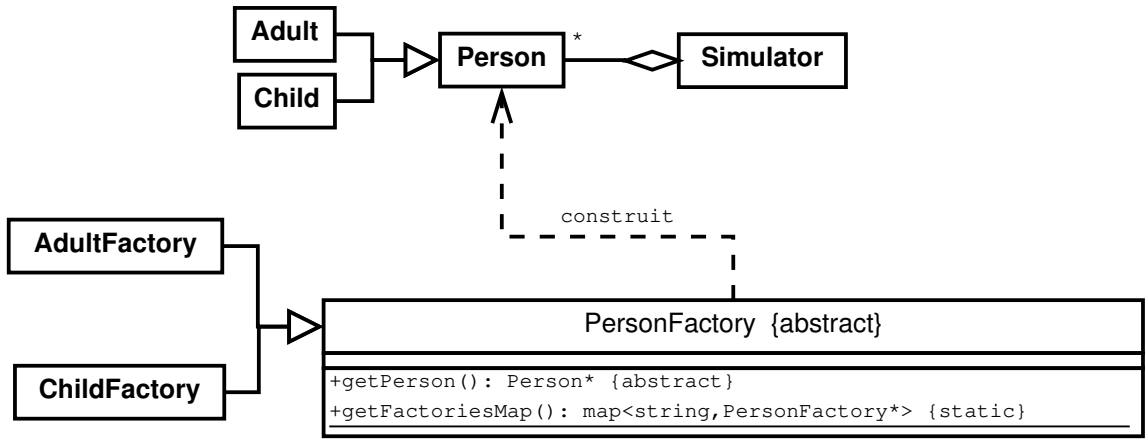
vecteur.h

Ici, la méthode `get_iterator()` est remplacée par la méthode `begin()` qui joue le même rôle. Cependant on voit apparaître la méthode `end()`, qui avec la surcharge des opérateurs **`operator`**== et **`operator`**`!=`, permet de tester si le parcours est terminé. Quant à l'opérateur

d'indirection **operator***, il joue le rôle de la méthode `current_item()` de la question précédente, et l'opérateur **operator**++ celui de la méthode `next()`.

Solution de l'exercice 65

1. Il s'agit du design pattern factory. La classe abstraite `PersonFactory` est un modèle pour toutes les classes constructeurs d'un objet `Person`. À chaque classe concrète de `Person` correspond une classe concrète `Factory`. La méthode statique `getFactoriesMap` permet d'avoir un accès unique vers l'ensemble des classes concrètes `Factory`.



2.
```
1  class PersonFactory {
2  public :
3   PersonFactory(){}
4   virtual ~PersonFactory(){}
5    virtual Person * getPerson(const unsigned int  & id)=0;
6    static map<string, PersonFactory*> getFactoriesMap();
7
8  };
9
10 class AdultFactory : public PersonFactory {
11 public :
12  AdultFactory():PersonFactory (){}
13   Person* getPerson(const unsigned int& id){return new Adult(id);}
14 };
15
16 class ChildFactory : public PersonFactory  {
17 public :
18  ChildFactory():PersonFactory(){}
19   Person* getPerson(const unsigned int& id){ return new Child(id); }
20 };
21
22 map<string, PersonFactory*> PersonFactory::getFactoriesMap(){
23     map<string, PersonFactory*> factories;
24     pair<string, PersonFactory*> entry;
25     entry.first = "Adult";
26     entry.second = new AdultFactory();
27     factories.insert(entry);
28     entry.first = "Child";
29     entry.second = new ChildFactory();
30     factories.insert(entry);
```

```
31      return factories;
32  }
33
34  class Simulator{
35   private :
36      std::vector<Person*> population;
37      map<string, PersonFactory*> factories;
38    public :
39    Simulator(const string& type, const unsigned int& nb){
40     factories = PersonFactory::getFactoriesMap();
41     for(unsigned int i=0; i< nb ;i++)
42     population.push_back(factories.at(type)->getPerson(i));
43    }
44  };
```

Si on ajoute une nouvelle classe concrète de `Person`, il suffit alors de créer une classe abstraite `Factory` correspondante et de compléter le code de `getFactoriesMap()`. On remarque que le code de la classe `Simulator` n'a plus besoin d'être modifié puisque son constructeur utilise directement la méthode `getFactoriesMap()` pour construite les objets dont elle a besoin.

UML

Le jeu du *chifoumi* comporte 2 joueurs (appelés J_1 et J_2 dans la suite). À chaque tour de jeu, les 2 joueurs jouent chacun et simultanément un des 3 coups possibles : *pierre*, *feuille* ou *ciseaux*. La pierre bat les ciseaux (en les émoussant), les ciseaux battent la feuille (en la coupant), la feuille bat la pierre (en l'enveloppant).
Dans certaines des variantes de ce jeu, de nouveaux symboles apparaissent : comme le puits bat la pierre ainsi que les ciseaux (en les faisant tomber au fond du puits), et est battu par la feuille (qui le recouvre).
Une de ces variantes a été popularisée par la série The Big Bang Theory. Il s'agit de *Pierre-Papier-Ciseaux-Lézard-Spock*. Ici, les règles classiques s'appliquent, mais il faut ajouter que le lézard mange le papier, empoisonne Spock, est écrasé par la pierre et est décapité par les ciseaux. Spock vaporise la pierre, casse les ciseaux, et est discrédité par le papier.
On remarque que la relation G (comme "gagne sur") liant les différents symboles (voir Exercice 75) doit être tournoie : pour toute paire de symboles distincts s_1 et s_2, on a soit $s_1 G s_2$, soit $s_2 G s_1$.

On veut concevoir une application qui permet de créer rapidement un jeu de chifoumi paramétré avec les symboles et les relations entre symboles que l'on veut, puis de jouer. Pour cela, l'application (objet d'une classe créée à cet effet) utilisera une sorte d'endorelation G (voir Exercice 75) sur des symboles (qui sont représentés par des chaines de caractères de type `string`) où l'on peut préciser en plus une phrase illustrant la relation entre deux symboles. Par exemple, si on utilise les règles classiques, on doit pouvoir ajouter les symboles *pierre* et *ciseaux* ainsi que le fait que *pierre G ciseaux* illustrée par la phrase "*la pierre émousse les ciseaux*". On doit pouvoir interroger la relation pour savoir si elle est "tournoie" (ou non) pour savoir si une partie peut commencer. L'application doit permettre de faire jouer deux humains, ou un humain contre un joueur simulé par le programme, ou encore, deux joueurs simulés par le programme. On peut demander à l'application de définir (ou de changer) J_1, et de définir (ou de changer) J_2. On peut demander à l'application de lancer une partie composée de plusieurs tours de jeu.
Pour simplifier, on supposera que dans un tour de jeu, J_1 joue toujours d'abord, et ensuite J_2 : on compte sur l'honnêteté de J_2 pour ne pas chercher à savoir ce qu'a joué J_1 pour choisir son coup. On peut demander à un joueur le nombre de tours de jeu qu'il a gagné (initialement nul). On peut demander à un joueur de *jouer*, il doit alors renvoyer le symbole qu'il joue. On peut aussi lui demander d'incrémenter le nombre de tour de jeu gagnés. Une partie termine lorsqu'un joueur a gagné au moins `diff_min` tours de jeu (un attribut paramétrable) de plus que l'autre. L'application est alors capable d'indiquer le gagnant.

Exercice 66 - Diagrammes de classes

En utilisant un diagramme UML, concevoir l'architecture objet de l'application *CHIFOUMI* en respectant les contraintes énoncées dans le texte. Faire apparaitre clairement les attributs et les méthodes des classes. Pour gérer la relation entre symboles, on pourra se contenter de ne reporter qu'une classe `endorelation` telle que définie dans l'Exercice 75. Préciser et justifier les design patterns que vous avez éventuellement utilisé.

Exercice 67 - Diagrammes de séquence

On suppose que l'utilisateur de l'application a déjà paramétré les symboles et la relation entre ses symboles ainsi que les joueurs J1 et J2. Modéliser avec un diagramme de séquence un scenario où une nouvelle partie est lancée et où un nouveau tour de jeu commence. J1 joue puis J2 joue. Enfin, l'application détermine le gagnant du tour de jeu et met à jour les scores. Ce diagramme de séquence doit bien évidemment être en total cohérence avec le diagramme de classe de la question 1.

Exercice 68 - Statecharts

L'application est caractérisée par un état *situation* qui indique si le jeu se trouve dans le mode *attente* (la partie n'a pas encore commencée), *en cours* (la partie a été démarrée et aucun des deux joueurs n'a

remporté au moins `diff_min` tours de jeu de plus que l'autre). Une partie en cours se caractérise par 2 états parallèles. L'un deux indique la situation d'un tour de jeu. L'autre le gagnant courant. Un tour de jeu commence dans l'état "*c'est à J1 de jouer*". Dès que J_1 a joué, le tour de jeu passe dans l'état "*c'est à J2 de jouer*". Dès que J_2 a joué, le tour de jeu passe dans l'état "*arbitrage*", puis dès que les scores ont été mis à jour, repasse dans l'état "**c'est à J1 de jouer**". Le gagnant courant peut être "*nul*" (état initial ou si les joueurs ont tous les deux remporté le même nombre de tours de jeu), "J_1 *gagnant*" (si le score de J_1 est supérieur à celui de J_2), "J_2 *gagnant*" ou "*indéterminé*" (lorsque que l'état du tour de jeu est en mode "*arbitrage*").

Question : Représenter les différents états de l'application par un statechart.

Exercice 69 - La cafetière du labo...

La cafetière du laboratoire `UMR CNRS-UTC 7253 Heudiasyc` est automatique : en appuyant simplement sur le bouton (front montant sur le bouton) `Shoot`, elle moue une certaine quantité de grains de café de la réserve G, elle percole le mou obtenu avec une certaine quantité d'eau de la réserve E (qu'elle éjecte par l'intermédiaire d'un tube distributeur), puis elle rejète le mou utilisé dans une réserve M. Outre le bouton *Shoot*, la cafetière a 3 autres boutons O, A et $+$.
Le logiciel de la cafetière dispose de plusieurs variables internes :

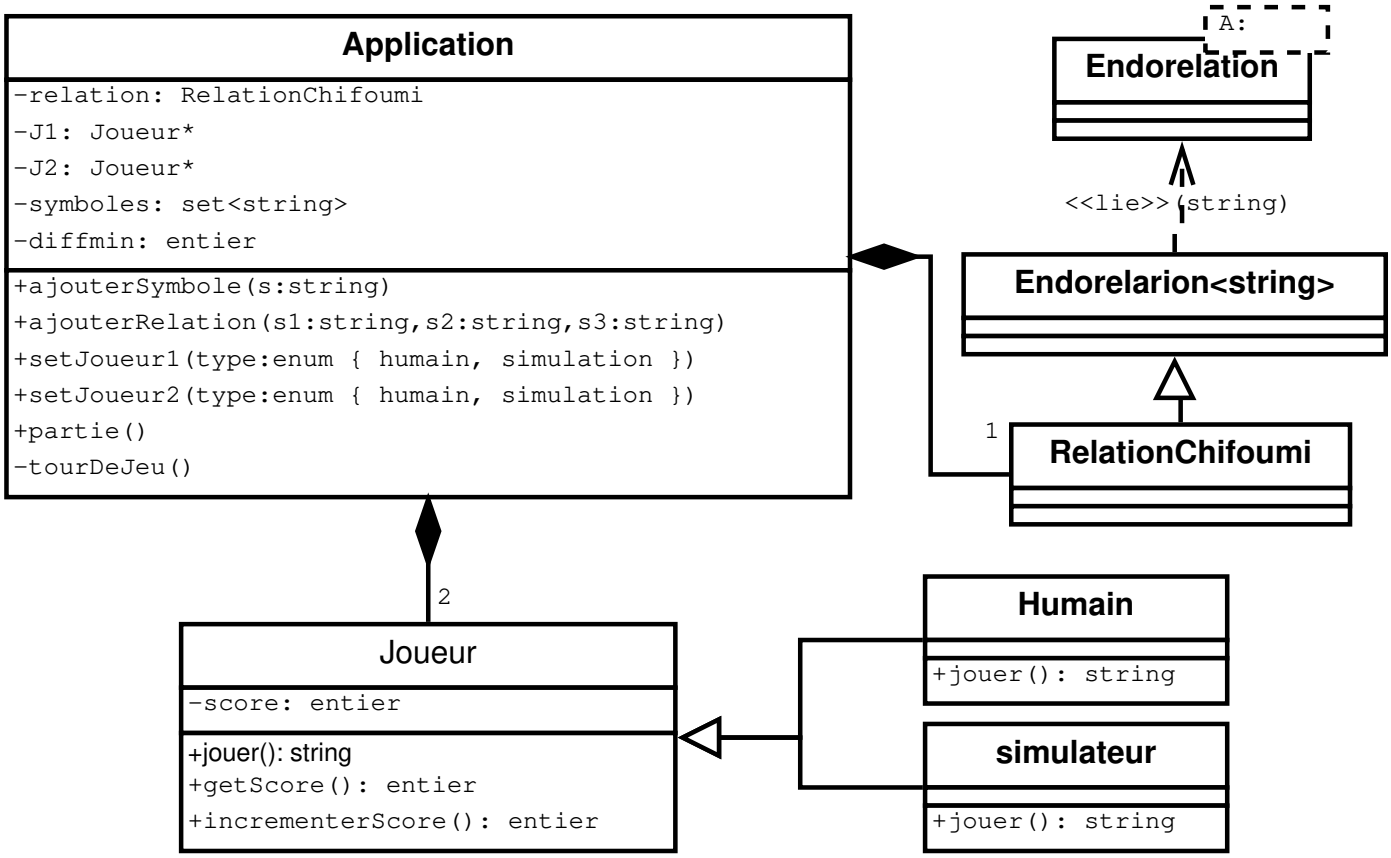
- *horloge* et *horaireAllumage* qui représentent l'état de l'horloge et l'horaire d'allumage automatique ;
- $progression \in \{0, \ldots, 100\}$ qui représente la progression courante d'un shoot de café (par défaut cette variable est égale à la valeur 100) ;
- le booléen $cafe$ qui est égale à *true* si la réserve G contient des grains et à $false$ sinon ;
- le booléen *eau* qui est égale à *true* si la réserve E contient de l'eau et à $false$ sinon ;
- le booléen *plein* qui est égale à *true* si la réserve de mou M est pleine et à $false$ sinon.

Les variables *horloge*, $cafe$, *eau*, *plein* sont mises à jour automatiquement par le logiciel grace à ses capteurs et processeurs.
La cafetière peut être dans 2 états principaux : en mode *on* (la cafetière est en attente d'évènements) ou en mode *veille* (où sa fonction est réduite à l'attente du passage automatique ou manuelle en mode *on*).
Lors de la mise sous tension de la cafetière, l'action *Reset* est exécutée par le logiciel de la cafetière pour initialiser correctement ces variables et la cafetière est automatiquement mise en mide *veille*.
Un front montant sur le bouton `O` permet de passer manuellement du mode *veille* au mode *on*. Elle passe automatiquement du mode *veille* au mode *on* (pour pré-chauffer la machine) lorsque l'allumage automatique est activé (état *auto_on*) et si l'horloge de la cafetière atteint l'horaire d'allumage automatique qui a été réglé.
L'état *on* est caractérisé par plusieurs sous-états : *attente* (par défaut), *percolation*, *erreur*, *reglages*.
Lorsque la cafetière est dans l'état *attente* ou *erreur*, un front montant sur le bouton `O` permet de passer manuellement du mode *on* au mode *veille*. La cafetière passe automatiquement du mode *on* au mode *veille* au bout d'1 heure d'inactivité. Lorsque la cafetière est repassée (automatiquement ou manuellement) dans l'état *on*, elle reviendra dans le dernier sous-état courant (*attente* ou *erreur*) avant le passage en veille.
Lorsque la cafetière est dans l'état *attente*, un front montant sur le bouton *Shoot* fait basculer la cafetière dans l'état *percolation* tout en mettant la variable *progression* à zéro. Durant la percolation, cette variable est augmentée automatiquement par le logiciel selon la quantité d'eau qui a été filtrée. Lorsque *progression* atteint la valeur 100, la cafetière rebascule dans l'état *attente*. À n'importe quel moment de la percolation (au début (broyage des grains), pendant (filtrage de l'eau) ou à la fin (éjection du mou)), s'il n'y a plus de grain dans G, s'il n'y a plus d'eau dans E, ou si M est plein, la cafetière bascule dans l'état *erreur* (qui indique à l'utilisateur qu'elle est en attente d'une action bienfaisante de sa part). La cafetière rebascule dans l'état *percolation* dès que $cafe = true$, $eau = true$ et $plein = false$.
Lorsque la cafetière est dans l'état *attente*, un front montant sur le bouton A fait basculer la cafetière de l'état *auto_on* à l'état $auto\_off$ ou de l'état *auto_on* à l'état $auto\_off$. Un appui long prolongé de 4s sur A fait basculer la cafetière dans l'état *reglages* composé de deux sous états : *reglageHorloge* (état

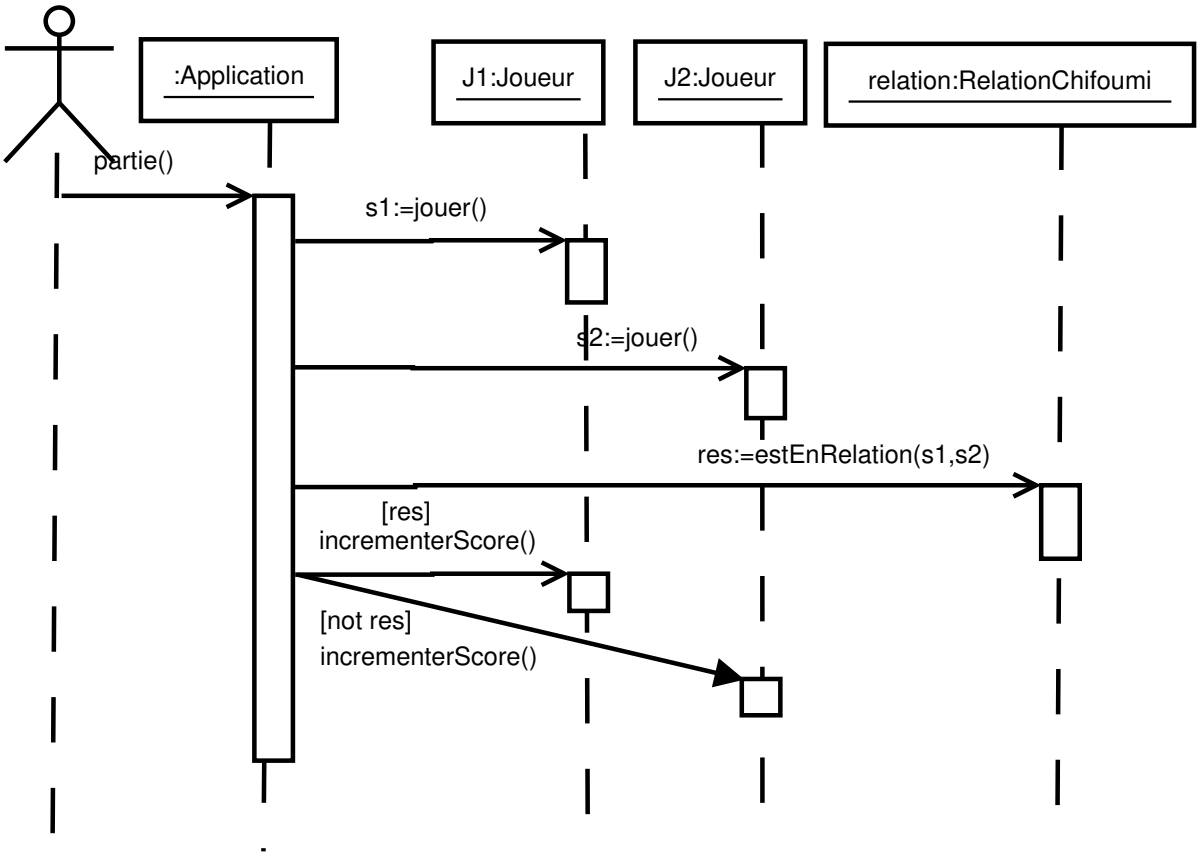
par défaut) et *reglageAllumage* ; le bouton A permet de passer de l'un de ces modes à l'autre, alors que le bouton $+$ permet d'incrémenter l'horloge ou l'horaire d'allumage automatique. Un appui long prolongé de 4*s* sur A refait basculer la cafetière dans l'état *attente*.

Question : Puisque le laboratoire se résume à une bande de geeks accro au café, il a été décidé d'afficher un statechart indiquant comment se servir de la cafetière. Dessiner ce statechart.

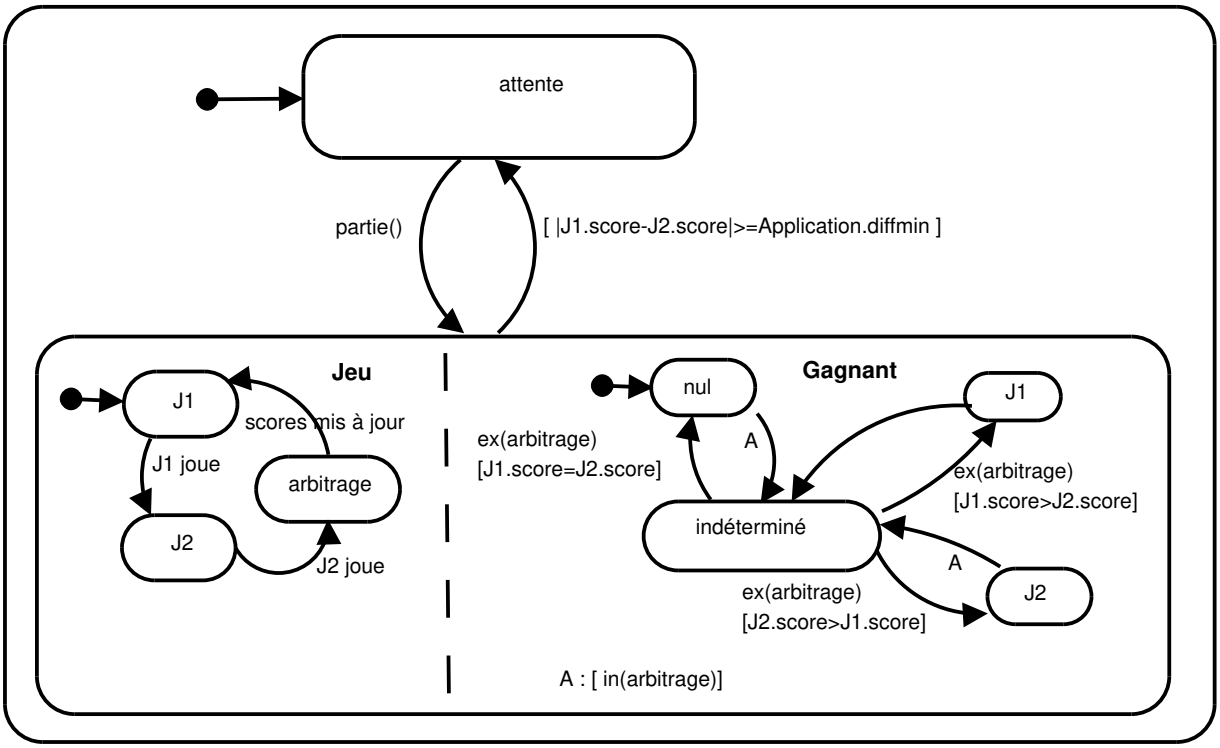
Solution de l'exercice 66



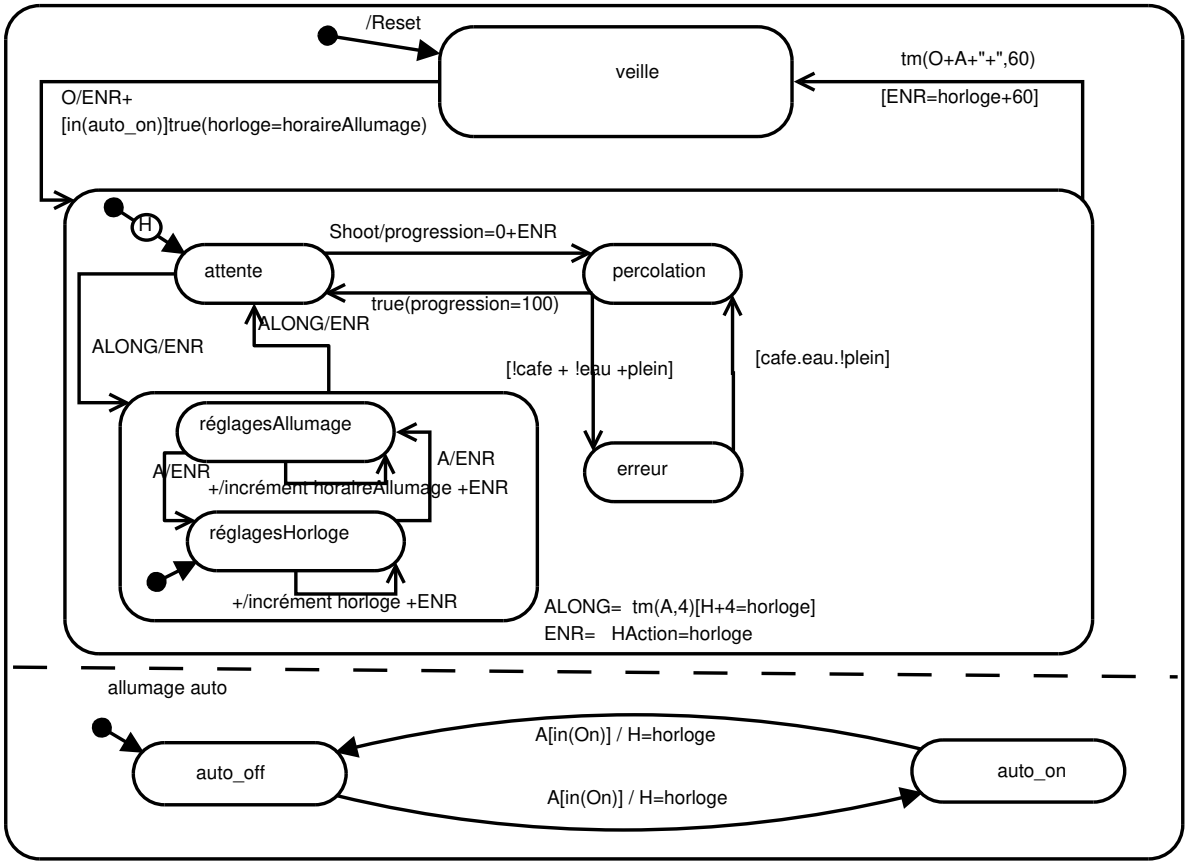
Solution de l'exercice 67



Solution de l'exercice 68



Solution de l'exercice 69



Problèmes

Exercice 70 - Nombres complexes

Un *nombre complexe* $z \in \mathbb{C}$ peut être représenté par la somme $z = a + ib$, où a et b sont des nombres réels ($a, b \in \mathbb{R}$) et i (l'unité imaginaire) est le nombre particulier tel que $i^2 = -1$. La valeur a est appelée la *partie réelle* du nombre z (notée aussi $Re(z)$), alors que b est appelée la *partie imaginaire* du nombre z (notée aussi $Im(z)$).
L'*opposé* de $z = a + ib$, noté $-z$, est le nombre complexe $-z = -a - ib$. Le *conjugué* de $z = a + ib$ est le nombre complexe $a - ib$. La *somme* de 2 nombres $z_1 = a_1 + ib_1$ et $z_2 = a_2 + ib_2$ est le nombre complexe $z_1 + z_2 = (a_1 + a_2) + i(b_1 + b_2)$. Le *produit* de 2 nombres $z_1 = a_1 + ib_1$ et $z_2 = a_2 + ib_2$ est le nombre complexe $z_1 * z_2 = a_1a_2 - b_1b_2 + i(a_1b_2 + b_1a_2)$.
A chaque nombre complexe $z = a + ib$ peut être associé le point M de coordonnées (a, b) dans un repère orthonormal direct $(O, \vec{u}, \vec{v})$ appelé plan complexe. Le vecteur $\vec{OM}$ est alors appelé image vectorielle de z. La distance entre O (l'origine du plan complexe) et M est appelée *module* de z (noté aussi $|z|$). On a alors $|z| = \sqrt{a^2 + b^2}$. L'angle polaire entre $\vec{u}$ et $\vec{OM}$ est appelé *argument* de z (noté aussi $Arg(z)$). Notons alors qu'un nombre complexe z peut aussi être représenté par ses coordonnées polaires (θ, r) où $r = |z|$ et $\theta = Arg(z)$. On a alors $z = rcos\theta + irsin\theta$, c'est à dire $Re(z) = rcos\theta$ et $Im(z) = rsin\theta$.

Question 1

Définir une classe `Complexe` qui est composée de deux attributs privés nommés `pRe` et `pIm` de type **`double`** représentant respectivement la partie réelle et la partie imaginaire d'un nombre complexe. Déclarer et définir également un ou plusieurs constructeurs publics qui permettent de construire un objet `Complexe` avec 0, 1 ou 2 arguments en initialisant la partie réelle et la partie imaginaire de façon à respecter l'exemple suivant :

```
1 Complexe z1; //Re(z1)=0, Im(z1)=0
2 Complexe z2(4); // re(z2)=4, Im(z2)=0
3 Complexe z3(-2,6); // re(z3)=-2, Im(z3)=6
```

Question 2

En vous inspirant du rappel sur les nombres complexes ci-dessus, expliquez quels sont les intérêts de respecter le *principe d'encapsulation* dans le cas de la classe `Complexe`.

Question 3

Déclarer et définir de façon **`inline`** les méthodes `re()` et `im()` pour la classe `Complexe`. La méthode `re()` renverra la partie réelle d'un objet `Complexe` et la méthode `im()` renverra sa partie imaginaire. Ces méthodes doivent-elles être **`const`** ? Ces méthodes doivent-elles être **`static`** ? Expliquer.

Question 4

Déclarer et définir la méthode `module()` qui renvoie le module d'un objet complexe. Pour cela, utiliser la fonction `sqrt` de la bibliothèque standard `cmath` définie dans le **`namespace`** `std`. Cette méthode ne sera pas définie de façon **`inline`** .

Question 5

Déclarer et définir (**`inline`** ou non) la méthode `conjugue()` qui renvoie le nombre complexe conjugué d'un objet `Complexe`. Le type de retour peut-il être de type `Complexe` ? de type **`const`** `complexe` ? de type `complexe&` ? de type **`const`** `complexe&` ? Expliquer à chaque fois.

Question 6

Déclarer et définir (**inline** ou non) la méthode **operator**+() de la classe Complexe qui permet de faire la somme de deux objets Complexe.

Question 7

Surcharger l'opérateur << de façon à pouvoir écrire un objet Complexe sur un flux ostream sous la forme a+bi où a et b sont les parties réelle et imaginaire de l'objet Complexe.

Question 8

Est-il nécessaire de définir un destructeur, un constructeur de recopie et un opérateur d'affectation pour la classe Complexe ? Expliquer.

Ensemble de nombres complexes. On veut maintenant construire une classe ComplexeSet qui permet de stocker un ensemble de nombres complexes. Voici les spécifications de cette classe ainsi que son interface partielle.
Un objet de la classe ComplexeSet permet de stocker des objets de type Complexe. Pour cela, cette classe est composée d'un tableau tab d'objets Complexe qui permet de stocker des objets Complexe. Ce tableau est *alloué dynamiquement*. Un attribut nb indique le nombre d'objets Complexe ajoutés à l'ensemble. Cette valeur est accessible par l'accesseur taille(). Le constructeur de cette classe permet d'initialiser la structure de données en allouant dynamiquement un tableau d'objets complexe. Initialement, l'ensemble est considéré comme vide (nb=0). L'opérateur << est surchargé de manière à pouvoir ajouter un objet de type Complexe dans l'ensemble. Lors de l'appel de cette méthode, l'objet Complexe passé en argument est stocké à la première place disponible dans tab (c'est à dire à la l'emplacement nb) et l'attribut nb est incrémenté. S'il n'y a pas suffisamment de place dans le tableau, il sera nécessaire au préalable *d'agrandir le tableau*. La méthode display permet d'afficher les nombres complexes appartenant à l'ensemble sur un plan dont les limites, la précision et le titre sont indiqués en paramètres de la méthode.
Une interface partielle de la classe ComplexeSet :

```
1  class ComplexeSet {
2  public:
3   ComplexeSet(); //constructeur
4   void operator<<(Complexe c); //ajout d'un complexe dans l'ensemble
5   void display(const std::string& title,double precision,
6                double x_min, double x_max,
7                double y_min, double y_max) const;
8    unsigned int taille() const { return nb; }
9  private:
10  Complexe* tab; //tableau alloué dynamiquement
11  unsigned int max; //taille du tableau tab
12  unsigned int nb; //nb de complexes stocké
13 };
```

Question 9

Dessiner un diagramme UML où apparaissent les classes Complexe, et ComplexeSet en indiquant leurs attributs, leurs méthodes, la visibilité de leurs membres ainsi que les relations entre ces classes.

Question 10

Pour la classe ComplexeSet, est-il nécessaire de définir un destructeur ? un constructeur de recopie ? un opérateur d'affectation ? Expliquer pour chaque cas. Compléter alors l'interface partielle donnée ci-dessus, seulement si nécessaire.

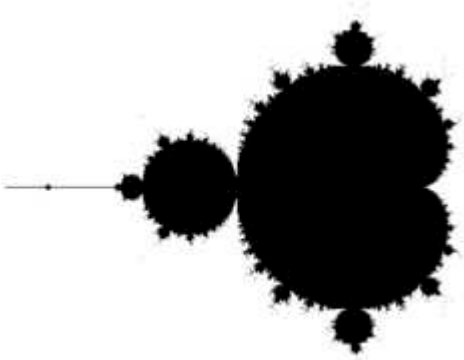
Question 11

Définir le constructeur, l'opérateur << ainsi que toutes les méthodes qui ont été jugées nécessaires dans la question précédente.

Question 12

Appliquer le design pattern *iterator* afin de pouvoir accéder séquentiellement aux nombres complexes appartenant à un objet `ComplexeSet`.

A titre d'exemple de la classe `ComplexeSet`, le code suivant permet de calculer un ensemble de nombres complexes représentatifs de l'ensemble (très connu) de Mandelbrot. L'*ensemble de Mandelbrot* est l'ensemble des nombres complexes c pour lesquels la suite définie par récurrence par $z_{n+1} = z_n^2 + c$ avec $z_0 = 0$ ne tend pas vers l'infini (en module). L'image ci-dessous a été obtenue en implémentant la méthode `display` (qui n'est pas à faire) avec la bibliothèque (gratuite et portable) **SFML** (Simple and Fast Multimedia Library) écrite en `C++`.



```
1  void Mandelbrot(){
2   // creation d'un ensemble pouvant contenir jusqu'à 80000 complexes
3   ComplexeSet e(80000);
4   const double p=0.005; const unsigned int nb_iter=50;
5   for(double x=-2; x<=1; x+=p){
6    for(double y=-1; y<=1; y+=p){
7     complexe c(x,y), z;
8     unsigned int i=0;
9     while (i<nb_iter && z.module()<2){z=z*z+c; i++;}
10    if (z.module()<=2) e<<c; // ajout de c à l'ensemble
11   }
12  }
13  e.display("Ensemble de Mandelbrot",0.005,-2,1,-1,1);
14 }
```

Solution de l'exercice 70

Question 1

```
1 class Complexe {
2 public:
3  Complexe(double r=0.0,double i=0.0):pRe(r),pIm(i){}
4 private:
5  double pRe;
6  double pIm;
7 };
```

complexe.h

Question 2

L'intérêt ici d'appliquer le principe d'encapsulation est de protéger l'utilisateur d'un éventuel changement de structure de données. En effet, il est possible d'implémenter un nombre complexe en utilisant des coordonnées polaires.

Question 3

```
1 class Complexe {
2 public:
3  //...
4  double re() const { return pRe; }
5  double im() const { return pIm; }
6 };
```

complexe.h

Question 4

```
1 class Complexe {
2 public:
3  //...
4   double module() const;
5 };
```

complexe.h

```
1 #include "complexe.h"
2 #include <cmath>
3 double complexe::module() const {
4  return std::sqrt(pRe*pRe+pIm*pIm);
5 }
```

complexe.cpp

Question 5

```
1 class Complexe {
2 public:
3  //...
4  Complexe conjugue() const { return Complexe(pre,-pim); }
5 };
```

complexe.h

Avec l'instruction utilisée, le type de retour ne peut pas être une référence. Cette dernière serait en effet invalide à l'issu de l'appel de la méthode `conjugue`, l'obet créé à l'intérieur de cette méthode étant local. Le seul mode possible de transmission reste le passage par valeur. Le type de retour **`const`** `Complexe` est alors aussi parfaitement valide. Remarquons qu'une transmission par référence serait possible en faisant une allocation dynamique d'un objet `Complexe` à l'intérieur de la fonction. Mais se poserait alors la question de la destruction de cet objet. Ce n'est donc pas a priori une bonne idée car l'utilisateur de la méthode peut ne pas savoir que cet objet a été alloué dynamiquement.

Question 6

```
1 class Complexe {
2 public:
3  //...
4  Complexe operator+(const Complexe& c) const {
5   return Complexe(pRe+c.pRe,pIm+c.pIm);
6  }
7 };
```

complexe.h

Notons que, bien qu'il soit demandé une *méthode* dans cette question, une fonction non membre serait bien plus pertinente afin de profiter de la conversion implicite de **`double`** vers `Complexe` mise en place grace au constructeur de la classe `Complexe`. Cette version, en tat que fonction non-membre doit utiliser les accesseurs publiques :

```
1 class Complexe {
2  //...
3 };
4 inline Complexe operator+(const Complexe& c1, const Complexe& c2) {
5  return Complexe(c1.re()+c2.re(),c1.im()+c2.im());
6 }
```

complexe.h

Question 7

```
1 #include<iostream>
2 class Complexe {
3 //...
4 };
5 inline std::ostream& operator<<(std::ostream& f, const Complexe& c){
6  f<<c.re()<<"+"<<c.im()<<"i";
7  return f;
```

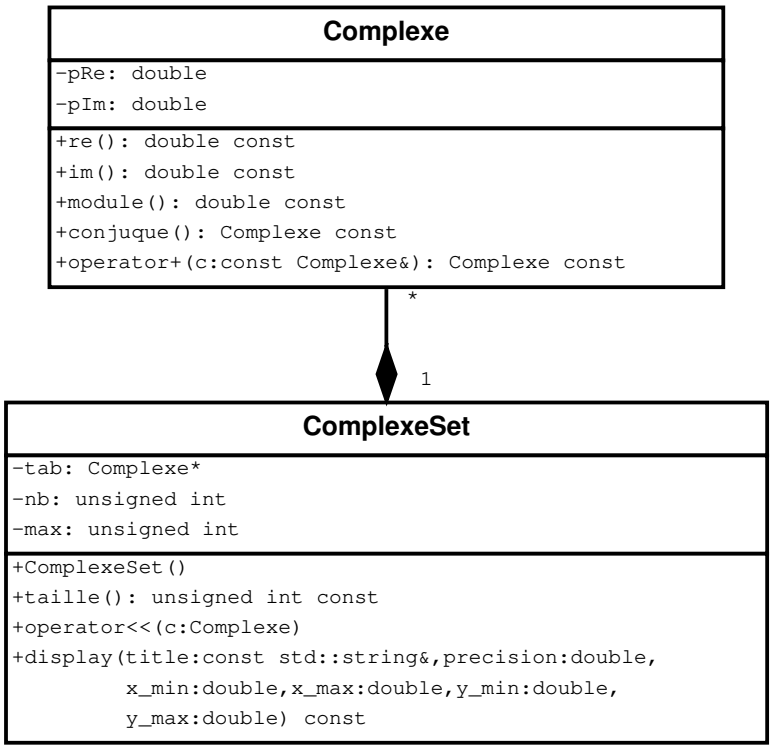
```
8 }
```

complexe.h

Question 8

Il n'est pas nécessaire de définir un destructeur, un constructeur de recopie et un opérateur d'affectation pour la classe `Complexe` puisque les méthodes correspondantes mises en place par défaut par le compilateur sont adéquates.

Question 9



Question 10

Il est nécessaire de définir un destructeur, un constructeur de recopie et un opérateur d'affectation pour la classe `ComplexeSet` puisque les méthodes correspondantes mises en place par défaut par le compilateur n'alloue pas de nouveau tableau d'objets `Complexe` pour l'objet créé ou assigné : la valeur de l'attribut `tab` (qui est un pointeur) de l'objet à recopier est simplement recopiée dans l'objet destinataire menant à deux objets `ComplexeSet` pointant sur le même tableau de `Complexe`. Ceci est évidemment en total contradiction avec l'indication de la composition utilisée dans le diagramme UML de la question précédente.

Question 11

```
1 class ComplexeSet {
2 //...
3 public:
4   void ComplexeSet():tab(nullptr), max(0), nb(0){}
5  void operator<<(Complexe c);
6  ComplexeSet(const ComplexeSet& c);
7  ComplexeSet& operator=(const ComplexeSet& c);
8  ~ComplexeSet();
9 };
```

complexe.h

```
//...
void ComplexeSet::operator<<(Complexe c){
 if (nb==max){
  Complexe* newtab=new Complexe[max*2];
  for(unsigned int i=0; i<nb; i++) newtab[i]=tab[i];
  Complexe* old=tab;
  tab=newtab; max*=2; delete[] old;
 }
 tab[nb++]=c;
}

ComplexeSet::ComplexeSet(const ComplexeSet& e):tab(new Complexe[e.max]),nb(e.nb)
    ,max(e.max){
 for(unsigned int i=0; i<nb; i++) tab[i]=e.tab[i];
}

ComplexeSet::ComplexeSet& operator=(const ComplexeSet& e){
 if (this!=&e){
  max=e.max;
  nb=e.nb;
  Complexe* newtab=new Complexe[max];
  for(unsigned int i=0; i<nb; i++) newtab[i]=e.tab[i];
  Complexe* old=tab; tab=newtab; delete[] old;
 }
 return *this;
}

ComplexeSet::~ComplexeSet() { delete[] tab; }
```

complexe.cpp

Question 12

```
class ComplexeSet {
//...
public:
 class iterator{
 public:
  iterator(Complexe* c):cur(c){}
  void operator++(){ cur++; }
  bool operator!=(iterator it) const { return cur!=it.cur; }
  Complexe& operator*() const { return *cur ; }
 private:
  Complexe* cur;
 };
 iterator begin() { return iterator(tab); }
 iterator end() { return iterator(tab+nb); }
```

```
15 };
```

complexe.h

Exercice 71 - Vecteurs et Matrices

Dans le sujet, on dispose d'un fichier d'entête `vecteur.h` et d'un fichier source `vecteur.cpp`. À chaque fois que vous répondrez à une question pour laquelle du code C++ doit être écrit, indiquez dans lequel des 2 fichiers il doit être placé.

Rappels sur les vecteurs : Un vecteur de $\mathbb{R}^n$ peut être représenté par un n-uplet $[u_1, \ldots, u_n]$ où n est appelé la dimension du vecteur et où $\forall i \in \{1, \ldots, n, u_i \in \mathbb{R}$. Soient 2 vecteurs U et V de $\mathbb{R}^n$ représentés par les n-uplets $[u_1, \ldots, u_n]$ et $[v_1, \ldots, v_n]$. Le produit scalaire de U et V est un nombre réel égal à $u_1v_1 + u_2v_2 + \ldots + u_nv_n$.

Dans ce problème, les situations exceptionnelles seront gérées en utilisant la classe d'exception `VecteurException` définie dans le fichier `vecteur.h` et qui contient le code suivant :

```
1 class VecteurException {
2 public:
3  VecteurException(const std::string& m):info(m){}
4  const std::string& getInfo() const { return info; }
5 private:
6  std::string info;
7 };
```

Question 1

Définir une classe `Vecteur` dont les objets représentent des vecteurs. Cette classe composera deux attributs privés : un attribut `dim` de type **`unsigned int`** représentant la dimension du vecteur et un attribut `val` de type **`double`**`*` qui pointera sur un tableau alloué dynamiquement de **`double`**. Définir un constructeur qui reçoit en argument une variable `n` de type **`unsigned int`** et qui permet de construire un vecteur de $\mathbb{R}^n[u1, \ldots, un]$ où $u_i = 0, \forall i \in \{1, \ldots, n\}$.

Question 2

Y a t-il un intérêt d'utiliser le mot clé **`explicit`** avec le constructeur de la question précédente ? Expliquer.

Question 3

Quel type d'association est instancié entre un objet de la classe `Vecteur` et le tableau pointé par son attribut `val` ? Expliquer.

Question 4

Déclarer et définir de façon **`inline`** une méthode `dimension()` qui renvoie la dimension d'un objet `Vecteur`. Cette méthode doit-elle être **`const`** ? Cette méthode doit-elle être **`static`** ? Expliquer.

Question 5

Déclarer et définir un destructeur et/ou le constructeur de recopie et/ou l'opérateur d'affectation pour cette classe, seulement si vous jugez que c'est nécessaire. Justifier.

Question 6

Déclarer et définir **la méthode `operator*` de la classe `Vecteur`** qui permet d'obtenir le résultat du produit scalaire de deux objets `Vecteur` de mêmes dimensions. Si les dimensions des deux vecteurs ne sont pas les mêmes, cette méthode déclenchera une exception de type `VecteurException`. Cette méthode ne sera pas **`inline`**. Le type de retour peut-il être de type **`double`** ? de type **`double`**`&` ? Expliquer à chaque fois.

Question 7

Surcharger (déclaration(s) et définition(s)) l'**`operator`**`[]` de manière à pouvoir accéder à une coordonnée d'un vecteur pour en lire ou modifier la valeur. Exemple : `v[3]=4` doit permettre de modifier la 3ème coordonnée de l'objet `Vecteur v` avec la valeur 4. On doit pouvoir accéder à une coordonnée

d'un objet Vecteur constant. Le(s) type(s) de retour peut-il (peuvent-ils) être de type **double** ? de type **double&** ? de type **const double&** ? Expliquer à chaque fois

Question 8

Déclarer et surcharger l'opérateur << de façon à pouvoir écrire un objet `Vecteur` sur un flux `ostream`.

On veut implémenter une classe `Matrice`. Pour cela, on choisit de prendre comme structure de données un tableau alloué dynamiquement de pointeurs de `Vecteur`. Chaque pointeur de ce tableau pointe vers un objet `Vecteur` alloué dynamiquement représentant une ligne de la matrice. Une interface partielle de la classe `Matrice` est fournie ci-dessous :

```
1 class Matrice{
2  Vecteur** M; //pointe sur un tableau de pointeurs de vecteurs
3  unsigned int nbLignes; unsigned int nbColonnes;
4 public:
5  unsigned int getNbLignes() const;
6  unsigned int getNbColonnes() const;
7 };
```

vecteur.h

Question 9

Expliquer pourquoi on ne peut pas utiliser plus simplement un tableau d'objets `Vecteur` à la place d'un tableau de pointeurs de `Vecteur`.

Question 10

Déclarer et définir le constructeur `Matrice(unsigned int n, unsigned int m)` qui permet de construire une matrice à partir de dimensions $n \times m$ et dont les cellules sont initialisées avec 0.

Question 11

Est-il nécessaire de définir un destructeur pour la classe `Matrice` ? un constructeur de recopie ? un opérateur d'affectation ? Justifier les réponses. Même si vous jugez que ces méthodes sont nécessaires, il n'est pas demandé ici de définir ces méthodes.

Question 12

Dessiner un diagramme UML avec les classes `Vecteur` et `Matrice` en indiquant leurs attributs, leurs méthodes, la visibilité de leurs membres ainsi que les associations entre classes.

Question 13

Appliquer le design pattern "iterator" afin de pouvoir accéder séquentiellement aux différentes lignes d'un objet `Matrice`. Ecrire un exemple comportant une suite d'instructions qui utilise cet itérateur pour afficher toutes les lignes d'un objet `Matrice`.

Solution de l'exercice 71

Question 1

```
1 class Vecteur {
2 private:
3  unsigned int dim;
4  double* val;
5 public:
6  explicit Vecteur(unsigned int n):dim(n),val(new double[n]){
7   for(unsigned int i=0; i<n; i++) val[i]=0;
8  }
9 };
```

vecteur.h

Variantes :

- Le constructeur peut être défini **inline** ou non.
- On peut aussi choisir de ne pas initialiser les attributs avec la notation " :" et d'utiliser plutôt des affectations pour initialiser les composantes du vecteur.
- Le motc clé **explicit** n'est pas encore exigé dans la réponse de cette question (question suivante).
- Le mot clé **private** est ici optionnel. Il ne l'est pas si la partie **public** est placée avant la partie **private**.

Question 2

Il y a un intérêt à utiliser le mot clé **explicit** devant le constructeur. En effet, s'il est omis, le compilateur accepte les conversions implicites d'une valeur de type **unsigned int** vers `Vecteur` (puisque l'on peut construire un objet `Vecteur` en fournissant uniquement un **unsigned int**. Une expression du type `Vecteur v=4;` serait alors acceptée, ce qui serait ambigü car concrètement cela mène à la création d'un vecteur de dimensions 4.

Question 3

L'association instanciée est une composition. En effet, un objet `Vecteur` est "composé" d'un tableau. Le cycle de vie d'un tableau est totalement lié à l'objet `Vecteur` qui le compose. Un tableau ne peut pas être partagé entre plusieurs objets `Vecteur`

Question 4

```
1 class Vecteur {
2 //...
3 public:
4  unsigned int dimension() const { return dim; }
5 };
```

vecteur.h

- Une fonction **inline** est définie à l'intérieur de la définition de la classe. Elle peut éventuellement être définie à l'extérieur de la définition de la classe. Cependant, on doit dans ce cas utiliser le mot clé **inline** et placer la définition dans le fichier `vecteur.h`.

— Cette méthode renvoie la valeur de l'attribut `dim`. Le retour peut se faire par valeur (type **`unsigned int`**) ou éventuellement avec une référence constante (type **`const unsigned int &`**).

— Cette méthode devrait être **`const`** puisque c'est une requête : elle n'a aucune bonne raison de modifier les attributs.

— Cette méthode ne doit pas être **`static`**. En effet `dim` n'est pas un attribut de classe, mais un attibut d'objet : chaque objet a sa propre valeur `dim` qui ne peut donc être un attribut **`static`**. Une méthode **`static`** n'ayant accès qu'aux attributs **`static`**, la méthode `dimension` ne peut pas être **`static`**.

Question 5

```
1  class Vecteur {
2  //...
3  public:
4   ~Vecteur() { delete[] val; }
5   Vecteur(const Vecteur& v):dim(v.dim),val(new double [v.dim]){
6    for(unsigned int i=0; i<dim; i++) val[i]=v.val[i];
7   }
8   Vecteur& operator=(const Vecteur& v){
9    double* newval=new double[v.dim];
10   for(unsigned int i=0; i<v.dim; i++) newval[i]=v.val[i];
11   double* old=val;
12   val=newval;
13   dim=v.dim;
14   delete[] old;
15   return *this;
16  }
17 };
```

vecteur.h

— On doit définir un destructeur afin de libérer la mémoire alloué dynamiquement pour le tableu `val`. Dans le destructeur, on fait alors appelle à l'opérateur **`delete[]`** sur cet attribut.

— Le constructeur de recopie et d'affectation sont nécessaires pour gérer la mémoire correctement. En effet, les méthodes correspondantes générées par défaut par le compilateur ne font que recopier la valeur de l'attribut `val` d'un objet vers l'autre. Cela signifie que deux objets `Vecteur` partageraient le même tableau ; ce qui n'est pas possible dans le cas d'une composition.

— L'unique paramètre du constructeur de recopie doit être référence sur `Vecteur` sinon ce serait un passage par valeur qui provoquerait l'appel du constructeur de recopie lui-même (cas de récursion infinie heureusement détectée par les compilateurs). Il est préférable que cette référence soit constante pour pouvoir dupliquer des objets `Vecteurs` constants (ou considérés comme constants par l'intermédiaire d'une référence **`const`**) et parcequ'il n'y aucune raison pour que le constructeur de recopie modifie l'objet `Vecteur` passé en argument. Dans le constructeur, il faut évidemment allouer un nouveau tableau dynamique et recopier les valeurs de chaque composante du vecteur dupliqué dans l'objet `Vecteur` créé.

— L'unique paramètre de l'opérateur d'affectation peut être de type `Vecteur`, `Vecteur&` ou **`const`** `Vecteur&`. Il est cependant préférable que ce soit une référence (pour éviter une recopie supplémentaire inutile du `Vecteur` dupliqué induite par un passage par valeur). Il est préférable que cette référence soit **`const`** pour les mêmes raisons que dans le constructeur de recopie. Dans l'opérateur d'affectation, il faut s'assurer qu'une auto-affectation ne pose pas de problème (soit en testant l'auto-affectation avec un test de comparaison entre **`this`** et l'adresse de l'objet passée

en argument, soit en faisant en sorte que l'ordre des opérations ne provoque jamais de perte de données). Le type de retour peut être **void**, Vecteur, Vecteur&, **const** Vecteur, **const** Vecteur&. Traditionnellement, on renvoie une valeur de type Vecteur& pour permettre des affectations en chaîne (non exigé dans l'exercice). Si le Vecteur dupliqué a une dimension différente de l'objet Vecteur affecté, il faut gérer la mémoire correctement en allouant un nouveau tableau de la taille adaptée et en désallouant l'ancien tableau.

— Ici, ces méthodes peuvent être définies **inline** ou non.

Question 6

— Une subtilité du texte (néanmoins très visible) exige que l'on implémente une méthode de classe avec un seul argument (plus l'argument implicite) et non une fonction indépendante de la classe :

```
1 class Vecteur {
2 //...
3 public:
4  double operator*(const Vecteur& v) const;
5 };
```

vecteur.h

```
1 #include "vecteur.h"
2 double Vecteur::operator*(const Vecteur& v) const {
3   if (dim!=v.dim} throw VecteurException("produit scalaire de deux vecteurs
       de dimensions differentes");
4   double res=0;
5   for(unsigned int i=0; i<dim; i++) res+=val[i]*v.val[i];
6   return res;
7 }
```

vecteur.cpp

— Il est préférable qu'un paramètre de cette méthode/fonction soit une référence constante.

— Le type de retour doit être **double** (ou **const double**) mais en aucun cas il ne peut être une référence. En effet, la variable res à l'intérieur de la fonction est locale à la fonction. Elle est donc détruite à la fin de la fonction. Renvoyer une référence de res consiste donc à fournir l'adresse d'un objet qui a été désalloué. Le passage par valeur pour le retour de fonction est donc obligatoire.

— Il faut utiliser le mot clé **throw** pour déclencher une exception.

— Puisqu'une méthode non-**inline** est exigée, on ne retrouvera que la déclaration de la méthode dans le fichier d'entête (vecteur.h). La définition de la méthode sera placée dans le fichier d'implémentation (vecteur.cpp).

— Notons qu'il est normalement possible de faire aussi une fonction non membre de la classe (qui utilise la méthode dimension() et l'opérateur **operator**[] de la question suivante pour accèder à chaque composante). Cette version n'est cependant pas en accord avec la question. En voici tout de même une version pour information :

```
1 class Vecteur {
2 //...
3 public:
4 };
5 double operator*(const Vecteur& v1, const Vecteur& v2);
```

vecteur.h

```
1 #include "vecteur.h"
2 double operator*(const Vecteur& v1, const Vecteur& v2){
3  if (v1.dimension()!=v2.dimension()} throw VecteurException("produit
      scalaire de deux vecteurs de dimensions differentes");
4   double res=0;
5   for(unsigned int i=0; i<v1.dimension(); i++) res+=v1[i]*v2[i];
6   return res;
7 }
```

vecteur.cpp

Question 7

```
1 #include<iostream>
2 class Vecteur {
3 //...
4 public:
5  double& operator[](unsigned int i) { return val[i]; }
6  double operator[](unsigned int i) const { return val[i]; }
7 };
```

vecteur.h

— On devrait avoir deux versions de la méthode **`operator[]`** : une version non-**`const`** pour la modification, et une version **`const`** pour la lecture.

— Dans les deux versions, le type du paramètre peut être **`int`** ou **`unsigned int`**.

— Pour la version non-**`const`** le type de retour doit être une référence. En effet, un retour par valeur ne permettrait pas de modifier une composante d'un vecteur puisque l'on ne disposerait alors que d'une copie.

— La version **`const`** doit renvoyer une référence constante ou une valeur de type **`double`** pour s'assurer que l'utilisateur ne puisse pas modifier une composante en utilisant cet opérateur.

Question 8

```
1 #include<iostream>
2 class Vecteur {
3 //...
4 };
5 std::ostream& operator<<(std::ostream&, const Vecteur&);
```

vecteur.h

```
1 #include "vecteur.h"
2 //...
3 std::ostream& operator<<(std::ostream& f, const Vecteur& v){
4  f<<"["<<v[0];
5  for(unsigned int i=1; i<v.dimension(); i++) f<<","<<v[i];
6  f<<"]";
7  return f;
```

```
8 }
```

vecteur.cpp

— L'opérateur doit obligatoirement être surchargé en tant que fonction. Par rapport à son utilisation dans la question, il ne peut pas être une méthode de la classe `Vecteur`. En effet, cela signifierait que l'argument gauche de l'opérateur serait un objet `Vecteur`. Pour que l'argument gauche soit un objet de la classe `ostream`, il faudrait surcharger l'opérateur `<<` en tant que méthode de la classe `ostream`, ce qui n'est évidemment pas possible.

— Un objet `ostream` ne peut être passé par valeur. En effet, la déclaration de son constructeur de recopie et son opérateur d'affectation dans la partie privée en empêche la duplication accidentelle. Le passage par valeur n'est donc pas possible.

— Puique **`operator`**`<<` est surchargé en tant que fonction indépendante de la classe `Vecteur`, on doit donc utiliser les accesseurs (`dimension()` et **`operator`**`[]` pour accéder aux informations utiles. C'est la version **`const`** de **`operator`**`[]` qui est utilisée si le paramètre droit est une référence constante. Ce devrait être le cas pour éviter une copie sans laisser la possibilité d'une éventuelle modification dans la méthode (on n'en tiendra cependant pas rigueur dans la notation).

Question 9

On ne peut pas allouer un tableau d'objets `Vecteur` avec l'opérateur **`new`**`[]`. En effet, la classe `Vecteur` ne dispose pas de constructeur sans argument. L'opérateur **`new`**`[]` ne peut alors pas construire les objets du tableau. Notons que ce n'est pas le cas de l'opérateur **`new`** qui permet d'allouer dynamiquement un seul objet en le faisant suivre directement de l'appel d'un constructeur (avec ou sans argument). Pour contourner la contrainte imposée par **`new`**`[]`, on peut alors utiliser un tableau de pointeurs de `Vecteur`. Il faudra cependant allouer un par un chacun des objets `Vecteur` pointés par les pointeurs du tableau.

Question 10

```
1  //...
2  class Matrice{
3   Vecteur** M; //pointe sur un tableau de pointeurs de vecteurs
4   unsigned int nbLignes; unsigned int nbColonnes;
5  public:
6   //...
7   Matrice(unsigned int n, unsigned int m):
8     M(new Vecteur*[n]), nbLignes(n), nbColonnes(m){
9    for(unsigned int i=0; i<n ; i++){
10    M[i]=new Vecteur(m);
11   }
12  }
13 };
```

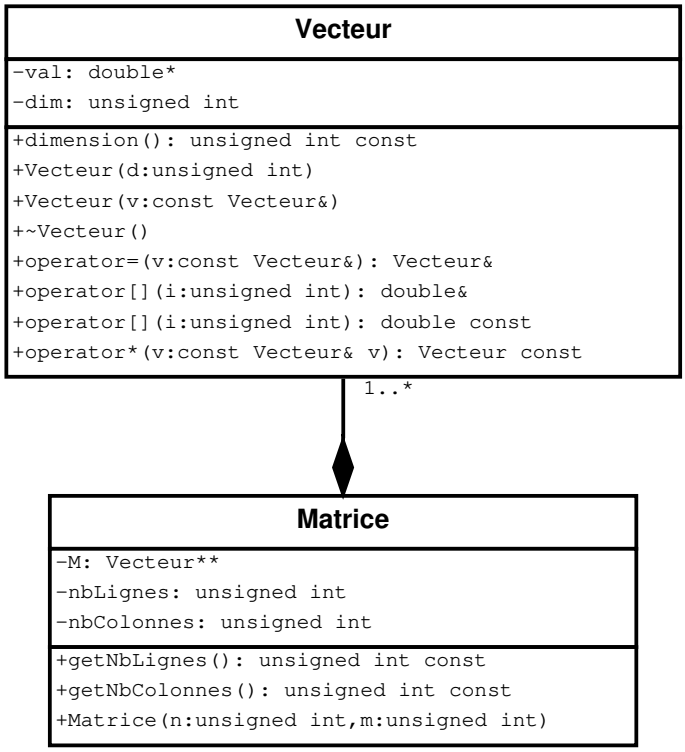
vecteur.h

— Après avoir alloué le tableau de pointeurs *M*, il faut allouer un par un chacun des objets `Vecteur` pointés par les pointeurs du tableau en utilisant le seul constructeur à un paramètre disponible. Les attributs `nbLignes` et `nbColonnes` doivent également être initialisés.

Question 11

Pour les mêmes raisons que pour la classe `Vecteur`, un destructeur, un constructeur de recopie et un opérateur d'affectation sont nécessaires.

Question 12



Question 13

Voici une proposition d'implémentation.

```
1  //...
2  class Matrice{
3  public:
4   //...
5   class Iterator {
6    const Vecteur** courant;
7   public:
8    const Vecteur& operator*() const { return **courant; }
9    bool operator!=(Iterator it) const { return courant!=it.courant; }
10   Iterator& operator++() { courant++; return *this; }
11   Iterator(const Vecteur* v=nullptr):courant(v){}
12  };
13  Iterator begin() const { return Iterator(M); }
14  Iterator end() const { return Iterator(M+nbLignes); }
15 };
```

vecteur.h

D'autres interfaces sont possibles.

Exercice 72 - Un gestionnaire d'étiquettes

Dans cet exercice, un *item* est un objet caractérisé par un nom et un ensemble d'autres items qui lui sont associés. On décide d'implémenter la classe `Item` représentant un tel concept. Dans cette implémentation, les cycles de vie de deux objets `Item` associés sont indépendants. La classe comporte un attribut `name` de type `std::string` et un attribut `tab` de type `Item**` destiné à recevoir l'adresse d'un tableau alloué dynamiquement de pointeurs de type `Item*`. Le tableau pointé par `tab` d'un objet `Item` contiendra les adresses des autres objets `Item` qui lui sont associés. La classe possède aussi deux autres attributs de type **`unsigned int`** : l'attribut `nb` est destiné à contenir le nombre d'adresses sauvegardées dans le tableau pointé par `tab` alors que l'attribut `max` est destiné à sauvegarder la taille du tableau pointé par `tab`. Cette classe possède un unique constructeur avec un paramètre de type **`const`** `string&` permettant d'initialiser l'attribut `name`. Initialement, un objet `Item` n'est associé à aucun autre objet `Item` et aucun tableau n'est alloué. La méthode de prototype **`void`** `addAssociatedItem(Item& item);` permettra d'ajouter une association avec un autre `Item` transmis en argument. Cette méthode sera chargée de gérer (au niveau mémoire) le tableau pointé par `tab`, et notamment, d'étendre la capacité de ce tableau si nécessaire. La méthode de prototype **`void`** `removeAssociatedItem(Item& item);` permettra d'enlever l'association avec l'objet `Item` transmis en argument.

Question 1

On suppose que l'on dispose d'un fichier d'entête `item.h` et d'un fichier source `item.cpp`. À chaque fois que vous répondrez à une question de cette partie pour laquelle du code C++ doit être écrit, indiquez dans lequel des 2 fichiers il doit être placé. Les situations exceptionnelles seront gérées en utilisant la classe d'exception `ItemException` que l'on supposera déjà définie au début du fichier `item.h` avec le code suivant :

```
1 //...
2 class ItemException {
3  std::string info;
4 public:
5  ItemException(const std::string& str):info(str){}
6  std::string getInfo() const { return info; }
7 };
```

1. Quel type de lien envisagez-vous entre un objet `Item` et le tableau pointé par son attribut `tab` ? Quelles en sont les conséquences au niveau du code à développer pour la classe `Item` ? Justifiez votre réponse.
2. Quel type de lien envisagez-vous entre un objet `Item` et les objets `Item` qui lui sont associés ? Quelles en sont les conséquences au niveau du code à développer pour la classe `Item` ? Justifiez votre réponse.
3. Définir la classe `Item` en ne définissant pour l'instant que le constructeur de façon **`inline`**. Les autres méthodes seront simplement déclarées pour l'instant. Définir aussi une méthode `getNbAssociatedItems` qui permet de connaître le nombre d'objets `Items` associés avec l'objet sur lequel s'applique la méthode. Définir aussi une méthode `getName()` qui permet de connaître le nom d'un objet `Item`. Le type de retour de cette méthode peut-il être de type `std::string` ? de type **`const`** `std::string` ? de type `std::string&`, de type **`const`** `std::string&` ? Justifiez.
4. Implémentez la méthode `addAssociatedItem`. Si l'objet `Item` dont la référence est reçue en argument est déjà associé à l'objet `Item` sur lequel s'applique la méthode, l'ajout sera ignoré. Le paramètre de la méthode aurait-il pu être de type `Item` (au lieu de `Item&`) ? Expliquez.
5. Implémentez la méthode `removeAssociatedItem`. Si l'objet `Item` dont la référence est reçue en argument n'est pas associé à l'objet `Item` sur lequel s'applique la méthode, elle déclenchera une exception.
6. Est-il nécessaire de définir un destructeur pour la classe `Item` ? Implémenter cette méthode uniquement si elle est nécessaire. Interdire la duplication d'un objet `Item`.

7. On veut offrir la possibilité à un utilisateur de pouvoir accéder à l'ensemble des `Item` associés à un objet `Item` tout en respectant le principe d'encapsulation. Proposez une définition de cette fonctionnalité.

Question 2

1. La classe `Item` est destinée à être spécialisée bien que les méthodes `getName`, `addAssociatedItem` et `removeAssociatedItem` ne devraient pas être redéfinies dans ses sous-classes. Apportez vous des changements à votre classe ? Si oui, le(s)quel(s) ? Justifiez votre réponse.
2. On souhaite que toute sous-classe de la classe `Item` implémente une méthode de prototype `std::string toString()const;`. Le comportement de cette méthode n'est pas connue lors de l'implémentation de la classe `Item`. Ce comportement sera précisé dans les sous-classes de la classe `Item`. Quelle est la nature de cette méthode ? Quelle(s) conséquence(s) cela engendre t-il sur la classe `Item`. Modifier la définition de la classe `Item` en conséquence.

   On veut maintenant développer un module qui permet de gérer un ensemble d'étiquettes (des tags) sur des fichiers. Pour cela on implémente la classe `TagManager` (définie dans les fichiers `tagManager.h` et `tagManager.cpp`) dont voici une définition partielle :

```
1  class TagManager {
2   Tag** tags;
3   File** files;
4   unsigned int nbTags;
5   unsigned int nbFiles;
6   unsigned int getTagIndice(const std::string& n) const;
7   unsigned int getFileIndice(const std::string& n) const;
8   unsigned int addTag(const std::string& n);
9   unsigned int addFile(const std::string& n);
10 public:
11  void addTagOnFile(const std::string& t, const std::string& f);
12 };
```

tag.h

   Pour représenter les tags et les fichiers, la classe `TagManager` utilise deux classes `Tag` et `File` qui spécialisent toutes les deux la classe `Item`. L'attribut `tags` (respectivement `files`) pointe sur un tableau alloué dynamiquement qui contient des valeurs de type `Tag*` (resp. `File*`).
   La méthode `addTagOnFile` permet d'ajouter un tag de nom `t` à un fichier de nom `f`. Lorsqu'un message `addTagOnFile(t,f)` est envoyé à un objet `TagManager`, la méthode recherche s'il existe déjà un objet `x` de type `Tag`, dont l'adresse est stockée dans le tableau pointé par `tags`, et dont le nom est égal à `t`. S'il n'en existe pas encore, elle en crée un dynamiquement et stocke son adresse dans le tableau pointé par `tags`.
   Elle recherche ensuite s'il existe déjà un objet `y` de type `File`, dont l'adresse est stockée dans le tableau pointé par `files` et dont le nom est égal à `f`. S'il n'en existe pas encore, elle en crée un dynamiquement et stoke son adresse dans le tableau pointé par `files`.
   Enfin, elle envoie des messages à l'objet `x` pour lui associer l'objet `y` et à l'objet `y` pour lui associer l'objet `x`.
   Pour implémenter ce comportement, la méthode `addTagOnFile` utilise les méthodes privées suivantes. La méthode `getTagIndice` (resp. `getFileIndice`) qui renvoie l'indice dans le tableau `tags` (resp. `files`) d'une adresse qui pointe sur un objet `Tag` (resp. `File`) dont le nom est `n`. Si une telle adresse n'existe pas, la méthode renvoie la valeur `nbTags` (resp. `nbFiles`). La méthode `addTag` (resp. `addFile`) qui crée dynamiquement un objet `Tag` (resp. `File`) de nom `n`, ajoute l'adresse de cet objet dans le tableau pointé par `tags` (resp. `files`), et renvoie l'indice de cette adresse dans le tableau pointé par `tags` (resp. `files`).

3. Proposer des définitions complètes des classes `Tag` et `File` de façon à pouvoir répondre à la fonctionnalité décrite ci-dessus. Définissez toute méthode qui vous parait nécessaire comme bon vous semble.
4. Quel type d'association existe-t-il entre la classe `TagManager` et les classes `Tag` et `File` ? Justifiez votre réponse. Dessiner un diagramme de classes qui fait intervenir les classes `Item`, `TagManager`, `Tag` et `File` en faisant clairement apparaître les relations qui existent entre ces différentes classes, et en ajoutant tous les détails que vous pouvez déduire du sujet. Accompagnez ce diagramme de classes avec un diagramme de séquence (cohérent avec le diagramme de classes) qui illustre le scénario d'un utilisateur ajoutant le tag `"POO"` au fichier `"LO21.pdf"`.

Solution de l'exercice 72

Question 1

1. Un objet `Item` compose le tableau pointé par `tab`. Un objet `item` est donc responsable de la création et de la destruction de son tableau de pointeurs.
2. Il est précisé dans le sujet que les cycles de vie de deux objets `Item` associés sont indépendants. On aura donc une agrégation réflexive entre la classe `Item` et elle-même.
3.

```
1  //...
2  class Item {
3   std::string name;
4   Item** tab;
5   unsigned int nb;
6   unsigned int max;
7  public:
8   Item(const std::string& n):name(n),tab(nullptr),nb(0),max(0){}
9   const std::string& getName() const { return name; }
10  unsigned int getNbAssociatedItems() const { return nb; }
11  ~Item();
12  void addAssociatedItem(Item& item);
13  void removeAssociatedItem(Item& item);
14 };
```

item.h

Le type de retour de `getName` peut être de type `std::string` et `const std::string`; le passage par valeur fera une recopie de l'attribut `name`; le fait que cette recopie soit **`const`** ou non ne change rien. Il peut être aussi `const std::string&`; la référence renvoyée sera initialisée avec l'attribut `name` et sera valide puisque l'attribut existe toujours à l'issue de la méthode. Par contre, il ne devrait pas être de type `std::string&` car la méthode est **`const`** et ne devrait donc pas fournir un moyen d'accès en écriture à l'un de ses attributs.

4.

```
1  //...
2  void Item::addAssociatedItem(Item& item){
3   unsigned int i=0;
4   while (i<nb && tab[i]!=&item) i++;
5   if (i==nb){
6    if (nb==max){
7     Item** newtab=new Item* [max+5];
8     for(unsigned int i=0; i<nb; i++) newtab[i]=tab[i];
9     Item** old=tab;
10    tab=newtab;
11    max+=5;
12    delete[] old;
13   }
14   tab[nb++]=&item;
15  }
16 }
```

item.cpp

Le paramètre de la méthode n'aurait pas pu être de type `Item`. En effet, ceci serait un passage par valeur. L'objet reçu en argument serait alors un objet local à la méthode et non l'objet dont on doit stocker l'adresse.

5.
```
//...
void Item::removeAssociatedItem(Item& item){ // ne pas faire dans l'exam
 unsigned int i=0;
 while (i<nb && tab[i]!=&item) i++;
 if (i==nb) throw ItemException("item inexistant");
 nb--;
 while(i<nb) { tab[i]=tab[i+1]; i++; }
}
```

item.cpp

6. Un destructeur est nécessaire pour libérer le tableau alloué dynamiquement pointé par `tab`.

```
//...
class Item {
//...
public:
 ~Item() { nb=0; max=0; delete[] tab; }
};
```

item.h

Pour interdir la duplication, il suffit de placer une déclaration du constructeur de recopie et de l'opérateur d'affectation de la partie privée :

```
//...
class Item {
//...
private:
 Item(const Item&);
 Item& operator=(const Item&);
};
```

item.h

7. Pour offrir la possibilité à un utilisateur de pouvoir accéder à l'ensemble des `Item` associés à un objet `Item` tout en respectant le principe d'encapsulation, il faut implémenter le design pattern `Iterator`. Voici une définition possible :

```
//...
class Item {
//...
public:
 class Iterator {
  Item** current;
  unsigned int nb;
  Iterator(Item** c, unsigned int n):nb(n){}
  friend class Item;
 public:
  bool ok() const { return nb>0; }
  void next() { if (ok()) { nb--; current++; } }
  Item& item() const { if (ok()) return **current; else throw ItemException
      ("indirection d'un iterateur nul"); }
  Iterator():current(nullptr),nb(0){}
```

```
15  };
16  Iterator getIterator() { return Iterator(tab,nb); }
17 };
```

item.h

Question 2

1. Le destructeur devrait être **`virtual`** afin de respecter le principe de substitution. En effet, si on détruit un objet par l'intermédiaire d'un pointeur `Item*` qui pointe en réalité sur un objet d'une classe dérivée, il faut que le destructeur de la classe dérivée soit appelé. Cela n'est possible que si le destructeur a été explicitement déclaré polymorphique dans la classe `item` :

```
1 //...
2 class Item {
3 //...
4 public:
5  virtual ~Item();
6 };
```

item.h

```
1 //...
2 Item::~Item(){ nb=0; max=0; delete[] tab; }
```

item.cpp

   Les méthodes `getName`, `addAssociatedItem` et `removeAssociatedItem` n'étant pas destinées à être redéfinies, elle devraient rester non-virtuelles. Cela reste bien évidemment une indication aux concepteurs des sous-classes.
2. Le comportement de la méthode `toString` n'étant pas encore connu lors de l'implémentation de la classe `Item`, la méthode `toString` sera virtuelle pure. La classe `Item` sera donc abstraite. Implémentation.

```
1 class Item {
2 //...
3 public:
4  virtual std::string toString() const=0;
5 };
```

item.h

3. Les classes `Tag` et `File` héritent directement et publiquement de la classe `Item`. Pour être fonctionnelles, ces classes doivent être concrètes (c'est à dire, non-abstraites). Elles doivent donc proposer une implémentation de la méthode `toString`. Cette méthode doit donc être déclarée dans la définition des sous-classes et être définie dans le fichier source (puisqu'elle est virtuelle).

```
1 class Tag : public Item {
2  Tag& operator=(const Tag&);
3 public:
4  Tag(const std::string& n):Item(n){}
5  std::string toString() const;
6 };
7
```

```
8  class File : public Item {
9   File& operator=(const File&);
10 public:
11  File(const std::string& n):Item(n){}
12  std::string toString() const;
13 };
```

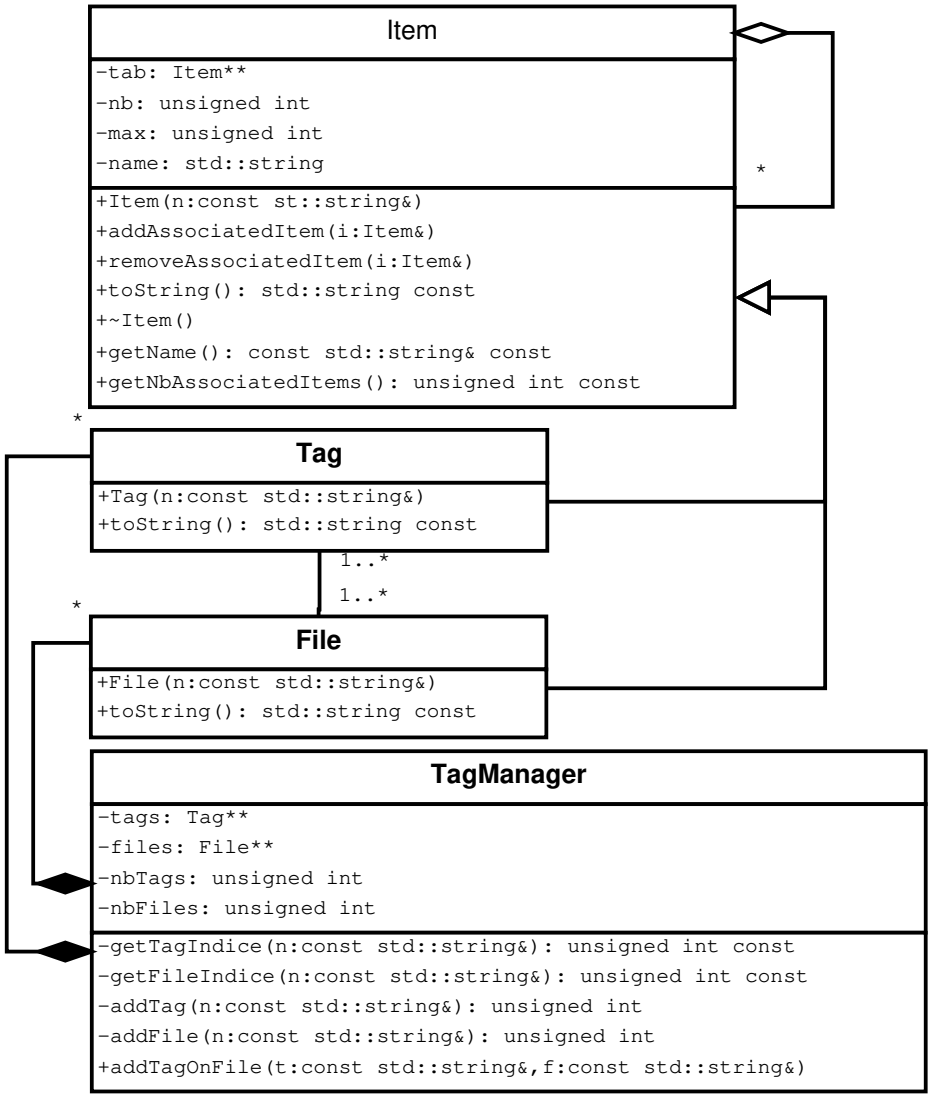
tagManager.h

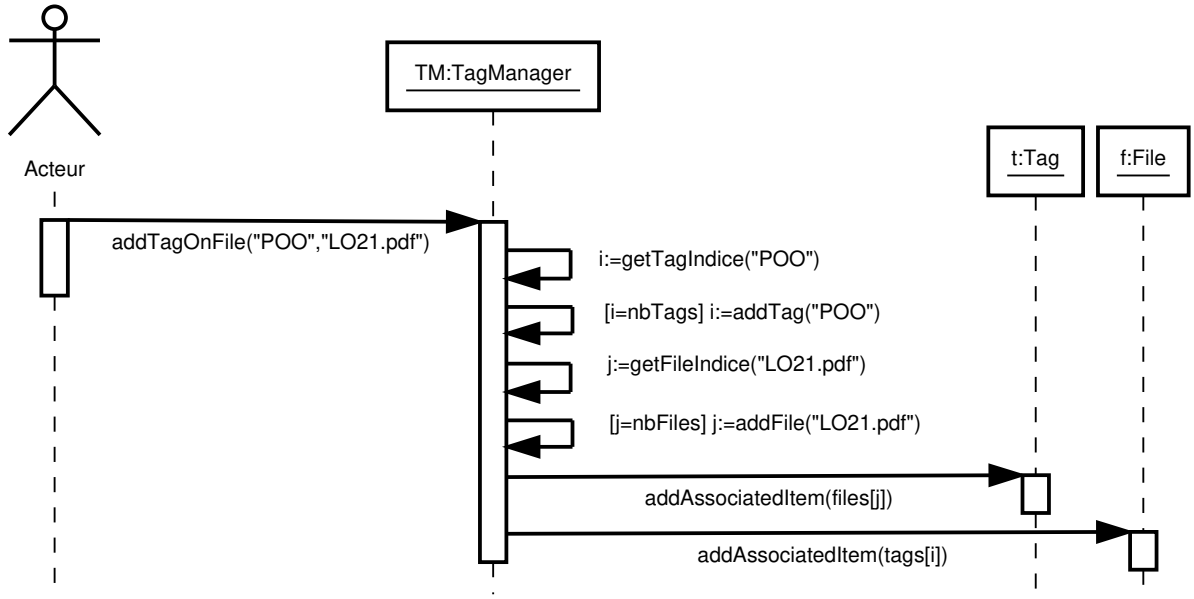
Ici, on propose une définition de ces méthodes en ajoutant `Tag :` ou `File :` devant le nom de l'item pour les différencier. Notons cependant, que l'on aurait pu utiliser le même code dans les deux méthodes.

```
1 std::string Tag::toString() const {
2  return std::string("Tag : ")+getName();
3 }
4
5 std::string File::toString() const {
6  return std::string("File : ")+getName();
7 }
```

tagManager.cpp

4. Le texte indique clairement que la classe `TagManager` gère le cycle de vie des objets `Tag` et `File` utilisés par le module. On a donc une composition entre la classe `TagManager` et les classes `Tag` et `File`.




Acteur
TM:TagManager
t:Tag
f:File
addTagOnFile("POO","LO21.pdf")
i:=getTagIndice("POO")
[i=nbTags] i:=addTag("POO")
j:=getFileIndice("LO21.pdf")
[j=nbFiles] j:=addFile("LO21.pdf")
addAssociatedItem(files[j])
addAssociatedItem(tags[i])

Exercice 73 - Angles et coordonnées géographiques

Question 1 - Angles

Les angles sont souvent exprimés en utilisant le système sexagésimal. Les angles se représentent alors avec 3 entiers positifs exprimés en degrés (°), minutes d'arc (′) et secondes d'arc (″). L'unité de base est le degré (1 tour complet = 360°). Les sous-unités du degré sont la minute d'arc ($1° = 60'$) et la seconde d'arc ($1' = 60''$). Soit un angle $x°y'z''$ exprimé dans ce système. Les valeurs y (minutes d'arc) et z (secondes d'arc) prennent toujours une valeur entière dans $\{0, \ldots, 59\}$. La valeur x (degrés) prend toujours une valeur dans $\{0, \ldots, 360\}$.

Il existe aussi **plusieurs autres unités d'angle**. Ainsi, le *Degré Décimal* est aussi beaucoup utilisé. Un angle de $x°y'z''$ est égal à la valeur réelle $x + y/60 + z/3600°$ (en degrés décimaux). Par exemple, $31°13'19'' = 31 + 13/60 + 19/3600 = 31.221\,944°$. On utilise aussi souvent le *radian* (symbolisé par rad), sachant que $x°$ (en degré décimaux) est égal à $\frac{x\pi}{180}$ rad.

Pour la suite, on supposera que l'on dispose d'un fichier d'entête `angle.h` et d'un fichier source `angle.cpp`. À chaque fois que vous répondrez à une question où du code C++ doit être écrit, indiquez dans lequel des 2 fichiers il doit être placé.

Questions :

1. Définir une classe `Angle` composée de trois attributs de type **`unsigned int`** nommés `deg`, `min` et `sec` représentant respectivement les parties degré, minute d'arc et seconde d'arc d'un angle. Définir également un seul constructeur qui permet de construire un objet `Angle` avec 1, 2 ou 3 arguments en **initialisant** les attributs de façon à respecter l'exemple suivant :

```
1 Angle a1(18); //deg=18, min=0, sec=0
2 Angle a2(45,32); //deg=45, min=32, sec=0
3 Angle a3(31,13,19); //deg=31, min=13, sec=19
```

2. Qu'est ce que le *principe d'encapsulation* ? S'assurer que le principe d'encapsulation a bien été respecté dans la question précédente. En s'inspirant du texte du sujet, expliquer quels sont les intérêts de respecter le principe d'encapsulation dans le cas de la classe `Angle`.
3. Que signifie déclarer une méthode ou une fonction ? Que signifie définir une méthode ou une fonction ?
4. Dans la classe `Angle`, définir de façon **`inline`** les méthodes `degre()`, `minute()`, `seconde()` qui permettent de connaître la valeur des différents attributs. Ces méthodes doivent-elles être **`const`** ? Ces méthodes doivent-elles être **`static`** ? Expliquer. Comment appelle-t-on couramment ces méthodes ?
5. Dans la classe `Angle`, définir un méthode `toDegreDecimal()` qui renvoie la valeur d'un angle exprimée en degrés décimaux. Cette méthode renverra une valeur de type **`double`** et **ne sera pas** définie de façon **`inline`**.
6. Définir (de façon **`inline`** ou non) la méthode `toRad()` qui renvoie la valeur (de type **`double`** d'un angle exprimée en radians. Le type de retour peut-il être de type **`double`** ? de type **`const double`** ? de type **`double&`** ? de type **`const double&`** ? Expliquer à chaque fois.
7. Surcharger l'opérateur << de façon à pouvoir écrire un objet `Angle` sur un flux `ostream` sous la forme $x°y'z''$.
8. Ajouter dans la classe `Angle` la définition des trois attributs `degMax`, `minMax`, `secMax`. Ces attributs seront **`static`** et seront respectivement initialisés avec les valeurs 360, 59 et 59. Expliquer pourquoi ces attributs doivent être **`static`** ?
9. Définir trois méthodes qui permettent de connaître la valeur des attributs `degMax`, `minMax`, `secMax` sans avoir besoin de créer un objet `Angle`. Écrire un exemple d'instruction qui permet d'afficher une de ces valeurs sur le flux `ostream` standard `cout` (sans créer d'objet `Angle`).
10. Redéfinissez le constructeur pour qu'il vérifie que si `x`, `y` et `z` sont les arguments fournis pour construire l'angle $x°y'z''$, on a `x`$\leq$`degMax`, `y`$<$`minMax` et `z`$<$`secMax`. Si au moins une valeur

n'est pas conforme, le constructeur devra déclencher une exception d'un type que vous choisirez.

11. Transformer le bloc d'instructions suivant de manière à traiter l'exception qui devrait être déclenchée.

```
1 Angle a(56,78,45);
2 cout<<a<<"\n";
```

12. Est-il nécessaire de définir un destructeur, un constructeur de recopie et un opérateur d'affectation pour la classe `Angle` ? Expliquer.

Question 2 - Mon GPS et moi

La **latitude** est un angle qui exprime le positionnement nord-sud d'un point sur la Terre. La latitude prend une valeur entre 0° et 90° et est associée à une direction vers le nord (*N*) ou vers le sud (*S*). Sur la Terre, les points de même latitude forment un cercle appelé parallèle. Les points situés à la latitude 0° forment l'équateur.

La **longitude** est un angle qui exprime le positionnement est-ouest d'un point sur la Terre. Tous les points situés à la même longitude coupent la surface de la Terre sur un demi-cercle dont le centre est le centre de la Terre, l'arc allant d'un pôle à l'autre. Un tel demi-cercle est appelé méridien. La longitude prend une valeur entre 0° et 180° et est associée à une direction vers l'ouest (*W* pour west en anglais) ou vers l'est (*E*). Les points sur la Terre qui ont une longitude de 0° sont les points du méridien référence qui s'appelle le *méridien de Greenwich*.

En combinant les deux angles (latitude et longitude), la **position d'un point sur la surface de la Terre** peut être spécifiée. Par exemple les coordonnées géographique de Shanghai sont : (Latitude : 31°13′19″ *N* ; Longitude : 121°27′29″ *E*).

Questions :

1. Définir une classe `GeoCoordonnee` comportant quatre attributs `latitude`, `dirLat`, `longitude` et `dirLong`. Les attributs `latitude` et `longitude` seront de type `Angle`. Les attributs `dirLat` et `dirLong` seront de type `Direction` où `Direction` est une énumération supposée définie dans le fichier `angle.h` comme suit :

```
1 enum Direction { N, S, W, E };
```

   On proposera un constructeur à 4 arguments (du même type que les attributs) pour initialiser les attributs d'un objet `GeoCoordonnee` en s'inspirant de l'exemple suivant :

```
1 GeoCoordonnee Shanghai(Angle(31,13,19),N,Angle(121,27,29),E);
2 GeoCoordonnee VinaDelMare(Angle(33,9,12),S,Angle(71,33),W);
```

   Rappel : Une énumération est un type défini par l'utilisateur qui se compose d'un ensemble d'identifcateurs appelées *énumérateurs*. À chaque énumérateur est assignée une valeur entière qui correspond à sa place dans l'ordre des valeurs de l'énumération. Les énumérateur sont visibles dans toute la portée où ils ont été déclarés. Une variable de type énumération peut être initialisée ou affectée avec n'importe lequel des énumérateurs.

2. Quelle relation lie les classes `GeoCoordonnee` et `Angle` ? Expliquer.

   On veut créer une classe `Chemin` afin de gérer des trajets générés par un système GPS. Un objet `Chemin` représente une suite de coordonnées géographiques correspondantes aux différentes étapes d'un trajet. Cette classe dispose d'un unique constructeur sans argument permettant de générer un trajet initialement vide. L'opérateur << permet d'ajouter des coordonnées à un objet `Chemin` comme dans l'exemple suivant :

```
1 Chemin t; // chemin initialement vide
2 // ajout de trois coordonnées dans le trajet
3 t<<GeoCoordonnee(Angle(49,25,4),N,Angle(2,49,34),E)
4  <<GeoCoordonnee(Angle(48,51,24),N,Angle(2,21,8),E)
5  <<GeoCoordonnee(Angle(45,46,38),N,Angle(3,5,13),E);
```

Un objet `Chemin` ne peut pas être affecté. On peut cependant construire un nouvel objet `Chemin` à partir d'un objet `Chemin` existant. On doit pouvoir parcourir séquentiellement les étapes du `Chemin` sans exposer la structure de données de la classe.

3. Quelle relation lie les classes `Chemin` et `GeoCoordonnee` ? Expliquer.
4. Définir entièrement cette classe de façon à implémenter toutes les spécifications décrites ci-avant.
5. En utilisant les fonctionnalités décrites ci-avant de la classe `Chemin`, surcharger l'opérateur `<<` de façon à pouvoir écrire les différentes étapes d'un objet `Chemin` sur un flux `ostream`.
6. Dessiner un diagramme de classes qui représente l'ensemble des classes décrites dans ce sujet. Reporter le plus de détails possible en se basant sur les indications du sujet et en faisant apparaître les attributs et les méthodes de chaque classe, ainsi que les associations qui existent entre ces classes.

Solution de l'exercice 73

Question 1

1.
```
1 class Angle {
2 private :
3  unsigned int deg;
4  unsigned int min;
5  unsigned int sec;
6 public :
7  Angle(unsigned int d, unsigned int m=0, unsigned int s=0):
8    deg(d),min(m),sec(s){}
9 };
```

angle.h

On notera que le constructeur peut être défini **inline** (dans `angle.h`) ou non. Cependant, les deux valeurs par défaut pour les paramètres du constructeur doivent forcément se trouver dans la définition de la classe. En cas de définition non-**inline** (*i.e.* dans `angle.cpp`), les valeurs par défaut ne doivent pas apparaître dans la définition de la méthode.
Normalement, une initialisation des attributs implique l'utilisation de la notation avec `":"`, sinon il s'agirait d'une affectation des attributs.

2. Le principe d'encapsulation consiste à empêcher l'accès aux données : les données sont encapsulées et leur accès ne se fait que par le biais de méthodes.
L'intérêt ici d'appliquer le principe d'encapsulation est de protéger l'utilisateur d'un éventuel changement de structure de données. En effet, il est possible d'implémenter une classe `Angle` en utilisant d'autres unités, par exemple, une seule valeur de type **double** exprimée en degrés décimaux.
3. Déclarer une méthode ou une fonction consiste à fournir uniquement le prototype (l'entête) de la méthode ou de la fonction. Définir une méthode ou une fonction consiste à fournir l'entête et le corps de la fonction (les instructions composant la fonction ou la méthode), *i.e.* à définir le comportement de la méthode.

4.
```
1 class Angle {
2 private :
3 /*...*/
4 public :
5  unsigned int degre() const { return deg; }
6  unsigned int minute() const { return min; }
7  unsigned int seconde() const { return sec; }
8 };
```

angle.h

Puisque les méthodes doivent être **inline**, leur définition est fournie à l'intérieur de la définition de la classe. Ces méthodes devraient être **const** puisqu'elles ne modifient pas la valeur des attributs. Elle ne doivent pas être **static** puisque chaque objet `Angle` possèdent ses propres valeurs `deg`, `min`, `sec`. Ces méthodes s'appellent des accesseurs.

5.
```
1 class Angle {
2 private :
3 /*...*/
4 public :
```

```
5  double toDegreDecimal() const;
6 };
```

angle.h

```
1 double Angle::toDegreDecimal() const {
2  return deg+min/60.0+sec/3600.0;
3 }
```

angle.cpp

Cette méthode doit aussi être **const**. Elle doit être déclarée dans le fichier d'entête (`angle.h`) et définie dans le fichier source (`angle.cpp`).

6.

```
1 class Angle {
2 private :
3 /*...*/
4 public :
5  double toRad() const { return toDegreDecimal()*3.14159/180; }
6 };
```

angle.h

Cette méthode doit aussi être **const**. Le type de retour peut être **double** ou **const double** (passage par valeur). Par contre, il ne devrait pas être **double**`&` ou **const double**`&` puisque sinon on renverrait une référence sur une variable automatique locale à la fonction et qui serait invalide une fois la fonction terminée.

7.

```
1 #include<iostream>
2 class Angle {
3 /*...*/
4 };
5
6 std::ostream& operator<<(std::ostream& f, const Angle& a);
```

angle.h

```
1 /*...*/
2 std::ostream& operator<<(std::ostream& f, const Angle& a){
3  f<<a.degre()<<"°"<<a.minute()<<"'"<<a.seconde()<<"''";
4  return f;
5 }
```

angle.cpp

L'opérateur **operator**`<<` ne peut pas être ici surchargée en tant que méthode de classe (puisque l'argument de gauche est un objet de la classe `ostream`. C'est donc une fonction avec deux paramètres et non une méthode de classe. Le type de la valeur de retour et du premier paramètre doit être `ostream&` (un objet `ostream` ne peut pas être transmis par valeur). On fera bien attention à ce que l'objet `ostream` transmis en référence soit renvoyé à l'issue de la fonction. On fera aussi attention à la bonne gestion du namespace `std` (avec l'opérateur de résolution de portée ou avec l'utilisation d'une **using**-instruction.

8.

```
1 #include<iostream>
2 class Angle {
3 /*...*/
```

```
4 private:
5  static unsigned int degMax;
6  static unsigned int minMax;
7  static unsigned int secMax;
8 };
```

angle.h

```
1 /*...*/
2 unsigned int Angle::degMax=360;
3 unsigned int Angle::minMax=59;
4 unsigned int Angle::secMax=59;
```

angle.cpp

Les attributs **static** doivent être initialisés dans le fichier source (`angle.cpp`). Ils pourraient (et ce serait même préférable) être de type **const unsigned int**. Notons que dans ce cas (et seulement dans ce cas), la nouvelle norme prévoit que les attributs **static** constants puisse être initialisés directement dans la dédinition de la classe (dans `angle.h`). Ces attributs doivent être **static** car ils sont des caractéristiques communes et partagés par tous les objets de la classe `Angle`.

9.

```
1 #include<iostream>
2 class Angle {
3 /*...*/
4 private:
5  static unsigned int getDegMax() { return degMax; }
6  static unsigned int getMinMax() { return minMax; }
7  static unsigned int getSecMax() { return secMax; }
8 };
```

angle.h

Ces méthodes doivent également être **static** afin de pouvoir les utiliser sans objet `Angle`. Elles peuvent être **inline** ou non. Elles ne doivent pas être **const** (cela n'a aucun sens dans le cas des méthodes **static** puisqu'elles n'ont pas d'argument implicite). L'intruction `std ::cout<<Angle::getDegMax();` permet par exemple d'afficher la valeur de `degMax`. On remarquera, qu'elle n'utilise pas d'objet `Angle` mais qu'elle utilise l'opérateur de résolution de portée `Angle::`.

10.

```
1 class Angle {
2 /*...*/
3 public :
4  Angle(unsigned int d, unsigned int m=0, unsigned int s=0):
5    deg(d),min(m),sec(s){
6     if (deg>getDegMax()||min>getMinMax()||sec>getSecMax())
7      throw "erreur : valeur hors limite";
8  }
9 };
```

angle.h

Ici, pour faire simple, on déclenche une exception de type **const char*** mais elle peut tout à fait être d'un autre type.

11.

```
1 try {
```

```
 Angle a(56,78,45);
 cout<<a<<"\n";
}catch(const char* e){
 cout<<e<<"\n";
}
```

Dans le **catch**, on fera attention à ce que le type de l'exception récupére soit compatible avec celle déclenchée dans la question précédente.

12. Il n'est pas nécessaire de faire un constructeur de recopie ou un opérateur d'affectation pour la classe `Angle`. Ceux générés par défaut conviennent tout à fait puisque tous les attributs sont automatiques.

Question 2

1.
```
enum Direction { N, S, W, E };
class GeoCoordonnee {
 Angle latitude;
 Direction dirLat;
 Angle longitude;
 Direction dirLong;
public:
 GeoCoordonnee(Angle lat, Direction dLat, Angle long, Direction dLong):
  latitude(lat), dirLat(dLat), longitude(dLong), dirLong(dLong) {}
};
```

angle.h

On remarquera que l'initialisation avec `":"` est ici obligatoire puisque la classe `Angle` ne dispose pas de constructeur sans argument. Notons, que l'on pouvait utiliser l'autre constructeur défini dans la classe `Angle`.

2. Il s'agit d'une composition puisque les attributs d'une classe sont toujours composés par la classe.
3. L'énoncé n'est pas sufisamment précis pour que l'on puisse déterminer s'il s'agit d'une composition ou d'une aggrégation. Aussi, les deux sont possibles. Il faudra par contre être vigilant à fournir une implémentation cohérente avec son choix.

4.
```
/*...*/
class Chemin{
 GeoCoordonnee** tab;
 unsigned int nb;
 unsigned int nbmax;
 Chemin& operator=(const Chemin& c);
public:
 GeoCoordonnee():tab(nullptr),nb(0),nbmax(0){}
 Chemin& operator<<(const GeoCoordonnee& g);
 ~Chemin() {
  for(unsigned int i=0; i<nb; i++) delete tab[i]; // composition uniquement
  delete[] tab; // composition + agrégation
 }
 Chemin(const Chemin& c);
```

```
15  class iterator {
16    GeoCoordonnee** current;
17   unsigned int nb;
18   friend class Chemin;
19   iterator(GeoCoordonnee** c, unsigned int n):current(c),nb(n){}
20  public:
21   iterator():current(nullptr),nb(0){}
22   const GeoCoordonnee& operator*() const { return **current; }
23   iterator& operator++() { current++; return *this; }
24   bool operator!=(iterator it) const { return current!=it.current; }
25  };
26  iterator begin() const { return iterator(tab,nb); }
27  iterator end() const { return iterator(tab+nb,0); }
28 };
```

angle.h

```
1  /*...*/
2  Chemin& Chemin::operator<<(const GeoCoordonnee& g){
3   if (nb==nbmax) { // le tableau doit être agrandit
4    // creation d'un tableau plus grand
5    GeoCoordonnee* newtab=new GeoCoordonnee*[nbmax+10];
6    // copie des adresses des objets déjà présents dans le tableau d'origine
7    for(unsigned int i=0; i<nb; i++) newtab[i]=tab[i];
8    // mise à jour des attributs
9    GeoCoordonnee* old=tab;
10   tab=newtab;
11   nbmax+=10;
12   //liberation de l'ancien tableau
13   delete[] old;
14  }
15  // si composition : on doit créer un nouvel objet par recopie
16  tab[nb++]=new GeoCoordonnee(g);
17  // si agrégation, on ajoute simplement l'adresse
18  tab[nb++]=const_cast<GeoCoordonnee&>(&g);
19  return *this; // permettre l'insertion multiple
20
21 }
22
23 Chemin::Chemin(const Chemin& c):tab(new GeoCoordonnee*[c.nb]),nb(c.nb),
       nbmax(c.nb){
24  for(unsigned int i=0; i<nb; i++) {
25     tab[i]=new GeoCoordonnee(*c.tab[i]); // si composition
26    tab[i]=c.tab[i]; // si agrégation
27  }
28 }
```

angle.cpp

Notons que l'opérateur d'affectation devrait être déclaré (inutile de le définir) dans la partie privée afin d'empêcher l'affectation entre objets `Chemin`.
Notons aussi la présence d'un destructeur afin de libérer le tableau dynamique d'objets `GeoCoordonnee` construit au cours de la durée de vie d'un objet `Chemin`. Ce tableau est créé

ou agrandit dans la méthode **operator**<< quand cela est nécessaire. Alors que le tableau est forcément composé par un objet `Chemin` (il est donc libéré), les objets dont les adresses sont contenus dans le tableau sont libérés seulement si on a décidé que les objets `Geocoordonnee` étaient composé par un objet `Chemin`.
La méthode **operator**<< doit renvoyer une référence sur l'objet qui appelle la méthode afin de faire de multiple insertions d'objet `GeoCoordonnee`. Dans le cas d'une composition des objets `GeoCoordonnee`, il faut créer un nouvel objet par recopie de l'argument. Dans le cas d'une agrégation des objets `GeoCoordonnee`, il faut simplement stocker l'adresse de l'argument. On note que normalement, il faudrait une conversion **const_cast** pour que cela soit possible.
Notons la présence de la redéfinition d'un constructeur de recopie puisque celui généré par défaut par le compilateur ne convient pas et qu'il faut pouvoir construire un objet `Chemin` à partir d'un autre objet `Chemin`. La même problématique de création d'un nouvel objet ou de la simple recopie de l'adresse de l'objet existant existe aussi au niveau de constructeur de recopie suivant que l'on a choisi une composition ou une agrégation.
Afin de parcourir séquentiellement un objet `Chemin` sans exposer la structure de données, on propose d'implémenter le design pattern iterator. On notera que plusieurs interfaces sont possibles. On a choisit ici l'implémentation d'un itérateur uniquement en lecture afin traiter la question suivante.

5.

```
1 /*...*/
2 class Chemin{
3 /*...*/
4 };
5 std::ostream& operator<<(std::ostream& f, Chemin& c);
```

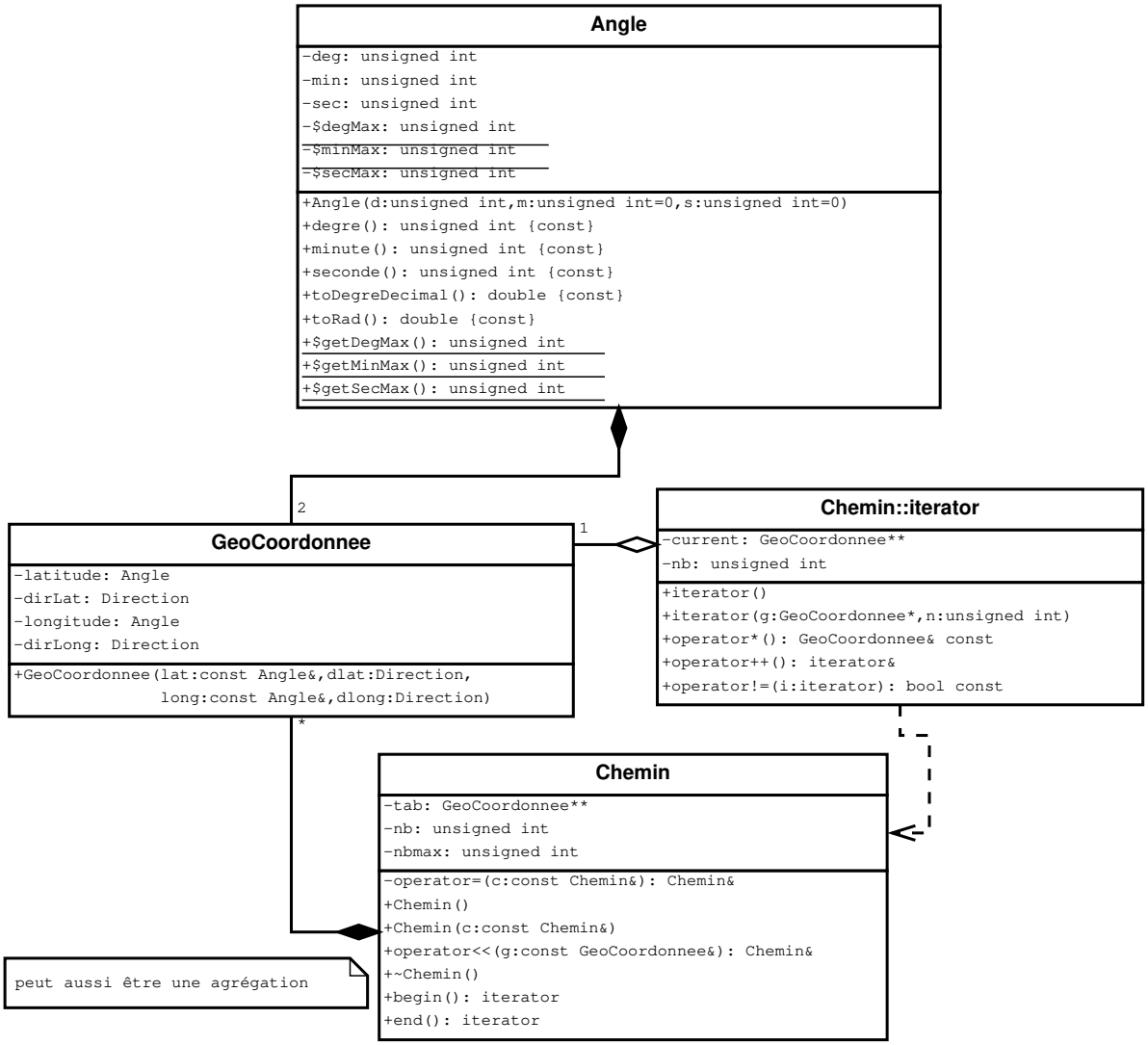
angle.h

```
1 /*...*/
2 std::ostream& operator<<(std::ostream& f, const Chemin& c){
3  for(Chemin::iterator it=c.begin(); it!=c.end(); ++it) f<<(*it)<<";";
4  return f;
5 }
```

angle.cpp

Il s'agit ici d'utiliser la classe d'itérateur définie dans la question précédente.

6.


Angle
-deg: unsigned int
-min: unsigned int
-sec: unsigned int
-$degMax: unsigned int
-$minMax: unsigned int
-$secMax: unsigned int
+Angle(d:unsigned int,m:unsigned int=0,s:unsigned int=0)
+degre(): unsigned int {const}
+minute(): unsigned int {const}
+seconde(): unsigned int {const}
+toDegreDecimal(): double {const}
+toRad(): double {const}
+$getDegMax(): unsigned int
+$getMinMax(): unsigned int
+$getSecMax(): unsigned int
2
GeoCoordonnee
-latitude: Angle
-dirLat: Direction
-longitude: Angle
-dirLong: Direction
+GeoCoordonnee(lat:const Angle&,dlat:Direction,
long:const Angle&,dlong:Direction)
1
Chemin::iterator
-current: GeoCoordonnee**
-nb: unsigned int
+iterator()
+iterator(g:GeoCoordonnee*,n:unsigned int)
+operator*(): GeoCoordonnee& const
+operator++(): iterator&
+operator!=(i:iterator): bool const
*
Chemin
-tab: GeoCoordonnee**
-nb: unsigned int
-nbmax: unsigned int
-operator=(c:const Chemin&): Chemin&
+Chemin()
+Chemin(c:const Chemin&)
+operator<<(g:const GeoCoordonnee&): Chemin&
+~Chemin()
+begin(): iterator
+end(): iterator
peut aussi être une agrégation

Exercice 74 - STELLARVIEW

Introduction Le but de cet exercice est de développer des composants destinés à faire parti de STELLARVIEW, un logiciel de visualisation de cartes astronomiques. Une étoile est un corps céleste qui émet sa propre lumière. Une constellation est un ensemble d'étoiles (au moins 2) auquel on a donné un nom (ces étoiles projetées sur la voûte céleste forment une figure particulière). Une étoile peut posséder différents noms selon le système utilisé. De même, les constellations peuvent posséder plusieurs noms selon la civilisation qui les ont imaginés. Par exemple les noms « Aldébaran », « α Tau », « HR 1457 » ou « HD 29139 » sont autant de noms qui désignent la même étoile. De même, les noms « la Grande Ourse », « septem triones », « the Plough » ou « Karr kamm » sont des noms qui désignent la même constellation.

Dans l'application, on représente les étoiles par des objets de la classe `Etoile` et les constellations par des objets de la classe `Constellation`. Les classes `Etoile` et `Constellation` possèdent chacune un attribut `noms` de type `Designations`. Un objet de la classe `Designations` permet de stocker plusieurs chaînes de caractères de type `string` qui désignent un même objet (une même étoile ou une même constellation). Cette classe possède une méthode `ajouterDesignation(const string& n)` qui permet d'ajouter un nouveau nom `n` à un objet. Les classes `Etoile` et `Constellation` possèdent alors chacune deux méthodes respectivement de prototype : « `Designation& getDesignations();` » et « `const Designation& getDesignations()const;` ». Un objet `Etoile` ou `Constellation` possède au moins une désignation. Un objet `Etoile` ou `Constellation` *n'est pas duplicable par recopie ou par affectation*. Ces classes *pourraient être spécialisées* par la suite.

Un objet `Etoile` est en plus caractérisé par un attribut `coord` de la classe `Coordonnees` et un attribut `magnitude` de type **`float`**. La classe `Coordonnees` ne sera pas détaillée dans ce sujet. On supposera que sa définition est disponible et que la classe dispose au moins d'un constructeur de recopie. La classe `Etoile` possède deux accesseurs `getCoordonnees()` et `getMagnitude()` qui permettent de connaître la valeur de ces attributs. De plus, la classe `Etoile` possède un unique constructeur prenant trois parammètres `d`, `c` et `m`, respectivement de type **`const`** `string&`, **`const`** `Coordonnees&` et **`float`** . Les paramètres `c` et `m` permettent d'initialiser les attributs `coord` et `magnitude`. Le paramètre `d` permet de créer une première désignation pour l'étoile.

Un objet `Constellation` est en plus caractérisé par un attribut `etoiles` de type `Etoile**` qui représente l'adresse d'un tableau de pointeurs d'étoiles stockant les adresses des objets `Etoile` impliqués dans la constellation. Les attributs `nb` et `nbMax` de type **`unsigned int`** représentent respectivement le nombre d'adresses réellement stockées dans le tableau et la taille du tableau `etoiles`. La méthode `ajouterEtoile` prenant en paramètre une référence d'objet `Etoile` permet d'ajouter une étoile à la constellation. L'unique constructeur de la classe `Constellation` a un paramètre de type **`const`** `string&` représentant une première désignation de la constellation et deux paramètres de type `Etoile&` référençant deux des étoiles impliquées dans la constellation.

Les objets `Etoile` et `Constellation` sont gérés par le singleton `StellarManager` qui est responsable de leur cycle de vie. Ainsi, seul le singleton `StellarManager` a le droit de créer et détruire des objets `Etoile` ou `Constellation`. La méthode `getManager()` de la classe `StellarManager` permet d'obtenir une référence sur le singleton `StellarManager`. Les méthodes de prototype « `Etoile & newEtoile(const string& d, const Coordonnees& c, float m)` » et « `Constellation& newConstellation(const string& d)` » permettent de créer une nouvelle étoile ou une nouvelle constellation. Les méthodes de prototype « **`const`** `Etoile& getEtoile(const string& d)const` » et « **`const`** `Constellation& getConstellation(const string& d)const` » permettent d'obtenir une référence **`const`** sur l'étoile ou la constellation dont une des désignations est `d`. Ces méthodes déclenchent une exception si un tel objet n'existe pas. On supposera qu'il n'existe qu'un seul objet ayant une désignation donnée.

On suppose que l'on dispose d'un fichier d'entête `StellarView.h` et d'un fichier source `StellarView.cpp`. Dans la suite, à chaque fois que vous répondrez à une question où du code `C++` doit être écrit, indiquez dans lequel des 2 fichiers il doit être placé. Les situations exceptionnelles seront gérées en utilisant la classe d'exception `StellarException` que l'on supposera déjà définie au début du fichier `StellarView.h` :

```
#ifndef STELLAR_h
#define STELLAR_h
#include<string>
#include<iostream>
using namespace std;
class StellarException{
public:
 StellarException(const string& message):info(message){}
 string getInfo() const { return info; }
private:
 string info;
};
#endif
```

Question 1

1. Quelle association lie la classe `Etoile` à la classe `Coordonnees` et à la classe `Designations` ? Quelle association lie la classe `Constellation` à la classe `Etoile` ? Quelle association lie les classes `Etoile` et `Constellation` à la classe `StellarManager` ? Justifier chaque réponse.
2. Dessiner un diagramme de classes qui représente l'ensemble des classes décrites . On fera apparaître toutes les associations qui existent entre les classes ainsi que les valeurs de multiplicité pertinentes. On fera apparaître le plus de détails possible fournis dans le texte d'introduction du sujet.
3. Pour la classe `Etoile` ou `Constellation`, quel est l'intérêt d'avoir deux méthodes `getDesignations()` ? Expliquer.
4. La classe `Etoile` nécessite-t-elle que l'on redéfinisse son destructeur ? Expliquer.
5. Définir la classe `Etoile` et l'ensemble des méthodes décrites dans le sujet en respectant toutes les propriétés décrites pour cette classe. Le constructeur de cette classe sera non-**`inline`**.

Question 2

Voici une définition partielle de la classe `Designations`. Les désignations sont stockées dans un tableau alloué dynamiquement d'objets `string` dont l'adresse est stockée dans l'attribut `items`. L'attribut `nb` désigne le nombre de désignations stockées alors que l'attribut `nbMax` représente la taille réelle du tableau `items`. La méthode `agrandirTableau()` est chargée des éventuels besoins en agrandissement du tableau.

```
class Designations {
 string* items;
 unsigned int nb;
 unsigned int nbMax;
 void agrandirTableau();
public:
 Designations():items(nullptr),nb(0),nbMax(0){}
 void ajouterDesignation(const string& n){
  if (nb==nbMax) agrandirTableau();
  items[nb++]=n;
 }
};
```

1. La classe `Designations` nécessite-t-elle que l'on redéfinisse son destructeur ? Expliquer. Si oui, définir ce destructeur.
2. Que proposez-vous afin de parcourir séquentiellement les différentes désignations d'un objet `Constellation` ou d'un objet `Etoile` sans exposer la structure de données de la classe `Designations`. Fournir le code complet implémentant cette solution. Dans cette question on n'implémentera que la solution qui ne permet pas la modification des désignations parcourues.
3. Quelle est la particularité de la méthode `getManager` de la classe `StellarManager` ? Expliquer.
4. Définir la classe `StellarManager` et l'ensemble des méthodes décrites dans le sujet en respectant toutes les propriétés décrites pour cette classe. On supposera que les objets `Etoile` sont gérés en utilisant un tableau de pointeurs d'objet `Etoile` nommé `etoiles` dont la taille réelle sera `nbMaxEtoiles` et dont le nombre de cellules réellement utilisées sera `nbEtoiles` . De même, on supposera que les objets `Constellation` sont gérés en utilisant un tableau de pointeurs d'objet `Constellation` nommé `constellations` dont la taille réelle sera `nbMaxConstels` et dont le nombre de cellules réellement utilisées sera `nbConstels`. Définir tous les éléments nécessaires au bon fonctionement de cette classe même s'ils n'ont pas été présentés explicitement dans le sujet. Dans cette question, on ne definiera pas les méthodes `newConstellation()` et `getConstellation()` (redondantes avec `newEtoile` et `getEtoile()`).

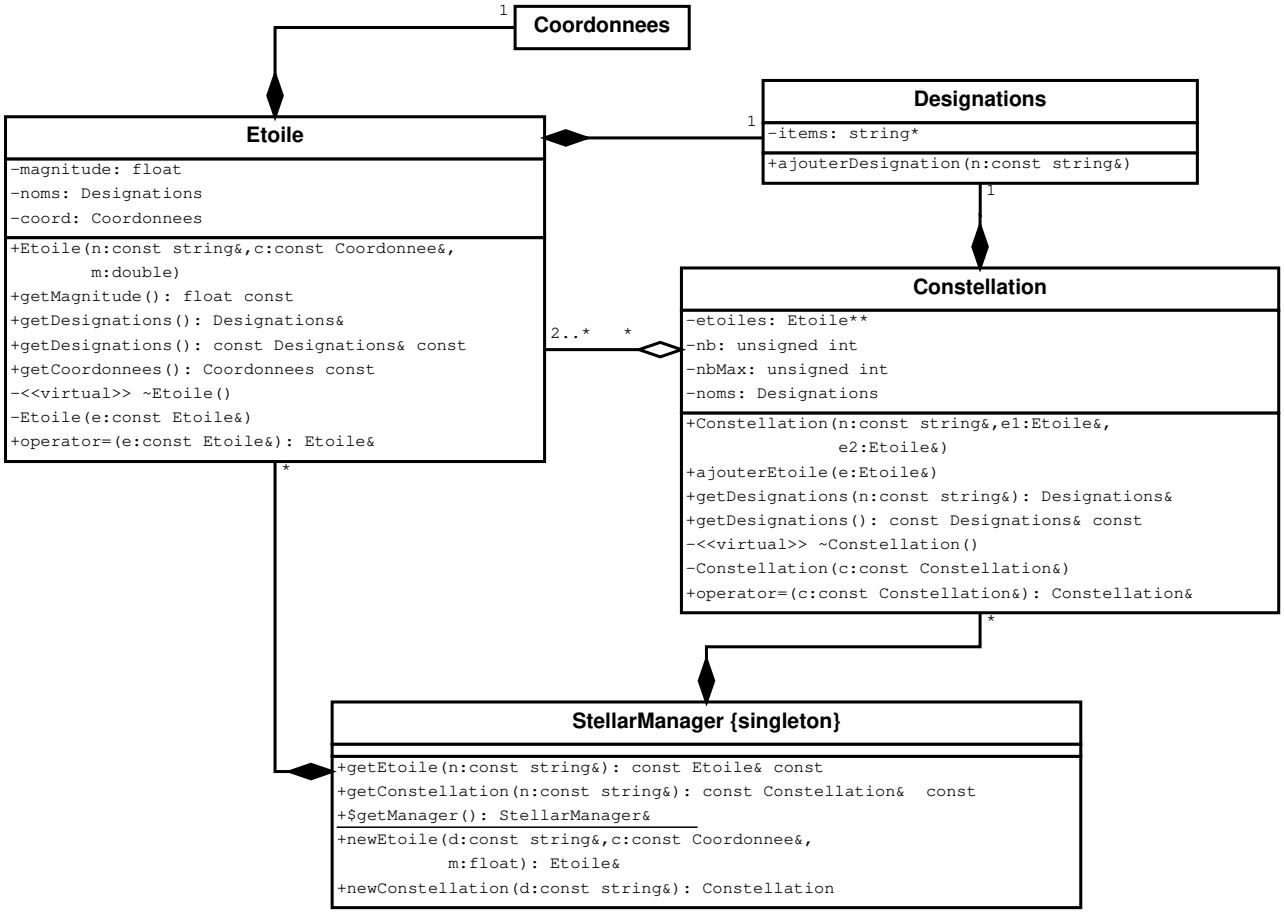
Question 3

1. La classe `Etoile` et `Constellation` ont des parties communes.
   a) Quelles transformations proposez-vous d'apporter à l'architecture présentée dans le sujet afin de factoriser les parties de codes qui sont communes aux deux classes ? Expliquez. Quelle(s) nouvelle(s) classe(s) devez vous éventuellement définir ? Donnez la définition de cette ou ces nouvelles classes si ce besoin existe.
   b) Représentez sous forme d'un diagramme de classes l'architecture obtenue en ne faisant apparaître que les détails pertinents pour comprendre cette transformation.
   c) Indiquez en quelques lignes les modifications à apporter aux classes `Etoile` et `StellarManager`.
2. En vous basant sur cette nouvelle architecture, définir la classe `Constellation` et l'ensemble des méthodes décrites dans le sujet en respectant toutes les propriétés décrites pour cette classe. Définir tous les éléments nécessaires au bon fonctionement de cette classe même s'ils n'ont pas été présentés explicitement dans le sujet.

Solution de l'exercice 74

Question 1

1. — Il y a une composition de la classe `Etoile` à la classe `Coordonnees` et à la classe `Designations` car `coord` et `noms` sont des attributs de la classe `Etoile` et qu'il y a toujours une association de composition entre une classe et ses attributs par définition.
   — Il ya une agrégation de la classe `Constellation` vers la classe `Etoile` car un objet `Constellation` est composé d'objets `Etoile` sans pour autant être responsable du cycle de vie des étoiles agrégées. De plus rien empêche qu'un objet `Etoile` soit impliquées dans plusieurs constellations (en particulier, si elles ont été créées par des civilisations différentes).
   — Il y a une composition de la classe `StellarManager` vers les classes `Etoile` et `Constellation` à la classe `StellarManager`. En effet, le singleton `StellarManager` est responsable du cycle de vie des objets `Etoile` et `Constellation`.
2. Voici une diagramme représentant les classes présentées dans le sujet.



   On peut y faire apparaître les méthodes, les attributs, les associations impliquées, les valeurs de multiplicité, les visibilités, la propriété statique d'une méthode.
3. La méthode non-**`const`** donne un accès en lecture et en écriture (pour ajouter par exemple une désignation) à l'attribut `noms`. La méthode **`const`**, pour les objets `Etoile` ou `Constellation` considérés comme constants (par l'intermédiaire d'une variable constante ou d'une référence **`const`**), permet d'accéder seulement en lecture à leurs désignations.
4. La classe `Etoile` ne nécessite a priori pas que l'on redéfinisse son destructeur, celui généré par le compilateur convenant dans la situation actuelle. Cependant, puisque la classe `Etoile` pourrait être spécialisée, il faut définir un destructeur virtuel n'exécutant aucune instruction afin de respecter le principe de substitution. De plus, si on souhaite interdire l'utilisation du

destructeur à l'unique classe `StellarManager`, il faudrait définir ce destructeur dans la partie privée de la classe.

5. Voici une définition de la classe `Etoile` :

```
1  class Etoile {
2   Coordonnees coord;
3   Designations noms;
4   float magnitude;
5   /* constructeur de recopie et affectation privés pour intedire la
       duplication */
6   Etoile(const Etoile& e);
7   Etoile& operator=(const Etoile& e);
8   /* constructeur et destructeur dans la partie privé et StellarManager amie
        */
9   Etoile(const string& n, const Coordonnees& c, float m);
10  friend class StellarManager;
11  virtual ~Etoile(){}
12 public:
13  float getMagnitude() const { return m; }
14  Coordonnees getCoordonnees
15  Designations& getDesignations() { return noms; }
16  const Designations& getDesignations() const { return noms; }
17 };
```

StellarView.h

```
1 /* constructeur non-inline */
2 Etoile::Etoile(const string& n, const Coordonnees& c, float m):coord(c),
     magnitude(m){
3  noms.ajouterDesignation(n); // ajout de la première désignation de l'
       étoile
4 }
```

StellarView.cpp

On sera vigilant :

— à faire une définition non-**`inline`** du constructeur (comme demandé dans le sujet) et l'ajout d'une première désignation dans l'attribut `noms` ;

— à mettre le constructeur de recopie et l'opérateur d'affectation dans la partie **`private`** de la classe afin d'interdire la duplication d'un objet `Etoile` (comme indiqué dans le sujet) ;

— à mettre le constructeur dans la partie privée et d'indiquer que `StellarManager` est une amie de la classe afin d'en faire leur unique utilisateur (comme indiqué dans le sujet).

— à mettre le destructeur dans la partie privée et d'indiquer qu'il est **`virtual`**.

Question 2

Voici une définition partielle de la classe `Designations`. Les désignations sont stockées dans un tableau alloué dynamiquement d'objets `string` dont l'adresse est stockée dans l'attribut `items`. L'attribut `nb` désigne le nombre de désignations stockées alors que l'attribut `nbMax` représente la taille réelle du tableau `items`. La méthode `agrandirTableau()` est chargée des éventuels besoins en agrandissement du tableau.

```
1 /*...*/
2 class Designations {
```

```
3  string* items;
4  unsigned int nb;
5  unsigned int nbMax;
6  void agrandirTableau();
7 public:
8  Designations():items(nullptr),nb(0),nbMax(0){}
9  void ajouterDesignation(const string& n){
10  if (nb==nbMax) agrandirTableau();
11  items[nb++]=n;
12 }
13 };
```

StellarView.h

1. Oui, il faut un destructeur afin de libérer le tableau dynamique :

```
1 /*...*/
2 class Designations {
3 public:
4  /* virtual  de préférence */ ~Designations() { delete[] items; }
5 };
```

StellarView.h

Le destructeur pourra être virtuel ou non. Rien n'indique pour l'instant sa nécessité.

2. Il faut implémenter le design pattern *iterator*. Il y a plusieurs interfaces possibles. En voici une similaire à l'interface de la STL :

```
1 /*...*/
2 class Designations {
3  string* items;
4  unsigned int nb;
5  /*...*/
6 public:
7  class const_iterator {
8   string* items;
9   const_iterator(string* i):items(i){}
10   friend class Designations;
11  public:
12   const_iterator():items(nullptr){}
13   bool operator!=(const_iterator it) const { return items!=it.items||nb!=it
        .nb; }
14   const_iterator& operator++() { ++items; return *this; }
15   const string& operator*() { return *items; }
16  };
17  const_iterator begin() const { return const_iterator(items); }
18  const_iterator end() const { return const_iterator(items); }
19 };
```

StellarView.h

On prendra soin que la méthode qui permet d'accéder à une désignation (ici `operator*`) renvoie un objet `string` par valeur ou une référence `const` sur une `string` afin de ne pouvoir modifier la désignation pointée. En théorie, les méthodes qui permettent d'obtenir des iterateurs

(ici `begin()` et `end()` devrait être **const** afin d'être utilisable avec des objets `Designations` considérés comme constants.

3. La méthode `getManager` de la classe `StellarManager` doit être statique afin de pouvoir être utilisée sans qu'il existe un objet `StellarManager`.
4. Il y a plusieurs possibilités. On sera néanmoins vigilant à fournir un code cohérent en fonction de la possibilité choisie. Voici une possibilité (la plus simple mais peut être pas la milleure...) :

```
1  /*...*/
2  class StellarManager {
3   Etoile** etoiles;
4   unsigned int nbEtoiles;
5   unsigned int nbMaxEtoiles;
6   // tous les constructeurs et destructeurs dans la partie privée
7   StellarManager():etoiles(nullptr), nbEtoiles(0), nbMaxEtoiles(0){}
8   ~StellarManager();
9   StellarManager(const StellarManager& m);
10  StellatManager& operator=(const StellarManager& m);
11 public:
12  static StellarManager& getManager() { static StellarManager m; return m; }
13  void newEtoile(const string& n, const Coordonnee& c, float m);
14  const Etoile& getEtoile(const string& d) const;
15 };
```

StellarView.h

```
1   StellarManager::~StellarManager(){
2    for(unsigned int i=0; i<nbEtoiles; i++) delete etoiles[i];
3    // éliminer les constellations
4    for(unsigned int i=0; i<nbConstels; i++) delete constellations[i];
5    // éliminer les constellations
6    delete[] etoiles;
7    delete[] constellations;
8   }
9
10  void StellarManager::newEtoile(const string& n, const Coordonnee& c, float
        m){
11   if (nbEtoiles==nbMaxEtoiles){
12    Etoiles** newtab=new Etoile** [nbEtoilesMax+5];
13    for(unsigned int i=0; i<nbEtoiles; i++) newtab[i]=etoiles[i];
14    nbMaxEtoiles+=5;
15    Etoiles** old=etoiles;
16    etoiles=newtab;
17    delete[] old;
18   }
19   etoiles[nbEtoiles++]=new Etoile(n,c,m);
20  }
21
22  const Etoile& StellarManager::getEtoile(const string& d)const{
23   for(unsigned int i=0; i<nbEtoiles; i++){
24    for(Designations::const_iterator it=etoiles[i]->getDesignations().begin
          ();
25      it!=etoiles[i]->getDesignations().end(); ++it){
```

```
26      if (*it==d) return etoiles[i];
27     }
28    }
29    throw StellarException("Il n'existe pas d'étoile avec cette désignation")
          ;
30  }
31 };
```

StellarView.cpp

Question 3

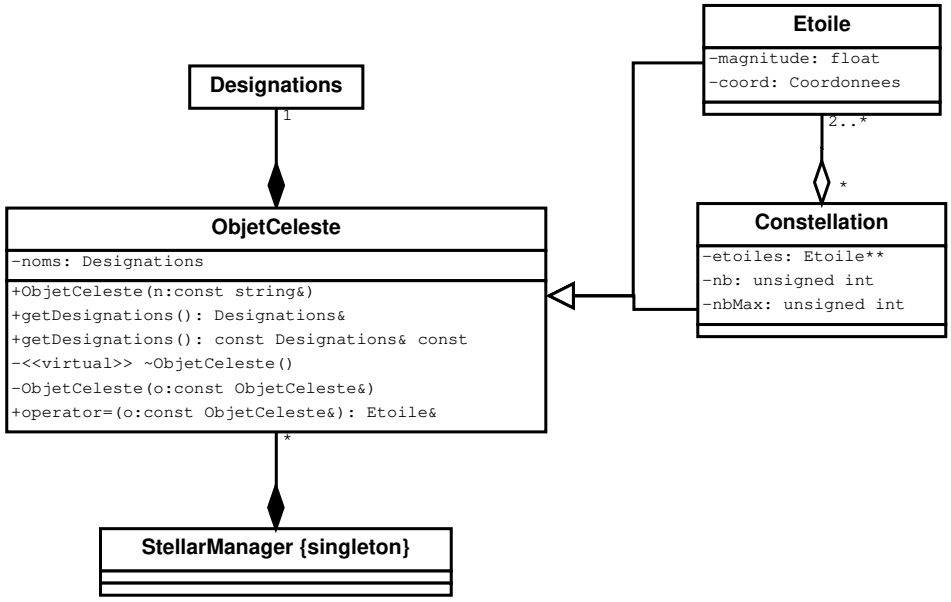
1. a) On peut créer la classe `ObjetCeleste` qui généralise la classe `Etoile` et la classe `Constellation` en regroupant les parties communes au sein de cette nouvelle classe. Les classes `Etoile` et `Constellation` héritent alors de cette classe. Voici la définition possible d'une telle classe :

```
1  /*...*/
2  class ObjetCeleste {
3   Designations noms;
4   /* constructeur de recopie et affectation privés pour intedire la
       duplication */
5   ObjetCeleste(const ObjetCeleste& e);
6   ObjetCeleste& operator=(ObjetCeleste& e);
7   /* constructeur et destructeur dans la partie privé et StellarManager
       amie */
8   ObjetCeleste(const string& n);
9   friend class StellarManager;
10  virtual ~ObjetCeleste(){}
11 public:
12  Designations& getDesignations() { return noms; }
13  const Designations& getDesignations() const { return noms; }
14 };
```

StellarView.h

Notons que le destructeur doit être **`virtual`** afin de respecter le principe de substitution.

b) Voici une diagramme représentant les transformations effectuées.



c) Dans la classe `Etoile`, il faudra enlever l'attribut `noms` et les méthodes associées qui sont maintenant dans la classe `ObjetCeleste`. Dans la classe `StellarManager`, on peut maintenant stocker les adresses des objets `Etoile` et `Constellation` dans un même objet. Cela modifiera en conséquences les méthodes « `Etoile& newEtoile(const string & d, const Coordonnees& c, float m)` » et « `Constellation& newConstellation (const string& d)` ».

2.

```
1  /*...*/
2  class Constellation : public ObjetCeleste {
3   Etoile** etoiles;
4   unsigned int nb;
5   unsigned int nbMax;
6   /* constructeur de recopie et affectation privés pour intedire la
       duplication */
7   Constellation(const Constellation& e);
8   ObjetCeleste& operator=(ObjetCeleste& e);
9   /* constructeur et destructeur dans la partie privé et StellarManager amie
        */
10  friend class StellarManager;
11  Constellation(const string& n);
12  ~Constellation(){ delete[] etoiles;  /* destruction du tableau d'étoiles
       */}
13 public:
14  void ajouterEtoile(Etoile& e);
15 };
```

StellarView.h

```
1  /*...*/
2  Constellation::Constellation(const string& n):
3   ObjetCeleste(n), // appel au constructeur de la classe de base
4   nb(0), // initialement pas d'étoiles
5   nbMax(0), // initialement pas de place,
6        // mais on aurait pu aussi allouer un tableau d'une certaine taille
7   etoiles(nullptr){}
8
9  void Constellation::ajouterEtoile(Etoile& e){
```

```
10  if (nb==nbMax) { // agrandissement du tableau
11   Etoile** newtab=new Etoile**[nbMax+5];
12   for(unsigned int i=0; i<nb; i++) newtab[i]=etoiles[i];
13   nbMax+=5;
14   Etoile** old=etoiles;
15   etoiles=newtab;
16   delete[] old;
17  }
18  etoiles[nb++]=&e;
19 }
```

StellarView.cpp

Exercice 75 - Relations

Dans la suite, il s'agit de définir des classes en C++ qui permettent de manipuler des relations. On se restreindra cependant aux relations binaires entre ensembles finis.

Relation : Soient deux ensembles E et F. Une relation binaire R entre E et F est un sous-ensemble de $E \times F = \{(a,b)/a \in E \wedge b \in F\}$ (où $E \times F$ est l'ensemble de tous les couples (a,b) tels que a appartient à E et b appartient à F). E est appelé l'*ensemble de départ* (ou *source*) de la relation et F l'*ensemble d'arrivée* (ou *destination*) de la relation. Une relation entre ensembles finis peut être définie en extension en précisant quels sont les couples (a,b) qui appartiennent à la relation.

Exemple 1 : $E = \{0,1,2,3,4,5,6,7,8,9\}$ et $F = \{a,e,i,o,u\}$. On définit une relation R entre E et F, en disant que pour $a \in E$ et $b \in F$ alors aRb (lire *a est en relation avec b*) si "la voyelle b est utilisée pour écrire le nom du chiffre a en toutes lettres". Ainsi 4 est en relation avec u, a et e ($4Ru$, $4Ra$ et $4Re$), car 4 s'écrit "quatre" en toutes lettres et contient u, a et e. R est alors définie (en extension) par l'ensemble des couples $\{(0,e);(0,o);(1,u);(2,e);(2,u);(3,o);(3,i);(4,u);(4,a);(4,e);(5,i);(6,i);(7,e);(8,u);(8,i);(9,e);(9,u)\}$.

Exemple 2 : $E = \{Bao, Charles, Cheng, Suzanne, Li, Pierre\}$. On définit la relation @ de E sur E, en disant que pour $a,b \in E$ alors $a@b$ si "a aime b". La relation *amour* de symbole @ est alors définie par l'ensemble des couples $\{(Bao, Pierre);(Charles, Suzanne);(Suzanne, Pierre);(Charles, Li);(Li, Cheng);(Pierre, Bao)\}$. On peut alors dire par exemple que *Charles* aime *Suzanne* ou *Charles@Suzanne* mais que *Suzanne* n'aime pas *Charles* (ou *Suzanne* n'est pas en relation avec *Charles* par rapport à la relation *amour*".

Une relation peut avoir un *nom* (comme "*amour*" dans l'exemple 2) et un *symbole* utilisé pour représenter une relation entre deux éléments (comme R dans l'exemple 1, ou @ dans l'exemple 2).
Une relation entre un ensemble E et le même ensemble E s'appelle une *endorelation* sur E. Différentes propriétés peuvent exister pour une endorelation comme la *réflexivité*, la *symétricité* ou la *transitivité*. Par exemple, on dit qu'une endorelation R sur E est symétrique si $\forall a,b \in E, aRb \Rightarrow bRa$ (si (a,b) appartient à la relation, alors (b,a) doit aussi appartenir à la relation). On peut remarquer que la relation de l'exemple 2 n'est pas symétrique puisque *Charles@Suzanne* mais que *Suzanne* n'aime pas *Charles*...
Une *fonction* f de E sur F (notée $f : E \to F$) est une relation entre E et F dont un élément de E est en relation avec au plus un élément de F. Différentes propriétés peuvent exister pour de telles relations comme l'*injectivité* ou la *surjectivité*.
Dans la suite, il s'agit de définir des classes en C++ qui permettent de manipuler des relations, des endorelations et des fonctions, comme définies précédemment. Voici les spécifications de ces classes :
Un objet de la classe `relation` permet de définir une relation entre 2 ensembles représentés par 2 objets de types `set<A>` et `set<B>` où `A` et `B` sont des types quelconques, et `set` est la classe standard du C++ ainsi nommée (voir quelques rappels à la fin du sujet). La classe `relation` sera donc un patron de classe paramétré avec les types `A` et `B`. La classe `relation` *agrège* (sans les *composer*) ces ensembles en utilisant des attributs de type **`const`** `set<A>*` et **`const`** `set<B>*` (c'est à dire des pointeurs sur les ensembles sur lesquels porte la relation). La classe `relation` compose aussi 2 attributs de type `string` pour y stocker le nom et le symbole de la relation.
Pour construire un objet `relation`, on transmet à l'unique constructeur de la classe, le nom et le symbole de la relation ainsi que les deux ensembles sur lesquels porte la relation. Tous les arguments du constructeur sont transmis en utilisant des références **`const`**. Des accesseurs en lecture `get_nom` et `get_symbole` permettent respectivement d'accéder au nom et au symbole de la relation. Des accesseurs en lecture `get_source` et `get_destination` permettent respectivement d'obtenir des références sur les ensembles de départ et de destination de la relation.
La classe relation *compose* l'ensemble des couples la définissant en utilisant un attribut de type `set <pair<A,B> >`. Une méthode `ajouter` prenant en paramètre un argument `a` de type **`const`** `A&` et un argument `b` de type **`const`** `B&` permet d'ajouter le couple `(a,b)` à la relation. Si `a` n'appartient pas à l'ensemble source de la relation ou si `b` n'appartient pas à l'ensemble destination de la relation,

cette méthode déclenche une exception . Une méthode `est_en_relation`, prenant en paramètre un argument `a` de type **`const`** `A&` et un argument `b` de type **`const`** `B&` permet de savoir si `a` est en relation avec `b` en renvoyant une valeur de type **`bool`**.
Un objet de la classe `endorelation` représente une relation sur un même ensemble. Cette classe patron est donc paramétrée par un seul type `A`. Les méthodes `est_transitive(),est_reflexive(),est_symetrique()` renvoient des valeurs de type **`bool`** qui permettent de connaître les propriétés d'une endorelation représentée par un objet de la classe.
Les objets de la classe `fonction` permettent de représenter des relations fonctionnelles. La méthode `ajouter` de cette classe déclenche une exception si la particularité associée à cette notion n'est pas respectée. Les méthodes `est_injective()` et `est_surjective()` renvoient des valeurs de type **`bool`** qui permettent de connaître les propriétés d'une relation fonctionnelle.

Question 1

Expliquer pourquoi les ensembles de départ et d'arrivée d'une relation sont agrégés (plutôt que composés) par la classe `relation`. Expliquer pourquoi les couples de la relation sont composés (plutôt qu'agrégés) par la classe `relation`.

Question 2

Pour quelles classes doit être (re)définie la méthode `est_en_relation` ? la méthode `ajouter` ? Expliquer.

Question 3

Quel type de visibilité (**`public`**, **`private`** ou **`protected`**?) faut-il envisager pour les attributs de la classe `relation` . Expliquer.

Question 4

Fournir une définition complète de la classe `relation` en déclarant tous ses attributs et toutes ses méthodes. Définir aussi tous les accesseurs (et uniquement les accesseurs) à l'intérieur de la définition de la classe .

Question 5

Définir en dehors du corps de la définition de la classe le constructeur et les méthodes `ajouter` et `est_en_relation`. Pour ces méthodes, faire attention de bien répondre aux spécifications mentionnées.

Question 6

Fournir une définition complète de la classe `endorelation` en déclarant tous ses attributs et toutes ses méthodes. Ne définir que le constructeur de cette classe ainsi que la méthode `est_symetrique` en dehors du corps de la définition de la classe. Redéfinir aussi la méthode `ajouter`, uniquement si nécessaire (c.f. partie 5).

Question 7

Fournir une définition complète de la classe `fonction` en déclarant tous ses attributs et toutes ses méthodes. Ne définir que le constructeur de cette classe ainsi que la méthode `ajouter`, uniquement si nécessaire (c.f. partie 5).

Question 8

En utilisant un adaptateur de la classe `set<pair<A,B> >::const_iterator`, implémenter le design-pattern *iterator* pour la classe `relation` de manière à pouvoir accéder de manière séquentielle à chacun des couples `(a,b)` qui définissent la relation en utilisant le type `relation<A,B>::iterator`. Indiquer le type d'adaptateur utilisé.

Question 9

Surcharger l'opérateur `<<` qui écrit les informations d'un objet `relation`, `endorelation` ou `fonction` sur un flux `ostream`. On utilisera le type `iterator` définie dans la question précédente.

Exemple (ce code vous est fourni à titre d'exemple pour vous aider à mieux comprendre le sujet).

```
set<unsigned int> chiffres;
for(unsigned int i=0; i<=9; i++) chiffres.insert(i);
set<char> voy;
voy.insert('a'); voy.insert('e'); voy.insert('i');
voy.insert('o'); voy.insert('u');
relation<unsigned int,char> R("chiffre-voyelle","R",chiffres,voy);
R.ajouter(0,'e'); R.ajouter(0,'o'); R.ajouter(1,'u');
R.ajouter(2,'e'); R.ajouter(2,'u'); R.ajouter(3,'o');
R.ajouter(3,'i'); R.ajouter(4,'u'); R.ajouter(4,'a');
R.ajouter(4,'e'); R.ajouter(5,'i'); R.ajouter(6,'i');
R.ajouter(7,'e'); R.ajouter(8,'u'); R.ajouter(8,'i');
R.ajouter(9,'e'); R.ajouter(9,'u');
std::cout<<R;
if (R.est_en_relation(8,'u')) cout<<"oui\n"; else cout<<"non\n";
set<individu> s;
s.insert("Bao"); s.insert("Charles"); s.insert("Cheng");
s.insert("Suzanne"); s.insert("Li"); s.insert("Pierre");
s.insert("Xiaokang");
endorelation<individu> A("amour","@",s);
A.ajouter(individu("Bao"),individu("Pierre"));
//...
A.ajouter(individu("Pierre"),individu("Bao"));
std::cout<<A;
```

L'affichage provoqué par l'exécution de la fonction précédente pourrait être le suivant :

```
Relation chiffre-voyelle :
0 R e ; 0 R o ; 1 R u ; 2 R e ; 2 R u ; 3 R i ; 3 R o ; 4 R a ;
4 R e ; 4 R u ; 5 R i ; 6 R i ; 7 R e ; 8 R i ; 8 R u ; 9 R e ;
9 R u ;
oui

Relation amour :
Bao @ Pierre ; Charles @ Suzanne ; Charles @ Li ;
Suzanne @ Pierre ; Li @ Cheng ; Pierre @ Bao ;
```

Solution de l'exercice 75

Question 1

— Les objets représentant les ensembles de départ et de destination ont leur propre cycle de vie. Ils peuvent être utilisés dans d'autres objets `relation` ou tout autre concept utilisé dans la même application. Il s'agit donc d'un cas d'agrégation.

— Les couples de la relation sont créés par l'objet représentant la relation. Quand la relation sera détruite, ces couples n'auront plus de raison d'exister : ils n'existent qu'au travers de la relation. C'est donc un cas de composition.

Question 2

La méthode `est_en_relation` n'a besoin d'être définie que dans la classe `relation`. Par héritage, les classes `endorelation` et `fonction` auront aussi cette méthode qui a exactement le même comportement
En revanche, la méthode `ajouter` doit être définie dans la classe `relation`, mais elle doit être redéfinie pour la classe `fonction` pour satisfaire une spécification supplémentaire ("La méthode `ajouter` de cette classe déclenche une exception si la particularité associée à cette notion n'est pas respectée").

Question 3

Afin de respecter le principe d'encapsulation, les attributs de la classe `relation` doivent être **`private`** ou **`protected`**. Pour offrir un accès facile aux méthodes des classes qui héritent de `relation`, **`protected`** paraît plus pertinent. Cependant, des accesseurs choisis judicieusement permet d'éviter cet écart par rapport au principe d'encapsulation et de n'utiliser que la partie **`private`** de la classe.

Question 4

```
1  #include<iostream> #include<set> #include<string>
2  using namespace std;
3
4  template<class A, class B>
5  class relation {
6  public:
7   relation(const string& n, const string& sym, const set<A>& s, const set<B>& d);
8   const string& get_nom() const { return nom; }
9   const string& get_symbole() const { return symbole; }
10  const set<A>& get_source() const { return *source; }
11  const set<B>& get_destination() const { return *destination; }
12  void ajouter(const A&, const B&);
13  bool est_en_relation(const A&, const B&) const;
14 protected :
15  string symbole;
16  string nom;
17  const set<A>* source;
18  const set<B>* destination;
19  set<pair<A,B> > couples;
20 };
```

relation.h

Question 5

```
1  //...
2
3  template<class A, class B>
4  relation<A,B>::relation(const string& n, const string& sym,
5                          const set<A>& s, const set<B>& d):
6   nom(n),symbole(sym),source(&s),destination(&d){}
7
8  template<class A, class B>
9  void relation<A,B>::ajouter(const A& a, const B& b){
10  if (source->find(a)==source->end()||
11       destination->find(b)==destination->end())
12      throw "erreur";
13  couples.insert(make_pair(a,b));
14 }
15
16 template<class A, class B>
17 bool relation<A,B>::est_en_relation(const A& a, const B& b) const {
18  return couples.find(make_pair(a,b))!=couples.end();
19 }
```

relation.h

Question 6

Il n'est pas nécessaire de redéfinir la méthode `ajouter`

```
1  //...
2
3  template<class A>
4  class endorelation : public relation<A,A> {
5  public:
6   endorelation(const string& n, const string& sym, const set<A>& sd):relation<A,A
       >(n,sym,sd,sd){}
7   bool est_transitive() const;
8   bool est_reflexive() const;
9   bool est_symetrique() const;
10 };
11
12 template<class A>
13 bool endorelation<A>::est_symetrique() const{
14  for(typename set<pair<A,A> >::const_iterator it=couples.begin(); it!=couples.
       end(); ++it) {
15   if (!this->est_en_relation(it->second,it->first)) return false;
16  }
17  return true;
18 }
```

relation.h

Question 7

Il est nécessaire de redéfinir la méthode `ajouter` :

```
1  //...
2
3  template<class A, class B>
4  class fonction : public relation<A,B> {
5  public:
6   fonction(const string& n, const string& sym, const set<A>& s, const set<B>& d):
       relation(n,sym,sd,sd){}
7   void ajouter(const A&, const B&);
8  };
9
10 template<class A, class B>
11 void fonction<A,B>::ajouter(const A& a, const B& b){
12  if (this->source->find(a)==this->source->end()||
13        destination->find(b)==destination->end()) throw "erreur";
14  for(typename set<pair<A,B> >::iterator it=couples.begin(); it!=couples.end();
       ++it) { if (it->first==a) throw "erreur"; }
15  couples.insert(make_pair(a,b));
16 }
```

relation.h

Question 8

On peut utiliser un adaptateur d'objet ou de classe. L'adaptateur de classe utilisant l'héritage permet d'implémenter très facilement un itérateur pour la classe `relation`. À part un constructeur pour la classe `relation::iterator`, il n'y a en effet aucune autre méthode à implémenter, l'interface et le comportement de la classe `set<pair<A,B> >::const_iterator` étant le même que ceux attendus pour la classe `relation::iterator`. En adaptant un peu le design pattern à notre situation, on remarque que dans ce cas on utilise alors l'héritage **`public`** à la place de l'héritage **`private`** ou **`protected`** (qui sont réservés aux cas où l'on ne souhaite pas hériter de l'interface) :

```
1  //...
2  template<class A, class B>
3  class relation {
4  public:
5  //...
6  class iterator : public set<pair<A,B> >::const_iterator {
7   public:
8    iterator(typename set<pair<A,B> >::const_iterator it
9                     =set<pair<A,B> >::const_iterator()):
10      set<pair<A,B> >::const_iterator(it){}
11   };
12   iterator begin() const { return iterator(couples.begin()); }
13   iterator end() const { return iterator(couples.end()); }
14 };
```

relation.h

Dans le cas de l'adaptateur d'objet, il faut réimplémenter toute l'interface :

```
1  //...
2  template<class A, class B>
```

```
3  class relation {
4  public:
5  //...
6  class iterator {
7   public:
8    iterator& operator++(){ ++current; return *this; }
9    const pair<A,B>& operator*() const { return *current; }
10   bool operator==(const iterator& it) const {
11    return current==it.current;
12   }
13   bool operator!=(const iterator& it) const {
14    return current!=it.current;
15   }
16   iterator(typename set<pair<A,B> >::const_iterator it
17                    =set<pair<A,B> >::const_iterator()):current(it){}
18  private:
19   typename set<pair<A,B> >::const_iterator current;
20  };
21  iterator begin() const { return iterator(couples.begin()); }
22  iterator end() const { return iterator(couples.end()); }
23 };
```

relation.h

Question 9

```
1  //...
2  template<class A, class B>
3  class relation {
4  //...
5  };
6
7  template<class A, class B>
8  ostream& operator<<(ostream& f, const relation<A,B>& r){
9   f<<"Relation "<<r.get_nom()<<" :\n";
10  for(relation<A,B>::iterator it=r.begin(); it!=r.end(); ++it) f<<(*it).first<<'
       '<<r.get_symbole()<<' '<<(*it).second<<"\n";
11  //r.print(f);
12  return f;
13 }
```

relation.h

Exercice 76 - Un distributeur de `T`

Dans cet exercice, on dispose d'un fichier d'entête `distributeur.h` et d'un fichier source `distributeur.cpp`. À chaque fois que vous répondrez à une question pour laquelle du code C++ doit être écrit, indiquez dans lequel des 2 fichiers il doit être placé.

`Distributeur` est le nom d'une classe dont les objets proposent un mécanisme pour stocker, récupérer et éliminer des éléments d'un type donné. Une classe `Distributeur` propose 4 méthodes. La méthode `put` permet de stocker un élément passé en argument de la méthode dans un objet `Distributeur`. La méthode `remove` (sans argument) permet d'éliminer un des éléments stockés d'un objet `Distributeur`. La méthode `item` (sans argument) renvoie une référence sur un des éléments stockés dans un objet `Distributeur` sans avoir la possibilité de modifier l'élément référencé. La méthode `size` (sans argument) renvoie le nombre d'éléments stockés dans un objet `Distributeur`.

Un distributeur ne fournit pas de moyen pour choisir spécifiquement l'élément stocké qui sera récupéré avec `item` ou éliminé avec `remove`. C'est pourquoi, ces méthodes n'ont pas de paramètre. Il existe alors plusieurs types de `Distributeur`, chacun se caractérisant par le choix de l'élément qu'il permet de renvoyer ou d'éliminer. Un objet `Pile` est un `Distributeur` qui suit le principe "dernier ajouté, premier éliminé" (Last In-First Out) et qui permet de récupérer (avec `item`) une référence sur la valeur du dernier élément ajouté ou d'éliminer cet élément (avec `remove`). Un objet `File` est un `Distributeur` qui suit le principe "premier ajouté, premier éliminé" (First In-First Out) et qui permet de récupérer une référence sur la valeur du premier élément qui a été stocké ou d'éliminer cet élément. Un `Tas` est un `Distributeur` qui permet de récupérer ou d'éliminer l'élément de plus grande valeur selon un critère donné. Dans cet exercice, on veut concevoir des classes `Distributeur` qui pourront stocker des objets de n'importe quel type `T` donné. Il s'agira donc de classes patron paramétrées par le type `T`.

Lire très attentivement les explications ci-dessus pour répondre aux questions.

1. Quel doit être le type de retour de la méthode `item` ? Expliquer.
2. Expliquez la particularité de la classe et de ses méthodes `put`, `remove`, `item` et `size` ? Quel type de relation lie les classes `Pile`, `File` et `Tas` à la classe `Distributeur` ? Justifiez vos réponses. Dessiner un diagramme UML qui fait apparaître tous les éléments du texte ci-dessus ainsi que leurs caractéristiques et les relations entre ces différents éléments. Ce diagramme devra être le plus complet possible.
3. Dans quel type de fichier faut-il écrire le code des patrons de classe ? Justifiez votre réponse en la mettant en regard avec le concept de *métaprogrammation*.
4. Définir en C++ la classe `Distributeur`.
5. Pour implémenter les classes `Pile`, `File` et `Tas`, il a été décidé que l'on réutiliserait la classe patron `list` de la STL. Pour cela, il existe deux techniques courantes de réutilisation orientée objet. Quelles sont ces 2 techniques ?
6. Utiliser une des deux techniques courantes de réutilisation (en indiquant son nom exact) pour définir la classe `Pile` et ses méthodes. Dessiner un diagramme UML qui fait apparaître les relations (qui seront nommées explicitement) entre les classes `Distributeur`, `Pile` et `list`.
7. Utiliser l'autre technique de réutilisation (en indiquant son nom exact) pour définir la classe `File` et ses méthodes en indiquant le nom exact de cette autre façon de faire. Dessiner un diagramme UML qui fait apparaître les relations (qui seront nommées explicitement) entre les classes `Distributeur`, `File` et `list`.
8. Est-il nécessaire de définir un constructeur par recopie et un opérateur d'affectation pour les classes `Pile` et `File` ? Expliquer.
9. En utilisant le type `const_iterator` de la classe `list`, écrire une fonction patron `max` qui renvoie un objet `const_iterator` désignant l'élément `T` maximum d'un objet conteneur `list <T>` dont une référence **`const`** est transmise en argument. Pour cela, on supposera que les éléments de type `T` peuvent être comparés avec l'opérateur **`operator`**<.
10. En utilisant le type `iterator` de la classe `list`, écrire une fonction patron `max` qui renvoie un objet `iterator` désignant l'élément `T` maximum d'un objet conteneur `list<T>` dont une référence est transmise en argument. Pour cela, on supposera que les éléments de type `T` peuvent

être comparés avec l'opérateur **operator**<. Cette fonction patron peut-elle cohabiter dans le même code source avec la fonction patron précédente. Si oui, comment le compilateur choisit-il la fonction à utiliser ?

11. Définir la classe `Tas` et ses méthodes en expliquant votre façon de faire.
12. Définir une méthode concrète `empty` (sans argument) dans la classe `Distributeur`. Cette méthode renvoie une valeur de type **bool**. La valeur renvoyée est **true** si l'objet `Distributeur` est vide et **false** sinon. Quel est le nom de la célèbre technique d'implémentation orientée objet que vous avez utilisé dans cette question ?

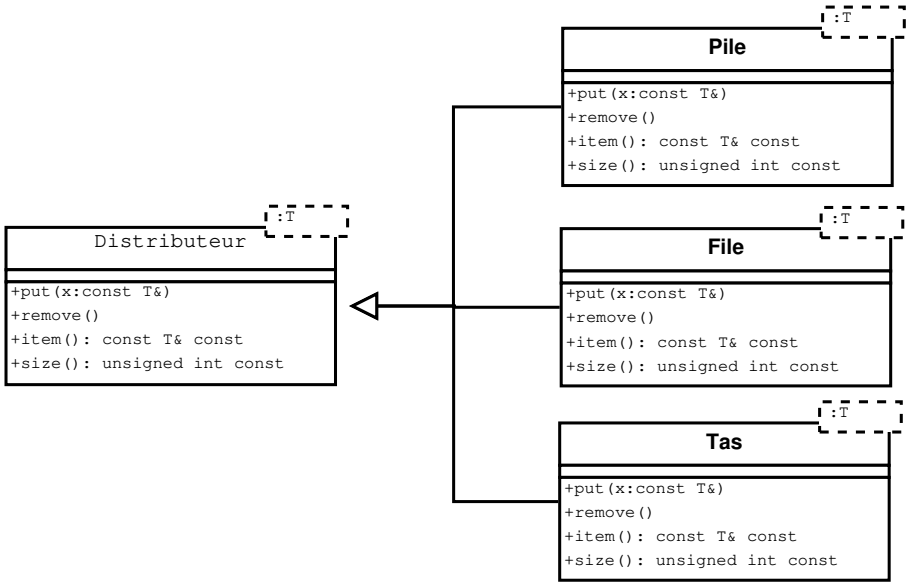
```
1  /**** PROGRAMME EXEMPLE ****/
2  #include "distributeur.h"
3  #include <iostream>
4  void main(){
5    //Distributeur<int> d; // provoque une erreur à la compilation
6    std::cout<<"ESSAI PILE :";
7    Pile p<int>; p.put(4); p.put(9); p.put(2); p.put(7);
8    std::cout<<p.item()<<" ";  p.remove(); std::cout<<p.item()<<"\n";
9    std::cout<<"size="<<p.size()<<"\n";
10   std::cout<<"ESSAI FILE :";
11   File<int> f; f.put(4); f.put(9); f.put(2); f.put(7);
12   std::cout<<f.item()<<" ";  f.remove(); std::cout<<f.item()<<"\n";
13   std::cout<<"ESSAI TAS :";
14   Tas<int> t; t.put(4); t.put(9); t.put(2); t.put(7); t.put(6);
15   std::cout<<t.item()<<" "; t.remove(); std::cout<<t.item()<<"\n";
16 }
17 /* Affichage obtenu :
18 ESSAI PILE :7 2
19 ESSAI FILE :4 9
20 ESSAI TAS :9 7 */
```

Voici une interface partielle de la classe list de la STL :

```
1  template <class T> class list {
2  public:
3   list(); // construit une nouvelle liste vide
4   void push_front(const T& e); // ajoute l'élément e en tête de liste
5   void push_back(const T& e); // ajoute l'élément e en queue de liste
6   const T& front() const; // renvoie la valeur de l'élément en tête de liste
7   const T& back() const; // renvoie la valeur de l'élément en queue de liste
8   void pop_front(); // élimine l'élément en tête de liste
9   void pop_back(); // élimine l'élément en queue de liste
10  unsigned int size() const; //renvoie le nombre d'éléments stockés
11  iterator erase(iterator it); // élimine l'élément désigné par l'iterator
        it
12  //...
13 };
```

Solution de l'exercice 76

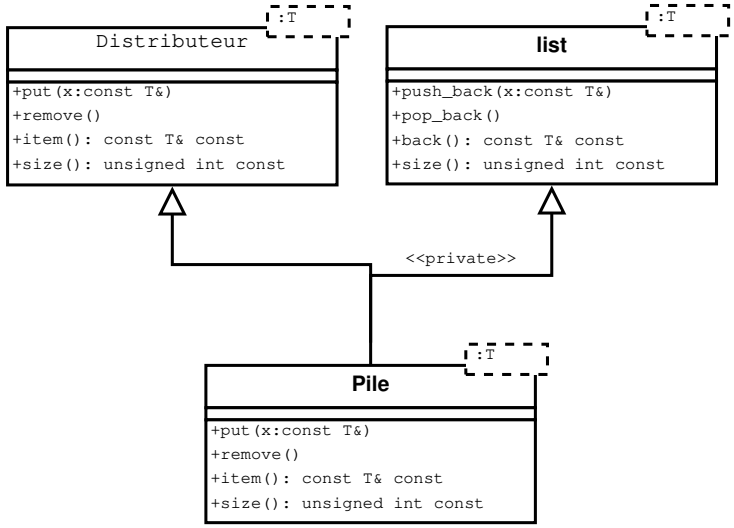
1. Le type de retour de la méthode `item` doit être **`const`** `T&` afin de ne pas offrir la possibilité à l'utilisateur de cette méthode d'utiliser la référence renvoyée pour modifier l'objet `T` référencé.
2. La classe `Distributeur` est abstraite et les méthodes `put`, `remove`, `item` et `size` sont virtuelles pures. En effet, aucune structure de données pour stocker les éléments `T` n'a été précisée. Cette classe représente un modèle d'interface pour les classes concrètes qui vont l'implémenter. Comme indiqué dans le sujet, c'est une relation "*est-un*" qui lie les classes `Pile`, `File`, `Tas` à `Distributeur`. En terme d'implémentation, les classes `Pile`, `File`, `Tas` hériteront donc de façon **`public`** de la classe `Distributeur`.



3. L'ensemble du code de ces patrons de classe doit être placé dans un fichier d'entête (`distributeur.h`) car le compilateur a besoin des patrons dans chaque unité de compilation où ils sont utilisés afin de pouvoir les instancier avec les types utilisés pour fabriquer le programme qui va être compilé. Lors de l'instanciation, on utilise un patron qui peut être vu comme un programme permettant de fabriquer un autre programme (la classe instanciée). C'est pourquoi on parle de méta programmation.
4.
```
1 template<class T>
2 class Distributeur{
3 public:
4  virtual void put(const T& x)=0;
5  virtual void remove()=0;
6  virtual const T& x item() const=0;
7  virtual unsigned int size() const=0;
8 };
```

distributeur.h

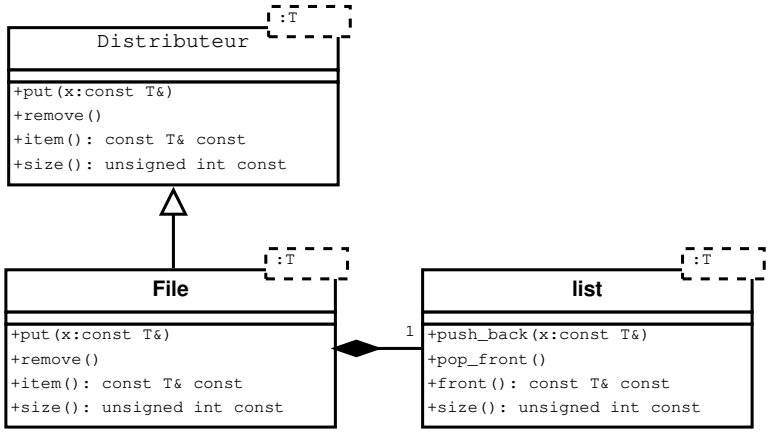
5. Il s'agit du design pattern "*Adapter*". En effet, il s'agit ici de réutiliser le comportement de la classe `list` mais pas son interface. La relation qui lie les classes `Pile`, `File`, `Tas` à la classe `list` est alors une relation "*est-implémentée-en-terme-de*". Il y a deux implémentations possibles : l'*adaptateur de classe* et l'*adaptateur d'objet*.
6. Dans cette question, on utilise le design pattern "*Adaptateur de Classe*". La classe `Pile` hérite de façon **`public`** de la classe `Distributeur` : elle va hériter de son interface pour l'implémenter. La classe `Pile` hérite de façon **`private`** de la classe `list` : elle va hériter du comportement de `list` (pour l'implémentation de `Distributeur`) mais n'exposera pas l'interface de `list` à ses utilisateurs.




```
1  #include<list>
2  using namespace std;
3  //...
4  template<class T>
5  class Pile : public Distributeur<T>, private list<T>{
6  public:
7   void put(const T& x);
8   void remove();
9   const T& x item() const;
10  unsigned int size() const;
11 };
12 template<class T> void Pile<T>::put(const T& x){ list<T>::push_back(x); }
13 template<class T> void Pile<T>::remove(){ list<T>::pop_back(); }
14 template<class T> const T& Pile<T>::item() const { return list<T>::back();
       }
15 template<class T> unsigned int Pile<T>::size() const { return list<T>::size
       (); }
```

distributeur.h

7. Dans cette question, on utilise le design pattern "*Adaptateur d'Objet*". La classe `File` hérite de façon **`public`** de la classe `Distributeur` : elle va hériter de son interface pour l'implémenter. La classe `File` compose un objet de la classe `list` pour l'implémentation des méthode de l'interface `Distributeur`).




```
1  #include<list>
```

```
2 using namespace std;//...
3 template<class T>
4 class File : public Distributeur<T>{
5  list<T> l;
6 public:
7  void put(const T& x);
8  void remove();
9  const T& item() const;
10  unsigned int size() const;
11 };
12 template<class T> void File<T>::put(const T& x){
13  l.push_back(x);
14 }
15 template<class T> void File<T>::remove(){ l.pop_front(); }
16 template<class T> const T& File<T>::item() const {
17  return l.front();
18 }
19 template<class T> unsigned int File<T>::size() const {
20  return l.size();
21 }
```

distributeur.h

8. La gestion de la mémoire de la classe `list` répond aux besoins des classes implémentées. Il n'est donc pas nécessaire de surcharger le constructeur de recopie et l'opérateur d'affectation des classes `Pile` et `File`.
9. La liste dont on cherche le maximum doit être transmise par référence (référence **`const`** ici) afin que l'objet `const_iterator`, désigne bien un objet de cette liste.

```
1 //...
2 template<class T>
3 list<T>::const_iterator max(const list<T>& l) {
4  list<T>::const_iterator m=l.begin();
5  for(list<T>::const_iterator it=l.begin(); it!=l.end(); ++it){
6   if (*it<*m) m=it;
7  }
8  return m;
9 }
```

distributeur.h

10. La fonction patron suivante peut cohabiter avec la précédente.

```
1 //...
2 template<class T>
3 list<T>::iterator max(list<T>& l) {
4  list<T>::iterator m=l.begin();
5  for(list<T>::iterator it=l.begin(); it!=l.end(); ++it){
6   if (*it<*m) m=it;
7  }
8  return m;
9 }
```

distributeur.h

Si on fait un appel de la fonction `max` avec un objet `list`, cette deuxième version sera utilisée avec les listes non constantes ou les listes référencées par des références non-**`const`**. C'est la première version qui sera utilisée avec les listes constantes ou les listes référencées par des références **`const`**.

11. En utilisant un adaptateur d'objet, l'implémentation de la classe `Tas` peut être écrite de la manière suivante :

```
1  #include<list>
2  using namespace std;
3  //...
4  template<class T>
5  class Tas : public Distributeur<T>{
6   list<T> l;
7  public:
8   void put(const T& x);
9   void remove();
10  const T& x item() const;
11  unsigned int size() const;
12 };
13 template<class T> void Tas<T>::put(const T& x){
14  l.push_back(x);
15 }
16 template<class T> void Tas<T>::remove(){ l.erase(max(l)); }
17 template<class T> const T& Tas<T>::item() const {
18  return *max(l);
19 }
20 template<class T> unsigned int Tas<T>::size() const {
21  return l.size();
22 }
```

distributeur.h

Notons que l'on ne cherche pas dans cet exercice à obtenir les complexités théoriques optimales pour cette structure de données.

12. La méthode `empty` est implémentée au niveau de la classe `Distributeur` en utilisant le design pattern *template method* : on utilise la méthode `size` virtuelle pure dont le comportement est précisé au niveau des classes filles concrètes.

```
1  #include<list>
2  using namespace std;
3  template<class T> class Distributeur{
4  public:
5   //...
6   virtual unsigned int size() const=0;
7   bool empty() const { return size()==0; }
8  };
```

distributeur.h

Exercice 77 - Se perdre dans les raccourcis...

Dans le cadre du développement d'un système d'exploitation, on représente des *raccourcis* par des objets de la classe `ShortCut`. Un objet `ShortCut` est caractérisé par :

- un attribut `title` (de type `string`) permettant de donner des informations sur l'utilité du raccourci à un utilisateur,
- un attribut `picture` (de type `string`) correspondant au nom de fichier d'une image utilisée pour l'icône représentant le raccourci (affichée en même temps que les informations stockées dans l'attribut `title`),
- un attribut `filename` (de type `string`) qui est un nom de fichier associé au raccourci,
- un attribut `paction` (de type `Action*`), un pointeur sur un objet de la classe `Action`.

L'objet `Action` pointé par `paction` caractérise l'action à réaliser avec le fichier `filename` quand le raccourci est utilisé. La classe `Action` dispose d'une méthode publique de prototype **`void`** `execute(string f)` qui réalise l'action. Lorsqu'un utilisateur clique sur l'icône représentant un objet `ShortCut`, la méthode `onClick()` de la classe `ShortCut` est exécutée. Cette méthode appelle la méthode `execute()` sur l'objet pointé par son attribut `paction` en lui transmettant la valeur de son attribut `filename` en argument du paramètre `f` de la méthode. Chaque objet `ShortCut` est responsable du cycle de vie de l'objet `Action` sur lequel son attribut `paction` pointe. Un objet `Action` n'est pas partagé et ne peut appartenir qu'à un seul objet `ShortCut`.

Il existe plusieurs sortes d'action, chacune étant représentée par une classe spécifique d'action dans laquelle est précisé le comportement de la méthode `execute` ainsi que les spécificités (attributs et méthodes) de ce type d'action. Par exemple, on distingue un objet de la classe `ActionLaunch` qui permet de représenter une action qui consiste à exécuter le programme représenté par le nom `filename` de l'objet `ShortCut` à laquelle l'action est associée. L'attribut `filename` du raccourci correspondra, dans ce cas, à l'exécutable d'une application. De même, un objet de la classe `ActionOpen` permet de représenter une action qui consiste à *ouvrir* le fichier `filename` avec une application (par exemple un fichier d'extension *.pdf* avec l'application *AcrobatReader*). La classe dispose alors d'un attribut supplémentaire application (de type `string`) représentant le nom de fichier exécutable de l'application qui doit ouvrir le fichier `filename`.

1. Quel type d'association existe-t-il entre la classe `ShortCut` et la classe `Action` ? Justifiez.
2. Définir la classe `ShortCut` (classe, méthodes et instructions) évoquées dans le texte). Ajouter la définition (prototypes et instructions) des accesseurs en lecture pour chaque attribut. Ajouter aussi la définition d'un constructeur ayant 3 paramètres de type **`const`** `string&` permettant d'initialiser les attributs `filename`, `picture` et `title`. L'attribut `paction` sera initialisé avec la valeur `nullptr`. Interdire la duplication par recopie ou affectation d'un objet `ShortCut`.
3. Pour la classe `ShortCut`, définir la méthode `setAction` qui prend en argument un pointeur sur un objet `Action` et qui permet d'indiquer quelle est l'action d'un objet `ShortCut`. Si un objet `ShortCut` possède déjà un objet `Action`, alors la méthode devra libérer l'ancienne action.
4. Quel type de relation existe-t-il entre la classe `Action` et les classes `ActionOpen` et `ActionLaunch` ? Justifiez votre réponse.
5. Précisez la particularité de la classe `Action` en justifiant votre réponse. Quelle est alors la nature de la méthode `execute()` ? D'un point de vue génie logiciel, quel est l'intérêt de définir une telle classe et une telle méthode ?
6. Définir la classe `Action` ainsi que ses méthodes.
7. Définir la classe `ActionLaunch` ainsi que ses méthodes. Les instructions du corps de la méthode execute seront remplacées par le commentaire `/*mettre ici les instructions */`.

   On décide finalement de définir concrètement le comportement général de la méthode `execute()` dans la classe `Action`. Cette méthode appelle maintenant successivement les méthodes de prototype **`void`** `onStart(string f)`, **`void`** `onAction(string f)` et **`void`** `onEnd (string f)`. Le comportement de ces méthodes ne sera spécifié que dans les classes dérivées de la classe `Action`.

8. Quelle est la particularité des méthodes `onStart`, `onAction` et `onEnd` ?
9. Redéfinir (classe, méthodes et instructions) la classe `Action`. Puisque le comportement de `Action::execute()` ne sera plus jamais redéfini, comment modifiez vous la déclaration de cette méthode ?

   On veut maintenant disposer d'une classe de raccourci plus générale, que l'on appelle `ShortCutG` et qui est indépendante d'un type d'action. La classe `ShortCutG` doit pouvoir être paramétrée avec n'importe quel type `A` qui possède une méthode `execute()`. Elle dispose des mêmes méthodes que la classe `ShortCut` et a les mêmes relations avec un objet `A`, que la classe `ShortCut` a avec la classe `Action`.
10. Définir la classe `ShortCutG` (ne reporter que ce qui change par rapport à la classe `ShortCut`).
11. Refaire la classe `ShortCut` (classe, méthodes et instructions) en utilisant la classe `ShortCutG`.
12. Quel(s) design pattern(s) pensez vous avoir appliqué durant le traitement de ce sujet ? Justifiez votre réponse.

13. Dessiner un diagramme de classe représentant l'ensemble des classes évoquées dans le sujet en ajoutant le maximum de détails possible (par rapport aux informations fournies) avec un diagramme de séquence modélisant le scénario suivant : un utilisateur clique sur une icone correspondant à un document `pdf`. Le document est ouvert avec *AcrobatReader*. On supposera pour cela qu'il existe un objet de la classe `System` qui peut recevoir un message *launch* recevant le nom d'un fichier exécutable d'une application à exécuter. On supposera alors qu'il existe un objet de la classe `AcrobatReader` diposant d'une méthode `openFile` recevant en argument un nom de fichier `pdf` à ouvrir.

Solution de l'exercice 77

1. Entre la classe `ShortCut` et la classe `Action`, il s'agit d'une composition : un objet `ShortCut` est responsable du cycle de vie de son objet `ShortCut` associé.
2.

```
1  #include<string>
2  // devra être précédé de la définition de la classe Action
3  class ShortCut {
4   Action* paction;
5   std::string title;
6   std::string picture;
7   std::string filename;
8   ShortCut& operator=(const ShortCut&);
9   ShortCut(const ShortCut&);
10 public:
11  void onClick() { paction->execute(filename); }
12  std::string getTitle() const { return title; }
13  std::string getPicture() const { return picture; }
14  std::string getFilename() const { return filename; }
15  ShortCut(const std::string &t, const std::string& p, const std::string& f)
        :
16      title(t), picture(p), filename(f),paction(nullptr){}
17  ~ShortCut() { delete paction; }
18 };
```

shortcut.h

On fera attention a bien déclarer les attributs dans la partie privée de la classe , à faire des méthodes **const** pour les accesseurs , à définir correctement le constructeur et à définir la méthode `onClick()` qui appelle la méthode `execute()` sur l'objet pointé par l'attribut `paction`. Pour interdire la duplication par recopie ou affectation d'un objet `ShortCut`, on déclare le constructeur de recopie et l'opérateur d'affectation de la classe `ShortCut` dans la partie privée de la classe. Il n'est pas nécessaire de définir ces méthodes.
Il faut un destructeur pour détruire l'objet pointé par `paction` . Voir code ci-dessus pour la définition.

3.

```
1  #include<string>
2  //...
3  class ShortCut {
4  //...
5  public:
6   void setAction(Action *p) {
7    if (paction!=nullptr) delete paction;
8    paction=p;
9   }
10 };
```

shortcut.h

Pour la classe `ShortCut`, on définit la méthode `setAction` qui prend en argument un pointeur sur un objet `Action` et qui permet d'indiquer quelle est l'action d'un objet `ShortCut` . Si un objet `ShortCut` possède déjà un objet `Action`, alors la méthode devra libére l'ancienne action .

4. Les classes `ActionOpen` et `ActionLaunch` héritent de la classe `Action` : ce sont des sortes d'`Action` .

5. La classe `Action` est abstraite : le comportement de la méthode `execute()` n'est précisé que dans les classes dérivées . La méthode `execute()` est alors virtuelle pure . D'un point de vue génie logiciel, l'intérêt est de définir une telle classe est d'avoir un modèle générique de classe utilisable dans la classe `ShortCut` en déportant la question de l'implémentation de ce comportement spécifique dans les classes où ce comportement est connu .

6.
```
1 #include<string>
2 //...
3 class Action {
4 public:
5  virtual void execute(std::string f)=0;
6 };
```

shortcut.h

On fera attention à l'utilisation de **`virtual`** et de `=0` qui permettent de faire de la méthode execute une méthode virtuelle pure et de la classe `Action` une classe abstraite.

7.
```
1 #include<string>
2 //...
3 class ActionLaunch : public Action {
4 public:
5  void execute(std::string f) { /* mettre ici les instructions */ }
6 };
```

shortcut.h

On fera attention à la façon de déclarer l'héritage de `ActionLaunch` par rapport à `Action` ainsi que la définition de la méthode `execute()`.

8. Ces méthodes sont virtuelles pures puisque leur comportement n'est précisé que dans les classes dérivées .

9.
```
1 #include<string>
2 //...
3 class Action {
4 public:
5  void execute(std::string f){ onStart(f); onAction(f); onEnd(f); }
6  virtual void onStart(std::string f)=0;
7  virtual void onEnd(std::string f)=0;
8  virtual void onAction(std::string f)=0;
9 };
```

shortcut.h

Puisque le comportement de `Action::execute()` ne sera plus jamais redéfini, la méthode n'est plus virtuelle .

10.
```
1 template<class A>
2 class ShortCutG {
3 private
4  A* paction;
5 public:
6  void setAction(A* p) {
7    if (paction!=nullptr) delete paction;
8    paction=p;
```

```
 }
//...
};
```

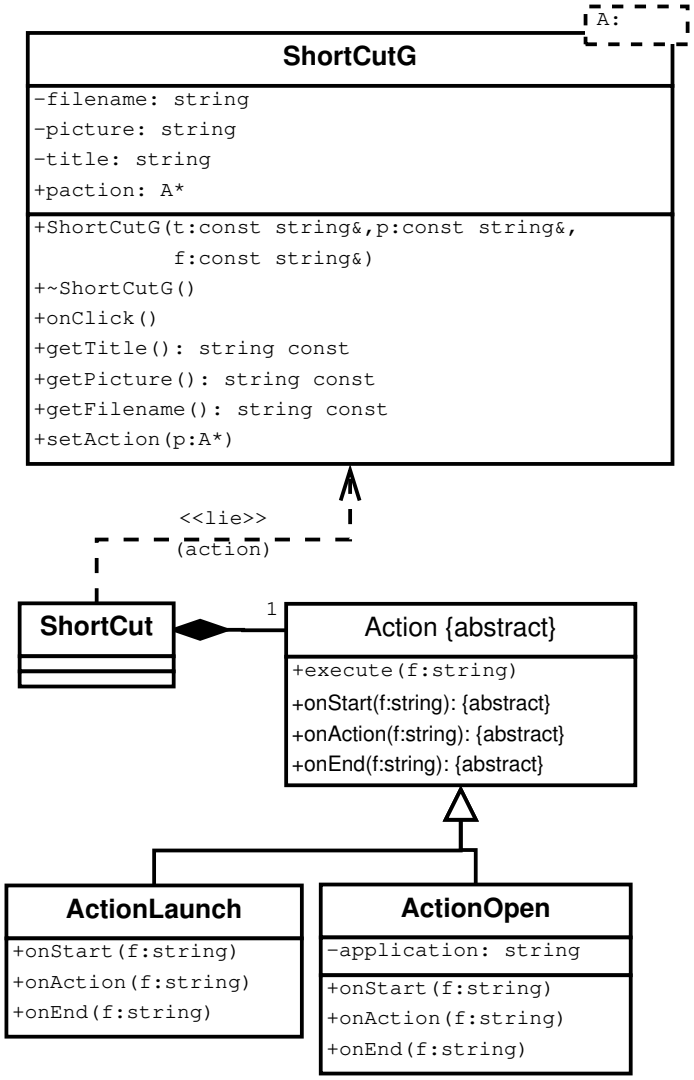
shortcut.h

11.

```
//...
class ShortCut : public ShortCutG<Action>{
};

//ou

typedef ShortCutG<Action> ShortCut;
```

shortcut.h

12. Le design pattern *template méthod* pour l'inversion de contrôle dans la classe `Action` et *Strategy* dans la classe `ShortCutG` ou dans la classe `ShortCut`.
13. Sur le diagramme de classe, on peut faire apparaître :
    — la composition entre la classe `ShortCut` et la clase `Action`
    — les indicateurs de visibilité + ou – au niveau des membres donnés
    — l'héritage entre les classes `ActionLaunch`, `ActionOpen` et la classe `Action`
    — l'introduction de la classe `ShortCutG` avec son paramètre `A` et la liaison entre `ShortCut` et cette classe .



Exercice 78 - Prise de notes

On souhaite développer l'application MyNotes destinée à éditer et gérer un ensemble de *notes* (des mémos) qui peuvent contenir du texte et/ou des images. Une note peut par exemple correspondre au compte-rendu d'une réunion ou à des notes prises lors d'une séance de cours.

Question 1 - Quelques simples notes pour commencer ...

D'un point de vue implémentation, un article est un objet instance de la classe `Note`. Tout objet `Note` est caractérisé par un attribut `titre` de type `string`. La méthode `getTitre()` permet d'accéder à cet attribut. L'unique constructeur de cette classe a un paramètre de type **`const`** `string&` qui permet d'initialiser cet attribut. La méthode `print(ostream& f)` permet d'afficher les informations d'un objet `Note` sur un flux `ostream f`. La définition de cette méthode dépend du type de note. Pour l'instant, on distingue deux types de notes :

- Les objets de la classe `Article` sont des objets `Note` qui représentent des notes contenant uniquement du texte. En plus de l'attribut `titre`, les objets `Article` possèdent un attribut `texte` de type `string`. La méthode `getTexte()` permet d'accéder à cet attribut. L'unique constructeur de cette classe a deux paramètres de type **`const`** `string&` qui permettent d'initialiser le titre et le texte d'un article.
- Les objets de la classe `Image` sont des objets `Note` qui représentent des images ou des photos. En plus de l'attribut `titre`, les objets `Image` possèdent un attribut `description` de type `string` (qui contient des informations sur le contenu de l'image) et un attribut `fichier` de type `string` indiquant le chemin du fichier physique de l'image. Les méthodes `getDescription()` et `getFichier()` permettent d'accéder à ces attributs. L'unique constructeur de cette classe a trois paramètres de type **`const`** `string&` qui permettent d'initialiser le titre, la description et le fichier d'une image.

Dans l'application, les objets `Note` sont gérés par un module appelé `NotesManager` qui est responsable de leur création (et destruction).

Pour la suite, on suppose que l'on dispose d'un fichier d'entête `notes.h` et d'un fichier source `notes.cpp`. À chaque fois que vous répondrez à une question où du code C++ doit être écrit, indiquez dans lequel des 2 fichiers il doit être placé. Les situations exceptionnelles seront gérées en utilisant la classe d'exception `NotesException` que l'on supposera déjà définie au début du fichier `notes.h` (cette classe n'est pas à faire). Voici l'**interface** de cette classe :

```
1 class NotesException {
2 public:
3    NotesException(const string& str);
4    string getInfo() const;
5 };
```

Questions :

1. Expliquez quelle est la particularité de la classe `Note` en sachant qu'il n'est pas possible d'instancier des d'objets de cette classe. Quelle est alors la particularité de la méthode `print` ? Justifiez vos réponses.
2. Quelle relation y a t-il entre la classe `Note` et la classe `Article` ? entre la classe `Note` et la classe `Image` ? entre la classe `Note` et la classe `NotesManager` ?
3. Dessiner un diagramme UML qui fait apparaître tous les éléments du texte ci-dessus ainsi que leurs caractéristiques et les relations entre ces différents éléments. Ce diagramme devra être le plus complet possible.
4. Définir en C++ la classe `Note` avec l'ensemble de ses méthodes évoquées dans le texte ci-dessus ainsi que toutes les méthodes nécessaires pour respecter le principe de substitution.
5. Définir en C++ la classe `Article` avec l'ensemble de ses méthodes évoquées dans le texte ci-dessus. Il devra être possible d'instancier un objet `Article`. Faire en sorte que seul un objet de la classe `NotesManager` puisse créer un objet `Article`.

Question 2 - ... accompagnées de quelques accords

On décide aussi de créer la classe `Document` dont les instances sont des objets `Note` qui regroupent virtuellement plusieurs autres objets `Note` afin de constituer un contenu de taille plus importante. Pour cela, un objet `Document` possède un attribut `notes` de type `stack<Note*>` contenant les adresses des objets `Note` regroupés par le document. Les cycles de vie d'un objet `Document` et des objets `Note` qu'il regroupe sont indépendants (la destruction d'un objet `Document` ne provoque pas la destruction des objets `Note` qu'il regroupe). L'unique constructeur de cette classe a un paramètre de type **`const`** `string&` qui permet d'initialiser le titre du document. Initialement, un objet `Document` ne regroupe aucune `Note`. La méthode `ajouterNote` permet d'ajouter l'adresse de l' objet `Note` référencé par le paramètre de cette méthode. Quand un objet `Document` reçoit le message `print`, son titre est affiché suivi des différentes notes qui forment le document.
Dans les questions suivantes on supposera que la classe `std::stack` dispose d'une classe d'itérateur (bien que cette classe d'itérateur n'existe pas dans la `STL` pour `std::stack`).

Questions :

1. Quelles sont les deux relations qui relient la classe `Document` à la classe `Note` ?
2. Dessiner un diagramme de classe faisant apparaître les classes `Note`, `Article`, `Image` et `Document` ainsi que les relations qui lient l'ensemble de ces classes. Au niveau des détails, vous ne ferez apparaître que ceux qui sont nouveaux par rapport au diagramme précédent.
3. Définir la classe `Document` avec l'ensemble de ses méthodes décrites dans le texte ci-avant. Pour la définition de la méthode `print`.
4. Que faut-il faire afin de pouvoir parcourir l'ensemble des objets `Notes` regroupés dans un objet `Document` sans exposer la structure de données de la classe `Document`. Définir tout ce qui est nécessaire pour implémenter cette fonctionnalité.
5. On suppose maintenant que le patron de classe `stack` de la `STL` n'existe pas. Redéfinir le patron de classe `stack` en réutilisant au mieux le patron `list` de la STL. Précisez quel est le design pattern que vous utilisez dans ce cas. La classe `stack` devra possèder une méthode `push` qui permet d'ajouter un élément, une méthode `pop` qui permet d'éliminer le dernier élément ajouté, une méthode `top` renvoyant une référence vers le dernier élément ajouté, une méthode `empty` (qui renvoie **`true`** si la pile est vide et **`false`** sinon), et une méthode `size` qui renvoie le nombré d'éléments dans la pile.
   Voici aussi une interface partielle de la classe `list` de la STL que vous devez utiliser :

```
1  template <class T> class list {
2  public:
3   list(); // construit une nouvelle liste vide
4   void push_front(const T& e); // ajoute l'élément e en tête de liste
5   void push_back(const T& e); // ajoute l'élément e en queue de liste
6   const T& front() const; // renvoie la valeur de l'élément en tête de liste
7   const T& back() const; // renvoie la valeur de l'élément en queue de liste
8   void pop_front(); // élimine l'élément en tête de liste
9   void pop_back(); // élimine l'élément en queue de liste
10  unsigned int size() const; //renvoie le nombre d'éléments stockés
11  bool empty() const; // renvoie true si la liste est vide et false sinon
12 };
```

Question 3 - ... et un peu de dynamique.

En pratique, l'objet `NotesManager` gère la persistance des objets `Note` en les stockant dans des fichiers ou une base de données.
On suppose qu'un objet `Note` est initialement (juste après sa création) dans l'état "*Enregistré*". Dès qu'une des caractéritiques d'un objet `Note` est modifiée (un évènement que l'on appelera "*Modification*"), l'objet entre dans l'état "*Modifié*". Dans cet état, un enregistrement automatique a lieu tous les

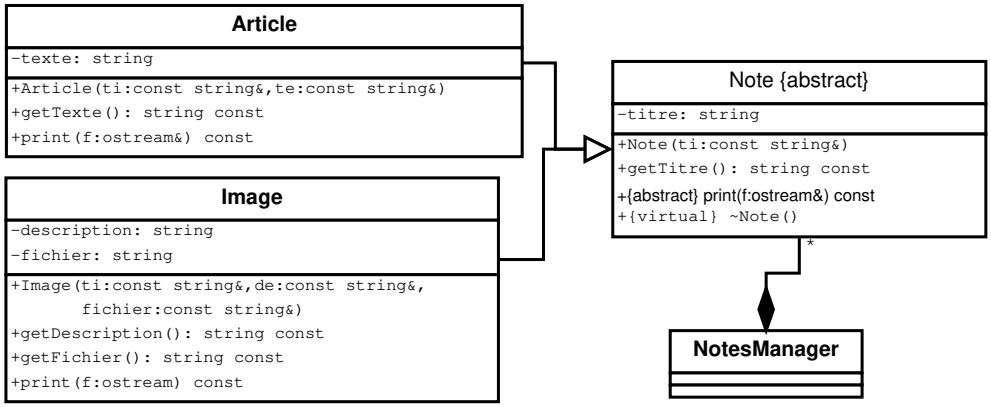
60s dans un fichier temporaire (action que l'on appelera "*saveInTemporaryFile()*"). Si l'utilisateur décide d'enregistrer explicitement la note associée (un évènement que l'on appelera "*Enregistrement*"), l'objet `Note` correspondant revient dans l'état "*Enregistré*". Durant cette transition, la note est définitivement enregistrée (avec une action que lon nommera "*save()*") et le fichier temporaire dans lequel l'objet avait éventuellement été enregistré automatiquement est éliminé (action que l'on nommera "*deleteTemporaryFile()*").
Il se peut que dans des cas critiques (comme un plantage, l'arrêt non prévu de la machine sur laquelle est exécutée l'application, ...), la note qui était en édition n'ait pas été enregistrée par l'utilisateur mais qu'une sauvegarde temporaire automatique ait été effectuée. Aussi, lorsqu'un objet `Note` vient d'être créé par l'application lors de l'ouverture d'une note (il est alors dans l'état "*Enregistré*"), s'il est déctécté qu'il existe un fichier temporaire lui correspondant (ce que l'on pourra vérifier avec la condition nommée "*temporaryFile*" qui est vraie si c'est le cas ou faux sinon), l'objet `Note` rentre dans l'état "*Rechargement*" qui consiste à proposer à l'utilisateur de choisir entre la dernière version enregistrée par l'utilisateur, ou la dernière version enregistrée automatiquement. Une fois que l'utilisateur a choisi entre les deux (évènement qui sera noté *"Sélection"*), l'objet `Note` revient dans l'état *"Enregistré"*.

Question : Modéliser le comportement décrit précédemment en utilisant un diagramme d'états.

Solution de l'exercice 78

Question 1

1. La classe `Note` est une classe abstraite car elle sert de modèle et on ne peut pas instancier cette classe. Comme la méthode `print` n'est pas définie au niveau de cette classe (elle le sera dans les classes dérivées), la méthode `print` est virtuelle pure.
2. Les classes `Article` et `Image` héritent de la classe `Note`. La classe `NotesManager` compose la classe `Note`.
3. Voici un Diagramme UML représentant les classes décrites :



4.
```
1  #include<string>
2  using namespace std;
3  /*...*/
4  class Note {
5   string titre;
6  public:
7   string getTitre() const { return titre; }
8   Note(const string& t):titre(t){}
9   virtual void print(ostream& f) const=0;
10  virtual ~Note{}
11 };
```

notes.h

Notons la syntaxe particulière d'une méthode virtuelle pure avec **`virtual`** et `=0`. Notons que pour respecter le principe de substitution, il est nécessaire que cette classe dispose d'un destructeur virtuel.

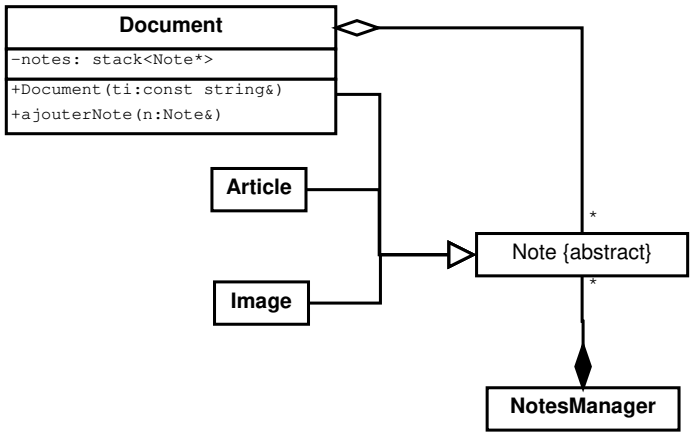
5.
```
1  /*...*/
2  class Article : public Note{
3  private:
4   Article(const string& ti, const string& te):Note(t),texte(te){}
5   friend class NotesManager;
6   string texte;
7  public:
8   string getTexte() const { return titre; }
9   void print(ostream& f) const{
10   f<<"Article : "<<getTitre()<<"\n"<<"texte:\n"<<getTexte()<<"\n";
11  }
12 };
```

notes.h

Notons l'héritage de la classe `Article` depuis la classe `Note`. Puisque la classe ne doit plus être abstraite, la méthode `Print` doit être définie. Afin de faire en sorte que seul un objet de la classe `NotesManager` puisse construire des objets `Notes`, il faut mettre le constructeur dans la partie privée et de déclarer la classe `NotesManager` comme étant amie de la classe `Article`.

Question 2

1. La classe `Document` hérite de `Note` car c'est un type particulier de `Note`. Elle agrège aussi des objets `Note` car un objet `Document` est composé d'objets `Note` sans qu'il y ait dépendance des cycles de vie des objets `Note` agrégés par un objet `Document`.
2. Voici un Diagramme UML représentant les classes décrites dont la classe `Document` :



3.
```
1  #include<stack>
2  /*...*/
3  class Document : public Note{
4   stack<Note*> notes;
5  public:
6   Document(const string& ti):Note(ti){}
7   void print(ostream& f) const{
8    f<<"Document : "<<getTitre()<<"\n";
9    for(stack<Note*>::const_iterator it=notes.begin(); it!=notes.end(); ++it)
10    f<<(*it)->print(f);
11  }
12  void ajouterNote(Note& n) { notes.push(&n); }
13 };
```

notes.h

Attention, la méthode `print` doit également être définie.

4. Afin de pouvoir parcourir l'ensemble des objets `Note` regroupés par un objet `Document` sans exposer la structure de données, il faut implémenter un itérateur. On peut le faire facilement en adaptant rapidement `stack<Note*>::iterator`. On peut aussi tout redéfinir.

```
1  /*...*/
2  class Document : public Note{
3  public:
4   typedef stack<Document>::iterator iterator;
5   iterator begin() { return notes.begin(); }
6   iterator end() { return notes.end(); }
7
8  };
```

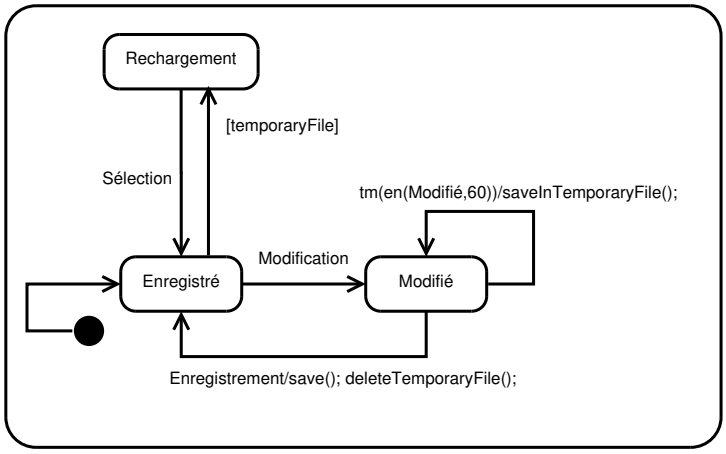
notes.h

5. On peut utiliser un adapter de classe ou d'objet. Ci-dessous, voici une solution utilisant l'adapter d'objet :

```
1  #include<list>
2  using namespace std;
3  template<class T>
4  class stack {
5   list<T> l;
6  public:
7   stack():list<T>(0){}
8   void push(const T& x) { l.push_back(x); }
9   void pop() { l.pop_back(); }
10  T& top() const { return l.back(); }
11  bool empty() const { return l.empty(); }
12  unsigned int size() const { return l.size(); }
13 };
```

stack.h

Question 3



Index