# Vers, strophes et poésies

*Travaux dirigés et pratiques - Programmation par objets 2 (GLIN603)*

**Objectifs** Lors de ces exercices, nous verrons les notions suivantes :
- classe, constructeurs, destructeur, instances, méthodes
- passages de paramètres, tableaux, pointeurs
- définition d'opérateurs

Le sujet dépasse volontairement ce qui peut être fait pendant 1h30 de travaux dirigés et vous permettra de vous exercer au-delà de la séance.

## 1 Les classes `Vers` et `Strophe`

Nous reprenons l'exemple vu en cours et qui porte sur la représentation des vers et des strophes dans une perspective métrique (étude des régularités de rime et de rythme dans la poésie).

Dans cet exemple, un *vers* est essentiellement une suite de mots à laquelle on peut attacher une rime, la sonorité terminale (écrite sous une forme normalisée). Un vers peut être saisi et affiché (voir la Figure 1). Aux deux attributs `suiteMots` et `rime` sont associées deux opérations particulières, habituellement appelées des accesseurs car elles se spécialisent dans l'accès en lecture (ex. `getSuiteMots`) ou en écriture (ex. `setSuiteMots`).

Une *strophe* est une suite de vers. Dans le cas général (qui admet la poésie en vers libres) nous admettrons que l'on peut saisir une strophe par saisie du nombre de vers puis saisie successive des différents vers (dans l'ordre d'apparition dans la strophe). L'affichage d'une strophe est l'affichage de ses vers successifs.

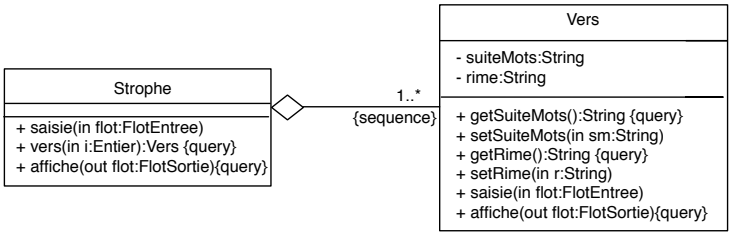


FIGURE 1 – Eléments de la représentation des strophes

```
//------------------------------------ Vers.h ------------------------------------------------

#ifndef vers_h
#define vers_h
class Vers
{
private:
    string suiteMots;     // suiteMots, attribut de type string
    string rime;               // rime, attribut de type string
public:
    Vers();
    Vers(string s);
    Vers(string s, string r);
    virtual ~Vers();
```

```
    virtual string getSuiteMots()const;
    virtual void setSuiteMots(string sm);
    virtual string getRime()const;
    virtual void setRime(string r);
    virtual void saisie(istream& is);
    virtual void affiche(ostream& os);
};
#endif

//------------------------------------ Vers.cc ----------------------------------------
#include<iostream>
#include<string>
using namespace std;
#include"Vers.h"

Vers::Vers(){}
Vers::Vers(string sm){suiteMots=sm;}
Vers::Vers(string sm, string r){suiteMots=sm;rime=r;}
Vers::~Vers(){}

string Vers::getSuiteMots()const  {return suiteMots;}
void Vers::setSuiteMots(string sm)  {suiteMots=sm;}
string Vers::getRime()const  {return rime;}
void Vers::setRime(string r) {rime=r;}

void Vers::saisie(istream& is)  {cout <<"vers puis rime" <<endl;is>>suiteMots>>rime;}

void Vers::affiche(ostream& os)  {os<<"<<"<<suiteMots<<">>";}

//------------------------------------ Strophe.h ----------------------------------------

#ifndef Strophe_h
#define Strophe_h
class Strophe
{
private:
  Vers ** suiteVers;  // suiteVers, attribut de type tableau de pointeurs vers des Vers
                           // implémente l'agrégation "une strophe se compose de vers"
  int nbVers;
public:
  Strophe();
  virtual ~Strophe();
  virtual void saisie(istream& is);
  virtual Vers* vers(int i)const;
  virtual void affiche(ostream& os)const;
};
#endif

//------------------------------------ Strophe.cc ----------------------------------------

#include<iostream>
#include<string>
using namespace std;
#include"Vers.h"
#include"Strophe.h"

Strophe::Strophe(){suiteVers=NULL; nbVers=0;}
Strophe::~Strophe(){if (suiteVers) delete[] suiteVers;}

Vers* Strophe::vers(int i)const
{if (i>=0 && i<nbVers) return suiteVers[i]; else return NULL;}
```

```
void Strophe::saisie(istream& is)
{
  if (suiteVers) delete[] suiteVers;
  cout << "Entrer le nombre de vers : " << endl;
  is>>nbVers; suiteVers = new Vers*[nbVers];
  for (int i=0; i<nbVers; i++)
    {
      Vers *v=new Vers(); v->saisie(is); suiteVers[i]=v;
    }
}
void Strophe::affiche(ostream& os)const
{
  for (int i=0; i<nbVers; i++)
    { suiteVers[i]->affiche(os); os << endl; }
}
```

## 2 Définition d'opérateurs

Pour définir un opérateur (par exemple +), on définit soit une fonction, soit une méthode nommée `operator`*symboledel'operateur* (par exemple `operator+`). Tous les opérateurs sont surchargeables sauf : `. .* :: ?: # ##`. On ne peut changer ni le nombre d'opérandes, ni la priorité, ni le sens d'associativité (droite à gauche, comme `=`, ou gauche à droite comme les opérateurs arithmétiques). Les opérateurs `=,[],()`, `->` doivent obligatoirement être écrits comme des méthodes. Les autres opérateurs peuvent être écrits sous forme de méthode ou de fonction, mais si on choisit de faire une méthode, elle doit obligatoirement appartenir à la classe du premier opérande.

Pour déterminer la signature, on peut se représenter l'appel sous une forme préfixe de l'opérateur. Par exemple, si on a défini dans une classe A la méthode `bool operator== (const A&)`, alors l'expression `x==y` équivaut à `x.operator==(y)`. De même, si on a défini la fonction `operator+(const A& a, const A& b)`, alors l'expression `x+y` équivaut à `operator+(x,y)`.

**Question 1** *Ajouter une définition des opérateurs << et >> aux deux classes* `Vers` *et* `Strophe` *afin d'insérer (resp. d'extraire) les objets de ces classes dans des flots de sortie (resp d'entrée). Ou dit plus familièrement afin d'afficher ou de saisir ces objets.*

**Question 2** *Ajouter à la classe* `Strophe` *une définition de l'opérateur = ayant la même sémantique que le constructeur par copie de manière à rendre la définition de la classe plus cohérente. Cela peut vous inspirer une réécriture du constructeur par copie.*

**Question 3** *Ajouter à la classe* `Strophe` *une définition de l'opérateur [] afin de donner accès au pointeur du vers d'un certain rang. Cette définition doit permettre d'écrire les instructions suivantes :*

```
Strophe S;
S.saisie(cin); // hypothèse : saisie de 8 vers
cout << *S[0] ; // affiche le premier vers
S[1]=new Vers("comme je descendais des fleuves impassibles","ible");
       // modifie le deuxième vers
```

*Réécrivez ensuite les instructions ci-dessus dans le cas où* `S` *est de type* `Strophe*` *(pointeur vers une strophe).*

## 3 Définition de la classe `Poesie`

**Question 4** *Une poésie est une suite ordonnée de strophes que l'on peut classer dans un genre (ballade, sonnet, libre, etc.). Proposez un schéma UML et une implémentation en C++ de cette classe.*

# 4 Manipulation de tableaux

Nous désirons à présent ajouter une méthode vérifiant qu'une strophe correspond à un schéma rimique passé en paramètre. Le schéma rimique est une chaîne de caractères qui exprime les équivalences en rimes dans la strophe et impose le nombre de vers. Par exemple dans un quatrain croisé le schéma rimique est `ABAB`, qui exprime le fait que le premier et le troisième (resp. le deuxième et le quatrième) vers riment ensemble. Nous ne ferons aucune hypothèse sur les schémas rimiques, qui sont donc simplement des suites finies de caractères contenant des répétitions.

La strophe ci-dessous est une strophe de schéma rimique `ABAB`.

*Deux fois je regarde ma montre,*
*Et deux fois à mes yeux distraits*
*L'aiguille au même endroit se montre :*
*Il est une heure... Une heure après.*
*T. Gautier, La montre, « Emaux et Camées »*

Celle qui suit a pour schéma `AABCCCB`.

*Et voici venir, dans les prés,*
*Les fiers coquelicots, parés*
*De leur pourpre cardinalice ;*
*Voici les jacinthes mouvant*
*Leurs cloches roses dans le vent,*
*Et les tulipes, élevant*
*Le saint calice !*
*J. Rameau, Procession des fleurs, « Nature »*

Pour faciliter la vérification, nous introduisons une structure intermédiaire qui transcrit le schéma rimique sous une forme plus facile à utiliser. Cette structure est un tableau d'entiers de même taille que le schéma rimique. Une case $i$ de ce tableau contient $-1$ lorsque la rime du ième vers apparaît pour la première fois dans la strophe, sinon elle contient l'indice $j$ du vers contenant la première occurrence de la rime (voir Figure 2). Il est fortement conseillé d'écrire une méthode ou une fonction auxiliaire qui construit cette structure intermédiaire, puis de se servir de cette structure dans la vérification, plutôt que du schéma rimique. Expliquez le choix de la protection que vous placez sur la méthode de calcul auxiliaire (private ou public).

| numero de vers | schema rimique | structure auxiliaire |
|---|---|---|
| 0 | A | −1 |
| 1 | A | 0 |
| 2 | B | −1 |
| 3 | C | −1 |
| 4 | C | 3 |
| 5 | C | 3 |
| 6 | B | 2 |

FIGURE 2 – Schéma rimique et structure auxiliaire pour la strophe de J. Rameau

**Question 5** *Ecrire la méthode calculant la structure auxiliaire transcrivant le schéma rimique.*

**Question 6** *Ecrire la méthode vérifiant qu'une strophe correspond à un schéma rimique passé en paramètre.*

# Les musées de campagne

Travaux dirigés et pratiques - Objets avancés (HLIN603)

## Objets de musée

Un musée de campagne possède des objets variés. On veut pouvoir stocker pour chacun de ces objets un descriptif (texte libre qui peut être vu comme une chaîne de caractères), et sa référence, nombre entier qui l'identifie de façon unique.
Parmi ces objets, certains sont de véritables objets d'art signés ; pour ceux-là, on veut stocker aussi le nom de l'auteur.
Un musée de campagne possède aussi, hélas, des objets légués par d'honorables citoyens et qu'il n'a pas pu refuser : renard empaillé, portrait du généreux donateur, etc.
Pour ces objets regrettables, on veut pouvoir stocker en plus le nom du donateur et l'année du don.

1. Quelles classes imaginez-vous pour représenter tous ces objets ?
2. Écrivez les fichiers `.h` et `.cc` correspondants, en supposant que chacune de ces classes possède :
   — des attributs privés ;
   — un constructeur sans paramètres qui initialise tous les attributs à des valeurs par défaut ;
   — un constructeur avec paramètres qui remplit tous les attributs avec les valeurs données ;
   — des accesseurs de consultation et de modification pour chaque attribut ;
   — une méthode *saisie* qui saisit les valeurs des attributs sur un flot d'entrée ;
   — une méthode *affiche* qui affiche les valeurs des attributs sur un flot de sortie.
3. Écrivez un *main* qui, pour chacune des classes, crée et affiche deux objets : un directement (déclaration de variable), et l'autre à travers un pointeur.

## Salles d'exposition

Le musée possède des salles d'exposition d'une certaine capacité (capacité = nombre maximum d'objets qu'on peut mettre en exposition dans la salle).

1. Écrivez une classe qui représente une salle, en respectant les caractéristiques suivantes :
   — une salle est créée vide, avec une certaine capacité (donnée, ou 10 par défaut) ;
   — on la remplit en ajoutant un objet donné à une place donnée ; la place est représentée par un numéro ; les objets peuvent appartenir à n'importe laquelle des classes précédentes, mais on veut une seule méthode *ajoute*, la même quel que soit le type de l'objet ;
   — on peut enlever un objet : on donne le numéro de place et on récupère l'objet ;
   — on veut pouvoir afficher le contenu de la salle ;
   — on veut pouvoir connaître le nombre d'objets présents dans la salle.
   — information technique : on est sûr de ne jamais faire de copie d'une salle, on demande donc que la classe ne contienne pas de constructeur par copie.
2. Faites un *main* pour tester le fonctionnement.

# Représentation des suites

*Travaux dirigés et pratiques - Objets avancés (HLIN603)*

Lors de ces travaux dirigés, nous allons retravailler sur l'héritage et faire un exercice préparatoire au cours portant sur la généricité paramétrique (*templates* en dialecte C++).

## 1 La classe paramétrée `vector`

C++ dispose, comme Java depuis la version 1.5, d'une librairie de structures de données paramétrées que nous explorerons plus en détails dans un prochain cours. Pour s'y préparer, nous commençons par utiliser la classe paramétrée `vector`. Le paramètre dont nous parlons ici est le type des objets qui seront placés dans le vecteur.
Nous présentons tout d'abord une partie simplifiée de l'interface de `vector` qui sera suffisante pour les exercices suivants.

```
template<class T>
class vector
{
private:
...
public:
  vector(); //crée un vecteur vide
  ...
  virtual bool empty()const; //retourne vrai si et seulement si le vecteur est vide
  virtual int size()const; //retourne le nombre d'éléments stockés
  virtual void push_back(const T& e); //ajout de l'élément e en fin du vecteur
  virtual void pop_back(); // enlève le dernier élément
  virtual T& back(); // accès par référence au dernier élément
                     // (qui doit exister)
  virtual T& operator[](int i); //accès par référence au ième élément
                                //(qui doit exister)
};
```

Voici également un exemple d'utilisation de cette classe paramétrée.

```
main()
{
  vector<int> v; // déclaration et création d'un vecteur d'entiers
  v.push_back(4); // ajout de 4 à la fin de v
  v.push_back(5);
  v.push_back(6);
  int d=v.back(); // récupération du dernier élément de v
  cout << "element qui va etre enleve = " << d << endl;
  v.pop_back();
  // affichage du contenu de v et appel de l'opérateur []
  for (int i=0; i<v.size(); i++) cout << v[i] <<" "; cout << endl;
}
```

L'exécution du programme affiche les deux lignes suivantes :

```
element enleve = 6
4 5
```

## 2 Les suites

On souhaite représenter des suites réelles, et plus précisément deux cas particuliers : les suites constantes à partir d'un certain rang et les suites arithmétiques.

On rappelle que ...

Une *suite réelle* est une application d'une partie de $\mathcal{N}$ dans $\mathcal{R}$. On considère ici les suites définies sur $\mathcal{N}$. Une suite ne peut pas toujours se décrire de manière finie par une relation de récurrence, une fonction, une raison et un premier terme (comme les suites arithmétiques), etc. On ne peut donc pas prévoir a priori un mode de stockage valable pour toutes les suites.

Une suite est *constante à partir d'un certain rang* s'il existe un entier naturel $n$ et un réel $a$ tels que pour tout entier naturel $n' \geq n$, $u(n') = a$.

Une suite est *arithmétique* si la différence de deux termes quelconques de la suite est une constante $r$ que l'on appelle la raison de la suite. Une suite arithmétique vérifie les deux propriétés suivantes :
- $\forall n, u_n = u_0 + n.r$
- la somme des $n$ premiers termes vaut $n.u_0 + n.(n-1).r/2$

## 3 Spécifications

Soit une suite `u` ; on doit pouvoir lui appliquer les opérations suivantes :
- calcul du n$^{\text{ième}}$ élément de la suite : `u(n)` (surcharge de l'opérateur `()`)
- calcul de la somme des n premiers éléments : `u.s(n)`
- si `u` est une suite arithmétique, accès à la raison : `u.raison()`
- si `u` est une suite constante à partir d'un certain rang : `u.rang()`
  (par exemple pour la suite (1 9 3 1 2 9 9 9 9 9 ...), le rang sera 5).

À partir de ces descriptions d'opérations, proposez des classes permettant de représenter les deux types de suites présentés ; faites vos choix (de classes, d'attributs, de méthodes) de façon à ce que l'ajout d'autres types de suites puisse se faire facilement. Il est fortement conseillé d'utiliser la classe `vector` pour les suites constantes à partir d'un certain rang.
Pensez à la surcharge des opérateurs de lecture et d'écriture, réfléchissez aux constructeurs par copie, opérateur d'affectation, destructeurs.

## 4 Implémentation

Réalisez l'implémentation des classes que vous avez spécifiées.

# Petits exercices sur la généricité paramétrique

*Travaux dirigés et pratiques - Objets avancés (HLIN603)*

## 1 Modèles de fonctions

Proposez des modèles de fonctions `echange` et `triBulles` paramétrés par le type des éléments échangés ou triés, puis écrivez un programme qui les utilise pour trier un tableau de chaînes de caractères.

```
void echange(int& e1, int& e2)
 {int aux=e1; e1=e2; e2=aux;}

void triBulles(int T[], int tailleT)
 {
  int i=tailleT-2,j; bool ech=true;
  while (i>=0 && ech)
    {
      ech=false;
      for (j=0; j<=i; j++)
        if (T[j]>T[j+1])
          {echange(T[j],T[j+1]); ech=true;}
      i--;
    }
 }
```

## 2 Modèles de classes

Soient les classes partielles suivantes représentant respectivement des œufs, des bouteilles, et des casiers de 6 bouteilles ou de 6 œufs :

```
class Bouteille
{ ... };

et son implementation
 ...
////////////////////////////////
class Oeuf
{ ... };

et son implementation
 ...
////////////////////////////////
class CasierBouteille
{
private:
  Bouteille* cases[6];
public:
  CasierBouteille();
  virtual ~CasierBouteille();
  virtual void range(Bouteille* bouteille, int numeroCase);
  ...
};
```

```
CasierBouteille::CasierBouteille()
 {for (int i=0; i<6; i++) cases[i]=NULL;}
CasierBouteille::~CasierBouteille(){}
void CasierBouteille::range(Bouteille* bouteille, int numeroCase)
 {cases[numeroCase]=bouteille;}
  ...
//////////////////////////////////////////////////////

class CasierOeuf
{
private:
  Oeuf* cases[6];
public:
  CasierOeuf();
  virtual ~CasierOeuf();
  virtual void range(Oeuf* oeuf, int numeroCase);
};

CasierOeuf::CasierOeuf()
 {for (int i=0; i<6; i++) cases[i]=NULL;}
CasierOeuf::~CasierOeuf(){}
void CasierOeuf::range(Oeuf* oeuf, int numeroCase)
 {cases[numeroCase]=oeuf;}
```

## Question 1

Proposez, en donnant le fichier `.h` et le fichier `.cc`, un modèle de classe `Casier6` paramétré par un type de produit et qui généralise les casiers de bouteilles et casiers d'œufs.

## Question 2

Ecrivez ce qu'il faut pour :

- instancier le modèle `Casier6` pour obtenir un casier d'œufs,
- créer un œuf (on suppose que la classe dispose d'un constructeur sans paramètres),
- le ranger dans la case 4 du casier.

# Une classe paramétrée `Dictionnaire`

*Travaux dirigés et pratiques - Objets avancés (HLIN603)*

Un dictionnaire est un ensemble d'associations, c'est-à-dire de couples (clé, valeur), où la clé est d'un type donné $TypeCle$ et la valeur d'un type donné $TypeValeur$.
Une association peut être représentée par la classe `Assoc` vue en cours.

## 1 Modélisation

Un dictionnaire possède habituellement les méthodes suivantes :

- `put` : place une clé et une valeur associée dans le dictionnaire ;
- `get` : prend une clé et renvoie la valeur associée ;
- `estVide` : dit si le dictionnaire est vide ;
- `taille` : retourne le nombre d'associations clé-valeur effectivement présentes dans le dictionnaire ;
- `contient` : prend une clé et dit si elle est présente dans le dictionnaire ;
- `affiche` : affiche le contenu du dictionnaire sur un flot de sortie.

## 2 Spécification

Écrivez la partie "signatures des méthodes" du fichier `.h` correspondant au dictionnaire générique. Ajoutez la surcharge de l'opérateur `<<` pour l'affichage et la surcharge de l'opérateur `=` pour l'affectation.

## 3 Implémentation par un tableau dynamique d'associations

Le tableau d'associations est alloué dynamiquement, par défaut avec une taille de 10, et augmente de 5 en 5 en cas de débordement.
L'indice d'une association est calculé grâce à une fonction de hachage appliquée à la clé (voir en annexe). Quand il faut agrandir le tableau, on redistribue les associations existantes dans le nouveau tableau en utilisant la fonction de hachage (voir en annexe aussi).
En utilisant les exemples de résultats de la fonction de hachage, faites quelques essais d'insertions "à la main" pour voir comment les choses se passent :
par exemple `put("abricot",235); put("amande",1023); put("ananas",242); put ("pomme",83);` ...

Complétez la partie "attributs" du fichier `.h`, et écrivez le fichier `.cc` correspondant.

Pour faciliter l'écriture de certaines méthodes, on peut écrire la méthode privée :

```
void CherchCl(const TypeCle& cl, int& i, int& res);
   /* cherche la cle cl dans le dictionnaire :
    si cl est presente: renvoie res=1, i indice de la case de cl dans T;
    si cl est absente et le dictionnaire non plein: renvoie res=0, i indice
      de case possible pour cl dans T;
    si cl est absente et le dictionnaire plein: renvoie res=2, i non
     significatif. */
```

# 4 Utilisation

Instanciez vos classes pour avoir un dictionnaire dont les clés sont des chaînes et les valeurs des entiers.

**Question 1**

Écrivez un programme simple qui teste le fonctionnement de ce dictionnaire (ajoutez des couples, affichez le dictionnaire, rajoutez des couples pour tester l'agrandissement, affichez, etc...).

**Question 2**

Utiliser la classe dictionnaire pour stocker les mots d'un texte lu sur l'entrée standard et comptabiliser le nombre d'occurrences de chacun.

.../...

# 5 Annexe

### Fonction de hachage

Voici un exemple de fonction de hachage simple :

```
#include<string>

int hash(string s, int tailleTab)
{int i=0;
 //calcul d'un entier associe a la string
 for (int j=0; j<s.length(); j++) i=i+(j+1)*s[j];
 //adaptation a la taille du tableau
 return (i % tailleTab);
}
```

Voici quelques résultats produits par cette fonction :

| taille tableau = 10 | taille tableau = 15 |
|---|---|
| hash(abricot, 10) = 8 | hash(abricot, 15) = 13 |
| hash(amande, 10) = 2 | hash(amande, 15) = 7 |
| hash(pomme, 10) = 2 | hash(pomme, 15) = 12 |
| hash(ananas, 10) = 3 | hash(ananas, 15) = 3 |
| hash(prune, 10) = 6 | hash(prune, 15) = 1 |
| hash(griotte, 10) = 3 | hash(griotte, 15) = 13 |
| hash(poire, 10) = 0 | hash(poire, 15) = 5 |
| hash(orange, 10) = 1 | hash(orange, 15) = 1 |
| hash(citron, 10) = 8 | hash(citron, 15) = 3 |
| hash(mangue, 10) = 6 | hash(mangue, 15) = 1 |
| | hash(papaye, 15) = 6 |

Lorsque la fonction *hash* produit une même valeur pour deux clés différentes, on est dans une situation de collision. La méthode de résolution de collisions la plus simple à implémenter est la méthode "recherche séquentielle simple" : on recherche, à partir de la case qui aurait dû être la bonne, la première case libre (si on arrive en fin de tableau, on recommence au début), et on place l'association dedans. Cette méthode n'est pas excellente (elle crée des "grappes"), mais elle suffit pour tester cet exercice.

### Agrandissement du tableau

Une bonne gestion d'un tel tableau consiste à l'agrandir dès qu'il est plein à 75%, de façon à éviter au maximum les collisions.

Dans le cadre de cet exercice, faites le contraire : laissez-le se remplir complètement, afin précisément de pouvoir observer des collisions.

Un essai que vous pouvez faire ensuite si vous avez le temps : agrandir le tableau d'un nombre aléatoire de cases, au lieu de prendre toujours 5 de plus. Ceci a l'avantage de désagréger mieux les grappes produites à l'étape précédente.

# Héritage multiple en C++

*Travaux dirigés et pratiques - Objets avancés (HLIN603)*

## 1 Comptes à dormir debout

1. Proposez quatre classes C++ représentant respectivement :
   — une classe `CompteBancaire` disposant d'un attribut `solde` et d'un destructeur qui affiche la valeur du solde que la banque est supposée rendre au client lors de la fermeture du compte.
   — une classe `CompteRemunere` représentant les comptes auxquels on sert un intérêt. En particulier, lors de la fermeture, le solde est augmenté de 10%.
   — une classe `CompteDepot` représentant les comptes de dépôt classiques. Lors de la fermeture, on prélève sur ces comptes des frais de gestion de 100frs.
   — une classe `CompteDepotRemunere`.
2. Représentez une instance de chaque classe dans le cas de l'héritage non virtuel, puis dans le cas de l'héritage virtuel.
3. Ajoutez une méthode `deposer` dans `CompteBancaire`, spécialisez-la dans le cas des `CompteRemunere` en ajoutant un intérêt de 1% à la somme déposée, et dans le cas des `CompteDepot` en retirant 1frs de frais de gestion et en ajoutant 10frs si le dépôt est supérieur à 1000frs. Effectuez un dépôt sur une instance de `CompteDepotRemunere`, que se passe-t-il dans chacun des cas d'héritage (virtuel ou non) ? Comment pouvez-vous résoudre le problème ?
4. Donnez l'ordre des opérations effectuées lorsqu'un compte est fermé (l'instance est "détruite" au sens de C++) dans chacun des cas d'héritage (virtuel ou non).
5. On suppose que `CompteDepotRemunere` possède une méthode `deposer`, et hérite "d'abord" de `CompteRemunere` puis de `CompteDepot`.
   Ajoutez une classe `CompteDepotAvecCarteCredit` et une classe `CompteDepotRemunereEtAvecCarteCredit` qui hérite "d'abord" de `CompteDepotAvecCarteCredit`. Dans le cas de l'héritage virtuel, que se passe-t-il lorsqu'on ferme un compte de la classe `CompteDepotRemunereEtAvecCarteCredit` ?

## 2 Hiérarchie d'adresses

L'organisation d'une conférence internationale nécessite l'envoi de divers courriers aux participants, donc l'impression d'adresses de formats de types différents. Proposez une hiérarchie de classes qui factorise le plus de choses possibles pour les formats ci-dessous.

— Adresse Française :

| | |
|---|---|
| Civilité Nom Prénom | Mr Toto Tartempion |
| Organisme | CIPN |
| NVoie, TypeVoie NomVoie | 28, rue des flâneurs |
| ComplémentAdresse | .... |
| Localité | Lantourloupe |
| CodeVille Ville | 22453 Larnac |
| France | France |

— Adresse Américaine :

| | |
|---|---|
| Civilité Prénom Nom | Dr David Padua |
| Organisme | Center for SuperExtraPlus |
| NVoie, NomVoie TypeVoie | 181, South street |
| Ville, Etat ZIP | Urbana, Illinois 61801-2932 |
| USA | USA |

— Adresse Japonaise :

| | |
|---|---|
| Civilité Prénom Nom | Prof. Akifume Yamaguishi |
| Organisme | Kyushi University |
| CodeQuart NomQuart, Localité, Ville CodeVille | 6-10-1 Hakozaki, Higashi-ku, Fukuoka-812 |
| Japon | Japon |

— Adresse Soviétique :

| | |
|---|---|
| URSS | URSS |
| Ville CodeVille | Volgograd 400033 |
| NomVoie NVoie | Naberejnaia 13 |
| Organisme | Universitet 1 |
| Nom Prénom | Dichliouk Nicolai |

# Héritage multiple en C++

*Travaux dirigés et pratiques - Objets avancés (HLIN603)*

**En croisière** Construisez la hiérarchie de la Figure 1, d'abord sans faire d'héritage virtuel, puis avec de l'héritage virtuel.

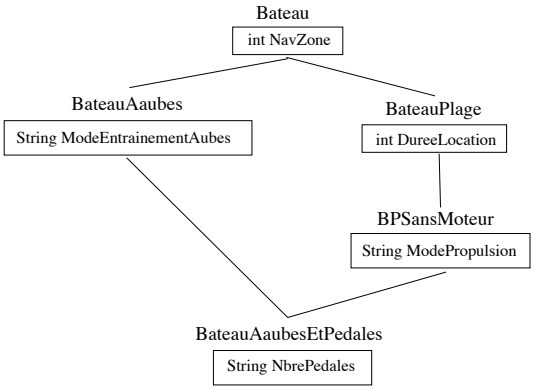


FIGURE 1 – Bateaux

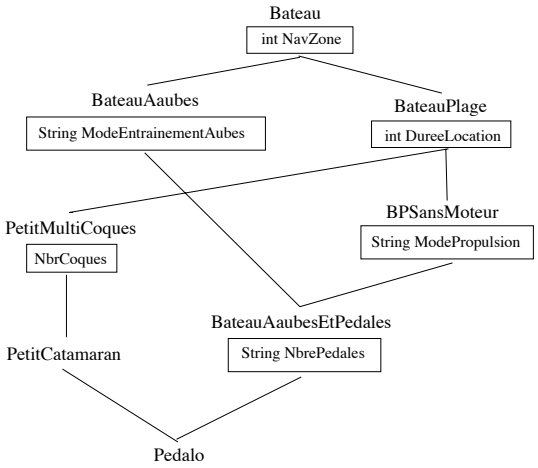


FIGURE 2 – Pédalo

Combien d'attributs possède une instance de *BateauAaubesEtPedales* dans chacun des deux modes d'héritage ? Comment pouvez-vous le vérifier ?

À partir de maintenant et pour toute la suite, on ne fait plus **QUE de l'héritage virtuel**.

**Des effets secondaires de l'ordre d'appel des constructeurs et destructeurs** Construisez la hiérarchie de la Figure 2. Dans les cas d'héritage multiple, déclarez les classes mères dans l'ordre "gauche à droite" du dessin.
Pour simplifier, chaque classe ne contient, en dehors de ses attributs, qu'un constructeur sans paramètres et un destructeur.
Dans le constructeur de la classe *BateauPlage* (resp. *BateauAaubes*), initialisez l'attribut *NavZone* avec la valeur 1 (resp. 100).Cet attribut représente la zone de navigation autorisée, c'est à dire la distance maximum d'éloignement du rivage (en kilomètres).
Créez une instance de la classe *BateauAaubesEtPedales*, une de la classe *PetitCatamaran*, et une de la classe *Pedalo*.
Affichez la valeur de *NavZone* pour ces trois objets. Est-ce que tout vous semble normal ?
Mettez un affichage dans chaque constructeur. Est-ce plus clair ?

**Tester la résolution des conflits** Ajoutez les méthodes prévues dans la Figure 3.

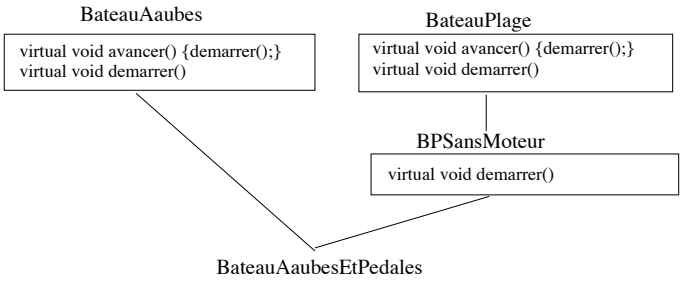


FIGURE 3 – Méthodes en conflit

Quel est l'effet de l'appel `demarrer()` pour une instance de *BateauAaubesEtPedales* ?
Quel est l'effet de l'appel `avancer()` pour une instance de *BateauAaubesEtPedales* ?

# Exceptions en C++

*Travaux dirigés et pratiques - PO2 (GLIN603)*

## 1 Examen des exceptions prédéfinies en C++

La figure 1 vous présente les exceptions standard prévues en C++. La classe mère, `exception`, a l'interface suivante :

```
class exception
{
public:
exception() throw();
exception(const exception&) throw();
exception& operator=(const exception&) throw();
virtual ~exception() throw();
virtual const char* what() const throw(); // affiche le type d'exception
};
```

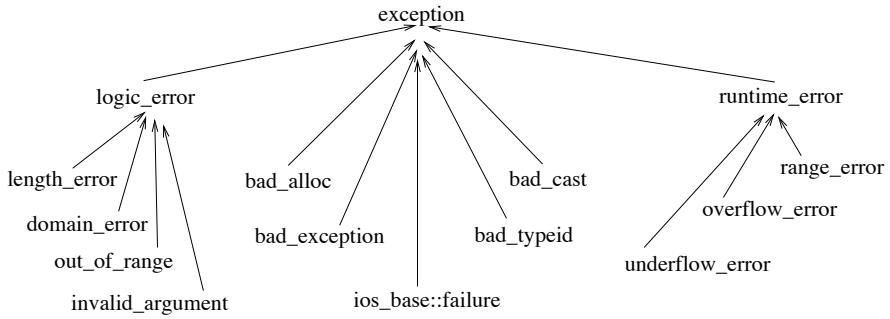


FIGURE 1 – Hiérarchie des exceptions en C++

Certaines exceptions sont signalées par les méthodes de la librairie standard C++. Lorsque vous aurez terminé les exercices suivants en TP, vous pouvez essayer de les faire apparaître.

## 2 File et file bornée

Proposez une classe paramétrée `File`, représentant les files d'attente (premier entré, premier sorti). La file sera implémentée à l'aide d'un `vector` et prévoyez une gestion d'exceptions pour cette classe en étudiant les pré-conditions et post-conditions des opérations. Les exceptions signalées par la file seront également paramétrées par le type des éléments stockés dans la pile ; dans le verdict, affichez le type de l'exception (utilisez la méthode `what`), le type des éléments stockés dans la file et le contexte (nom de la méthode) où s'est produite l'erreur ; placez les exceptions dans la hiérarchie déjà existante. Les opérations à prévoir sont les suivantes :

- — création,
- — consultation du nombre d'éléments stockés,
- — prédicat permettant de savoir si la file est vide,
- — retrait et retour de l'élément le plus ancien,
- — ajout d'un élément,
- — consultation de l'élément le plus ancien.

Proposez une classe paramétrée `FileBornée`, spécialisation de `File` et complétez la gestion des exceptions. La capacité d'une file bornée doit être pour cet exercice un paramètre de généricité, c'est volontaire notamment pour vous amener à étudier les conséquences d'un tel choix.
Ecrivez une opération (méthode ou fonction ?) permettant de transvaser des éléments d'une file bornée d'un certain type T (ex. Enfant) dans une file bornée du même type.

Même question en transvasant dans une file bornée d'un type plus général (ex. Personne).
Etudiez l'opération inverse (transvaser une file de personnes dans une file d'enfants) et le cas général où les types des deux files sont quelconques.
Ecrire un petit programme incluant des manipulations de files et de files bornées et des récupérations d'exceptions.

# 3 Distributeur de friandises

Nous étudions le fonctionnement d'un distributeur de friandises relativement standard.
Le distributeur est décrit par une marque et un numéro de série et se compose de différents compartiments numérotés de taille fixe qui se comportent comme des files d'attente bornées (la taille ne peut changer d'un compartiment à un autre).
Un compartiment stocke des friandises qui doivent être toutes de même type.
Les friandises sont décrites par le code de l'atelier de fabrication, une date de péremption et un type décrit par un nom ("nougat", "nuts", etc.) et un prix qui ne doit pas dépasser la constante `prixMaximum`.
Sur le distributeur, on peut :

- associer (par une sorte d'étiquette) un type de friandise à un compartiment de numéro donné, pour cela le compartiment doit être vide,
- désaffecter un compartiment : aucun type de friandise n'y est associé,
- modifier le prix d'un type de friandise stocké dans un compartiment de numéro donné,
- ajouter une friandise dans un compartiment de numéro donné,
- regarnir de friandises un compartiment de numéro donné,
- examiner (sans la retirer) la friandise qui est sur le devant d'un compartiment de numéro donné,
- retirer une friandise dans un compartiment de numéro donné,
- vider un compartiment de numéro donné.

Ces méthodes seront réalisées par délégation du traitement au compartiment donné.
Proposez un schéma des classes. Prévoyez les exceptions associées à cette modélisation : attachez-vous à signaler, dans les méthodes d'une classe, des erreurs dont la signification est en rapport avec la sémantique de la classe. Par exemple `indiceHorsBornes` a un sens pour un tableau, mais pas pour un distributeur de friandises, pour ce dernier, c'est au contraire une erreur `compartimentInexistant` qui aurait un sens.
Ecrivez le code des méthodes en récupérant les exceptions d'un niveau interne de description (par exemple des exceptions qui ont un sens pour une file bornée) pour signaler des exceptions du niveau adéquat (par exemple des exceptions qui ont un sens pour un compartiment).
Votre solution convient-elle pour traiter des distributeurs dans lesquels la taille pourrait varier d'un compartiment à un autre ?

# Travaux dirigés sur STL (Standard Template Library)

*Travaux dirigés et pratiques - PO2 (UE GLIN603)*

**Compléments sur les conteneurs** L'interface des classes conteneurs contient un ensemble de définitions de types publiques, telles que (pour un conteneur paramétré par le type `T`) :

- `typedef T value_type`, type des éléments,
- `typedef ... iterator`, type des itérateurs sur le conteneur (la partie en pointillé dépend de l'implémentation choisie),
- `typedef ... const_iterator`, type des itérateurs constants sur le conteneur (la partie en pointillé dépend de l'implémentation choisie).

Ces définitions de types permettent d'écrire des fonctions très générales, s'appliquant à tous les conteneurs. Pour les utiliser, il faut les faire précéder du mot-clef `typename`. Comme le dit B. Stroustrup lui-même « Having to add typename before the names of the members types of a template is a nuisance » qu'il justifie par la difficulté des compilateurs à reconnaître ces types. Admettons, le résultat est que dans les compilateurs actuels, cette nuisance s'applique comme vous le montre le code suivant pour une fonction `somme` que nous vous invitons à examiner pour faire la somme des éléments d'un vecteur d'entiers et d'un set de float.

```
template<typename C>
// le parametre doit etre un conteneur
// C::value_type doit etre un type muni de l'addition
// et 0 son élément neutre
typename C::value_type somme(const C&c)
{
typename C::value_type s=0;
typename C::const_iterator p = c.begin();
while (p != c.end()){s+=*p; p++;}
return s;
}
```

**Comptage de mots** Tout comme vous l'avez fait lors du TP sur le dictionnaire associatif, utilisez son équivalent en STL, la classe `map`, pour compter le nombre de mots d'un texte et afficher le résultat du comptage.

**Effacement conditionnel** Nous vous proposons d'examiner l'algorithme d'effacement conditionnel `remove_if`. Sa sémantique est très particulière : la fonction ne supprime pas d'éléments en mémoire, elle les écrase, et retourne la fin valide du conteneur. Remarquez au passage que les tableaux se manipulent comme les conteneurs, et où se trouve l'itérateur (la position) de fin du conteneur ! Essayez les mêmes manipulations avec un vecteur d'entiers. Notez le type de la fonction passée en paramètre à `remove_if`.

```
bool odd (int a)
{ return a % 2;}

int numbers[6] = { 0, 0, 1, 1, 2, 2 };

int main ()
{
  int* ite =
  remove_if (numbers, numbers + 6, odd);
  for (int i = 0; i < 6; i++)
    cout << "adresse element " << i << " = " << (int)(&numbers[i])
         << " valeur = " << numbers[i] << endl;
  cout << endl;
  cout << "adresse retournee par remove_if =" << (int)ite;
  cout << endl;
  return 0;
}
```

**Un effacement conditionnel plus satisfaisant** Ecrivez une fonction `MonRemove_if`, paramétrée par un conteneur et un prédicat (une fonction booléenne) qui supprime effectivement les éléments du conteneur vérifiant le prédicat. Utilisez `remove_if` et `erase` dans un programme de test.

**Classes fonctions** Vous trouverez ci-dessous une classe dont les instances peuvent être considérées comme des fonctions testant la divisibilité.

```
class divisiblePar {
private:
  int diviseur;     // donnée membre : le diviseur
public:
  divisiblePar(int div) : diviseur(div) {}; // constructeur
  bool operator()(const int& dividende) { //
     return (dividende%diviseur)==0 ;
  }
};
```

Analysez-la. Comparez cette classe et ses instances à une fonction classique *divisiblePar* à deux paramètres.

```
void TesteDivisiblePar()
{
 divisiblePar f(2);
 divisiblePar g(3);

 cout << "f(4) = " << f(4) << endl;
 cout << "f(8) = " << f(8) << endl;
 cout << "f(5) = " << f(5) << endl;
}
```

Que vaudrait `g(9)` ?

**Affichage** Ecrivez un modèle de fonction d'affichage, paramétrée par un type `C`, supposé être un type de conteneur, prenant comme paramètre un conteneur (passé par référence), et affichant sur le flot de sortie standard `cout` tous les éléments du conteneur. Vous utiliserez `C::value`, qui représente le type des éléments du conteneur, pour paramétrer l'`ostream_iterator`.

**Crible d'Erathostènes** On souhaite connaître (en les affichant, par exemple) tous les nombres premiers jusqu'à une certaine borne `B`. Utilisez une liste (`list` en STL) pour stocker tous les entiers de l'intervalle [2, B]. Puis supprimez successivement tous les multiples des nombres premiers rencontrés, en vous servant de `MonRemove_if`. Vous pouvez extraire ou non les nombres premiers de la liste quand vous les trouvez.

**Ensemble ordonné** Cet exercice est volontairement plus libre et vous demande un peu de conception préalable. Ecrivez la classe `EnsembleOrdonne` vue en cours. Les paramètres de généricité sont le type des éléments stockés et une fonction de comparaison de ces éléments. On doit pouvoir ajouter des éléments, en retirer, retourner les plus petits ou les plus grands éléments, etc.

**Fonctions partielles** Nous vous proposons d'étudier la modélisation et l'implémentation d'une classe représentant les fonctions partielles définies en extension et dont le domaine et le codomaine sont finis.
Pour donner un exemple, une fonction partielle particulière serait déterminée par :

- un ensemble de départ, $E = \{Sylvie, Astrild, Nestor, Geraldine, Hector\}$
- un ensemble d'arrivée, $F = \{13, 12, 9, 8\}$
- une partie $G$ du produit cartésien $E \times F$ telle que pour tout $x \in E$, il existe au plus un $y \in F$ avec $(x, y) \in G$,
  par ex. $G = \{(Sylvie, 12), (Astrild, 13), (Nestor, 9), (Geraldine, 12)\}$

La fonction est totale si pour tout $x \in E$, il existe un $y \in F$ avec $(x, y) \in G$, ce qui n'est pas le cas dans l'exemple ci-dessus.
Nous supposerons que $E$ et $F$ sont des sous-ensembles des types $TE$ et $TF$.
Proposez une première version d'une classe paramétrée représentant les fonctions partielles comprenant :

- la déclaration de la classe, paramètres, etc.,
- les attributs nécessaires à la représentation mémoire (il est conseillé d'utiliser les classes paramétrées de STL),
- un constructeur,
- quelques méthodes simples pour créer une fonction, savoir si elle est totale, connaître son domaine de définition, etc..

Instanciez cette classe paramétrée pour créer la fonction donnée dans l'exemple ci-dessus.
Pour implémenter la composition de fonctions, il y a au moins deux possibilités théoriquement :

- écrire une méthode de classe (`static`) qui prenne deux fonctions en arguments, par ex. $f : E \to F$ et $g : F \to G$ et qui retourne la composée de $f$ et de $g$, soit $h : E \to G$, avec $h(x) = g(f(x))$
- écrire une méthode d'instance qui prend donc seulement $g : F \to G$ comme argument et retourne la composée de *this* (supposée de $E \to F$) et de $g$ , soit $h : E \to G$, avec $h(x) = g(f(x))$

Etudier les deux implémentations correspondantes.

# Encapsulation en C++

*Travaux dirigés et pratiques - Objets avancés (HLIN603)*

## 1 Compréhension des mécanismes

Donnez les accès pour les programmes suivants.

### 1.1 Accès à `f` protected

```
class C1
{
protected: virtual void f(){}
friend class A;
friend class B;
public:
virtual void mc1();
};

class C2 : public virtual C1
{public:
virtual void mc2();
};

void C1::mc1(){C1 *c1; c1->f(); C2 *c2; c2->f();}
void C2::mc2(){C1 *c1; c1->f(); C2 *c2; c2->f();}

class A
{public:
virtual void ma(){C1 *c1; c1->f(); C2 *c2; c2->f();}
};

class B : public virtual A
{public:
virtual void mb(){C1 *c1; c1->f(); C2 *c2; c2->f();}
};

class D
{public:
virtual void md(){C1 *c1; c1->f(); C2 *c2; c2->f();}
};

int main(){C1 *c1; c1->f(); C2 *c2; c2->f();}
```

### 1.2 Accès à `f` private

Avec le même programme que ci-dessus.

### 1.3 Avec une redéfinition de `f`

La classe `C2` du programme initial est modifiée ainsi.

```
class C2 : public virtual C1
{
protected: virtual void f(){}
public:
virtual void mc2();
friend class D;
};
```

### 1.4 Accès à une partie d'une classe

Chercher les directives d'accès permettant de représenter les accès et les classes de la figure 1.

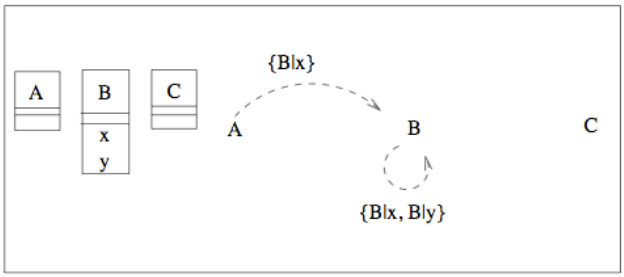


FIGURE 1 – Graphe d'accès en C++

Alternativement, vous pouvez chercher à représenter ces accès en utilisant l'héritage.

## 2 Modélisation

### 2.1 Héritage d'implémentation

Complétez l'implémentation de la classe `Pile` vue en cours, qui hérite de manière privée (héritage d'implémentation) de `vector`.

### 2.2 Bibliothèque

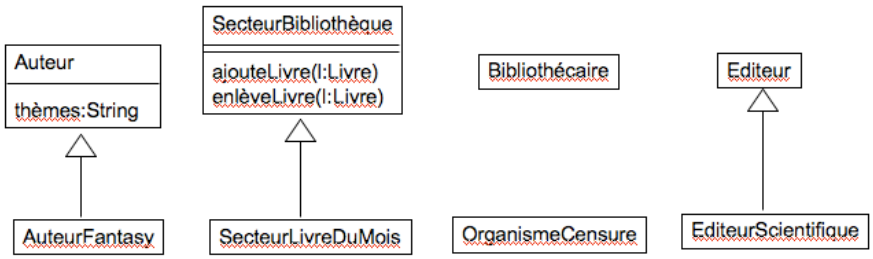


FIGURE 2 – Diagramme de classes pour une bibliothèque

Pour le diagramme de classes de la figure 2, étudiez les accès autorisés par les contraintes suivantes (représentez-les dans un graphe).

- Les bibliothécaires peuvent ajouter et enlever des livres dans tous les secteurs de bibliothèque.
- Seuls les bibliothécaires ont le droit d'ajouter des livres dans les secteurs de type « le livre du mois » car le choix des livres de ces secteurs, correspondant à l'actualité du moment, leur appartient.
- Tous les éditeurs (incluant les éditeurs scientifiques) peuvent ajouter des livres dans les secteurs de bibliothèque (à l'exception des secteurs de type « le livre du mois »).
- Tous les auteurs (incluant les auteurs de fantasy) peuvent ajouter et enlever des livres (généralement leurs propres livres) dans les secteurs de bibliothèque, mais sur les secteurs « le livre du mois » ils ne peuvent qu'enlever des livres.
- Les organismes de censure peuvent seulement enlever des livres, et ceci dans tous les secteurs de bibliothèque, y compris les secteurs « le livre du mois ».

Puis discutez une solution s'en approchant en C++ que vous testerez lors des travaux pratiques. Vous comparerez les accès demandés dans la modélisation et ceux autorisés par le programme que vous avez écrit.