Progression

TD	Titre	Exercices
TD1-6	Eléments de C++	1 à 18
TD7-10	Classes	19, 20, 21
TD11-15	Associations, patrons de conception	22, 23
TD16	STL	24
TD17	Modélisation en UML	25
TD18-20	héritage, spécialisation, redéfinition, polymorphisme, stockage hétérogène, classes abstraites	26, 28, 29, 30
TD21-22	iterator, prototype, transtypage, template method, adapter	31, 32, 34
TD23-25	Programmation générique	35, 36, 37
TD26	STL	39

Exercice 1 - Création d'un projet en C++

sous Visual : Créer un projet en C++ de la manière suivante :

- Aller dans le menu "Fichier / Nouveau / Projet".
- Sélectionner "Projet vide".
- Entrer un nom pour le projet, par exemple "TD1".
- Sélectionner un emplacement où le projet sera enregistré (par exemple sur une clé USB).
- Cliquer sur "OK".

Créer un nouveau fichier source "fonction.cpp" de la manière suivante :

- Aller dans le menu "Projet / Ajouter un nouvel élément".
- Sélectionner "Fichier C++ (.cpp)" dans la liste.
- Entrer le nom du fichier en face de "Nom :" (ici `fonction`).
- Cliquer sur le bouton "Ajouter".

sous Code : :Blocks : Créer un projet en C++ de la manière suivante :

- Aller dans le menu "File / New / Project".
- Sélectionner "Console Application".
- Cliquer "Next>".
- Sélectionner le langage C++ puis cliquer sur "Next>".
- Entrer un titre pour le projet, par exemple "TD1".
- Sélectionner un emplacement où le projet sera enregistré (par exemple sur une clé USB).
- Cliquer sur "Next>" puis "Finish". Noter qu'un fichier "main.cpp" a déjà été créé avec une fonction `main` : **supprimer le pour cette fois**.

Créer un nouveau fichier source "fonction.cpp" de la manière suivante :

- Aller dans le menu "File / New / File...".
- Sélectionner "C/C++ source" puis cliquer sur "Go".
- Sélectionner le langage C++ puis cliquer sur "Next>".
- Entrer le nom du fichier (ici `fonction`).
- Cocher les contextes de compilation dans lesquels seront utilisés les fichiers (a priori aussi bien "Debug" que "Release" (si vous ne faites pas cette opération, votre fichier source sera ignoré à la compilation).
- Cliquer sur "Finish".

Question 1

Ajouter le code suivant :

```
#include<iostream>
#include<string>
using namespace std;
void bonjour() {
 cout<<"Entrez votre prenom :";
 string prenom;
 cin>>prenom;
 cout<<"Bonjour "<<prenom<<"\n";
}
```

fonction.cpp

Compiler le projet avec "Générer/Générer *nom_du_projet*". Que se passe t-il ? Expliquer.

Question 2

À la suite de la fonction `bonjour()`, ajouter le code suivant :

```
/*...*/
int main() {
 bonjour();
 return 0;
}
```

fonction.cpp

Re-compiler le projet. Que se passe t-il ? Expliquer.

Exercice 2 - Fichier source *vs.* fichier d'en-tête

Reprendre l'exercice précédent en ajoutant un fichier d'entête `fonction.h` de la manière suivante :

sous Visual :

- Aller dans le menu "Projet / Ajouter un nouvel élément".
- Sélectionner "Fichier d'en-tête (.h)" dans la liste.
- Entrer le nom du fichier en face de "Nom :" (ici "fonction").
- Cliquer sur le bouton "Ajouter".

sous Code : :Blocks :

- Aller dans le menu "File / New / File...".
- Sélectionner "C/C++ header" puis cliquer sur "Go".
- Sélectionner le langage C++ puis cliquer sur "Next>".
- Entrer le nom du fichier (ici `fonction`).
- Cocher les contextes de compilation dans lesquels seront utilisés les fichiers.
- Cliquer sur "Finish".

Question 1

Ajouter le texte suivant dans `fonction.h` :

```
Une université de technologie,
est un établissement à caractère scientifique, culturel et professionnel
qui a pour mission principale la formation des ingénieurs,
le développement de la recherche et de la technologie.
```

fonction.h

Compiler le projet. Que se passe t-il ? Expliquer.

Question 2

Ajouter l'instruction « `#include "fonction.h"` » en haut du fichier `fonction.cpp`. Compiler le projet. Que se passe t-il ? Expliquer.

Exercice 3 - Fichier d'en-tête et déclaration

Reprendre l'exercice précédent en ajoutant un fichier source `main.cpp`. On repart alors des trois fichiers `fonction.h`, `fonction.cpp` et `main.cpp` qui contiennent respectivement les codes suivants :

```
// vide pour l'instant
```

fonction.h

```
#include<iostream>
#include<string>
using namespace std;
void bonjour() {
 cout<<"Entrez votre prenom :";
 string prenom;
 cin>>prenom;
 cout<<"Bonjour "<<prenom<<"\n";
}
```

fonction.cpp

```
int main() {
 bonjour();
 return 0;
}
```

main.cpp

Question 1

Compiler le projet. Que se passe t-il ? Expliquer.

Question 2

Corriger le problème de deux façons différentes : une fois en modifiant uniquement le fichier `main.cpp`, une fois en modifiant aussi le fichier `fonction.h`.

Exercice 4 - Instructions d'inclusions conditionnelles

- Ajouter au projet un fichier d'entête `essai.h`.
- Ajouter l'instruction **`#include`** `"fonction.h"` dans le fichier `essai.h`.
- Ajouter l'instruction **`#include`** `"essai.h"` dans le fichier `fonction.h`.
- Compiler le projet.

Que se passe t-il ? Expliquer. Corriger le problème.

Exercice 5 - E/S en C++, définition de variables

Réécrire le programme suivant en ne faisant appel qu'aux nouvelles possibilités d'entrées-sorties du C++ (*c.-à-d.* en évitant les appels à `printf` et `scanf`). Définir le plus tard possible les variables. Utiliser une constante plutôt que l'instruction **#define** du préprocesseur.

```
#include<stdio.h>
#define PI 3.14159
void exerciceA(){
 int r; double p, s;
 printf("donnez le rayon entier d'un cercle:");
 scanf("%d",&r);
 p=2*PI*r;
 s=PI*r*r;
 printf("le cercle de rayon %d ",r);
 printf("a un perimetre de %f et une surface de %f\n",p,s);
}
```

fonction.cpp

Exercice 6 - Définition - Initialisation - Affectation

Dans la fonction `main()`, 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** et l'affecter avec la valeur `3.14`. Tentez d'afficher sa valeur avant et après affectation. Que se passe-t-il ?

Exercice 7 - Variables constantes

Définir une variable constante `pi` de type **double**, lui donner la valeur `3.14`, et afficher sa valeur. Tenter ensuite d'affecter cette variable avec une autre valeur.

Exercice 8 - Espaces de noms

Voici deux fonctions qui portent le même nom :

```
void bonjour() {
 cout<<"nichao\n";
}
```

et

```
void bonjour() {
 cout<<"hello\n";
}
```

À partir d'un nouveau projet avec 3 fichiers `fonction.h`, `fonction.cpp`, `main.cpp`, déclarer les deux fonctions `bonjour` dans le fichier `fonction.h` en utilisant deux namespaces différents. Définir ces fonctions dans le fichier `fonction.cpp`. Enfin, utiliser ces deux fonctions dans la fonction `main`. Modifier le programme pour ne pas utiliser l'instruction **using namespace** `std;`.

Exercice 9 - Surcharge de fonction - Fonctions **inline**

Soit les fonctions de nom `fct` suivantes :

```
/* ...*/
int fct(int x);
int fct(float  y);
int fct(int x, float  y);
float fct(float x, int  y);
```

fonction.h

```
/* ...*/

int fct(int x){ std::cout<<"1:"<<x<<"\n"; return 0; }
int fct(float  y){ std::cout<<"2:"<<y<<"\n"; return 0; }
int fct(int x, float  y){ std::cout<<"3:"<<x<<y<<"\n"; return 0; }
float fct(float x, int  y){ std::cout<<"4:"<<x<<y<<"\n"; return 3.14; }

void exercice_surcharge(){
 int i=3,j=15;
 float x=3.14159,y=1.414;
 char c='A';
 double z=3.14159265;
 fct(i); //appel 1
 fct(x); //appel 2
 fct(i,y); //appel 3
 fct(x,j); //appel 4
 fct(c); //appel 5
 fct(i,j); //appel 6
 fct(i,c); //appel 7
 fct(i,z); //appel 8
 fct(z,z); //appel 9
}
```

fonction.cpp

Question 1

Les appels de fonction sont-ils corrects et, si oui, quelles seront les fonctions effectivement appelées et les conversions mises en place ? Expliquer les appels incorrects et les corriger.

Question 2

Transformer le programme précédent (corrigé) pour que les fonctions fct deviennent des fonctions en ligne.

Exercice 10 - Pointeurs - Pointeurs **const**

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

```
double* pt0=0;
double* pt1=4096;
double* pt2=(double*)4096;
void* pt3=pt1;
pt1=pt3;
pt1=(double*)pt3;
double d1=36;
const double d2=36;
double* pt4=&d1;
const double* pt5=&d1;
*pt4=2.1;
*pt5=2.1;
pt4=&d2;
pt5=&d2;
double* const pt6=&d1;
pt6=&d1;
*pt6=2.78;
double* const pt6b=&d2;
const double* const pt7=&d1;
pt7=&d1;
*pt7=2.78;
double const* pt8=&d1;
pt8=&d2;
pt8=&d1;
*pt8=3.14;
```

dans une unité de compilation .cpp

Exercice 11 - Référence - Références `const`

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

```
double& d1=36;
double d2=36;
double& ref=d2;
ref=4;
const double d3=36;
const double& d4=36;
const double& d5=d2;
d5=21;
const double& d6=d3;
double& ref2=d6;
int i=4;
double& d7=i;
const double& d8=i;
d8=3;
```

dans une unité de compilation .cpp

Exercice 12 - Passage d'argument par adresse et par référence

Question 1

Écrire la fonction qui permet d'inverser les valeurs de deux variables entière passées en argument en utilisant des passages par adresse (prototype : `void inverse(int* a, int* b);`).

Question 2

Écrire la fonction qui permet d'inverser les valeurs de deux variables entière passées en argument en utilisant des passages par référence (prototype : `void inverse(int& a, int& b);`).

Exercice 13 - Passage d'argument par adresse et par référence

Soit le modèle de structure suivant :

```
/*...*/
struct essai {
 int n;
 float x;
};
```

fonction.h

Ecrire une fonction nommée `raz` permettant de remettre à zéro les 2 champs d'une structure de ce type transmise en argument.
Faire l'exercice une fois en utilisant un passage par adresse, une fois en utilisant un passage par référence. Dans les deux cas, on écrira un petit programme d'essai de la fonction. Il affichera les valeurs d'une structure de ce type après appel de la dite fonction.

Exercice 14 - Arguments par défaut

Soit la structure suivante :

```
/*...*/
struct point {
 int x;
 int y;
 int z;
};
```

fonction.h

Après avoir placé les déclarations de cette structure dans le fichier d'en-tête `fonction.h`, faire les modifications nécessaire pour simplifier le programme suivant en utilisant les nouvelles possibilités du C++. Ajouter aussi les déclarations qui semblent utiles dans le fichier d'entête.

```
#include "fonction.h"
/*...*/
void init(point* pt, int _x, int _y, int _z) {
 pt->x=_x; pt->y=_y; pt->z=_z;
}
void init(point* pt, int _x, int _y) {
 pt->x=_x; pt->y=_y; pt->z=0;
}
void init(point* pt, int _x) {
 pt->x=_x; pt->y=0; pt->z=0;
}
void init(point* pt) {
 pt->x=0; pt->y=0; pt->z=0;
}
void essai_init() {
 point p;
 init(&p);
 init(&p,1);
 init(&p,1,2);
 init(&p,1,2,3);
}
```

fonction.cpp

Exercice 15 - Allocation dynamique

Écrire plus simplement en C++ les instructions suivantes, en utilisant les opérateurs **new**, **new**[], **delete** et **delete**[] :

```
void essai_alloc(){
 int* pt_int;
 double* pt_double;
 pt_int=(int*)malloc(sizeof(int));
 pt_double=(double*)malloc(sizeof(double)*100);
 free(pt_int);
 free(pt_double);
}
```

fonction.cpp

Exercice 16 - Structures et tableaux

Soit la structure définie dans le fichier `fonction.h` :

```
/*...*/
struct personne {
 char nom[30];
 unsigned int age;
};
```

fonction.h

Question 1

Écrire une fonction `raz` qui permet d'initialiser le champ `nom` et le champ `age` d'une variable de type `personne` transmise en argument, respectivement avec la chaîne de caractères vide et la valeur 0.

Question 2

Écrire une fonction `affiche_struct` qui permet d'afficher les attributs d'une donnée de type `personne` transmise en argument.

Question 3

Écrire une fonction `affiche_tab` qui permet d'afficher les attributs d'un tableau de `personne` dont l'adresse et le nombre d'éléments sont passés en argument.

Question 4

Écrire une fonction `init_struct` qui permet d'initialiser une structure dont l'adresse est passée en argument avec une chaîne de caractères et un entier passés en arguments. Ne pas utiliser de fonction de type `strcpy` pour cet exercice.

Question 5

Écrire une fonction `copy_struct` qui permet de copier les différents champs d'une donnée de type `personne` dans une autre donnée de type personne.

Question 6

Écrire une fonction `copy_tab` qui permet de copier un tableau de variables de type `personne` dans un autre tableau du même type.

Question 7

Écrire une fonction qui utilise l'ensemble de ces fonctions.

Exercice 17 - Classe `string`

Transformer le programme précédent de manière à utiliser des variables de type `string` (entête `<string`) plutôt que des chaînes de caractères de type `C`.

Exercice 18 - Retour de fonction par référence

Soit le modèle de structure suivant :

```
#include<string>
struct compte {
 string id;
 int solde;
};
```

fonction.h

Ecrire une fonction `operation` qui permet de faire les opérations suivantes en leur donnant du sens.
Le premier argument de cette fonction est un tableau de structures de type `compte`. Le deuxième argument est une chaîne de caractères qui permet d'identifier dans ce tableau le compte particulier à utiliser : il s'agit de celui dont le champs `id` est égal à cette chaîne. Il s'agit de faire en sorte que le solde du compte soit modifié conformément à l'opération souhaitée. On supposera que le compte voulu par l'utilisateur existe bien dans le tableau.

```
void essai_comptes(){
 compte tab[4]={ {"courant", 0},{"codevi", 1500 },
      {"epargne", 200 }, { "cel", 300 } };
 operation(tab,"courant")=100;
 operation(tab,"codevi")+=100;
 operation(tab,"cel")-=50;
 for(int i=0; i<4; i++) cout<<tab[i].id<<" : "<<tab[i].solde<<"\n";
}
```

fonction.cpp

Exercice 19 - La classe **Fraction**

Créer un projet vide et ajouter trois fichiers `fraction.h`, `fraction.cpp` et `main.cpp`. Définir la fonction principale `main()` dans le fichier `main.cpp`. S'assurer que le projet compile correctement. Dans cet exercice, on tâchera de mener une approche "compilation séparée". Au fur et à mesure de l'exercice, on pourra compléter la fonction principale en utilisant les éléments créés.

Question 1

Définir en C++ une classe `Fraction` qui comportera deux attributs `numerateur` et `denominateur` de type **`int`**. Englober cette classe dans un espace de noms `MATH`. Dans la fonction `main()`, définir un objet `Fraction` avec la valeur `3/4`.

Question 2

Faire en sorte que les attributs de la classe ne soient plus accessibles par un utilisateur de la classe (si ce n'était pas déjà le cas). Déclarer et définir des accesseurs en lecture des attributs de la classe (`getNumerateur()` et `getDenominateur()`) et un accesseur en édition des attributs la classe (`setFraction(int n, int d)`). Faire attention à la validité des valeurs stockées dans les attributs. Les accesseurs en lecture seront (**`inline`**) alors que l'accesseur en écriture (édition) sera définis dans le fichier `fraction.cpp`.

Question 3

Déclarer et définir un ou plusieurs constructeurs pour cette classe. Faire attention à la validité des valeurs stockées dans les attributs. Faire cette question deux fois : une fois en utilisant des affectations pour donner une valeur initiale aux attributs, et une fois en utilisant des initialisations.

Question 4

Ajouter à la classe `Fraction` **la méthode privée** `simplification()` suivante (en tant que méthode non-**`inline`**) qui permet de simplifier une fraction. Utiliser cette méthode pour améliorer le(s) constructeur(s) et/ou l'accesseur en écriture quand cela vous paraît utile.

```
// à ajouter en tant méthode privée de la classe Fraction
void MATH::Fraction::simplification(){
 // si le numerateur est 0, le denominateur prend la valeur 1
 if (numerateur==0) { denominateur=1; return; }
 /* un denominateur ne devrait pas être 0;
    si c'est le cas, on sort de la méthode */
 if (denominateur==0) return;
 /* utilisation de l'algorithme d'Euclide pour trouver le Plus Grand Commun
    Denominateur (PGCD) entre le numerateur et le denominateur */
 int a=numerateur, b=denominateur;
 // on ne travaille qu'avec des valeurs positives...
 if (a<0) a=-a; if (b<0) b=-b;
 if (denominateur==1) return;
 while(a!=b){ if (a>b) a=a-b; else b=b-a; }
 // on divise le numerateur et le denominateur par le PGCD=a
 numerateur/=a; denominateur/=a;
 // si le denominateur est négatif, on fait passer le signe - au numérateur
 if (denominateur<0) { denominateur=-denominateur; numerateur=-numerateur; }
}
```

fraction.cpp

Question 5

Ecrire la méthode `somme` qui permet de faire une addition de 2 fractions. Notons que $\frac{a}{b} + \frac{c}{d} = \frac{ad+cb}{bd}$. Réfléchir aux types de retour possibles pour cette fonction.

Question 6

Ajouter un destructeur à la classe `Fraction`. Ce destructeur contiendra une instruction permettant d'afficher sur le flux `cout` un message de destruction avec l'adresse de l'objet concerné. De même, ajouter un message de construction dans le ou les constructeurs de la classe `Fraction`. Recopier le code suivant dans le fichier `main.cpp`, exécuter le programme et analyser les constructions et destructions des objets.

```cpp
#include <iostream>
#include "fraction.h"
using namespace std;
using namespace MATH;

Fraction* myFunction(){
 Fraction fx(7,8);
 Fraction* pfy=new Fraction(2,3);
 return pfy;
}

int main(){
  Fraction f1(3,4);
  Fraction f2(1,6);
  Fraction* pf3=new Fraction(1,2);
  cout<<"ouverture d'un bloc\n";
  Fraction* pf6;
  {
   Fraction f4(3,8);
     Fraction f5(4,6);
     pf6=new Fraction(1,3);
  }
  cout<<"fin d'un bloc\n";
  cout<<"debut d'une fonction\n";
  Fraction* pf7=myFunction();
  cout<<"fin d'une fonction\n";
  cout<<"desallocations controlee par l'utilisateur :\n";
  delete pf6;
  delete pf7;
  return 0;
}
```

fraction.cpp

Question 7

Représenter la classe `Fraction` en UML.

Exercice 20 - La classe `Fraction` : surcharge d'opérateurs

Question 1

Surcharger l'opérateur binaire + de façon à pouvoir effectuer la somme entre deux objets `Fraction`.

Question 2

Après avoir étudié la possibilité d'une conversion implicite du type **`int`** vers le type `Fraction`, surcharger (si besoin) l'opérateur + de manière à rendre possible cette opération entre valeurs de type **`int`** et valeurs de type `Fraction`.

Question 3

Surcharger l'opérateur ++ en version préfixe et postfixe pour la classe `Fraction`.

Question 4

Surcharger l'opérateur << qui pourra permettre d'afficher une fraction en utilisant un flux `ostream` comme `cout`. Faire attention au type de retour de la fonction.

Exercice 21 - La classe `Fraction` : Exceptions

Question 1

Réécrire la méthode `Fraction::setFraction` de façon à traiter l'erreur d'un éventuel dénominateur nul en déclenchant une exception. Le mot clé **`throw`** sera utilisé avec une littérale chaîne de caractère décrivant l'erreur.

Question 2

Dans la fonction `main`, ajouter les instructions qui permettent de capturer une éventuelle exception due à un dénominateur nul et d'afficher le message correspondant.

Question 3

Ecrire une classe `FractionException` destinée à gérer les situations exceptionnelles de la classe `Fraction`. Cette classe comportera un attribut `info` de type **`char`**`[256]` destinée à stocker un message d'erreur et une méthode `getInfo()` permettant d'accéder à cette information. La classe comportera également un constructeur permettant de construire un objet `FractionException` avec une chaine de caractères.

Question 4

Réécrire la méthode `Fraction::setFraction` de façon à traiter l'erreur d'un éventuel dénominateur nul en déclenchant une exception de type `FractionException`.

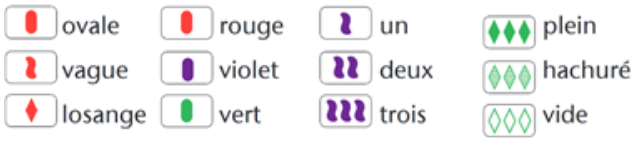
Question 5

Dans la fonction `main`, modifier les instructions de manière à capturer les exceptions de type `FractionException`.

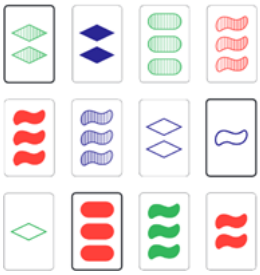
Exercice 22 - SET !

Remarque : Le but de cet exercice est de se familiariser avec les notions d'association, d'agrégation et de composition entre classes et de leurs conséquences au niveau de l'implémentation des classes.

Le but est de constituer un ensemble de classes permettant de jouer au SET! qui est un jeu de cartes édité par GIGAMIC. Il est composé de 81 cartes toutes différentes. Chaque carte combine des symboles en utilisant une forme parmi {ovale, vague, losange}, une couleur parmi {rouge, violet, vert}, un nombre parmi {1, 2, 3}, et un remplissage parmi {plein, vide, hachure}.



Un SET est un ensemble de 3 cartes telles que, pour chacune de ces cartes, chacune des 4 caractéristiques est soit strictement identique, soit totalement différente aux 2 autres cartes.



Les trois cartes forment donc un SET si et seulement si :
- la couleur des symboles est 3 fois strictement identique ou 3 fois totalement différente ;
- la forme des symboles est 3 fois strictement identique ou 3 fois totalement différente ;
- le nombre de symboles est 3 fois strictement identique ou 3 fois totalement différent ;
- le remplissage des symboles est 3 fois strictement identique ou 3 fois totalement différent.

Au début de la partie, une pioche est constituée avec les cartes mélangées. Au fur et à mesure des besoins, on tire des cartes dans la pioche. Le jeu débute avec 12 cartes tirées et posées sur le plateau de jeu. Les joueurs doivent identifier le plus vite possible un SET sur ces 12 cartes. Le joueur qui identifie le premier un SET remporte les 3 cartes correspondantes qui sont retirées du jeu. Le tirage est alors complété de manière à avoir au moins 12 cartes. Il arrive (fréquemment) que dans le tirage courant, il n'existe pas de SET sur l'ensemble des 12 cartes (ou que les joueurs ne sont pas assez bons pour identifier un des SETs du tirage). Le tirage est alors complété avec une 13^e carte, puis une 14^e carte, etc., jusqu'à ce qu'un joueur parvienne à identifier un SET. La partie termine quand la pioche est épuisée et quand les joueurs n'arrivent plus à trouver de SET. Le gagnant est le joueur qui a remporté le plus de SETs.

Préparation : Créer un projet vide et ajouter les trois fichiers `set.h`, `set.cpp` et `main.cpp` fournis en ressource. Le fichier `set.h` définit les énumérations `Couleur`, `Nombre`, `Forme` et `remplissage` qui permettront de caractériser les cartes du jeu. Différentes fonctions permettent de manipuler des variables de ces types (transformation en texte, affichage sur un flux `ostream`). Des fonctions telles que `printCouleurs()` sont fournies à titre d'exemple pour vous familiariser à l'utilisation des listes `Couleurs`, `Nombres`, `Formes` et `Remplissages` qui permettent d'itérer facilement sur les différentes valeurs que peuvent prendre les variables d'un de ces type **enum**. Les situations exceptionnelles seront gérées en utilisant la classe d'exception `SetException`.

Dans cet exercice, on tâchera de mener une approche "compilation séparée". Au fur et à mesure de l'exercice, on pourra compléter la fonction principale en utilisant les éléments créés.
Au fur et à mesure du traitement des questions, on complétera un diagramme de classes représentant les différentes associations entre les classes décrites et leurs multiplicités.

D'un point de vue implémentation, une carte du jeu correspond à un objet instance d'une classe `Carte` permettant de la manipuler. Le cycle de vie (création, destruction) des 81 cartes est géré par une instance de la classe `Jeu`. Une instance de la classe `Pioche` va permettre de mélanger et distribuer les cartes au fur et à mesure des besoins de la partie. L'ensemble des cartes visibles par les joueurs est représenté par une instance de la classe `Plateau`. Un objet de la classe `Combinaison` représente un ensemble de 3 cartes formant potentiellement un SET. Un objet de la classe `Controleur` va permettre de contrôler le déroulement du jeu.

La classe `Carte` comporte 4 attributs qui permettent de représenter les 4 caractéristiques d'une carte (couleur, nombre, forme, remplissage). Les méthodes `getCouleur()`, `getNombre()`, `getForme()`, `getRemplissage()`

permettent de connaître la valeur de ces attributs. La classe `Carte` possède un unique constructeur à 4 arguments permettant d'initialiser chacune des caractéristiques de la carte. On peut écrire (afficher) un objet `Carte` sur un flux `ostream`. Il est (pour l'instant) possible de dupliquer un objet `Carte` par construction ou affectation à partir d'un autre objet `Carte`.

1. Définir la classe `Carte` et ses fonctions associées. Définir le destructeur, le constructeur de recopie et l'opérateur d'affectation de la classe `Carte` uniquement si c'est nécessaire.
2. Est-il possible de définir un tableau contenant des objets `Carte` sans initialisateur ? Est-il possible de définir un tableau contenant des pointeurs d'objets `Carte` sans initialisateur ?

Une instance de la classe `Jeu` permet de gérer le cycle de vie de tous les objets `Carte` nécessaires pour une partie. La classe possède un attribut `cartes` de type **`const`** `Carte* [81]`, *c.-à-d.* un tableau de 81 pointeurs de type **`const`** `Carte*`. Chaque valeur de ce tableau correspond à l'adresse d'un des objets `Carte` qui seront alloués dynamiquement par l'unique constructeur sans argument de la classe `Jeu`. La méthode `getNbCartes()` renvoie le nombre de cartes du jeu (81). La classe `Jeu` est responsable du cycle de vie de ces objets `Carte`. La méthode de prototype « **`const`** `Carte& getCarte(size_t i)` **`const`** » permet d'obtenir une référence sur la carte pointée par le i^e pointeur de `cartes`. Cette méthode déclenche une exception si `i` est supérieur ou égal à 81. Il est impossible de dupliquer un objet `Jeu` par construction ou affectation à partir d'un autre objet `Jeu`.

3. Quelle association lie les classes `Carte` et `Jeu` ?
4. Quel est l'intérêt d'avoir utilisé un attribut de type **`const`** `Carte*[81]` plutôt que `Carte*[81]` ?
5. Définir la classe `Jeu` et les fonctions associées. Définir le destructeur de la classe `Jeu` uniquement si c'est nécessaire. Ajouter des instructions pour interdire la construction par recopie ou l'affectation entre objets `Jeu`.

La classe `Pioche` permet de gérer une pioche de cartes. Son unique constructeur prend en paramètre une variable `j` de type **`const`** `Jeu&` référençant le jeu de cartes. Le constructeur alloue un tableau d'une taille égale au nombre de cartes du jeu et stocke son adresse dans l'attribut `cartes` de type **`const`** `Carte**`. Chaque cellule de ce tableau contient une adresse de type **`const`** `Carte*` pointant sur une carte du jeu. Initialement la pioche contient toutes les adresses des cartes de l'objet `Jeu` transmis en paramètre. Un objet `Pioche` n'a aucune responsabilité sur le cycle de vie des objets `Carte` dont elle contient l'adresse (c'est l'objet `Jeu` transmis en paramètre qui s'en occupe). L'attribut `nb` de type `size_t` est égal au nombre de cartes de la pioche. La méthode `getNbCartes()` permet de connaître sa valeur. La méthode `estVide()` renvoie **`true`** si la pioche ne contient plus de carte et **`false`** sinon. La classe dispose de la méthode `piocher()` pour obtenir une carte au hasard. Si la pioche est vide, cette méthode déclenche une exception, sinon elle renvoie une référence sur la carte qui est enlevée de la pioche (l'attribut `nb` diminue).

6. Quelle association lie les classes `Carte` et `Pioche` ?
7. Définir la classe `Pioche`. On étudiera l'intérêt d'utiliser **`explicit`** devant le constructeur de la classe. Définir le destructeur de la classe `Pioche` uniquement si c'est nécessaire. Ajouter des instructions pour interdire la construction par recopie ou l'affectation entre objets `Pioche`.

La classe `Plateau` permet de stocker les cartes visibles par les joueurs. Initialement le plateau est vide (il ne contient pas de carte). L'unique constructeur de cette classe est sans argument. La méthode `ajouter()` permet d'ajouter une carte au plateau. La classe contient un attribut `cartes` de type **`const`** `Carte**` qui pointe sur un tableau alloué dynamiquement de pointeurs de type **`const`** `Carte*`. À chaque fois que la méthode `ajouter()` est appelée elle stocke l'adresse de l'objet `Carte` dont une référence est transmise à la méthode. La méthode `retirer()` permet de retirer du plateau la carte dont une référence est transmise à la méthode. Le tableau de pointeurs s'agrandira au fur et à mesure des besoins. L'attribut `nb` sera égal au nombre de cartes du tableau alors que l'attribut `nbMax` sera égal à la taille réelle du tableau alloué dynamiquement. La méthode `print()` permet d'afficher les cartes du plateau sur un flux `ostream`.

8. Quelle association lie les classes `Carte` et `Plateau` ?
9. Définir la classe `Plateau`. Définir le destructeur de la classe `Plateau` uniquement si c'est nécessaire. Ajouter des instructions pour permettre la construction par recopie ou l'affectation entre objets `Plateau` pour le cas où, pour un futur besoin, nous souhaiterions faire l'historique des différents états du plateau lors d'une partie.

La classe `Combinaison` représente un ensemble de 3 cartes choisies par un joueur. Elle contient 3 attributs `c1`, `c2`, `c3` de type **`const`** `Carte*` pointant sur les cartes concernées. L'unique constructeur de cette méthode possède 3 paramètres de type **`const`** `Carte&` permettant d'initialiser ses attributs. Les méthodes `getCarte1()`, `getCarte2()` et `getCarte3()` permettent d'obtenir des références sur ces cartes. La méthode `estUnSET()` renvoie **`true`** si la combinaison forme un `SET` et **`false`** sinon. On peut écrire (afficher) un objet `Combinaison`

sur un flux `ostream`. Il est possible de dupliquer un objet `Combinaison` par construction ou affectation à partir d'un autre objet `Combinaison`.

10. Quelle association lie les classes `Carte` et `Combinaison` ?
11. Définir la classe `Combinaison`. Définir le destructeur, le constructeur de recopie et l'opérateur d'affectation de la classe `Combinaison` uniquement si c'est nécessaire.

La classe `Controleur` permet de gérer les parties de SET !. Pour l'instant, elle possède un attribut `jeu` de type `Jeu`, un attribut `plateau` de type `Plateau`, un attribut `pioche` de type `Pioche*`. L'unique constructeur sans argument de cette classe alloue dynamiquement une pioche (dont l'objet `Controleur` est responsable) et stocke son adresse dans `pioche`. La méthode `distribuer()` permet de transférer des cartes de la pioche vers le plateau. Si le plateau contient moins de 12 cartes, `distribuer()` tente de complèter le plateau pour qu'il contienne 12 cartes. Si le plateau contient au moins 12 cartes, `distribuer()` tente d'ajouter une seule carte au plateau. Ces actions sont effectuées si la pioche contient des cartes. Dans le cas contraire, la méthode ne fait rien.

12. Quelle association lie les classes `Controleur` et `Jeu` ? Quelle association lie les classes `Controleur` et `Pioche` ?
13. Définir la classe `Controleur`. Définir le destructeur de la classe `Controleur` uniquement si c'est nécessaire. Ajouter des instructions pour interdire la construction par recopie ou l'affectation entre objets `Controleur`. À titre d'exemple, le programme

```
#include "set.h"
using namespace Set;

int main() {
 try {
  Controleur c;
  c.distribuer();
  cout << c.getPlateau();
  c.distribuer();
  cout << c.getPlateau();

 }
 catch (SetException& e) {
  std::cout << e.getInfo() << "\n";
 }
 return 0;
}
```

pourrait produire l'affichage suivant :

```
(2,M,~,H) (2,V,~,H) (2,R,~,H) (1,V,L,P)
(3,V,~,H) (3,V,O,P) (1,R,~,P) (1,V,~,P)
(1,M,O,P) (1,V,L,H) (1,M,O,_) (1,R,L,P)

(2,M,~,H) (2,V,~,H) (2,R,~,H) (1,V,L,P)
(3,V,~,H) (3,V,O,P) (1,R,~,P) (1,V,~,P)
(1,M,O,P) (1,V,L,H) (1,M,O,_) (1,R,L,P)
(1,M,L,P)
```

14. Établir un modèle UML où apparaissent les différentes classes utilisées dans l'application ainsi que les associations entre ces classes. On fera aussi apparaître les valeurs de multiplicité de ces classes.

Exercice 23 - Problèmes de conception

Dans l'application, les objets `Carte` sont gérés par un module appelé `Jeu` qui est responsable de leur création et destruction.

1. Expliciter des intérêts de mettre en place le design pattern *Singleton* pour la classe `Jeu`. Transformer la classe `Jeu` en singleton. On étudiera les différentes possibilités.
2. Modifier les classes qui utilisent `Jeu` en profitant de l'accès central de l'instance `Jeu`. Proposer une nouvelle version du diagramme de classe pour tenir compte de cela.
3. Faire en sorte que seule l'instance de la classe `Jeu` puisse créer des objets `Cartes`.

La méthode `Jeu::getCarte()` permet d'accéder une carte à partir d'un numéro qui est arbitraire du point de vue de l'utilisateur et qui expose la structure de données utilisée.

4. Afin de pouvoir parcourir séquentiellement les cartes du jeu, appliquer le design pattern *Iterator* à cette classe en déduisant son implémentation du code suivant :

```
void afficherCartes(){
  for(Jeu::Iterator it= Jeu::getInstance().getIterator();!it.isDone();it.next
      ())
   std::cout<<it.currentItem()<<"\n";
}
```

   Mettre la méthode `Jeu::getCarte()` dans la partie privée en rendant son accès exlusif à la classe `Jeu::Iterator`. Modifier les éléments du code impacté par la modification de l'interface de la classe `Jeu`.

En fait, plusieurs types d'itérateurs peuvent être proposés par une classe afin d'offrir des services particuliers.

5. Compléter la classe `Jeu` afin de pouvoir parcourir séquentiellement uniquement les cartes du jeu ayant une forme donnée :

```
void afficherCartes(Forme f){
  for(Jeu::FormeIterator it= Jeu::getInstance().getIterator(f);!it.isDone();it
      .next())
   std::cout<<it.currentItem()<<"\n";
}
```

6. Appliquer le design pattern *Iterator* pour parcourir les cartes d'un plateau en proposant une interface d'itérateur similaire à celle utilisée par les conteneurs standards du C++ (STL) :

```
void afficherCartes(const Plateau& p){
 for(Plateau::const_iterator it=p.begin();it!=p.end();++it)
   std::cout<<*it<<"\n";
}
```

Remarque : Puisque les cartes (du jeu ou d'un plateau) sont non modifiables, on remarquera que tous les itérateurs implémentés dans cet exercice proposent un accès uniquement en lecture. En pratique, dans une interface similaire aux conteneurs standards du C++, on a aussi la classe `iterator` (en plus de `const_iterator`). L'opérateur d'indirection $\star$ renvoie alors une référence non-**const** de l'objet désigné par l'itérateur, ce qui permet de modifier éventuellement cet objet. Quand les deux itérateurs (**const** et non-**const**) sont implémentés, les versions non-**const** de `begin()` et `end()` renvoie un `iterator`, alors que les versions **const** de `begin()` et `end()` renvoie un `const_iterator`. Afin de pouvoir obtenir un `const_iterator` à partir d'un objet non constant, on implémente aussi souvent les méthodes `cbegin()` et `cend()`, disponibles en version **const** uniquement (elles sont donc appelables à partir d'un objet **const** ou non-**const**) et qui renvoient un `const_iterator`.

7. **(question d'approfondissement à faire chez soi ou en TD s'il reste du temps)** Finaliser le jeu de manière à pouvoir jouer au SET !.

Exercice 24 - Graphes et STL

Un graphe dirigé peut être défini comme un couple $G = (V, E)$ où V est un ensemble de *sommets* et E un ensemble de couples $(i, j) \in V \times V$ que l'on appelle *arcs*. Un sommet $j \in V$ est dit *successeur* d'un sommet $i \in V$ si $(i, j) \in E$. Un sommet $j \in V$ est dit *prédécesseur* d'un sommet $i \in V$ si $(j, i) \in E$.

Remarque : en anglais, "sommet" se dit "vertex" (pluriel : "vertices"), et "arc" se dit "edge".

Exemple :
- $V = \{0, 1, 2, 3, 4, 5, 6\}$.
- $E = \{(2, 0), (2, 1), (2, 3), (3, 1), (1, 4), (6, 0), (6, 5), (6, 3), (0, 4), (4, 0), (4, 5), (4, 6)\}$.
- L'ensemble des successeurs du sommet 2 est $\{0, 1, 3\}$. L'ensemble des prédecesseurs de 4 est $\{0, 1\}$.

Pour manipuler un graphe dans un programme, une des structures de données les plus utilisées est la liste d'adjacence. Elle consiste en un tableau dont l'entrée i donne la liste des successeurs du sommet i.

Le but de cet exercice est de se familiariser avec la STL en implémentant une classe de graphe.

Préparation : Créer un projet vide et ajouter trois fichiers `graph.h`, `graph.cpp`, et `main.cpp`. Les situations exceptionnelles seront gérées en utilisant la classe d'exception suivante (à recopier dans le fichier `graph.h`) :

```
#if !defined(_GRAPH_H)
#define _GRAPH_H
#include<string>
#include<stdexcept>
using namespace std;
class GraphException : public exception {
 string info;
public:
 GraphException(const string& i) noexcept :info(i){}
 virtual ~GraphException() noexcept {}
 const char* what() const noexcept { return info.c_str(); }
};
#endif
```

Définir la fonction principale `main` dans le fichier `main.cpp`. S'assurer que le projet compile correctement. Implémentez une classe graphe qui utilise un objet `vector<list<unsigned int> >` pour représenter la liste d'adjacence d'un graphe dont les n sommets sont identifiés par les nombres de $\{0, 1, ..., n-1\}$. Implémentez toutes les méthodes de l'interface suivante :

```
#include<list>
#include<vector>
#include<iostream>
#include<string>
using namespace std;
class Graph {
 vector<list<unsigned int> > adj;
 string name;
public:
 Graph(const string& n, size_t nb);
 const string& getName() const;
 size_t getNbVertices() const;
 size_t getNbEdges() const;
 void addEdge(unsigned int i, unsigned int j);
 void removeEdge(unsigned int i, unsigned int j);
 const list<unsigned int>& getSuccessors(unsigned int i) const;
 const list<unsigned int> getPredecessors(unsigned int i) const;
};
ostream& operator<<(ostream& f, const Graph& G);
```

Le constructeur de la classe `Graph` prend en argument le nom du graphe ainsi que le nombre de sommets le constituant (qui n'évoluera pas au cours de la vie du graphe). Exploitez au mieux les algorithmes de la STL.

Exemple :

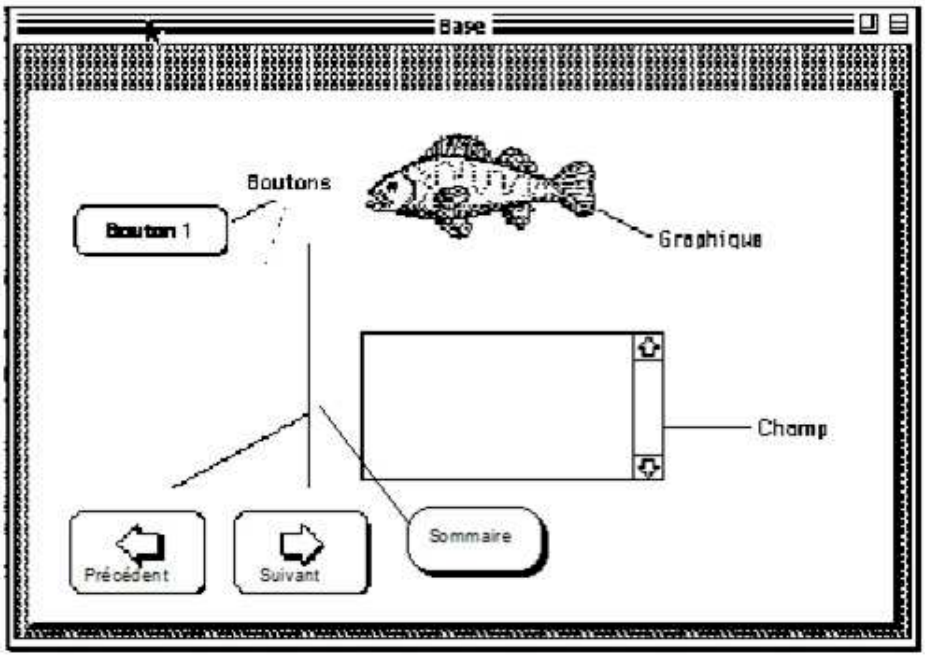
```
try{
  Graph G1("G1",5); cout<<G1;
  G1.addEdge(0,1); G1.addEdge(0,2); G1.addEdge(1,2); G1.addEdge(1,3);
  G1.addEdge(1,4); G1.addEdge(3,0);
  cout<<G1;
}catch(exception& e){ std::cout<<e.what()<<"\n"; }
```

Affichage obtenu :

```
graph G1 (5 vertices and 0 edges)
0:
1:
2:
3:
4:
graph G1 (5 vertices and 6 edges)
0:1 2
1:2 3 4
2:
3:0
4:
```

Exercice 25 - UML - Hiérarchies de classes - Diagramme de classes

Hypercard est une application apparue dans les années 1980 et qui était très en avance sur son époque. En effet, elle contenait déjà les concepts objet ainsi que la navigation hypertexte. Les documents manipulés par Hypercard sont appelés des "**piles**". Une pile est composée de "**cartes**" (à l'image des fiches bristol, ancêtres en papier du concept de fichier). Toute pile comporte une carte particulière appelée la **carte sommaire** qui reste toujours sur le dessus de la pile. L'ordre des autres cartes peut être changé mais la notion d'ordre des cartes est toujours conservée : toute carte a une suivante (sauf évidemment la dernière), et une précédente (sauf évidemment la carte sommaire qui est la première). La carte sommaire est utilisée généralement pour faire une "hyper table des matières" permettant d'accéder directement à des cartes particulières appelées **cartes "tête de chapitre"**.
Les cartes contiennent des "**widgets**" (window objects) qui peuvent être des **boutons**, des **champs texte** (pouvant servir pour saisir une entrée ou afficher un résultat) et des **graphiques bitmap** ou **vectoriel**. Un widget qui doit figurer dans toutes les cartes est placé sur un "**fond de carte**" (équivalent à ce que l'on appelle "masque de diapositive" dans l'application Powerpoint).
Les widgets possèdent un script (pouvant faire des tâches relativement complexes comme des calculs) écrit dans le langage Hypertalk (petit langage objet). Les méthodes du script sont exécutées lorsque l'objet widget reçoit un message. Par exemple, le messages `onClick()` est envoyé à l'objet lorsqu'un utilisateur clique sur l'objet. D'autres messages provenant de l'utilisateur sont par exemple envoyés à un champ texte lorsqu'une donnée y est saisie, etc.
On ne s'intéressera dans cet exercice qu'aux boutons qui permettent la navigation hypertexte : lorsque l'on clique sur un de ces boutons, le script exécuté permet de naviguer jusqu'à une autre carte. Ces boutons particuliers sont appelés **"boutons lien"**. Il existe trois boutons liens particuliers (prédéfinis dans une bibliothèque de Hypercard) : les boutons "précédent", "suivant" et "sommaire" qui permettent respectivement la navigation vers la carte précédente, la carte suivante suivante et la carte sommaire (voir Figure). Il existe aussi un autre type de bouton lien appelé "bouton chapitre". Ces boutons sont généralement utilisés dans la carte sommaire pour offrir un accès rapide aux cartes tête de chapitre.



Dans l'exercice, on fera les hypothèses suivantes :

- L'affichage d'une carte est réalisé en lui envoyant un message `afficher()`. La carte s'adresse alors à tous ses widgets en leur envoyant un message `afficher()`.
- Tout bouton possède un attribut nommé `intitulé` qui contient une chaine de caractères (éventuellement vide). Les boutons "précédent", "suivant" et "sommaire" sont identifiés par cet attribut contenant ces mêmes chaînes de caractères. D'autres attributs peuvent être ajoutés : en particulier les boutons chapitres possèdent un attribut supplémentaire appelé `numéro` donnant le numéro du chapitre.
- La responsabilité de la navigation est située au niveau des cartes. Chaque carte connait sa carte précédente et sa carte suivante et la carte sommaire connait toutes les cartes tête de chapitre. Les boutons sont de simples objets réactifs dont la seule fonction est d'informer la carte à laquelle ils appartiennent qu'ils viennent d'être cliqués. De plus la carte ne connaît alors que l'identité du bouton, et doit se débrouiller pour récupérer les attributs dont elle a besoin.

Construire un modèle UML cohérent de l'univers décrit dans le texte précédent.
Ce modèle sera composé d'un diagramme de classes.

Exercice 26 - Les évènements - Héritage

Afin de pouvoir élaborer un agenda, on désire implémenter un ensemble de classes permettant de gérer des évènements de différents types (anniversaires, rendez-vous, fêtes, jours fériés, etc). Un évènement se passe à une date précise. On identifie un évènement avec un sujet (une description). Certains évènements sont aussi caractérisés par un horaire et une durée. Parmi ces évènements, on distingue les rendez vous avec une ou plusieurs personnes qui ont lieu à un endroit déterminé.
Toutes les classes suivantes seront définies dans l'espace de nom `TIME`. On dispose aussi de classes simples : `Date`, `Duree`, `Horaire` fournies avec le sujet dans les fichiers `timing.h` et `timing.cpp`.
Le polymorphisme étant mis en œuvre, on utilisera la dérivation publique. On définira les constructeurs, destructeurs et accesseurs dans toutes les classes implémentées dans la suite. On fera attention à la gestion des espaces **private**, **protected** et **public** des classes.

Préparation : Créer un projet vide et ajouter trois fichiers `evenement.h`, `evenement.cpp` et `main.cpp`. Définir la fonction principale `main` dans le fichier `main.cpp`.
On suppose qu'un évènement simple qui a lieu un jour est décrit par une date et un sujet. On a donc définit la classe `Evt1j` suivante (à ajouter dans le fichier `evenement.h`) :

```
#if !defined(_EVENEMENT_H)
#define _EVENEMENT_H
#include <iostream>
#include <string>
#include "timing.h"

namespace TIME{
 class Evt1j {
 private:
  Date date;
  std::string sujet;
 public:
  Evt1j(const Date& d, const std::string& s):date(d),sujet(s){}
  const std::string& getDescription() const { return sujet; }
  const Date& getDate() const { return date; }
  void afficher(std::ostream& f= std::cout) const {
   f<<"***** Evt ********"<<"\n"<<"Date="<<date<<" sujet="<<sujet<<"\n";
  }
 };
}
#endif
```

evenement.h

Les instructions suivantes (à mettre dans le fichier `main.cpp`) permettent de construire des objets `Evt1j` :

```
#include <iostream>
#include "evenement.h"
int main(){
 using namespace std;
 using namespace TIME;
 Evt1j e1(Date(4,10,1957),"Spoutnik");
 Evt1j e2(Date(11,6,2013),"Shenzhou");
 e1.afficher();
 e2.afficher();
 return 0;
}
```

main.cpp

S'assurer que le projet compile correctement. Dans cet exercice, on tâchera de mener une approche "compilation séparée". Au fur et à mesure de l'exercice, on pourra compléter la fonction principale en utilisant les éléments créés.

Question 1 - Hiérarchie de classes

Après avoir lu les questions 2 et 3, dessiner un modèle UML représentant la hiérarchie des classes mises en œuvre.

Question 2 - Héritage - Spécialisation

On désire aussi gérer des évènements liés à un jour mais qui comporte aussi un horaire de début et une durée. Un objet de la classe `Evt1jDur` doit permettre de réprésenter de tels évènements.

1. Implémenter la classe `Evt1jDur` qui hérite de la classe `Evt1j`.
2. Ajouter les accesseurs manquants et redéfinir la méthode `afficher`. **Exemple :**

```
#include <iostream>
#include "evenement.h"
int main(){
 using namespace std;
 using namespace TIME;
 Evt1j e1(Date(4,10,1957),"Spoutnik");
 Evt1j e2(Date(11,6,2013),"Shenzhou");
 Evt1jDur e3(Date(11,6,2013),"Lancement de Longue Marche",Horaire(17,38),Duree
    (0,10));
 e1.afficher();
 e2.afficher();
 e3.afficher();
 system("pause");
 return 0;
}
```

main.cpp

Question 3 - Héritage - Spécialisation

On désire aussi gérer des évènements représentant des rendez-vous. Un objet de la classe `Rdv` est un objet `Evt1jDur` avec un lieu et une ou plusieurs personnes.

1. Implémenter la classe `Rdv` (rendez-vous). On utilisera la classe `string` pour ces deux attributs (un seul objet `string` pour toutes les personnes).
2. Ajouter les accesseurs manquants et redéfinir la méthode `afficher`.

Question 4 - Héritage - Construction et destruction

Ajouter un affichage sur le flux `cout` dans les constructeurs des classes `Evt1j`, `Evt1jDur` et `Rdv` en écrivant un message du type "construction d'un objet de la classe X". Définir un destructeur dans chacune de ces classes en y ajoutant un affichage sur le flux `cout` en écrivant un message du type "destruction d'un objet de la classe X". Enfin, exécuter les instructions suivantes :

```
{ // début de bloc
  Rdv e(Date(11,11,2013),"reunion UV",Horaire(17,30),Duree(60),"Intervenants UV","
     bureau");
  std::cout<<"RDV:";
  e.afficher();
} // fin de bloc
```

En déduire la façon dont les différentes parties d'un objet sont construites et détruites.

Exercice 27 - Redéfinition de la duplication par défaut *-Exercice optionnel d'approfondissement-*

Redéfinir le constructeur de recopie et l'opérateur d'affectation de la classe `Rdv` (Voir Exercice 26).

Exercice 28 - Les évènements - Polymorphisme

Question 1 - Polymorphisme

Exécuter les instructions suivantes :

```
Evt1j e1(Date(4,10,1957),"Spoutnik");
Evt1j e2(Date(11,6,2013),"Shenzhou");
Evt1jDur e3(Date(11,6,2013),"Lancement de Longue Marche",Horaire(17,38),Duree
    (0,10));
Rdv e4(Date(11,4,2013),"reunion UV",Horaire(17,30),Duree(60),"Intervenants UV","
   bureau");
e1.afficher(); e2.afficher(); e3.afficher(); e4.afficher();
Evt1j* pt1= &e1; Evt1j* pt2=&e2; Evt1j* pt3=&e3; Evt1j* pt4=&e4;
pt1->afficher(); pt2->afficher(); pt3->afficher(); pt4->afficher();
```

1. Qu'observez vous ? Assurez-vous que le polymorphisme est bien mis en œuvre ou faire en sorte qu'il le soit...
2. Surcharger (une ou plusieurs fois) l'opérateur **operator**<< afin qu'il puisse être utilisé avec un objet `std ::ostream` et n'importe quel évènement.

Question 2 - Polymorphisme et destruction

Ajouter un affichage sur le flux `cout` dans les constructeurs des classes `Evt1j`, `Evt1jDur` et `Rdv` en écrivant un message du type "construction d'un objet de la classe X". Définir un destructeur dans chacune de ces classes en y ajoutant un affichage sur le flux `cout` en écrivant un message du type "destruction d'un objet de la classe X". Enfin, exécuter les instructions suivantes :

```
Rdv* pt5= new Rdv(Date(12,11,2013),"reunion UV",Horaire(17,30),Duree(60),"
   Intervenants UV","bureau");
pt5->afficher();
delete pt5;

Evt1j* pt6= new Rdv(Date(12,11,2013),"reunion UV",Horaire(17,30),Duree(60),"
   Intervenants UV","bureau");
pt6->afficher();
delete pt6;
```

Qu'observez vous ? Corriger les problèmes si nécessaire.

Exercice 29 - Polymorphisme et stockage hétérogène

On veut maintenant disposer d'une classe `Agenda` qui permet de stocker des évènements.

1. Implémenter une classe `Agenda` qui pourra permettre de gérer des évènements de tout type (`Evt1j`, `Evt1jDur`, `Rdv`). Pour cela, on utilisera un conteneur standard dans lequel son stockera des pointeurs sur `Evt1j`. Interdire la duplication (par affectation ou par recopie) d'un objet `Agenda`.
2. Définir un opérateur `Agenda& Agenda::operator<<(Evt1j& e)` qui permet d'ajouter un évènement dans un objet `agenda`. Prendre simplement l'adresse de l'évènement passé en argument sans dupliquer l'objet.
3. Quel type d'association y a t-il entre la classe `Agenda` et les classes d'évènements ? Compléter le diagramme de classe de la question 1 avec la classe `Agenda` en conséquence.
4. Définir la fonction **`void`** `afficher(std::ostream& f=std ::cout)` **`const`** qui permet d'afficher tous les évènements d'un objet `Agenda`.

Exercice 30 - Les évènements - Classes abstraites, Généralisation

On suppose maintenant que certains évènements durent plusieurs jours (conférences, festival, fête). On souhaite alors définir une classe `EvtPj` (évènement de plusieurs jours).
Auparavant, on a donc besoin de généraliser le concept lié à la classe `Evt1j` en introduisant une classe `Evt` qui n'est pas contraint par le nombre de jours. Un objet `Evt1j` est alors un objet `Evt` avec une date et un objet `EvtPj` est un objet `Evt` avec une date de début et une date de fin.

1. Implémenter une classe abstraite `Evt` qui comportera la fonction virtuelle pure `afficher()`.
2. Vérifier que la classe `Evt` n'est pas instanciable.
3. Modifier les schémas de dérivation des classes précédentes pour prendre en compte cette nouvelle classe. Remonter l'attribut `sujet` dans la classe `Evt`.
4. Modifier les classes `Evt1j` et `Agenda` et la fonction **`operator`**`<<()` afin de tenir compte de ces changements (un objet `Agenda` doit maintenant contenir des objet `Evt`).
5. Définir la classe `EvtPj` (évènement de plusieurs jours).
6. Modifier le diagramme de classe en prenant en compte toutes ces modifications.

Exercice 31 - Les évènements - Design Patterns

Question 1 - Design pattern Iterator

Implémenter le design pattern Iterator pour la classe `Agenda` afin de pouvoir parcourir séquentiellement les évènements d'un objet `agenda`. L'itérateur implémenté devra être bidirectionnel (il devra être possible de revenir en arrière dans la séquence).

Question 2 - Design pattern Prototype

Faire en sorte maintenant qu'un objet `Agenda` ait la responsabilité de ses évènements en obtenant une duplication dynamique de l'objet passé en argument. Quelle type d'association y a t-il maintenant entre la classe `agenda` et les classes d'évènements ? Compléter et modifier le diagramme de classe en conséquence.

Exercice 32 - Les évènements - Transtypage dynamique

Question 1 - Transtypage - mécanisme

Corriger le code suivant de façon à ce qu'il compile et qu'il s'exécute sans erreur :

```
Evt1j e1(Date(4,10,1957),"Spoutnik");
Evt1j e2(Date(11,6,2013),"Shenzhou");
Evt1jDur e3(Date(11,6,2013),"Lancement de Longue Marche",Horaire(17,38),Duree
    (0,10));
Rdv e4(Date(11,4,2013),"reunion UV",Horaire(17,30),Duree(60),"Intervenants UV","
   bureau");
Evt1j* pt1= &e1; Evt1j* pt2=&e2; Evt1j* pt3=&e3; Evt1j* pt4=&e4;
Evt1j& ref1=e1; Evt1j& ref2=e2; Evt1j& ref3=e3; Evt1j& ref4=e4;

Rdv* pt=pt1; pt->afficher();
pt=pt2; pt->afficher();
pt=pt3; pt->afficher();
pt=pt4; pt->afficher();

Rdv& r1=ref1; r1.afficher();
Rdv& r2=ref2; r2.afficher();
Rdv& r3=ref3; r3.afficher();
Rdv& r4=ref4; r4.afficher();
```

Question 2 - Transtypage - application

Définir un opérateur **operator**<() afin de comparer deux évènements dans le temps. Etudier les conversions (up-casting et down-casting) entre objets qui ont un lien de parenté.

Exercice 33 - Reconnaissance de type à l'exécution *-Exercice d'approfondissement optionnel-*

Définir une méthode agenda::statistiques qui permet de connaître le nom des différents types d'évènement présents dans un agenda ainsi que le nombre d'occurrence d'évènements pour chaque type d'évènement. On supposera que ces types ne sont pas connus à l'avance (mais on pourra supposer qu'il n'existe pas plus de 10 types différents).

Exercice 34 - Les évènements - Design Patterns - II

Question 1 - Design pattern Template Method

Appliquer le design pattern template method en procédant de la manière suivante :

- Déclarer dans la classe `Evt`, la méthode virtuelle pure `string Evt::toString()const` qui renvoie une chaine de caractères décrivant un objet évènement.
- Implémenter cette méthode pour chacune des classes concrètes de la hiérarchie de classe en utilisant la classe standard `stringstream`.
- Rendre la méthode `afficher` concrète dans la classe `Evt` et éliminer les anciennes implémentations de cette méthode dans les classes concrètes.

Question 2 - Design pattern Adapter

Lors du développement d'un nouveau système, nous avons besoin d'un objet qui puisse faire l'historique des différents évènements importants qui peuvent survenir (erreurs, authentifications, écritures/lectures dans un fichier). Soit l'interface suivante (qui sera placé dans le fichier `log.h`) :

```
#if !defined(LOG_H)
#define LOG_H
#include "timing.h"
#include<iostream>
class Log {
public:
 virtual void addEvt(const TIME::Date& d, const TIME::Horaire& h, const std::string
    & s)=0;
 virtual void displayLog(std::ostream& f) const=0;
};
#endif
```

log.h

La méthode `addEvt` doit permettre d'ajouter un nouvel évènement système caractérisé par une date, un horaire et une description. La méthode `displayLog` doit afficher tous les évènement d'un historique sur un flux `ostream` avec un évènement par ligne sous le format : date - horaire : description.

- Développer une classe concrète `MyLog` qui implémente cette interface en réutilisant au mieux les classes développées précédemment. Pour cela, on appliquera le design pattern Adapter. On fera la question une fois en utilisant un **adaptateur de classe** et une fois en utilisant un **adaptateur d'objet**.
- Compléter le fichier `log.h` en ajoutant une classe d'exception `LogError` qui hérite de la classe d'exception `std::exception`. Dans la méthode `MyLog::addEvt`, déclencher une exception de type `LogError` si l'évènement ajouté est antérieur (date/horaire) au dernier évènement de l'historique (indiquant une probable corruption du système).
- Dans le fichier `main.cpp`, ajouter un bloc **`try-catch`** qui englobe des instructions suceptibles de déclencher des exceptions de type `LogError`. Utiliser un gestionnaire de type `std::exception` pour traiter l'exception.

Exercice 35 - Les conteneurs

Dans cet exercice, il s'agit de développer un ensemble de classes qui permettent de stocker des objets de n'importe quel type (tableaux d'objets, liste chaînée d'objets, pile d'objets, etc). Les objets peuvent par exemple être des entiers, des réels, des fractions, des évènements, etc. Le terme générique *conteneur* est utilisé pour désigner les classes qui permettent de contenir d'autres objets. On souhaite que chaque conteneur implémenté ait un mode d'utilisation commun et donc une interface commune obligatoire à tous les conteneurs.
On appelle *taille d'un conteneur* le nombre d'objets qu'il contient. Un conteneur est vide lorsqu'il ne contient pas d'objet. On considère que les objets sont indicés à partir de 0. Le premier objet d'un conteneur sera donc le 0^e objet du conteneur.
Dans la suite, on appelle `T`, le type des objets contenus dans les conteneurs. L'interface commune à chaque conteneur est la suivante :

- `size_t size()const;` qui renvoie la taille du conteneur.
- **bool** `empty()const;` qui renvoie vrai si le conteneur est vide et faux sinon.
- `T& element(size_t i);` qui renvoie une référence sur le i^e élément d'un conteneur.
- **const** `T& element(size_t i)const;` qui renvoie une référence **const** le i^e élément d'un conteneur.
- `T& front();` qui renvoie une référence **const** sur le premier objet contenu dans le conteneur.
- **const** `T& front()const;` qui renvoie une référence **const** sur le premier objet contenu dans le conteneur.
- `T& back();` qui renvoie une référence sur le dernier objet contenu dans le conteneur.
- **const** `T& back()const;` qui renvoie une référence **const** sur le dernier objet contenu dans le conteneur.
- **void** `push_back(const T& x);` qui ajoute un objet `x` au conteneur après le dernier objet.
- **void** `pop_back();` qui retire le dernier objet du conteneur.
- **void** `clear();` qui retire tous les objets du conteneur.

Préparation : Créer un projet vide et ajouter deux fichiers `container.h` et `main.cpp`. Les situations exceptionnelles seront gérées en utilisant la classe d'exception suivante (à recopier dans le fichier `container.h`) :

```
#if !defined(_Container_T_H)
#define _Container_T_H
#include<string>
#include<stdexcept>

namespace TD {
class ContainerException : public std::exception {
protected :
 std::string info;
public:
 ContainerException(const std::string& i="") noexcept :info(i){}
 const char* what() const noexcept { return info.c_str(); }
 ~ContainerException()noexcept{}
};
}
#endif
```

container.h

Définir la fonction principale `main` dans le fichier `main.cpp`. S'assurer que le projet compile correctement.
Dans la suite, vous déclarerez et définirez chaque constructeur, chaque destructeur, chaque attribut et chaque méthode (de l'interface obligatoire) partout où cela est nécessaire. Vous définirez aussi les constructeurs de recopie et les opérateurs d'affectation nécessaires.

Question 1

Analyser et modéliser les deux classes du problème dans un diagramme de classe. Implémenter la classe abstraite `Container` modèle de tous les autres conteneurs en exploitant au mieux le design pattern *"template method"* pour utiliser le moins de méthodes virtuelles pures possible.

Question 2

Implémenter une classe `Vector` qui sera basée sur le modèle `Container`. Cette classe utilisera un attribut de type `T*` qui pointera sur un tableau de `T` alloué dynamiquement pour composer ses éléments. Pour cela, on suppose que le type `T` dispose d'un constructeur sans argument.
Le constructeur "principal" prendra en argument la taille initiale du tableau et la valeur avec laquelle les objets initialement présents dans le tableau doivent être initialisés. On étudiera éventuellement la possibilité d'avoir des

arguments par défaut. Surcharger en plus l'opérateur **operator**`[]` qui permettra de *modifier* ou de *lire* la valeur d'un élément particulier du tableau.

Exercice 36 - Conteneurs - Design pattern Adaptateur et Stratégie

En utilisant astucieusement le design pattern *"adapter"*, implémenter une classe `Stack` qui ne devra avoir comme seule interface possible que les méthodes suivantes :

- **bool** `empty()` **const;**
- **void** `push(` **const** `T& x);` qui empile un objet dans la pile.
- **void** `pop();` qui dépile le dernier élément empilé de la pile.
- `size_t size()` **const;**
- `T& top();` qui renvoie une référence sur le dernier objet empilé de la pile
- **const** `T& top()` **const;**
- **void** `clear();`

On réfléchira à la possibilité de pouvoir "adapter" n'importe quel conteneur pour implémenter cette classe (design pattern *"stratégie"*). On fera cet exercice deux fois : une fois en utilisant un *adaptateur de classe*, une fois en utilisant un *adaptateur d'objet*. On dessinera les diagrammes de classe correspondants.

Exercice 37 - Conteneurs - Design pattern Iterator et algorithmes

Question 1

Implémenter le design pattern "itérateur" en créant le type `iterator` pour les classes `Vector` et `Stack` :

- Pour accéder à l'élément désigné par un itérateur, on utilisera l'opérateur **`operator`**$\star$.
- Pour qu'un itérateur désigne l'élément suivant, on lui appliquera l'opérateur **`operator`**++.
- Afin de comparer deux itérateurs, on surchargera les opérateurs **`operator`**== et **`operator`**!= : on suppose que deux itérateurs sont égaux s'ils désignent le même élément.
- Pour les classes `Vector` et `Stack`, on implémentera la fonction `begin()` qui renvoie un itérateur désignant le premier élément.
- Pour les classes `Vector` et `Stack`, on implémentera aussi la fonction `end()` qui renvoie un itérateur désignant l'élément (fictif) qui suit le dernier élément, c'est à dire l'itérateur que l'on obtient si on applique l'opérateur ++ sur un itérateur désignant le dernier élément.
- Pour le type `Stack::iterator`, préciser les différentes possibilités d'implémentation.
- Avec un simple copier/coller et quelques modifications, on implémentera aussi un type `const_iterator` ainsi que les méthodes `begin()` et `end()` correspondantes.

Question 2

Implémenter la fonction `minimum_element` qui prend en arguments deux itérateurs `it1` et `it2` de n'importe quel conteneur et qui permet de renvoyer un itérateur désignant l'élément minimum dans le conteneur entre les itérateurs `it1` et `it2` (`it2` non compris), par rapport à l'opérateur **`operator`**<. On supposera pour cela que cet opérateur a été surchargé pour le type d'élément contenu dans le conteneur.

Question 3

Implémenter la fonction `minimum_element` qui prend en arguments deux itérateurs `it1` et `it2` de n'importe quel conteneur, ainsi qu'un prédicat binaire définissant un ordre sur les éléments (design pattern *"Strategy"*). La fonction permet de renvoyer un itérateur désignant l'élément minimum par rapport au prédicat binaire dans le conteneur entre les itérateurs ‘`it1` et `it2` (`it2` non compris). Le prédicat binaire doit renvoyer **`true`** ou **`false`**. Il pourra être soit une fonction prenant en arguments deux objets du type de ceux contenus dans le conteneur, soit un *objet fonction* dont l'opérateur **`operator`**`()` prend en arguments deux objets du type de ceux contenus dans le conteneur, soit une lambda-expression équivalente.

Exercice 38 - Pour aller plus loin avec les conteneurs...

-Exercice d'approfondissement optionnel-

Question 1

Implémenter une classe `List` qui sera une liste doublement chaînée.
Le constructeur "principal" prendra en argument la taille initiale de la liste et la valeur avec laquelle les objets initialement présents dans la liste doivent être initialisés. On étudiera éventuellement la possibilité d'avoir des arguments par défaut.
Définir les fonctions **`void`** `push_front(`**`const`** `T& x)` et **`void`** `pop_front()` qui ajoute ou retire un élément en tête de liste.
Définir la fonction **`bool`** `remove(`**`const`** `T& x)` qui retire le premier élément de la liste qui a la valeur `x`. La fonction renverra **`true`** si l'opération réussit et **`false`** sinon (si `x` n'existe pas dans la liste).

Question 2

On suppose maintenant que le type `T` ne dispose pas forcément d'un constructeur sans argument. Modifier votre classe `Vector` de manière à prendre cet aspect en compte.
Pour cela, on utilisera la classe standard `allocator` du C++ (voir poly) qui permet de séparer l'allocation et la désallocation d'une zone mémoire de la construction et de la destruction d'objets (ou de tableaux d'objets) dynamique.
Le constructeur "principal" prendra en argument la taille initiale du tableau et la valeur avec laquelle les objets initialement présents dans le tableau doivent être initialisés. On étudiera éventuellement la possibilité d'avoir des arguments par défaut.
On appellera *"capacité d'un tableau"* le nombre maximum d'objets qu'il peut contenir avant de devoir refaire une réallocation. Initialement, cette capacité est égale à la taille du tableau. Cependant, on pourra changer la capacité (sans pour autant changer la taille) grâce à la méthode **`void`** `reserve(size_t n)`. Cette méthode n'a une action que si `n` est strictement supérieur à la taille du tableau (dans le cas contraire, la méthode ne fait rien). Le tableau dispose alors d'une réserve supplémentaire qu'il peut utiliser lorsque le nombre d'éléments augmente (par ex avec la méthode `push_back`) sans devoir pour autant faire une réallocation. Implémenter la méthode `size_t capacity()`**`const`** permettant de connaître la capacité du vecteur.
On implémentera aussi la méthode **`void`** `resize(size_t t=0,` **`const`** `T& initialize_with=T());` qui permet de changer le nombre d'éléments du tableau. Si la taille diminue, la capacité ne change pas : les objets qui doivent disparaitre du tableau sont "détruits" mais les cellules qui ne sont plus utilisées sont gardées en réserve (elles ne sont pas désallouées). Si la taille augmente, les nouveaux emplacements utilisées sont initialisées avec la valeur `initialize_with`. Si la nouvelle taille est inférieure à la capacité il n'y a pas de réallocation. Si la nouvelle taille est supérieure à la capacité, un nouveau tableau est réalloué et la capacité devient égale à la nouvelle taille.
De même, la méthode `push_back` ne provoque pas de réallocation tant que la capacité le permet. La méthode `clear` ne provoque pas non plus une désallocation mais seulement une destruction des objets du tableau.

Exercice 39 - Graphes et STL et patrons

Question 1

Après l'exercice 24, on veut développer une classe plus flexible qui permet d'utiliser n'importe quel type pour représenter des sommets. Pour cela, on utilise maintenant un attribut de type `map<Vertex,set<Vertex> >` où `Vertex` est un paramètre de type pour représenter la liste d'adjacence. Un sommet est alors la clé qui permet d'accéder à un ensemble de sommets adjacents.
Implémentez la classe paramétrée dont l'interface est la suivante :

```
#include<map>
#include<set>
#include<string>
#include<iostream>
using namespace std;
template<class Vertex>
class GraphG {
 map<Vertex,set<Vertex> > adj;
 string name;
public:
 GraphG(const string& n);
 const string& getName() const;
 size_t getNbVertices() const;
 size_t getNbEdges() const;
 void addVertex(const Vertex& i);
 void addEdge(const Vertex& i, const Vertex& j);
 void removeEdge(const Vertex& i, const Vertex& j);
 void removeVertex(const Vertex& i);
 void print(ostream& f) const;
};

template<class V> ostream& operator<<(ostream& f, const GraphG<V>& G);
```

Les deux méthodes `addVertex` et `addEdge` permettent d'ajouter des sommets et des arcs librement. L'ajout d'un arc entre un ou des sommets qui n'existent pas encore provoque leur création. La suppression d'un sommet provoque la suppression de tous les arcs liés à ce sommet.

Question 2 - Exercice d'approfondissement optionnel

Créez les types `vertex_iterator` et `successor_iterator` implémentant le design pattern *iterator* :

- Un objet `vertex_iterator` permet de parcourir séquentiellement tous les sommets du graphe.
- Un objet `successor_iterator` permet de parcourir séquentiellement tous les successeurs d'un sommet donné.

S'inspirer de l'exemple suivant pour l'interface de ces types. Pour créer ces types, on utilisera des *adaptateurs de classe* des types `map<Vertex,set<Vertex> >::const_iterator` et `set<Vertex>::const_iterator`.

Question 3 - Exercice d'approfondissement optionnel

Utilisez l'algorithme standard `std::for_each` avec un objet fonction (voir rappels dans le poly) pour implémenter la fonction `operator<<(ostream&, const GraphG<V>&)` (même si c'est inutilement compliqué !).

Exemple :

```
try{
  GraphG<char> G2("G2");
  G2.addVertex('a'); G2.addVertex('b'); G2.addEdge('a','c');
  G2.addEdge('a','d'); G2.addEdge('d','e'); G2.addEdge('e','b');
  cout<<G2;
  cout<<"vertices of G2 are: ";
  for(GraphG<char>::vertex_iterator it=G2.begin_vertex();
      it!=G2.end_vertex(); ++it) cout<<*it<<" ";
  cout<<"\nsuccessors of a: ";
  for(GraphG<char>::successor_iterator it=G2.begin_successor('a');
```

```
      it!=G2.end_successor('a'); ++it){ std::cout<<*it<<" "; }
  GraphG<string> G3("Pref");
  G3.addEdge("LO21","IA01"); G3.addEdge("IA02","IA01"); G3.addEdge("IA01","NF17");
  G3.addEdge("IA02","NF16"); G3.addEdge("NF93","NF16");
  cout<<G3;
}catch(exception e){ std::cout<<e.what()<<"\n"; }
```

Affichage obtenu :

```
graph G2 (5 vertices and 4 edges)
a:c d
b:
c:
d:e
e:b
vertices of G2 are: a b c d e
successors of a: c d
graph Pref (6 vertices and 5 edges)
IA01:NF17
IA02:IA01 NF16
LO21:IA01
NF16:
NF17:
NF93:NF16
```