POO sous C++ - TP5

Objectifs

Les flux d'entreé et de sortie, les patrons de fonctions et de classes et les exceptions

---

-Manipulation Cin Cout

La librairie '**iostream.h**' fournit un certain nombre de mots clés qui permettent de modifier les caractéristiques d'un flux. Ces commandes sont utilisées directement avec l'opérateur `<<` en respectant la syntaxe : '`cout << manipulateur ;`'.

Manipulateur	Rôle
`dec`	Convertir en base décimale
`hex`	Convertir en base hexadécimale
`oct`	Convertir en base octale
`ws`	Supprimer les espaces
`endl`	Ajouter un saut de ligne en fin de flux (\n)
`ends`	Ajouter un caractère de fin de chaîne
`flush`	Vider un flux de sortie
`setbase(int a)`	Choisir la base (0, 8, 10 ou 16). 0 est la valeur par défaut
`setfill(int c)`	Choisir le caractère de remplissage (padding)
`setprecision(int c)`	Indiquer le nombre de chiffres d'un nombre décimal
`setw`	Choisir la taille du champ (padding)

En plus de ces manipulateurs, la classe iostream fournit un ensemble de méthodes.

Méthodes	Rôle
`fill()`	Renvoie le caractère de remplissage
`fill(char c)`	modifie le caractère de remplissage
`précision()`	renvoie la précision pour le nombre décimal
`précision(int n)`	modifie la précision pour le nombre décimal
`setf(long flag)`	modifie une propriété de formatage (format)
`setf(long flag,long champ)`	Modifie une propriété de formatage d'un champ
`width()`	Renvoie la valeur d'affichage
`width(int n)`	Renvoie la valeur d'affichage

Exemple (voire code source Demos/ES/coutin.cpp).

-Saisir au clavier avec 'cin'

Dans l'exemple suivant :

```
char chaine[10];
cout<<"saisir une chaine : ";
cin.width(sizeof(chaine)); cin>>chaine;
```

La fonction '**width**' fixe le nombre de caractères à affecter à la variable '*chaine*' sans débordement.
La classe istream contient la fonction membre **getline** qui permet de saisir des chaînes de caractères comprenant des espaces non acceptés par l'opérateur `>>`.

Le prototype de getline est :

```
istream &getline(char* chaine, int nombre, char fin= '\n');
```

Exemple (voire code source Demos/ES/clsvoiture.cpp).

-Redéfinition les opérateurs de flux : Pour redéfinir les opérateurs de flux (`<< >>`), on doit respecter la syntaxe telle qu'elle est définie dans l'exemple suivant :
Exemple (voire code source Demos/ES/clsvoitureOperator.cpp).

```
class demo
{
     public :
       friend ostream& operator<<(osream&,demo&);
       friend istream& operator>>(isream&,demo&);
};
Ostream & operator<<(osream& …,demo& …)
{ …  }
Istream & operator>>(isream& …,demo& …)
{ …  }
```

Exercice 1
Écrire un programme qui enregistre (sous forme "**binaire**", et non pas formatée), dans un fichier de nom fourni par l'utilisateur, une suite de nombres entiers fournis sur l'entrée standard. On conviendra que l'utilisateur fournira la valeur 0 (qui ne sera pas enregistrée dans le fichier) pour préciser qu'il n'a plus d'entiers à entrer.

Exercice 2
Écrire un programme permettant de lister (sur la sortie standard) les entiers contenus dans un fichier tel que celui créé par l'exercice précédent.

Exercice 3
Écrire un programme permettant à un utilisateur de retrouver, dans un fichier tel que celui créé dans l'exercice 1, les entiers dont il fournit le "rang". On conviendra qu'un rang égal à 0 signifie que l'utilisateur souhaite mettre fin au programme.

Exercice 4
Créer un patron de fonctions permettant de calculer le carré d'une valeur de type quelconque (le résultat possédera le même type). Écrire un petit programme utilisant ce patron.

Exercice 5
Créer un patron de fonctions permettant de calculer la somme d'un tableau d'éléments de type quelconque, le nombre d'éléments du tableau étant fourni en paramètre (on supposera que l'environnement utilisé accepte les "paramètres expression"). Écrire un petit programme utilisant ce patron.

Exercice 6
a. Créer un patron de classes nommé **pointcol**, tel que chaque classe instanciée permette de manipuler des points colorés (deux coordonnées et une couleur) pour lesquels on puisse "choisir" à la fois le type des coordonnées et celui de la couleur. On se limitera à deux fonctions membre : un constructeur possédant trois arguments (sans valeur par défaut) et une fonction affiche affichant les coordonnées et la couleur d'un "point coloré".
b. Dans quelles conditions peut-on instancier une classe patron **pointcol** pour des paramètres de type classe ?

Exercice 7
On a défini le patron de classes suivant :

```
template <class T> class point
{
T x, y ; // coordonnees
public :
point (T abs, T ord) { x = abs ; y = ord ; }
void affiche () ;
} ;
template <class T> void point<T>::affiche ()
{
cout << "Coordonnees : " << x << " " << y << "endl" ;
}
```

a. Que se passe-t-il avec ces instructions :

```
point <char> p (60, 65) ;
p.affiche () ;
```

b. Comment faut-il modifier la définition de notre patron pour que les instructions précédentes affichent bien : `Coordonnees : 60 65`

Exercice 8
Créer un patron de classes permettant de représenter des "vecteurs dynamiques" c'est-à- dire des vecteurs dont la dimension peut ne pas être connue lors de la compilation (ce n'est donc pas obligatoirement une expression constante comme dans le cas de tableaux usuels). On prévoira que les éléments de ces vecteurs puissent être de type quelconque. On surdéfinira convenablement l'opérateur [] pour qu'il permette l'accès aux éléments du vecteur (aussi bien en consultation qu'en modification) et on s'arrangera pour qu'il n'existe aucun risque de "débordement d'indice". En revanche, on ne cherchera pas à régler les problèmes posés éventuellement par l'affectation ou la transmission par valeur d'objets du type concerné.

N.B. Il ne faut pas chercher à utiliser les composants standard introduits par la norme. En effet, le patron vector répondrait intégralement à la question.

Exercice 9
Comme dans l'exercice précédent, réaliser un patron de classes permettant de manipuler des vecteurs dont les éléments sont de type quelconque mais pour lesquels la dimension, supposée être cette fois une expression constante, apparaîtra comme un paramètre (expression) du patron. Hormis cette différence, les "fonctionnalités" du patron resteront les mêmes.

Exercice 10
Quels résultats produira ce programme ?

```
#include <iostream>
using namespace std ;
class erreur{
public :
int num ;
};
class erreur_d : public erreur{
public :
int code ;
};
class A{
public :
A(int n){
if (n==1) {
erreur_d erd ; erd.num = 999 ;
erd.code = 12 ; throw erd ; }
}
} ;
main(){
try{
A a(1) ;
cout << "apres creation a(1)" << endl;
}
catch (erreur er){
cout << "exception erreur : " << er.num << endl ;
}
catch (erreur_d erd){
cout << "exception erreur_d : " << erd.num << " " << erd.code << "\n" ;
}
cout << "suite " <<endl;
try{
A b(1) ;
cout << "apres creation b(1)" <<endl ;
}
catch (erreur_d erd){
cout << "exception erreur_d : " << erd.num << " " << erd.code << endl ;
}
catch (erreur er){
cout << "exception erreur : " << er.num << endl ;
}
}
```

Exercice 11
Écrire une classe point (à deux coordonnées entières) qui dispose d'un constructeur à deux arguments levant une exception lorsque les deux composantes sont égales. De plus, l'appel d'un constructeur sans argument ou à un seul argument devra également lever un autre type d'exception. Ici, on limitera les fonctions membre de point aux seuls constructeurs. Écrire un programme d'utilisation de la classe point, interceptant convenablement les exceptions prévues, en mentionnant la cause.

Exercice 12

1. Quels résultats fournira ce programme :

```
#include <iostream>
using namespace std ;
main(){
void f() ;
try{
f() ;
}
catch (int n){
cout << "except int dans main : " << n << endl;
}
catch (...){
cout << "exception autre que int dans main " << endl;
}
cout << "fin main " << endl;
}
void f(){
try{
int n=1 ; throw n ;
}
catch (int n)
{
cout << "except int dans f : " << n << endl ;
throw ;
}
}
```

2. Même question si l'on remplace la fonction **f** par :

```
void f(){
try{
float x=2.5 ; throw x ;
}
catch (int n){
cout << "except int dans f : " << n << endl ;
throw ;
}
}
```