

TD 3 : Pratiquer les STL et les concepts avancés en C++.

Exercice 1

Effectuer les opérations arithmétiques sur des données complexes à l'aide d'une classe et d'un objet cette fois il faut **redéfinir** les opérateurs. Le programme doit demander la partie réelle et imaginaire de deux nombres complexes et afficher les parties réelle et imaginaire de l'opération demandée.

Exercice 2

Écrire un programme qui fait entrer des entiers à l'utilisateur, qui les met au fur et à mesure dans une liste (conteneur List de la STL), qui trie cette liste puis qui affiche la liste triée. Ecrire **trois méthodes** pour trier cette liste selon les algorithmes : *tri à bulle*, *tri par sélection* et *tri par insertion*.

Exercice 3

Écrire un programme qui fait entrer des personnes « nom, prénom, âge », qui les met au fur et à mesure dans une liste (conteneur List de la STL), qui trie cette liste selon l'ordre alphabétique nom et prénom puis qui affiche la liste triée.

Exercice 4

1. Créer un set paramètre pour le type int que vous remplirez avec tous les entiers de 1 à 100.
2. Écrire une fonction de recherche qui renvoie un booléen pour indiquer si une valeur donnée est présente ou non dans un set que vous passerez en paramètre avec la valeur à rechercher.
3. Modifiez cette fonction en lui passant maintenant en paramètres 2 littérateurs de set<int>(début et fin) ainsi que la valeur à rechercher.
4. Passer la fonction en Template de façon à ce qu'elle marche peu importe le type d'itérateur qui lui est fourni (itérateur de liste, de vecteur, d'ensemble, ...).

Donner un exemple d'appel sur : un vecteur de string, une liste d'entiers, un tableau 'classique' de float.

Exercice 5

Complétez le programme suivant pour que les erreurs susceptibles de se produire soient gérées jusqu'à ce qu'un calcul soit effectivement mené à bout, le cas de déclenchement d'une exception il faut arrêter le programme.

```
#include <iostream>
using namespace std;
class Test{
public:
    static int tableau[] ;
public :
    static int division(int indice, int diviseur){
        return tableau[indice]/diviseur;
    }
};
int Test::tableau[] = {17, 12, 15, 38, 29, 157, 89, -22, 0, 5} ;
int main() {
    int x, y;
    cout << "Entrez l'indice de l'entier à diviser: " << endl;
    cin >> x ;
    cout << "Entrez le diviseur: " << endl;
    cin >> y ;
    cout << "Le résultat de la division est: "<< endl;
    cout <<Test::division(x,y) << endl;
    return 0; }
```

Exercice 6

Soit une classe **vect** permettant de manipuler des « vecteurs dynamiques » d'entiers, dont la déclaration (fournie dans un fichier vect.h) se présente ainsi (notez la présence de membres protégés) :

```
class vect
{ protected : // en prévision d'une éventuelle classe dérivée
  int nelem ; // nombre d'éléments
  int * adr ; // adresse zone dynamique contenant les éléments
public :
  vect (int) ; // constructeur
  ~vect () ; // destructeur
  int & operator [] (int) ; // accès à un élément par son "indice"
} ;
```

On suppose que le constructeur alloue effectivement l'emplacement nécessaire et que l'opérateur [] peut être utilisé indifféremment dans une expression ou à gauche d'une affectation. En revanche, comme on peut le voir, cette classe n'a pas prévu de constructeur par recopie et elle n'a pas surdéfini l'opérateur d'affectation.

L'affectation et la transmission par valeur d'objets de type **vect** posent donc les « *problèmes habituels* ». Créer une classe **vectok**, dérivée de **vect**, telle que l'affectation et la transmission par valeur d'objets de type **vectok** s'y déroulent convenablement. Pour faciliter l'utilisation de cette nouvelle classe, introduire une fonction membre *taille* fournissant la dimension d'un vecteur.

Écrire un petit programme d'essai.

Exercice 7

Quels seront les résultats fournis par ce programme :

```
#include <iostream>
using namespace std ;
class A
{ int n ;
  float x ;
public :
  A (int p = 2)
  { n = p ; x = 1 ;
    cout << "*** construction objet A : " << n << " " << x << "\n" ;
  }
} ;
class B
{ int n ;
  float y ;
public :
  B (float v = 0.0)
  { n = 1 ; y = v ;
    cout << "*** construction objet B : " << n << " " << y << "\n" ;
  }
} ;
class C : public B, public A
{ int n ;
  int p ;
public :
  C (int n1=1, int n2=2, int n3=3, float v=0.0) : A (n1), B(v)
  { n = n3 ; p = n1+n2 ;
    cout << "*** construction objet C : " << n << " " << p << "\n" ;
  }
} ;
main()
{ C c1 ;
  C c2 (10, 11, 12, 5.0) ;
}
```

Exercice 8

Créer une Template de fonction permettant de calculer le carré d'une valeur de type quelconque (le résultat possèdera le même type). Écrire un petit programme utilisant cette Template.

Exercice 9

On a défini la Template 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 << "\n" ;
}
```

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 10

Soit la définition suivante des classes erreur et A :

```
class erreur
{ public :
int num ;
} ;

class A
{ public :
A(int n)
{ if (n==1) { erreur er ; er.num = 999 ; throw er ; }
}
} ;
```

Quels résultats fournira ce programme utilisant ces deux classes :

```
#include <iostream>
using namespace std ;
main()
{ void f() ;
try
{ f() ;
}
catch (erreur er)
{ cout << "dans main : " << er.num << "\n" ;
}
cout << "suite main\n" ;
} void f()
{ try
{ A a(1) ;
}
catch (erreur er)
{ cout << "dans f : " << er.num << "\n" ;
}
```