

GL / C++

Chapitre 10

Compléments C++

- Espace de noms
- Spécification **"inline"**
- Exceptions

Espace de noms : à quoi ça sert ?

- Dans le cas d'un gros programme ou d'un programme utilisant de nombreuses bibliothèques, on peut avoir un problème de "**pollution de l'espace des noms**" : un même identificateur a été utilisé plusieurs fois
- Pour résoudre ce problème, la norme **Ansi** du C++ a défini le concept "d'espace de nom" qui consiste à nommer un espace de déclarations

Espace de noms : utilisation

- Pour déclarer un espace de nom :

```
namespace monEspaceDeNoms {  
    // déclarations de variables, classes, fonctions, ...  
    // qui appartiennent à l'espace "monEspaceDeNoms"  
}
```

- Pour utiliser un espace de nom :

```
using namespace monEspaceDeNoms;  
// à partir d'ici, les identificateurs définis dans  
// l'espace "monEspaceDeNoms" sont disponibles
```

- Pour lever une ambiguïté si plusieurs espaces de noms comportent les mêmes identificateurs :

```
Espace1::Point p1;  
// fait référence à la classe Point définie dans Espace1  
Espace2::Point p2;  
// fait référence à la classe Point définie dans Espace2
```

Espace de noms : utilisation

- Si un même identificateur appartient à plusieurs espaces, on peut choisir d'en utiliser un par défaut, sans préfixer, par :

```
using Espace1::Point;  
// à partir d'ici, Point fait référence  
// à la classe Point définie dans Espace1
```

- Tous les identificateurs des bibliothèques standard sont définis dans l'espace de noms "std". D'où la présence, en haut de la plupart des programmes, après les directives **include**, de : **using namespace** std;

Spécification **inline** - Rappels sur les macro

- En C, on dispose de la notion de macro qui est une directive destinée au préprocesseur du compilateur
- Une macro ressemble à une fonction dans sa forme, mais se comporte différemment : chaque fois qu'on utilise une macro, le préprocesseur remplace l'appel à la macro par le "code source" de la macro.
- Il n'y a donc pas d'appel et de passage de paramètres comme dans le cas d'une fonction normale.
- Il s'agit d'une sorte de rechercher/remplacer comme dans un traitement de texte...
- Intérêt : plus rapide lors de l'exécution

Spécification **inline** - Rappels sur les macro

■ Problème : il peut y avoir des "effets de bord"

```
#define carre(x)  x * x
a=2;
b=carre(a);
// transformé par le préprocesseur en b = a * a ; donc a = 4
b=carre(a++);
// transformé en b = a++ * a++ ; donc a = 2*3 = 6 et non 4 !
```

■ Les fonctions en ligne (**inline**) de C++ permettent de pallier ce problème !

```
inline int carre(int x) {
    return x*x;
}
a=2;
b=carre(a++); // correct : b=4, a=3
// une fonction inline s'utilise comme une fonction normale
```

Spécification **inline**

- Le mot clé "**inline**" peut être utilisé devant une fonction/méthode
- Sa présence indique au compilateur qu'à chaque appel à la fonction/méthode, celui-ci devra incorporer, à l'endroit de l'appel, les instructions (en assembleur) correspondant à la fonction/méthode.
- Intérêt : gain de temps à l'exécution, plus d'effets de bord comme avec les macro
- Inconvénient : on ne peut pas compiler séparément les fonctions "**inline**". Il faut donc les implémenter dans les fichier en-tête (.h) des classes/modules où on veut les utiliser.

Exemple : Entier.h

```
class Entier {  
    int x;  
    public :  
        Entier(); // implémenté dans Entier.cc  
        inline int getX() {return x;} // implémenté dans Entier.h  
        ...  
};
```

Exceptions

Exceptions

- Pour gérer « proprement » les conditions exceptionnelles
- Permettent de distinguer la détection de l'incident et son traitement
- Indispensables pour développer des composants réutilisables destinés à être exploités dans différents programmes

Principe

- Une exception est une rupture de séquence déclenchée par une instruction **throw** qui comporte un paramètre de type *donnée*
- Il y a alors branchement à un bloc d'instructions appelé **gestionnaire d'exception**.
- Le gestionnaire appelé est déterminé par la valeur (le paramètre) de l'exception levée par **throw**

Exemple

```
class Vect
{ int nbElem ;
  int * adr ;
public :
  Vect (int) ;
  ~Vect () ;
  int & operator [] (int) ;
} ;

/* spécif et implémentation d'une classe VectLimite (vide pour l'instant) */
class VectLimite
{ } ;

/* implémentation de la classe vect */
Vect::Vect (int n)
{ adr = new int [nbElem = n] ; }

Vect::~Vect ()
{ delete [] adr ; }

int & Vect::operator [] (int i)
{ if (i<0 || i>nbElem)
  { VectLimite l ; throw (l) ;
  }
  return adr [i] ;
}
```

Gestionnaire d'exception

- Inclure dans un bloc (bloc **try**) toutes les instructions qui pourraient lever des exceptions que l'on souhaite détecter
- Faire suivre ce bloc de la définition de tous les gestionnaires d'exception (blocs **catch**) que l'on souhaite implémenter
- En « paramètre formel » de chaque bloc catch, on précise la ou les exceptions gérées par ce gestionnaire

Exemple

```
int main ()
{ try
  { Vect v(10) ;
    v[11] = 5 ;    /* indice trop grand */
  }
  catch (VectLimite vl) /* nom d'argument (vl) superflu ici */
  { cout << "exception limite \n" ;
    exit (-1) ;
  }
  return 0;
}
```

Propriétés

- On peut transmettre des informations, lors de la levée d'une exception, au gestionnaire qui va traiter cette exception, grâce au paramètre transmis
- On peut définir des hiérarchies d'exceptions en lançant différentes exceptions dont les classes dérivent d'une même classe commune.
- Plusieurs types d'exceptions qui dérivent d'une même classe peuvent alors être traités par un seul gestionnaire associé à la « classe mère »

Exemple 2

```
/* spécif de la classe Vect */
class Vect
{ int nbElem ;
  int * adr ;
public :
  Vect (int) ;
  ~Vect () ;
  int & operator [] (int) ;
} ;

/* définition des deux classes exception */
class VectLimite
{ public :
  int hors ; // valeur indice hors limites (public)
  VectLimite (int i) // constructeur
    { hors = i ; }
} ;

class VectCreation
{ public :
  int nb ; // nombre d'éléments demandés (public)
  VectCreation (int i) // constructeur
    { nb = i ; }
} ;
```

Exemple 2 - suite

```
/* définition de Vect */
Vect::Vect (int n)
{ if (n <= 0)
    { VectCreation c(n) ;           // déclaration anomalie
      throw c ;                     // levée exception
    }
  adr = new int [nbElem = n] ; // construction normale
}

Vect::~~Vect ()
{ delete [] adr ;
}

int & Vect::operator [] (int i)
{ if (i<0 || i>nbElem)
    { VectLimite l(i) ;           // déclaration anomalie
      throw l ;                   // levée exception
    }
  return adr [i] ;               // fonctionnement normal
}
```

Exemple 2 - fin

```
/* test exception */
int main ()
{
    try
    { Vect v(-3) ;           // provoque l'exception VectCreation
      v[11] = 5 ;           // provoquerait l'exception VectLimite
    }
    catch (VectLimite vl)
    { cout << "exception indice " << vl.hors << " hors limites \n" ;
      exit (-1) ;
    }
    catch (VectCreation vc)
    { cout << "exception creation vect nb elem = " << vc.nb << "\n" ;
      exit (-1) ;
    }
    return 0;
}
```

Choix du Gestionnaire & « appel »

- Lorsqu'une exception est levée, C++ applique un ensemble de règles précis pour choisir le bon gestionnaire à appeler
- Une fois le choix fait, on réalise **une copie** de l'expression mentionnée dans le **throw** pour la passer au gestionnaire choisi
- Cette copie est nécessaire puisque les variables automatiques déclarées dans le bloc **try** disparaissent lorsque l'exception est levée

Règles de choix du gestionnaire

Lors d'un `throw(new T(...))` :

1. Choix d'un gestionnaire ayant le **type exact**
`catch(T e)`, `catch(T & e)`, `catch(const T e)`, `catch(const T & e)`
2. Choix d'un gestionnaire correspondant à la **classe de base de T** (classe dont dérive la classe T)
3. Choix d'un gestionnaire correspondant à un **pointeur sur une classe dérivée de la classe T**
4. Choix d'un gestionnaire correspondant à un **type quelconque**, noté par des points de suspension :
`catch(...)`

Exception utilisateur dérivant de la classe "standard" exception

```
#include <exception>
using namespace std;
```

```
class MonException1 : public exception
{ public :
    MonException1 () {}
    const char * what() const throw() { return "mon exception 1" ; }
} ;
```

```
class MonException2 : public exception
{ public :
    MonException2 () {}
    const char * what() const throw() { return "mon exception 2" ; }
} ;
```

Exception utilisateur dérivant de la classe "standard"

exception – suite & fin

```
#include <iostream>
using namespace std;
int main() {
    try {
        cout << "bloc try 1\n" ;
        throw MonException1() ;
    }
    catch (exception & e) {
        cout << "exception : " << e.what() << "\n" ;
    }
    try {
        cout << "bloc try 2\n" ;
        throw MonException2() ;
    }
    catch (exception & e) {
        cout << "exception : " << e.what() << "\n" ;
    }
    return 0;
}
```

Propagation d'une exception

```
void f()  
{ try  
  { int n=2 ;  
    throw n ;           // lève une exception de type int  
  }  
  catch (int)  
  { cout << "exception int traitée dans f \n" ;  
    throw ; // throw, sans argument, propage de l'exception  
  }  
}  
  
int main()  
{ try  
  {  
    f() ;  
  }  
  catch (int)  
  { cout << "exception int traitée dans main \n" ;  
    exit(-1) ;  
  }  
  return 0;  
}
```

Exercice 1

- Quels résultats donne le programme suivant si on lui fournit comme donnée 1, 2, 3 ou 4 ?

```
int main() {
    int n ; float x ; double z ;
    cout << "donnez un entier : " ; cin >> n ;
    try
    { switch (n) {
        case 1 : throw n ;          break ;
        case 2 : x = n ; throw x ;  break ;
        case 3 : z = n ; throw z ;  break ;
    }
    }
    catch (int n) { cout << "catch entier - n = " << n << "\n" ; }
    catch (float x) { cout << "catch flottant - x = " << x << "\n" ;
        exit (-1) ;
    }
    cout << "suite et fin du programme\n" ;
    return 0 ;
}
```

Exercice 2

- Ecrire une classe Point à deux coordonnées entières qui comporte un constructeur à 2 arguments levant une exception lorsque les deux coordonnées sont égales. De plus, l'appel d'un constructeur sans arguments ou à un seul argument devra lever un autre type exception.
- Ecrire un programme principal qui utilise Point et qui intercepte convenablement les exceptions prévues avec un message explicite