Cours C++ Avancé

Anis Sahbani
anis.sahbani@upmc.fr

Université Pierre et Marie Curie - Paris 6

Introduction

Le principe d'encapsulation consiste à regrouper dans le même objet informatique ("concept"), données et traitements qui lui sont spécifiques :

- Les données inclues dans un objet seront appelées les attributs de cet objet,
- Les traitements/fonctions défini(e)s dans un objet seront appelées les méthodes de cet objet.

Introduction

Un intérêt de l'encapsulation est que cela permet d'abstraire. Elle permet de définir deux niveaux de perception :

- Le niveau externe : partie visible de l'objet :
  - prototype des méthodes et attributs accessibles hors de l'objet
  - c'est l'interface de l'objet avec l'extérieur résultat du processus d'abstraction
- Le niveau interne : (détails d')implémentation de lobjet
  - méthodes et attributs accessibles uniquement depuis l'intérieur de l'objet
  - définition de l'ensemble des méthodes de l'objet : c'est le corps de l'objet

Introduction

En programmation Objet :

- le résultat du processus d'abstraction s'appelle une classe

  classe = catégorie d'objets

- une classe définit un type (au sens du langage de programmation)

- une réalisation particulière d'une classe s'appelle une instance

  instance = objet

Introduction

Tout ce qu'il n'est pas nécessaire de connaître à l'extérieur d'un objet devrait être dans le corps de l'objet et identifié par le mot clé private :

```
class Rectangle {
   double surface() const { ... }
   private :
      double hauteur ;
      double largeur ;
} ;
```

Attribut d'instance privée = inaccessible depuis l'extérieur de la classe. C'est également valable pour les méthodes.

Exemple

```
#include <iostream>
using namespace std ;
// définition de la classe
class Rectangle {
public :
   // définition des méthodes
   double surface() const { return (hauteur * largeur) ; }
   double getHauteur() const { return hauteur ; }
   double getLargeur() const { return largeur ; }
   void setHauteur(double hauteur) { this->hauteur = hauteur ; }
   void setLargeur(double largeur) { this->largeur = largeur ; }

private :
   // déclaration des attributs
      double hauteur ;
      double largeur ;
} ;
```

Exemple

```
//utilisation de la classe
int main() {
    Rectangle rect ;
    double lu ;
    cout << "Quelle hauteur ? " ;
    cin >> lu ;
    rect.setHauteur(lu) ;
    cout << 'Quelle largeur ? " ;
    cin >> lu ;
    rect.setLargeur(lu) ;
    cout << "surface = " << rect.surface() << endl ;
    return 0 ;
}
```

Constructeur

Un constructeur est une méthode :

- invoquée automatiquement lors de la déclaration d'un objet (instanciation d'une classe)
- assurant l'initialisation des attributs.

Exemples :

```
// Le constructeur par défaut
Rectangle() : hauteur(1.0), largeur(2.0) {}
// 2ème constructeur
Rectangle(double c) : hauteur(c), largeur(2.0*c) {}
// 3ème constructeur
Rectangle(double h, double L) : hauteur(h), largeur(L) {}
```

Il est possible de regrouper les 2 premiers constructeurs en utilisant les valeurs par défaut des arguments :

```
Rectangle(double c = 1.0) : hauteur(c), largeur(2.0*c) {}
```

Constructeur de Copie

C++ fournit également un moyen de créer la copie d'une instance : le constructeur de copie

Ce constructeur permet d'initialiser une instance en utilisant les attributs d'une autre instance du même type.

L'invocation du constructeur de copie se fait par une instruction de la forme :

```
Rectangle r1(12.3, 24.5) ;
Rectangle r2(r1) ;
```

r1 et r2 sont deux instances distinctes mais ayant des mêmes valeurs pour leurs attributs.

Destructeur

Si l'initialisation des attributs d'une instance implique la mobilisation de ressources : fichiers, périphériques, portions de mémoire (pointeurs), etc.

→ il est alors important de libérer ces ressources après usage !

La syntaxe de déclaration d'un destructeur pour une classe NomClasse est :

```
~NomClasse() {
   // opérations (de libération)
}
```

Entrées - Sorties

Les interactions les plus simples avec un programme se font via l'écran pour la sortie et le clavier pour les entrées.

En C++, ces flots sont représentés respectivement par cout et cin.

cin et cout sont définis dans le fichier de définitions iostream.

```
#include <iostream>
```

Entrees- Sorties

Exemple :

```
int lu(0) ;
do {
   cout << "entrez un nombre entre 1 et 10 : " << flush ;
   cin >> lu ;
   if (cin.fail()) { // teste si cin "ne va pas bien"
      cout << "Je vous ai demandé d'entrer un nombre ! ! ! " << endl ;
      // remet cin dans un état lisible
      cin.clear() ;
      // jette tout le reste de la ligne
      cin.ignore(numeric_limits <streamsize> : : max(), '\ n') ;
   }
} while ((lu < 1) || (lu > 10)) ;
```

Sortie Erreur Standard

En plus de cin et cout, il existe une sortie d'erreur standard, cerr.

Par défaut cerr est envoyée sur le terminal, comme cout. Mais il s'agit bien d'un flot séparé !

De plus cerr n'a pas de mémoire tampon. L'écriture sur cerr se fait donc directement (on n'a pas besoin de flush).

Les Types "stream"

Pour pouvoir utiliser les flots, il faut tout d'abord inclure les fichiers de définitions correspondant, iostream et fstream :

```
#include <iostream>
#include <fstream>
```

Deux nouveaux types sont alors disponibles :

- ifstream (pour input file stream) qui définit un flot d'entrée (similaire à cin)
- ofstream (pour output file stream) qui définit un flot de sortie (similaire à cout).

Les Types "stream"

Dans le cas des fichiers textes, l'association d'un flot d'entrée-sortie avec le fichier se fait par le biais de la fonction spécifique open.

Exemple :

```
ifstream entree ;
entree.open("test") ;
```

associe le stream entree avec le fichier physique test.

Dans le cas des fichiers binaires, il faut ajouter un argument supplémentaire :

ios : : in|ios : : binary pour la lecture

ios : : out|ios : : binary pour l'écriture

Exemple :

```
ifstream entree ;
entree.open("a_lire.zip", ios : : in|ios : : binary) ;
ofstream sortie ;
sortie.open("a_ecrire.exe", ios : : out|ios : : binary) ;
```

Remarque

Par défaut, un ostream ouvre le fichier en mode écrasement

On souhaite parfois pouvoir ouvrir le fichier en mode ajout ("append")

Cela se fait aussi en ajoutant un argument supplémentaire à open :

```
ofstream sortie ;
sortie.open("a_poursuire.txt", ios : :out|ios : :app) ;
```

Dans le cas de fichiers en binaire :

```
ofstream sortie("a_completer.data", ios : :out|ios : :binary|ios : :app) ;
```

Lecture à partir d'un fichier

```
main() {
   string nom_fichier("test") ;
   ifstream entree(nom_fichier.c_str()) ;
   if (entree.fail())
      cerr << "Erreur : impossible de lire le fichier " << endl ;
   else {
      string mot ;
      while ( !entree.eof()) {
         entree >> mot ;
         cout << mot << endl ;
      }
      entree.close() ;
   }
}
```

Ecriture dans un fichier

```
main() {
   string nom_fichier ;
   cout << "Dans quel fichier voulez vous écrire ? " << flush ;
   cin >> ws ;
   getline(cin, nom_fichier) ;
   ofstream sortie(nom_fichier.c_str()) ;
   if (sortie.fail())
      cerr << "Erreur : impossible d'écrire dans le fichier " << endl ;
   else {
      string phrase ;
      cout << "Entrez une phrase : " << flush ;
      cin >> ws ;
      getline(cin, phrase) ;
      sortie << phrase << endl ;
      sortie.close() ;
   }
}
```

Opérateurs ?

- Un opérateur est une opération sur un ou entre deux opérande(s) (variable(s)/expression(s)) :
- opérateurs arithmétiques (+, -, *, /, ...), opérateurs logiques (&&, ||, !), opérateurs de comparaison (==, >=, <=, ...), opérateur d'incrément (++), ...
- En pratique, un appel à un opérateur est similaire à un appel de fonction.
- A Op B se traduisant par : A.operatorOp(B)
- et Op A (unaire) par : A.operatorOp()

Opérateurs ?

- Exemples :

a + b	est la même chose que	a.operator+(b)
b + a		b.operator+(a)
a = b		a.operator=(b)
++a		a.operator++()
!a		a.operator !()
not a		a.operatornot()
cout « a		cout.operator<<(a)

Surcharge ?

- Rappel : surcharge de fonction

  Deux fonctions ayant le même nom mais pas les mêmes arguments.

- Exemple :

  ```
  int max(int, int) ;
  double max(double, double) ;
  ```

- Presque tous les opérateurs sont surchargeables (sauf, parmi ceux que vous connaissez, `::` et `.`).

- Leur surcharge ne pose généralement pas plus de problèmes que la surcharge des fonctions.

- La surcharge des opérateurs peut être réalisée soit à l'intérieur, soit à l'extérieur de la classe à laquelle ils s'appliquent.

Intérêt ?

Exemple avec les nombres complexes :

```
class Complexe  ... ;
Complexe a, b, c ;
```

plutôt que d'avoir à écrire :

```
a = b.addition(c) ;
```

écrire :

```
a = b + c ;
```

est quand même plus naturel...

Intérêt ?

Exemple de division de polynômes :

```
...
while ((dq >= 0) && (reste != POLYNOME_NUL))
{
met_coef( reste[degre(reste)] /
      denominateur[degre(denominateur)],dq, quotient ) ;
soustrait(reste, monome_mult(quotient[dq],
      dq,denominateur)) ;
...
division(a, b, p, q) ;
```

Intérêt ?

n'est-ce pas plus facile d'écrire :

```
...
while ((dq >= 0) && (reste  != 0))
{
quotien[dq] = reste.haut() / denominateur.haut() ;
reste -= Polynome(quotient[dq], dq) * denominateur ;
...

p = a / b ; q = a % b ;
```

Surcharge des opérateurs

Pour surcharger un opérateur Op dans une classe Classe, il faut ajouter la définition de la méthode prédéfinie operatorOp dans la classe en question :

```
class Classe {
...
// prototype de l'opérateur Op
type_retour operatorOp (type_argument) ;
...
} ;
// définition de l'opérateur Op
type_retour Classe : : operatorOp(type_argument) {
...
}
```

Surcharge des opérateurs

Exemple 1 :

```
class Complexe {
...
Complexe operator-(Complexe const& z2) const ;
// exemple : z3 = z1 - z2 ;
void operator-=(Complexe const& z2) ;
// exemple : z1 -= z2 ;
} ;

void Complexe : :operator-=(Complexe const& z2) {
x -= z2.x ;
y -= z2.y ; }

Complexe Complexe : :operator-(const Complexe& z2) const {
Complexe z3(*this) ; // utilise le constructeur de copie : copie z1 dans
z3.
z3 -= z2 ; // utilise l'opérateur -= redéfini ci-dessus
return z3 ; }
```

Surcharge des opérateurs

Exemple 2 :

```
class Polynome {
...
Polynome operator-(Polynome const& q) const ; // ex : r = p-q ;
Polynome& operator-=(Polynome const& q) ; // ex : p -= q ;
} ;

Polynome& Polynome : : operator-=(Polynome const& q) {
// l'ancienne fonction soustrait
while (degre() < q.degre()) p.push_back(0) ;
for (unsigned int i(0) ; i <= q.degre() ; ++i) p[i] -= q.p[i] ;
simplifie() ;
return *this ; }

Polynome Polynome : : operator-(const Polynome& q) const {
return Polynome(*this) -= q ;
// utilise le constructeur de copie et l'opérateur -= }
```

Opérateurs d'affectation et constructeur de copie

L'opérateur d'affectation = (utilisé par exemple dans a = b) est similaire au constructeur de copie, sauf que le premier s'appelle lors d'une affectation et le second lors d'une initialisation.

En général, on pourra créer une méthode (privée) unique pour les deux, par exemple copie(), qui est appelée à la fois par le constructeur de copie et par l'opérateur d'affectation :

```
Classe& Classe : : operator=(Classe const& source) {
if (&source != this) {
// Copie effective des données
copie(source) ; }
return *this ; }
```

Surcharge Externe

La surcharge externe est utile pour des opérateurs concernés par une classe, mais pour lesquels la classe en question n'est pas l'opérande de gauche.

Exemples :

1. multiplication d'un polynôme par un double
   double a ;
   Polynome p, q ;
   q = a * p ;
   s'écrirait a.operateur*(p) ;
   ce qui n'a pas de sens (a n'est pas un objet mais de type élémentaire double).

2. écriture sur cout : cout << p
   Il s'agit bien ici de cout.operateur<<, mais on souhaite le surcharger dans la classe de p et non pas dans la classe de cout (ostream).

Surcharge Externe

Dans ces cas on utilise des opérateurs externes, c'est-à-dire ne faisant pas partie de la classe.

Le opérateurs externes se déclarent avec un argument de plus que les opérateurs internes : la classe doit être ici explicitée.

Déclarations (hors de la classe) :

```
Polynome operateur*(double, Polynome const&) ;
ostream& operator<<(ostream&, Polynome const&) ;
```

Parfois, il peut être nécessaire d'ajouter, dans la classe, ce même prototype, précédé du mot clé friend :

```
friend Polynome operateur*(double, Polynome const&) ;
friend ostream& operator<<(ostream&, Polynome const&) ;
```

Le mot clé friend signifie que ces opérateurs, bien que ne faisant pas partie de la classe, peuvent avoir accès aux attributs et méthodes privés de la classe. (passer par les accesseurs (« méthodes get »))

Définitions : les définitions sont mises hors de la classe (et sans le mot clé friend)

Surcharge Externe

Exemple 1 :

```
Complexe operateur*(double a, Complexe const& z) {
return z * a ; // utilisation ici de l'operateur interne
}

ostream& operator<<(ostream& out, Complexe const& z) {
out << '(' << z.x << ", " << z.y << ' ) ' ;
return out ;
}
```

ou mieux (via les accesseurs) :

```
ostream& operator<<(ostream& out, Complexe const& z) {
out << '(' << z.get_x() << ", " << z.get_y() << ')' ;
return out ;
}
```

Surcharge Externe

Exemple 2 :

```
Polynome operateur*(double a, Polynome const& p) {
return p * a ; // utilisation ici de l'operateur interne
}

ostream& operator<<(ostream& out, Polynome const& p) {
// plus haut degré : pas de signe + devant
Degre i(p.degre()) ;
affiche_coef(out, p.coef(i), i, false) ;
// degré de N à 0 : + a * X^i
for (--i ; i >= 0 ; --i) affiche_coef(out, p.coef(i), i) ;
// degré 0 : afficher quand meme le 0 si rien d'autre
if ((p.degre() == 0) && (p.coef(0) == 0.0))
out << 0 ;
return out ;
}
```

Attention ! ! !

Attention de ne pas utiliser la surcharge des opérateurs à mauvais escient et à les écrire avec un soin particulier

Les performances du programme peuvent en être gravement affectées par des opérateurs surchargés mal écrits.

En effet, l'utilisation inconsidérée des opérateurs peut conduire à un grand nombre de copies d'objets

Utiliser des références quand cela est approprié

Attention ! ! !

Exemple :

```
Polynome Polynome : : operator-=(Polynome q) {
   Polynome local ;
   local = this ;
   while (local.degre() < q.degre()) local.ajoute(0) ;
   for (unsigned int i(0) ; i <= q.degre() ; ++i)
      local[i] -= q[i] ;
   local.simplifie() ;
   return local ;
}
```

Plein de copies inutiles ! Alors que :

```
Polynome& Polynome : : operator-=(Polynome const& q) {
   while (degre() < q.degre()) p.push_back(0) ;
   for (unsigned int i(0) ; i <= q.degre() ; ++i)
      p[i] -= q.p[i] ;
   simplifie() ;
   return *this ;
}
```

Quelle surcharge d'opérateurs ?

En pratique, vous pouvez choisir un degré divers de complexité dans la surcharge des opérateurs :

1. ne pas du tout faire de surcharge des opérateurs ;
2. surcharger simplement les opérateurs arithmétique de base (+, -, ...) sans leur version auto-affectation (+=, -=, ...) ; libre à vous ici de choisir ou non la surcharge (externe) de l'opérateur d'affichage (<<) ;
3. surcharger les opérateurs en utilisant leur version auto-affectation, mais sans valeur de retour :
   `void operator+=(MaClasse const&) ;`
4. faire la surcharge complète comme présentée précédemment, avec gestion de la valeur de retour des opérateurs d'auto-affectation :
   `MaClasse& operator+=(MaClasse const&) ;`

Exemples de surcharges usuelles

```
bool operator==(Classe const&) const ; // ex : p == q
bool operator<(Classe const&) const ; // ex : p < q

Classe& operator=(Classe const&) ; // ex : p = q

Classe& operator+=(Classe const&) ; // ex : p += q
Classe& operator-=(Classe const&) ;

Classe& operator++() ; // ex : ++p
Classe& operator++(int useless) ; // ex : p++

Classe& operator*=(autre_type const) ; // ex : p *= x ;

Classe operator+(Classe const&) const ; // r = p + q
Classe operator-(Classe const&) const ;

Classe operator-() const ; // ex : q = -p ;

[friend] ostream& operator<<(ostream&, Classe const&) ; // ex : cout « p ;

[friend] Classe operator*(autre_type, Classe const&) ; // ex : q = x * p ;
```

Liste des opérateurs pouvant être surchargés

1. Le but de cette liste est juste de vous donner tous les symboles possibles pour la surcharge d'opérateurs.
2. Évitez de trop changer le sens (la sémantique) d'un opérateur lorsque vous le surchargez.
3. Évitez absolument de surcharger :

,	
->	->*
new	new[]
delete	delete[]

Liste des opérateurs pouvant être surchargés

=	+	-	*	/	∧	%
==	+=	-=	*=	/=	∧=	%=
<	<=	>	>=	<<	>>	<<=
>>=	++	–	&	|	!	&=
|=	!=	,	->	->*	[]	()
∼	&&	||	xor	xor_eq	and	and_eq
or	or_eq	not	not_eq	bitand	bitor	compl
new	new[]	delete	delete[]			

Surcharges d'opérateurs

```
class Classe { ...
type_retour operatorOp(type_argument) ; // prototype de l'operateur Op
... } ;
// définition de l'operateur Op
type_retour Classe : : operatorOp(type_argument) { ... }
// operateur externe
type_retour operatorOp(type_argument, Classe&) { ... }
```

Quelques exemple de prototypes :

```
bool operator==(Classe const&) const ; // ex : p == q
bool operator<(Classe const&) const ; // ex : p < q
Classe& operator=(Classe const&) ; // ex : p = q
Classe& operator+=(Classe const&) ; // ex : p += q
Classe& operator++() ; // ex : ++p
Classe& operator*=(const autre_type) ; // ex : p *= x ;
Classe operator-(Classe const&) const ; // ex : r = p - q
Classe operator-() const ; // ex : q = -p ;
// operateurs externes
ostream& operator<<(ostream&, Classe const&) ;
Classe operator*(double, Classe const&) ;
```

C'est quoi une S.D.A. ?

La notion de structure de données abstraite (S.D.A.) est indépendante de tout langage de programmation

Une S.D.A. est un ensemble organisé d'informations (ou données) reliées logiquement et pouvant être manipulées non seulement individuellement mais aussi comme un tout.

Vous connaissez déjà des structures de données abstraites, très simples : les types élémentaires.
Par exemple, un int
interactions : affectation, lecture de la valeur, +, -, *, /

Spécifications des SDA ?

Une S.D.A. est caractérisée par :

- son contenu
- les interactions possibles (manipulation, accès, ...)

Du point de vue informatique, une structure de données abstraite peut être spécifiée à deux niveaux :

- niveau fonctionnel / logique : spécification formelle des données et des algorithmes de manipulation associés
- niveau physique (programmation) : comment est implémentée la structure de données abstraite dans la mémoire de la machine

  $\Longrightarrow$ déterminant pour l'efficacité des programmes utilisant ces données.

C'est quoi une SDA ?

Au niveau formel (modèle), on veut généraliser cette idée «dobjets» manipulables par des opérateurs propres, sans forcément en connaître la structure interne et encore moins l'implémentation.

Par exemple, vous ne pensez pas un int comme une suite de 32 bits, mais bien comme un «entier» (dans un certain intervalle) avec ses opérations propres : +, -, *, /

Une structure de données abstraite définit une abstraction des données et cache les détails de leur implémentation.

abstraction : identifier précisément les caractéristiques de l'entité (par rapport à ses applications), et en décrire les propriétés.

Spécifications des SDA ?

Une structure de données abstraite modélise donc l'«ensemble des services» désirés plutôt que l'organisation intime des données (détails d'implémentation)

On identifie usuellement 4 types de « services » :

1. les modificateurs, qui modifient la S.D.A.
2. les sélecteurs, qui permettent « d'interroger » la S.D.A.
3. les itérateurs, qui permettent de parcourir la structure
4. les constructeurs

Exemple

Tableau dynamique

modificateur : affectation d'un élément (t[i]=a)

sélecteur : lecture d'un élément (t[i])

sélecteur : le tableau est-il vide ? (t.size() == 0)

itérateur : index d'un élément ([i] ci-dessus)

Exemples de SDA ?

Il y a beaucoup de structures de données abstraites en Informatique.
Dans ce cours, nous n'allons voir que les 2 plus fondamentales :

- les listes
- et les piles

Autres :

- files d'attente (avec ou sans priorité)
- multi-listes
- arbres (pleins de sorte...)
- graphes
- tables de hachage

Les Listes

Spécification logique :
Ensemble d'éléments successifs (pas d'accès direct), ordonnés ou non

Interactions :

- accès au premier élément (sélecteur)
- accès à l'élément suivant d'un élément (sélecteur)
- modifier l'élément courant (modificateur)
- insérer/supprimer un élément après(/avant) l'élément courant (modificateur)
- tester si la liste est vide (sélecteur)
- parcourir la liste (itérateur)

Réalisation d'une liste

- Une liste peut être vu comme une structure récursive :

  liste = élément + liste OU liste = vide

- réalisation statique :

  tableau

- réalisation dynamique (liste chaînée) :

  vector
  ou
  classe :

```
class ListeChainee {
type_el donnee ;
ListeChainee suivant ;
}
```

Réalisation d'une liste

...

```
class ListeChainee {
type_el donnee ;
ListeChainee* suivant ;
}
```

...

```
typedef Cellule* PtrCell ;
class Cellule {
type_el donnee ;
PtrCell suivant ;
}
```

...

```
class Cellule ;
typedef Cellule* PtrCell ;
class Cellule {
type_el donnee ;
PtrCell suivant ;
}
```

Pourquoi les listes dynamiques ?

Les tableaux sont un type de données très utile en programmation mais présentent 2 limitations :

1. les données sont contiguës (les unes derrières les autres) et donc l'insertion d'un nouvel élément au milieu du tableau demande la recopie (le décalage) de tous les éléments suivants.
   $\Longrightarrow$ insertion en O(n)
2. pour les tableaux statiques, augmenter la taille (si elle n'est pas connue a priori) nécessite la création d'un nouveau tableau
   $\Longrightarrow$ O(n)

Exemple d'implémentation d'une liste

```
class Cellule ;
typedef Cellule* PtrCell ;
const PtrCell LISTE_VIDE(0) ;
// Une cellule de la liste
class Cellule{
public :
   Cellule(double un_double) : donnee(un_double), suite(LISTE_VIDE){}
   Cellule(double un_double, PtrCell suite) : donnee(un_double),
                                              suite(suite){}
   PtrCell getSuite(){return suite ;} ;
   double getDonnee(){return donnee ;} ;
   void setSuite(PtrCell une_suite){suite = une_suite ;}
private :
   double donnee ;
   PtrCell suite ;
} ;
```

Exemple d'implémentation d'une liste

```
// Le type Liste chainee
class Liste{
public :
   Liste() : queue(LISTE_VIDE){}
   bool est_vide() ;
   void insere(double un_double) ;
   void insere(Cellule& cell, double un_double) ;
   unsigned int taille() ;
private :
   PtrCell tete ; //un pointeur sur le premier élément
   PtrCell queue ; //un pointeur sur le dernier élément
} ;
```

Exemple d'insertion d'élément

- En queue de liste

```
void Liste : : insere(double un_double)
{
   if (est_vide()) {
      tete = new Cellule(un_double) ;
      queue = tete ;
   }
   else
      insere((*queue), un_double) ;
}
```

- Après un élément donné de la liste

```
void Liste : : insere(Cellule& existante, double un_double)
{
   PtrCell suite(existante.getSuite()) ;
   PtrCell c(new Cellule(un_double,suite)) ;
   existante.setSuite(c) ;
   if (c->getSuite() == LISTE_VIDE) queue = c ;
}
```

Calcul de la taille d'une liste

```
unsigned int Liste : : taille()
{
   unsigned int taille(0) ;
   PtrCell courant(tete) ;
   while(courant != LISTE_VIDE)
   {
      ++taille ;
      courant = courant->getSuite() ;
   }
   return taille ;
}
```

Les Piles

Spécification :

Une pile est une structure de données abstraite dynamique contenant des éléments homogènes (de type non précisé) à 1 point d'accès et permettant :

- d'ajouter une valeur à la pile (empiler ou push) ;
- de lire la dernière valeur ajoutée ;
- d'enlever la dernière valeur ajoutée (dépiler ou pop) ;
- de tester si la pile est vide.

On ne connait donc de la pile que le dernier élément empilé (son sommet).

Spécification physique :

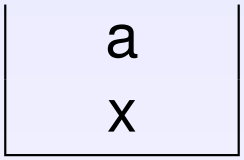
liste chaînée
ou

tableau dynamique

Exemple

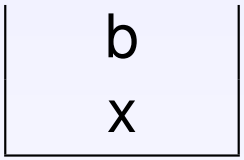
empiler x

empiler a

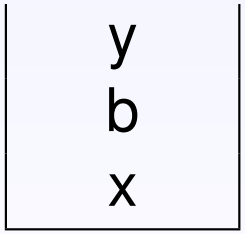


dépiler

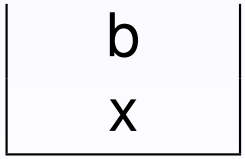
empiler b



empiler y



dépiler



Exemples d'utilisation des piles

Le problème des parenthèses :

Etant donnée une expression avec des parenthèses, est-elle bien ou mal parenthésée ?

$((a + b) * c - (d + 4) * (5 + (a + c))) * (c + (d + (e + 5 * g) * f) * a)$
$\Longrightarrow$ correct

$(a + b)($ $\Longrightarrow$ incorrect
$([])[()(()[])]$ $\Longrightarrow$ correct
$([)]$ $\Longrightarrow$ incorrect

Autres exemples d'utilisation des piles :

- tours de Hanoi
- notation postfixée : $(4\ 2 + 5 *) \Longrightarrow (5 * (4 + 2))$

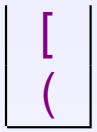
Algorithme de vérification des parenthèses

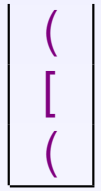
Tant que lire caractère c
Si c est (ou [
  empiler c
Sinon
  Si c est) ou]
    Si pile vide
      Echec
    Sinon
      c′ ← lire la pile
      Si c et c′ correspondent
        dépiler
      Sinon
        Echec
Si pile vide
  Ok
Sinon
  Echec

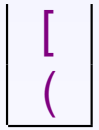
Exemple

Entrée : ([()])

empiler (

empiler [

empiler (

lu) $\longrightarrow$ correspond $\Longrightarrow$ dépile 

lu [$\longrightarrow$ correspond $\Longrightarrow$ dépile 

lu) $\longrightarrow$ correspond $\Longrightarrow$ dépile 

pile vide $\Longrightarrow$ Ok

Code C++

```
bool check(string s) {
    Pile p ;
    for (unsigned int i(0) ; i < s.size() ; ++i) {
        if ((s[i] == '(') || (s[i] == '['))
            p.empile(s[i]) ;
        else if (s[i] == ')') {
            if (( !p.est_vide()) && (p.top() == '('))
                p.depile() ;
            else
                return false ;
        } else if (s[i] == ']') {
            if (( !p.est_vide()) && (p.top() == '['))
                p.depile() ;
            else
                return false ;
        }
    }
    return p.est_vide() ;
}
```

Programmation générique : Introduction

Une cellule de notre liste chaînée du chapitre précédent se présentait comme suit :

```
// Une cellule de la liste
class Cellule{
public :
      //....
private :
      double donnee ;
      PtrCell suite ;
} ;
```

Si l'on veut une liste de int ?

$\Longrightarrow$ c'est exactement le même code pour Liste et Cellule sauf qu'il faut remplacer le type de la données pouvant être stockée dans une Cellule

$\Longrightarrow$ Duplication de code ! !

Programmation générique

L'idée de base est de passer les types de données comme paramètres pour décrire des traitements très généraux

$\Longrightarrow$ Il s'agit donc d'un niveau d'abstraction supplémentaire.

De tels modèles de classes/fonctions s'appellent aussi classes/fonctions génériques ou patrons, ou encore template.

Vous en connaissez déjà sans le savoir. Par exemple la classe vector n'est en fait pas une classe mais un modèle de classes :

$\Longrightarrow$ c'est le même modèle que l'on stocke des char (vector<char>), des int (vector<int>), ou tout autre objet.

Exemple

Exemple d'une fonction échangeant la valeur de 2 variables. Avec 2 entiers vous écririez une fonction comme :

```
// échange la valeur de ses arguments
void echange(int& i, int& j) {
   int tmp(i) ;
   i = j ;
   j = tmp ;
}
```

Mais vous vous rendez bien compte que vous pourriez faire la même chose avec deux double, ou même deux objets quelconques, pour peu qu'ils aient un constructeur de copie (Obj tmp(i) ;) et un opérateur de copie (operator=).

Exemple

L'écriture générale serait alors quelque chose comme :

```
// échange la valeur de ses arguments
void echange(Type& i, Type& j)
{
      Type tmp(i) ;
      i = j ;
      j = tmp ;
}
```

où Type est une représentation générique du type des objets à échanger.

Exemple

La façon exacte de le faire en C++ est la suivante :

```
// échange la valeur de ses arguments
template<typename Type>
void echange(Type& i, Type& j)
{
	Type tmp(i) ;
	i = j ;
	j = tmp ;
}
```

Exemple

On pourra alors utiliser la fonction echange avec tout type/classe pour lequel le constructeur de copie et l'opérateur d'affectation (=) sont définis.

Par exemple :

```
int a(2), b(4) ;
echange(a,b) ;

double da(2.3), db(4.5) ;
echange(da,db) ;

vector<double> va, vb ;
...
echange(va,vb) ;

string sa("ca marche"), sb("coucou") ;
echange(sa, sb) ;
```

Généralisation aux classes

Ce que l'on a fait ici avec une fonction, on peut le généraliser à n'importe quelle classe.

On pourrait par exemple vouloir créer une classe qui réalise une liste d'objets quelconques :

```
template<typename T1>
class Liste {
      //...
} ;
```

Rq. : un tel modèle de classe existe dans la STL : list.

Généralisation aux classes

On pourrait par exemple vouloir créer une classe qui réalise une paire d'objets quelconques :

```
template<typename T1, typename T2>
class Paire {
   public :
      Paire(const T1& un, const T2& deux) : premier(un), second(deux) {}
      virtual ~Paire(){}
      T1 get1() const { return premier ; }
      T2 get2() const { return second ; }
      void set1(const T1& val) { premier = val ; }
      void set2(const T2& val) { second = val ; }
   protected :
      T1 premier ;
      T2 second ;
} ;
```

Généralisation aux classes

et par exemple créer la classe « paire string-double » :

Paire<string,double>

ou encore la classe « paire char-unsigned int » :

Paire<char,unsigned int>

Rq. : un tel modèle de classe existe dans la STL : pair.

Les modèles de classes sont donc un moyen condensé d'écrire plein de classes potentielles à la fois.

Déclaration d'un modèle

Pour déclarer un modèle de classe ou de fonction, il suffit de faire précéder sa déclaration du mot clé template suivit de ses arguments (qui sont donc des noms génériques de type) suivant la syntaxe :

```
template<typename nom1, typename nom2, ...>
```

Exemple :

```
template<typename T1, typename T2>
class Paire {
...
```

Les types ainsi déclarés peuvent alors être utilisés dans la définition qui suit, exactement comme tout autre type.

Rq. : on peut aussi utiliser le mot class à la place de typename :

```
template<class T1, class T2>
class Paire {
...
```

Déclaration d'un modèle

Il est également possible de définir des types par défaut, avec la même contrainte que pour les paramètres de fonction : les valeurs par défaut doivent être placées en dernier.

Exemple :

```
template<typename T1, typename T2 = unsigned int>
class Paire {
...
```

qui permettrait de déclarer la classe «paire char-unsigned int» simplement par :

```
Paire<char>
```

Définitions externes des méthodes de modèles de classes

Si les méthodes d'un modèle de classes sont définies en dehors de cette classe, elle devront alors aussi être définies comme modèle et être précédées du mot clé template, mais...

...il est absolument nécessaire d'ajouter les paramètres du modèle (les types génériques) au nom de la classe
[pour bien spécifier que dans cette définition c'est la classe qui est en modèle et non la méthode.]

Exemple

```
template<typename T1, typename T2>
class Paire {
   public :
      Paire(const T1&, const T2&) ;
      ...
} ;

// définition du constructeur
template<typename T1, typename T2>
Paire<T1,T2> : : Paire(const T1& un, const T2& deux) :
      premier(un), second(deux) { }
```

Instanciation des modèles

La définition des modèles ne génère en elle-même aucun code : c'est juste une description de plein de codes potentiels.

Le code n'est produit que lorsque tous les paramètres du modèle ont pris chacun un type spécifique.

Lors de l'utilisation d'un modèle, il faut donc fournir des valeurs pour tous les paramètres (au moins ceux qui n'ont pas de valeur par défaut).
On appelle cette opération une instanciation du modèle.

Instanciation des modèles

L'instanciation peut être implicite lorsque le contexte permet au compilateur de décider de l'instance de modèle à choisir.

Par exemple, dans le code :

```
double da(2.3), db(4.5) ;
echange(da,db) ;
```

il est clair (par le contexte) qu'il s'agit de l'instance `echange<double>` du modèle `template<typename T> void echange(T&,T&)` ; qu'il faut utiliser.

Instanciation des modèles

Mais dans la plupart des cas, on explicite l'instanciation lors de la déclaration d'un objet.

C'est ce qui vous faites lorsque vous déclarez par exemple
vector<double> tableau ;

Il suffit dans ce cas de spécifier le(s) type(s) désiré(s) après le nom du modèle de classe et entre <>.

L'instanciation explicite peut aussi être utile dans les cas où le contexte n'est pas suffisamment clair pour choisir.

Par exemple avec le modèle de fonction

```
template <typename Type>
Type monmax(const Type& x, const Type& y) {
      if (x < y) return y ;
      else return x ;
}
```

l'appel monmax(3.14, 7) ; est ambigu. Il faudra alors écrire
monmax<double>(3.14, 7) ;

Modèles, surcharge et spécialisation

Les modèles de fonctions peuvent très bien être surchargés comme les fonctions usuelles (puisque, encore une fois, ce sont juste une façon condensée d'écrire plein de fonctions à la fois).

Par exemple :

```
template<typename Type>
void affiche(const Type& t) {
      cout << "J'affiche " << t << endl ;
}
// surcharge pour les pointeurs : on préfère ici écrire
// le contenu plutôt que l'adresse.
template<typename Type> void affiche(Type* t) {
      cout << "J'affiche " << *t « endl ;
}
```

Rq. : on aurait même pu faire mieux en écrivant :

```
template<typename Type> void affiche(Type* t) {
      affiche<Type>(*t) ;
}
```

qui fait appel au premier modèle.

Modèles, surcharge et spécialisation

Mais les modèles (y compris les modèles de classes) offrent un mécanisme supplémentaire : la spécialisation qui permet de définir une version particulière d'une classe ou d'une fonction pour un choix spécifique des paramètres du modèle.

Par exemple, on pourrait spécialiser le second modèle ci-dessus dans le cas des pointeurs sur des entiers :

```
template<> void affiche<int>(int* t) {
    cout << "J'affiche le contenu d'un entier : " << *t << endl ;
}
```

La spécialisation d'un modèle (lorsqu'elle est totale) se fait en :

- ajoutant template<> devant la définition
- nommant explicitement la classe/fonction spécifiée
  C'est le <int> après affiche dans l'exemple ci-dessus.

Exemple de spécialisation de classe

```
template<typename T1, typename T2>
class Paire {
    ...
};
// spécialisation pour les paires <string,int>
template<> class Paire<string,int> {
public :
        Paire(const string& un, int deux) : premier(un), second(deux) {}
        virtual ~Paire() {}
        string get1() const { return premier ; }
        int get2() const { return second ; }
        void set1(const string& val) { premier = val ; }
        void set2(int val) { second = val ; }
        // une méthode de plus
        void add(int i) { second += i ; }
protected :
        string premier ;
        int second ;
};
```

Spécialisation : remarques

- La spécialisation peut également s'appliquer uniquement à une méthode d'un modèle de classe sans que l'on soit obligé de spécialiser toute la classe.
  Utilisée de la sorte, la spécialisation peut s'avérer particulièrement utile.
- La spécialisation n'est pas une surcharge car il n'y a pas génération de plusieurs fonctions de même nom (de plus que signifie une surcharge dans le cas d'une classe ?) mais bien une instance spécifique du modèle.
- il existe aussi des spécialisations partielles (de classe ou de fonctions), mais cela nous emmènerait trop loin dans ce cours.

Modèles de classes et compilation séparée

Les modèles de classes doivent nécessairement être définis au moment de leur instanciation afin que le compilateur puisse générer le code correspondant.

Ce qui implique, lors de compilation séparée, que les fichiers d'en-tête (.h) doivent contenir non seulement la déclaration, mais également la définition complète de ces modèles ! !

On ne peut donc pas séparer la déclaration de la définition dans différents fichiers... Ce qui présente plusieurs inconvénients :

- Les mêmes instances de modèles peuvent être compilées plusieurs fois,
- et se retrouvent en de multiples exemplaires dans les fichiers exécutables.
- On ne peut plus cacher leurs définitions (par exemple pour des raisons de confidentialité, etc...)

Modèles de classes et compilation séparée

Notez également que les fichiers contenant des modèles ne sont pas des fichiers sources classiques, puisque sans instanciation, ils ne génèrent aucun code machine.

Pour résoudre ces problèmes, le langage C++ donne en principe la possibilité d'exporter les définitions des modèles dans des fichiers complémentaires à l'aide de la directive export.

Malheureusement celle-ci n'est pour l'instant pas gérée par les compilateurs actuels :
warning : keyword 'export' not implemented, and will be ignored

En l'état, il faut donc joindre prototype et définitions des modèles dans les mêmes fichiers d'en-tête.

Le seul « point noir » des templates est donc lié à la faiblesse des compilateurs actuels qui ne gèrent pas la directive export.

Récapitulatif

Déclarer un modèle de classe ou de fonction :

template<typename nom1, typename nom2, ...>

Définition externe des méthodes de modèles de classes :
template<typename nom1, typename nom2, ...>
NomClasse<nom1, nom2, ...> : : NomMethode(...

Instanciation : spécifier simplement les types voulus après le nom de la classe/fonction, entre <> (Exemple : vector<double>)

Spécialisation (totale) de modèle pour les types type1, type2... :
template<> NomModele<type1,type2,...> ...suite de la declaration...

Compilation séparée : pour les templates, il faut tout mettre (déclarations et définitions) dans le fichier d'en-tête (.h).

Objectifs

L'objectif de ce chapitre est de vous présenter (sommairement) un certains nombre d'outils standards existant en C++

Le but ici n'est pas d'être exhaustif, mais simplement de vous :

- informer de l'existence des principaux outils
- faire prendre conscience d'aller lire/chercher dans la documentation les éléments qui peuvent vous être utiles

Bibliothèque Standard

La bibliothèque standard C++ facilite la programmation et permet de la rendre plus efficace, si tant est que l'on connaisse bien les outils qu'elle fournit.

Cette bibliothèque est cependant vaste et complexe, mais elle peut dans la plupart des cas s'utiliser de façon très simple, facilitant ainsi la réutilisation des structures de données abstraites et des algorithmes sophistiqués qu'elle contient.

La bibliothèque standard est formée de 3 composants principaux :

- la STL : Standard Template Library
- les autres outils de la bibliothèque standard C++
- les bibliothèques C

Contenu de la bibliothèque standard

La bibliothèque standard C++ contient 51 «paquets» différents dont 16 forment la STL :

<algorithm>	(STL) plusieurs algorithmes utiles
<bitset>	gestions d'ensembles de bits
<complex>	les nombres complexes
<deque>	(STL) tableaux dynamiques à double sens
<exception>	diverses fonctions aidant à la gestion des exceptions
<fstream>	manipulation de fichiers
<functional>	(STL) foncteurs
<iomanip>	manipulation de l'état des flots
<ios>	définitions de base des flots
<iosfwd>	flots anticipés
<iostream>	flots standards
<istream>	flots d'entrée
<iterator>	(STL) itérateurs (inclus dans les containers associés)
<limits>	diverses bornes concernant les types numériques

Contenu de la bibliothèque standard

<list>	(STL) listes doublement chaînées
<locale>	contrôles liés au choix de la langue
<map>	(STL) tables associatives clé-valeur
<memory>	(STL) gestion mémoire pour les containers
<new>	gestion mémoire
<numeric>	(STL) fonctions numériques
<ostream>	flots de sortie
<queue>	(STL) files d'attente
<set>	(STL) ensembles (au sens mathématique)
<sstream>	flots dans des chaînes de caractères
<stack>	(STL) piles
<stdexcept>	gestion des exceptions
<streambuf>	flots avec tampon (buffer)
<string>	chaînes de caractères
<strstream>	flots dans des chaînes de caractère (en mémoire)
<typeinfo>	information sur les types
<utility>	(STL) divers utilitaires
<valarray>	tableaux orientés vers les valeurs
<vector>	(STL) tableaux dynamiques

Bibliothèque standard

Il y a aussi les 18 «paquets» venant du langage C :

<cassert>	test d'invariants lors de l'exécution
<cctype>	divers informations sur les caractères
<cerrno>	code d'erreurs retournés dans la bibliothèque standard
<cfloat>	diverses informations sur la représentation des réels
<ciso646>	variantes ISO 646 de certains opérateurs
<climits>	diverses informations sur la représentation entiers
<clocale>	adaptation à diverses langues
<cmath>	divers définitions mathématiques
<csetjmp>	branchement non locaux
<csignal>	contrôle des signaux (processus)
<cstdarg>	nombre variables d'arguments
<cstddef>	divers définitions utiles (types et macros)
<cstdio>	entrées sorties de base
<cstdlib>	diverses opérations de base utiles
<cstring>	manipulation des chaînes de caractère à la C
<ctime>	diverses conversions de date et heures
<cwchar>	encore d'autres type de caractères (et de chaînes)
<cwctype>	classification des codes de caractères étendus

Outils Standards

On distingue plusieurs types d'outils. Parmi les principaux :

- les containers de base
- les containers avancés (appelés aussi « adaptateurs »)
- les itérateurs
- les algorithmes
- les outils numériques
- les traitements d'erreurs
- les chaînes de caractères
- les flots

Outils Standards

Les outils les plus utilisés par les débutants sont :

- les chaînes de caractères (string)
- les flots (stream)
- les tableaux dynamiques (vector) [container]
- les listes chaînées (list) [container avancé]
- les piles (stack) [container avancé]
- les algorithmes de tris (sort)
- les algorithmes de recherche (find)
- les itérateurs (iterators)

Plan

Nous détaillerons dans la suite certains de ces outils :

- list [container]
- set [container]
- iterator
- map [container]
- stack [container avancé]
- queue [container avancé]
- sort
- find
- complex
- cmath

Containers

Comme le nom l'indique, les containers sont des structures de données abstraites (SDA) servant à contenir («collectionner») d'autres objets.

Vous en connaissez déjà plusieurs : les tableaux, les piles et les listes chaînées.

Il en existe plusieurs autres, parmi lesquels, les files d'attentes (queue), les ensembles (set) et les tables associatives (map).

Containers

Les files d'attente sont des piles où c'est le premier arrivé (empilé) qui est dépilé le premier. Alors que dans une pile «normale», c'est le dernier arrivé qui est dépilé en premier.

Les set permettent de gérer des ensembles (finis !) au sens mathématique du terme : collection d'éléments où chaque élément n'est présent qu'une seule fois.

Les tables associatives sont une généralisation des tableaux où les index ne sont pas forcément des entiers.

Imaginez par exemple un tableau que l'on pourrait indexer par des chaînes de caractères et écrire par exemple tab["Informatique"]

Containers

Tous les containers contiennent les méthodes suivantes :

bool empty() : le containers est-il vide ?

unsigned int size() : nombre d'éléments dans le container

void clear() : vider le container

iterator erase(it) : supprime du container l'élément pointé par it. it est un itérateur (généralisation de la notion de pointeur)

Ils possèdent également tous les méthodes begin() et end() que nous verrons avec les itérateurs.

Liste doublement chaînées

Elle sont, comme les tableaux dynamiques, des SDA séquentielles : stockant des séquences (ordonnées) d'éléments.

Par contre dans une liste chaînée, l'accès direct à un élément n'est pas possible, contrairement aux tableaux dynamiques.

Les listes chaînées sont définies dans la bibliothèque list et se déclarent de façon similaire à des tableaux dynamiques :
list<double> maliste ;

Quelques méthodes des listes chaînées :

Type& front()	retourne le premier élément de la liste
Type& back()	retourne le dernier élément de la liste
void push_front(Type)	ajoute un élément en tête de liste
void push_back(Type)	ajoute un élément en queue de liste
void pop_front()	supprime le premier élément
void pop_back()	supprime le dernier élément
void insert(iterator, Type)	insertion avant un élément de la liste désigné par un itérateur

Tableaux dynamiques

Pour accéder directement à un élément d'un tableau dynamique (vector) on utilise l'opérateur [] : tab[i].

Il existe une autre méthode pour cet accès : at(n) qui, à la différence de [n], lance l'exception out_of_range (de la bibliothèque <stdexcept>) si n n'est pas un index correct.

Exemple :

```
#include <vector>
#include <stdexcept>
...
vector<int> v(5,3) ; // 3, 3, 3, 3, 3
int n(12) ;
try {
      cout << v.at(n) << endl ;
}
catch (out_of_range) {
      cerr << "Erreur : " << n << " n'est pas correct pour v" << endl
            << "qui ne contient que " << v.size() << "éléments." << endl ;
}
```

Les ensembles

Les ensembles (au sens mathématique) sont implémentés dans la bibliothèque <set>. Ils ne peuvent cependant contenir que des éléments du même type.

On déclare un ensemble comme les autres containers, en spécifiant le type de ses éléments : set<char> monensemble ;

Les ensembles n'étant pas des SDA séquentielles, l'accès direct à un élément n'est pas possible.

Quelques méthodes des ensembles :

void insert(Type)	insère un élément s'il n'y est pas déjà
void erase(Type)	supprime l'élément (s'il y est)
iterator find (Type)	retourne un itérateur indiquant l'élément recherché

À noter que la bibliothèque <algorithm> fournit des fonctions pour faire la réunion, l'intersection et la différence d'ensembles.

Les ensembles

```
#include <set>
...
set<char> voyelles ;
voyelles.insert('a') ;
voyelles.insert('b') ;
voyelles.insert('e') ;
voyelles.insert('i') ;
voyelles.erase('b') ;
voyelles.insert('e') ; // n'insere pas 'e' car il y est déjà
```

Comment parcourir cet ensemble ?

```
for (int i(0) ; i < voyelles.size() ; ++i)
      cout << voyelles[i] << endl ;
```

ne fonctionne pas car c'est une SDA non-séquentielle : {a, b} = {b, a}

$\Longrightarrow$ utilisation d'itérateurs

Les itérateurs

Les itérateurs sont une SDA généralisant d'une part des accès par index (SDA séquentielles) et d'autre part les pointeurs, dans le cas de containers.

Ils permettent :

- de parcourir de façon itérative les containers
- d'indiquer (de pointer sur) un élément d'un container

Il existe en fait 5 sortes d'itérateurs, mais nous ne parlons ici que de la plus générale, qui permet de tout faire : lecture et écriture du containers, aller en avant ou en arrière (accès quelconque en fait).

Les itérateurs

Un itérateur associé à un container C<type> se déclare simplement comme C<type> : : iterator nom ;

Exemples :
vector<double> : : iterator i ;
set<char> : : iterator j ;

Il peut s'initialiser grâce aux méthodes begin() ou end() du container, voire d'autres méthodes spécifiques, comme par exemple find pour les containers non-séquentiels.

Exemples :
vector<double> : : iterator i(monvect.begin()) ;
set<char> : : iterator j(monset.find(monelement)) ;

L'élément indiqué par l'itérateur i est simplement *i, comme pour les pointeurs.

Retour sur l'exemple des ensembles

Pour parcourir notre ensemble précédent, nous devons donc faire :

```
for (set<char> : : iterator i(voyelles.begin()) ; i != voyelles.end() ; ++i)
        cout << *i << endl ;
```

Exemple d'utilisation de find :

```
set<char> : : iterator i(voyelles.find('c')) ;
if (i == voyelles.end())
        cout << "pas trouvé" << endl ;
else
        cout << *i << " trouvé" << endl ;
```

Exemple

```
#include <set>
#include <iterator>
#include <iostream>
using namespace std ;
int main() {
   set<char> voyelles ;
   voyelles.insert('a') ;        voyelles.insert('b') ;
   voyelles.insert('e') ;        voyelles.insert('i') ;
   voyelles.insert('a') ; // ne fait rien car 'a' y est déjà
   voyelles.erase('b') ; // supprime 'b'
   // parcours l'ensemble
   for (set<char> : : iterator i(voyelles.begin()) ; i !=voyelles.end() ; ++i)
      cout << *i << endl ;
   // recherche d'un élément
   set<char> : : iterator element(voyelles.find('c')) ;
   if (element == voyelles.end())
      cout << "je n'ai pas trouvé l'élément que vous cherchez." << endl ;
   else
      cout << *element << " trouvé !" << endl ;
   return 0 ;
}
```

Suppression d'un élément d'un container

On a vu que tout container possédait une méthode

iterator erase(it)

permettant de supprimer un élément, mais...

Attention ! on ne peut pas continuer à utiliser l'itérateur it sans autre ! (plus exactement : erase rend invalide tout itérateur et référence situé(e) au delà du premier point de suppression)

Exemple d'erreur classique :

```
vector<double> v ;
...
for (vector<double> : :iterator i(v.begin()) ; i != v.end() ; ++i)
    if (cond(*i)) v.erase(i) ;
```

(avec bool cond(double) ;)
n'est pas correct («Segmentation fault»)

pas plus que :

```
for (vector<double> : : iterator i(v.begin()) ; i != v.end() ; ++i)
    if (cond(*i)) i = v.erase(i) ;
```

Suppression d'un élément d'un container

Ce qu'il faut faire c'est :

```
vector<double> : : iterator next ;
for (vector<double> : : iterator i(v.begin()) ; i != v.end() ; i = next)
      if (cond(*i)) { next = v.erase(i) ; }
      else { next = ++i ; }
```

ou mieux en utilisant remove_if (ou remove) de <algorithm> :

```
      v.erase(remove_if(v.begin(), v.end(), cond), v.end()) ;
```

En effet, un tableau dynamique n'est pas la bonne SDA si l'on veut détruire un élément au milieu et garder l'ordre (utiliser plutôt des listes chaînées pour cela)

Note : si l'on ne tient pas à garder l'ordre, on peut toujours faire :

```
for (unsigned int i(0) ; i < v.size() ; ++i)
   if (cond(v[i])) {
      v[i] = v[v.size()-1] ;
      v.pop_back() ;
      --i ;
   }
```

Tables associatives

Les tables associatives sont une généralisation des tableaux où les index ne sont pas forcément des entiers.

Imaginez par exemple un tableau que l'on pourrait indexer par des chaînes de caractères et écrire par exemple tab["Informatique"]

On parle d'« associations clé-valeur »

Les tables associatives sont définies dans la bibliothèque <map>.

Elles nécessitent deux types pour leur déclaration : le type des «clés» (les index) et le type des éléments indexé.

Par exemple, pour indexer des nombres réels par des chaînes de caractères on déclarera :

```
map<string,double> une_variable ;
```

Exemple

```
#include <map>
#include <string>
#include <iostream>
using namespace std ;
int main()
{
   map<string,double> moyenne ;
   moyenne["Informatique"] = 5.5 ;
   moyenne["Physique"] = 4.5 ;
   moyenne["Histoire des maths"] = 2.5 ;
   moyenne["Analyse"] = 4.0 ;
   moyenne["Algèbre"] = 5.5 ;
   // parcours de tous les éléments
   for (map<string,double> : : iterator i(moyenne.begin()) ; i !=
moyenne.end() ; ++i)
      cout << "En " << i->first << ", j'ai " « i->second << " de moyenne." <<
endl ;
// recherche
   cout << "Ma moyenne en Informatique est de " ;
   cout << moyenne.find("Informatique")->second << endl ;
   return 0 ;
}
```

Piles et Files

Pour utiliser les piles de la STL : #include <stack>

Les files d'attente sont des piles où c'est le premier arrivé (empilé) qui est dépilé le premier. Elles sont définies dans la bibliothèque <queue>.

Une pile de type type se déclare par stack<type> et une file d'attente par queue<type>. Par exemple :

```
stack<double> une_pile ;
queue<char> attente ;
```

Méthodes :

Type top()	accède au premier élément (sans l'enlever)
void push(Type)	empile/ajoute
void pop()	dépile/supprime
bool empty()	teste si la pile/file est vide

Piles - Exemple

```
#include <stack>
using namespace std ;
...
bool check(string s) {
   stack<char> p ;
   for (unsigned int i(0) ; i < s.size() ; ++i) {
      if ((s[i] == '(') || (s[i] == '['))
         p.push(s[i]) ;
      else if (s[i] == ')') {
         if (( !p.empty()) && (p.top() == '('))
            p.pop() ;
         else
            return false ;
      } else if (s[i] == ']') {
         if (( !p.empty()) && (p.top() == '['))
            p.pop() ;
         else
            return false ;
      }
   }
   return p.empty() ;
}
```

La bibliothèque Algorithm

La bibliothèque algorithm (i.e. #include <algorithm>) définit différents types d'algorithmes généraux :

- de séquencement : for_each, find, random_shuffle, copy
- de tris : sort, ...
- numériques : inner_product, partial_sum, adjacent_difference

Dans ce cours , les 3 suivants seront présentés :

- find
- copy et les output_iterators
- sort

pour les autres : référez-vous à la documentation

find

find est un algorithme général permettant de faire des recherches dans (une partie d')un container.

Son prototype général est :

```
iterator find(iterator debut, iterator fin, Type valeur) ;
```

qui cherche valeur entre debut (inclu) et fin (exclu). Il retourne un itérateur sur le contenu correspondant à la valeur recherchée ou fin si cette valeur n'est pas trouvée.

Exemple :

```
list<int> uneliste ;
uneliste.push_back(3) ;
uneliste.push_back(1) ;
uneliste.push_back(7) ;

list<int> : : iterator result(find(uneliste.begin(), uneliste.end(), 7)) ;
if (result != uneliste.end()) cout << "trouvé" ;
else cout << "pas trouvé" ;

cout << endl ;
```

copy

copy est un algorithme général pour copier (une partir d')un container dans un autre.

Son prototype général est :

OutputIterator copy(InputIterator debut, InputIterator fin, OutputIterator resultat) ;

qui copie le contenu compris entre debut (inclus) et fin (exclus) vers resultat (inclus) et les positions suivantes (itérateurs).

La valeur de retour est resultat + (fin - debut).

Notez bien que cela copie des éléments, mais ne fait pas d'insertion : il faut absolument que resultat ait (i.e. pointe sur) la place nécessaire !

Exemple :

```
copy(unensemble.begin(), unensemble.end(), untableau.begin()) ;
```

Notez que l'on peut ainsi copier des données d'une SDA dans une autre SDA d'un autre type.

copy

copy peut être très utile pour afficher le contenu d'un container sur un flot en utilisant un ostream_iterator :

Exemple :

```
copy(container.begin(), container.end(),
        ostream_iterator<int>(cout, ", ")) ;
```

container contenant ici des int, son contenu sera affiché sur cout, séparé par des ' , '.

Exemple

```
#include <iostream>
#include <set>
#include <vector>
#include <iterator>
using namespace std ;
int main() {
   set<double> unensemble, unautre ;
   unensemble.insert(1.1) ;
   unensemble.insert(2.2) ;
   unensemble.insert(3.3) ;
   // copy(unensemble.begin(), unensemble.end(), unautre.begin()) ;
   // ne fonctionne pas ("assignment of read-only location")
   // car unautre n'a pas la taille suffisante.
   vector<double> untableau(unensemble.size()) ; // prévoit la place
   copy(unensemble.begin(), unensemble.end(), untableau.begin()) ;
   // output
   cout << "untableau = " ;
   copy(untableau.begin(), untableau.end(),
   ostream_iterator<double>(cout, ", ")) ;
   cout << endl ;
   return 0 ;
}
```

sort

sort permet de trier des SDA implémentées sous forme de containers. La version la plus simple de tri est :

void sort(iterator debut, iterator fin)

qui utilise operator< des éléments contenus dans la partie du container indiquée par debut et fin (les objets qui y sont stockés doivent donc posséder cet opérateur).

Exemple 1 :

```
list<double> uneliste ;
...
sort(uneliste.begin(), uneliste.end()) ;
```

Exemple 2 :

```
const unsigned int N(6) ;
int montableau[N] = { 1, 4, 2, 8, 5, 7 } ;
sort(montableau, montableau + N) ;
copy(montableau, montableau + N,
ostream_iterator<int>(cout, " ")) ;
cout << endl ;
```

Le résultat sera "1 2 4 5 7 8".

Nombres Complexes

La bibliothèque <complex> définit les nombres complexes. Ils se déclarent par complex<double>. Ils possèdent un constructeur à 2 arguments permettant de préciser les parties réelle et imaginaire, e.g.

```
complex<double> c(3.2,1.4), i(0,1) ;
```

Par contre, il n'existe pas de constructeur permettant de créer un nombre complexe à partir de ses coordonnées polaires.

En revanche, la fonction polar, qui prend comme paramètres la norme et l'argument du complexe à construire, permet de le faire. Cette fonction renvoie le nombre complexe nouvellement construit :

```
c = polar(sqrt(3.0), M_PI / 12.0) ;
```

Les méthodes des nombres complexes sont real() qui retourne la partie réelle, imag() qui retourne la partie imaginaire, et bien sûr les operateurs usuels.

Nombres Complexes

Ce qui est plus inattendu c'est que les opérations de norme, argument, et conjugaison n'ont pas été implémentées sous forme de méthodes, mais de fonctions :

double abs(const complex<double>&)	retourne la norme du nombre complexe
double norm(const complex<double>&)	retourne le carré de la norme
double arg(const complex<double>&)	retourne l'argument du nombre complexe
complex<double> conj(const complex<double>&)	retourne le complexe conjugué

La bibliothèque fournit de plus les extensions des fonctions de base (trigonométriques, logarithmes, exponentielle) aux nombres complexes.

Récapitulatif

Il existe beaucoup d'outils prédéfinis dans la bibliothèque standard de C++

Le but n'est évidemment pas les connaître tous par coeur, mais de savoir qu'ils existent pour penser aller chercher dans la documentation les informations complémentaires.

Objectifs

Apprendre à gérer les erreurs dans vos programmes :

- à trouver et corriger vos propres erreurs

  $\Longrightarrow$ utilisation d'un debugger

- à anticiper les erreurs des utilisateurs de votre programme, voire d'autres programmeurs utilisant vos fonctions.

  $\Longrightarrow$ utilisation des exceptions

Erreurs en programmation

Il existe plusieurs types d'erreurs :

- erreurs de syntaxe
- erreurs d'implémentation
- erreurs d'algorithme
- erreurs de conception

Erreurs en programmation

1. erreurs de syntaxe : le programme est mal écrit et le compilateur ne comprend pas ce qui est écrit.
   Erreurs relativement faciles à trouver : le compilateur signale le problème, indiquant souvent l'endroit de l'erreur.

2. erreurs d'implémentation : la syntaxe du programme est correcte (il compile), mais ce que fait le programme est erroné (par exemple une division par zéro se produit, ou une variable n'a pas été initialisée correctement).
   Ces erreurs ne se détectent qu'à l'exécution du programme, soit par un arrêt prématuré (cas de la division par zéro), soit par des résultats erronés (cas de la mauvaise initialisation).

Erreurs en programmation

3. erreurs d'algorithme : l'algorithme implémenté ne fait pas ce que l'on croit (ce qu'il devrait)

   assez proche du cas précédent. Mais ici, c'est plus la méthode globale qui est erronée, plutôt qu'une étourderie ou un manque de précision dans une des étapes du codage de l'algorithme.

   Il existe pour ce type d'erreurs des tests formels permettant de trouver les erreurs.

   Mais ce genre de techniques est trop complexe pour être abordé dans ce cours.

4. erreurs de conception : ici c'est carrément l'approche du problème qui est erronée, souvent en raison d'hypothèses trop fortes ou non explicitées.

   Elles relèvent du domaine de l'ingénierie informatique (le «génie logiciel»), et ne seront pas traitées dans ce cours.

Débuggeur

L'utilisation d'un « debugger » permet d'ausculter en détails l'exécution d'un programme, et en particulier

- localiser les erreurs
- exécuter un programme pas à pas
- suivre la valeur de certaines variables

Débuggeur

1. Pour pouvoir déboguer un programme, il faut le compiler avec l'option -g
   Cela indique au compilateur de rajouter des informations supplémentaires dans le programme, utiles au débuggeur.
   c++ -g -o monprogramme monprogramme.cc

2. Ensuite il faut lancer le débuggeur (gdb, ddd, ...).
   ddd monprogramme

3. Il faut ensuite exécuter le programme à corriger/étudier dans le débuggeur.
   Cela se fait souvent à l'aide de la commande run.

Débuggeur

4. On peut décider de suspendre l'exécution du programme à des endroits précis en y plaçant des breakpoints (points d'arrêt)

   b nom_de_fichier : numero_de_ligne

5. Une fois le programme stoppé à un point d'arrêt, on peut continuer à l'exécuter
   - soit pas à pas avec la commande next qui exécute les pas de programme au même niveau que le point d'arrêt (mais ne «descend» pas dans les fonctions appelées)
   - soit pas à pas avec la commande step qui exécute les pas élémentaires de programme et donc entre dans les fonctions appelées
   - soit en continu jusqu'au prochain point d'arrêt avec la commande cont

Débuggeur

6. On peut regarder le contenu d'une variable

   - soit en mettant la souris dessus (sous ddd)
   - soit à l'aide de la commande print qui affiche la valeur de la variable à ce moment là :
     print nom_variable
   - soit à l'aide de la commande display. La valeur de la variable est alors affichée à chaque pas de programme :
     display nom_variable

Gestion des exceptions

Les exceptions permettent d'anticiper les erreurs qui pourront potentiellement se produire lors de l'utilisation d'une portion de code.

Exemple :
on veut écrire une fonction qui calcule l'inverse d'un nombre réel quand c'est possible :

f
entrée : x sortie : 1/x

Si x = 0
  erreur
Sinon
  retourner 1/x

mais que faire concrètement en cas d'erreur ?

Gestion des erreurs

1. retourner une valeur choisie à l'avance :

```
double f(double x) {
    if (x != 0) return 1.0 / x ;
    else return DOUBLE_MAX ;
}
```

Mais cela :

- n'indique pas à l'utilisateur potentiel qu'il a fait une erreur
- retourne de toutes façons un résultat inexact ...
- suppose une convention arbitraire (la valeur à retourner en cas d'erreur)

Gestion des erreurs

2. afficher un message d'erreur mais que retourner effectivement en cas d'erreur ?
on retombe en partie sur le cas précédent

```
double f(double x) {
   if (x != 0) return 1.0 / x ;
   else {
      cout << "Erreur d'utilisation de la fonction f :"
           << "division par 0" « endl ;
return DOUBLE_MAX ;
   }
}
```

De plus, cela est très mauvais car produit de gros effets de bord : modifie cout alors que ce n'est pas dans le rôle de f

Pensez par exemple au cas où l'on veut utiliser f dans un programme avec une interface graphique : ouvrir une fenêtre d'alerte par exemple.

Gestion des erreurs

3. retourner un code d'erreur :

```
int f(double x, double& resultat) {
  if (x != 0) {
    resultat = 1.0 / x ;
    return PAS_D_ERREUR ; constante déjà définie
  }
  else return ERREUR_DIV_ZERO ; constante déjà définie
}
```

Cette solution est déjà mieux car elle laisse à la fonction qui appelle f le soin de décider quoi faire.

Cela présente néanmoins l'inconvénient d'être assez lourd à gérer pour finir :

- cas de l'appel d'appel d'appel .... d'appel de fonction,
- mais aussi écriture peu intuitive :

  ```
  if (f(x,y) == PAS_D_ERREUR) ...
  ```

  au lieu de

  ```
  y=f(x) ;
  ```

Les exceptions

Il existe une solution permettant de généraliser et d'assouplir cette dernière solution : déclencher une exception

$\Longrightarrow$ mécanisme permettant de prévoir une erreur à un endroit et de la gérer à un autre endroit

Principe :

- lorsque qu'une erreur a été détectée à un endroit, on la signale en «lançant» un objet contenant toutes les informations que l'on souhaite donner sur l'erreur («lancer» = créer un objet disponible pour le reste du programme)
- à l'endroit où l'on souhaite gérer l'erreur (au moins partiellement), on peut «attraper» l'objet «lancé» («attraper» = utiliser)
- si un objet «lancé» n'est pas attrapé du tout, cela provoque l'arrêt du programme : toute erreur non gérée provoque l'arrêt.

Un tel mécanisme s'appelle une exception.

Exceptions

Avantages de la gestion des exceptions par rapports aux codes d'erreurs retournés par des fonctions :

- écriture plus facile, plus intuitive et plus lisible
- la propagation de l'exceptions aux niveaux supérieurs d'appel (fonction appelant une fonction appelant ...) est fait automatiquement plus besoin de gérer obligatoirement l'erreur au niveau de la fonction appelante
- une erreur peut donc se produire à n'importe quel niveau d'appel, elle sera toujours reportée par le mécanisme de gestion des exceptions

Synthaxe

On cherche à remplir 3 tâches élémentaires :

1. signaler une erreur
2. marquer les endroits réceptifs aux erreurs
3. leur associer (à chaque endroit réceptif aux erreurs) un moyen de gérer leurs erreurs

On a donc 3 mots clés dédiés à la gestion des exceptions :

throw	indique l'erreur («lance» l'exception)
try	indique un bloc réceptif aux erreurs
catch	gère les erreurs associées (les «attrape»)

Notez bien que :

- L'indication des erreurs (throw) et leur gestion (try/catch) sont le plus souvent à des endroits bien séparés dans le code
- Chaque bloc try a son/ses catch associé(s)

throw

throw est l'instruction qui signale l'erreur au reste du programme.

Syntaxe : throw expression

l'expression peut être de tout type : c'est le résultat de son évaluation qui est «lancé» au reste du programme pour être «attrapé»

Exemples :

```
throw 21 ; // "lance" un entier
throw string("quelle erreur !") ; // "lance" une chaine de caractère
struct Erreur {
    int code ;
    string message ;
} ;
...
Erreur faute ;
...
faute.code = 12 ;
faute.message = "Division par 0" ;
throw faute ;
```

try

try introduit un bloc réceptif aux exceptions lancées par des instructions, ou des fonctions appelées à l'intérieur de ce bloc (ou même des fonctions appelées par des fonctions appelées par des fonctions ... à l'intérieur de ce bloc)

Exemple 1 :

```
try {
    ...
    if (x == 0.0) throw string("valeur nulle") ;
    ...
}
```

Exemple 2 :

```
try {
    ...
    y = f(x) ; // f pouvant lancer une exception
    ...
}
```

catch

catch est le mot-clé introduisant un bloc dédié à la gestion d'une ou plusieurs exceptions.

Tout bloc try doit toujours être suivi d'un bloc catch gérant les exceptions pouvant être lancées dans ce bloc try.

Si une exception est lancée mais n'est pas interceptée par le catch correspondant, le programme s'arrête («Aborted»).

Syntaxe :

```
catch (type nom) {
    ...
}
```

intercepte toutes les exceptions de type type lancées depuis le bloc try précédent

Remarques

- « catch(...) » permet d'intercepter n'importe quel type d'exceptions
- comme pour les fonctions, on préfèrera passer les exceptions de type complexe par références constantes :

```
catch (Erreur const& e)
```

Exemple

```
try {
    ...
    if (x == 0.0) throw string("valeur nulle") ;
    ...
    if (j >= 3) throw j ;
}

// capture les exceptions lancées sous forme de string
catch(string const& erreur) {
    cerr << "Erreur : " << erreur << endl ;
}

// capture les exceptions lancées sous forme d'int
catch(int erreur) {
    cerr << "Avertissement : je n'aurais pas du avoir la valeur "
         << erreur << endl ;
}
```

Relancement

Une exception peut être partiellement traitée par un bloc catch et attendre un traitement plus complet ultérieur (c'est-à-dire à un niveau supérieur).

Il suffit pour cela de «relancer» l'exception au niveau du bloc n'effectuant que le traitement partiel.
(Il faudra bien sûr pour cela que ce bloc catch soit lui-même dans un autre bloc try à un niveau supérieur ou un autre).

Pour «relancer» une exception, il suffit simplement d'écrire throw (sans argument)

Exemple :

```
catch (int erreur) {
    // traitement partiel :
    cerr << "Hmm... pour l'instant je ne sais pas trop "
         << "quoi faire" << endl
         << "avec l'erreur " << erreur << endl ;
    // relance l'exception reçue :
    throw ;
}
```

Exemple 1

```
#include <iostream>
#include "mesures.h"
#include "acquisition.h"
#include "plot.h"
using namespace std ;
void plot_temp_inverse(Mesures const&) ;
double inverse(double) ;
int main()
{
   Mesures temperatures ;
   acquerir_temp(temperatures) ;
   plot_temp_inverse(temperatures) ;
   return 0 ;
}
void plot_temp_inverse(Mesures const& t)
{
   for (unsigned int i(0) ; i < t.size() ; ++i)
      plot(inverse(t[i])) ;
}
double inverse(double x)
{
   return 1.0/x ;
}
```

Exemple 1

```
...
using namespace std ;

const int DIVZERO(33) ;

void plot_temp_inverse(Mesures const&) ;
double inverse(double) ;

int main()
{
   Mesures temperatures ;
   acquerir_temp(temperatures) ;
   plot_temp_inverse(temperatures) ;
   return 0 ;
}

void plot_temp_inverse(Mesures const& t)
{
   for (unsigned int i(0) ; i < t.size() ; ++i)
      plot(inverse(t[i])) ;
}

double inverse(double x)
{
   if (x == 0.0) throw DIVZERO ;
   return 1.0/x ;
}
```

Exemple 1

```
...
int main()
{
   Mesures temperatures ;
   acquerir_temp(temperatures) ;
   try {
      plot_temp_inverse(temperatures) ;
   }
   catch (int i) {
      if (i == DIVZERO) {
         cerr << "Courbe des températures erronée" <<endl ;
         // on fait quelque chose, par exemple refaire les mesures,
         // mais à ce stade le programme n'est pas stoppé
         //..
      }
   }
   return 0 ;
}

...
```

Exemple 1

```
...
int main() {
   ...
   try {
      plot_temp_inverse(temperatures) ;
   }
   catch (int i) {
      if (i == DIVZERO) {
         cerr << "Courbe des températures erronée" <<endl ;
         // effectue ici un traitement de plus haut niveau
         ...
   }
...
}
void plot_temp_inverse(Mesures const& t)
{
   for (unsigned int i(0) ; i < t.size() ; ++i) {
      try {
         plot(inverse(t[i])) ;
      }
      catch (int j) {
         /* Traiter partiellement le problème et relancer l'exception. *
         * Cette partie du programme peut par exemple signaler *
         * l'indice de la valeur erronée. */
         cerr << "problème avec la valeur " << i << endl ;
         throw ;
      }
   }
```

Exemple Complet

```
#include <iostream>
#include "mesures.h"
#include "acquisition.h"
#include "plot.h"
using namespace std ;
const int DIVZERO(33) ;
void plot_temp_inverse(Mesures const&) ;
double inverse(double) ;
int main()
{
   Mesures temperatures ;
   unsigned int const MAX_ESSAIS(2) ;
   unsigned int nb_essais(0) ;
   bool restart(false) ;
   do {
      ++nb_essais ; restart=false ;
      acquerir_temp(temperatures) ;
      try {
         plot_temp_inverse(temperatures) ;
      }
```

Exemple Complet

```
        catch (int i) {
            if (i == DIVZERO) {
                if (nb_essais < MAX_ESSAIS) {
                    cout << "Il faut re-saisir les valeurs" << endl ;
                    restart = true ;
                } else {
                    cout << "Il y a déjà eu au moins " << MAX_ESSAIS << " essais."
                    cout << " -> abandon" << endl ;
                }
            } else {
                cout << "Ne sais pas quoi faire -> abandon" << endl ;
            }
        }
    } while (restart) ;
    return 0 ;
}
```

Exemple Complet

```
void plot_temp_inverse(Mesures const& t)
{
   for (unsigned int i(0) ; i < t.size() ; ++i) {
   // Exemple de traitement local partiel du problème (ce n'est pas obligatoire).
      try {
         plot(inverse(t[i])) ;
      }
      catch (int j) {
         cerr << "Erreur : " ;
         if (j == DIVZERO) {
            cerr << "la valeur " << i << " est nulle." ;
         } else {
            cerr << "problème avec la valeur " << i ;
         }
         cerr << endl ;
         throw ;
      }
   }
}
double inverse(double x)
{
   if (x == 0.0) throw DIVZERO ;
   return 1.0/x ;

}
```

Spécification des exceptions

Il est toujours bon en programmation d'être le plus explicite possible sur ce que fait chaque bloc, et chaque fonction.

Si une fonction peut lancer des exceptions, il est donc sage de l'indiquer.

Cela se fait au niveau du prototype en ajoutant

throw(type1, type2, ...)

derrière le prototype de la fonction.

Cela ne fait rien de particulier, mais indique simplement que la fonction peut lancer des exceptions de types type1 ou type2.

Exemple : double f(double) throw(Erreur) ;

Remarque : throw(), tout seul sans argument, indique que la fonction ne déclenche pas d'exceptions

Exception lancée par new

new (allocation dynamique de pointeur), retourne une exception de type bad_alloc si l'allocation dynamique ne se passe pas correctement.

Il est donc conseillé d'écrire par exemple :

```
#include <new>
try {
    ...
    ptr = new ... ;
    ...
}
catch (std : : bad_alloc const& e) {
    cerr << "Erreur : plus assez de mémoire !" << endl ;
    exit 1 ;
}
```

Récapitulatif

throw expression ; lance l'exception définie par l'expression

try { ... } introduit un bloc sensible aux exceptions

catch (type nom) { ... } bloc de gestion de l'exception

Tout bloc try doit toujours être suivit d'un bloc catch gérant les exceptions pouvant être lancées dans ce bloc try.

Si une exception est lancée mais n'est pas interceptée par le catch correspondant, le programme s'arrête.

Références

1. D. Sam-Haroud, Cours Informatique III, Laboratoire d'Intelligence Artificielle, Ecole Polytechnique Fédérale de Lausanne.
2. S. Meyers, Effective C++, Third edition, Addison Wesley Professional Computing Series.
3. S. Meyers, Effective STL, Addison Wesley Professional Computing Series.
4. H. Garreta, Le Langage C++, Faculté des Sciences de Luminy.