Séance 8
Patrons – Exceptions

Patrons : but

Ensemble d'entiers, sans doublon, stockés dans un tableau de taille fixe :

```
class Ensemble
{
public :
   Ensemble();
   void ajouter(int val);
   bool contient(int val) const;
   void afficher() const;

private :
   int elem[100];
   int nb;
};
```

```
Ensemble::Ensemble() {
   nb = 0;
}


void Ensemble::ajouter(int val) {
   if (!contient(val) {
      elem[nb] = val;
      nb++;
   }
}
```

```
bool Ensemble::contient(int val) const {
   for (int i = 0; i < nb; i++)
      if (val == elem[i])
         return true;
   return false;
}

void Ensemble::afficher() const {
   for (int i = 0; i < nb; i++)
      cout << elem[i] << " ";
   cout << endl;
}
```

But : sur ce modèle, créer des classes ensemble de tout type (float, String, Complexe, …).

Ecriture d'un patron

Patron (modèle) pour des classes d'ensemble de tout type T :

```
template <class T>
class Ensemble
{
public :
   Ensemble();
   void ajouter(T val);
   bool contient(T val) const;
   void afficher() const;

private :
   T elem[100];
   int nb;
};
```

```
template <class T>
Ensemble<T>::Ensemble() {
   nb = 0;
}

template <class T>
void Ensemble<T>::ajouter(T val) {
   if (!contient(val) {
      elem[nb] = val;
      nb++;
   }
}
```

etc

tout mettre dans un .h : déclaration de la classe + contenu des fonctions

Utilisation d'un patron

```
Ensemble<int> E1;
E1.ajouter(10);

Ensemble<int> E2;


Ensemble<string> E3;
string s1("bonjour");
E3.ajouter(s1);


Ensemble<Complexe> E4;    ERREUR
```

les classes Ensemble<int> et Ensemble<string> sont créées automatiquement à partir du patron Ensemble

Complexe ne possède pas certains opérateurs utilisés par le patron Ensemble (ex. : <<)

Exceptions : but

Mécanisme d'exception

Ecrire du code (appel de fonctions membres, …) sans se soucier des cas d'erreur :

- pas de test des retours des fonctions,
- sur arrivée d'une erreur, déroutement automatique vers la séquence de gestion des erreurs.

Exemple :

En C : débordement d'un entier short => valeur tronquée

si short sur 16 bits : [-32768, +32767]

short x = *40000*; x vaut *-25536*
short a = *-20000*, b = *-30000*; short c = a + b; c vaut *15536*

A faire : utiliser le mécanisme d'exception pour signaler et gérer les débordements
=> créer et utiliser une classe SHORT : SHORT x(40000);

Générer une exception : throw

```
class SHORT
{
public :
   SHORT(long v);
   …
private :
   short val;
};
```

```
SHORT::SHORT(long v) {
   if (v > 32767) {
      // générer une exception
      Deborde deb1(v, "débord….positif");
      throw deb1;
   }
   else if (v < -32768) {
      // générer une exception
      Deborde deb2(v, "débord….négatif");
      throw deb2;
   }
   else
      val = (short) v;
}
```

throw : analogue à return (sortie fonction, envoi résultat à l'appelant)

```
class Deborde
{
public :
   Deborde(long vd, const char* m) {
      vdeb = vd;    strcpy(msg, m);
   }
   long getval() const    { return vdeb; }
   void afficherMsg() const    { … }

private :
   long vdeb;          // valeur du débordement
   char msg[100];  // msg d'erreur
};
```

but : contenir des infos sur l'exception

Capturer une exception : try/catch

```
main
…
…
try
{
   …
   SHORT S(…);
   …
}
catch (Deborde ex)
{
   cout << ex.getval() …
   …
}
// suite de main
…
```

ex = copie de l'objet envoyé par throw

code de traitement de l'exception

Si exception :
1. déroutement vers catch
2. exécution de catch
3. suite de main

une exception entraine la non continuation du try

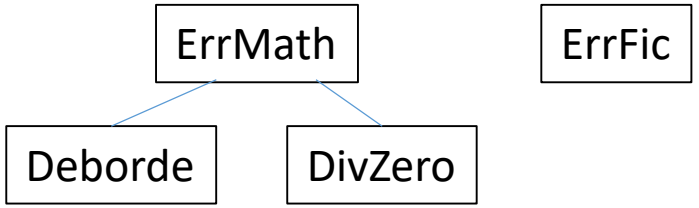
Si pas exception :
1. try exécuté entièrement
2. suite de main

try/catch : propriétés

Un try peut être suivi de plusieurs catch

```
try
{
    …
}
catch (Deborde ex)
{
    …
}
catch (DivZero ex)
{
    …
}
```

On peut regrouper les catch (par l'héritage)




```
try {
  …
}
catch (Deborde ex) {
  …
}
catch (DivZero ex) {
  …
}
catch (ErrFic ex) {
  …
}
```

⇨

```
try {
  …
}
catch (ErrMath ex) {
  …
}
catch (ErrFic ex) {
  …
}
```

capture les exceptions Deborde et DivZero