Notions avancées en C++

IUT Lyon 1

Yosra ZGUIRA

yosra.zguira@insa-lyon.fr

2016 - 2017

Les templates

Principe (1/2)

- Les **templates** (modèles, patrons en Français) sont un élément important de la généricité en C++.
- Il permettent de créer automatiquement des modèles génériques de **fonctions** ou de **classes** qui pourront s'adapter (se paramétrer) pour plusieurs types de données.
- **Le patron de fonction**, par exemple, sert à généraliser un même algorithme, une même méthode de traitement, à différents cas ou types de données.
- Le fait de fournir un paramètre à un modèle générique s'appelle la **spécialisation**.
- La spécialisation d'un template est transparente et invisible.

Principe (2/2)

- Elle est effectuée **lors de la compilation**, de manière interne au compilateur, en fonction des arguments donnés au template (il n'y a pas de code source généré quelque part).

- Le mot clé **template** est suivi de la liste des paramètres du patron entre les signes < et >, suivi de la définition de la fonction ou de la classe.

```
template <typename T>
```

- L'instanciation d'un patron permet la création effective d'une fonction ou d'une classe.

Patron de fonction (1/3)

- Exemple avec un patron de fonction calculant la valeur maximale d'un tableau de données (le type étant le paramètre du patron de fonction) :

```
template<typename T>
 T max(T array[], int length) {

      T vmax = array[0];
      for (int i = 1; i < length; i++)
            if (array[i] > vmax)
                  vmax = array[i];

       return vmax;
}
```

- Le mot-clé **template** indique que c'est une fonction template.
- Le mot-clé **typemap** est un type de données.

Patron de fonction (2/3)

- Exemple d'utilisation:

```
int values[]={ 71, 22, 43, 65, 102, 91, 23, 114 };

cout << max(values, 8);
```

- Le compilateur crée une fonction **max** à partir du type des arguments, en remplaçant le paramètre **T** par le type **int**.

- Après remplacement le compilateur génère la fonction suivante :

Patron de fonction (3/3)

```
int max(int array[], int length) {

      int vmax = array[0];
      for (int i = 1; i < length; i++)
            if (array[i] > vmax)
                  vmax = array[i];

       return vmax;
}
```

- Si on utilise la fonction **max** avec différents types (int, double, ...), le compilateur générera autant de fonctions.
- Les patrons permettent donc d'éviter d'écrire plusieurs fonctions pour chaque type de donnée traité.
- Quand on sait que la duplication inutile de code est une source d'erreur, on comprend l'intérêt de **mettre en facteur plusieurs fonctions** potentielles dans un même patron de fonction.

Patron de classe (1/5)

- La déclaration d'un **patron de classe** utilise une syntaxe similaire que celle de fonction.

- Les classes templates ont des membres (attributs et méthodes) qui utilisent des paramètres template.

- Exemple d'une classe gérant un tableau de données :

Patron de classe (2/5)

```
template<typename T, int maxsize>
class CDataArray {

    public:
        CDataArray();
        int getSize();
        T get(int index);
        bool add(T element);      // true si l'élément a pu être ajouté

    private:
        T array[maxsize];
        int length;
};
```

Patron de classe (3/5)

- L'implémentation du constructeur et des méthodes de la classe exige une syntaxe un peu plus complexe.

- Par exemple, pour le constructeur :

```
template<typename T,int maxsize>                    //patron de classe
CDataArray<T,maxsize>::CDataArray() : length(0) // Liste d'initialisation des données membres
{}
```

➔ Le nom du constructeur est précédé du nom de la classe suivi des paramètres du patron de classe, afin que le compilateur détermine de quelle classe il s'agit.

Patron de classe (4/5)

- Les méthodes sont implémentées de manière identique :

```
template<typename T,int maxsize>
int CDataArray<T,maxsize>::getSize() {

return length;

}

template<typename T,int maxsize>
T CDataArray<T,maxsize>::get(int index) {

    return array[index];
}
```

```
template<typename T,int maxsize>
bool CDataArray<T,maxsize>::add(T element)
{

    if (length>=maxsize)
        return false;
        array[length++]=element;
        return true;
 }
```

Patron de classe (5/5)

- Contrairement aux patrons de fonctions, l'instanciation d'un patron de classe exige la présence de la valeur des paramètres à la suite du nom de la classe.

```
CDataArray<int,100> listeNumeros;

listeNumeros.add(8);

listeNumeros.add(12);

 cout << listeNumeros.getSize() << endl;
```

- Le compilateur génère une classe pour chaque ensemble de valeurs de paramètres d'instanciation différent.

La bibliothèque standard (STL)

Introduction (1/2)

- La **STL (Standard Template Library)** est une bibliothèque C++ normalisée, puissante et pratique.
- La STL a été mise au point par Alexander Stepanov et Meng Lee.
- La STL a été proposée au comité ISO de standardisation du C++ qui l'a acceptée en juillet 1994.
- Les résultats des travaux de recherche ont été publiés officiellement dans un rapport technique en novembre 1995.
- Ces travaux de recherche ont été une avancée majeure pour le C++, qui était aussi à l'époque le seul langage capable d'offrir les mécanismes de programmation générique nécessaires à la mise au point de cette bibliothèque.

Introduction (2/2)

- Elle a d'ailleurs influencé les autres parties de la future bibliothèque du C++ (notamment la future classe string) et aussi l'évolution du langage.

- Elle fournit les fonctionnalités suivantes :

  ✓ des **conteneurs:** ce sont des classes qui gèrent des structures de données évoluées : vecteurs, listes chaînées, ...

  ✓ des **itérateurs**: ils fournissent un moyen simple et générique de parcourir les conteneurs. C'est une généralisation du concept de pointeur.

  ✓ des **algorithmes génériques:** ce sont les algorithmes classiques de l'algorithmique tels que les algorithmes de remplissage, recherche, tri, . . .

  ✓ une classe **string** efficace.

Les conteneurs (1/15)

- En c++, **les conteneurs** sont des classes offrant au programmeur une implémentation permettant de gérer des **collections dynamiques d'objets du même type** , c'est à dire pour lesquels le nombre d'objets contenus peut varier à l'exécution.

- Un conteneur est un objet permettant de stocker d'autres objets.

- De plus les conteneurs sont conçus de manière à être compatible avec les algorithmes de la bibliothèque standard.

- Les conteneurs sont fournis par l'espace de nom **std**.

Les conteneurs (2/15)

- Les différents conteneurs peuvent être partagés en deux catégories selon que les éléments sont classés dans la mémoire à la suite les uns des autres ou non.

  ➔ On parle dans un cas de **séquences** et dans l'autre de **conteneurs associatifs.**

- La liste de tous les conteneurs de la STL triés suivant leur catégorie:

Séquences :

- ✓ vector
- ✓ deque
- ✓ list
- ✓ stack
- ✓ queue
- ✓ priority_queue

Conteneurs associatifs :

- ✓ set
- ✓ multiset
- ✓ map
- ✓ multimap

Les conteneurs (3/15)

- Pour utiliser ces conteneurs, il faut inclure dans le fichier d'entête:

```
#include <nom_conteneur>
```

- Pour utiliser des **list**, il faut ajouter cette ligne à votre code:

```
#include <list>
```

Les conteneurs (4/15)

❑ **Méthodes communes des conteneurs:**

- Les concepteurs de la STL ont donné les mêmes noms aux méthodes communes de tous les conteneurs.

- Par exemple, la méthode **size()** renvoie la taille d'un **vector**, d'une **list** ou d'une **map**.

- La méthode **empty()** qui renvoie **true** si le conteneur est vide et **false** sinon.

- La méthode **clear()** qui permet de vider le conteneur.

- La méthode **swap()** qui permet d'échanger le contenu de deux conteneurs de même type au lieu de faire la copie à la main.

```
vector<int> a(5,8);  //Un vector contenant 5 fois le nombre 8
vector<int> b(8,2);  //Un vector contenant 8 fois le nombre 2
a.swap(b);
```

Les conteneurs (5/15)

❑ Les séquences et leurs adaptateurs:

❖vector:

- **vector** est un tableau dynamique où il est particulièrement aisé d'accéder directement aux divers éléments par un index et d'en ajouter ou en retirer à la fin.

- On accède aux éléments via les crochets **[]**, comme pour les tableaux statiques.

Méthode	Description
push_back()	Ajout d'un élément à la fin du tableau.
pop_back()	Suppression de la dernière case du tableau.
front()	Accès à la première case du tableau.
back()	Accès à la dernière case du tableau.
assign()	Modification du contenu d'un tableau.

En pratique, en C++ on utilisera **std::vector** pour remplacer avantageusement une déclaration sous forme de tableau dynamique sous la forme:

type* var=new type[N]

Les conteneurs (6/15)

❖**deque("Double ended queue", file à double entrée) :**

- **deque** est un tableau auquel on peut ajouter des éléments aux deux extrémités.

- Les vector proposent les méthodes push_back() et pop_back() pour manipuler ce qui se trouve à la fin du tableau ce qui rend impossible la modification de ce qui se trouve au début.

- Les **deque** lèvent cette limitation en proposant des méthodes **push_front()** et **pop_front()**.

Les conteneurs (7/15)

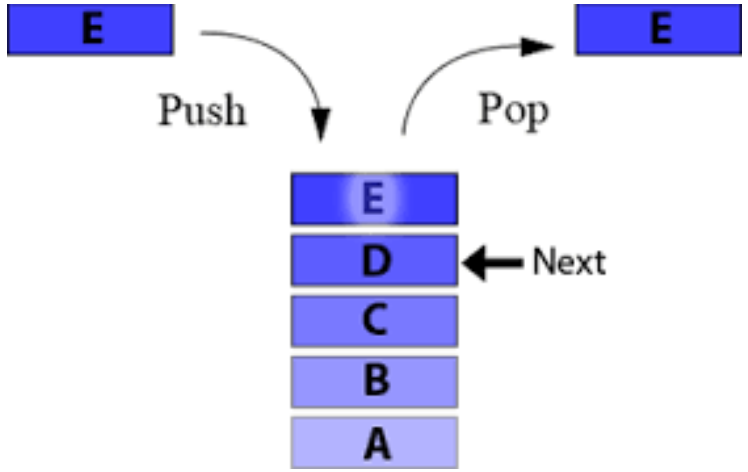
❖**stack:**

- **stack** implémente une interface de pile (LIFO, Last In First Out : dernier arrivé, premier sorti).

- **La pile** est un conteneur qui n'autorise l'accès qu'au dernier élément ajouté.

- Il n'y a que 3 opérations autorisées :

  - ✓ Ajouter un élément
  - ✓ Consulter le dernier élément ajouté
  - ✓ Supprimer le dernier élément ajouté



Les conteneurs (8/15)

❖**stack:**

- Exemple:

```
#include <stack>
#include <iostream>
using namespace std;

int main()
{
   stack<int> pile;        //Une pile vide
   pile.push(6);          //On ajoute le nombre 6 à la pile
   pile.push(5);
   pile.push(4);

   cout << pile.top() << endl;        //On consulte le sommet de la pile (le nombre 4)

   pile.pop();                    //On supprime le dernier élément ajouté (le nombre 4)

   cout << pile.top() << endl;         //On consulte le sommet de la pile (le nombre 5)

   return 0;
}
```

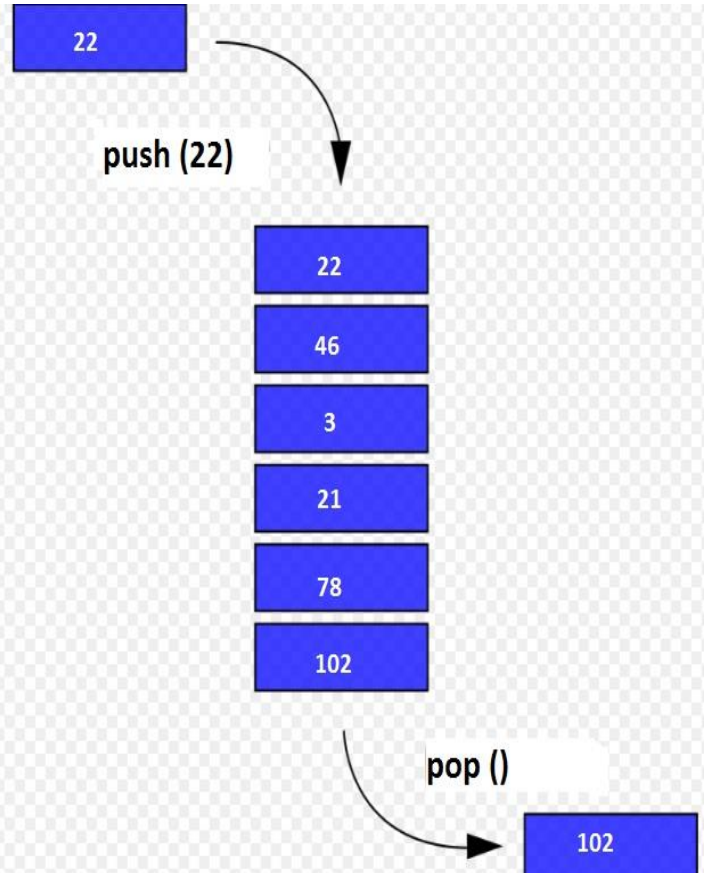
Les conteneurs (9/15)

❖**queue:**

- **queue** implémente une interface de file d'attente (FIFO, First In First Out : premier arrivé premier sorti).

- La différence par rapport aux piles est que l'on ne peut accéder qu'au premier élément ajouté.

- On utilise **front()** pour accéder à ce qui se trouve à l'avant de la file au lieu de **top()**.



Les conteneurs (10/15)

❖**priority_queue:**

- **priority_queue** implémente une interface de file d'attente où les éléments peuvent être comparés entre eux (par niveau de *priorité*).
- Les éléments sont classés dans la file suivant l'ordre spécifié.
- Elles permettent de traiter des données suivant des niveaux de priorité de manière efficace.
- Les méthodes sont exactement les mêmes que dans le cas des files simples.
- Il est défini dans le même fichier que la file simple: **#include <queue>**

Les conteneurs (11/15)

❖priority_queue:

- Exemple:

```
#include <queue>
#include <iostream>
using namespace std;

int main()
{
   priority_queue<int> file;
   file.push(1);
   file.push(2);
   file.push(3);

   cout << file.top() << endl;  //Affiche le plus grand des éléments insérés (le nombre 3)

   return 0;
}
```

Les conteneurs (12/15)

❖**list:**

- **list** est une liste doublement chainée.

- L'insertion et la suppression d'élément ou de groupes continus d'éléments est efficace partout dans la liste, mais il n'est pas possible d'accéder directement aux différents éléments.

- Il est forcé de les parcourir avec les itérateurs.

Les conteneurs (13/15)

❑ Les conteneurs associatifs:

- Dans un vector ou une deque, les éléments sont accessibles via leur **index**, un nombre entier positif.

- Les conteneurs associatifs sont des structures de données qui autorisent l'emploi de n'importe quel type comme index.

- Les conteneurs associatifs supposent donc l'utilisation d'une "clé de recherche" (un numéro d'identification, ou des chaines classées par ordre alphabétique par exemple) et implémentent des fonctions efficaces de tri et de recherche.

Les conteneurs (14/15)

❖map:

- **map** est une table associative permettant de stocker des paires **clé-valeur**.

```
#include <map>
#include <string>
using namespace std;

map<string, int> b;
```

- On déclare une table associative qui stocke des entiers mais dont les indices sont des chaînes de caractères.

- On accède à un élément via les crochets [] comme ceci :

```
b["bonjour"] = 2;
```

Les conteneurs (15/15)

❖set:

- Les **set** sont utilisés pour représenter les ensembles.

- On peut insérer des objets dans l'ensemble et y accéder via une méthode de recherche. Par contre, il n'est pas possible d'y accéder via les crochets.

❖multiset , multimap:

- Les **multiset** et **multimap** sont des copies des set et map où chaque clé peut exister en plusieurs exemplaires.

Les itérateurs (1/8)

- Les **itérateurs** permettent d'itérer sur les objets d'un conteneur, c'est-à-dire d'en parcourir le contenu en passant par tous ses objets.

- Un itérateur est comparable à un pointeur sur un élément d'un conteneur.

- Les itérateurs sont des pointeurs spéciaux permettant le déplacement dans les conteneurs comme le ferait un pointeur dans la mémoire.

- Un itérateur permet à la fois d'accéder à un élément d'une séquence, d'aller à l'élément suivant dans une séquence et de tester l'atteinte du début ou de la fin d'une séquence.

- Il existe deux types d'itérateurs :

  - ✓ **iterator** ou **const iterator** : parcours d'un conteneur du début à la fin

  - ✓ **reverse iterator** ou **const reverse iterator** : parcours d'un conteneur en sens inverse

Les itérateurs (2/8)

- Un itérateur associé à un conteneur (vecteur, liste, map ..) se déclare de la manière suivante :

```
vector<int> tableau(6,2);          //Un tableau de 6 entiers valant 2
vector<int>::iterator it;          //Un itérateur sur un vector d'entiers

map<string, int>::iterator it1;    //Un itérateur sur les tables associatives string-int

deque<char>::iterator it2;         //Un itérateur sur une deque de caractères

list<double>::iterator it3;        //Un itérateur sur une liste de nombres à virgule
```

Les itérateurs (3/8)

- Des méthodes génériques de conteneurs permettent de gérer les itérateurs :

  - ✓ **begin()**: retourne un itérateur qui pointe sur le premier élément.
  - ✓ **end()**: retourne un itérateur qui pointe juste "après" le dernier élément. Il faut donc itérer jusqu'à end() exclu.
  - ✓ **++**: permet d'incrémenter l'itérateur en le faisant passer à l'élément suivant.
  - ✓ **insert():** permet d'insérer un élément dans un conteneur.
  - ✓ **erase():** permet de supprimer un élément dans un conteneur.

Les itérateurs (4/8)

- Exemple 1:

```
#include<vector>
#include <iostream>
using namespace std;

int main()
{
   vector<int> Tab(6,2);                    //Un vector de 6 éléments valant 2
   vector<int>::iterator it;                //Un itérateur sur un vector d'entiers

   for(it = Tab.begin(); it!=Tab.end(); ++it)     //On itère sur le vector
   {
       cout << *it << endl;                 //On accède à l'élément pointé via l'étoile
   }
    return 0;
}
```

- Les itérateurs ne sont pas optimisés pour l'opérateur de comparaison.
- On ne devrait donc pas écrire **it<d.end()** comme on en a l'habitude avec les index de tableau.
- Utiliser != est plus efficace.

Les itérateurs (5/8)

- Exemple 2:

```
#include <vector>
#include <string>
#include <iostream>
using namespace std;

int main()
{
   vector<string> Tab;                          //Un tableau de mots
   vector<string>::iterator it;

   Tab.push_back("au");                          //On ajoute trois mots dans le tableau
   Tab.push_back("cours");
   Tab.push_back("C++");

   Tab.insert(tab.begin(), "Bienvenu");       //On insère le mot "Bienvenu" au début
```

Les itérateurs (6/8)

- Exemple 2 (suite):

```
for(it=Tab.begin(); it!=Tab.end(); ++it)    //Affiche la chaîne "Bienvenu au cours C++"
   {
      cout << *it << " ";
   }

 Tab.erase(tab.begin());                    //On supprime le premier mot

for(it=tab.begin(); it!=tab.end(); ++it)    //Affiche la chaîne "au cours C++"
   {
      cout << *it << " ";
   }

   return 0;
}
```

Les itérateurs (7/8)

❑ **Les différents itérateurs:**

- Ils existent cinq catégories d'itérateurs, seuls deux sont utilisés pour les conteneurs: les bidirectional iterators et les random access iterators.

❖**Bidirectional iterators:**

- Ce sont des itérateurs qui permettent d'avancer et de reculer sur le conteneur.
- On peut avancer et reculer un seul élément via les opérateurs **++** et **--**.
- Pour accéder au septième élément d'un conteneur, il faut partir de la position **begin()** puis appeler six fois l'opérateur **++**.
- Ce sont les itérateurs utilisés par les list, map et set.

Les itérateurs (8/8)

❖**Random access iterators:**

- Ces itérateurs permettent d'accéder directement au milieu d'un conteneur.

- Ils proposent de nouveaux opérateurs **+** et **–** permettant d'avancer de plusieurs éléments d'un seul coup.

```
vector<int> Tab(50,2);                          //Un tableau de 50 entiers valant 2

vector<int>::iterator it = Tab.begin() + 9;     //Un itérateur sur le 9ème élément
```

Les algorithmes

- **Les algorithmes de la STL** sont des fonctions permettant d'effectuer des traitements sur des données.

- Les principaux algorithmes sont :

  - ✓ **les algorithmes numériques**: minimum, maximum, sommes partielles, produits
  - ✓ **les algorithmes de tri**
  - ✓ **les algorithmes modifiant les conteneurs** : remplissage, copie, échanges, union, intersection,
  - ✓ **les algorithmes ne modifiant pas les conteneurs** : recherche, ...

Les algorithmes

- Pour les utiliser, il faut ajouter dans l'entête de vos codes:

```
#include <algorithm>
```

- Les algorithmes les plus classiques : **count**, **find** et **sort**.

- Tous travaillent sur une plage du conteneur [debut, fin[.

- Exemple:

```
#include <algorithm>
#include <vector>

std::vector <int> tab(3);        // tableau de 3 entiers
tab.push_back(1) ;
tab.push_back(2) ;
tab.push_back(3) ;
```

Les algorithmes

- **count()** compte le nombre d'occurrences de val dans la plage du conteneur :

```
size_t count(iterator debut, iterator fin, const T& val);

size_t nb = count(tab.begin(), tab.end(), 2) ;      // retourne 1
```

- **find()** cherche la première instance de la valeur **val** dans le conteneur, en partant du début jusqu'à la fin.

find() retourne un itérateur qui pointe à l'emplacement où val a été trouvée. Si non trouvé, find retourne fin :

```
iterator find(iterator debut, iterator end, const T& val) ;

std::vector<int>::iterator it ;
it = find(tab.begin(), tab.end(), 2) ;
```

Les algorithmes

- **sort()** trie les éléments du conteneur en utilisant l'opérateur **<** (ordre croissant).

```
void sort(iterator debut, iterator fin) ;

sort(tab.begin(), tab.end());
```

Les algorithmes

❑ **Les algorithmes de séquence non modifiants:**

- for_each
- find
- find_if
- find_end
- find_first_of
- adjacent_find
- count
- count_if
- mismatch
- equal
- search
- search_n

Les algorithmes

❑ **Les algorithmes de séquence modifiants:**

- Copies
  - ✓ copy
  - ✓ copy_backward
- échanges
  - ✓ swap
  - ✓ swap_ranges
  - ✓ iter_swap
- transformations
  - ✓ transform
- remplacements
  - ✓ replace
  - ✓ replace_if
  - ✓ replace_copy
  - ✓ replace_copy_if
- remplissages
  - ✓ fill
  - ✓ fill_n
- générations
  - ✓ generate
  - ✓ generate_n

Les algorithmes

❑ **Les algorithmes de séquence modifiants (suite):**

- suppressions
  - ✓ remove
  - ✓ remove_if
  - ✓ remove_copy
  - ✓ remove_copy_if
- éléments uniques
  - ✓ unique
  - ✓ unique_copy
- ordre inverse
  - ✓ reverse
  - ✓ reverse_copy
- rotations
  - ✓ rotate
  - ✓ rotate_copy
- permutations aléatoires
  - ✓ random_shuffle
- Répartitions
  - ✓ partition
  - ✓ stable_partition

Les algorithmes

❑ **Les algorithmes de tri et les opérations apparentés:**

- tris
  - ✓ sort
  - ✓ stable_sort
  - ✓ partial_sort
  - ✓ partial_sort_copy
  - ✓ nth_element
- recherches dichotomiques
  - ✓ lower_bound
  - ✓ upper_bound
  - ✓ equal_range
  - ✓ binary_search
- fusions
  - ✓ merge
  - ✓ inplace_merge
- opérations d'ensemble
  - ✓ includes
  - ✓ set_union
  - ✓ set_intersection
  - ✓ set_difference
  - ✓ set_symmetric_difference

Les exceptions

Principe (1/4)

- Une exception est l'interruption de l'exécution du programme à la suite d'un événement particulier.
- Le but des exceptions est de réaliser des traitements spécifiques aux événements qui en sont la cause.
- Ces traitements peuvent rétablir le programme dans son mode de fonctionnement normal, auquel cas son exécution reprend.
- Il se peut aussi que le programme se termine, si aucun traitement n'est approprié.

Principe (2/4)

- Le C++ supporte les **exceptions logicielles** dont le but est de gérer les erreurs qui surviennent lors de l'exécution des programmes.

- Lorsqu'une telle erreur survient, le programme doit lancer une exception.

- L'exécution normale du programme s'arrête dès que l'exception est lancée et le contrôle est passé à un gestionnaire d'exception.

- Lorsqu'un gestionnaire d'exception s'exécute, on dit qu'il a **attrapé** l'exception.

Principe (3/4)

- Les exceptions permettent une gestion simplifiée des erreurs, parce qu'elles en reportent le traitement.
- Le code peut alors être écrit sans se soucier des cas particuliers, ce qui le simplifie grandement.
- Les cas particuliers sont traités dans les gestionnaires d'exception.

Principe (4/4)

- Les mot-clés pour traiter les exceptions sont :

  ✓ **try{...}** est le bloc de code dans lequel une erreur peut survenir.

  ✓ **throw** lance l'exception (déclenche le traitement de l'exception), en lançant un objet.

  ✓ **catch(...) {...}** est le bloc de code (handler ou gestionnaire d'exception) qui attrape l'exception, récupère l'objet lancé et gère l'erreur comme le veut le programmeur.

  ➔ Dans un bloc **try**, on détecte un problème, donc on lance avec **throw** une exception qui va être traitée dans le bloc **catch**.

Fonctionnement (1/2)

- Une fois l'exception lancée elle remonte l'arbre d'appel des fonctions, jusqu'à être attrapée par une fonction active.

Toutes les fonctions traversées qui n'attrapent pas l'exception sont dépilées sans être terminées : en général, une fonction qui détecte une erreur d'exécution ne peut pas se terminer normalement.

- Si une exception traverse toutes les fonctions actives sans être attrapée, elle entraîne la terminaison du programme (crash brutal).

- Une interruption attrapée peut être relancée à nouveau (throw) pour continuer à la faire remonter.

Fonctionnement (2/2)

- L'objet lancé peut être de n'importe quel type et doit pouvoir bien caractériser l'exception.

Cela peut être par exemple une chaîne de caractères décrivant l'erreur, un numéro d'erreur, l'heure à laquelle est survenue l'erreur, . . .

Lancer une exception

- Lancer une exception consiste à retourner une erreur sous la forme d'une valeur (message, code, objet exception) dont le type peut être quelconque (int, char*, MyExceptionClass, ...).

- Le lancement se fait par l'instruction **throw** :

```
throw 0;
```

Attraper une exception (1/2)

- Pour attraper une exception, il faut qu'un bloc encadre l'instruction directement, ou indirectement, dans la fonction même ou dans la fonction appelante, ou à un niveau supérieur.

- Dans le cas contraire, le système récupère l'exception et met fin au programme.

- Les instructions **try** et **catch** sont utilisées pour attraper les exceptions.

```
try {
     ...                    // code lançant une exception (appel de fonction, ...)
 }
catch (int code)
{
     cerr << "Exception " << code << endl;
}
```

Attraper une exception (2/2)

- Exemple:

```
int division(int a, int b)
{
      if (b == 0) {
           throw 0;          //lancer l'exception: division par zéro
      } else
          return a / b;
}

void main()
{
    try  {
        cout << "5/0 = " << division(5, 0) << endl;
     }
     catch (int code)
     {
           cerr << "Exception " << code << endl;
     }
}
```

Attraper toutes les exceptions

- Il est possible de définir un **gestionnaire d'exception** (i.e un catch) **universel**, qui récupérera toutes les exceptions possibles, quel que soit leur type.

- Ce gestionnaire d'exception doit prendre comme paramètre **trois points de suspension** entre parenthèses dans sa clause catch.

- Bien entendu, dans ce cas, il est impossible de spécifier une variable qui contient l'exception, puisque son type est indéfini.

```
try {
    ...
}
catch (...)        // on ne peut pas récupérer l'objet ici (exception inconnue)
{
    cerr << "Exception inattendue !!" << endl;
}
```

Déclarer des exceptions lancées (1/3)

- La déclaration d'une fonction lançant un **nombre limité de type d'exception**, telle que la fonction division de l'exemple précédent, peut être suivie d'une liste de ces types d'exceptions dans une clause **throw** :

```
int division(int a, int b) throw(int)
 {
      if (b == 0)
          throw 0;          // division par zéro
      else
          return a / b;
}
```

Déclarer des exceptions lancées (2/3)

- Par défaut, la fonction peut lancer n'importe quel type d'exception.

- La déclaration de la fonction **division** sans clause throw est donc équivalent à la déclaration suivante :

```
int division(int a, int b) throw(…)      //n'importe quel type d'exception
 {
      if (b == 0)
          throw 0;          // division par zéro
      else
          return a / b;
}
```

Déclarer des exceptions lancées (3/3)

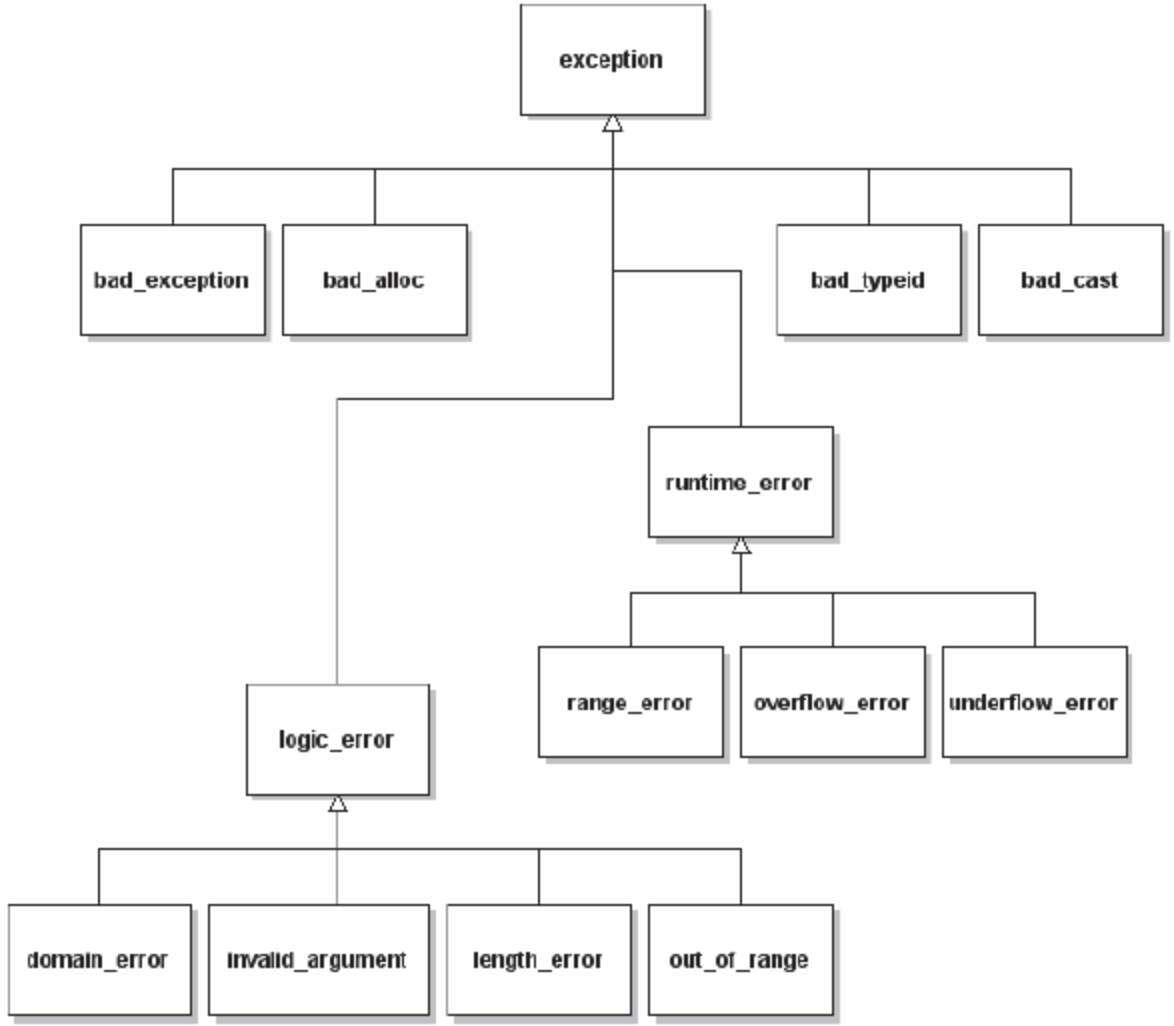
- Si une fonction ne lance aucune exception, on peut la déclarer avec une clause throw vide :

```
int soustraction(int a, int b) throw()      //aucune exception
{
    return a-b ;
}
```

Exceptions standards

- C++ définit des exceptions standards, dans **<stdexcept>**, que l'on peut utiliser.

- **Exemples:** la classe mère **std::exception** et une de ses classes filles **std::bad_alloc** sont lancées en cas de problème par **new.**



Les casts

Introduction

- Les deux langages C et C++ partagent en effet bien des fonctionnalités… mais ont également de grandes différences.

- Parmi ces différences, on trouve les **opérateurs de conversion C++** permettant de convertir des types d'objets (variable, pointeur, référence, etc).

- On peut distinguer quatre types de conversion possibles et réalisables en C++ :

  ✓ La conversion **statique** de types (**static_cast**)

  ✓ La totale **ré-interprétation** des données d'un type vers un autre (**reinterpret_cast**)

  ✓ La conversion d'un pointeur (ou référence) **constant(e)** vers un pointeur (ou référence) non-constant(e) (**const_cast**)

  ✓ La conversion de types **dynamique** (**dynamic_cast**)

static_cast (1/3)

- **Le cast statique** permet d'expliciter les conversions de types implicites évitant tout avertissement que donnerait le compilateur si la conversion peut entraîner un risque.

  Exemple: double vers int.

- Il permet de caster des types de même famille (les pointeurs, les références, etc).

- Ils existent certaines conversions entre types de même famille (comme de *double* vers float** par exemple) qui ne sont pas réalisables avec un simple **static_cast**.

- **static_cast<>** est l'opérateur de conversion le plus utilisé.

- Il ne permet que de réaliser des conversions sûres.

static_cast (2/3)

- Exemple1: convertir vers et depuis n'importe quel type pointé à partir d'un **void***.

```
void* x;
long* y = x;
```

```
long* y = static_cast<long*>(x);
```

- void*: un pointeur à void dont le type pointé est inconnu.
- On ne devrait jamais avoir à utiliser void* en C++ mais c'est juste pour l'exemple.

➔ **Ce code ne compile pas !**

- Indiquer le type de destination entre "<>" suivi de la variable à caster **entre parenthèses**.

static_cast (3/3)

- Exemple2:

```
double a;
float b = static_cast<float>(a);
```

- Dans cet exemple, une conversion implicite suffit mais le compilateur cracherait au moins un warning sans static_cast.

➔ **Il faut bien expliciter vos conversions implicites en utilisant static_cast.**

reinterpret_cast (1/3)

- Il s'agit de **la ré-interprétation des données.**

- Son rôle est de dire au compilateur : *"réinterprète-moi la représentation binaire de ce type en tant qu'un autre type".*

- Il permet :

  ✓ de convertir **n'importe quel** type pointé en un autre, même lorsque ceux-ci n'ont **aucun rapport**. Exemple: **int*** vers **double***.

  ✓ de convertir un type pointé en sa **représentation intégrale** et vice et versa.
  Exemple: **int*** vers **int**.

reinterpret_cast (2/3)

- Par exemple, avec un **static_cast**, il est impossible de convertir *un double* vers un float**.*

- La solution serait de **ré-interpréter les données stockées par les pointeurs**.

➔ On devrait pouvoir récupérer la valeur hexadécimale stockée par la variable *double** et la considérer comme l'adresse d'une variable *float*.

```
double* d;
float* f = reinterpret_cast<float*>(d);
```

reinterpret_cast (3/3)

- **reinterpret_cast** ne se limite pas aux types de même famille.

- Il est possible de réaliser des cast entre certains types de familles différentes et cela peut parfois s'avérer très pratique.

const_cast (1/2)

- Il s'agit **des cast de pointeurs sur constante**.

- **Exemple:** on a un pointeur (ou une référence) sur constante et on ne peut pas modifier l'élément pointé (car celui-ci est protégé par le const).

```
int x = 20;
const int& ref = x;
ref = 30;
```

- La référence **ref** est déclarée "sur constante", donc il n'y a pas moyen de modifier **x** en passant par **ref**.

```
int x = 20;
const int& ref = x;
int& ref2 = ref;
```

- Il est donc également **impossible** d'assigner le contenu de cette référence à une référence du même type (ref2) mais non-"sur constante".

const_cast (2/2)

- Le but d'utiliser le **const** est d'assurer la sécurité et c'est très pratique dans certain cas de pouvoir ainsi empêcher le programmeur de toucher à certaines choses.

  ➔ Mais, il existe bien une solution pour cracker cette sécurité, en utilisant **const_cast**.

```
int x = 20;
const  int&  ref  =  x;
int&  ref2  = const_cast<int&>(ref);
```

- Ce type de cast permet de supprimer les attributs const.

- **const_cast** ne fonctionne que sur des pointeurs ou des références et n'est pas fait pour modifier la valeur d'une variable constante d'un type d'une autre famille.

  ➔ **On ne devrait jamais avoir à utiliser const_cast dans un programme à moins de savoir exactement ce que l'on fait. Cela peut devenir dangereux.**

dynamic_cast (1/4)

- **Une conversion de type dynamique** est une conversion qui va s'effectuer *pendant l'exécution de votre programme* et non par le compilateur comme c'est le cas pour les trois autres cast existants.

- Il s'agit toujours de types personnalisés (définir à l'aide des classes).

- Le fonctionnement est un petit peu plus délicat que celui des autres cast.

- Exemple:
  - on a une classe mère **polygone** et d'une classe dérivée **carre** (carré).
  - **carre** hérite donc de **polygone** car tout carré est un polygone.

dynamic_cast (2/4)

- Supposons qu'on dispose d'une référence sur un objet **carre** et qu'on aimerait considérer ce carré comme un polygone en copiant cette référence vers une référence sur un **polygone**.

- C'est possible car **un carré est un polygone**.

```
#include <iostream>

class polygone
{
   public :
   virtual void f() {}
};

class carre : public polygone {};
```

```
int main()
{
   carre monCarre;
   carre& r_carre = monCarre;    // référence sur un objet carre

   try {
      polygone& r_polygone = dynamic_cast<polygone&>(r_carre);
   }
   catch (const std::exception& e)
   {
      std::cerr << e.what();
   }
   return EXIT_SUCCESS;
}
```

dynamic_cast (3/4)

- Pour une question de **polymorphisme**, il faut que la classe mère possède au moins une fonction virtuelle.

- Si on compile le précédent code, on remarque qu'il fonctionne bien, aucune exception n'est lancée.

- Par contre, si on essaie d'inverser les types (remplacer carre par polygone et inversement), on aura l'exception **std::bad_cast** car un polygone n'est pas forcément un carré et donc on ne peut pas considérer tout polygone comme étant un carré.

- **Aussi, il est également impossible de caster avec un *dynamic_cast* entre deux classes dérivant d'une même classe (un pentagone n'est pas un carré).**

dynamic_cast (4/4)

- Exemple2:

```
int main()
{
   carre monCarre;
   polygone& r_polygone = monCarre;

   try {
      carre& r_carre = dynamic_cast<carre&>(r_polygone);
   }
   catch (const std::exception& e)
   {
      std::cerr << e.what();
   }
   return EXIT_SUCCESS;
}
```

- On utilise bien **dynamic_cast** d'une classe mère vers une classe fille.

- La seule condition pour que cela fonctionne: **l'affectation d'un objet de type carre à la référence de type polygone&.**

➔ Dès le début du programme, on sait que le polygone "pointé" par r_polygone est un carré donc un cast vers une référence sur un carre est possible.