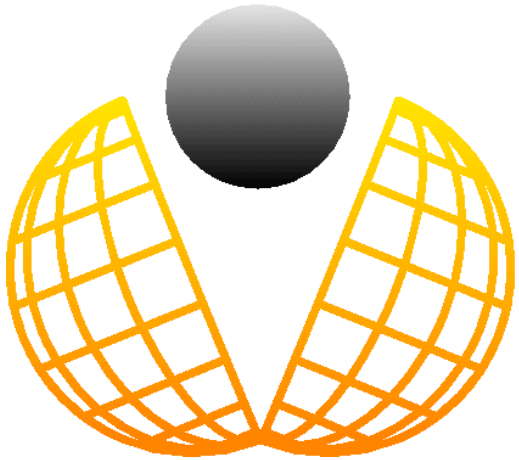
Bibliothèque standard C++
Partie 1/2

ISIMA - ZZ2 – 2011



Bruno Garcia
Loic Yon
et David Hill

Evolution de la programmation

Le but de la manipulation est d'écrire un programme qui affichera "HELLO

```
BASIC au Lycée

   10 PRINT "HELLO WORLD"

   20 END
```

En PREPA, DEUG, BTS ou DUT

Mon bon Blaise ☺

```
program HELLO(input, output)
      begin
         writeln('HELLO WORLD')
      end.
```

En 1ère année d'Ecole d'Ingénieur…

LISP, Scheme et Cie…

```
(defun HELLO
      (print
        (cons 'HELLO (list 'WORLD))
       )
 ).
```

ZZ1 Essai sur tableau de pointeurs…

C la vie…

```
#include <stdio.h>

int main(int argc, char ** argv)

{

  char * message[] = {"HELLO ", "WORLD"};
  int    i;

  for(i = 0; i < 2; ++i)
  {
     printf("%s", message[i]);
  }
  printf("\n");
}
```

C++ en deuxième année

Étudiant ZZ2 « très expérimenté… »

```
#include <iostream.h>
#include <string.h>

class string
{
   private:

      int size;
      char *ptr;

   public:

      string() : size(0), ptr(new char('\0')) {}
      string(const string &s) : size(s.size)
      {
         ptr = new char[size + 1];
         strcpy(ptr, s.ptr);
      }

      ~string()
      {
         delete [] ptr;
      }

      friend ostream &operator <<(ostream &, const string &);

      string &operator=(const char *);

};
```

La suite…

```
ostream &operator<<(ostream &stream, const string &s)
{
   return(stream << s.ptr);
}

string & string::operator=(const char *chrs)
{
   if (this != &chrs)
   {
      delete [] ptr;
      size = strlen(chrs);
      ptr = new char[size + 1];
      strcpy(ptr, chrs);
   }

   return(*this);
}


// et enfin…

int main(int, char **)
{
    string str;

    str = "HELLO WORLD";
    cout << str << endl;

    return(0);
}
```

Genèse de la STL

- Demande de la communauté C++
  - Pas de bibliothèque de classes conteneur
    - Chacun développe sa bibliothèque dans son coin
    - Réutilisabilité nulle
    - Apprentissage nécessaire à chaque fois
    - Fiabilité douteuse
  - Pas de classe chaîne de caractères
    - Mêmes conséquences
- L'existant
  - Bibliothèque ADA
  - Non orientée objet

Résultat : Librairie Standard du C++ ex STL

Contenu :

- Classes
  - string
  - conteneurs
    - de base (vector, deque, list)
    - spécialisés (stack, queue, priority_queue)
    - associatifs (set, map)
  - iostream revu et corrigé
  - utilitaires
- fonctions
  - algorithmes qui travaillent sur les conteneurs génériques grâce à la notion d'itérateur

Notion d'itérateur

Définition :

Balise localisant un emplacement dans une collection

Vous en utilisez déja : pointeurs dans tableau

```
int   tableau[10];
int   niveau = 10;
int * courant= tableau; // &tab[0]
int * fin    = tableau+niveau;


while (courant != fin)
{
  // action sur *courant, ex
  cout << *courant << endl;
  ++courant;
}
```

Vous comprennez ce code ?
Alors, vous savez utiliser la STL

Notion d'itérateur BIS

Tout itérateur est isomorphe à un pointeur dans un tableau

Même code avec STL :

```
#include <vector>
using namespace std;
typedef vector<int> VecInt;

VecInt v(10);
VecInt::iterator courant=v.begin();
VecInt::iterator fin=v.end();

while (courant != fin)
{ // Travail sur l'élément courant
  cout << *courant << endl;
  ++ courant;
}
```

Dissection du code précédent

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

Notion de namespace

- Elimine les collisions de noms de classes
- Ex 2 classes Matrice dans 2 bibliothèques

Avec namespace : préfixe de nommage

- Ex : stats::Matrice ou pde::Matrice
- Suppression du préfixe :
  - Classe/fonction isolée: `using stats::Matrice;`
  - Tout l'espace : `using namespace stats;`

Particularité : pas de .h dans les headers de la librairie standard du C++

Dissection du code (le retour)

```
typedef vector<int> IntVector;
```

Généricité : tous les types sont template

Les itérateurs sont en sous classe des conteneurs

Souvent :

```
typedef IntVector::iterator IntVectorIt ;
```

Les classes fondamentales

La classe string

Encore une classe template ... sur le type char de base !

Supporte toutes les operations de base avec les opérateurs classiques

Conversion vers et depuis char *

Les conteneurs fondamentaux

Trois classes de base

- vector
  - Modélise un vecteur à croissance dynamique
  - Operations en bout de vecteur et acces direct en O(1) amorti
- deque
  - Liste spécialisée dans les opérations aux 2 bouts
  - Acces direct en O(log (n)) aux 2 bouts en O(1)
- list
  - Liste doublement chaîne circulaire classique
  - Toute insertion / deletion en O(1)
  - Accès directe en O(n)

Les specialisées !

Basées sur une collection de base mais avec operations spécifiques

Collections :

- stack
  - objet pile (pop, push et top)
- queue
  - objet file (pop, push, front et back)
- priority_queue
  - file à propriété, implémentée sous la forme d'un tax minimax
  - pop, push, top

Les conteneurs associatifs

Utilisent une clef (paire, valeur)

Deux types et 2 catégories :

- set (clef et valeur confondues)
- map (clef et valeur distinctes)
- set et map : une seule valeur par clef
- multiset et multimap : plusieurs valeurs autorisées par clef

Méthodes les plus courantes

Méthode	Action
`iterator begin()`	Premier élément du vecteur
`iterator end()`	Après le dernier élément du vecteur
`int size()`	Nb d'éléments présents dans le vecteur
`int capacity()`	Capacité d'accueil actuelle du vecteur
`void push_back(const T& élem)`	Ajoute un élément au bout du vecteur
`void pop_back()`	Retire l'élément au bout du vecteur
`void push_front(const T& elem)`	Ajoute un élément au début du vecteur
`void pop_front()`	Retire l'élément de début du vecteur
`const T& front()`	Ref sur l'élément en tête de vecteur
`const T& back()`	Ref sur l'élément en fin de vecteur
`empty()`	True si vecteur vide
`T& operator[](int idx)`	Ref sur l'élément d'index idx
`const T& operator[] (int i) const`	Idem mais en version constante

Opérations avec itérateurs

Méthode	Action
`void erase (iterator it)`	Supprime l'élément spécifié
`void erase (iterator debut, iterator fin)`	Supprime les éléments [`debut`, `fin`[
`void insert(iterator place, const T& elem)`	Insère `elem` à l'emplacement `place`
`void insert(iterator place, iterator debut, iterator fin)`	Insère à la position `place`, les éléments de [`debut`, `fin`[

Les algorithmes

- Ensemble d'opérations communes
  - Copie d'éléments entre conteneurs
    - Ecrasement
    - Insertion
  - Transfert d'éléments entre conteneurs
  - Recherche d'éléments
- Fonctions plutôt que méthodes
  - STL initialement non orientée objet
  - Utilise abondamment les itérateurs

Algorithmes courants

Copie d'éléments

```
copy (debut, fin, destination);
```

Attention ! remplace les éléments !

Pour ajouter des éléments :

```
copy(debut, fin, inserter(destination));
```

Recherche d'éléments

```
place = find (debut, fin, element);
```

Affichage d'une collection

```
ostream_iterator<Type> oi(cout, "  ");
copy(debut, fin, oi);
```

La STL & les Exceptions en C++

- Lancement d'exceptions
  - Par fonctions ou méthodes
  - Théoriquement tout type de données
  - Habituellement des classes spécialisées
- Traitement des exceptions
  - Blocs de codes surveillés
  - Gestionnaires d'exceptions : code spécialisé

Une exception non traitée entraîne l'arrêt du programme

Exemple

```
class Chaine
{
  public:
    class ExceptionBornes {};
  …
    char &operator[](int index)
    {
      if ((index < 0) || (index > taille_))
      {
        throw ExceptionsBornes();
      }
      return tab_[index];
    }
};
```

Lancement de l'exception c'est un **objet**

```
try
{
  c=chaine[3];
}
catch (Chaine::ExceptionBornes &e)
{
  // traitement
}
```

Début du bloc d'instructions surveillées

Traite exceptions

Type d'exception traitée

Le type des exceptions

- Éviter d'encombrer l'espace de nommage
  - Classes imbriquées
- Exceptions hiérarchisées
- Librairie standard du C++

```
class exception // throw() sans param. ne lance par d'excpt.
{
  public:
    exception() throw();
    exception(const exception& rhs) throw();
    exception& operator=(const exception& rhs) throw();
    virtual ~exception() throw();
    virtual const char *what() const throw();
};
```

- Bon comportement dériver de `exception`

```
#include <exception>
```

Exemple de hiérarchies d'exceptions

```
class Vecteur
{
  public:
  class ExptVecteur : public exception
  {
    public:
      const char *what() const throw()
      {
        return "Exception de vecteur";
      }
  };
  class ExptVecteurBornes : public Vecteur::ExptVecteur
  {
    // …
  };
  class ExptVecteurBorneInf : public Vecteur:ExptVecteurBornes …
  class ExptVecteurBorneSup : public Vecteur:ExptVecteurBornes …
  class ExptVecteurAllocation : public Vecteur::ExptVecteur
  {
    // …
  };
};
```

Traitement des hiérarchies d'exception

Toujours traiter les exceptions les plus spécialisées d'abord

Utiliser un objet passé par référence pour éviter une recopie

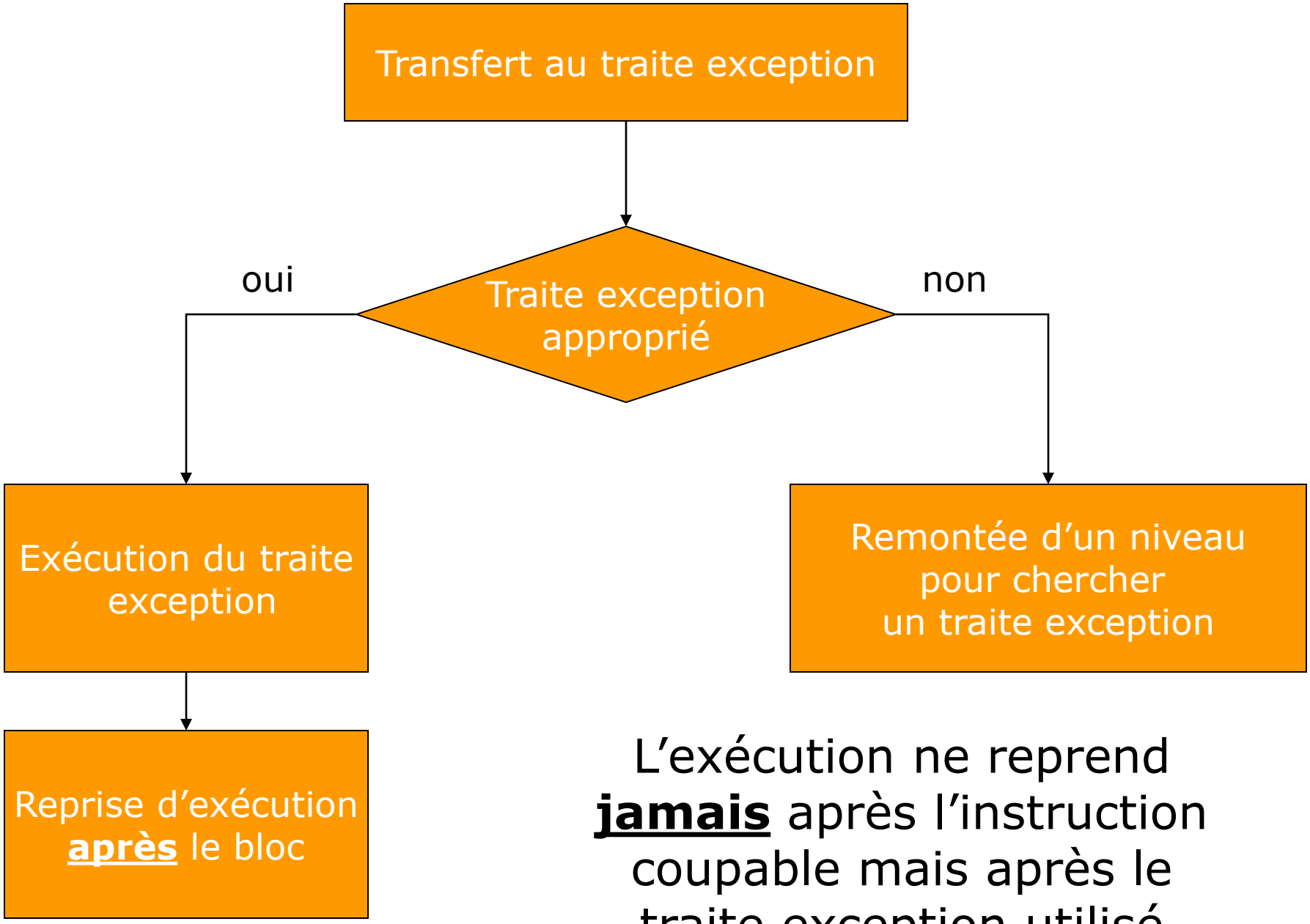
Prévoir un traitement pour chaque `exception`

Et un gestionnaire spécialisé `catch(…)` qui ramasse « tout ce qui traîne »

Exemple de traitement des hiérarchies

```
try
{
 // Code susceptible de lancer une exception
}
catch (Vecteur::ExptVecteurAllocation &e)
{
  // Traitement exception spécialisée ExptVecteurAllocation
}
catch (Vecteur::ExptVecteurBorneInf &e)
{
  // Traitement exception spécialisée ExptVecteurAllocation
}
catch (Vecteur::ExptVecteur &e)
{
  // Traitement exception plus générale ExptVecteur
}
catch (exception)
{
  // Traitement sommet hiérarchie
}
catch (...)
{ /* Fourre tout */ }
```

Que se passe t ’il lorsqu’une exception est levée ?



L’exécution ne reprend **jamais** après l’instruction coupable mais après le traite exception utilisé

Relancer une exception

- Pourquoi ?
  - Traite exception incapable d'assurer le retour à la normale
- Conséquences
  - Traitement d'une même exception à plusieurs niveaux
  - Terminaison éventuelle du programme
- Syntaxe
  - Très simple : `throw;`

La procédure `terminate`

Met fin au programme lorsqu'une exception n'a pas été traitée

Ne ferme pas les fichiers ni ne libère les ressources

Prévoir les libérations de ressources dans chaque classe ...

Où prévoir une alternative à l'aide de la fonction `set_terminate`

Prototype :

```
void fRemplacement(void);
```

Spécificateurs d'exceptions

But :

- Définir l'ensemble d'exceptions que l'utilisateur peut s'attendre à voir lever par une méthode ou une fonction

Difficulté

- Difficile à définir car le spécificateur doit prévoir tous les sous appels !

Si une exception non prévue est levée :

- Appel de la procédure `unexpected`
- Par défaut celle-ci appelle `terminate`
- Possible de redéfinir ce comportement (`set_unexpected` similaire à `set_terminate`)

Syntaxe des spécificateurs d'exceptions

Syntaxe :

```
Typeretour ident(params) throw (liste);
```

Exemple :

```
char & Chaine::operator[](int index) throw (Vecteur::ExptVecteurBornes);
```

Remarques :

- Spécification de `ExptVecteurBornes`
- Inclut les sous classes
- Les méthodes propres des classes dérivées de `exception` ne sont pas sensées pouvoir lever une exception