ENPC – MOPSI

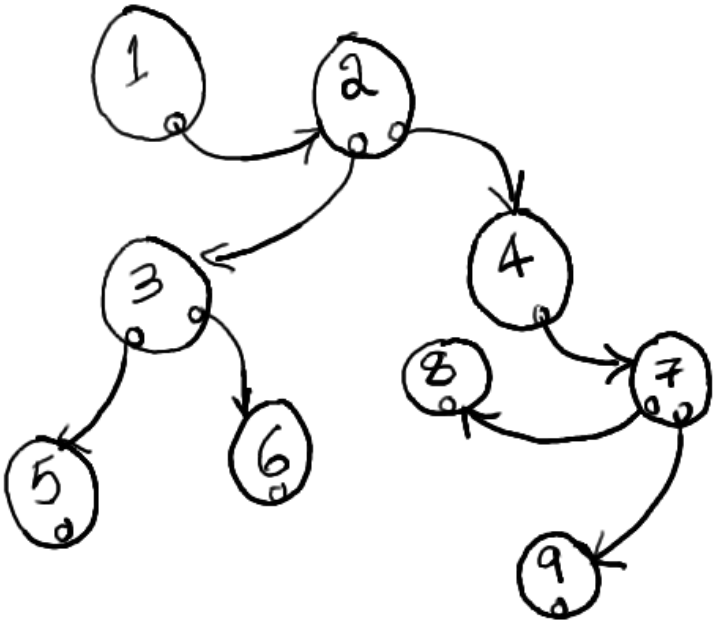
Notions de structures de données

Renaud Marlet
Laboratoire LIGM-IMAGINE

http://imagine.enpc.fr/~marletr

Structure de données

- Organisation de l'information
  - structure logique
  - moyens d'accès
- Exemples :
  - tableau
  - vecteur, matrice
  - liste, file, pile
  - arbre, graphe
  - table de hachage, ...



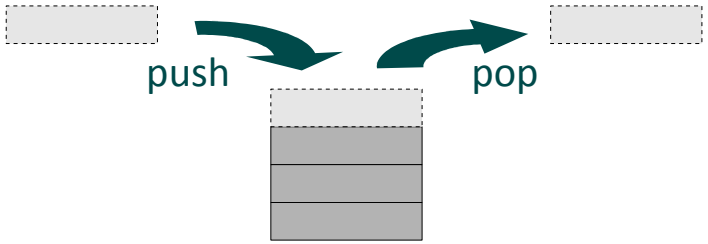
Pas d'informatique sans structures de données

- Omniprésent
  - derrière presque tout ce qui est discret (vs continu) ou tout ce qui est composite
  - simulations et calculs complexes
    - ex. physique (maillages...)
    - ex. finance (réseaux bayésiens...)

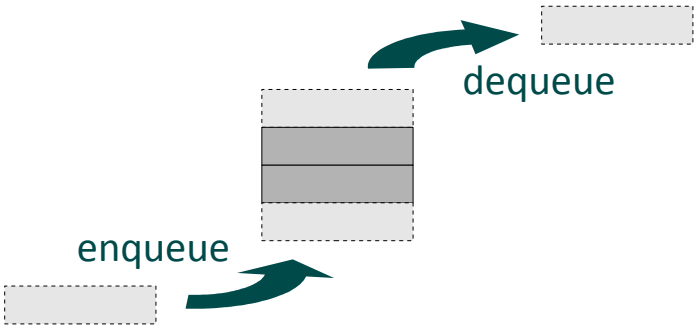
- Aspects théoriques
  - complexité : pire cas, en moyenne...
- Aspects pratiques
  - bibliothèques logicielles : STL, Imagine++, Boost, Eigen…

Type abstrait : données + opérations

- Exemple 1 : pile (stack) [Last In First Out, LIFO]
  - vide?
  - empiler (push)
  - dépiler (pop)



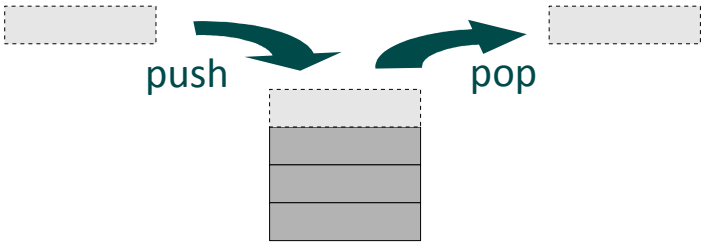
- Exemple 2 : file (queue) [First In First Out, FIFO]
  - vide?
  - ajouter (enqueue)
  - extraire (dequeue)



Type abstrait : données + opérations

- Implémentation avec langage orienté objet : classe
  - données : champs/variables d'instance (+ allocations)
  - opérations : méthodes/fonctions membres

- Exemple : pile (stack)



  - les données :
    - valeurs rangées par ordre d'arrivée, p.ex. stockées dans un tableau
  - les opérations :
    - accès aux données via des fonctions isEmpty(), push(val), pop()

Exemple d'implémentation (simple) d'une pile d'entiers

```
class stack {
    int elem[20];
    int level;
public:
    bool isEmpty() { return level == 0; }
    void push(int i) { elem[level++] = i; }
    int  pop() { return elem[--level]; }
};

int main() { // Exemple d'usage
    stack s;
    s.push(1);
    s.push(2);
    cout << s.pop() <<" "<< s.pop() << endl;
    // Affiche : 2 1
```

Exemple d'implémentation (simple)

```
class stack {
    int elem[20];
    int level;
public:
    bool isEmpty() { return level == 0; }
    void push(int i) { elem[level++] = i; }
    int  pop() { return elem[--level]; }
};

int main() { // Exemple d'usage
    stack s;
    s.push(1);
    s.push(2);
    cout << s.pop() <<" "<< s.pop() << endl;
    // Affiche : 2 1
```

Quels sont les défauts ?
(au moins 4)

Exemple d'implémentation (simple)

```
class stack {
    int elem[20]; // taille fixe
    int level;  // pas initialisé
public:
    bool isEmpty() { return level == 0; }
    void push(int i) { elem[level++] = i; } // overflow
    int  pop() { return elem[--level]; } // underflow
};

int main() { // Exemple d'usage
    stack s;
    s.push(1);
    s.push(2);
    cout << s.pop() <<" "<< s.pop() << endl;
    // Affiche : 2 1
```

Quels sont les défauts ?
(au moins 4)

Exemple d'implémentation (simple)

```
class stack {
    int elem[20]; // taille fixe
    int level;  // pas initialisé
public:
    bool isEmpty() { return level == 0; }
    void push(int i) { elem[level++] = i; } // overflow
    int  pop() { return elem[--level]; } // underflow
};

int main() { // Exemple d'usage
    stack s;
    s.push(1);
    s.push(2);
    cout << s.pop() <<" "<< s.pop() << endl;
    // Affiche : 2 1
```

Comment les corriger ?

Exemple d'implémentation (simple)

```
class stack {
    int elem[20]; // taille fixe → allocation + redimensionnement dynamiques
    int level;  // pas initialisé → initialisation (p. ex. dans un constructeur)
public:
    bool isEmpty() { return level == 0; }
    void push(int i) { elem[level++] = i; } // overflow → resize
    int  pop() { return elem[--level]; } // underflow → tester
}; // + ajout d'un destructeur pour libérer la mémoire dynamique

int main() { // Exemple d'usage
    stack s;
    s.push(1);
    s.push(2);
    cout << s.pop() <<" "<< s.pop() << endl;
    // Affiche : 2 1
```

Comment les corriger ?

Exemple d'implémentation (robuste, plus complète, plus sûre)

```
class stack {
    int *elem;
    int level, size;
public:
    stack() : level(0), size(20) { elem = new int[size]; }
    ~stack() { delete[] elem; }
    bool isEmpty() { return level == 0; }
    void push(int i) {
        if (level == size) {      // Redimensionner si la pile est pleine :
            int newSize = size + 10;              // Augmenter la taille
            int *newElem = new int[newSize];      // Allouer plus grand
            for (int j=0; j < size; j++)          // Copier l'ancienne mémoire
                newElem[j]=elem[j];               //    dans la nouvelle
            delete[] elem;                        // Libérer l'ancienne mémoire
            size = newSize;                       // Mettre à jour la taille
            elem = newElem; }                     // Utiliser la nouvelle mém.
        elem[level++] = i; }      // Toujours de la place pour empiler
    int pop() {
        if (level == 0) return 0;// Rend une valeur par défaut si pile vide
        return elem[--level]; }   // Jamais d'accès hors du tableau
};
```

En résumé

- Constructeur
  - pensez à toutes les initialisations
- Destructeur
  - avec destruction récursive éventuelle des contenus
- Tests de débordement
  - supérieurs (overflow), inférieurs (underflow)
- Gestion des cas d'erreur

Parenthèse sur la gestion des cas d'erreur

- Taille/complexité croissante des systèmes
  [mesure courante : "(source) lines of code" = (S)LOC → KLOC, MLOC...]
  - noyau Linux : 5 MLOC (2003), 16 MLOC (2012)
  - distribution Debian : 55 MLOC (2000), 324 MLOC (2009)
- **Pas de programme sans erreur**

  Attention, estimation très grossière !
  - programme « ordinaire » : ≈ 1 erreur pour 100 LOC
    - davantage encore pour de gros programmes, plus complexes
  - navette spatiale US (à bord) : 1 erreur pour 420.000 LOC
- Les erreurs sont la norme, **il faut savoir les gérer**
  - Ariane 5 : 10 ans de travail, 1 milliard €, crash au premier lancement (1996) pour une gestion d'erreur non activée

Qu'est-ce qu'une erreur ?

- Terminologie IEEE
  - **anomalie :** toute chose qui dévie de ce qui est prévu
  - **bogue :** faute qui cause un comportement non voulu ou non anticipé
  - **erreur :** écart entre une valeur calculée/observée et sa valeur théorique ; souvent aussi synonyme de faute
  - **faute :** ensemble d'instructions incorrectes qui cause un comportement non voulu ou non anticipé
- Mais dans l'usage, les termes restent flous/ambigus

Erreur manifeste vs erreur silencieuse

- Erreur manifeste « incompatible » avec l'exécution
  - ex. déréférence d'un pointeur nul, accès à zone interdite
  - effet : segmentation fault (segfault, bus error...)
  - fin brutale de l'exécution
  - données en cours de traitement perdues
- Erreur silencieuse « compatible » avec l'exécution
  - ex. lire/écrire hors des limites d'un tableau si zone permise
  - le programme continue avec une valeur aberrante
  - il se plante/trompe plus loin → bogue très difficile à trouver
- Observable (ou non) : pas de résultat, résultat faux

Stratégies de gestion d'erreur
(forme de tolérance aux fautes)

- Détection explicite de cas d'anomalie
  - cela suppose qu'on a une idée de ce qui est attendu
    - plage de valeurs, type d'objet, cohérence entre valeurs...
  - exhaustivité impossible, et peu souhaitable (risques d'erreur dans la gestion d'erreur !) → limiter à cas plausibles
  - rattrapage de cas d'erreur génériques
- Réactions
  - signalement ou non, à qui ?
  - poursuite de l'exécution avec valeur plausible (par défaut)
  - poursuite de l'exécution avec traitement spécial alternatif
  - ou arrêt de l'exécution

Traitement simple : détecter, avertir, sauvegarder, terminer

```
int pop()
{
  if (level <= 0) { // Error condition
    cout << "Trying to pop from empty stack" << endl;
    savePreciousData();
    exit(17);         // Error code for popping from empty stack
  }
  return elem[--level] ;
}
```

- message spécifique à l'erreur
- sauvegarde éventuelle des données importantes
- terminaison propre
- diagnostic possible à l'extérieur du programme

Quelles sont les limitations ?

Traitement simple : détecter, avertir, sauvegarder, terminer

```
int pop()
{
  if (level <= 0) { // Error condition
    cout << "Trying to pop from empty stack" << endl;
    savePreciousData();
    exit(17);         // Error code for popping from empty stack
  }
  return elem[--level] ;
}
```

- message spécifique à l'erreur, mais lecteur = développeur de `stack`, utilisateur de `stack` ou utilisateur final ?
- sauvegarde éventuelle des données importantes, mais souvent pas accessibles depuis `stack`, ni même connues
- terminaison propre, mais brutale
- diagnostic possible à l'extérieur du programme mais difficile

Messages d'erreur

- À l'attention d'un utilisateur interne (développeur)
  - canal de sortie spécifique pour erreurs : `cerr` (≠ `cout`)

```
cerr << "Trying to pop from empty stack" << endl;
```

  - écriture dans des fichiers de log
    - log = journal de bord (toutes activités, pas seulement les erreurs)
- À l'attention d'un utilisateur final
  - boîtes de dialogue
- Penser à l'internationalisation :
  - développeur → anglais | utilisateur final → choix de langue
  - templates avec trous, spécifiques à chaque langue

Terminer proprement

- Terminaison « propre »
  - fermeture des fichiers ouverts (au moins un flush)
  - effacement des fichiers temporaires
  - contrôle rendu à l'environnement hôte
- void exit(int status)

```
#include <stdlib.h>
...
exit(1);
```

  - le statut permet un diagnostic en dehors du programme
    - 0 ou EXIT_SUCCESS : exécution réussie
    - 1 ou EXIT_FAILURE : échec en cours d'exécution
    - toute valeur différente de 0 : différentes formes/raisons d'échec

Garantir de terminer gracieusement

- int atexit (void (*function) (void))

```
#include <stdlib.h>
void cleanUp1 (void) { cout << "Doing cleanup 1"; … }
void cleanUp2 (void) { cout << "Doing cleanup 2"; … }
int main ()
{
    atexit (cleanUp1);
    atexit (cleanUp2);
    cout << "Doing main work"; ...
    return 0;
}
```

- Appel (en ordre inverse) des fonctions enregistrées
  - Doing main work, Doing cleanup 2, Doing cleanup 1

Assertion : Gestion d'erreur conditionnelle

```
#include <assert.h>
...
assert(level > 0);        // invariant: condition à satisfaire
return elem[--level];
```

- En phase de mise au point
  - condition toujours testée : plus sûr, mais plus lent
  - en cas d'insatisfaction, message et terminaison (brutale)
    - assertion failed: *expression*, file *filename*, line *line number*
- Une fois le code mis au point (ex. à la livraison)
  - élimination des tests : pas sûr, mais plus rapide
    - #define NDEBUG avant l'inclusion de assert.h
    - ligne de commande du compilateur : /DNDEBUG, -DNDEBUG

Traitement simple : détecter, avertir, sauvegarder, terminer

```
int pop()
{
  if (level <= 0) { // Error condition
    cout << "Trying to pop from empty stack" << endl;
    savePreciousData();
    exit(17);         // Error code for popping from empty stack
  }
  return elem[--level] ;
}
```

- message spécifique à l'erreur, mais lecteur = développeur de `stack`, utilisateur de `stack` ou utilisateur final ?
- sauvegarde éventuelle des données importantes, mais souvent pas accessibles depuis `stack`, ni même connues

☛ Meilleure situation dans l'appelant (plus de contexte, accès aux données) → y faire là le traitement d'erreur

Mécanismes de signalement d'erreur et rattrapage associé

- Retour d'une valeur impossible
  - peut être testée et un comportement approprié choisi
- Positionnement d'une variable spécifique
  - peut être testée et un comportement approprié choisi
- Retour d'une valeur par défaut (forme de rattrapge)
  - l'exécution continue (en croisant les doigts)
- Levée d'une exception
  - l'exception est rattrapée par qui sait la traiter
- Envoi d'un signal (y compris ex. segfault) et rattrapage

Signalement et rattrapage d'erreur : retour d'une valeur impossible

- [C] char *strchr (char* str, int ch) // search character
  - retourne pointeur nul si caractère non trouvé
- [Java] int indexOf(int ch)
  - retourne -1 si caractère non trouvé
- [C++] size_t string::find (char c, size_t pos = 0)
  - retourne plus grande valeur de size_t si caractère non trouvé
- [C++] iterator set::find (key_type &x)
  - retourne set::end is non trouvé (itérateur vers dernier élément)

- À tester après l'appel

```
it = s.find(x);
if (it != s.end()) { … }
```

Signalement et rattrapage d'erreur : positionnement d'une variable

- double sqrt (double x)
  - si x < 0.0, positionne `errno` à une valeur ≠ 0 (EDOM, ERANGE)
- FILE* fopen(char* filename, char* mode)
  - en cas d'échec, positionne la variable `errno`:
    - EACCES: Permission denied
    - EINVACC: Invalid access mode
    - EMFILE: No file handle available
    - ENOENT: File or path not found

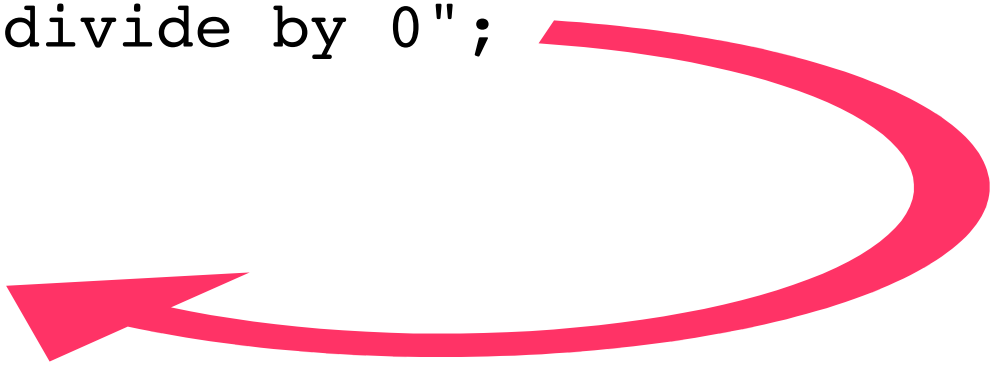
- À tester **immédiatement** après l'appel

```
#include <error.h>
y = sqrt(x);
if (errno != 0) ...
```

  - valeurs possibles de `errno` : cf. `error.h` (~100 valeurs)

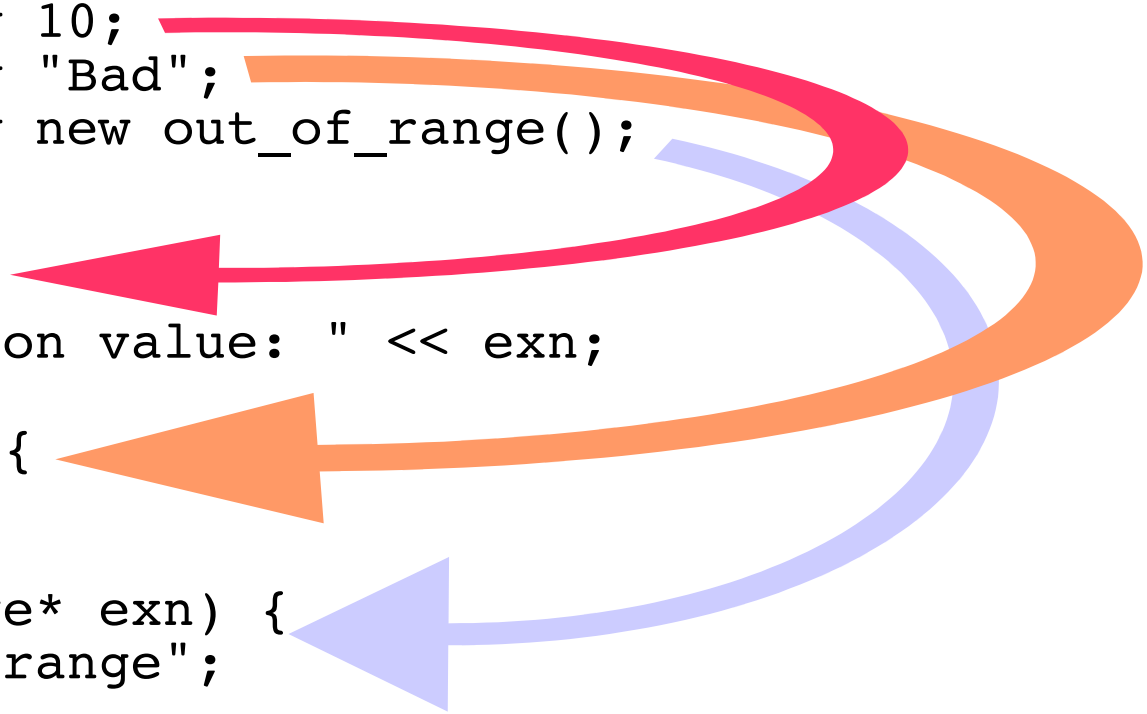
Signalement et rattrapage d'erreur : levée/rattrapage d'exception

```
try
{
   if (x == 0)
     throw "Cannot divide by 0";
   y = 1/x;
   ...
}
catch (char* exn)
{
   // Garder une trace de l'erreur
   cerr << exn << endl;
   // Utiliser la plus grande valeur possible (~infinity)
   y = FLT_MAX;
}
// Exécuté dans tous les cas
z = 1+y/3.0;
...
```



Signalement et rattrapage d'erreur : rattrapage d'exception par type

```
try {
  if (cond1) throw 10;
  if (cond2) throw "Bad";
  if (cond3) throw new out_of_range();
  ...
}
catch (int exn) {
  cout << "Exception value: " << exn;
}
catch (char* exn) {
  cout << exn;
}
catch (out_of_range* exn) {
  cout << "Out of range";
}
catch (exception* exn) { // Subtyping
  cout << "Some exception occurred";
}
catch (...) {              // In all other cases...
  cout << "Something wrong happened";
}
```



Signalement et rattrapage d'erreur : hiérarchie d'exceptions (ex. STL)

```
class exception {
public:
  exception () throw();
  exception (const exception&) throw();
  exception& operator= (const exception&) throw();
  virtual ~exception() throw();
  virtual const char* what() const throw();
}

// Définies par #include <stdexcept> :

class logic_error : public exception {
public:
  explicit logic_error (const string& what_arg);
};

class out_of_range : public logic_error {
public:
  explicit out_of_range (const string& what_arg);
};
```

Signalement et rattrapage d'erreur : hiérarchie d'exceptions (ex. STL)

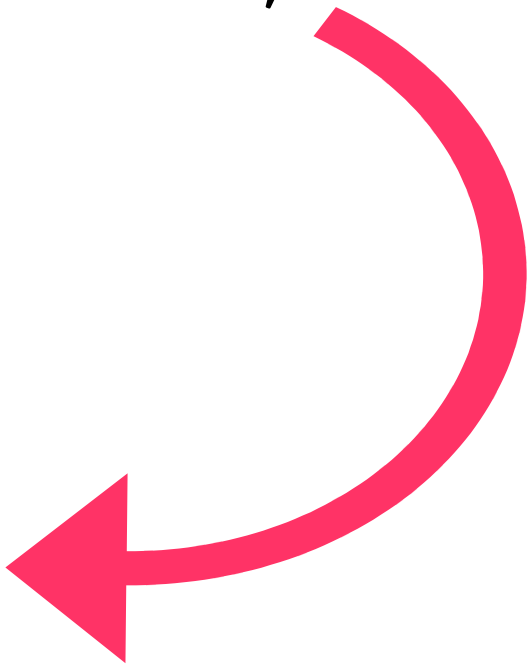
- Exception **logic_error** et sous-classes
  - utilisées pour signaler des erreurs indépendantes des entrées de l'utilisateur, liées à la logique du programme : violation de préconditions, d'invariants...
  - **domain_error**, **invalid_argument**, **length_error**, **out_of_range**, **future_error** (pour les threads)
- Exception **runtime_error** et sous-classes
  - utilisées pour signaler des erreurs causées par les entrées de l'utilisateur, détectable seulement lors d'une exécution
  - **range_error**, **overflow_error**, **underflow_error**, **system_error**

Signalement et rattrapage d'erreur : rattrapage d'exception par (sous-)type

```
// Hiérarchie de classe :
//  out_of_range, length_error < logic_error < exception

try {
  if (cond) throw new length_error();
  ...
}
catch (out_of_range* exn) {  // Type inapproprié
  cout << "Out of range";
}
catch (exception* exn) {     // Premier surtype testé
  cout << "Exception";
}
catch (logic_error* exn) {   // Meilleur surtype mais testé trop tard
  cout << "Logic error";     // Jamais exécuté !
}
catch (length_error* exn) {  // Type exact mais testé trop tard
  cout << "Length error";    // Jamais exécuté !
}
...
```

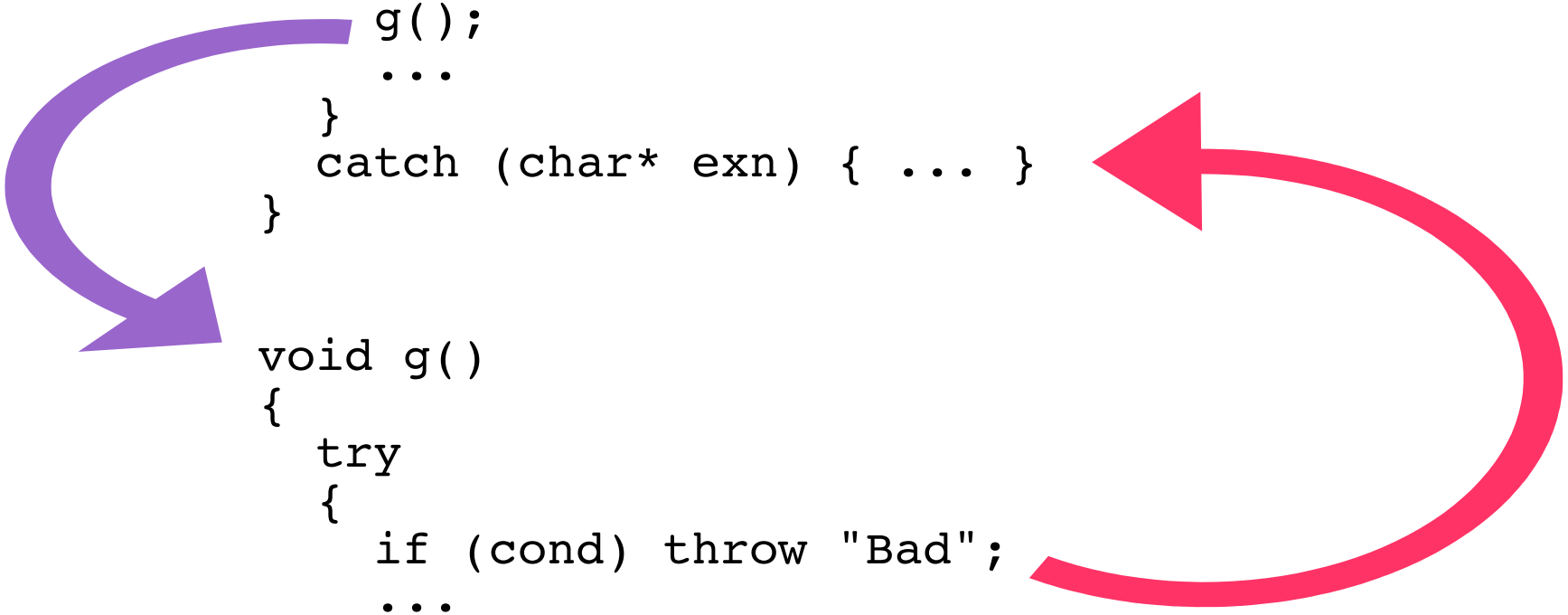
Signalement et rattrapage d'erreur : exceptions emboîtées intraprocédurales

```
try
{
  try
  {
    if (cond) throw "Bad";
    ...
  }
  catch (int exn)
  {
    ...
  }
  ...
}
catch (char* exn)
{
  ...
}
```



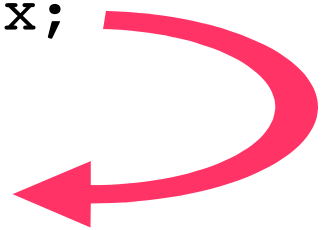
Signalement et rattrapage d'erreur : exceptions emboîtées interprocédurales

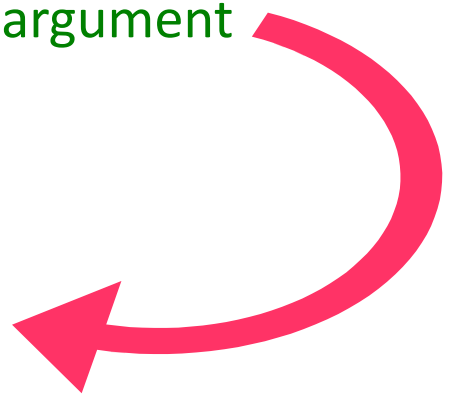
```
void f()
{
  try
  {
    ...
    g();
    ...
  }
  catch (char* exn) { ... }
}


void g()
{
  try
  {
    if (cond) throw "Bad";
    ...
  }
  catch (int exn) { ... }
  ...
}
```



Signalement et rattrapage d'erreur : exception relancée

```
try {
  try {
    if (cond) throw x;
    ...
  }
  catch (int exn) {
    if (exn <= 5)
      // Cas d'erreur spécifique que l'on sait traiter
      y = restart_value(exn);
    else
      // Cas d'erreur que l'on ne sait pas traiter
      // À traiter dans le bloc supérieur → relancer l'exception
      throw;  // Note : pas d'argument
  }
  ...
}
catch (int exn) {
  // Traitement d'erreur général
  ...
```





Ex. gestion des accès aux éléments de vecteurs avec la STL

- Sans vérification d'index: **`v[i]`**
  - même comportement « arbitraire » qu'avec un tableau t[i]
- Avec vérification d'index: **`v.at(i)`**
  - levée d'exception : out_of_range

```
vector<int> myvector(10);
try {
  myvector.at(15)=20;
  ...
}
catch (out_of_range &exn) {
  cerr << "Out of range: "
       << exn.what() << endl; // message associé
}
//Ex. avec g++, cela affiche: "Out of range: vector::_M_range_check"
```

Mieux vaut prévenir que guérir

- Rendre une opération inaccessible lorsqu'elle est impossible
  - ex. faire en sorte que `pop()` ne soit jamais appelé quand la pile est vide : toujours tester `isEmpty()` auparavant
- À tous les niveaux (développeur à utilisateur final)
  - ex. boutons grisés dans barre d'outils et menus
- Robustesse améliorée mais pas garantie
  - la fonction `pop()` reste mécaniquement accessible même lorsque la pile est vide

☛ Tjs prévoir le pire cas : rattraper/relancer à haut niveau

Retour à la pile : Exemple d'implémentation (plus simple)

```
class stack {
    vector<int> v; // Pour pouvoir agrandir la taille facilement
    int level;
public:
    stack() : v(20), level(0) {}
    // Pas de destructeur explicite nécessaire
    bool isEmpty() { return level == 0; }
    void push(int i) {
        if (level == v.size()) v.resize(v.size()+10);
        v[level++] = i;
    }
    int pop() {
        if (level == 0) throw new length_error("Empty stack");
        return v[--level];
    }
};
```

Retour à la pile : Exemple d'implémentation (plus simple)

```
class stack {
    vector<int> v; // Pour pouvoir agrandir la taille facilement
    int level;
public:
    stack() : v(20), level(0) {}
    // Pas de destructeur explicite nécessaire
    bool isEmpty() { return level == 0; }
    void push(int i) {
        if (level == v.size()) v.resize(v.size()+10);
        v[level++] = i;
    }
    int pop() {
        if (level == 0) throw new length_error("Empty stack");
        return v[--level];
    }
};
```

Comment améliorer encore ?

Paramétrage, valeurs par défaut

```
template <typename T>
class stack {
    vector<T> v;
    int level;
public:
    stack(int size = 20) : v(size), level(0) {}
    // Pas de destructeur explicite nécessaire
    bool isEmpty() { return level == 0; }
    void push(T x) {
      if (level == v.size()) v.resize(v.size()+10);
      v[level++] = x;
    }
    T pop() {
      if (level == 0) throw new length_error("Empty stack");
      return v[--level];
    }
};
```

Paramétrage, valeurs par défaut

```
template <typename T>
class stack {
    vector<T> v;
    int level;
public:
    stack(int size = 20) : v(size), level(0) {}
    // Pas de destructeur explicite nécessaire
    bool isEmpty() { return level == 0; }
    void push(T x) {
      if (level == v.size()) v.resize(v.size()+10);
      v[level++] = x;
    }
    T pop() {
      if (level == 0) throw new length_error("Empty stack");
      return v[--level];
    }
};
```

Comment améliorer encore ?

Paramétrage, valeurs par défaut

```
template <typename T, int initSize = 20, int resizeStep = 10>
class stack {
    vector<T> v;
    int level;
public:
    stack(int size = initSize) : v(size), level(0) {}
    // Pas de destructeur explicite nécessaire
    bool isEmpty() { return level == 0; }
    void push(T x) {
      if (level == v.size()) v.resize(v.size()+resizeStep);
      v[level++] = x;
    }
    T pop() {
      if (level == 0) throw new length_error("Empty stack");
      return v[--level];
    }
};
```

Structure de données paramétrée

- Type : objet composite → type des données élement.
  - template [en français « patron »] avec argument
- Taille des conteneurs
  - tailles initiales, paramètres d'adaptation
  - impact sur le temps d'exécution, pas sur les résultats
- Valeurs par défaut
  - cas courants sans besoin d'optimisation particuliers
- Spécialisations éventuelles
  - ex. vecteur de bool stockés de manière dense

Spécialisation de template en C++

- Définition générique

```
template <typename T> class vector {
    T* vec_data;     // Tableau pour stocker les éléments
    int vec_size ;   // Taille du tableau (variable)
    int length; …    // Nombre d'éléments dans le vecteur
```

  - vector<bool> v(100) : 100 mots mémoire de 64 (ou 32) bits

- Définition spécifique

```
template <> class vector<bool> {
    unsigned int *vec_data; // Tableau de bits
    int vec_size ;          // Taille du tableau
    int length; …           // Nombre de bits
```

  - vector<bool> v(100) : 2 (ou 4) mots mémoire

Type abstrait

- Principe de l'encapsulation (information hiding)
  - interface : opaque, minimale
    - déclaration uniquement de ce qui a besoin d'être public
  - implémentation : complexe, optimisée...
    - organisation en mémoire et traitements
- Avantages
  - modification de l'implémentation sans impact sur le reste du code (programme qui utilise le type abstrait)
    - ex. remplacement de tableau par vector dans stack
  - ni réécriture, ni même recompilation suivant les langages (inutile en Java , parfois nécessaire en C++)

Type abstrait en C++

- Principe de l'encapsulation (information hiding)
  - interface : opaque, minimale fichier .h
    - déclaration uniquement de ce qui a besoin d'être public
  - implémentation : complexe, optimisée... fichier .cpp
    - organisation en mémoire et traitements
- Avantages
  - modification de l'implémentation sans impact sur le reste du code (programme qui utilise le type abstrait)
    - ex. remplacement de tableau par vector dans stack
  - ni réécriture, ni même recompilation suivant les langages (inutile en Java , parfois nécessaire en C++)

Interface : intstack.h

```
class intstack {
private:
    vector<int> v;
    int level;
public:
    stack(int size = 20);
    bool isEmpty();
    void push(int x);
    int pop();
};
```

Idéalement à cacher complètement, mais pas facile à masquer en C++. Mais au moins, c'est inaccessible.

Partie visible
pour les utilisateurs
de la structure de données

Implémentation : intstack.cpp

```
stack(int size) : v(size), level(0) {}

bool isEmpty() { return level == 0; }

void push(int x)
{
    if (level == v.size()) v.resize(v.size()+10);
    v[level++] = x;
}

int pop()
{
    if (level == 0) throw new length_error("Empty stack");
    return v[--level];
}
```

Partie **invisible** pour les utilisateurs de la structure de données

Interface : stack.h

```
template <typename T>
class stack {
private:
    vector<T> v;
    int level;
public:
    stack(int size = 20);
    bool isEmpty();
    void push(T x);
    T pop();
};
```

Idéalement à cacher complètement, mais pas facile à masquer en C++. Mais au moins, c'est inaccessible.

Partie visible
pour les utilisateurs
de la structure de données

Implémentation : stack.h aussi ! (à cause des templates)

```
template <typename T>
stack<T>::stack(int size) : v(size), level(0) {}

template <typename T>
bool stack<T>::isEmpty() { return level == 0; }

template <typename T>
void stack<T>::push(T x) {
    if (level == v.size()) v.resize(v.size()+10);
    v[level++] = x;
}

template <typename T>
T stack<T>::pop() {
    if (level == 0) throw new length_error("Empty stack");
    return v[--level];
}
```

Partie **malheureusement visible**
pour les utilisateurs
de la structure de données

Compilation séparée avec des templates

- Compilation des templates (paramétrés)
  - fichier stackUsage.ccp

```
#include "stack.cpp"
template class stack<int>;
template class stack< stack<int> >;
```

- Usage dans un programme (ex. main.cpp)

```
#include "stack.h"
stack<int> s;
```

- Édition de liens
  - main.cpp stackUsage.cpp

Pas beau, mais mieux que rien

Type abstrait en C++

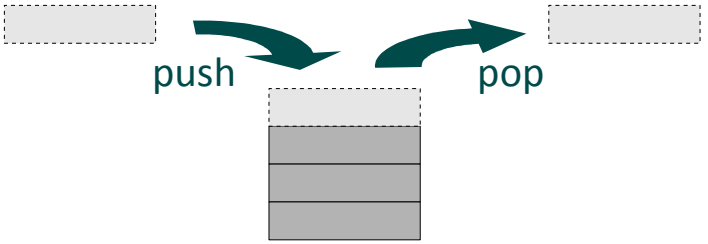
- Principe de l'encapsulation (information hiding)
  - interface : opaque, minimale — fichier .h
    - déclaration uniquement de ce qui a besoin d'être public
  - implémentation : complexe, optimisée... — fichier .cpp ou .h si template
    - organisation en mémoire et traitements
- Avantages
  - modification de l'implémentation sans impact sur le reste du code : ni réécriture, ni recompilation (excepté si templates)

C++ est très mauvais pour définir des types abstraits (comparer à Objective C, Java...)

Il n'y a pas *une* vérité : il existe toujours *des* variantes

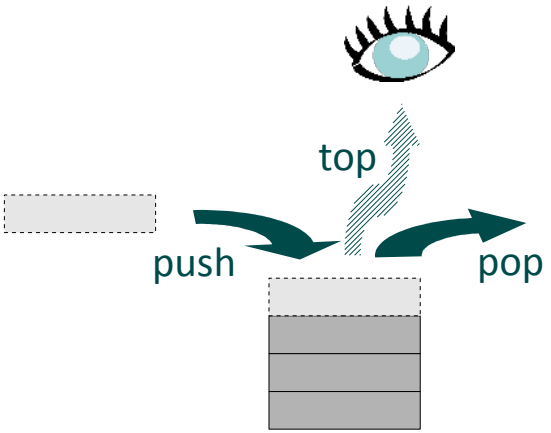
- Exemple de pile 1 :

```
bool isEmpty();
void push(T x);
T pop();
```



- Exemple de pile 2 :

```
bool isEmpty();
void push(T x);
T top();
void pop();
```



Exemple d'implémentation (encore plus simple)

```
template <typename T>
class stack {
    vector<T> v;
public:
    bool isEmpty() { v.empty(); }
    void push(T x) { v.push_back(x); }
    T    top() { return v.back(); }
    void pop() { v.pop_back(); }
};
```

Exemple d'implémentation (il n'y a pas plus simple !)

```
#include <stack>
```

Moralité

- Chercher d'abord si la structure de données est prédéfinie dans une bibliothèque (STL...)
  - 99% de chance qu'une bibliothèque existe déjà !
- Mais attention, les implémentations peuvent différer :
  - efficacité, facilité d'utilisation, portage, gestion d'erreur...
    - ex. overflow et underflow non signalées par la STL

      sauf certaines fonctions, ex. `v.at(i)` plutôt que `v[i]`

- Si implémentation par soi-même
  - faire une interface minimale (extension a posteriori facile)
  - concevoir la gestion d'erreur et la documenter