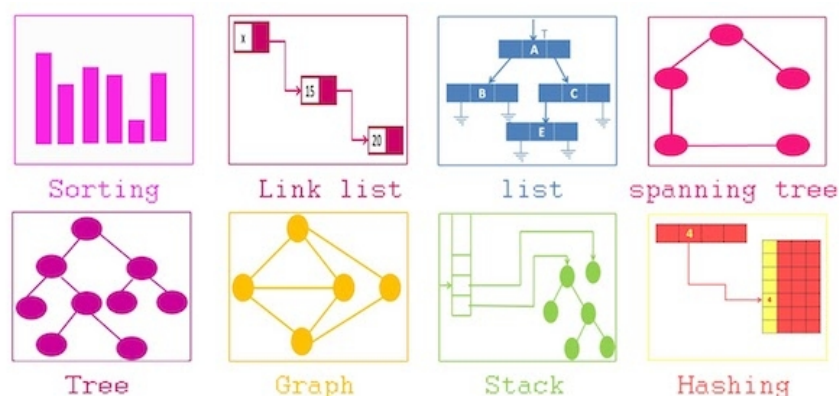


ENPC – MOPSI

Notions de structures de données



By ...axvixomxchopography.com

Renaud Marlet
Laboratoire LIGM-IMAGINE

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

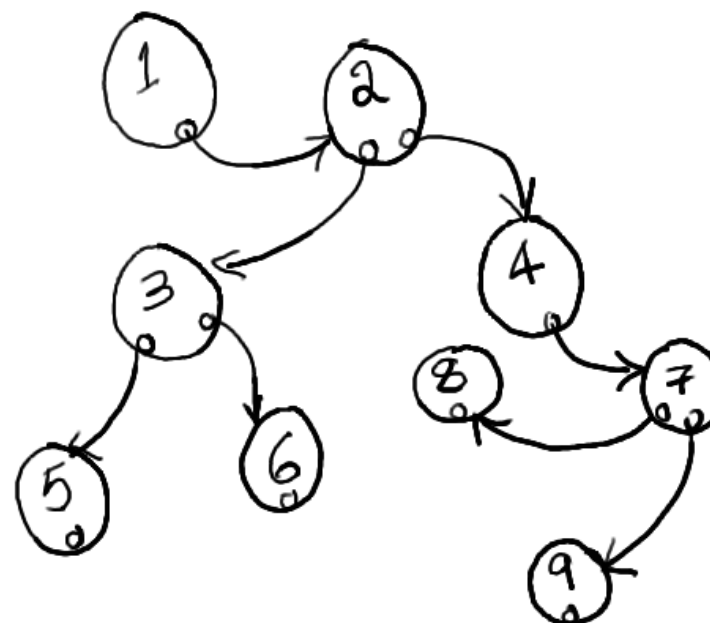
Structure de données

- **Organisation de l'information**

- structuration logique
- moyens d'accès

- **Exemples :**

- tableau
- vecteur, matrice
- liste, file, pile
- arbre, graphe
- table de hachage, ...



Pas d'informatique sans structures de données

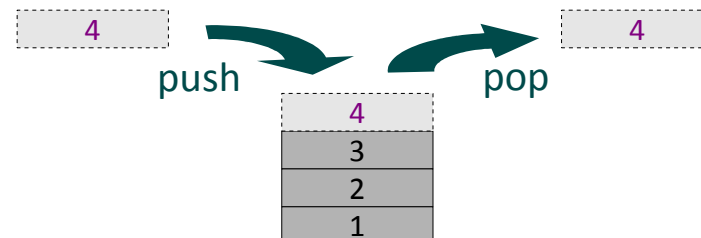
- Omniprésent
 - presque tout ce qui comporte du **discret** (vs continu) ou tout ce qui est **composite**
 - simulations et calculs complexes
 - ex. physique (maillages...)
 - ex. finance (réseaux bayésiens...)
- Aspects théoriques
 - complexité : pire cas, en moyenne...
- Aspects pratiques
 - bibliothèques logicielles : STL, Imagine++, Boost, Eigen...



Type abstrait : données + opérations

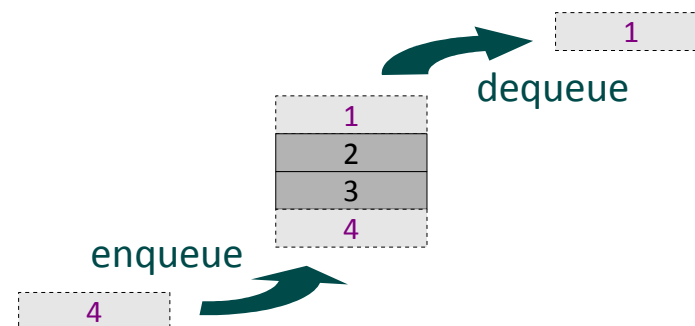
- Exemple 1 : pile (stack) [Last In First Out, LIFO]

- vide?
- empiler (push)
- dépiler (pop)



- Exemple 2 : file (queue) [First In First Out, FIFO]

- vide?
- ajouter (enqueue)
- extraire (dequeue)

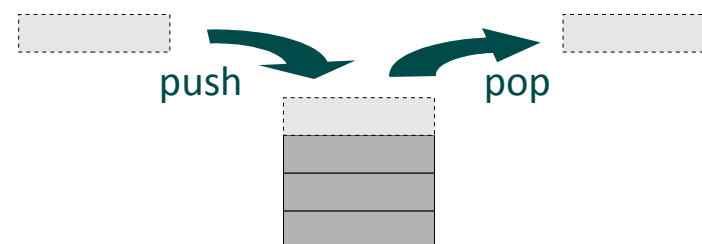


Type abstrait : données + opérations

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

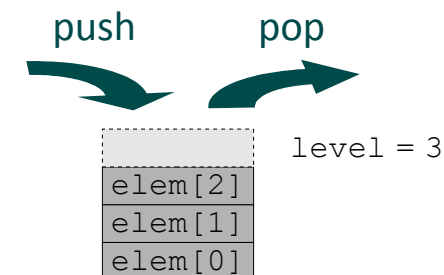
- Exemple : pile (stack)

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



Exemple d'implémentation (simple)

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



$Y = X++ \Leftrightarrow Y = X; X++;$
 $Y = ++X \Leftrightarrow X++; Y = X;$

$T[X++] = Y \Leftrightarrow T[X] = Y; X++;$
 $T[++X] = Y \Leftrightarrow X++; T[X] = Y;$

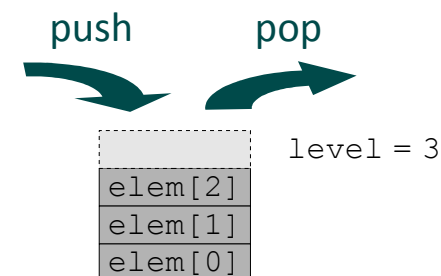
```
int main() { // Exemple d'usage
    stack s;
    s.push(1);
    s.push(2);
    cout << s.pop() << " " << s.pop() << endl;
    // Affiche : 2 1
}
```

Exemple d'implémentation (simple)

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

Quels sont les défauts ?
(au moins 4)

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

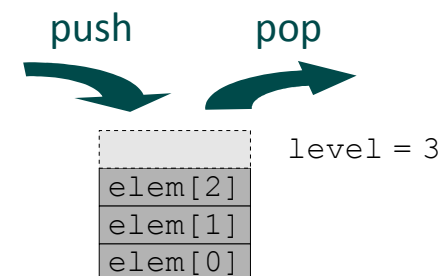


Exemple d'implémentation (simple)

```
class stack { // Type abstrait = données + opérations
    int elem[20]; // taille fixe, codée en dur
    int level; // pas initialisé
public:
    bool isEmpty() { return level == 0; }
    void push(int i) { elem[level++] = i; } // risque d'overflow
    int pop() { return elem[--level]; } // risque d'underflow
};
```

Quels sont les défauts ?
(au moins 4)

```
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 { // Type abstrait = données + opérations
    int elem[20]; // taille fixe, codée en dur
    int level; // pas initialisé
public:
    bool isEmpty() { return level == 0; }
    void push(int i) { elem[level++] = i; } // risque d'overflow
    int pop() { return elem[--level]; } // risque d'underflow
};
```

Quels sont les défauts ?
(au moins 4)

```
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 { // Type abstrait = données + opérations
    int elem[20]; // taille fixe → allocation + redimensionnement dynamiques
    int level; // pas initialisé → initialisation (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 → erreur
}; // + ajout d'un destructeur pour libérer la mémoire dynamique ou valeur par défaut

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 ?

Rappel sur les pointeurs

$$X[Y] \Leftrightarrow (* (X+Y))$$

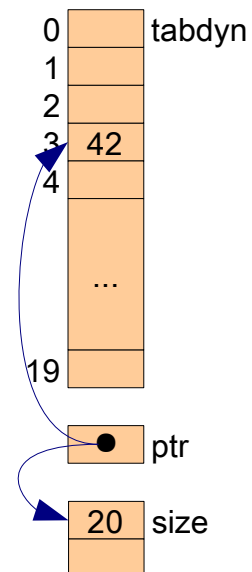
$$X \rightarrow \text{name} \Leftrightarrow (*X).\text{name}$$

- Allocation statique**

```
int tabstat[20];
tabstat[3] = 42;
```

- Allocation dynamique**

```
int *tabdyn;
int size = f(var)+1;
tabdyn = new int[size];
tabdyn[3] = 42;
int* ptr = tabdyn+3;
*ptr=42; ptr[0]=42; // idem
ptr = &size; (*ptr)++; // 21
delete[] tabdyn;
```



- Allocation statique**

```
class Point { public:
    int x, y;
    Point(int u,int v){...} };
Point p(); p.x = 17; p.y = 5;
Point q(17,5);
int *ptr = &p.x; *ptr = 42;
```

- Allocation dynamique**

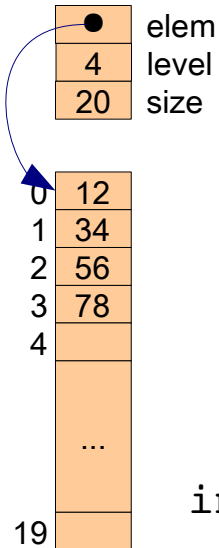
```
Point *p = new Point();
p->x = 17; p->y = 5;
Point *q = new Point(17,5);
int *ptr = &p->x; *ptr = 42;
```

$$*X++ \Leftrightarrow *(X++)$$

Exemple d'implémentation (plus robuste)

```
class stack {
    int *elem;
    int level, size;
public:
    stack() : level(0), size(20) { elem = new int[size]; } // Constructeur
    ~stack() { delete[] elem; } // Destructeur : libération de la mémoire
    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 mémoire
        }
        elem[level++] = i; // Garantie: toujours de la place pour empiler
    }
    int pop() {
        if (level == 0) return 0; // Rendre une valeur par défaut si pile vide
        return elem[--level]; // Garantie: jamais d'accès hors du tableau
    }
};
```

$Y = X++ \Leftrightarrow Y = X; X++;$ $T[X++] = Y \Leftrightarrow T[X] = Y; X++;$
 $Y = ++X \Leftrightarrow X++; Y = X;$ $T[++X] = Y \Leftrightarrow X++; T[X] = Y;$

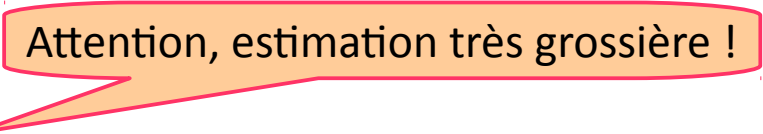


En résumé

- Constructeur
 - bien initialiser
- Destructeur
 - libérer la mémoire allouée dynamiquement (récursif)
- Vérifications de validité, de cohérence
 - débordement inférieur/supérieur (underflow/overflow), ...
- Adaptation pour poursuivre l'exécution, ou non

Gestion des cas d'erreur 🖱️

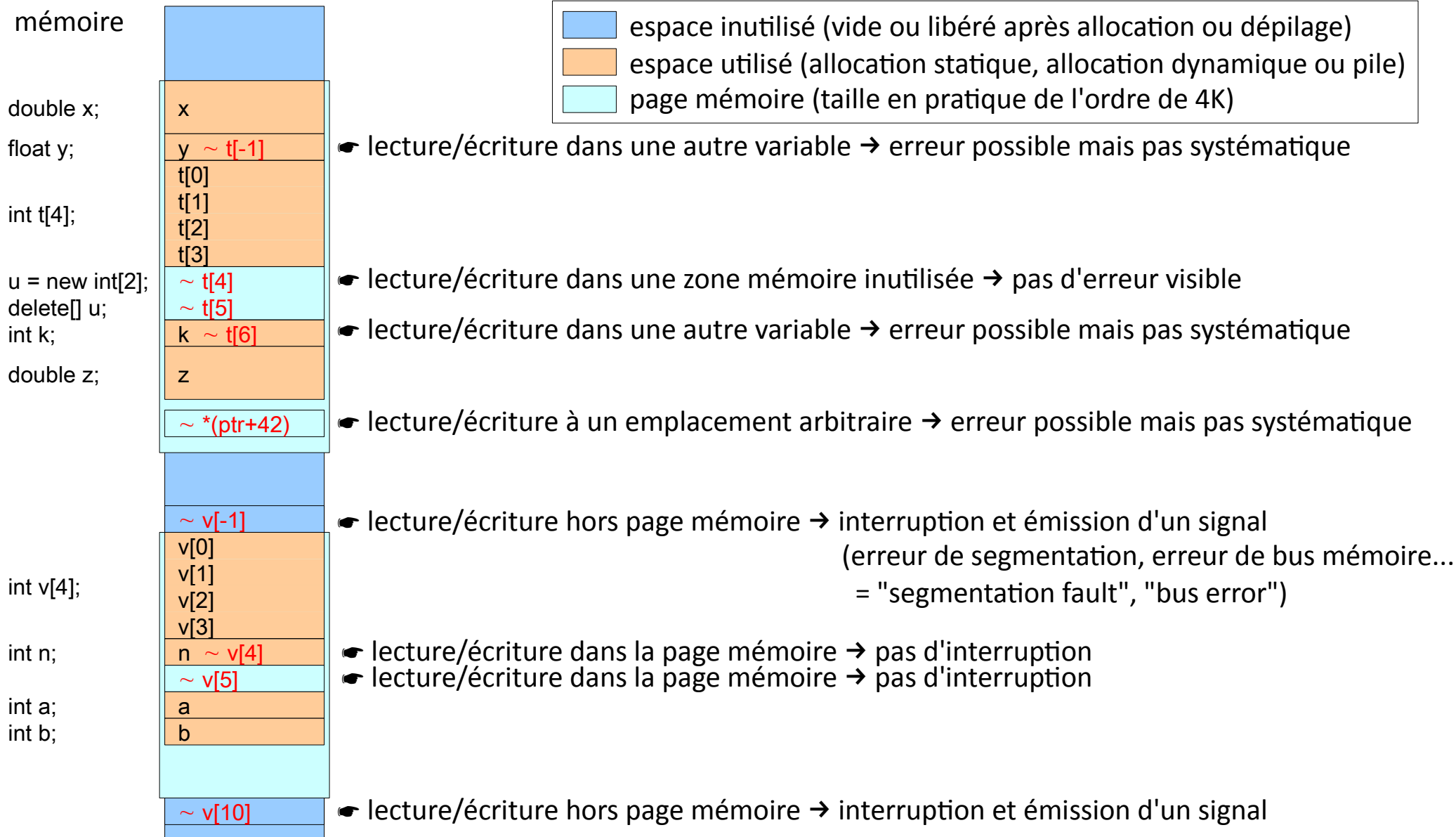
Y a un bug ?!

- Taille/complexité croissante des systèmes
[mesure courante : “(source) lines of code” = (S)LOC → KLOC, MLOC...]
 - noyau Linux : 5 MLOC (2003), 16 MLOC (2012), >20 (2017)
 - distribution Debian : 55 MLOC (2000), 419 MLOC (2012)
- **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
- **Erreurs = la norme, pas l'exception** ➡ **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 (Institute of Electrical and Electronics Engineers)
 - **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
- **Dans l'usage, les termes restent flous/ambigus ☹**
 - non voulu/anticipé pour qui ? : “it's not a bug, it's a feature”

Erreurs typiques d'accès à la mémoire



Erreur manifeste vs erreur silencieuse

- **Erreur manifeste**, « incompatible » avec l'exécution
 - ex. déréréférence d'un pointeur nul, accès à zone interdite
 - erreur système : segmentation fault (segfault, bus error...)
 - fin brutale de l'exécution, données en traitement perdues
 - **pas de résultat**
- **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
 - **résultat potentiellement faux, mais on ne le sait pas**

Stratégies de gestion d'erreur

(forme de tolérance aux fautes)

- **Détection**

if (error condition) then action

- **explicite** : test dans le programme
 - hypothèse : on a une idée de ce qui est attendu/normal
ex. plage de valeurs, type d'objet, cohérence entre valeurs...
 - test de cas simples seulement sinon erreur dans gestion d'erreur...
- **implicite** : rattrapage d'exceptions, abonnement à signaux

- **Réaction**

- **signalement** de l'erreur ou non ?
- **à qui ?** → programmeur ou utilisateur final ?
- **poursuite** de l'exécution ou arrêt ?
 - avec valeur par défaut ? → peu sûr : on croise les doigts...
 - avec un véritable traitement alternatif !

Tentative de traitement : détecter, avertir, sauvegarder, terminer

```
int pop()  
{  
    if (level <= 0) { // If popping from empty stack  
        cout << "Trying to pop from empty stack" << endl;  
        savePreciousData();  
        exit(17);      // Quit execution, indicating specific error code  
    }  
    return elem[--level] ;  
}
```

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

Tentative de traitement : détecter, avertir, sauvegarder, terminer

```
int pop()
{
    if (level <= 0) { // If popping from empty stack
        cout << "Trying to pop from empty stack" << endl;
        savePreciousData();
        exit(17);      // Quit execution, indicating specific error code
    }
    return elem[--level] ;
}
```

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

Tout va bien ?
Est-ce pratique ?

Tentative de traitement : détecter, avertir, sauvegarder, terminer

```
int pop()  
{  
    if (level <= 0) { // If popping from empty stack  
        cout << "Trying to pop from empty stack" << endl;  
        savePreciousData();  
        exit(17);      // Quit execution, indicating specific error code  
    }  
    return elem[--level] ;  
}
```

```
% prog tqPopOK  
% echo $?  
0  
% prog tqPopEmpty  
% echo $?  
17
```

Non, ce n'est
pas pratique 😞

- message spécifique à l'erreur, mais pour qui ?
développeur de stack, utilisateur de stack ou utilisateur final ?
- sauvegarde des données précieuses, mais souvent pas accessibles
(pas visibles) depuis stack, ou inconnues de ce développeur
- terminaison propre, mais brutale (pas de seconde chance)
- diagnostic possible à l'extérieur du programme, mais difficile

Messages d'erreur (ou d'avertissement)

- À l'attention d'un **utilisateur interne** (développeur)
 - **canal de sortie** spécifique pour erreurs : **cerr** (\neq cout)
 - `cerr << "Trying to pop from empty stack" << endl;`
 - `% prog 2>errfile`
 - `% prog 2>&1 | more`
 - écriture dans un **fichier de log** (= journal de bord)
 - plus général que pour les seules erreurs : trace de toute activité
 - ajout toujours en fin de fichier → histoire complète de l'exécution
 - consultable pendant et après exécution, et exécutions multiples
 - ex. Unix : plusieurs fichiers pour différents services, voir `/var/log`
- À l'attention d'un **utilisateur final**
 - boîtes de dialogue (interface graphique)

0: entrée standard (stdin, cin)
1: sortie standard (stdout, cout)
2: sortie d'erreur (stderr, cerr)

Messages d'erreur (ou d'avertissement)

- Penser à l'internationalisation :
 - message pour les développeurs → généralement en anglais
 - pour une diffusion large, dans des équipes internationales
 - message pour les utilisateurs finaux → langue paramétrable
 - masque avec trous, spécifiques à chaque langue

```
msg[FRENCH] = "Le bilan est de %d sur %d\n";  
msg[ENGLISH] = "The balance is %d out of %d\n";  
msg[ITALIAN] = "Il bilancio è di %d su %d\n";  
...  
if (...) lang = FRENCH; // Choix de langue  
...  
printf(msg[lang], bal, tot); // Msg contextuel
```

Terminer proprement

- Terminaison « propre » :

- fermeture des fichiers ouverts
avec vidage (flush) des flux bufferisés
- effacement des fichiers temporaires
- contrôle rendu à l'environnement hôte

flux bufferisé (buffered stream):
entrées/sorties sur disque différées
pour des raisons d'efficacité
→ écriture de données par bloc

- Terminaison propre automatique avec **exit**

```
#include <stdlib.h>
```

```
...
```

```
exit(1); // void exit(int status)
```

- et 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

Nettoyage spécifique au programme lors de la terminaison

- 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

Tentative de traitement : détecter, avertir, sauvegarder, terminer

```
int pop()  
{  
    if (level <= 0) { // If popping from empty stack  
        cout << "Trying to pop from empty stack" << endl;  
        savePreciousData();  
        exit(17); // Quit execution, indicating specific error code  
    }  
    return elem[--level] ;  
}
```

- message spécifique à l'erreur, mais pour qui ?
développeur de stack, utilisateur de stack ou utilisateur final ?
- sauvegarde des données précieuses, mais souvent pas accessibles
depuis stack, ou inconnues du développeur

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

Mécanismes de signalement d'erreur

- Affecter un code d'erreur dans une variable
 - peut être testée et un comportement approprié choisi
- Retourner une valeur impossible (ex. -1 pour une taille)
 - peut être testée et un comportement approprié choisi
- Lancer une exception (\exists dans beaucoup de langages)
 - rattrapable par l'appelant
- Envoyer un signal (y compris ex. segfault)
 - reçu par une fonction définie au préalable pour le traiter

Signalement d'erreur : variable d'erreur

- Variable d'erreur définie par le système : **errno**
 - ex. racine carrée : `double sqrt (double x)`
si $x < 0$, la variable **errno** reçoit une valeur $\neq 0$ (EDOM)

```
#include <error.h>      ou <cerrno>
y = sqrt(x);
if (errno != 0) ...
```
 - ex. ouverture de fichier : `FILE* fopen(char* filename,...)`
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 ...
 - valeurs possibles : voir `error.h` (~100 valeurs)

Signalement d'erreur : variable d'erreur

- Définition/affectation d'une variable d'erreur
 - par l'appel d'une fonction standard : `errno`
 - par le programmeur (réutilisation possible de `errno`)
 - **Utilisation très délicate**
 - `errno` pas remis à 0 à chaque opération
→ peut indiquer une erreur ancienne
 - à tester **immédiatement** après l'appel

```
y = sqrt(x);
afficher(x);
if (errno != 0)
    cout << "sqrt neg"; // Faux si afficher() modifie errno
```
- ☞ Pratique peu recommandée (lourd, risque d'erreurs)

Signalement 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 la plus grande valeur de `size_t` si caractère non trouvé
- [C++] `iterator set::find (key_type &x)`
 - retourne `set::end` si non trouvé (itérateur vers dernier élément)
- À tester après l'appel

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

`size_t` : type entier ≥ 0
représentant une taille
ou une position mémoire

Signalement d'erreur : retour d'une valeur impossible + variable d'erreur

- **Variable d'erreur peu recommandée... mais OK pour qualifier un type d'erreur signalée**
 - ex. `FILE* fopen(char* filename, char* mode)`
 - si échec, retourne un pointeur nul **et** positionne `errno`:
 - `EACCES`: Permission denied
 - `EINVACC`: Invalid access mode ...
 - tester après l'appel

```

FILE* fp = fopen("file", "r"); // Essaie d'ouvrir le fichier en lecture
if (!fp) {                      // En cas d'échec (si pointeur nul)
    perror("Cannot open file"); // Affiche message + cause sur cerr
    // Affiche par ex. : "Cannot open file : No such file or directory"
    exit(1) ; }
fscanf(fp,"%d",&n);             // Lit un entier dans le fichier

```

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

```

try
{ // Bloc d'exécution à l'intérieur duquel des exceptions peuvent être lancées
  if (x == 0)
    throw "Cannot divide by 0"; // Lancement d'une exception
  y = 1/x;
  ...
  ...
  ...
}
catch (char* exn) // Rattrapage d'une exception de type char*
{
  // Par exemple, avertir de l'erreur
  cerr << exn << endl;
  // Tolérance à la faute : utiliser la plus grande valeur possible (~ infinity)
  y = FLT_MAX;
}
// Code exécuté dans tous les cas (qu'une exception soit lancée ou pas)
z = y-3.0;

```

si l'exception
est lancée

si l'exception
n'est pas
lancée

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

```
try {  
    if (cond1) throw 10;  
    if (cond2) throw "Bad";  
    if (cond3) throw 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) { // Surtype de out_of_range (au cas où...)  
    cout << "Some exception occurred";  
}  
catch (...) { // Attrape toutes les exceptions qui passent  
    cout << "Something wrong happened";  
}
```

The diagram illustrates the flow of exception handling. Three curved arrows originate from the right side of the code: a pink arrow points to the `catch (int exn)` block, an orange arrow points to the `catch (char* exn)` block, and a light blue arrow points to the `catch (out_of_range &exn)` block. These arrows represent the specific exception types being caught. A fourth, larger light blue arrow originates from the right side and points towards the `catch (...)` block, representing a catch-all handler for any exception that is not caught by the previous blocks.

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 : 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**

en pratique :
∃ zone grise

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

```
try {  
    if (cond) throw length_error();  
    ...  
}
```

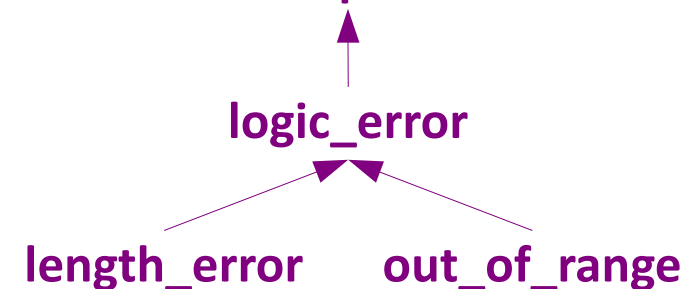
```
catch (out_of_range &exn) { // Type inapproprié  
    cout << "Out of range";  
}
```

```
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 !  
}  
...
```

Hierarchie de classes : **exception**



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

```

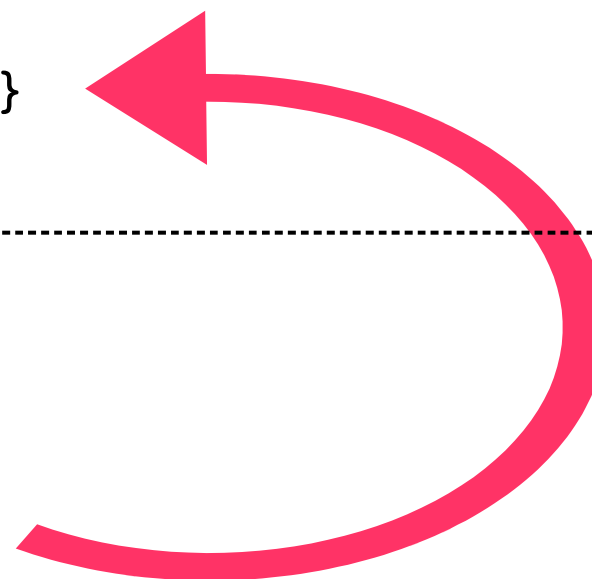
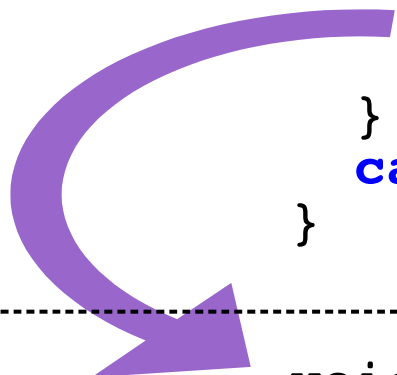
void f()
{
    try
    {
        ...
        g();
        ...
    }
    catch (char* exn) { ... }
}

```

```

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

```



Signalement et rattrapage d'erreur : relancement d'exception

```


try {
    ...
    try {
        ...
        if (cond) throw e;
        ...
    }
    catch (...) { // Type de l'exception quelconque
        doSomethingGeneric();
        throw; // Relance la même exception
               // (sans la connaître)
    }
    ...
}
catch (type exn) {
    doSomethingSpecific(exn);
    throw exn; // Relance la même exception
              // (ici en la réutilisant explicitement)
}

```


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> v(10);  
try {  
    v.at(15)=20;  
    ...  
}  
catch (out_of_range &exn) {  
    cerr << "Out of range: "  
        << exn.what() << endl; // message associé  
}
```



//Ex. avec g++ et libc++4.2.1, cela affiche: "Out of range: vector"

Au fait, ayez le réflexe de lire la doc !

public member function

std::vector::operator[]

<vector>

```
reference operator[] (size_type n);  
const_reference operator[] (size_type n) const;
```

Access element

Returns a reference to the element at position *n* in the **vector** container.

[...]

Exception safety

If the container **size** is greater than *n*, the function never throws exceptions (no-throw guarantee). Otherwise, the behavior is undefined.

public member function

std::vector::at

<vector>

```
reference at (size_type n);  
const_reference at (size_type n) const;
```

Access element

Returns a reference to the element at position *n* in the **vector**.

The function automatically checks whether *n* is within the bounds of valid elements in the **vector**, throwing an **out_of_range** exception if it is not (i.e., if *n* is greater than, or equal to, its **size**). This is in contrast with member **operator[]**, that does not check against bounds.

Mieux vaut prévenir que guérir

- Tester qu'une opération est possible avant de la faire
 - ex. toujours tester `isEmpty()` avant d'appeler `pop()`
- Rendre une opération inaccessible lorsqu'elle est impossible ou n'a pas de sens
 - 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
- ☞ Toujours prévoir le pire cas → rattraper/relancer

Assertion :

Gestion d'erreur conditionnelle

```
#include <assert.h>
...
assert(level > 0) ; // assert(invariant)
                    // = condition qui doit toujours être satisfaite
return elem[--level];
```

- **En phase de mise au point**

- condition de assert toujours testée : plus sûr, mais plus lent
 - en cas d'insatisfaction, message et terminaison (brutale)
 - assertion failed: *expression*, file *filename*, line *line number*
- [généralement implémenté via des exceptions]

- **Une fois le code mis au point (ex. à la livraison)**

- test éliminés : pas sûr, mais plus rapide
 - #define NDEBUG avant l'inclusion de assert.h
 - ligne de commande du compilateur : /DNDEBUG, -DNDEBUG

Quoi utiliser quand ?

- Limites d'utilisation
 - variable d'erreur : toujours utilisable, mais lourd et peu sûr
 - valeur de retour : si \exists valeurs impossibles, sinon ajout statut d'erreur
 - lourd, avec le type `Pair` de la STL, ex. `Pair(result,error_status)`
 - sauf cas du type `void`, transformé en type `error_status`
 - **exception** : plus souple, moins lourd pour cas courants → à privilégier
- Veiller à la cohérence du choix :
 - ex. tout par valeur de retour ou tout par exception
- Assertion
 - OK pour test de la cohérence **interne** d'un programme/bibliothèque
 - pas pour signaler une erreur à un utilisateur
(idée = suppression du `assert` à la livraison du code binaire)

Retour à la pile :

Implémentation plus simple, plus robuste

```
class stack {
    vector<int> v; // Pour pouvoir agrandir la taille facilement
    int level;
public:
    stack() : v(20), level(0) {}
    // Pas de destructeur explicite : v détruit automatiquement quand un stack est détruit
    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 length_error("Empty stack");
        return v[--level];
    }
};
```

Retour à la pile :

Exemple d'utilisation

```
stack s;
s.push(1);
s.push(2);
...
```

```
for (int i = 0; i < 4; i++) // Dépile 4 éléments
    cout << s.pop() << endl;
...
```

// Affiche (ex. avec g++) :

2

1

libc++abi.dylib: terminating with uncaught exception of type std::length_error: EmptyStack

Retour à la pile :

Exemple d'utilisation

```
stack s;  
s.push(1); // Empile 2 éléments  
s.push(2);  
...  
try  
{  
    for (int i = 0; i < 4; i++) // Dépile 4 éléments  
        cout << s.pop() << endl;  
    ...  
}  
catch (exception &exn) {  
    cerr << "Something bad happened: " << exn.what() << endl;  
    cerr << "Trying to continue nonetheless" << endl;  
}  
// Affiche :  
2  
1  
Something bad happened: Empty stack  
Trying to continue nonetheless
```


Throw-catch immédiat : ???

```
int pop() {  
    try {  
        if (level == 0)  
            throw length_error("Empty stack");  
    }  
    catch (length_error e) {  
        cerr << "Something bad happened: " << e.what() << endl;  
        cerr << "Trying to continue nonetheless" << endl;  
        return 0;  
    }  
    return v[--level];  
};
```

Ça vous va ?

Throw-catch immédiat : pas de sens

```
int pop() {  
    try {  
        if (level == 0)  
            throw length_error("Empty stack");  
    }  
    catch (length_error e) {  
        cerr << "Something bad happened: " << e.what() << endl;  
        cerr << "Trying to continue nonetheless" << endl;  
        return 0;  
    }  
    return v[--level];  
};
```

Équivalent

```
int pop() {  
    if (level == 0) {  
        cerr << "Something bad happened: Empty stack" << endl;  
        cerr << "Trying to continue nonetheless" << endl;  
        return 0;  
    }  
    return v[--level];  
};
```

Throw-catch immédiat : pas de sens

```
int pop() {  
    try {  
        if (level == 0)  
            throw length_error("Empty stack");  
    }  
    catch (length_error e) {  
        cerr << "Something bad happened: " << e.what() << endl;  
        cerr << "Trying to continue nonetheless" << endl;  
        return 0;  
    }  
    return v[--level];  
};
```

Programmation
inutilement
lourde



Équivalent

```
int pop() {  
    if (level == 0) {  
        cerr << "Something bad happened: Empty stack" << endl;  
        cerr << "Trying to continue nonetheless" << endl;  
        return 0;  
    }  
    return v[--level];  
};
```

Throw-catch immédiat : pas de sens

```
int pop() {  
    try {  
        if (level == 0)  
            throw length_error("Empty stack");  
    }  
    catch (length_error e) {  
        cerr << "Something bad happened: " << e.what() << endl;  
        cerr << "Trying to continue nonetheless" << endl;  
        return 0;  
    }  
    return v[--level];  
};
```

Programmation
inutilement
lourde



Équivalent

```
int pop() {  
    if (level == 0) {  
        cerr << "Something bad happened: Empty stack" << endl;  
        cerr << "Trying to continue nonetheless" << endl;  
        return 0;  
    }  
    return v[--level];  
};
```

Peu/pas sûr,
peu/pas exploitable

Throw-catch immédiat : pas de sens

```
int pop() {
    try {
        if (level == 0)
            throw length_error("Empty stack");
    }
    catch (length_error e) {
        cerr << "Something bad happened: " << e.what() << endl;
        cerr << "Trying to continue nonetheless" << endl;
        return 0;
    }
    return v[--level];
};
```



Je passe à la moulinette ceux qui me répondront ça
(ou un équivalent comme : try { v.at(level) } catch ...)
dans l'exercice (TP) à la fin du cours !!!

Oui, il faut lancer une exception, mais qui devra
être attrapée (ou non) **dans l'appelant**

Retour à la pile :

Implémentation plus simple, plus robuste

Rappel

```
class stack {  
    vector<int> v; // Pour pouvoir agrandir la taille facilement  
    int level;  
public:  
    stack() : v(20), level(0) {}  
    // Pas de destructeur explicite : v détruit automatiquement quand un stack est détruit  
    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 length_error("Empty stack");  
        return v[--level];  
    }  
};
```

Comment améliorer encore ?
(fonctionnellement)

Paramétrage, valeurs par défaut

```
template <typename T> // Pile d'objets de type quelconque
class stack {
    vector<T> v;
    int level;
public:
    stack(int size = 20) : v(size), level(0) {}
    // Création possible avec une taille initiale quelconque (pas juste 20 par défaut)
    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 length_error("Empty stack");
        return v[--level];
    }
};
```

Paramétrage, valeurs par défaut

```
template <typename T> // Pile d'objets de type quelconque
class stack {
    vector<T> v;
    int level;
public:
    stack(int size = 20) : v(size), level(0) {}
    // Création possible avec une taille initiale quelconque (pas juste 20 par défaut)
    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 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;           // Pas de retailage resizeStep paramétré statiquement
    int level;
public:
    stack(int size = initSize) : v(size), level(0) {}
    // Taille initiale par défaut statique, mais taille quelconque possible dynamiquement
    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 length_error("Empty stack");
        return v[--level];
    }
};
```

Templates (= patrons) pour des structures de données paramétrées

- Arguments
 - type des données élémentaires
 - valeurs codées en dur
 - valeurs par défaut
 - pour cas courants sans besoin d'optimisation particuliers
 - ex. taille des conteneurs
 - tailles initiales, paramètres d'adaptation
 - impact sur le temps d'exécution, pas sur les résultats
- 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 (=spécialisée pour un type donné)

```
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 :

données + opérations + encapsulation

- 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
 - implémentation modifiable sans impact sur reste du code
 - ex. remplacement facile de tableau par vector dans stack
 - ni réécriture, ni même recompilation suivant les langages
 - inutile en Java , parfois nécessaire en C++
 - confidentialité : propriété intellectuelle, sécurité...

Type abstrait : **en C++**

données + opérations + encapsulation

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

Interface : intstack.h

```
class intstack {
private:
    std::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++.
Au moins, c'est inaccessible par le programmeur.

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

Implémentation : intstack.cpp

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

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

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

int stack::pop()
{
    if (level == 0) throw 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++.
Au moins, c'est inaccessible par le programmeur.**

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 length_error("Empty stack");
    return v[--level];
}
```

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

Le code du template doit rester disponible/visible
car il est régénéré à chaque instance du template

Compilation séparée

- Éviter de recompiler tous les fichiers à chaque modification mineure
 - pratique pour les petits projets
 - **indispensable** pour les projets moyens ou gros
- Makefile (derrière CMake)
 - expression des dépendances entre fichiers
 - pas de recompilation si un fichier indépendant est modifié
- Problème
 - réduire les dépendances au minimum

Compilation séparée avec des templates

- Déclaration de cas d'utilisation courants → précompil.

- ex. fichier stackUsage.cpp

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

- Usage dans un programme

- ex. fichier main.cpp

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

Pas beau mais
mieux que rien

- Compilation séparée

- fichiers : stack.h, stack.cpp, stakeUsage.cpp, main.cpp
- stack<int> pas recompilé si seulement main.cpp est modifié

Type abstrait en C++

- Principe de l'encapsulation (information hiding)

- interface : opaque, minimale

fichier .h (public)

- déclaration uniquement de ce qui a besoin d'être public

- implémentation : complexe, optimisée

- organisation en mémoire et traitements

fichier .cpp (caché)
ou fichier .h si template

- Avantages (excepté si templates !)

- modification de l'implémentation sans impact sur le reste du code
- confidentialité

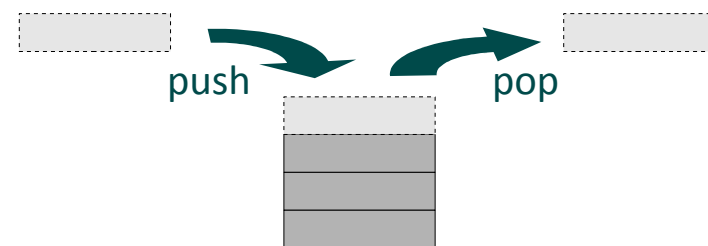
C++ est très mauvais pour définir des types abstraits (comparer à Objective C, Java...)

- Mais templates très courants... ☹

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

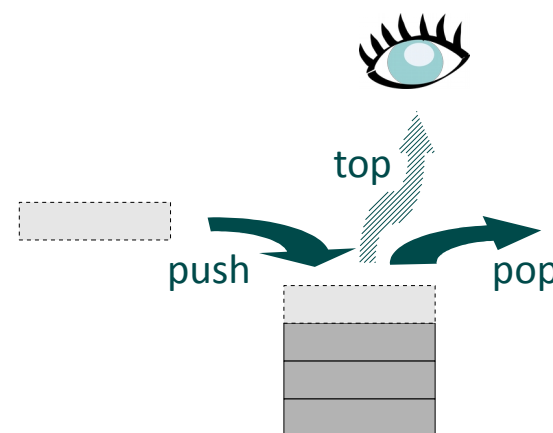
- Exemple de pile 1 :

```
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(); }
};
```

// Attention !, pas de signalement d'erreur avec vector::back() ni vector.pop_back()

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

```
#include <stack>
```

```
// Attention !, pas de signalement d'erreur avec stack::top() ni stack::pop()
```

Moralité

- Chercher d'abord si la structure de données est prédéfinie dans une bibliothèque (ex. STL...)
 - **99% de chance qu'une bibliothèque existe déjà pour vos besoins !**
- Attention : lire la documentation associée
 - **il existe différentes implémentations d'une même fonctionnalité**
 - efficacité, facilité d'utilisation, portage, gestion d'erreur...
 - ex. overflow et underflow généralement non signalées par la STL
sauf pour certaines fonctions, ex. `v.at(i)` plutôt que `v[i]`
- Si implémentation par soi-même nécessaire
 - faire une **interface minimale** (puis étendre selon besoins)
 - concevoir la **gestion d'erreur** et la **documenter**