Programmation Orientée Objet

POO en C++

Léa Brunschwig
✉ lea.brunschwig@univ-pau.fr

L2 Informatique

Université de Pau et des Pays de l'Adour
Collège STEE
Département Informatique

01.
Introduction à la POO
Définition POO, objet et classe.

02.
Concepts de la POO
Encapsulation, héritage, polymorphisme, abstraction, …

03.
POO en C++
Apprentissage du C++.

C++

- langage créé dans les années 80 basé sur la syntaxe du langage C

Avantages

- Plus rigoureux que C,
- Fortement typé,
- Programmation objet :
  - Abstraction des données,
  - Généricité,
  - ...

Inconvénients

- Compatible avec C,
- Présence des deux syntaxes :
  - Utilisation de fonctions C.

Deux sortes de fichiers :

- **.cpp** : contient l'implémentation,
- **.h** : contient les déclarations.

La Syntaxe

Premier programme

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

int main() {
    cout << "Hello World" << endl;
    return 0;
}
```

Commentaires

```
1 #include <iostream> // Commentaire en fin de ligne
2
3 // Un autre commentaire
4
5 /* Un commentaire
6 sur plusieurs
7 lignes */
```

Entrées/sorties

Utilisation des flux (`iostream`) :

- **Entrée :**
  - `cin`
  - `>>` : extrait des valeurs d'un flux d'entrée
- **Sortie :**
  - `cout` : sortie standard
  - `cerr` : sortie d'erreur
  - `clog` : sortie de log
  - `<<` : envoie des valeurs dans un flux de sortie

Intérêts des flots :

- vitesse d'exécution **plus rapide**,
- vérifie les types ➔ pas d'affichage erroné,
- s'utilise avec les types utilisateurs.

Rq : pour ne pas écrire ***`std::`****`cout`* ajouter *`using namespace std;`* en début de fichier.

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

int main() {
	int entier;
	cout << "Entrez un entier : ";
	cin >> entier;
	cout << "entier = " << entier << endl;

	char chaine[100];
	cout << "Entrez une phrase : ";
	cin >> chaine;
	cout << chaine << endl;

	return 0;
}
```

```
Entrez un entier : 2024
entier = 2024
Entrez une phrase : C++ est mieux que C
C++
```

Les variables

- peuvent être déclarées **n'importe où**,
- ont leur portée de l'**endroit de leur déclaration** jusq à la **fin du bloc courant**.

L'opérateur ***: :*** permet d'**accéder aux variables globales** plutôt qu'aux locales.

```
Bloc intérieur : 15
Bloc intérieur, réf à la variable du niveau sup : 5
Fonction principale : 10
```

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

// Variable globale
int maVariable = 5;

int main() {
    // Variable locale de la fonction principale
    int maVariable = 10;

    for (int i = 0; i < 1; i++) {
        // Nouvelle variable dans un bloc intérieur
        int maVariable = 15;

        // Accès à la variable du bloc intérieur
        cout << "Bloc intérieur : " << maVariable << endl;

        // Accès à la variable du niveau supérieur
        cout << "Bloc intérieur, réf à la variable du
                  niveau sup : " << ::maVariable << endl;
    }

    // Accès à la variable de la fonction principale
    cout << "Fonction principale : " << maVariable << endl;

    return 0;
}
```

Les constantes et les structures

Les constantes :

- *`const`*
- n'est **pas modifiable** durant *toute la durée* de sa vie,
- doit être **initialisée** *au moment* de sa définition.

```
1 const int JOURS_PAR_SEMAINE = 7;
```

Les stuctures :

- sont similaires à C,
- *`typedef`* n'est **plus obligatoire** pour renommer un type.

```
1 struct Personne {
2     char* prenom;
3     int age;
4 };
5
6 Personne unPersonne, uneAutrePersonne;
```

Gestion de la mémoire

Les variables références

Une référence :

- variable “**synonyme**” d’une autre,
- *affecte* le contenu de **l’autre** *si* elle est *modifiée*,
- *doit être **initialisée***,
- le ***type*** de la *variable **initial*** doit être **le même** que celui de la *variable **référence***.

Différence avec les pointeurs :

- *initialisation **obligatoire***,
- toutes opérations sur la référence ***agit** sur la **variable référencé*** (et pas sur l’adresse),
- la valeur de référence *ne peut pas être **modifiée***.

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      int entier;
6      // refEntier : référence de la variable entier
7      int & refEntier = entier;
8      int *pointeur;
9      entier = 1;
10     cout << "entier = " << entier << " refEntier = " <<
                                          refEntier << endl;
11     refEntier = 2;
12     cout << "entier = " << entier << " refEntier = " <<
                                          refEntier << endl;
13     pointeur = &refEntier;
14     *pointeur = 3;
15     cout << "entier = " << entier << " refEntier = " <<
                                          refEntier << endl;
16     return (0);
17 }
```

```
entier = 1 refEntier = 1
entier = 2 refEntier = 2
entier = 3 refEntier = 3
```

L’allocation mémoire

Deux opérateurs :

- `new` : remplace *`malloc`*, **réserve** l’espace mémoire et l'**initialise**,
- `delete` : remplace *`free`*, **libère** l’espace mémoire alloué par *`new`* à un **seul objet**,
- `delete[]` : **libère** l’espace mémoire alloué à un **tableau d’objets**.

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      int *ptr1, *ptr2, *ptr3;
6
7      ptr1 = new int;
8      ptr2 = new int[10];
9      ptr3 = new int(10);
10
11     delete ptr1;
12     delete[] ptr2;
13     delete ptr3;
14
15     struct sDate {int jour, mois, an;};
16     sDate *ptr4, *ptr5, *ptr6, date = {25, 4, 1952};
17
18     ptr4 = new sDate;
19     ptr5 = new sDate[10];
20     ptr6 = new sDate(date);
21
22     delete ptr4;
23     delete[] ptr5;
24     delete ptr6;
25
26     return (0);
27 }
```

Les Fonctions

Passage de paramètres par valeur

Une **copie** de l'argument est passée à la fonction qui **ne modifie pas** l'original.

```
1  #include <iostream>
2  using namespace std;
3
4  void echange(int a, int b) {
5         int temp = a;
6         a = b;
7         b = temp;
8  }
9
10 int main() {
11        int i=2, j=3;
12        echange(i, j);
13        cout << "i = " << i << " j = " << j << endl;
14        return (0);
15 }
```

```
i = 2 j = 3
```

Passage de paramètres par pointeur (ou par adresse)

L'**adresse** de l'argument est passée à la fonction qui **manipule** les adresses.

```
1  #include <iostream>
2  using namespace std;
3
4  void echange(int * a, int * b) {
5         int temp = * a;
6         *a = *b;
7         *b = temp;
8  }
9
10 int main() {
11        int i=2, j=3;
12        echange(&i, &j);
13        cout << "i = " << i << " j = " << j<< endl;
14        return (0);
15 }
```

```
i = 3 j = 2
```

Passage de paramètres par référence

L'argument est passée par **référence** à la fonction qui le **manipule** et **répercute** les modifications sur le paramètre réel.

```
1  #include <iostream>
2  using namespace std;
3
4  void echange(int &a, int &b) {
5         int temp = a;
6         a = b;
7         b = temp;
8  }
9
10 int main() {
11        int i=2, j=3;
12        echange(i, j);
13        cout << "i = " << i << " j = " << j<< endl;
14        return (0);
15 }
```

```
i = 3 j = 2
```

Bilan passage de paramètres

➔ **Favoriser le passage par référence.**

Ajout du mécanisme `const` à un **paramètre** :

- **signale** au compilateur qu'il **ne doit pas modifier** ces valeurs recopiées,
- sécurité des programmes,
- aide à la correction,
- **excellente habitude**.

Ne pas confondre :

- *`const type * ptr`* : ptr est un pointeur sur un objet constant,
- *`type * const ptr`* : ptr est un pointeur constant sur un objet type !

Valeur par défaut des paramètres

- Possibilité de déclarer des **valeurs par défaut** dans le prototype de la fonction.

```
1  #include <iostream>
2  using namespace std;
3
4  void uneFonction(int a, float b=10.0, int c=10) {
5        cout << "Paramètres de uneFonction : " << a << " " << b << " " << c <<endl;
6  }
7
8  int main() {
9        uneFonction(1, 1.0, 2);
10       uneFonction(1, 1.0);
11       uneFonction(1);
12       return (0);
13 }
```

```
Paramètres de uneFonction : 1 1 2
Paramètres de uneFonction : 1 1 10
Paramètres de uneFonction : 1 10 10
```

Rq : les paramètres par défaut doivent être les **derniers de la liste**.

Les Classes et les Objets

Classe

- ensembles de *sous-programmes* et de *données* ***membres***,
- ***sections*** *délimitées* par les mots-clés `public`, `private` et `protected`,
- **méthodes** (ou sous-programmes membres)
  - sont définies :
    - dans un **module séparé** (*Personne.cpp*),
    - plus loin dans le code source (*Personne.h*) – déconseillé
  - ont leur nom précédé par le nom de la classe et **`::`**
  - ont le pointeur `this` :
    - **paramètre *caché*** qui permet d'accéder à l'**objet** *qui appelle* la méthode.

```cpp
// Personne.h
#ifndef PERSONNE_H
#define PERSONNE_H

#include <string>

class Personne {
public:
    // Méthodes publiques
    void afficherDetails();

private:
    // Membres de la classe (privés)
    std::string _nom;
    int _age;
};

#endif // PERSONNE_H
```

```cpp
// Personne.cpp
#include "Personne.h"
#include <iostream>
using namespace std;

void Personne::afficherDetails() {
    cout << "Nom : " << this->_nom << ",
       Age : " << _age << " ans." << endl;
}
```

Constructeurs et Destructeurs

- compilateur les crée *par défaut* mais peuvent être *personnalisés*,
  - si on a personnalisé, alors le compilateur *ne propose plus* de constructeur par défaut.
  - **un seul** destructeur et **un ou plusieurs** constructeurs (*polymorphisme*)

```h
// Personne.h
#ifndef PERSONNE_H
#define PERSONNE_H

#include <string>
using namespace std;

class Personne {
public:
    // Constructeur
    Personne(); // par défaut
    Personne(string _nom, int _age); // personnalisé

    // Destructeur
    ~Personne();

    // Méthodes publiques
    void afficherDetails();

private:
    // Membres de la classe (privés)
    string _nom;
    int _age;
};

#endif // PERSONNE_H
```

```cpp
// Personne.cpp
#include "Personne.h"
#include <iostream>
using namespace std;

// Constructeur par défaut
Personne::Personne() { }

// Constructeur personnalisé
Personne::Personne(string nom, int age) {
    this->_nom = nom;
    this->_age = age;
}

// Destructeur
Personne::~Personne() {
    cout << "Destruction de l'objet Personne : " <<
                                this->_nom << endl;
}

void Personne::afficherDetails() {
    cout << "Nom : " << _nom << ", Age : " << _age
         << " ans." << endl;
}
```

Accesseurs et mutateurs

- cf. ch2 s10

```cpp
// Personne.h
#ifndef PERSONNE_H
#define PERSONNE_H

#include <string>
using namespace std;

class Personne {
public:
    ...
    // Accesseurs
    string getNom() const;
    int getAge() const;

    // Mutateurs
    void setNom(const string& nom);
    void setAge(int age);

private:
    // Membres de la classe (privés)
    string _nom;
    int _age;
};

#endif // PERSONNE_H
```

```cpp
// Personne.cpp
#include "Personne.h"
#include <iostream>
using namespace std;

...

// Accesseurs
string Personne::getNom() const {
    return _nom;
}

int Personne::getAge() const {
    return _age;
}

// Mutateurs
void Personne::setNom(const string& nom) {
    _nom = nom;
}

void Personne::setAge(int age) {
    _age = age;
}
```

Objet

- est une **instance** d'une classe,
- **accès** aux *attributs* et *méthodes* :
  - `.` pour les variables
  - `->` pour les pointeurs

```
1  // main.cpp
2  #include "Personne.h"
3
4  int main() {
5
6        // Déclaration d'objet
7        Personne unePersonne;            // une instance simple
8        Personne groupe[10];             // un tableau
9        Personne *uneAutrePersonne;      // un pointeur non initialisé
10
11       // Création dynamique
12       uneAutrePersonne = new Personne();
13
14       // Manipulation attribut
15       unePersonne.setNom("Brunschwig");
16       unePersonne.setAge(28);
17       groupe[0].setNom("Simon");
18       groupe[0].setAge(19);
19       uneAutrePersonne->setNom("Martin");
20       uneAutrePersonne->setAge(51);
21
22       // Manipulation méthode
23       unePersonne.afficherDetails();
24       groupe[0].afficherDetails();
25       uneAutrePersonne->afficherDetails();
26
27       return (0);
28 }
```

Fonctions particulières

Fonctions amies

- `friend` : casse le mécanisme d'encapsulation
  - **entorse** aux concepts de la POO – à éviter

Les *friends* peuvent être :

- des fonctions **indépendantes**,
- des fonctions **membres d'une autre classe**,
- des fonctions **amies de plusieurs classes**,
- des **classes amies**.

```h
// Personne.h
    ...

class Personne {
public:
    ...
    // Déclaration de la fonction amie
    friend bool ontMemeAge(const Personne& p1, const
                                        Personne& p2);
private:
    ...
};
```

```cpp
// Personne.cpp
#include "Personne.h"
#include <iostream>
using namespace std;

...

// Implémentation de la fonction amie
bool ontMemeAge(const Personne& p1, const Personne& p2) {
    return p1._age == p2._age;
}
```

```cpp
// main.cpp
#include "Personne.h"

int main() {
    Personne unePersonne("Brunschwig", 28);
    Personne uneAutrePersonne("Martin", 19);
    if(ontMemeAge(unePersonne, uneAutrePersonne)) {
        std::cout << "Même âge." << std::endl;
    } else {
        std::cout << "Pas le même âge." << std::endl;
    }
    return (0);
}
```

```
Pas le même âge.
```

Méthodes constantes

- indiquent qu'une méthode *n'est pas* **intrusive**,
  - pas de modification du pointeur `this`.

- **obligatoire** pour porter sur un *objet* ***constant***.

```
1  // Accesseur
2  string Personne::getNom() const {
3      return _nom;
4  }
5
6  void uneFonction(const Personne& p) {
7       p.getNom(); // OK
8  }
```

Surcharge des opérateurs

Règles d'utilisation de la surcharge d'opérateurs

Objectif : Étendre les *opérateurs de base* pour qu'ils puissent s'**appliquer** *aux objets*.

Règles :

- veiller à **respecter** l'*esprit de l'opérateur*,
- on *ne peut pas* surcharger pour les *types de base* (int, …)
- il n'est **pas possible** de :
  - changer sa *priorité*,
  - changer son *associativité*,
  - changer sa *pluralité* (unaire, binaire, ternaire),
  - créer de *nouveaux opérateurs*.

```
// Personne.h
    …

class Personne {
public:
    ...
    // Surcharge de l'opérateur d'affectation
    Personne& operator=(const Personne& autre);
private:
    ...
};
```

```
// Personne.cpp
    ...

Personne& Personne::operator=(const Personne& autre) {
    if (this != &autre) { // Vérifier l'auto-affectation
        _nom = autre._nom;
        _age = autre._age;
    }
    return *this;
}
```

```
// main.cpp
    ...

int main() {
    Personne unePersonne("Brunschwig", 28);
    Personne uneAutrePersonne;

    // Utilisation de l'opérateur d'affectation
    unePersonne = uneAutrePersonne;

    return (0);
}
```

Implémentation

Deux implémentations possibles :

- Fonction **membre** :
  - accès à `this`
- Fonction **non membre** :
  - doit être *amie* pour accéder aux membres

Comment choisir entre membre et non membre ?

- `=`, `()`, `[]`, `->` : nécessairement fonctions membres,
- **Opérateurs unaires** : fonctions membres,
- **Opérateurs d'affectation** : fonctions membres,
- **Opérateurs binaires** : fonctions non membres amies

Il est recommandé de redéfinir :

- l'opérateur d'**affectation** `=`,
- les opérateurs d'**égalité** `==` et de **différence** `!=`,
- les opérateurs `<<` et `>>`,
- l'opérateur d'**indexation** `[]`.

```
// Personne.h
    ...
class Personne {
public:
    ...
    //membre
    void operator+=(int);
    // non membre
    friend ostream& operator<<(ostream&, const Personne&);
private:
    ...
};
```

```
// Personne.cpp
    ...
// membre
void Personne::operator+=(int entier) {
    age += entier;
}
// non membre
ostream& operator<<(ostream& out, const Personne& p) {
    cout << p._age << " ans" << endl;
    return out;
}
```

```
a = b + c;
a = operator+(b, c); // fonction globale
a = b.operator+(c);  // fonction membre
//les trois lignes sont équivalentes
```

La forme Canonique de Coplien

La forme canonique de Coplien

- concerne les classes **non triviales**,
- fournit un **cadre à respecter**.

Définition :

Une classe **T** est dite sous la **forme canonique** (ou forme normale) si elle présente les méthodes suivantes :

- constructeur par **défaut**,
- constructeur par **recopie**,
- **destructeur**,
- opérateur d'**affectation**.

```
1  class T {
2  public:
3       T();
4       T(const T&);
5       ~T();
6       T &operator=(const T&);
7  };
```

Coplien : *Constructeur par défaut*

- ***n'a pas** d'argument* ou des *argument avec une **valeur par défaut***,
- si *aucun* constructeur n'est spécifié, le *compilateur en **fournit un** par défaut*,

```
1  class T {
2  public:
3       T(); // Constructeur par défaut
4       T(const T&);
5       ~T();
6       T &operator=(const T&);
7  };
```

Exemple d'utilisation : *lorsque l'on crée des tableaux d'objets (impossible de passer un constructeur à l'opérateur* `new []`*).*

Coplien : *Constructeur par recopie*

- **création** d'un objet **à partir d'**un *autre objet*,
  - hors création ➜ *opérateur d'affectation*,
- pour **passer** un objet *par valeur* à une méthode,
  - garantie de *ne pas modifier* l'objet *original*,
- pour **retourner** un *objet*,
  - crée un objet placé dans une pile durant sa manipulation,
- si *aucun* constructeur n'est spécifié, le *compilateur en **fournit un** par défaut*,
  - recopie *par défaut* ne recopie que les *pointeurs*.

```
1  class T {
2  public:
3       T();
4       T(const T&); // Constructeur par recopie
5       ~T();
6       T &operator=(const T&);
7  };
```

Coplien : *Opérateur d'affectation*

- **duplication** d'un objet **à partir** d'un *autre objet*,
- **mêmes règles** que le constructeur *par recopie*,
- si l'objet recevant l'affectation *n'est pas vierge*, le **destructeur** est *appelé avant* la recopie.

```
1  class T {
2  public:
3       T();
4       T(const T&);
5       ~T();
6       T &operator=(const T&); // Opérateur d'affectation
7  };
```

L’héritage

Héritage

- précisé à la **déclaration** de la classe dérivée avec le *mode d'**encapsulation***,

```
1 class A {
2     // membres
3 };
4
5 class B : public A {
6     // membres ajoutés ou redéfinies
7 };
```

Héritage et encapsulation

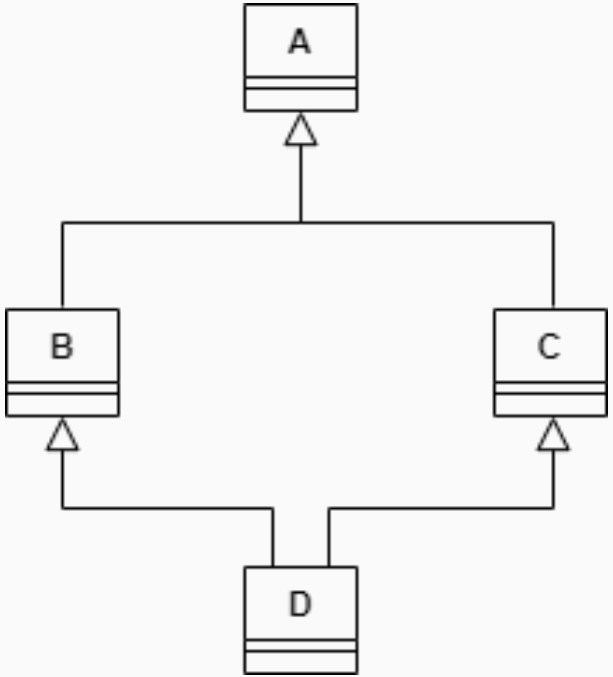
Mode	Statut dans la classe de base	Statut dans la classe dérivée
public	public	public
	protected	protected
	private	inaccessible
protected	public	protected
	protected	protected
	private	inaccessible
private	public	private
	protected	private
	private	inaccessible

Héritage multiple

- liste séparée de *virgule* des classes **mères**,
- **mode d'encapsulation** *précisé* pour *chaque* classe de base,
- **opérateur de portée : :** précise la classe du membre à accéder lors des conflits de nom.

```
1  class A {
2  public:
3        void fA();
4  protected:
5        int _x;
6  };
7
8  class B {
9  public:
10       void fB();
11 protected:
12       int _x;
13 };
14
15 class C : public A, public B {
16 public :
17       void fC();
18 };
19
20 voic C::fC() {
21       int i;
22       fA();
23       i = A::_x + B::_x;
24 }
```

Diamond problem




```
1  class A {
2  public:
3        int attribut;
4        void print();
5  };
6  class B : public A {};
7  class C : public A {};
8  class D : public B, public C {};
9
10 void A::print() {
11       cout << "attribut = " << this->attribut << endl;
12 }
13
14 int main() {
15       D d;
16       // d.attribut = 2;  ERROR
17       d.B::attribut = 2;
18       d.C::attribut = 3;
19       // d.print()         ERROR
20       d.B::print();
21       d.C::print();
22       return 0;
23 }
```

```
attribut = 2
attribut = 3
```

Méthodes virtuelles et classes abstraites

Méthodes virtuelles

- `virtual` : **supplante** une *fonction membre* d'une *classe parent* depuis une *classe dérivée*, à condition qu'elle ait la **même signature**,

Remarques :

- un constructeur *ne peut pas être **virtuel***,
- si **une** méthode est **virtuelle** alors le *destructeur* doit **l'être** aussi.

```
A::f1()
A::f2()
B::f1()
B::f2()
A::f1()
A::f2()
A::f1()
B::f2()
A::f1()
B::f2()
```

```
1  class A {
2  public:
3          void f1() { cout << "A::f1()" << endl; }
4          virtual void f2() { cout << "A::f2()" << endl;}
5  };
6
7  class B : public A {
8  public:
9          void f1() { cout << "B::f1()" << endl; }
10         void f2() { cout << "B::f2()" << endl; }
11 };
12
13 int main () {
14         A a;
15         a.f1();
16         a.f2();
17         B b;
18         b.f1();
19         b.f2();
20         // copie non polymorphe
21         a = b;
22         a.f1();
23         a.f2();
24         // utilisation polymorphe de B (par pointeur)
25         A *ptrA = &b;
26         ptrA->f1();
27         ptrA->f2();
28         // utilisation polymorphe de B (par référence)
29         A &refA = b;
30         refA.f1();
31         refA.f2();
32         return 0;
33 }
```

Classes abstraites

- **pas d'instance** possible,
- **au moins une** méthode **virtuelle pure**.

Méthode virtuelle pure :

- *fonction virtuelle suivie de "= 0"* dans sa déclaration,
- *doit être supplantée* par une fonction d'une classe fille.

```
1  class Personne {
2  public:
3       // Méthode virtuelle pure
4       virtual void afficherDetails() = 0;
5
6       // Destructeur virtuel
7       virtual ~Personne();
8  };
```

Généricité

Classes génériques

- crée un **modèle** pour implémenter **un même concept** pour des **types différents**,
- **types** en paramètre de la classe pour **construire des instances spécifiques**.

Un template n'est pas une classe :

- pas de compilation du template,
- seules les classes générées sont "*compatibles*".

```cpp
int main() {
    // Instanciation d'une boite avec un entier
    Boite<int> boiteEntier(42);

    // Affichage du contenu de la boite
    boiteEntier.afficherContenu();

    return 0;
}
```

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

// Classe générique (template)
template <typename T> class Boite {
private:
    T contenu;

public:
    // Constructeur
    Boite(const T& valeur) {
        contenu = valeur;
    }

    // Méthode générique pour afficher le contenu
    void afficherContenu() const {
        cout << "Contenu de la boite : " << contenu << endl;
    }

    // Méthode générique qui compare le contenu avec une autre boite
    template <typename U>
    bool estIdentique(const Boite<U>& autreBoite) const {
        return contenu == autreBoite.getContenu();
    }

    // Accesseur pour obtenir le contenu
    T getContenu() const {
        return contenu;
    }
};
```

Gestion des erreurs

Gestion des erreurs

Loi de Murphy :

- Une erreur est une *circonstance exceptionnelle*,
- Nécessite de se préparer aux circonstances exceptionnelles.

Traiter les circonstances exceptionnelles :

- (1) *interrompre* le programme,
- (1) *informer* l'utilisateur
  (2) *arrêter* correctement le programme,
- (1) *informer* l'utilisateur
  (2) essayer de *corriger*
  (3) *poursuivre* l'exécution,
- (1) *traiter* l'erreur
  (2) en *poursuivant l'exécution* du programme sans inquiéter l'utilisateur.

Types d'erreurs :

- faute de syntaxe,
- erreurs logiques (mauvaise solution à un problème),
- problèmes systèmes inhabituels mais prévisibles (manque de ressources).

Exceptions

Gestion des exceptions :

- `try` : identifier et définir la **portion de code à risque**,
- `catch` : définit le type d'exception que l'on traite,
  - appelé le **gestionnaire**, peut y en avoir plusieurs consécutifs à un bloc *`try`*.

```
1  try {
2       uneFonctionDangereuse();
3  } catch (...) {
4  }
```

Démarche :

1. *identifier* les zones à risques,
2. *mettre* dans un bloc `try`,
3. *créer* des blocs `catch` pour récupérer les exceptions générées,
4. bloc `catch` pour *chaque type* d'exception spécifique.

```cpp
int main() {
     int haut = 90;
     int bas = 0;

     try {
          cout << "haut / 2" << (haut/2) << endl;
          cout << "haut / bas" << (haut / bas) << endl;
          cout << "haut / 3" << (haut/3) << endl;
     } catch (...) {
          cout << "Il y a eu un problème !" << endl;
     }
}
```

```
Il y a eu un problème !
```

Créer ses propres exceptions

- créer ses propres gestionnaires pour les exceptions significatives de l'application,
- `throw` : amène l'instruction `try` à réagir.

Fonctionnement de la détection des exceptions :

1. Une exception se déclenche, la pile des appels est examinée ;
2. Déroulement de la pile pour chaque bloc du code ;
3. À mesure que la pile est déroulée, les destructeurs locaux sont appelés ;
4. Si un bloc `catch` correspond à l'exception, celle-ci est considérée comme traitée,
5. Si le déroulement remonte jusqu'au `main`, appel du gestionnaire par défaut.

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

// Classe d'exception personnalisée
class MonException {
public:
    string message;

    // Constructeur prenant un message en paramètre
    MonException(const string msg) {
        message = msg;
    }
};

// Fonction qui lance une exception personnalisée
void fonctionAvecException() {
    throw MonException("Une erreur personnalisée s'est produite !");
}

int main() {
    try {
        fonctionAvecException();
    } catch (const MonException& e) {
        cerr << "Exception attrapée : " << e.message << endl;
    }

    return 0;
}
```

Exceptions multiples

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

class MonExceptionA {
public:
    string message;
    MonExceptionA() {
        message = "Exception de type MonExceptionA";
    }
};

class MonExceptionB {
public:
    string message;
    MonExceptionB() {
        message = "Exception de type MonExceptionB";
    }
};

int fonctionA(int valeur) {
    if (valeur < 0) {
        throw MonExceptionA();
    }
    return valeur * 2;
}
```

```cpp
int fonctionB(int valeur) {
    if (valeur == 0) {
        throw MonExceptionB();
    }
    return 10 / valeur;
}

int main() {
    try {
        int resultatA = fonctionA(-5);
        int resultatB = fonctionB(0);

        cout << "Résultat A : " << resultatA << endl;
        cout << "Résultat B : " << resultatB << endl;
    } catch (const MonExceptionA& e) {
        cerr << "Exception attrapée : " << e.message <<
                                                    endl;
    } catch (const MonExceptionB& e) {
        cerr << "Exception attrapée : " << e.message <<
                                                    endl;
    } catch (const exception& e) {
        cerr << "Exception inattendue ! " << endl;
    }

    return 0;
}
```

```
Exception attrapée : Exception de type MonExceptionA
```

Références

Le contenu de ce cours est basé sur les supports pédagogiques des Dr. Nicolas Belloir et Dr Samson Pierre.