

Programmation Modulaire Objets et Classes

Viviane Pons

Mail : `viviane.pons@universite-paris-saclay.fr`

Page professionnelle : `http://www.lri.fr/~pons`

3 février 2025

1 Qu'est-ce que la programmation par objets ?

2 Notion de classe

- Déclaration et définition des méthodes
- Modification et méthodes constantes
- Le pointeur `this`
- Compléments

3 Les constructeurs

- Digression : rappel sur les exceptions
- Constructeur : introduction
- Constructeur : déclaration, définition, appel
- Constructeurs spéciaux
 - Constructeur par défaut
 - Délégation de constructeur
 - Constructeurs de conversion
 - Gestion des ressources

4 Bilan

Plan

- 1 Qu'est-ce que la programmation par objets ?
- 2 Notion de classe
- 3 Les constructeurs
- 4 Bilan

Pourquoi la programmation par objets ?

Résumé des épisodes précédents : on a vu un certain nombre de techniques de programmation pour **organiser ses données de manière hiérarchique** :

- Les **données**, stockées dans des **variables**, sont organisées grâce à leurs **types** (structures, énumérations, tableaux. .).
- Les **propriétés** de ces données sont définies par la collection des **opérations que l'on peut leur appliquer** (décrites par des fonctions et opérateurs).

Pourquoi la programmation par objets ?

On a des problèmes **méthodologiques** quand on veut vérifier qu'un code est correct.

Exemple : une fraction (modélisée par le type Rat)

```
struct Rat { int numer, denom; };
```

est **correcte** si

- denom est non nul et positif;
- numer et denom sont premiers entre eux.

Mais, **rien n'empêche le programmeur d'écrire**

```
Rat r = {4, 2};
void maFonction(Rat r) {
    ...
    r.denom = 0;
    ...
}
```

Pourquoi la programmation par objets ?

Problème

- Il n'est pas clair de savoir **quelles sont les opérations que l'on peut appliquer à une variable donnée.**
- Très souvent les données doivent respecter des **contrainte d'intégrité** (en informatique, on dit «**invariant**»)
- pour le vérifier, on est forcé d'**inspecter tout le code**, ce qui inclut le code client. **C'est n'est pas faisable.**

Exemple d'**invariant** :

```
struct Rat { int numer, denom; };
```

Un rationnel est **correct** si

- **denom** est non nul et positif;
- **numer** et **denom** sont premiers entre eux.

Pourquoi la programmation par objets ?

Problème

- Il n'est pas clair de savoir **quelles sont les opérations que l'on peut appliquer à une variable donnée.**
- Très souvent les données doivent respecter des **contrainte d'intégrité** (en informatique, on dit «**invariant**»)
- pour le vérifier, on est forcé d'**inspecter tout le code**, ce qui inclut le code client. **C'est n'est pas faisable.**

Exemple d'**invariant** :

```
struct Rat { int numer, denom; };
```

Un rationnel est **correct** si

- `denom` est non nul et positif;
- `numer` et `denom` sont premiers entre eux.

Pourquoi la programmation par objets ?

Problème

- Il n'est pas clair de savoir **quelles sont les opérations que l'on peut appliquer à une variable donnée.**
- Très souvent les données doivent respecter des **contrainte d'intégrité** (en informatique, on dit «**invariant**»)
- pour le vérifier, on est forcé d'**inspecter tout le code**, ce qui inclut le code client. **C'est n'est pas faisable.**

Exemple d'**invariant** :

```
struct Rat { int numer, denom; };
```

Un rationnel est **correct** si

- `denom` est non nul et positif;
- `numer` et `denom` sont premiers entre eux.

Pourquoi la programmation par objets ?

Problème

- Il n'est pas clair de savoir **quelles sont les opérations que l'on peut appliquer à une variable donnée.**
- Très souvent les données doivent respecter des **contrainte d'intégrité** (en informatique, on dit «**invariant**»)
- pour le vérifier, on est forcé d'**inspecter tout le code**, ce qui inclut le code client. **C'est n'est pas faisable.**

Exemple d'**invariant** :

```
struct Rat { int numer, denom; };
```

Un rationnel est **correct** si

- `denom` est non nul et positif;
- `numer` et `denom` sont premiers entre eux.

Idée de la programmation objet

Problème

On a besoin d'avoir **la liste des fonctions** qui accèdent à la structure et qui sont **susceptibles de casser les invariants**.

Pour ceci, on va réorganiser le code :

- **rassembler les fonctions** qui accèdent à la structure
- **contrôler** quelles fonctions ont droit de modification
- **séparer** clairement la structure et les fonctions qui la manipulent du code client qui l'utilise

Idée de la programmation objet

Problème

On a besoin d'avoir **la liste des fonctions** qui accèdent à la structure et qui sont **susceptibles de casser les invariants**.

Pour ceci, on va réorganiser le code :

- **rassembler les fonctions** qui accèdent à la structure
- **contrôler** quelles fonctions ont droit de modification
- **séparer** clairement la structure et les fonctions qui la manipulent du code client qui l'utilise

Qu'est-ce qu'une classe ?

Mettre les données au centre des programmes !

Retenir

Une *classe* est composée de

- une structure (déclarée par `struct` ou `class`)
- des *fonctions membres* (que l'on appelle aussi méthodes)
- des *droits d'accès* (`public`, `private`, `protected`)
- des *opérateurs associés* (`<<`, `+`, ...)

Qu'est-ce que la programmation objet ?

Compléments

La programmation objet c'est la notion de **classe** plus des techniques de **réutilisation de code** : héritage, programmation générique, interface, patron (angl. template).

Ce cours est une initiation à la notion d'objet. On verra les notions en deux temps :

- 1 objets, classes, fonctions membres, constructeurs
- 2 programmation modulaire, encapsulation, interface, implémentation

Attention

Les notions avancées de réutilisation de code seront vue en deuxième année.

Qu'est-ce que la programmation objet ?

Compléments

La programmation objet c'est la notion de **classe** plus des techniques de **réutilisation de code** : héritage, programmation générique, interface, patron (angl. template).

Ce cours est une initiation à la notion d'objet. On verra les notions en deux temps :

- 1** objets, classes, fonctions membres, constructeurs
- 2** programmation modulaire, encapsulation, interface, implémentation

Attention

Les notions avancées de réutilisation de code seront vue en deuxième année.

Qu'est-ce que la programmation objet ?

Compléments

La programmation objet c'est la notion de **classe** plus des techniques de **réutilisation de code** : héritage, programmation générique, interface, patron (angl. template).

Ce cours est une initiation à la notion d'objet. On verra les notions en deux temps :

- 1 objets, classes, fonctions membres, constructeurs
- 2 programmation modulaire, encapsulation, interface, implémentation

Attention

Les notions avancées de réutilisation de code seront vues en deuxième année.

Qu'est-ce que la programmation objet ?

Compléments

Pour en savoir plus, je recommande la lecture de

«Programmation C++ : La couche objet»

sur wikibooks et en particulier la section d'introduction intitulée «Généralités».

Plan

- 1 Qu'est-ce que la programmation par objets ?
- 2 Notion de classe**
- 3 Les constructeurs
- 4 Bilan

Un mini exemple

entier.cpp

```

5  struct Entier {
6      int val;
7
8      /** La valeur de l'entier
9       * @return l'entier
10     **/
11     int get_val() const;
12
13     /** Change la valeur de l'entier
14     * @param[in] i : un entier
15     **/
16     void set_val(int i);
17
18 };
19
22 int Entier::get_val() const {
23     return val; // accès direct à val
24 }
25
26 void Entier::set_val(int i) {
27     val = i; // modification de val
28 }
```

Exemple d'utilisation de la
classe Entier :

entier.cpp

```

33 int main() {
36     Entier e, f;
37
38     e.set_val(4);
39     cout << "val e = " << e.get_val() << endl;
40     // affiche: 4
41     e.set_val(6);
42     f.set_val(12);
43     cout << "val e = " << e.get_val() << ", "
44     // affiche: 6
45     << "val f = " << f.get_val() << endl;
46     // affiche: 12
47 }
```

Plan

1 Qu'est-ce que la programmation par objets ?

2 Notion de classe

- Déclaration et définition des méthodes
- Modification et méthodes constantes
- Le pointeur `this`
- Compléments

3 Les constructeurs

4 Bilan

Rappel : déclaration et définition

Retenir

Une **déclaration** informe le compilateur (et aussi le lecteur) que l'on va utiliser quelque chose (variable, fonction, type, classe).

Bon endroit pour écrire la **documentation**.

Il faudra donner le code proprement dit c'est-à-dire la **définition**.

Déclaration d'une classe

Une classe est définie en deux temps :

- 1 **Déclaration** des attributs (champs) et méthodes (fonctions membres)
- 2 **Définition** des méthodes

Syntaxe (Déclaration d'une classe)

une `struct` avec déclarations de *fonctions membres* :

```
struct NomClasse {
    type nom_du_champs1;
    type nom_du_champs2;
    ...
    fonction_membre1;
    fonction_membre2;
    ...
};
```

Déclaration d'une fonction membre

Syntaxe

*La déclaration d'une **fonction membre** est identique à une déclaration de fonction*

```
type_de_retour nom_method(paramètres);
```

*sauf qu'il n'y a **pas de paramètre pour l'objet courant** sur lequel la fonction membre est appelée. Il sera passé **implicitement**.*

On écrit :

```
void set_val(int i);
```

et pas

```
void set_val(Entier/*this/ int i);
```

Note : **this** est en gros l'équivalent de self en Python.

Exemple de déclaration d'une classe

entier.cpp

```
5 struct Entier {  
6     int val;  
7  
8     /** La valeur de l'entier  
9      * @return l'entier  
10    **/  
11    int get_val() const;  
12  
13    /** Change la valeur de l'entier  
14     * @param[in] i : un entier  
15    **/  
16    void set_val(int i);  
17  
18 };
```

Définition d'une fonction membre

Syntaxe

La définition d'une **fonction membre** est très similaire à une définition de fonction :

```
type_de_retour NomClasse::nom_methode(paramètres) {  
    code de la fonction  
    return ...  
}
```

Différences :

- Le nom de la méthode est préfixé par **NomClasse::**
- Les champs, qu'on appelle aussi **attributs**, sont accessibles directement (pas de notation pointée)
- On peut **modifier les attributs**

Exemple de définition d'une fonction membre

Attention

À l'intérieur d'une fonction membre, on accède directement aux attributs sans passer par la notation pointée.

entier.cpp

```
22 int Entier::get_val() const {
23     return val; // accès direct à val
24 }
25
26 void Entier::set_val(int i) {
27     val = i;    // modification de val
28 }
```

Appel d'une fonction membre

Syntaxe

On utilise la **notation pointée** pour *appeler les méthodes* :

```
nom_var.nom_methode(paramètres)
```

l'objet contenu dans la variable nom_var est **transmis implicitement** à la méthode.

Exemple d'appel d'une fonction membre

entier.cpp

```
36  Entier e, f;
37
38  e.set_val(4);
39  cout << "val e = " << e.get_val() << endl;  // affiche: 4
40  e.set_val(6);
41  f.set_val(12);
42  cout << "val e = " << e.get_val() << ", "    // affiche: 6
43      << "val f = " << f.get_val() << endl;  // affiche: 12
```

Plan

1 Qu'est-ce que la programmation par objets ?

2 Notion de classe

- Déclaration et définition des méthodes
- Modification et méthodes constantes
- Le pointeur `this`
- Compléments

3 Les constructeurs

4 Bilan

Modification des attributs

Retenir

Les **modifications** des attributs dans une méthode ont lieu **directement sur l'objet**.

Il n'y a **pas de copie** de l'objet courant lors de l'appel d'une méthode (passage par pointeur).

entier.cpp

```

36  Entier e, f;
37
38  e.set_val(4);
39  cout << "val e = " << e.get_val() << endl;  // affiche: 4
40  e.set_val(6);
41  f.set_val(12);
42  cout << "val e = " << e.get_val() << ", "    // affiche: 6
43      << "val f = " << f.get_val() << endl;    // affiche: 12
    
```

Méthode constante

Retenir

*On peut signaler à l'utilisateur de la classe et au compilateur qu'**une méthode ne modifie pas l'objet** : on ajoute le mot clé **const** après la liste des paramètres, à la fois **dans la déclaration et la définition**.*

Dans le programme sur les entiers :

```
struct Entier {
    [...]
    int get_val() const;
    [...]
};

int Entier::get_val() const {
    return val;
}
```

Méthode constante : avantages

Retenir

Avantage des méthodes constantes :

- *documentation*
- meilleure *sûreté du code*
- *seules les méthodes constantes peuvent s'appliquer à un **objet constant** (variable déclarée `const`)*
- *permet certaines **optimisations** au compilateur*

⇒ On essayera de mettre systématiquement les `const`

Méthode constante : sûreté du code

Attention

Le compilateur signale une erreur si l'on essaye de modifier un champ dans une méthode constante :

```
entier.cpp:22:6: error: cannot assign to non-static data member  
                within const member function 'get_val'
```

```
    val++;  
    ~~~~
```

```
entier.cpp:21:13: note: member function 'Entier::get_val' is declared const here  
int Entier::get_val() const {  
~~~~~
```


Plan

1 Qu'est-ce que la programmation par objets ?

2 Notion de classe

- Déclaration et définition des méthodes
- Modification et méthodes constantes
- Le pointeur `this`
- Compléments

3 Les constructeurs

4 Bilan

Accès à l'objet lui-même

Problem

*Dans une méthode, on a parfois besoin d'**accéder à tout l'objet** lui-même au lieu d'un attribut individuel.*

Exemple : faire une **copie avant modification** :

```
struct Date {
    /** Avance une Date au lendemain */
    void avance();
    /** Le lendemain d'une Date */
    Date lendemain() const;
};

Date Date::lendemain() const {
    Date res = ??? copie de l'objet lui-même;
    res.avance();
    return res;
}
```

Accès à l'objet lui-même

Problem

*Dans une méthode, on a parfois besoin d'**accéder à tout l'objet** lui-même au lieu d'un attribut individuel.*

Exemple : faire une **copie avant modification** :

```
struct Date {  
    /** Avance une Date au lendemain **/  
    void avance();  
    /** Le lendemain d'une Date **/  
    Date lendemain() const;  
};
```

```
Date Date::lendemain() const {  
    Date res = *this;  
    res.avance();  
    return res;  
}
```

Le pointeur `this`

Retenir

- Lors d'un appel de méthode, le programme appelant **transmet implicitement l'adresse mémoire de l'objet courant**.
- Cette adresse est stockée dans un paramètre nommé `this`.
- `this` est un pointeur (de type `MaClasse*`).
- On accède à l'objet avec `*this`.

On pourrait donc écrire `(*this).attr` ou le raccourci `this->attr` pour accéder à l'attribut `attr` à l'intérieur d'une méthode. On ne le fait pas sauf cas très particulier.

Le pointeur `this`

Retenir

- Lors d'un appel de méthode, le programme appelant **transmet implicitement l'adresse mémoire de l'objet courant**.
- Cette adresse est stockée dans un paramètre nommé `this`.
- `this` est un pointeur (de type `MaClasse*`).
- On accède à l'objet avec `*this`.

On pourrait donc écrire `(*this).attr` ou le raccourci `this->attr` pour accéder à l'attribut `attr` à l'intérieur d'une méthode. On ne le fait pas sauf cas très particulier.

Le pointeur `this` dans d'autres langages

Compléments

Certains langages préfèrent être plus **explicite** !

En Python, on écrit toujours explicitement le pointeur (appelé usuellement `self`).

```
1 class MyClass:
2     ...
3     def setName(self, nm):
4         self._name = nm
5
6     def getName(self):
7         return self._name
```

En Java, `this` est implicite, mais beaucoup recommandent de l'écrire quand on accède à un attribut.

```
1 class MyClass {
2     String name;
3     void setName(String nm) {
4         this.name = nm;
5     }
6
7     String getName() {
8         return this.name;
9     }
```

Plan

1 Qu'est-ce que la programmation par objets ?

2 Notion de classe

- Déclaration et définition des méthodes
- Modification et méthodes constantes
- Le pointeur `this`
- Compléments

3 Les constructeurs

4 Bilan

Appel d'une autre méthode

Retenir

Il est possible d'appeler une autre méthode de l'objet courant dans une méthode : comme dans le cas des attribut, on écrit le nom de la méthode sans l'objet.

```
struct Date {  
    /** Le nombre de jours écoulé depuis le début janvier 01 */  
    int ordinal() const;  
    /** Le jour de la date (0=Dimanche) */  
    int jourSemaine() const;  
};  
  
int Date::jourSemaine() const {  
    return (ordinal() + 6) % 7;  
}
```


Méthode en ligne

Retenir

*Quand le code d'une méthode est très court (une ligne ou deux), on peut **mettre la définition directement dans la déclaration de la classe**.*

Plutôt que de générer un appel de fonction, le compilateur va simplement recopier le code (en ligne = angl. **inline**).

```
struct Date {  
    ...  
  
    /** Avance une Date de n jours **/  
    void avance(int n); // méthode normale définie plus tard  
  
    /** Avance une Date au lendemain **/  
    void avance() { avance(1); } // méthode en ligne  
};
```

Surcharge d'opérateur par méthode

Compléments

Alternativement à ce que l'on a déjà vu au chapitre 2, on peut **surcharger un opérateur avec une méthode**.

L'opération $a + b$ correspond à l'appel `a.operator+(b)` :

```
struct Date {  
    /** Avance une Date de n jours **/  
    void avance(int n);  
  
    Date operator+(int i) const;  
};  
  
Date Date::operator+(int i) const {  
    Date res = *this; res.avance(i); return res;  
}
```

Surcharge d'opérateur par méthode (2)

```
struct Date {
    bool operator==(const Date &d) const;
    void operator+=(int n) { avance(n); }

    bool operator<(const Date& autre) const;
};

bool Date::operator==(const Date &d) const {
    return jour == d.jour and mois == d.mois and annee == d.annee;
}

bool Date::operator<(const Date& autre) const {
    return (annee < autre.annee)
        || (annee == autre.annee &&
            ((mois < autre.mois) ||
             (mois == autre.mois && jour < autre.jour)));
}
```

Exemple complet

Voir le fichier en ligne !

```

10 struct Date {
11     int jour, mois, annee;
12
13     /** Teste si une Date est correcte **/
14     bool estCorrecte() const;
15     /** Demande une Date à l'utilisateur **/
16     void lit();
17
18     /** Avance une Date de n jours **/
19     void avance(int n);
20     /** Avance une Date au lendemain **/
21     void avance() { avance(1); }
22
23     /** Le lendemain d'une Date **/
24     Date lendemain() const;
25
26     bool estAvant(const Date& autre) const;
27     bool operator<(const Date& autre) const { return estAvant(autre); }
28
29     /** Le nombre de jours écoulé depuis le début janvier 01 **/
30     int ordinal() const;
31
32     /** Le nombre de jours écoulé entre this et autre **/
33     int difference(const Date& autre) const {
34         return autre.ordinal() - ordinal();
35     }
36     /** Le jour de la date (0=Dimanche) **/
37     int jourSemaine() const;
38
39     bool operator==(const Date& d) const {
40         return jour == d.jour and mois == d.mois and annee == d.annee;
41     }
42     bool operator!=(const Date& d) const { return not (*this == d); }
43
44     Date operator+(int i) const;
45     void operator+=(int n) { avance(n); }
46     int operator-(const Date& d) const { return difference(d); }
47 };
date.cpp

```

Plan

- 1 Qu'est-ce que la programmation par objets ?
- 2 Notion de classe
- 3 Les constructeurs**
- 4 Bilan

Plan

1 Qu'est-ce que la programmation par objets ?

2 Notion de classe

3 Les constructeurs

- Digression : rappel sur les exceptions
- Constructeur : introduction
- Constructeur : déclaration, définition, appel
- Constructeurs spéciaux

4 Bilan

Digression : rappel sur les exceptions

Voir le cours du premier semestre :

<https://nicolas.thiery.name/Enseignement/Info111/Devoirs/Semaine7/cours-exceptions.html>

Retenir (Rappel)

Exception : message permettant à une fonction de *signaler aux fonctions appelantes qu'elle ne peut pas faire son travail*. Remonte la pile d'appels jusqu'à ce que

- l'une des fonctions appelantes sache gérer le problème en **rattrapant l'exception**
- si pas de rattrapage **sortie du programme avec une erreur**

Quelques exceptions standards :

```
runtime_error, invalid_argument, out_of_range,  
length_error, logic_error, bad_alloc, system_error
```

Exceptions : exemple

exception-gestion.cpp

```
6  int factorielle(int n) {
7      if ( n < 0 )
8          throw invalid_argument("n doit être positif");
9      if ( n == 0 ) return 1;
10     return n * factorielle(n-1);
11 }
12
13 int main() {
14     int n;
15     cin >> n;
16     try {
17         cout << factorielle(n) << endl;
18     } catch (invalid_argument &e) {
19         cout << "Valeur de n invalide : " << e.what() << endl;
20     }
21 }
```


Plan

1 Qu'est-ce que la programmation par objets ?

2 Notion de classe

3 Les constructeurs

- Digression : rappel sur les exceptions
- **Constructeur : introduction**
- Constructeur : déclaration, définition, appel
- Constructeurs spéciaux

4 Bilan

Pourquoi les constructeurs ?

Si l'on maîtrise maintenant la liste des fonctions qui accèdent à une structure, on voudrait empêcher l'utilisateur de faire des **erreurs lors de la fabrication de l'objet**.

Problème

À la **création** de l'objet, on voudrait contrôler que les **contraintes d'intégrité (invariants)** sont bien mises en place.

Exemples d'erreurs possibles :

```
struct Rat { int numer, denom; };  
Rat r {1, 0};           // rationnel 1/0 invalide
```

```
struct Date {  
    int jour, mois, annee;  
};  
Date d {32, 01, 2000};  // date 32/01/2000 invalide
```

Pourquoi les constructeurs ?

Si l'on maîtrise maintenant la liste des fonctions qui accèdent à une structure, on voudrait empêcher l'utilisateur de faire des **erreurs lors de la fabrication de l'objet**.

Problème

À la **création** de l'objet, on voudrait contrôler que les **contraintes d'intégrité (invariants)** sont bien mises en place.

Exemples d'erreurs possibles :

```
struct Rat { int numer, denom; };  
Rat r {1, 0};           // rationnel 1/0 invalide  
  
struct Date {  
    int jour, mois, annee;  
};  
Date d {32, 01, 2000};  // date 32/01/2000 invalide
```

Quel est l'intérêt des constructeurs ?

Retenir

Un **constructeur** est une *méthode particulière utilisée pour construire un nouvel objet.*

On l'utilise pour :

- **remplir automatiquement** certains attributs
Exemple : construction d'un rationnel à partir d'un entier. On met automatiquement le dénominateur à 1.
- **vérifier** que l'on **ne construit que des objets valides**
Exemple : construction d'un rationnel à partir d'une fraction. On signale une erreur si le dénominateur est 0.
- **effectuer des calculs** pendant la construction
Exemple : construction d'un rationnel à partir d'une fraction. On veut simplifier la fraction.

Plan

1 Qu'est-ce que la programmation par objets ?

2 Notion de classe

3 Les constructeurs

- Digression : rappel sur les exceptions
- Constructeur : introduction
- **Constructeur : déclaration, définition, appel**
- Constructeurs spéciaux

4 Bilan

Rôle des constructeurs

Retenir

*Le but du constructeur est d'**initialiser un objet** :*

*Comme les méthodes, il travaille sur un **objet courant** dont l'emplacement mémoire est **alloué avant l'appel au constructeur**.*

Il a deux rôles :

- **contrôler les paramètres** de création de l'objet courant.
- **d'initialiser les attributs** de l'objet courant.

Attention

Un constructeur n'a **pas ni résultat, ni valeur de retour**.

Rôle des constructeurs

Retenir

*Le but du constructeur est d'**initialiser un objet** :*

*Comme les méthodes, il travaille sur un **objet courant** dont l'emplacement mémoire est **alloué avant l'appel au constructeur**.*

Il a deux rôles :

- **contrôler les paramètres** de création de l'objet courant.
- **d'initialiser les attributs** de l'objet courant.

Attention

Un constructeur n'a **pas ni résultat, ni valeur de retour**.

Syntaxe : déclaration d'un constructeur

Syntaxe (déclaration d'un constructeur)

- Porte le *nom de la classe, pas de valeur de retour*

Exemple : déclaration des constructeurs de la classe Rat :

```
struct Rat {
    ...
    Rat();                // constructeur par défaut
    Rat(int num, int den); // à partir d'une fraction
    Rat(int n);           // à partir d'un entier
    Rat(array<int, 2> r);  // à partir d'un tableau
    Rat(pair<int, int> p); // à partir d'une paire
    ...
};
```


Syntaxe : définition d'un constructeur

Syntaxe (définition d'un constructeur)

```
NomCls::NomCls(⟨ paramètres ⟩)⟨ : constr. attrs. ⟩ {
    ⟨ code construction ⟩
}
```

- *⟨...⟩ : optionnel*
- *Porte le **nom de la classe**, pas de valeur de retour*
- *Liste des paramètres : comme une fonction habituelle*
- *optionnellement : appel de **constructeur pour des attributs** (voir plus loin : appel d'un constructeur, page 40)*

```
NomAttr1{param. attr1}, NomAttr2{param. attr2}, ...
```

Exemples de définition d'un constructeur

date-constr.cpp

```
50 Date::Date(int j, int m, int a) : jour{j}, mois{m}, annee{a} {
51     if (mois <= 0 or mois > 12
52         or jour <= 0 or jour > nbJourMois(mois, annee))
53         throw invalid_argument("date incorrecte");
54 }
```

rational-class.cpp

```
117 Rat::Rat(int num, int den) : numer{num}, denom{den} {
118     if (denom == 0)
119         throw invalid_argument("denominateur nul");
120     if (denom < 0) {
121         numer = -numer;
122         denom = -denom;
123     }
124     int d = pgcd(numer, denom);
125     numer = numer / d;
126     denom = denom / d;
127 }
```

Définition d'un constructeur en ligne

Retenir

Comme dans le cas d'une méthode, si le code est très court, on peut directement mettre le constructeur dans la déclaration de la classe ([définition en ligne](#)).

Exemple : construction d'un rationnel à partir d'un entier :

```
struct Rat {  
    int numer, denom;  
  
    ...  
    Rat(int n) : numer{n}, denom{1} {}  
    ...  
};
```

Appel d'un constructeur

Syntaxe (Appel d'un constructeur)

La syntaxe conseillée est :

{ paramètres }

Si besoin est, on peut préciser le type avec :

NomDuType { paramètres }

Exemples d'appels de constructeurs

Initialisation d'une variable :

```
Date d{3, 12, 2020};
Rat zero{};
Rat rat12{1, 2};
int i{2};
```

Retour d'une fonction :

```
Date bugday() {
    return {1, 1, 2000};
}
```

```
Rat Rat::operator*(Rat b) const {
    return {numer * b.numer, denom * b.denom};
}
```

Appel de fonction :

```
d.estAvant({15, 7, 2021});
```

Objet anonyme avec le type :

```
Date{1, 1, 2000}.lendemain()
```

```
Rat{-12, 15}.abs()
```

```
Rat{1, 2} + 1
```

Appel d'un constructeur

Attention

Pour des raisons de compatibilité avec les anciennes normes du C++, il y a de nombreuses variantes qui rendent la situation confuse...

J'essaye de vous donner des règles simples et claires. **Elles ne s'appliquent que pour la norme 2011 du C++.**

Faites attention à bien **passer `-std=c++11` à votre compilateur.**

Note : même les experts trouvent la situation difficile :

- Nicolai Josuttis, *The Nightmare of Initialization in C++* (CppCon 2018)
<https://www.youtube.com/watch?v=7DT1WPgX6zs>.
- <https://h-deb.clg.qc.ca/Sujets/AuSecours/initialisation-uniforme.html>

Ça devrait aller mieux avec les normes 2017 et 2020 du C++...

Autres syntaxes d'appel d'un constructeur

Pour des raisons essentiellement historiques, le C++ fournit d'autres syntaxes de construction :

Compléments

L'initialisation d'une variable est souvent écrite :

```
NomType NomVar = { paramètres };
```

```
Date d = {3, 12, 2020};
```

Attention

On trouve également souvent la vieille syntaxe :

```
( paramètres )
```

Elle pose des problèmes d'ambiguïté et est donc déconseillée (avec la norme 2011 du C++).

Cas particulier des vecteurs

Attention

Dans le cas des vecteurs, parenthèses $\neq$ accolades.

- (long) : longueur (valeur par défaut)
- (long, val) : longueur et valeur initiale
- {valeurs} : liste de valeurs (version très courante)
- {{valeurs}} : liste de valeurs (version recommandée)

vector-constr.cpp

```

16 vector<int> v1(5);           // 5 éléments de valeur par défaut 0
17 vector<int> v2{5};           // 1 élément de valeur 5
18 vector<int> v3(5, 2);        // 5 élément de valeur 2
19 vector<int> v4{5, 2};        // 2 éléments de valeur 5 et 2
20 // vector<int> v5(1, 2, 3); // Err. : no matching constructor
21 vector<int> v6{1, 2, 3};      // 3 éléments de valeur 1, 2 et 3
22 vector<int> v7{{1, 2, 3}};    // 3 éléments de valeur 1, 2 et 3
    
```


Cas des tableaux

Retenir

*Dans le cas des **tableaux** ainsi que des **vecteurs**, je conseille d'utiliser systématiquement **deux paires d'accolades**, sauf dans le cas de la construction d'un **vecteur vide**.*

Exemples :

array-constr.cpp

```
array<int, 1> a1{{5}};           // éléments: 5
array<int, 5> a2{{2}};           // éléments: 2, 0, 0, 0, 0
array<int, 5> a3{{1, 2, 3}};     // éléments: 1, 2, 3, 0, 0
```

Cas particulier :

```
vector<int> v{};                 // Vecteur vide
vector<int> v{{}};               // Vecteur [0] !!!
```

Cas des tableaux (2)

Retenir

Dans le cas des *tableaux* ainsi que des *vecteurs*, je conseille d'utiliser systématiquement *deux paires d'accolades*, sauf dans le cas de la construction d'un *vecteur vide*.

Voici un exemple où les doubles accolades sont nécessaires, sinon le compilateur refuse le code car il y a une ambiguïté :

```
1  enum class Couleur { Rouge, VertClair, VertFonce,
2      Noir, Bleu, Orange, Violet, Vide };
3  struct Carte {
4      Couleur coul;
5      int     val;  // entre 1 et 9
6  };
7  using Groupe = array<Carte, 3>;
8  Groupe g1 {{ {Couleur::Rouge, 4}, {Couleur::Rouge, 5},
9      {Couleur::Rouge, 6} }};
```

Composition d'appels de constructeur

Retenir

*Pour l'initialisation de structures imbriquées, il suffit d'**imbriquer les accolades** en accord.*

struct-comp.cpp

```
struct Date {
    int jour, mois, annee;
    Date(int j, int m, int a) : jour{j}, mois{m}, annee{a} {}
};

struct AssureSocial {
    string nom, prenom; Date naissance;
};

array<AssureSocial, 2> a {{ // doubles accolades nécessaires
    {"Anna", "Konda", {7, 2, 2003}},
    {"Jean", "Némare", {12, 4, 2017}}
}};
```

Composition d'appels de constructeur (2)

Les règles données fonctionnent avec tableaux et vecteurs 2D :

struct-comp.cpp

```
array<array<AssureSocial, 2>,2> a2D {{
    {{ {"Marc", "Assin", {12, 4, 2017}},
      {"Oussama", "Lehrbon", {7, 2, 2003}}
    }},
    {{ {"Jean", "Peuplu", {11, 7, 1948}},
      {"Julie", "Dedroitagauche", {13, 1, 2001}}
    }}
}};

vector<vector<AssureSocial>> v2D {{
    {{ {"Gérard", "Menvussa", {11, 7, 1948}},
      {"Agathe", "Veblouse", {13, 1, 2001}}
    }},
    {{ {"Ted", "Oukoi", {12, 4, 2017}},
      {"Bill", "Oukoi", {7, 2, 2003}},
      {"Alain", "Térier", {5, 12, 2007}},
      {"Alex", "Térier", {5, 12, 2007}}
    }}
}};
```

Plan

1 Qu'est-ce que la programmation par objets ?

2 Notion de classe

3 Les constructeurs

- Digression : rappel sur les exceptions
- Constructeur : introduction
- Constructeur : déclaration, définition, appel
- **Constructeurs spéciaux**
 - Constructeur par défaut
 - Délégation de constructeur
 - Constructeurs de conversion
 - Gestion des ressources

4 Bilan

Constructeur par défaut

Après la déclaration :

```
struct Date {  
    int jour, mois, annee;  
    Date(int j, int m, int a);  
}
```

La ligne

```
    Date aujourd'hui;
```

déclenche l'erreur de compilation suivante :

```
date-constr.cpp:268:8: error: no matching constructor  
                        for initialization of 'Date'  
    Date aujourd'hui; // ERREUR !
```

Constructeur par défaut

Attention

Dès que l'on a déclaré un constructeur, les constructeurs normaux des structures sont supprimés. En particulier, **le constructeur avec aucun arguments n'est plus possible.**

Retenir

*Le constructeur sans arguments est appelé **constructeur par défaut**. Il est appelé systématiquement s'il l'on ne fournit pas d'initialisation.*

Si on veut un constructeur par défaut, il faut le définir !

Exemple : on construit par défaut le rationnel 0.

```
Rat::Rat() : numer{0}, denom{1} {}
```

Constructeur par défaut

Attention

Dès que l'on a déclaré un constructeur, les constructeurs normaux des structures sont supprimés. En particulier, **le constructeur avec aucun arguments n'est plus possible.**

Retenir

*Le constructeur sans arguments est appelé **constructeur par défaut**. Il est appelé systématiquement s'il l'on ne fournit pas d'initialisation.*

Si on veut un constructeur par défaut, il faut le définir !

Exemple : on construit par défaut le rationnel 0.

```
Rat::Rat() : numer{0}, denom{1} {}
```


Délégation d'un constructeur

Retenir

Un constructeur peut en appeler un autre de la même classe !

*On dit qu'il y a **délégation de constructeur** :*

```
NomClass(params) : NomClass(autre params) { ... }
```

rational-class.cpp

```
53 Rat(int num, int den);
54 Rat(int n) : numer{n}, denom{1} {} // conversion int -> Rat
55
56 Rat() : Rat{0} {} // constructeur par défaut et délégation
57 Rat(array<int, 2> r) : Rat{r[0], r[1]} {} // délégation
58 Rat(pair<int, int> p) : Rat{p.first, p.second} {} // délégation
```

Constructeur de conversion

Compl ments

Si un constructeur n'a qu'un **seul param tre**, c'est un **constructeur de conversion implicite (coercion)**. Il est appel  automatiquement si besoin est.

Exemple :

```
Rat(int n)  // conversion implicite int -> Rat
```

On peut alors  crire :

rational-class.cpp

```
147     CHECK(Rat{1, 2} + 1 == Rat{3, 2});  
148     CHECK(Rat{1, 2} + Rat{3, 2} == 2);  
149     CHECK(Rat{1, 2} - 1 == Rat{-1, 2});
```

Constructeur de conversion explicite

Compléments

Si un constructeur n'a qu'**un seul paramètre**, c'est un **constructeur de conversion**. Si l'on veut une conversion explicite, on préfixe le constructeur avec le mot clé `explicit`

Exemple :

```
explicit Rat(int n)  // conversion explicite int -> Rat
```

Exemple de constructeur de conversion explicite

Avec la déclaration :

```
explicit Rat(int n) // conversion explicite int -> Rat
```

la ligne

```
Rat x = Rat{1, 2} + 1;
```

déclenche l'erreur :

```
error: invalid operands to binary expression ('Rat' and 'int')
```

```
Rat x = Rat{1, 2} + 1;
          ~~~~~ ^ ~
```

```
note: candidate function not viable:
```

```
no known conversion from 'int'
```

```
Rat operator+ (Rat b) const;
```

Conversion vers un autre type

Compléments

On peut aussi déclarer des **conversions vers d'autres types** !

La syntaxe est

```
operator nomDuType() const;
```

Une telle conversion peut être **implicite ou explicite**.

Déclaration d'une conversion explicite vers int :

rational-class.cpp

```
66 explicit operator int() const;
```

Définition :

rational-class.cpp

```
198 Rat::operator int() const {  
199     if (denom != 1)  
200         throw invalid_argument("rationnel non entier");  
201     return numer;  
202 }
```

Gestion des ressources

La mécanique des constructeurs est particulièrement utile quand on doit **gérer des ressources**. Par exemple, si l'on veut programmer la classe des vecteurs, il faut prévoir l'allocation (réservation de la mémoire). Il faut

- un **constructeur de copie** qui s'occupe de **réserver la place mémoire et de faire la copie** ; il est utilisé pour faire la copie, par exemple lors d'un **passage par valeur à une fonction**.
- **surcharger l'opérateur d'affectation** `operator=` pour faire la même chose.
- un **destructeur** qui s'occupe de **libérer la mémoire** quand on a plus besoin de l'objet.

Plan

- 1 Qu'est-ce que la programmation par objets ?
- 2 Notion de classe
- 3 Les constructeurs
- 4 Bilan**

Rappel : Pourquoi la programmation par objets ?

Le problème que l'on essaye de résoudre :

Problème

On veut

- savoir **quelles sont les opérations que l'on peut appliquer à une variable donnée.**
- vérifier sans **inspecter tout le code** qu'une donnée respectera toujours une **contrainte d'intégrité** (on dit **invariant**).

Bilan de ce que l'on sait sur la programmation objet

Résumé

On a vu que l'on peut

- regrouper les variables dans une **structure**
- ajouter des **méthodes** à la structure
- ajouter des constructeurs à la structure

Si l'utilisateur se contente d'appeler **les méthodes et les constructeurs que l'on a écrit** (**interface**), alors on contrôle tout le code qui accède à la structure...

Contournement de l'interface

Problème

Si l'utilisateur accède directement aux attributs d'une classe, il peut casser un invariant.

```
Rat r {1};  
r.denom = 0;
```

La suite!!!

On va enfermer les données de la structure dans une boîte que l'utilisateur ne peut pas ouvrir. C'est **l'encapsulation** !

Le C++ fournit des moyens de forcer l'utilisateur du code à respecter l'interface que l'on a décidée...