INFO-F105 – Langages de programmation 1
ADTs

1 Introduction

Un *Abstract Data Type* (ADT) est un type théorique qui définit des opérations sur ses données. Il s'agit d'un type conceptuel qui n'est donc **pas encore implémenté**.

Un exemple d'ADT que vous connaissez est le *stack* (cf. Pointeurs & Tableaux).

> Un stack contient une collection de données et définit deux opérations :
>
> - *Push* : ajoute une donnée sur le stack
> - *Pop* : retire la donnée la plus récemment ajoutée sur le stack

Il s'agit là de la définition conceptuelle de l'ADT. Comme on peut le voir, il n'y a rien qui indique la façon dont les données sont organisées ou les opérations sont implémentées.

2 Classes

Une *classe* est une façon d'implémenter un ADT en C++.

```
1  class Date {
2   private:
3    unsigned _day;
4    unsigned _month;
5    unsigned _year;
6   public:
7    Date(unsigned day, unsigned month, unsigned year);
8    unsigned getDay() const;
9    unsigned getMonth() const;
10   unsigned getYear() const;
11   void setDay(unsigned day);
12   void setMonth(unsigned month);
13   void setYear(unsigned year);
14 };
```

La définition d'une classe contient les déclarations de ses *membres* :

- *Attributs* : des variables internes (lignes 3-5)
- *Méthodes* : des fonctions internes (lignes 7-13)

Question : quel autre type aurait-on pu utiliser pour l'attribut `_month` ?

Instanciation

La méthode à la ligne 7 est un *constructeur* et permet de créer une *instance* de la classe.

```
Date date = Date(1, 2, 2025);     // Appel au constructeur
unsigned year = date.getYear();   // year = 2025
```

Les méthodes publiques peuvent être appelées sur l'instance `date`, qu'on appelle un *objet*.

L'opérateur `->` permet d'accéder aux membres de l'objet derrière un pointeur.

```
Date* ptr = new Date(1, 2, 2025);
unsigned year = ptr->getYear();
```

De ce point de vue, une `class` est assez proche d'une `struct`.

Getter & Setter

Un *getter* (lignes 8-10) est une méthode qui renvoie une valeur "stockée" par l'objet. Dans cet exemple, les *getters* sont `const` car ils ne permettent pas de modifier le contenu de l'objet.

Un *setter* (lignes 11-13) est une méthode qui modifie une valeur "stockée" par l'objet.

Note : un *getter* n'est pas forcément `const` lorsqu'il retourne un pointeur ou une référence.

```
class Date {
 public:
  unsigned& getDayRef();
  const unsigned& getDayRefConst() const;
  unsigned* getDayPtr();
  const unsigned* getDayPtrConst() const;
};

Date date = Date(1, 2, 2025);

date.getRefDay() = 5;     // date.getDay() == 5
*date.getPtrDay() = 10;   // date.getDay() == 10
```

Rappel : une référence permet d'accéder à la donnée référencée plutôt qu'à une copie.

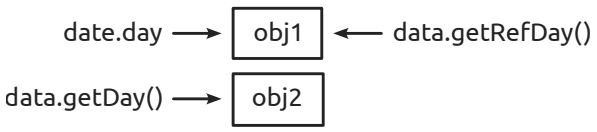

Figure 1: Référence vs copie

Encapsulation

Les lignes 2 et 6 délimitent des portions de la classe qui peuvent être :

- `private` : les membres ne sont pas accessibles depuis l'extérieur
- `public` : les membres sont accessibles depuis l'extérieur

L'*encapsulation* est le fait de limiter la partie publique aux opérations définies par l'ADT. Tout ce qui dépend de l'implémentation reste alors privé, en particulier les attributs.

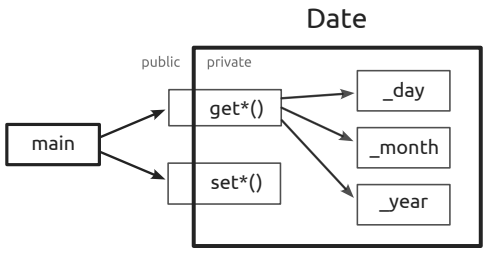

Figure 2: Encapsulation de l'implémentation

L'encapsulation est l'un des éléments principaux de la *programmation orientée objet*, un paradigme de programmation qui sera développé en détail en INFO-F202.

Implémentation

```
1  #include "date.hpp"
2  Date::Date(unsigned day, unsigned month, unsigned year)
3    : _day(day), _month(month), _year(year) { /* Rien */ }
4  unsigned Date::getMonth() {
5    return this->_month;
6  }
7  void Date::setMonth(unsigned month) {
8    if (month < 12) {
9      this->_month = month;
10   }
11 }
```

Les méthodes sont préfixées par `Date::` pour indiquer dans quel namespace elles se trouvent.

Les lignes 2 et 3 implémentent le constructeur. Dans cet exemple, le constructeur ne contient rien dans son corps car tous les attributs sont initialisés dans une *liste d'initialisation*.

La variable `this` aux lignes 5 et 9 est un pointeur vers l'objet sur lequel la méthode a été appelée. Il est facultatif ici car `_month` n'entre en conflit avec aucune autre variable. C'est pour éviter de tels conflits qu'on préfixe souvent les noms d'attributs par un '`_`'.

Note : il aurait probablement été judicieux de définir les *getters* et *setters* comme `inline`.

2.1 Constructeurs

Il est possible de définir plusieurs fonctions du même nom mais avec différents paramètres. On appelle cela une *surcharge* de la fonction (*overload*).

Les constructeurs ne font pas exception. On en distingue plusieurs catégories :

```
Date();                     // Par défaut
Date(const Date& other);    // Copie
Date(Date&& other);         // Transfert
Date(unsigned timestamp);   // Conversion
Date(unsigned day, unsigned month, unsigned year);  // Paramétrique
```

Le constructeur de copie est utilisé lorsque l'objet est copié depuis un objet source, et celui de transfert lorsque l'objet source est une valeur temporaire comme un retour de fonction.

```
Date date1 = Date(1, 2, 2025);
Date date2 = date1;  // Utilisation du constructeur de copie
Date date3 = foo();  // Utilisation du constructeur de transfert
```

Les constructeurs de conversion permettent des conversions implicites vers la classe.

```
Date date4 = 1738400000000;  // date4 = Date(1, 2, 2025)
```

Note : le *timestamp* est une valeur numérique qui s'incrémente chaque fraction de seconde.

Implémentation par défaut

Les trois premiers constructeurs sont définis automatiquement par le compilateur. Il n'est donc pas nécessaire de le faire soi-même si le comportement par défaut convient à la classe.

Par exemple, le constructeur de copie par défaut copie tous les attributs de l'objet source vers l'objet cible en utilisant leurs propres constructeurs de copie.

Il est toutefois possible de supprimer complètement ces constructeurs en utilisant `delete`.

```
class Date {
 public:
  Date(const Date& other) = delete;
};
```

Destructeur

Le *destructeur* d'une classe est une méthode appelée automatiquement lorsque l'objet est supprimé, notamment à la fin de sa portée.

```
class Date {
 public:
  ~Date();
};
```

Le destructeur peut être utile pour `delete` des pointeurs lorsqu'ils deviennent hors de portée.

2.2 Opérateurs

Il est possible de surcharger la plupart des opérateurs[1] pour définir leur effet sur notre classe.

```
class Date {
 public:
  void operator=(const Date&);  // Copie
  void operator=(Date&&);       // Transfert
  operator unsigned() const;    // Conversion
  Date operator+(const Date& other) const;  // Opérateur binaire
  Date& operator++();                       // Opérateur unaire
  Date& operator+=(const Date& other);
  bool operator<=(const Date& other) const;
};
```

Les opérateurs de conversion permettent des conversions implicites depuis la classe.

```
assert(Date(1, 2, 2025) == 1738400000000);
```

Cela peut être utile de passer par un opérateur lorsqu'il est aussi intuitif qu'une fonction.

```
assert(Date(1, 2, 2025) <= Date(30, 6, 2025));
```

Dans cet exemple, on comprend que `<=` indique si la première date précède la première.

Opérateur d'index

Il peut parfois être utile d'accéder aux éléments d'un objet comme s'il s'agissait d'un tableau.

```
class Array {
 public:
  inline int operator[](size_t index);
};
```

Opérateur d'appel de fonction

Dans certains cas, cela peut faire sens d'"appeler" l'objet comme s'il s'agissait d'une fonction.

```
class Comparator {
 public:
  inline bool operator()(int a, int b) {
    return a == b;
  }
};

Comparator comp = Comparator();
bool isEqual = comp(5, 5);  // true
```

Il est difficile d'expliquer l'intérêt d'un tel code sans parler d'héritage ou de polymorphisme, des concepts essentiels de la programmation orientée objet qui seront abordés en INFO-F202.

Stream insertion

Il existe des situations (très rares !) où l'on souhaiterait partiellement briser l'encapsulation et permettre à une fonction extérieure spécifique d'accéder aux membres privés de la classe.

```
1  class Date {
2   public:
3    friend std::ostream& operator<<(std::ostream&, const Date&);
4  };

5  std::ostream& operator<<(std::ostream& stream, const Date& date) {
6    stream << date._day << " / " << date._month << " / " << date._year;
7    return stream;
8  }

9  std::cout << Date(1, 2, 2025) << std::endl;
```

La fonction `friend` à la ligne 5 permet de "*print*" un objet `Date` en accédant directement à ses attributs privés. Cette fonction devrait être définie dans le même fichier que `Date`.

2.3 Membres statiques

Tous les membres que nous avons vus jusqu'à maintenant sont liés à une instance de la classe.

Il est possible de définir des attributs et méthodes *statiques*, qui sont liés à la classe elle-même.

```
1  class Date {
2   private:
3    static constexpr unsigned N_MONTHS = 12;

4   public:
5    static Date now();
6  };
```

Ces membres sont accessibles en ajoutant le préfixe `Date::`

```
7  void Date::setMonth(unsigned month) {
8    if (month < Date::N_MONTHS) {
9      _month = month;
10   }
11 }

12 void foo() {
13   Date date = Date::now();
14 }
```

Le préfixe à la ligne 8 n'est pas nécessaire car la fonction se trouve déjà dans `Date::`

Dans cet exemple, la méthode statique `now` fabrique et retourne un objet `Date`. Il s'agit d'un *design pattern* [1] appelé *Factory* que vous reverrez en INFO-F307.

[1] Un *design pattern* est une façon réutilisable d'organiser son code[2]

3 Classes en Python

Les classes existent également en Python.

```
1  from datetime import date

2  class Date:
3    N_MONTHS = 12

1    def __init__(self, day, month, year):
2      self.__day = day
3      self.__month = month
4      self.__year = year

8    @staticmethod
9    def now():
10     today = date.today()
11     return Date(today.day, today.month, today.year)

12   def getMonth(self):
13     return self.__month

14   def setMonth(self, month):
15     if month < Date.N_MONTHS:
16       self.__month = month

17   def __le__(self, other):
18     # TODO return if self <= other
```

On remarque certaines différences syntaxiques avec C++ :

- Le pointeur `this` est remplacé par le paramètre `self`, déclaré dans chaque méthode.
- Les noms des membres privés commencent obligatoirement par `__`.
- La variable `self` doit être précisée explicitement pour accéder à un membre.
- Les surcharges d'opérateurs s'écrivent avec une syntaxe particulière `__method__`.
- Les attributs d'instance sont déclarés à l'intérieur même du constructeur, contrairement aux variables statiques qui sont déclarées en dehors de toute méthode.
- Les méthodes statiques sont précédées du *décorateur* [2] `@staticmethod`.

La fonction `today()` utilisée à la ligne 10 est une méthode de classe, une sorte de méthode statique avec certaines subtilités qui nécessitent de comprendre l'orienté objet.

Références

[1] "Operator overloading – cppreference.com." Available: https://en.cppreference.com/w/cpp/language/operators

[2] "Design patterns." Available: https://refactoring.guru/design-patterns

[2] Un *décorateur* permet d'étendre le comportement d'une fonction.