Rappels de C++

Programmation générique

Vincent Lemaire
vincent.lemaire@upmc.fr

Surcharge d'opérateurs

- Par fonction membre
- Par fonction amie
- Functors (Function objects)
- Exemple

Programmation générique

- Fonctions génériques
- Classes génériques

STL : Standard Template Library

- Conteneurs
- Itérateurs
- Algorithmes

Surcharge d'opérateurs

Permet de redéfinir les opérateurs usuels pour une nouvelle `classe`.

- opérateurs unaires : `++`, `--`
- opérateurs d'affectation : `=`, `+=`, `-=`, `*=`, `/=`, `%=`
- opérateurs arithmétiques (binaires) : `+`, `-`, `*`, `/`, `%`
- opérateurs de comparaison : `==`, `!=`, `<`, `>`, `<=`, `>=`
- opérateurs « informatiques » : `()`, `[]`, `*`, `&`, `->`, **`new[]`**, **`delete[]`**
- opérateurs de flux : `<<`, `>>`
- opérateurs de conversion : **`double`**, **`int`**, **`char`**, `...`

En général, la surcharge des opérateurs unaires, d'affectation, de conversion et « informatiques », se fait par des fonctions membres et la surcharge des opérateurs binaires (arithmétiques et comparaisons) se fait par des fonctions amies.

Obligatoirement, la surchage des opérateurs de flux se fait par des fonctions amies.

Surcharge par fonction membre

Surcharge par une fonction membre de l'opérateur $\bullet$: la fonction doit se nommer **`operator`**$\bullet$ et

$$\texttt{obj}\bullet \text{ et } \bullet\texttt{obj} \Longleftrightarrow \texttt{obj.}\mathbf{\texttt{operator}}\bullet(\ldots)$$

$$\texttt{obj1}\bullet\texttt{obj2} \Longleftrightarrow \texttt{obj1.}\mathbf{\texttt{operator}}\bullet(\text{obj2})$$

Surcharge par fonction membre

Surcharge par une fonction membre de l'opérateur $\bullet$: la fonction doit se nommer **`operator`**$\bullet$ et

$$\texttt{obj}\bullet \text{ et } \bullet\texttt{obj} \iff \texttt{obj.}\textbf{\texttt{operator}}\bullet(\ldots)$$

$$\texttt{obj1}\bullet\texttt{obj2} \iff \texttt{obj1.}\textbf{\texttt{operator}}\bullet(\text{obj2})$$

Syntaxe pour l'opérateur d'affectation `[]` (unaire)

- Lecture : `elt` **`operator[](int) const`** ;
- Ecriture : `elt&` **`operator[](int)`** ;

Surcharge par fonction membre

Surcharge par une fonction membre de l'opérateur $\bullet$: la fonction doit se nommer **`operator`**$\bullet$ et

$$\texttt{obj}\bullet \text{ et } \bullet\texttt{obj} \Longleftrightarrow \texttt{obj.}\mathbf{\texttt{operator}}\bullet(\ldots)$$

$$\texttt{obj1}\bullet\texttt{obj2} \Longleftrightarrow \texttt{obj1.}\mathbf{\texttt{operator}}\bullet(\text{obj2})$$

Syntaxe pour l'opérateur d'affectation `[]` (unaire)

- Lecture : `elt` **`operator[](int) const`**;
- Ecriture : `elt&` **`operator[](int)`**;

Syntaxe pour opérateurs `++` et `--` qui peuvent être préfixe (`++n`) ou suffixe (`n++`) : on utilise 2 fonctions différentes :

- Opérateur préfixe : `obj&` **`operator++();`**
- Opérateur suffixe : `obj` **`operator++(int);`**

Surcharge par fonction amie

Surcharge par une fonction amie de l'opérateur $\bullet$: la fonction doit se nommer **`operator`**$\bullet$ et

$$\texttt{obj}\bullet \text{ et } \bullet\texttt{obj} \iff \textbf{\texttt{operator}}\bullet(\text{obj})$$

$$\texttt{obj1}\bullet\texttt{obj2} \iff \textbf{\texttt{operator}}\bullet(\text{obj1, obj2})$$

Surcharge par fonction amie

Surcharge par une fonction amie de l'opérateur $\bullet$: la fonction doit se nommer **`operator`**$\bullet$ et

$$\texttt{obj}\bullet \text{ et } \bullet\texttt{obj} \iff \textbf{\texttt{operator}}\bullet(\text{obj})$$

$$\texttt{obj1}\bullet\texttt{obj2} \iff \textbf{\texttt{operator}}\bullet(\text{obj1}, \text{obj2})$$

Syntaxe pour les opérateurs de flux << et >>

- Injection :

```
std::ostream& operator<<(std::ostream &o, const Obj &x)
```

- Extraction :

```
std::istream& operator>>(std::istream &i, Obj &x)
```

Functor : objet fonctionnel

On appelera Functor une classe (ou par extension objet) dont l'opérateur fonctionnel `()` est surchargé.

Dans la suite de ce cours, on utilisera une **`struct`** pour définir les functors.

Exemple :

```
struct sommeur {
    double operator()(double x) { return state += x; };
    double etat() const { return state; };
    double& etat() { return state; };
    private:
        double state;
};
```

Utilisation (pour l'instant, il est dur de voir l'intérêt...)

```
sommeur S;
for (k = 1; k <= 10; k++) {
    S(k * 0.1);
};
cout << S.etat() << endl;
```

Exemple : retour sur p_adic

Surcharge de quelques opérateurs pour la classe p_adic :

```
class p_adic {
  public:
      p_adic(unsigned p = 2, unsigned n = 0);
      p_adic(unsigned p, unsigned r, unsigned a[]);
      ~p_adic() { delete [] coeff; }

      // surcharge par fonction membre
      p_adic& operator=(p_adic const &x);
      p_adic& operator++();           // version prefixee
      p_adic& operator++(int);        // version postfixee

      // surcharge par fonction amie
      friend bool operator<(p_adic const &a, p_adic const &b);
      friend bool operator==(p_adic const &a, p_adic const &b);
      friend p_adic operator+(p_adic const &a, p_adic const &b);
      friend std::ostream& operator<<(std::ostream &o, p_adic const &x);
  private:
      unsigned n;
      unsigned p, r, * coeff;
      void increment();
};
```

Définition des fonctions

```
// definition des fonctions membres
p_adic& p_adic::operator=(p_adic const &x) {
    if (&p != this) {
        n = x.n; p = x.p; r = r.p;
        delete [] coeff;
        coeff = new unsigned[r];
        for (int k = 0; k < r; ++k) coeff[k] = x.coeff[k];
    }
    return *this;
}
p_adic& p_adic::operator++() {
    this->increment();
    return *this;
}
p_adic& p_adic::operator++(int) {
    p_adic sauv(*this); // ou p_adic sauv = *this;
    this->increment();
    return sauv;
}

// definition des fonctions amies
bool operator<(p_adic const &a, p_adic const &b) {
    return (a.n < b.n);
}
```

Exemple d'utilisation

Quels sont les constructeurs et opérateurs utilisés dans l'exemple ci-dessous ?

```
#include <iostream>
#include "p_adic.hpp"

p_adic max(p_adic tab[], int size) {
    p_adic result = tab[0];
    for (int k = 1; k < size; ++k)
        if (result < tab[k]) result = tab[k];
    return result;
}

int main() {
    srandom(time(NULL));
    int n = 10;
    p_adic tableau[n];
    for (int k = 0; k < n; ++k)
        tableau[k] = p_adic(3, random());
    std::cout << max(tableau, n) << std::endl;
    return 0;
}
```

Exemple d'utilisation

Quels sont les constructeurs et opérateurs utilisés dans l'exemple ci-dessous ?

```
#include <iostream>
#include "p_adic.hpp"

p_adic max(p_adic tab[], int size) {
    p_adic result = tab[0];
    for (int k = 1; k < size; ++k)
        if (result < tab[k]) result = tab[k];
    return result;
}

int main() {
    srandom(time(NULL));
    int n = 10;
    p_adic tableau[n];
    for (int k = 0; k < n; ++k)
        tableau[k] = p_adic(3, random());
    std::cout << max(tableau, n) << std::endl;
    return 0;
}
```

Que pensez-vous de la fonction `max` ?

Fonction générique

On pourrait définir la fonction `max` exactement de la même façon pour un tableau de **`double`** ou d'**`int`** :

```
p_adic max(p_adic tab[], int size) {
    p_adic result = tab[0];
    for (int k = 1; k < size; ++k)
        if (result < tab[k]) result = tab[k];
    return result;
}

double max(double tab[], int size) {
    double result = tab[0];
    for (int k = 1; k < size; ++k)
        if (result < tab[k]) result = tab[k];
    return result;
}

int max(int tab[], int size) {
    int result = tab[0];
    for (int k = 1; k < size; ++k)
        if (result < tab[k]) result = tab[k];
    return result;
}
```

Fonction générique -2-

En C++, il est possible de définir un modèle de fonction (ou patron ou template) qui s'adapte à la compilation en fonction des besoins.

On définit un modèle pour la fonction `max` en utilisant la syntaxe suivante

```
template <typename MonType>
MonType max(MonType tab[], int size) {
    MonType result = tab[0];
    for (int k = 1; k < size; ++k)
        if (result < tab[k]) result = tab[k];
    return result;
}
```

- le mot-clé **`template`** est suivi d'une liste composé
  - de types génériques précédés du mot-clé **`typename`** (ou **`class`**)
  - de constantes entières précédées du mot-clé **`int`**
- précède la définition d'une fonction ou d'une classe qui sera qualifiée de générique

```
template <typename Type1, typename Type2, int N>
// fonction ou classe dans laquelle Type1 et Type2 sont des types
// et N est une constante entière
```

Instanciation

Le compilateur crée des instances du modèle de la fonction `max` en fonction des appels.

La fonction `max<p_adic>` est une instance de la fonction `max` avec le type `p_adic`.

```
p_adic tableau[n];
for (int k = 0; k < n; ++k)
    tableau[k] = p_adic(3, random());
std::cout << max<p_adic>(tableau, n) << std::endl;
double tab[n];
for (int k = 0; k < n; ++k)
    tab[k] = random() / (1 + RAND_MAX);
std::cout << max<double>(tab, n) << std::endl;
```

Instanciation

Le compilateur crée des instances du modèle de la fonction `max` en fonction des appels.

La fonction `max<p_adic>` est une instance de la fonction `max` avec le type `p_adic`.

```
p_adic tableau[n];
for (int k = 0; k < n; ++k)
    tableau[k] = p_adic(3, random());
std::cout << max<p_adic>(tableau, n) << std::endl;
double tab[n];
for (int k = 0; k < n; ++k)
    tab[k] = random() / (1 + RAND_MAX);
std::cout << max<double>(tab, n) << std::endl;
```

Dans ces 2 appels l'instance de la fonction `max` peut être déterminée à partir des arguments `tableau` et `tab`.
L'écriture implicite suivante est autorisée :

```
std::cout << max(tableau, n) << std::endl;
std::cout << max(tab, n) << std::endl;
```

Classes génériques

La même syntaxe permet de définir une classe générique.

```
template <typename T>
class SmartTab {
  public:
      SmartTab(int n) : size(n), data(new T[n]) {}
      ~SmartTab() { delete [] data; }

      // constructeur de copie
      // operateur d'affectation =

      T & operator[](int i) { if (check(i)) return data[i]; }
      T operator[](int i) const { if (check(i)) return data[i]; }
  private:
      T *data;
      int size;
      bool check(int i) const { return ((i >= 0) || (i < n)); }
};
```

Une instance (ou une version) de cette classe générique est `SmartTab<T>` où `T` est un type comme **double**, `p_adic`, ...

Pas d'écriture implicite.

Classes génériques -2-

La définition des fonctions membres d'une classe générique doit se faire dans le même fichier que la classe (un header généralement).
La ligne **template** doit être précisée pour chaque fonction.

```
// constructeur de copie
template <typename T>
SmartTab(SmartTab<T> const & tab) {
    size = tab.size;
    data = new T[size];
    for (int k = 0; k < size; ++k)
        data[k] = tab[k];
};

// operateur d'affectation =
template <typename T>
SmartTab<T> & operator=(SmartTab<T> const & tab) {
    if (tab != this) {
        size = tab.size;
        delete [] data;
        data = new T[size];
        for (int k = 0; k < size; ++k) data[k] = tab.data[k];
    }
    return this;
};
```

Classes génériques -2-

La définition des fonctions membres d'une classe générique doit se faire dans le même fichier que la classe (un header généralement).
La ligne **template** doit être précisée pour chaque fonction.

```
// constructeur de copie
template <typename T>
SmartTab<T>::SmartTab(SmartTab<T> const & tab) {
    size = tab.size;
    data = new T[size];
    for (int k = 0; k < size; ++k)
        data[k] = tab[k]; // ou data[k] = tab.data[k];
};

// operateur d'affectation =
template <typename T>
SmartTab<T> & SmartTab<T>::operator=(SmartTab<T> const & tab) {
    if (&tab != this) {
        size = tab.size;
        delete [] data;
        data = new T[size];
        for (int k = 0; k < size; ++k) data[k] = tab.data[k];
    }
    return *this;
};
```

`typedef` et `typename`

- **`typedef`** : existe aussi en C, permet de définir un nouveau type à partir d'un autre type, en gros : copier/coller à la compilation.
  Permet d'écrire un code plus court, plus lisible ! **A utiliser !**
  Syntaxe : **`typedef`** *ancientypelongetcomplique* *NouveauType*
- **`typename`** : mot-clé indiquant que le « terme » suivant est un type.
  Deux usages :
  - Lorsqu'il y a un litige possible sur le « terme » qui suit.
    Exemple : dans la classe abstraite `processus`.
  - Dans l'argument d'un **`template`** pour définir le ou les noms des types génériques (peut-être remplacé par le mot-clé **`class`**).
    Exemple : `template <typename T1, typename T2 = T1>`

`typedef` et `typename`

- **`typedef`** : existe aussi en C, permet de définir un nouveau type à partir d'un autre type, en gros : copier/coller à la compilation.
  Permet d'écrire un code plus court, plus lisible ! **A utiliser !**
  Syntaxe : **`typedef`** *ancientypelongetcomplique* *NouveauType*
- **`typename`** : mot-clé indiquant que le « terme » suivant est un type.
  Deux usages :
  - Lorsqu'il y a un litige possible sur le « terme » qui suit.
    Exemple : dans la classe abstraite `processus`.
  - Dans l'argument d'un **`template`** pour définir le ou les noms des types génériques (peut-être remplacé par le mot-clé **`class`**).
    Exemple : **`template`** `<`**`typename`** `T1,` **`typename`** `T2 = T1>`

```
template <typename T>
class Test {
  public:
      typedef SmartTab<T> STab;  // définition du type STab
      ...
  private:
      STab data;
}
```

```
typename Test<T>::STab;          // utilisation du type STab
```

STL : Standard Template Library

Librairie standardisée, due à Alexander Stepanov, utilisant les templates qui repose sur 3 concepts :

- les conteneurs : peuvent être utilisés avec tout type de données
- les itérateurs : une abstraction des pointeurs qui permettent de parcourir les conteneurs
- les algorithmes : agissent sur les conteneurs par le biais des itérateurs et d'objets fonctionnels (functors)

Il existe de nombreuses versions de cette librairie dont des versions parallélisées.

Conteneurs

Conteneurs séquentiels

- `vector` : tableau évolué, opérateur `[]` d'accès direct, ajout/suppression d'éléments, changement de taille
- `list` : liste doublement chaînée
- `deque` : un `vector` plus spécialisé pour l'ajout en début et en fin

Il existe des adptateurs qui changent le comportement de ces conteneurs :

- `stack` : piles LIFO
- `queue` : piles FIFO
- `priority_queue` : file d'attente

Conteneurs associatifs (non utilisés dans ce cours)

- `map, multimap` : correspondance clé/valeur (clé de recherche => valeur stockée)
- `set, multiset` : représente un ensemble

Itérateurs

Une abstraction des pointeurs qui permet de se déplacer dans tout type de conteneurs.

- Bidirectional iterator : l'itérateur de `list` qui permet une lecture/écriture et un déplacement d'un pas en avant ou en arrière
  - **`operator`**$*$ déréférencement surchargé
  - **`operator`**++ et **`operator`**--
- Random access iterator : l'itérateur de `vector`, `deque` qui permet une lecture/écriture et un déplacement libre
  - **`operator`**$*$ déréférencement surchargé
  - **`operator`**++ et **`operator`**--
  - **`operator`**+ et **`operator`**- (opération avec un entier)

Il existe aussi des itérateurs de lecture uniquement ou d'écriture uniquement...

Algorithmes

cf. fiche