Programmation Modulaire
Surcharge de fonctions et d'opérateurs

Florent Hivert

Mél : Florent.Hivert@lri.fr
Adresse universelle : http://www.lri.fr/~hivert

1 Préliminaire : les manipulateurs d'entrée sortie

2 Fonction : déclaration et définition

3 Pourquoi la surcharge ?

4 Surcharge de fonctions

5 Surcharge d'opérateurs

Plan

1 Préliminaire : les manipulateurs d'entrée sortie

2 Fonction : déclaration et définition

3 Pourquoi la surcharge ?

4 Surcharge de fonctions

5 Surcharge d'opérateurs

Configurer ses affichages

Retenir

La bibliothèque `iomanip` *permet de manipuler les affichages :*

- `setw(n)` *la largeur (width) sur n caractères*
- `setfill(c)` *complète avec c si pas assez large (ex :* `003`*)*
- `left` *aligné à gauche*
- `right` *aligné à droite*
- `scientific` *notation scientifique (ex :* `3.1415926e+00`*)*
- `fixed` *notation à virgule fixe (ex :* `40.3356`*)*
- `setprecision` *nombre de chiffre (ex :* `40.33`*)*
- *et beaucoup d'autres (*`hex`, `boolalpha`, `showpos`*...)*

`http://www.cplusplus.com/reference/library/manipulators/`

Configurer ses affichages : exemple

iomanip.cpp

```
1  #include <iostream>
2  #include <iomanip>
3  using namespace std;
4
5  int main() {
6    cout << setw(4) << setfill('*') << 42 << endl;  // **42
7    float pi = 3.1415926;
8    cout << fixed;
9    cout << setprecision(2) << 4*pi << endl;  // 12.57
10   cout << setprecision(6) << 4*pi << endl;  // 12.566370
11   cout << scientific << 4*pi << endl;       // 1.256637e+01
12 }
```

Configuration de l'affichage

Attention

On ne change pas la manière dont la variable est codée en mémoire, mais seulement la façon dont elle est affichée.

affbool.cpp

```
1    bool b = true;
2    cout << b << " " << int(b) << endl;  // affiche 1 1
3    cout << boolalpha;
4    cout << b << " " << int(b) << endl;  // affiche true 1
```

Plan

1 Préliminaire : les manipulateurs d'entrée sortie

2 Fonction : déclaration et définition

3 Pourquoi la surcharge ?

4 Surcharge de fonctions

5 Surcharge d'opérateurs

Fonction : déclaration et définition

Retenir

*Une **déclaration** informe le compilateur (et aussi le lecteur) que l'on va utiliser une certaine fonction avec le détail des types des paramètres et du résultat.*

Bon endroit pour écrire la documentation de la fonction.

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

Fonction : déclaration

Retenir

*On **déclare une fonction** en écrivant son **entête** terminé par « ;».*

```
int somme(int, int);
```

- *Pour pouvoir utiliser une fonction, il faut qu'elle ait été **déclarée avant**.*
- *On peut déclarer plusieurs fois la même fonction.*
- *On peut écrire les noms des paramètres :*

```
int somme(int a, int b);
```

Fonction : définition

Retenir

*On **définit une fonction** en écrivant son **code complet**.*

```
int somme(int a, int b) {
  return a + b;
}
```

- *Si la fonction n'a pas déjà été déclarée, la **définition sert aussi de déclaration**.*
- *Il est **interdit de re-définir** une fonction. Pour qu'un programme utilisant une fonction puisse être exécuté, il doit y avoir **exactement une définition** quelque part dans le code.*

Déclaration et définition : exemple complet

somme.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  // Déclaration
5  int somme(int, int);
6
7  int main() {
8    int x, y;
9    cin >> x >> y;
10   cout << somme (x, y) << endl;
11 }
12
13 // Définition
14 int somme(int a, int b) {
15   return a + b;
16 }
```

Intérêt d'une déclaration

- Plus lisible : on rassemble toutes les déclarations au début du fichier, voire dans un fichier à part (`.h` ou `.hpp`)
- Ordre plus logique : on peut rassembler les fonctions reliées logiquement, sans être obligé de devoir mettre en premier les fonctions qui sont appelées
- Fonctions mutuellement récursives (f appelle g et g appelle f)

Plan

1 Préliminaire : les manipulateurs d'entrée sortie

2 Fonction : déclaration et définition

3 Pourquoi la surcharge ?

4 Surcharge de fonctions

5 Surcharge d'opérateurs

Qu'est-ce que la surcharge

Retenir

*C'est le fait d'avoir **plusieurs fonctions ou opérateurs avec le même nom** mais qui s'appliquent à des types de paramètres différents.*

to-string.cpp

```
1    int i = 2;
2    float f = 3.14159;
3    string si = to_string(i), sf = to_string(f);
4    // to_string est appelé un int et sur un float
5    cout << si << " " << sf << endl;
```

Pas pratique !

En C, la surcharge n'est pas possible... Voici quelques conséquences :

Exemple

Calcul de la valeur absolue :

```
int abs(int j);
long int labs(long int j);
long long int llabs(long long int j);
double fabs(double x);
float fabsf(float x);
long double fabsl(long double x);
```

6 fonctions différentes avec des conventions de noms incompatibles !

Pratique !

Retenir

En C++, on écrit `abs` *et le compilateur choisit la bonne version en fonction du type du paramètre (utilise* `cmath`*).*

abs.cpp

```
1  #include <cmath>
2  #include <iostream>
3  #include <iomanip>
4  using namespace std;
5
6  int main() {
7    // J'utilise la notation scientifique
8    // pour faire la différence entre les floats et les int.
9    std::cout << scientific;
10   std::cout << abs(-3) << endl;   // affiche 3
11   std::cout << abs(-3.) << endl;  // affiche 3.000000e+00
12 }
```

Pas pratique !

En C, la surcharge n'est pas possible... Voici quelques conséquences :

Exemple

Fonctions similaires :

```
// Cherche x dans tous le tableau
int chercheTableau(vector<int> v, int x);
// Cherche x entre les positions min et max
int chercheTableauMinMax(vector<int> v, int x,
                         int min, int max);
```

On voudrait par exemple pouvoir écrire

```
pos2 = cherche(v, 2);
pos2debut = cherche(v, 2, 0, v.size()/2);
```

Pas pratique !

Exemple (Nouveau type de nombres :)

Si l'on ne sait faire que des fonctions :

```
Complex add(Complex x, Complex y);
Complex mul(Complex x, Complex y);
Complex f(Complex a, Complex b, Complex c) {
  return add(add(mul(a,b), mul(b,c)), mul(c,a));
}
```

Il est beaucoup plus lisible d'écrire :

```
Complex operator+(Complex x, Complex y);
Complex operator*(Complex x, Complex y);
Complex f(Complex a, Complex b, Complex c) {
  return a*b + b*c + c*a;
}
```

Bilan : surcharge

Retenir

La surcharge permet d'écrire de manière plus simple et plus lisible.

Attention

Si l'on surcharge une fonction, il faut s'assurer que la sémantique des différentes surcharges est compatible. Sinon le programme risque de devenir incompréhensible.

Exemple de ce qu'il ne faut **pas faire** :

```
int   add(int a,   int b)   { return a + b; }
float add(float a, float b) { return a - b; }
```

Bilan : surcharge

Retenir

La surcharge permet d'écrire de manière plus simple et plus lisible.

Attention

Si l'on surcharge une fonction, il faut s'assurer que la sémantique des différentes surcharges est compatible. Sinon le programme risque de devenir incompréhensible.

Exemple de ce qu'il ne faut **pas faire** :

```
int   add(int a,   int b)   { return a + b; }
float add(float a, float b) { return a - b; }
```

Plan

1 Préliminaire : les manipulateurs d'entrée sortie

2 Fonction : déclaration et définition

3 Pourquoi la surcharge ?

4 Surcharge de fonctions

5 Surcharge d'opérateurs

Comment surcharger une fonction

Retenir

*Pour surcharger une fonction, il suffit de déclarer/définir une fonction avec le **même nom**, mais des **paramètres de types différents**.*

trans.cpp

```
1  string transmogrify(string s) {
2    return "(-:" + s + ":-)";
3  }
4  string transmogrify(int n) {
5    return ":" + string(n, '-') + ")";
6  }
7
8  int main() {
9    cout << transmogrify("toto") << endl;  // affiche (-:toto:-)
10   cout << transmogrify(3) << endl;       // affiche :---)
11 }
```

Type de retour

Retenir

*Il n'y a **pas de contrainte sur les types de retour** de deux surcharges. Ils peuvent être identiques ou différents.*

double.cpp

```
1  string duplique(string s) {
2    return s + s;
3  }
4  int duplique(int n) {
5    return 2 * n;
6  }
7
8  int main() {
9    cout << duplique("toto") << endl;  // affiche tototototo
10   cout << duplique(3) << endl;       // affiche 6
11 }
```

Choix de la fonction

Retenir

Le compilateur choisit la fonction surchargée ***dont les types des paramètres correspondent*** *aux types des valeurs transmises.*

choix.cpp

```
1  void foo() { cout << "rien" << endl; }
2  void foo(int i) { cout << "un entier " << i << endl; }
3  void foo(float f) { cout << "un réel " << f << endl; }
4  void foo(int i, int j) {
5    cout << "deux entiers " << i << " " << j << endl;
6  }
7  void foo(int i, float f) {
8    cout << "entier et réel " << i << " " << f << endl;
9  }
10 int main() {
11   int n = 5; float x = 3.14;
12   foo();         // affiche "rien"
13   foo(2);        // affiche "un entier 2"
14   foo(2, 7*n);   // affiche "deux entiers 2 35"
15   foo(x);        // affiche "un réel 3.14"
16   foo(1, x);     // affiche "entier et réel 1 3.14"
17 }
```

Conversion implicite

Retenir

Au besoin, le compilateur peut faire une conversion implicite s'il ne trouve pas de correspondance directe.

choix.cpp

```
1  void foo() { cout << "rien" << endl; }
2  void foo(int i) { cout << "un entier " << i << endl; }
3  void foo(float f) { cout << "un réel " << f << endl; }
4  void foo(int i, int j) {
5    cout << "deux entiers " << i << " " << j << endl;
6  }
7  void foo(int i, float f) {
8    cout << "entier et réel " << i << " " << f << endl;
9  }
10 int main() {
11   foo(true);    // affiche "un entier 1" (conversion bool -> int)
12 }
```

Retenir (Ambiguïté en C++)

*S'il n'y a pas de correspondance directe, mais **plusieurs choix** en utilisant des conversions, le compilateur signale une **erreur**, avec la liste de tous les choix possibles.*

```
choix.cpp:22:3: error: call to 'foo' is ambiguous
  foo(double(3.14));
  ^~~
choix.cpp:6:6: note: candidate function
void foo(int i) { cout << "un entier " << i << endl; }
     ^
choix.cpp:7:6: note: candidate function
void foo(float f) { cout << "un réel " << f << endl; }
```

Retenir

On résout l'ambiguïté en convertissant explicitement les paramètres vers les types voulus.

Dans l'exemple, foo(int(3.14)) ou foo(float(3.14)) sont possibles.

Plan

1 Préliminaire : les manipulateurs d'entrée sortie

2 Fonction : déclaration et définition

3 Pourquoi la surcharge ?

4 Surcharge de fonctions

5 Surcharge d'opérateurs

Les opérateurs sont des fonctions

Retenir

Chaque opérateur correspond à une fonction dont le nom est `operator` *suivi du symbole de l'opérateur.*

oper.cpp

```
1    string a = "toto", b = "bla";
2    // Les deux lignes suivantes sont équivalentes:
3    cout << a + b << endl;
4    cout << operator+(a, b) << endl;
```

La ligne 4 ci-dessus est pour la démonstration. On écrit jamais ça dans un programme : c'est illisible.

Compléments

Note : sur les types de bases (`int`, `float`,...), la fonction associée n'existe pas.

Les opérateurs sont des fonctions

Retenir

Chaque opérateur correspond à une fonction dont le nom est `operator` *suivi du symbole de l'opérateur.*

oper.cpp

```
1    string a = "toto", b = "bla";
2    // Les deux lignes suivantes sont équivalentes:
3    cout << a + b << endl;
4    cout << operator+(a, b) << endl;
```

La ligne 4 ci-dessus est pour la démonstration. On écrit jamais ça dans un programme : c'est illisible.

Compléments

Note : sur les types de bases (`int`, `float`,...), la fonction associée n'existe pas.

Surcharges d'opérateurs

Retenir

*La **surcharge d'un opérateur** se fait en **définissant la fonction associée**.*

rational.cpp

```
53  struct Rat { // Rationnel
54    int numer, denom;
55  };

80  // Construit un Rat à partir de la fraction num/den
81  Rat Ratio(int num, int den);

141 Rat operator+(Rat a, Rat b) {
142   return Ratio(a.numer * b.denom + a.denom * b.numer,
143                a.denom * b.denom );
144 }
```

Utilisation d'un d'opérateur surchargé

Attention

On appelle directement l'opérateur et surtout pas la fonction associé.

rational.cpp

```
191     // NON !!! C'est illisible
192     cout << operator+(rat1, operator/(rat1, rat2)) << endl;
193
194
195     // Ahh, c'est clair !!!
196     cout << rat1 + rat1/rat2 << endl;
```

Les opérateurs surchargeables

Retenir

Liste non exhaustive d'opérateurs surchargeables :

arithmétique	logique	comparaison
`+a`		`a == b`
`-a`	`!a`	`a != b`
`a + b`	`a || b`	`a < b`
`a - b`		`a > b`
`a * b`	`a && b`	`a <= b`
`a / b`		`a >= b`
`a % b`		

La surcharge des opérateurs `!`, `||` *et* `&&`, *modifie aussi le comportement des variantes syntaxiques* `not`, `or` *et* `and`.

`https://en.cppreference.com/w/cpp/language/operator_arithmetic`

La surcharge des opérateurs de comparaison

On peut surcharger == != < > <= >=, mais :

Attention

Chaque opérateur doit être surchargé indépendamment.

Par exemple :

- Si l'on surcharge ==, ça ne surcharge pas !=.
- Si l'on surcharge <, ça ne surcharge pas >.
- Si l'on surcharge <=, ça ne surcharge pas >=.

La surcharge des opérateurs de comparaison

Attention

Chaque opérateur doit être surchargé indépendamment.

Retenir

Donc, sauf cas très particulier,

- *soit on surcharge* `==` *et* `!=`*.*
- *soit on surcharge les 6 opérateurs* `==` `!=` `<` `>` `<=` `>=`

Note : Pour éviter de dupliquer le code, les opérateurs peuvent s'appeler les uns les autres.

Comparaison des nombres rationnels

rational.cpp

```
132  bool operator==(Rat a, Rat b) {
133    return a.numer == b.numer and a.denom == b.denom;
134  }
135  bool operator!=(Rat a, Rat b) {
136    return not (a == b);
137  }

167  bool operator<=(Rat a, Rat b) { return (a-b).numer <= 0; }
168  bool operator>=(Rat a, Rat b) { return b <= a; }
169  bool operator<(Rat a, Rat b) { return (a-b).numer < 0; }
170  bool operator>(Rat a, Rat b) { return b < a; }
```

La surcharge des opérateurs d'affichage

Retenir

L'affichage est effectué par l'opérateur << appelé « ***opérateur d'insertion****». Il prend en paramètre une référence & sur flux de sortie (de type* `ostream`*) et doit retourner ce flux.*

```
ostream& operator<<(ostream& sortie, T x);
```

rational.cpp

```
116  ostream& operator<<(ostream& sortie, Rat a) {
117    if (a.denom == 1) sortie << a.numer;
118    else sortie << a.numer << "/" << a.denom;
119    return sortie;
120  }
```

Flux de sortie

Compléments

Explication du type du paramètre :

```
ostream& operator<< (ostream &sortie, T x);
```

- `ostream` est le type des flux de sortie comme `cout` ou `cerr`, ou encore, comme les fichiers ouverts en écriture :

```
// ofstream est un sous-type de ostream
ofstream fichier;                       // Déclaration
fichier.open("bla.txt");                // Ouverture
fichier << "Noel " << 42 << endl; // Écriture;
fichier.close();                        // Fermeture
```

- On fait un passage par référence car on va modifier le flux.

Flux de sortie (2)

Compléments

Explication de la valeur de retour :

```
ostream& operator<< (ostream &sortie, T x);
```

- On retourne le flux de manière à pouvoir enchaîner les affichages sur une seule ligne :

  ```
  cout << "i=" << i << endl;
  ```

  se lit :

  ```
  ((cout << "i=") << i) << endl;
  ```

  ou encore :

  ```
  operator<<(operator<<(operator<<(cout, "i="), i), endl);
  ```

Flux de sortie (3)

Compléments

Pour éviter la recopie de l'objet à afficher (ici `x` de type `T`), on trouve très souvent le type suivant :

```
ostream& operator<<(ostream &sortie, const T& x);
```

où l'on passe `x` par référence sur une constante.

La surcharge des opérateurs de saisie

Retenir

*La saisie est effectuée par l'opérateur >> appelé «**opérateur d'extraction**». Il prend en paramètre*

- *une référence & sur un flux d'entrée (de type `istream`) et doit retourner ce flux.*
- *une référence sur l'objet à saisir (de manière à pouvoir le modifier, voir l'UE ASD).*

rational.cpp

```
123  istream& operator>>(istream& in, Rat& a) {
124    int num, den;
125    in >> num >> den;
126    a = Ratio(num, den);
127    return in;
128  }
```

Surcharge des opérateurs de modification

Compléments

On peut aussi surcharger les opérateurs qui modifient l'objet comme ++ où += :

```
R& operator++(T& x);        // opérateur préfixe  ++x
R  operator++(T& x, int);   // opérateur postfixe   x++
```

Le paramètre int n'est jamais utilisé. Il est là pour faire la différence entre les versions préfixe et postfixe.

```
R& operator +=(T& x, S y);  // opérateur x += y
```

Surcharge et objets

Attention

Les classes et la programmation objet fournissent une autre syntaxe pour surcharger les opérateurs. **C'est souvent cette autre syntaxe qui est utilisée.**

Compléments

Les classes fournissent aussi la possibilité de surcharger d'autres opérateurs comme

- l'opérateur d'assignement =. On peut donc changer le sens de `a = b;` pour les types définis par l'utilisateur.
- l'opérateur d'appel de fonction `x(...)`. On peut donc faire des objets qui se comportent comme des fonctions.

Les dangers de la surcharge

Attention

Encore plus qu'avec les fonctions, la surcharge des opérateurs peut rapidement rendre un programme surprenant voir illisible.

rev-string.cpp

```
1  // Retourne une chaîne de caractères
2  string operator-(string s) {
3    string res;
4    for (int i = s.size()-1 ; i >= 0; i--)
5      res.push_back(s[i]);
6    return res;
7  }
8
9  int main() {
10   string s1 = "coucou";
11   cout << -s1 << endl;   // ??????????
12 }
```

Les dangers de la surcharge

Attention

Encore plus qu'avec les fonctions, la surcharge des opérateurs peut rapidement rendre un programme surprenant voir illisible.

Compléments

Pour cette raison, certains langages ont décidé de ne pas permettre la surcharge des opérateurs (par exemple Java).