PG208
Programmation Orientée Objet / Langage C++
TP3

N. Ajmi Y. Bornat

January 29, 2025

Lors de cette séance, nous allons continuer à casser les limitations du C. Aujourd'hui les types de données sont à l'honneur. En C, il s'agissait simplement de définir comment les données étaient stockées. Maintenant, nous allons voir comment il est également possible d'influencer comment les données seront interprétées et traitées, uniquement grâce à leur type.

1 Surcharges

Vous avez remarqué qu'une fonction peut avoir des comportements différents selon la nature et le nombre de ses arguments ? Par exemple, l'opérateur de flux `<<` ne réagit pas de la même façon avec des entiers et des flottants. Si `a` vaut 1, la ligne `std::cout << a << std::endl;` affichera `1` ou `1.0` selon son type. C'est parce que, selon le cas, il s'agit tout simplement de fonctions différentes qui ont le même nom.

On parle de fonctions surchargées (*overloaded* en anglais).

1.1 La triche : Valeurs par défaut

La première possibilité, qui dans les fait n'est *pas* une surcharge, c'est de fournir une valeur par défaut aux arguments. En effet, à partir du moment où un argument a une valeur par défaut, il devient optionnel. Ainsi, la fonction `add` suivante peut être utilisée avec deux, trois ou quatre arguments :

```
1  float add(float x1, float x2, float x3 = 0.0, float x4 = 0.0) {
2      return x1 + x2 + x3 + x4;
3  }
```

Si utilisée avec deux arguments (`add(2,5)`), les deux valeurs seront attribuées à `x1` et `x2` respectivement. La fonction utilisera 0.0 pour `x3` et `x4`. Bien qu'il soit techniquement possible d'utiliser `add` avec un ou même aucun argument, il faut reconnaitre que c'est un peu inutile dans notre cas. On ne s'est donc pas embêté à mettre des valeurs par défaut pour `x1` et `x2`.

Si la fonction est utilisée avec trois arguments, ce dernier sera attribué à `x3` dont la valeur initiale sera ignorée. De la même façon, si on fournit quatre arguments, il sera attribué à `x4`.

1.2 La vraie surcharge

Le principe de la surcharge est que les fonctions ne sont plus distinguées *que* par leur nom, mais aussi par leurs arguments. Ainsi, une fonction `void toto(int)` peut cohabiter avec une fonction `void toto(float)` sans qu'il y ait la moindre chose en commun si ce n'est le nom. Par contre, il n'est pas possible de surcharger deux fonctions si elles ne diffèrent que par la valeur qu'elles renvoient. En effet, dans le cas `foo(bar(x))`, la valeur de retour de `bar(x)` est utilisée pour déduire quelle est la bonne fonction `foo` à utiliser. Elle ne peut donc pas servir à déduire la bonne fonction `bar`.

1.3 Quelques précautions

Cette technique a bien sûr des limites : il faut que les arguments fournis permettent de définir quelle fonction doit être appelée de façon non ambigüe. Par exemple, les surcharges suivantes sont interdites :

```
1  // value addition
2  float add(float x1, float x2, float x3 = 0.0, float x4 = 0.0);
3
4  // add a single value to all terms
5  float add(std::vector<float> v1, float x2 = 0.0);
6
7  // term to term addition
8  float add(std::vector<float> v1, std::vector<float> v2 = DEFAULT_VALUE);
```

En effet, si on appelle la fonction `add` sur un simple `std::vector<float>`, il n'est pas possible de savoir quelle fonction est ciblée.

1.4 Exercices

Comme exemple, nous allons reprendre différentes versions de la fonction du TP1 qui permettait d'additionner deux conteneurs (bon OK, deux vecteurs). Nous l'appellerons `add`. Écrivez les différentes versions suivantes :

- `std::vector<double> add(std::vector<double> v1, std::vector<double> v2)` : La version de base, qui renvoie la somme éléments par éléments des deux vecteurs.
- `std::vector<double> add(std::vector<double> v, double d)` : Pour cette version, on additionne `d` à chaque élement de `v`
- `double add(double d, std::vector<double> v)` : Cette fois, c'est `d` qui est en premier, donc on va additionner `d` et la somme des éléments de `v`. OK, cette version est un peu inutile, mais c'est pour l'exercice. Elle prend tout son sens sur des données ou l'addition n'est pas commutative.

Bien évidemment, l'intérêt est que *toutes* ces fonctions soient déclarées et utilisées dans le même programme[1]. Là où en C, il aurait fallu plusieurs fonctions avec des noms différents et donc tout un ecosystème à connaitre, avec la surcharge, il suffit de connaitre une fonction et, si les surcharges sont bien pensées et intuitives, l'usage d'un nouveau type de données devient presque instinctif. Par contre, même si l'usage pour l'addition est instinctif, ça reste lourd à lire. Comme si on allait s'arrêter en si bon chemin, jetez un oeil à la partie suivante...

2 Opérateurs en ligne

Maintenant que nous avons vu comment créer une fonction d'addition qui soit capable d'accepter toute une variété d'arguments, ce qui serait vraiment bien, c'est de l'utiliser avec l'opérateur `+` comme une vraie addition. Ça tombe bien, c'est prévu. Pour ça, il suffit que votre fonction s'appelle `operator+` et accepte deux arguments (pas plus). La ligne `a + b` est alors équivalente à `operator+(a, b)`. Reprenez les exemples de la partie précédente si vous ne me croyez pas.

C'est le principe de surcharge des fonctions (ici un opérateur) qui permet d'appeler la fonction appropriée.

2.1 Liste non exhaustive des opérateurs surchargeables

Il y a assez peu de limites aux opérateurs qui peuvent être surchargés. Il suffit de déclarer la fonction `operator`XX en remplaçant *XX* par l'opérateur de votre choix.

Il est notamment possible de surcharger (au moins) les opérateurs binaires suivant : `+`, `-`, `*`, `/`, `==`, `!=`, `>=`, `<=`, `>`, `<`, `&`, `&&`, `|`, `||`, `^`, `>>`, `<<`, `+=`, `-=`, `*=`, `/=`, `&=`, `&&=`, `|=`, `||=`, `^=`, `>>=` et `<<=`.

Mais il y a aussi les opérateurs unaires `+`, `-`, `~` et `!`.

Une restriction tout de même : Il n'est pas possible de surcharger les opérateurs pour des types élémentaires comme `int`, `char`, `float` ou `double`. Cette restriction s'applique même si l'opérateur n'est pas défini (par exemple, il n'est pas possible de définir `operator<<(float, int)` pour faire `3.14<<2`).

[1] Je vous vois, celles et ceux qui commentent les fonctions qui ne leur servent plus. Pour cet exercice, c'est interdit. ;)

2.2 Petites remarques pour finir

Le compilateur ne comprend et ne restreint pas les opérations associées aux pointeurs. Vous pouvez donc aussi surcharger les opérateurs unaires `*` et `&`, ce n'est cependant pas conseillé pour éviter de perdre le programmeur/lecteur.

Il faut aussi rester cohérent avec les règles mathématiques, notamment pour les règles de précédence (priorité), de commutativité ou autres. À ce titre, la possibilité de concaténer deux `std::string` grâce à l'opérateur `+` est décriée car elle n'est pas commutative, contrairement à l'addition mathématique sur des entier ou des flottants.

2.3 Exercices

- Étant donné qu'il est possible de concaténer des `std::string` avec l'opérateur `+`, il semble logique que pour une `std::string s`, on définisse `2*s` comme étant `s+s`, ou encore `3*s == s+s+s`. Surchargez l'opérateur `*` pour cela.
- Toujours en surchargeant l'opérateur `*`, proposez le calcul du produit scalaire de deux `std::vector`.
- Cette fois-ci, nous allons surcharger l'opérateur `<<` pour faire une rotation des éléments d'un vecteur. Ainsi, si `v` vaut `{1,2,3,4}`, `v << 1` donnera `{2,3,4,1}`. Testez votre programme en utilisant une ligne de la forme `v1 = v2 << 2`, et une autre version en utilisant une ligne de la forme `v <<= 3`.

3 Templates

Pour revenir sur le problème de l'exercice d'addition élément par éléments. Il n'était bien sûr pas envisageable de faire des surcharges en écrivant une fonction différente pour chaque longueur. La déclaration impose de donner la taille du tableau dans le type et c'est un peu bloquante d'un certain abord, mais peut être très utile au final. En effet, on peut imposer dès la déclaration de la fonction, que deux conteneurs passés en argument soient de même taille.

Pour cela, on définit un type temporaire uniquement dans le cadre de la fonction, on peut forcer ce même type sur les deux arguments de la fonction, et sur le résultat. Au final, on s'est un peu embêtés, mais on est sûrs que les deux conteneurs en entrée auront la même longueur et que la sortie aura cette même longeur. Pour cela, le mot-clef est `template`.

3.1 Syntaxe

Pour créer un *template* (ou *modèle* en bon français que personne n'utilise dans ce contexte), la syntaxe est `template <std::size_t SIZE>`, où `SIZE` est le nom que l'on veut lui donner. Ainsi, une fonction d'addition élément par élément de tableaux ressemblera à :

```
1  template <std::size_t SIZE>
2  std::array<int, SIZE> operator+(const std::array<int, SIZE>& t1, const std::array<int, SIZE>& t2){
3      std::array<int, SIZE> res;
4      for (int i {0}; i<t1.size(); i++) {
5          res[i] = t1[i] + t2[i];
6      }
7      return res;
8  }
```

Remarques :

- Dans la boucle, on fait appel à `t1.size()`, on aurait aussi pu faire appel à `t2.size()` car la vérification des type de données à la compilation garantit que les deux tableaux ont la même taille. À l'usage, pour être encore plus efficace, on utilisera directement `SIZE`, car le compilateur peut mieux optimiser les choses.
- Le type de SIZE est `std::size_t`. Techniquement, il s'agit d'un entier, mais le fait de préciser `size_t` permet d'être plus précis.
  - il détermine la taille de quelque chose.
  - son occupation mémoire est directement liée à la capacité de la machine.

Techniquement, toutes les fonctions de longueur, d'accès aux éléments d'un tableau, ou même `std::sizeof()` ne renvoient pas un `int` ou un `long` mais un `std::size_t`[2] (c'est vrai aussi pour le C). C'est juste que le compilateur fait habituellement la conversion implicitement car les types sont compatibles. Ici, ce n'est pas le cas. Il faut être rigoureux sur les types de données pour éviter les problèmes lors des surcharges

- Les deux arguments sont passés comme des références constantes. WTF ? Si on déclare les entrées avec `const`, cela signifie qu'on ne peut pas les modifier. mais si on ne peut pas les modifier, pourquoi utiliser des références ??? Nous l'avons déjà vu, c'est tout simplement pour éviter une copie. Si les tableaux ont une grande taille, éviter la copie réduit l'occupation mémoire et le temps de calcul, c'est bon pour la planète[3].

- Mais alors pourquoi ne pas retourner une référence ? Pour les mêmes raison qu'on ne renvoie jamais un pointeur vers une variable locale. La variable va être détruite et la référence cassée. Il est alors inévitable de consommer beaucoup de mémoire et de temps de copie pour implémenter cette fonction comme une surcharge de `+`, c'est une des raisons qui explique qu'elle ne soit pas présente nativement.

- Vous avez remarqué qu'il n'y a pas de ; à la fin de ligne template ? c'est tout simplement parce que cette ligne fait partie de la déclaration de la fonction mais par convention, on place le template sur la ligne au-dessus pour améliorer la lisibilité. Cela signifie aussi que le template est local (spécifique à la fonction).

3.2 La généricité

Il est possible de pousser le concept encore plus loin. En effet, le type de donnée que contient le tableau a peu d'importance. Il suffit que l'opérateur `+` soit définit pour ce type. Il est donc possible d'écrire une fonction unique, indifférente du type contenu dans le tableau.

```
1  template <typename CONTENT, std::size_t SIZE>
2  std::array<CONTENT, SIZE> operator+(const std::array<CONTENT, SIZE>& t1, const std::array<CONTENT,
   SIZE>& t2){
3      std::array<CONTENT, SIZE> res;
4      for (int i {0}; i<SIZE; i++) {
5          res[i] = t1[i] + t2[i];
6      }
7      return res;
8  }
```

Remarques :

- Cette surcharge va fonctionner avec tous les tableaux contenant tout type de donnée pour lequel l'opérateur `+` est défini. C'est à dire (à la louche), les `int`, `char`, `short`, `long`, `float`, `double` (classique), mais aussi `std::string` (eh oui !) et surtout `std::array`, puisque qu'on vient de le définir. En définissant l'addition sur les tableaux, on a, presque par accident, aussi défini l'addition sur les tableaux de tableaux, ou les tableaux de tableaux de tableaux ... .

- Si pour un besoin quelconque il est nécessaire de créer une variable ayant le type contenu dans le tableau, il suffira de la déclarer par `CONTENT variable { ...};`, puisque le template prévient le compilateur que `CONTENT` est un type de donnée. Ici, l'initialisation uniforme prend tout son sens, car on ne sait pas si `CONTENT` est un type de base (`int`, `double` ...) un conteneur, ou même un `struct` quelconque ...

Le grand atout de la généricité est de pouvoir définir des conteneurs indépendamment du type de données contenues. Par exemple, si vous avez besoin de trois listes chainées, mais que chacune de ces listes contient un type de données différent, En C, il faut écrire trois types différents et répéter trois fois chacune des fonctions associées. En C++, on écrit un type de donnée comme suit :

[2]Pour celles et ceux qui activent le flag `-Wall` à la compilation, vous avez dû vous en rendre compte en essayant de compiler `for (int i  0 ; i < v.size() ; i++)`. Un `std::size_t` comptant comme un entier non-signé, `g++` va vous prévenir que vous comparez un entier signé et un entier non-signé.

[3]Techniquement, la planète survivra, pour être plus précis, c'est bon pour la biodiversité et donc pour l'espèce humaine. Vous pouvez également être égoïste à plus court terme en disant que c'est bon pour mieux vendre son produit

```
template <typename CONTENT>
struct list {
    CONTENT        value;
    list<CONTENT> *next;
}
```

Pour créer une variable de ce type, on utilise le mot-clé `list<CONTENT>`. L'usage de `struct` n'est plus nécessaire en C++.

3.3 Exercices

- Écrivez la fonction `total` qui calcule la somme de tous les éléments d'un vecteur, sans restriction sur le type stocké dans ce vecteur, à condition que ce type soit supporté par `operator+`.
- Écrivez la fonction `display`, qui, pour tout conteneur, affiche son contenu sur plusieurs lignes de la forme : `indice : element`, à raison d'un élément par ligne (pour celle-ci il y a un piège :-)).

4 Conclusion

Cette séance a permis d'expérimenter des comportements de niveau plus élevé que ce que vous connaissiez en C *tout court*. Nous n'avons malheureusement qu'un aperçu des possibilités du langage, mais l'essentiel transparait déjà : dans la programmation orientée objet, l'essentiel est de bien décrire le problème dans des structures de données, et, si besoin, créer les structures sur mesure pour obtenir une bonne description. Le reste se règle uniquement en définissant (surchargeant) les opérateurs utiles à ces structures.