12
Pointeurs et références

Il apparaît souvent au fil des discussions sur les forums que vous êtes nombreux à ne pas trop savoir quand utiliser des valeurs, des pointeurs ou des références, et, surtout, quand avoir recours à l'allocation dynamique de la mémoire.

C'est d'autant plus vrai si vous venez de C ou de Java qui sont des langages pour lesquels l'utilisation de `malloc` (pour C) ou de `new` (pour Java) fait partie des pratiques courantes et incontournables.

Ceux qui viennent de C ont un avantage par rapport à ceux qui viennent de Java : ils ont l'habitude de gérer la durée de vie de leurs objets (comprenez : de décider du moment où la mémoire allouée à un objet doit être libérée), alors que ceux qui viennent de Java ont tendance à se reposer essentiellement sur le garbage collector.

Mais dans tous les cas, si vous êtes l'un d'eux, attendez-vous à devoir jeter vos habitudes à la poubelle si vous voulez développer efficacement en C++.

7. Démystifions les pointeurs

De nombreux problèmes concernant les pointeurs viennent simplement d'une mécompréhension de ce qu'est un pointeur, ou, du moins d'une connaissance insuffisante de ce que c'est et de ce que cela implique.

Commençons donc par démystifier les pointeurs en expliquant clairement ce qu'ils sont : Un pointeur n'est rien d'autre qu'une variable numérique entière (généralement) non signée et particulière dans le sens où elle représente l'adresse mémoire à laquelle on trouvera (ou du moins, à laquelle on est censé trouver) effectivement un élément du type indiqué. Lorsque vous manipulez un pointeur, vous ne faites donc rien d'autre que manipuler une valeur numérique entière.

Pour sa part, le compilateur manipulera les pointeurs exactement de la même manière que celle qu'il utilise pour manipuler les valeurs numériques entières – avec cependant quelques spécifications propres aux pointeurs pour tout ce qui a trait à l'arithmétique des pointeurs – avec tout ce que cela implique.

Une fois que l'on a compris qu'un pointeur n'est rien de plus que cela, il devient beaucoup plus facile d'envisager sereinement de les utiliser.

Mais il reste cependant énormément d'écueils à leur emploi. Les sections suivantes vous aideront à en prendre conscience et à trouver, chaque fois que ce sera possible, des solutions pour vous en passer.

8. Qu'est-ce qu'une référence ?

Une référence représente tout simplement un alias de l'objet référencé et ne fait que donner un autre nom à la variable qu'elle référence. Tout ce qui peut arriver à une référence arrive donc forcément à l'objet référencé.

Si l'on souhaite s'assurer que l'objet référencé ne sera pas modifié, il faut déclarer la référence comme constante (voir chapitre *La constance, autant que faire se peut*).

9. Quelle différence entre les deux ?

La différence la plus importante, en dehors de la syntaxe utilisée pour accéder aux membres d'une structure ou d'une classe, est qu'un pointeur peut parfaitement pointer vers une adresse connue comme étant invalide (`NULL`, ou, si vous disposez de C++11, `nullptr`) alors que la référence vous apportera justement une garantie de non-nullité. Cette garantie est un invariant appliqué dans le contexte d'utilisation de la référence.

Autrement dit, le pointeur vous permettra, au travers de la valeur `NULL` (de `nullptr` en C++11), de représenter la *non-existence* de l'objet, alors que la référence vous apportera justement la garantie que l'objet référencé existe bel et bien.

La deuxième différence est que la référence doit être directement initialisée avec l'objet auquel elle fait référence. En effet, bien qu'il ne soit jamais conseillé de déclarer un pointeur sans l'initialiser, un code proche de

```
int * ptr ; // on déclare le pointeur sans lui donner de valeur
           // particulière
/* plusieurs instructions, tant qu'on ne touche pas à ptr */
ptr = new int ; // on n'initialise ptr qu'ici
```

sera tout à fait valide et accepté par le compilateur.

Par contre, vous devrez systématiquement indiquer l'objet référencé par une référence, ce qui fait qu'un code proche de

```
int & ref ; // déclarer une référence sans l'initialiser est
            // interdit
/* plusieurs instructions, tant qu'on ne touche pas à ref */
int i ;
ref = i ; // on n'initialise ref qu'ici
```

est illégal en C++ et il en résultera une erreur de compilation.

Enfin, il est possible de faire pointer un pointeur vers un objet différent de celui d'origine, comme le montre ce petit bout de code :

```
#include <iostream>
int main(){
    int i = 4;
    int j = 6;
    int * ptr = & i; // ptr contient l'adresse mémoire de i
    (*ptr)*=2;
    std::cout<<i<<std::endl;// affiche 8
    ptr = & j; // ptr contient maintenant l'adresse mémoire de j
    (*ptr)*=2;
    std::cout<<i<<" "<<j<<std::endl; // affiche 8 12
    return 0;
}
```

Mais il est impossible de changer l'objet référencé par une référence comme le montre cet autre morceau de code :

```
include <iostream>
int main(){
    int i = 4;
    int j = 6;
    int & ref = i; //ref fait référence à i
    ref*=2;
    std::cout<<i<<std::endl; // affiche 8
    ref =  j; //affecte la valeur de j à ref... c'est à dire à i ❶
    std::cout<<i<<" "<<j<<std::endl; // affiche ... 6 6
    i*=2;
    std::cout<<i<<" "<<j<<std::endl; // affiche ... 12 6
    return 0;
}
```

Ici, l'affectation `ref = j ;` ❶ a strictement le même effet que si on avait écrit `i = j ;` et l'opérateur d'affectation s'applique donc bel et bien à la variable `i`, sans que cela n'influe en aucune manière sur ce qui pourra arriver à `j` par la suite.

10. Transmettre des paramètres : par valeur ou par référence ?

La réponse à cette question nécessite de s'intéresser à quatre aspects bien distincts :

- le type de l'objet passé en paramètre ;
- l'espace mémoire nécessaire à la représentation en mémoire de l'objet passé en paramètre ;
- le besoin de s'assurer que les modifications apportées au paramètre seront répercutées vers l'objet utilisé comme paramètre ;
- la nécessité de disposer de comportements polymorphes.

Le type de l'objet passé en paramètre ?

Le type de l'objet à passer en paramètre est important à plus d'un titre lorsque l'on doit décider de le transmettre par valeur ou par référence.

En effet, certains types vont nécessiter beaucoup de ressources lors de leur copie, que ce soit du point de vue de la mémoire nécessaire à la copie ou du point de vue du temps nécessaire à la copie complète d'un objet.

Le fait de passer ce genre d'objet par référence plutôt que par valeur permet d'en éviter la copie et donc de gagner en performance (parfois de manière assez spectaculaire) à très peu de frais, sans imposer de changement particulier à l'utilisation de la fonction par rapport au passage par valeur.

De plus, nous verrons plus loin que certains types d'objets n'ont, purement et simplement, pas vocation à être copiés, bien qu'il soit malgré tout intéressant de pouvoir les utiliser comme argument pour une fonction.

Le passage par référence, ou par pointeur, devient alors la seule solution qui nous permette de transmettre ce genre d'objet à une fonction, tout en tenant compte de l'impossibilité d'en créer une copie.

L'espace mémoire nécessaire ?

Je ne vais pas vous mentir : si vous observez attentivement le code assembleur généré lors de l'utilisation d'une référence sur un objet, vous observerez qu'il correspond trait pour trait au code généré lorsque vous utilisez un pointeur sur ce même objet.

Le passage d'un objet par référence nécessitera donc, au niveau du code assembleur généré, l'utilisation d'une variable numérique entière dont la taille correspond strictement à la taille utilisée pour représenter un pointeur.

Il peut donc être – en dehors de toute autre considération que l'espace mémoire nécessaire à la représentation de la donnée – contre-productif de transmettre par référence des objets pour lesquels l'espace mémoire nécessaire à leur représentation est inférieur à la taille d'un pointeur.

Autrement dit, si vous n'avez aucune autre raison pour transmettre un type primitif (ou une structure personnelle de taille équivalente) par référence, vous avez sans doute intérêt à le transmettre simplement par valeur.

Attention *› Bien qu'une référence soit traitée au niveau du code assembleur exactement de la même manière qu'un pointeur, une référence apporte de nombreuses garanties que les pointeurs ne sont pas à même d'apporter au niveau du code que vous écrivez. Vous ne devez donc jamais utiliser l'argument du "mais c'est de toute façon pareil du point de vue de l'assembleur" pour justifier de l'utilisation d'un pointeur plutôt que d'une référence : vous avez décidé de coder en C++ et non en assembleur, que diable !*

La nécessité de répercuter les modifications ?

L'idéal est toujours d'éviter autant que possible que les modifications apportées au paramètre d'une fonction soient répercutées vers l'objet qui a servi d'argument[1], cependant il y a des circonstances où cela reste malgré tout la moins mauvaise solution.

Seule la transmission du paramètre par référence ou par pointeur est en mesure de faire en sorte qu'une modification effectuée dans la fonction appelée soit répercutée vers la variable utilisée comme paramètre dans la fonction appelante, car le passage par valeur implique que la fonction appelée travaille sur une copie temporaire de l'argument qui est propre à la fonction appelée.

Je me dois d'insister encore sur le fait que travailler de la sorte peut occasionner de nombreux soucis dont j'ai déjà parlé dans le chapitre relatif aux effets de bord.

D'un autre côté, comme le passage par référence non constante permet cette utilisation et que vous risquez de vous trouver dans une situation où vous auriez besoin de cette possibilité, il ne serait pas honnête de ma part de ne pas vous en parler.

Gardez néanmoins en tête qu'il est largement conseillé d'éviter autant que faire se peut le recours à cette possibilité.

[1] Voir le chapitre *Éviter les effets de bord*.

Disposer de comportements polymorphes ?

Comme je vous l'expliquais dans le chapitre *Les grands principes de la POO*, le propre de la programmation orientée objet est de permettre ce que l'on appelle la substituabilité.

La *substituabilité* est la capacité qui nous est donnée de transmettre un objet de type dérivé à une fonction qui attend une référence (ou un pointeur) vers un objet du type de base tout en observant, le cas échéant, des comportements adaptés au type dérivé. Le fait qu'un argument "connu pour être du type de base" puisse avoir un comportement adapté au type réel de l'objet qui a servi d'argument s'appelle le *polymorphisme*.

Un type d'objet ou un comportement qui présente la capacité d'utiliser le polymorphisme est alors appelé un type ou un comportement polymorphe.

En C++, on ne peut profiter du polymorphisme que lorsqu'on manipule les objets sous la forme de références (éventuellement constantes) ou sous la forme de pointeurs.

Outre le fait qu'un objet polymorphe n'a généralement pas vocation à pouvoir être copié[2], le passage par valeur occasionnerait la copie de la seule partie de l'objet qui correspond au type de base (on appelle cela le *slicing*) et le comportement observé correspondrait alors au comportement du type de base et non au comportement du type réel de l'objet ayant servi d'argument.

Note *› Il existe en réalité plusieurs types de polymorphisme : lorsque l'on parle (par abus de langage) de polymorphisme sans donner de précision, on parle essentiellement de polymorphisme d'inclusion (voir Comprendre le polymorphisme).*

Le seul moyen de profiter de comportements polymorphes en C++ passe donc par l'utilisation de pointeur ou de référence.

[2] Voir le chapitre *Salade de sémantique*.