Chapitre 3 : pointeurs, références, tableaux

1 Les pointeurs

1.1 Principe

Comme vu dans le chapitre précédent, il est possible de manipuler les *adresses* des variables que l'on a définies. Par exemple, la déclaration `int n;` a pour effet de réserver un certain nombre [1] de cases mémoires (le nombre nécessaire pour stocker une variable de type `int`) pour stocker la variable `n`. La syntaxe `&n` correspond alors à l'adresse de la variable `n`. Cet objet est de type *pointeur sur* `int`, ce qui se note `int *`. Par exemple, si l'on souhaite stocker la valeur de `&n` dans une variable `adresse_n`, on devra préalablement déclarer `int *adresse_n;`. La syntaxe `*adresse_n` permet alors d'accéder au contenu de la case mémoire située à l'adresse `adresse_n`. Notamment, si `n` est une variable, alors les expression `n` et `*&n` renvoient la même valeur.

Il est possible de définir des variables de type pointeur vers n'importe quel autre type de variable. La différence entre deux types de pointeurs, par exemple entre `int *` et `double *` est l'effet de l'artithmétique sur les pointeurs : si `p` est un pointeur de type `int *` (respectivement `double *`, etc.) alors `p+1` désigne la prochaine case mémoire pouvant stocker une variable de type `int` (respectivement `double`, etc.).

Il est possible de convertir n'importe quel type de pointeur vers n'importe quel autre type de pointeur : si `p` est une variable de type pointeur, on peut forcer l'interprétation de `p` comme un pointeur d'un autre type avec `(double *)p` (ici pour une conversion en pointeur sur `double`).

Comme il existe des pointeurs sur n'importe quel type de variable, il existe en particulier des pointeurs sur pointeurs. Par exemple le type `int**` désigne les pointeurs sur pointeur sur `int`. Une déclaration de ce type de variable se fait par `int **p`. L'expression `*p` est alors un pointeur sur `int` et `**p` est un `int`. Ce type servira pour manipuler des tableaux à double entrées.

1.2 Passage par adresse

L'intérêt de manier des variables de type pointeur est de permettre à une fonction de modifier ses arguments. La manière habituelle de passer des arguments à une fonction est appelée *passage par valeur*. Par exemple, considérons une variable `m` de type `int` et de valeur `0`, et considèrons la définition

```
void plus_un_1(int n)
{
n ++;
}
```

L'appel de `plus_un_1(m)` aura l'effet suivant :

— On évalue la valeur de `m` (ici `0`) ;

— Cette valeur est recopiée dans une variable, ici `n`, locale à la fonction `plus_un_1` ;

1. Ce nombre de cases dépend de la machine, est peut être obtenu par `sizeof(int)` (où `int` peut bien sûr être remplacé par n'importe quel autre type).

— Cette variable est modifée par l'opérateur `++`, et vaut donc `1` ;
— La fonction se termine et la variable `n`, locale à la fonction disparaît, avec sa valeur.

On voit notamment que la valeur de `m` n'a pas été modifiée. En fait, l'appel de `plus_un_1(m)` est équivalent à celui de `plus_un_1(0)`, qui ne peut donc pas "voir" la variable `m`.

Pour pouvoir contourner cette limite et modifier les valeurs des arguments, on va utiliser ce qui s'appelle un *passage par adresse*. Considérons le code suivant :

```
void plus_un_2(int *n)
{
(*n) ++;
}
```

Ici, on doit fournir à `plus_un_2` l'*adresse* d'un `int`. On va donc appeler `plus_un_2(&m)`, qui aura pour effet :

— D'évaluer la valeur de `&m` (qui est l'adresse de `m`) ;
— De recopier cette valeur dans la variable `n`, locale à la fonction `plus_un_2`. `n` contient donc aussi l'adresse de `m` ;
— L'appel de `(*n)++` augmente de un la valeur présente à l'adresse `n`. Il s'agit de la valeur de `m` qui vaut donc maintenant `1` ;
— La fonction se termine et la variable `n` (qui contient toujours l'adresse de `m`) disparaît. En revanche, on dispose toujours de la variable `m`, qui contient maintenant la valeur `1` !

Il n'y a pas de contradiction avec le fait qu'une fonction ne puisse pas modifier ses arguments : ici la fonction a été appelée avec pour argument `&m`, dont la valeur n'a pas été modifiée (la variable `m` est toujours située dans la même case mémoire).

Le passage par adresse a un autre avantage. En effet, dans un passage par valeur, le valeur avec laquelle la fonction est appelée est recopiée dans une variable locale à la fonction. Cela ne pose pas de problèmes si cette valeur est de type `int` ou `double`, mais si la valeur est d'un type plus complexe, et si l'appel de la fonction est très fréquent, on peut être amené à recopier un grand nombre de fois une grande quantité de données. En revanche, lors d'un passage par adresse, on ne recopie *que l'adresse* de la variable

Un autre intérêt du passage par adresse est de permettre de modifier la valeur de retour d'une fonction. Par exemple, la fonction suivante

```
int *max(int *a, int *b)
{
return (*a>*b)? a : b;
}
```

Renvoie l'adresse de la variable ayant la plus grande valeur parmi `a` et `b`. Il est alors possible de modifier cette plus grande valeur avec une instruction telle que `(*max(&a,&b)) ++;`.

2 Les références

Les références sont une innovation du `C++` par rapport au `C`. Elle permettent de simplifier la syntaxe du passage par adresse. Si l'on reprend l'exemple de la fonction `plus_un_2` ci-dessus, deux points peuvent rendre la syntaxe un peu lourde :

— Dans la fonction, chaque utilisation de la variable nécessite l'appel de `*n`, au lieu de simplement `n`.

— À chaque appel de la fonction, on doit fournir comme argument l'*adresse* de la variable, et donc écrire `&m` plutôt que `m`.

La notion de *référence* permet de simplifier cette syntaxe. Les références sont en quelque sorte des pointeurs directement gérés par la machine. Le type référence sur `int` (par exemple) se note `int &`. L'utilisation des référence permet essentiellement de corriger les deux points de syntaxe ci-dessus. Par exemple, avec des références, les deux fonctions `plus_un_2` et `max` se réécriraient :

```
void plus_un_ref(int &n)
{
n++;
}
```

et

```
int & max_ref(int &a, int &b)
{
return (a>b)? a : b;
}
```

Leur utilisation serait alors par exemple `plus_un(m)` et `max(a,b)++`.

La syntaxe est donc plus simple en utilisant les références. Le prix à payer est qu'une référence ne peut plus être modifiée après avoir été initialisée. Par exemple, si `n` et `m` sont deux variables entières, on peut faire les déclarations `int *p=n;` et `int &r=n`. Les variables `p` et `r` sont alors respectivement un pointeur et une référence vers `n`. Pour le pointeur il est possible d'écrire `p=&m` (pour que `p` pointe vers `m`) ou `*p=m` (pour que la valeur pointée par `p`, c'est-à-dire `n`, prenne la valeur de `m`). En revanche, avec des références, on ne peut pas différencier ces deux écriture : `r=m` remplace la valeur pointée par `r`, c'est-à-dire `n`, par `m`. En conclusion, une référence ne peut pas pointer vers une autre variable que celle qui a été choisie à la déclaration.

3 Tableaux

3.1 Tableaux statiques

La déclaration d'un tableau de `N` variables de même type peut se faire avec la syntaxe `int tab[N]`. Cette instruction a pour effet de réserver les `N` cases mémoires `tab`, `tab+1`,..., et `tab+N-1`, pour stocker des entiers. La syntaxe `tab[i]` permet d'accéder à la case d'adresse `tab+i`. Notamment, les éléments de `tab` *sont numérotés de `0` à `N-1`*. On remarquera que la variable `tab` est de type pointeur sur `int`.

Ici, on dit que l'on déclare un tableau *statique*. La mémoire à réserver pour le tableau est choisie à la compilation, ce qui soumet ce type de déclaration à deux contraintes :

— la taille du tableau doit être connue à la compilation (elle ne doit pas être le résultat d'un calcul),

— la taille du tableau ne doit pas être trop grande.

On utilisera donc la plupart du temps un autre type de tableau, appelé tableaux *dynamiques*. Un cas notable où il est naturel d'utiliser des tableaux statiques est le fait de repérer un point dans le plan ou dans l'espace : les points seront représenté par des tableaux de taille deux ou trois.

On peut également définir des tableaux à deux entrée (matrices...), avec des déclaration comme `double M[3][3]`. Dans ce cas, la variable `M` est de type `double**` (pointeur sur pointeur sur `double`). L'expression `M[i]`, de type `double*`, correspond à la `i`$^{\text{ème}}$ ligne de la matrice. `M[i][j]`, correspond à l'élément de la `i`$^{\text{ème}}$ ligne et à la `j`$^{\text{ème}}$ colonne.

3.2 Tableaux dynamiques

Si `p` est une variable de type pointeur sur `double` (par exemple), on peut déclarer un tableau dynamique par la syntaxe `p = new double[N]`. Ici, la mémoire va être réservée à l'exécution du programme (et non à la compilation). Par conséquent, la valeur de `N` peut être le résultat d'un calcul.

Un tableau dynamique à double entrée doit être réservé en deux étapes : si `M` est de type `double **`, on écrira d'abord `M=new double*[n];` puis, dans une boucle avec `i` variant de `0` à `n-1`, on écrira `M[i] = new double[m]`. On a alors défini un tableau de taille `n` par `m`.

Une fois qu'un tableau dynamique est devenu inutile, il faut libérer la mémoire qui lui a été allouée, par la syntaxe `delete p[];`. Oublier de libérer la mémoire peut poser problème si on appelle de nombreuse fois une fonction qui réserve beaucoup de mémoire sans la libérer après : en effet, on se retrouve alors avec beaucoup de mémoire réservée à laquelle on n'a plus accès.