

Chapitre 3 : pointeurs, références, tableaux

Fondamentaux du C++

Master 2 Ingénierie mathématique
Sorbonne Université

Automne 2018

Plan

1 Pointeurs

2 Références

3 Tableaux

Adresses mémoire et pointeurs

- Une adresse mémoire (par exemple `&n` pour l'adresse de la variable `n`) peut être stockée dans une variable ;
- La variable en question est de type *pointeur sur int* (pour l'adresse d'un `int`), ce qui se note `int *` ;
- Exemple :

```
int *adresse_n = &n;
```

- `*adresse_n` est le contenu de la case d'adresse `adresse_n` (par exemple, si `n` est une variable, les expressions `n` et `*&n` renvoient la même valeur) ;

Passage par valeur/par adresse

Intérêts des pointeurs

- Permettre à une fonction de **modifier ses arguments** ;
- **Éviter des recopies** inutiles de variables.
- Permettre de **stocker plusieurs valeurs** sans déclarer une variable à chaque fois ;

Passage par valeur/par adresse

Passage par valeur

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

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

- On **évalue** la valeur de `m` ;
- Cette valeur est **recopiée** dans la variable `n`, **locale à la fonction** ;
- `n` est modifiée par l'opérateur `++` ;
- La fonction se termine et la variable `n`, locale à la fonction, **“disparaît” avec sa valeur.**

`m` **n'a pas** été modifiée (si `m` vaut 0, `plus_un_1(m)` est équivalent à `plus_un_1(0)`, qui ne “voit” pas la variable `m`).

Passage par valeur/par adresse

Passage par adresse

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

Quand `plus_un_2(&m)` est appelé :

- On **évalue** `&m` (l'adresse de `m`) ;
- On **recopie** cette valeur dans `n`, variable locale à `plus_un_2` ;
- `(*n)++` augmente de un la valeur **présente à l'adresse `n`** (=la valeur de `m`) ;
- La fonction se termine et `n` (qui contient toujours **l'adresse** de `m`) disparaît (mais on dispose toujours de la variable `m`).

L'argument `&m` n'a pas été modifié, contrairement à `m`.

Passage par valeur/par adresse

Autre avantage

- Les arguments des fonctions sont **recopiés** dans des variables locales à la fonction ;
- Pas de problèmes si cette valeur est de type `int` ou `double` ;
- Très **coûteux** si les variables ont un type plus complexe (**matrices 1000×1000 !**) ;
- Lors d'un passage par adresse, on ne **recopie que l'adresse** de la variable (peu coûteux).

Passage par valeur/par adresse

Le passage par adresse permet aussi de modifier la valeur de retour d'une fonction. Par exemple :

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

On renvoie l'**adresse** de la variable ayant la plus grande valeur parmi a et b. On peut **modifier** cette plus grande valeur avec une instruction telle que

```
*max(&a,&b) = 7;
```


Plan

1 Pointeurs

2 Références

3 Tableaux

Références

Deux lourdeurs de syntaxe dans les fonctions utilisant des pointeurs :

- Dans la fonction, on utilise la variable avec `*n`, plutôt que `n`.
- Pour appeler la fonction, on fournit l'*adresse* de la variable, soit `&m`, plutôt que `m`.

Références

- Ajout du C++ par rapport au C ;
- Permet de *simplifier la syntaxe* du passage par adresse.
- Nouveau type : `int &` désigne le type *référence sur int* (par exemple).

Les références

Les deux fonctions `plus_un_2` et `max` écrites avec des références :

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

et

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

Utilisation

```
plus_un(m);
max(a,b) += 2;
```

Les références

Modification d'un pointeur et d'une référence

```
int n, m;  
int *p = &n;  
int &r = n;  
*p = m; // On change la valeur pointée  
p = &m; // p pointe vers une autre case  
r = m; // On change la valeur pointée (comme *p = m)  
// Pas d'équivalent à p = &m pour les références
```

En résumé

- Syntaxe plus simple ;
- Prix à payer : une référence ne peut plus être modifiée après initialisation.

Plan

1 Pointeurs

2 Références

3 Tableaux

Tableaux statiques

- Déclaration d'un tableau **statique** de N variables de même type :

```
int tab[N] ;
```

- tab est de type **pointeur sur int**.
- Réserve les N cases mémoires d'adresses **tab**, **tab+1**, ..., et **tab+N-1**, pour stocker des entiers
- Accès à la ième case : **tab[i]** (raccourci pour ***(tab+i)** (éléments de tab numérotés **de 0 à N-1**);
- Mémoire à réserver choisie à la **compilation** (ne doit donc pas être le **résultat d'un calcul** et pas "trop grande").
- On utilisera donc la plupart du temps un autre type de tableaux : tableaux *dynamiques*.
- Un cas où il est naturel d'utiliser des tableaux statiques : repérer un point dans le plan ou dans l'espace (tableaux de taille deux ou trois).

Tableaux dynamiques

- p une variable de type pointeur sur double (par exemple) ;
- On peut déclarer un tableau dynamique par

```
p = new double[N] ;
```

- Mémoire réservée à l'**exécution** du programme (et non à la compilation) ;
- N peut être le résultat d'un calcul.
- Une fois qu'un tableau dynamique est devenu inutile, on “libère” la mémoire allouée, par

```
delete [] p;
```

- Oublier de libérer la mémoire : pose problème si on appelle de nombreuses fois une fonction qui réserve beaucoup de mémoire sans la libérer après (mémoire réservée à laquelle on n'a plus accès).