420-C21-IN Programmation II

Godefroy Borduas – Automne 2021

Module 4

Les pointeurs – Mécanisme des adresses mémoires

1

Notre premier défi ! Les pointeurs

- Qu'est-ce que c'est ?
- Comment connaitre l'adresse mémoire d'une variable ?
- Comment déclarer et utiliser un pointeur ?
- Petits détails intéressants

2

Qu'est-ce que c'est ?

3

Rappelez-vous, la mémoire et les variables

Adresse	Espace d'encodage
0x…1	**null**
0x…2	**int**
0x…3	
0x…4	
0x…5	
0x…6	**null**
0x…7	**null**
0x…8	**int**
0x…9	
0x…10	
0x…11	
0x…12	**null**

- Chaque espace contient 8 bits (1 octet)
- Une variable prend autant de cases que nécessaire
  - Sa valeur sera consignée à l'intérieur
  - Le nombre de cases dépend du type
- Supposons que les cases qui ne sont pas utilisées sont à *null*

4

Est-ce un problème ?

Adresse	Espace d'encodage
0x…1	**null**
0x…2	**int**
0x…3	
0x…4	
0x…5	
0x…6	**null**
0x…7	**null**
0x…8	**int**
0x…9	
0x…10	
0x…11	
0x…12	**null**

- Dans des cas simples, Non !
- Dans des cas plus complexes, Oui !
- Parfois, il est plus simple de se rappeler l'adresse de la case mémoire au lieu de la variable au complet
  - Pourquoi d'après vous ?

5

Imaginer un tableau de char

Adresse	Espace d'encodage
0x…1	**null**
0x…2	**char[4]**
0x…3	
0x…4	
0x…5	
0x…6	**null**
0x…7	**null**
0x…8	**null**
0x…9	**null**
0x…10	**null**
0x…11	**null**
0x…12	**null**

- Créons un tableau de 4 char
- Imaginons que vous devions le copier
- C'est assez simple

6

Imaginer un tableau de char

Adresse	Espace d'encodage
0x…1	**null**
0x…2	**char[4]**
0x…3	
0x…4	
0x…5	
0x…6	**null**
0x…7	**null**
0x…8	**char[4]**
0x…9	
0x…10	
0x…11	
0x…12	**null**

- Créons un tableau de 4 char
- Imaginons que vous devions le copier
- C'est assez simple

7

Imaginer un tableau de char

Adresse	Espace d'encodage
0x…1	**null**
0x…2	**char[24]**
0x…3	
0x…4	
0x…5	
0x…6	
0x…7	
0x…8	
0x…9	
0x…10	
0x…11	
0x…12	
0x…13	

Adresse	Espace d'encodage
0x…14	**char[24]**
0x…15	
0x…16	
0x…17	
0x…18	
0x…19	
0x…20	
0x…21	
0x…22	
0x…23	
0x…24	
0x…25	
0x…26	**null**

- Maintenant, créons un tableau de 24 char
- Imaginons que vous devions le copier
- Ça devient déjà plus compliqué

8

Existe-t-il une solution ?

- Se rappeler où l'on a rangé la variable
- Une variable de type pointeur conserve donc l'adresse où une variable est mémorisée
- Le pointeur **pointe vers** l'emplacement mémoire d'une variable.

9

Comment connaitre l'adresse mémoire d'une variable ?

10

Utilisez l'opérateur d'adresse &

- En C++, il est possible de connaître l'adresse mémoire de toutes les variables qui ont été déclarées.
  - Peu importe le type
- Pour y parvenir, la syntaxe est simplement :

```
&nom_variable
```

- Regarder le code **adresse.cpp** pour avoir un exemple

11

Comment déclarer et utiliser un pointeur ?

12

Déclarer un pointeur, c'est aussi simple que la déclaration de variable

- La syntaxe de déclaration d'un pointeur est aussi simple que :

```
type *nom_du_pointeur;
type *nom_du_pointeur1, *nom_du_pointeur2;
```

- Remarquez le symbole « * » devant le nom du pointeur
  - C'est ce symbole qui permet la déclaration du pointeur
  - Le symbole « * » doit être placé **devant chaque** variable
- Pour affecter une valeur à un pointeur, nous passons par l'opérateur d'adresse
  - Regarder le code **pointeur.cpp**

13

Travailler sur la valeur pointée

- Grâce aux pointeurs, vous connaissez l'adresse d'une variable
- Si vous connaissez l'adresse d'une variable, vous pouvez modifier la valeur contenue à cette adresse
  - Si vous savez où les biscuits sont rangés, vous pouvez en prendre
- Pour y accéder, nous ajoutons le symbole « * » devant la **variable pointeur**
  - Regarder le code **pointeur.cpp**

14

Lire et écrire une valeur

- Lire la valeur contenue à une adresse pointée

```
Ex : float b = *pointeur;
```

- Écrire une valeur dans une adresse pointée

```
Ex : *pointeur = 6 + 4;
```

15

L'autre avantage, la transmission de paramètre

- Dans le cours sur les fonctions, nous avons vu que les paramètres étaient transmis **par valeur**
  - Chaque variable était copiée
- Les pointeurs permettent un autre type de passage : le passage **par référence**
  - Les variables ne sont plus copiées
  - On transfère la référence vers ces valeurs
    - La référence n'est ni plus ni moins que les pointeurs

16

ATTENTION !

- Le passage par référence a **une conséquence**
- Comme il n'y a plus de copie, toute modification change directement la variable référée

17

Le passage par référence passe par des paramètres de type pointeur

- Afin que vos fonctions acceptent un passage par référence, il faut programmer vos paramètres en conséquence
- Le passage par référence est disponible pour les types pointeurs **seulement**
- Dans l'exemple qui suit, la fonction PuissanceValeur aura un passage **par valeur** et la fonction PuissanceRefence aura un passage **par référence**
  - La différence est que tous les types attendus sont des pointeurs et non des valeurs

18

Exemple de passage

Passage par **valeur**

```
double PuissanceValeur(int a, int b) {
        int Resultat = 1;
        for (int i = 0; i < b; i++) {
                Resultat *= a;
        }
        return Resultat;
}
```

Passage par **référence**

```
double PuissanceValeur(int *a, int *b) {
        int Resultat = 1;
        for (int i = 0; i < *b; i++) {
                Resultat *= *a;
        }
        return Resultat;
}
```

19

La fonction peut modifier son paramètre

- Grâce aux pointeurs, vous pouvez modifier directement une variable sans avoir de retour à effectuer
  - Regardez l'exemple dans **fonctions.cpp**
- Nous verrons d'autres applications avec les tableaux dans le prochain module

20

Petits détails intéressants

21

Qu'est-ce que le type void* ?

- Nous avons que chaque type possède une version pointeur
- Le type pointeur **void*** représente un pointeur général dont le type n'est pas spécifié
  - On ne peut pas utiliser ce type directement
  - Il sert à définition de retour des fonctions
  - Pratique quand le type de pointeur n'est pas connu à l'avance
    - Nous verrons ça dans le prochain module

22

Renvoyez deux valeurs

- Normalement, une fonction ne peut pas renvoyer plus d'une valeur
- Les pointeurs permettent de passer outre cette limite
- Dans l'exemple **renvoie.cpp**, la fonction DecoupeMinute renvoie le nombre d'heures et elle modifie la variable minute afin de renvoyer deux informations

23

L'indentation et les pointeurs

- L'opération d'accès à la valeur (*) a la même priorité que les opérateurs unitaires (**!**, **++**, **--**)
- Leur calcul est réalisé de droite à gauche
- Imaginons les variables `int r = 5; int *p = &r;`
- Il est possible d'incrémenter **r** sans problème avec la notation préfix

  `++*p;`
  - On accède à la valeur contenue à l'Adresse de p et **après** on incrémente (de droite à gauche)

24

L'inverse est faux

- Maintenant, imaginons l'opération suivante

  `*p++;`

  - L'incrémentation est réalisée avant sur le pointeur
  - Après on accède à la case mémoire
- Note : L'incrémentation d'un pointeur correspond à déplacer l'adresse d'un nombre d'octets équivalent à son type
  - Pour un int, ceci équivaut à 4 octets
- La bonne notation est donc : `(*p)++`
  - Les parenthèses ont la plus grande priorité

25

Le pointeur NULL

- Il existe un pointeur spécial
  - Contenu dans `iostream`
- Ce pointeur ne pointe nulle part
- Il s'agit du pointeur **`NULL`**

```
int *p = NULL;
if (p == NULL) cout << "p ne pointe nulle part" << endl;
```

26

Exercice

- Reprenez la fonction Ackermann et modifiez-la pour qu'elle accepte les pointeurs en paramètres
- Reprenez les fonctions récursives du module sur les fonctions et transformez-les pour qu'elle accepte et retourne des pointeurs

27