Série 11: pointeur

Lien avec le MOOC Initiation à la Programmation (en C++)

Exercices semaine7 du MOOC : pointeur et allocation dynamique

- **Document Exercices semaine 7 du MOOC**
    - **Exercice 21 : généricité (niveau 1)**
        - Premier exemple d'application de la notion de pointeur
    - **Exercice 22 : référence (structures, références, niveau 1)**
        - Intérêt et limitation des références

Exercice complémentaire (ExC)

ExC 4 : évaluation de code sans le compiler (niveau 1)

```
#include <iostream>
using namespace std;

int main()
{
    int a(12);
    int b(34);

    int *p1(nullptr);
    int *p2(nullptr);
    int *p3(nullptr);

    p1 = &a;
    p2 = &b;

    p3 = p2;

    *p2 = 15;
    *p1 = (*p1) +  (*p3);
    *p3 = (*p2) + 5;

    cout << a  << " " << b << endl;

    return 0;
}
```

Soit le morceau de programme ci-contre. Les valeurs de **a** et **b** sont elles modifiées pendant l'exécution de ce programme?

si oui quelles sont leurs valeurs finales.

Pour appuyer votre réponse, dessiner les liens entres les variables avec la représentation par boite (pour les variables) et flèche (lorsqu'un pointeur pointe sur une variable).

ExC 5 : Type et valeur d'expressions avec des pointeurs et un tableau à-la-C (niveau 1)

Soient les déclarations suivantes:

```
int a(10), b(2);
int c[3] = { 1, 2, 7 };

int *ptr_1(&a);
int *ptr_2(&b);
int *ptr_3(c); // le nom du tableau c (tableau à-la-C)
               // est un pointeur constant sur son premier élément
```

a) Quelle est la valeur des expressions suivantes (faire un dessin boite/flèche pour illustrer l'état des variables):

- **`*ptr_1 + *ptr_2`**
- **`c [1] == *ptr_2`**
- **`*c + b`**
- **`*(c+b)`**
- **`ptr_1 == ptr_2 ? 2 * *ptr_1 : 3 * *ptr_2;`**

b) Quelle est la valeur des variables a et b après exécution de l'instruction ***ptr_1=*ptr_2 + a;**

c) Quel est le type de l'expression &ptr_1 ?

Même question pour l'expression *ptr_1

d) L'instruction c++; est-elle valide ?

Même question pour l'instruction ptr_3++;

ExC6 : passage d'arguments à la fonction main() et manipulations de chaînes à-la-C (niveau 1)

1) Ecrire un programme exploitant la forme de la fonction main avec les deux paramètres **argc** et **argv** et qui affiche l'ensemble des chaînes reçues au moment du lancement du programme sur la ligne de commande du terminal. Ecrire les chaines les unes à la suite des autres avec un espace comme séparateur.

2) Ce mécanisme de transmission est utilisé par les commandes linux pour indiquer toutes sortes d'options pour ces commandes. Vous allez adopter un style similaire d'option pour offrir un fonctionnement différent de la question 1 :

- si la première chaine ne commence PAS par le caractère '-' alors afficher sur une ligne seulement les chaines qui suivent le nom du programme, dans l'ordre dans lequel elles sont données.

Sinon

- si la première chaine fournie à la suite du nom de l'exécutable est « **-r** »
    - alors afficher les chaines supplémentaires dans l'ordre inverse

- si la première chaine fournie à la suite du nom de l'exécutable est « **-s** »
    - alors afficher les chaines supplémentaires dans l'ordre normal mais une seule par ligne

- si la première chaine fournie à la suite du nom de l'exécutable est « **-rs** » ou « **-sr** »
    - alors prendre en compte les deux options précédentes

- sinon afficher le message « *Incorrect option : accepted options are r for reverse and s for single* »

Comment faire pour accéder à un caractère d'une chaîne transmise à main ?

→ il suffit d'utiliser le pointeur qui donne l'accès à une chaine comme un nom de chaine-à-la-C : on y ajoute un second opérateur [ ] avec une valeur d'indice pour isoler un caractère en particulier.

Exemple1 : soit l'exécutable **prog** qui est lancé sur la ligne de commande avec : **./prog**
La fonction main() reçoit avec **argv[0]** un pointeur sur la chaîne à-la-C qui contient « **./prog** » et qui se termine avec le caractère **\0**.

L'expression argv[0][0] donne le premier caractère de cette chaîne, c'est-à-dire le caractère '**.**' . On peut accéder et tester individuellement chaque caractère de chaque chaîne transmise à main().

Exemple2 :la commande **./prog –r un deux trois**
Produit l'affichage suivant : **trois deux un**

ExC 7 : Réseau d'amis (niveau 2)

Soient les déclarations suivantes définissant une structure Personne qui permet de mémoriser un ensemble de liens vers d'autres structures Personnes pour représenter son groupe d'amis:

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

// prédéclaration nécessaire pour le typedef
struct Personne;

typedef vector<const Personne*> Liste_Personnes;

struct Personne
{
  string nom;
  Liste_Personnes amis;
};
```

a) proposer une fonction **afficher_ami ()** qui admet en paramètres une référence constante d'une Personne **pers** et qui affiche son nom suivi de la liste de ses amis. Par exemple :

```
Les amis de Alice sont :
…..
```

Pour tester cette fonction on peut créer quelques structures de type Personne puis leur ajouter manuellement des amis comme dans cet exemple :

```
Personne alice={"Alice",{}} ;
Personne bob={"Bob",{}} ;
…
alice.amis.push_back(&bob) ;

…
```

b) proposer une fonction **est_ami_avec()** qui admet en paramètres deux références constantes, une vers une Personne **pers** et une string **nom** et qui renvoie le booléen true dès qu'on trouve que la string est le nom d'une Personne appartenant à la liste d'amis de **pers**.

```
bool est_amis_avec(const Personne& pers, const string& nom) ;
```

Tester cette fonction comme à la question précédente.

c) Plutôt que d'ajouter les amis manuellement, proposer une fonction **ajouter_ami ()** qui admet en paramètres deux références, la première vers une Personne **pers** à qui on veut ajouter un ami **new_friend** (seconde référence constante). L'ajout est fait seulement si **new_friend** n'est pas déjà dans la liste d'amis de **pers**.

```
void ajouter_amis(Personne& pers, const Personne& new_friend) ;
```

Tester votre fonction en faisant afficher les amis avant et après l'appel de cette fonction pour une personne pers.

d) Malheureusement, il faut aussi proposer une fonction **enlever_ami ()** qui admet en paramètres deux références, la première vers une Personne **pers** à qui on veut enlever un ami **old_friend** (seconde référence constante). Le retrait est fait seulement si **old_friend** est dans la liste d'amis de **pers**.

```
void enlever_amis(Personne& pers, const Personne& old_friend);
```

La difficulté de cette question provient du fait que la personne à enlever n'est pas forcément la dernière de la liste d'amis. Cette difficulté peut être contournée simplement car il suffit d'échanger la place de **old_friend** avec le dernier élément de la liste d'amis avant d'utiliser la méthode **pop_back()** et le tour est joué.

Tester votre fonction en faisant afficher les amis avant et après l'appel de cette fonction pour une personne pers. Essayer différents scénarios : la personne est dans la liste d'amis ou pas, elle est en dernière position ou pas, etc…