Les pointeurs en C

Introduction

Les adresses mémoire

Nous allons introduire un concept important pour la programmation des systèmes d'exploitation en `C` : Il s'agit des pointeurs.

Les étudiants ont souvent du mal à maîtriser les pointeurs et c'est tout à fait normal : les cours de programmation et d'algorithmique classiques utilisent uniquement des variables 'standards' qui vont contenir des valeurs comme des nombres ou des caractères. Mais pour nos besoins en programmation système, nous allons introduire des variables qui vont contenir des adresses mémoire.

Une variable est stockée en mémoire et du coup elle possède une adresse. Cette adresse est aussi une valeur qui peut être contenue dans une autre variable. Avec ce raisonnement, nous venons de définir ce qu'est un pointeur: c'est une variable qui a pour contenu une adresse mémoire. C'est aussi simple que cela!

L'opérateur de déclaration de pointeur *

Pour déclarer qu'une variable est un pointeur, nous allons utiliser l'opérateur * (étoile)

Dans une déclaration de variable, si le nom de la variable est précédé par *, alors celle-ci est considérée comme un pointeur vers un contenu qui est nécéssairement du type spécifié.

Voici quelques exemples:

```
int  *addresseInt ; // pointeur de int
char *addresseChar ; // pointeur de char
float *addresseFloat ; // pointeur de float
double *addresseDouble ; // pointeur de double
```

L'opérateur & pour obtenir l'adresse

Pour récupérer l'adresse d'une variable en C, vous pouvez utiliser l'opérateur &: placé devant le nom d'une variable, dans une expression, il récupère son adresse. Nous pouvons ensuite utiliser cette adresse pour par exemple la stocker dans une variable pointeur compatible.

Un petit exercice pour se familiariser avec ces notions:

TD2_EXO1

```c
//Includes
#include <stdio.h>
#include <stdlib.h>
//Declarations
int maVariableGlobale = 0;
void test();
//Implementation
int main(){
    test();
}
void test(){
    char maVariableLocal = 'X';
    char *p_c;
    int *p_i;
    p_i = &maVariableGlobale;
    printf(" Globale: %p \n", p_i);
    p_c = &maVariableLocal ;
    printf(" Locale: %p \n", p_c);
    printf(" Taille de char: %ld \n", sizeof(maVariableLocal) ) ;
    printf(" Taille de int: %ld \n", sizeof(maVariableGlobale) ) ;
    printf(" Taille de pointeur de char: %ld \n", sizeof(p_c)) ;
    printf(" Taille de pointeur de int: %ld \n", sizeof(p_i)) ;

}
```

A faire:

- Compilez et exécutez le code proposé
- Analysez les valeurs affichées
- Un entier occupe 4 fois plus de places qu'un caractère. Que peut-on dire sur leurs pointeurs ?

L'opérateur d'indirection *

Nous avons vu que l'adresse d'une variable est obtenue en utilisant l'opérateur &. Maintenant, comment allons-nous faire pour récupérer le contenu à partir d'une adresse ?

Pour cela nous allons utiliser l'opérateur * qui, placé devant un pointeur, récupère le contenu se trouvant à l'adresse en question.

Attention : Il y a un véritable risque de confusion ici : nous avons utilisé * dans deux situations distinctes. Une fois pour déclarer une variable pointeur et cette fois-ci pour récupérer le contenu pointé. Il faut bien voir que même s'il s'agit

du même symbole * nous avons deux opérateurs différents : le premier cherche à spécifier le type du pointeur que nous allons créer; le second, récupère une valeur à partir d'un pointeur existant pour l'utiliser dans une expression.

Exo

```
void permuter(int *adr_i, int *adr_j)
{


}
```

Complétez ce programme pour réaliser un échange entre deux variables de type int.

Pointeurs et tableaux

Un tableau est une séquence de variables rangées dans espace mémoire contigu: c'est à dire sans espace entre deux variables.

En réalité une variable tableau désigne l'adresse de la première variable de la séquence. Ainsi, il est tout à fait possible d'écrire le code suivant:

```
int tab[10] ; // un tableau de 10 cases de int
int *adr_i ;  // declarer une variable qui contient une @ d'un int.

adr_i = tab ; //tab contient l'adresse de la premiere variable de la sequence.
```

Arithmétique des pointeurs

Les pointeurs permettent aussi le déplacement dans les zones mémoire à partir d'un point de départ.

Pour se déplacer, nous allons réaliser des opérations de décalage qui ressemblent à des additions et à des soustractions; mais attention ces opérations ont un sens très spécifique qu'il faut bien comprendre!

```
char *depart;
char *destination;
destination = depart + 1;
```

Dans l'exemple ci-dessus, la variable `destination` contient une adresse qui est à un décalage de `1` unité à partir de l'adresse contenue dans `depart`.

Le mot unité est à retenir ici: l'unité de déplacement dépend du type du pointeur. Dans l'exemple, le pointeur `depart` et de type `char*` donc l'unité de déplacement vaut: `1 * sizeof(char)`

Pour rappel, `sizeof(char)` est égal à 1 octet, donc `destination` désigne une adresse qui se trouve à un décalage d'un octet à partir de `depart`.

Maintenant, observons le code suivant:

```
int *depart;
int *destination;
destination = depart + 1;
```

Où se trouve `destination` par rapport à `depart` ? Et bien, destination se trouve toujours à une unité de déplacement à partir de depart. Seulement l'unité de déplacement maintenant vaut `sizeof(int)` car le pointeur est de type `int*`

Pour rappel, `sizeof(int)` vaut 4 octets, donc `destination` désigne une adresse qui est à un décalage de 4 octets à partir de `depart` !

Exo

```
int length(char* chaine);
```

Donnez le code de cette fonction qui retourne la longueur de la chaîne de caractères. Pour rappel, une chaîne de caractères en C se termine avec le caractère spécial '\0'.

Exo

```
int strcmp(char* chaine1,char* chaine2);
```

Donnez le code de cette fonction qui retourne 1 en cas d'égalité entre les deux chaînes de caractères et 0 sinon.

Exo

```
void strcpy(char* dest,char* src);
```

Donnez le code de cette fonction qui copie le contenu de `src` dans `dest`.

Pointeur de structures

Une structure en C est un assemblage de plusieurs variables de types différents dans un même enregistrement. Vous pouvez imaginer une structure en C comme un objet dans un langage objet mais sans les méthodes.

```
struct _coordonnees2D {
    int positionX ;
    int positionY ;
}
```

Ici nous avons déclaré une structure pour représenter les coordonnées d'un point dans un plan 2D.

Le type de cette structure est `struct _coordonnees2D`. Faites attention, nous avons bien répété le mot clé `struct` pour designer le nouveau type.

Pour ne pas répéter ce mot clé nous pouvons renommer le type avec :

```
typedef struct _coordonnees2D Coordonnees2D
```

Nous pouvons alors utiliser simplement `coordonnees2D` pour désigner le type de cette structure.

Pour déclarer une variable de ce type, nous allons procéder comme pour n'importe quelle autre variable:

```
Coordonnees2D point1;
```

Et nous pouvons aussi déclarer une variable de type adresse sur une structure. Il suffit pour cela d'ajouter l'opérateur * lors de la déclaration :

```
Coordonnees2D *ptr ;
```

Il existe cependant une différence concernant l'accès aux attributs d'une structure entre une variable standard et une variable pointeur. Pour accéder à un attribut à partir d'une variable standard, nous allons utiliser un point `.`

Par exemple:

```
Coordonnees2D point1;

point1.positionX // OK !
```

Par contre, l'accès aux attributs à partir d'un pointeur se fera en utilisant un autre opérateur: la flèche `->`

Par exemple:

```
Coordonnees2D point1;
Coordonnees2D *ptr;

ptr = &point1 ;
ptr->positionX ;// OK !

assert( ptr->positionX == point1.positionX); // un test toujours vrai
```

Pointeurs de fonctions

Une fonction `C` est une suite d'instructions qui seront exécutées lors de l'appel de celle-ci. Concrètement, une fonction écrite en `C` va être transformée en un tableau d'instructions exécutables. Or ce tableau possède une adresse en mémoire et par

conséquence nous pouvons tout à fait ranger cette adresse dans une variable qui deviendra alors un pointeur de fonction.

La déclaration d'un tel pointeur peut parfois surprendre, mais nous allons la comprendre par analogie.

Pour déclarer une variable de type `int`, nous devions écrire cela : `int nom ;`

Et pour déclarer une fonction, nous devions écrire cela: `int mafonction (int i);`

Maintenant, pour déclarer un pointeur sur un entier, nous devions placer l'opérateur de type * devant le nom de la variable: `int *nom_ptr ;`

Et bien, pour un pointeur de fonction, nous allons aussi ajouter l'opérateur de type * devant le nom de variable tout en prenant soin de bien mettre les parenthèses pour éviter toute ambiguïté : `int ( *mafonction_ptr ) (int i);`

Ici nous avons déclaré un pointeur vers une fonction qui prend comme paramètre un int et retourne un int.

Pour bien comprendre l'utilité des parenthèses, observons la situation en cas d'omission :

```
int  * mafonction_ptr  (int i);
```

Ici, il y a une ambiguïté : sommes-nous en train de déclarer une fonction classique qui prend un entier (`int`) et renvoie un pointeur d'entier (`int*`) ou bien sommes nous en train de déclarer un pointeur de fonction qui prend un entier (`int`) et renvoie un entier (`int`)

Pour lever cette ambiguïté, nous allons utiliser les parenthèses en cas de déclaration d'un pointeur de fonction :

```
int  ( *mafonction_ptr ) (int i);
```

Voici un exemple d'utilisation:

```
#include <stdio.h>

int  ( *mafonction_ptr ) (int i);
int incremente(int i) { return i+1 ; }
int decremente(int i) { return i-1 ; }

void main(){
int i = 0;
mafonction_ptr = &incremente ; // utilsation correcte

i = mafonction_ptr( i ); //appel de la fonction a partir du pointeur
printf("i = %d\n",i ); // i = 1
```

```
mafonction_ptr = &decremente ; // utilsation correcte

i = mafonction_ptr( i ); //appel de la fonction a partir du pointeur
printf("i = %d\n",i ); // i = 0
}
}
```

Un dernier point avant de terminer cette section sur les pointeurs de fonctions. Dans le code précédent, nous avons utilisé un seul pointeur de fonction `mafonction_ptr`.

Si nous souhaitons utiliser un autre pointeur sur les fonctions qui ont la même signature, alors nous devrions tout réécrire:

```
int  ( *mafonction_ptr ) (int i);
int  ( *autre_ptr ) (int i);
```

Pour éviter de devoir réécrire toute la signature de la fonction à chaque déclaration d'un pointeur de fonction, nous allons donner un nom à ce type de pointeur pour le réutiliser plus tard.

Pour créer un nouveau nom de type, nous allons utiliser `typedef` :

```
typedef  int  ( *fonction_int_int ) (int i);
```

Il faut faire attention ici: le mot clé `typedef` est très important; Il nous permet de créer un nouveau type nommé `fonction_int_int` et non une nouvelle variable !

Voici un exemple d'utilisation avec création d'un nouveau type.

```
#include <stdio.h>

typedef int  ( *fonction_int_int) (int i); //introduction nouveau type

fonction_int_int mafonction_ptr; //une variable avec le nouveau type
fonction_int_int autre_fonction ; //une autre variable avec le nouveau type

int incremente(int i) { return i+1 ; }
int decremente(int i) { return i-1 ; }
void main(){
int i = 0;
mafonction_ptr = &incremente ; // utilsation correcte
autre_fonction = &decremente ; // utilsation correcte
i = mafonction_ptr( i ); //appel de la fonction a partir du pointeur
printf("i = %d\n",i ); // i = 1
i = autre_fonction( i ); //appel de la fonction a partir du pointeur
printf("i = %d\n",i ); // i = 0
}
```

Travaux Pratiques

Nous allons réaliser une pile LIFO (Last In First Out) en C.

Nous allons ranger dans cette pile des entiers de sorte que le dernier entier inséré sera le premier qui sera retiré en cas de retrait.

Voici un premier type pour gérer les éléments de la pile:

```
struct Element
{
    int data;
    Element *next;
};
typedef struct Element Element;
```

La structure de pile donne simplement le premier élément:

```
struct Stack
{
    Element *head;
};
typedef struct Stack Stack;
```

Voici une fonction qui permet de créer un élément:

```
Element * Element_new(int value)
{
    Element * ptr = (Element *) malloc(sizeof(Element));
    ptr->data = value;
    return ptr;
}
```

Et une fonction qui permet de créer une pile:

```
Stack * Stack_new()
{
    Stack * ptr = (Stack  *) malloc(sizeof(Stack));
    ptr->head = NULL;
    return ptr;
}
```

A faire:

- implémentez la fonction `void Stack_push(Stack *stk, int nb);` qui permet d'insérer un nombre dans la pile

- implémentez la fonction `int Stack_pop(Stack *stk, int* res_ptr);` qui permet de retirer un nombre de la pile pour le mettre dans l'adresse `res_ptr`; cette fonction retourne -1 si la pile est vide et 0 sinon.
- implémentez la fonction `void Stack_print(Stack *stk)` qui affiche le contenu de la pile.
- Donnez des exemples d'utilisation de vos fonctions dans programme exécutable.