Développement C **L2 CUPGE 2018–2019**

Travaux pratiques Types

I Exercice : *Déclarations de variables*

Pré-requis : Thème Entrées/Sorties, exercice 1.

Rappel : Elle est déclarée par son type (qui est le type des valeurs qu'elle peut contenir) et son adresse dans la mémoire. Quelques règles pour l'utilisation de variables :

1. toute variable doit être déclarée dans une fonction ;
2. Le compilateur assure que toute variable déclarée est définie (a une case correspondante dans la mémoire) de manière unique pour chaque utilisation de la fonction ;

   En pratique, on ne fait pas de différence entre une variable et la case dans la mémoire qui lui est associée.

3. une variable est déclarée par un nom et un type :

   Exemple 1 :
   — `int x` déclare une variable de nom `x` et de type `int`
   — `char c` déclare une variable de nom `c` et de type `char`

4. Pour les déclarations complexes, on écrit les opérations qu'il faut faire sur la variable pour obtenir le type au début de la déclaration :

   Exemple 2 : `char * argv[]` signifie que `* argv[i]` (pour un entier `i`) est de type `char` ;
   Donc `argv[i]` est l'adresse d'une case qui contient un `char`
   Donc `argv` est un tableau dont les cases contiennent des adresses de cases de type `char`.

(a) Dans le répertoire ~/workspace/src/Types, ouvrir un nouveau fichier `exercice0.c`, et écrire un programme minimal.

```
#include <stdio.h>
int
main ( int argc , char * argv[ ] )
{
  return 0 ;
}
```

(b) Modifiez le programme pour afficher la valeur de `argc`.

```
#include <stdio.h>
int
main ( int argc , char * argv[ ] )
{
  printf ( "argc vaut %d\n" , argc ) ;
  return 0 ;
}
```

(c) Compilez et exécutez le programme.

(d) Exécutez le programme en mettant des mots et des nombres après le nom du programme (sur la même ligne). Que vaut `argc` ?

`argc` vaut le nombre de mots (séparés par des espaces) qui sont après le nom du programme sur la même ligne.

II Exercice : *Adresses*

Les adresses ne sont pas des entiers, car les opérations possibles sont différentes (on ne peut pas additionner, diviser ou multiplier des adresses), mais elles sont stockées dans la mémoire dans le même format que les entiers. L'adresse d'une variable `x` est obtenue avec `& x`. Une adresse peut être affichée avec la directive `%p`.

(a) Créez un nouveau programme déclarant deux variables `x0` et `x1` de type `int`, et affichez les adresses de ces 2 variables.

```
#include <stdio.h>

int
main ( int argc , char * argv[] )
{
  int x0 ;
  int x1 ;
  printf ( "l'adresse␣de␣x0␣est␣%p.\n" , & x0 ) ;
  printf ( "l'adresse␣de␣x1␣est␣%p.\n" , & x1 ) ;
  return 0 ;
}
```

(b) On peut soustraire une adresse à une autre. Le résultat est un entier long (de type `long int`, avec la directive d'affichage `%ld`). Modifiez le programme précédent pour afficher la différence entre les adresses de `x0` et `x1`.

```
#include <stdio.h>

int
main ( int argc , char * argv[] )
{
  int x0 ;
  int x1 ;
  printf ( "l'adresse␣de␣x0␣est␣%p.\n" , & x0 ) ;
  printf ( "l'adresse␣de␣x1␣est␣%p.\n" , & x1 ) ;
  printf ( "&␣x1␣-␣&␣x0␣=␣%ld\n" , & x1 - & x0 ) ;
  return 0 ;
}
```

Suivant le choix, la différence n'est que 1 ou -1, alors que si on regarde les adresses en hexadécimal, la différence est de 4.

(c) Ajoutez une variable `x2` et calculez les différences.

```
#include <stdio.h>
```

```
int
main ( int argc , char * argv[] )
{
  int x0 ;
  int x1 ;
  int x2 ;
  printf ( "l'adresse de x0 est %p.\n" , & x0 ) ;
  printf ( "l'adresse de x1 est %p.\n" , & x1 ) ;
  printf ( "l'adresse de x2 est %p.\n" , & x2 ) ;
  printf ( "& x1 - & x0 = %ld\n" , & x1 - &x0 ) ;
  printf ( "& x2 - & x1 = %ld\n" , & x2 - & x1 ) ;
  printf ( "& x2 - & x0 = %ld\n" , & x2 - & x0 ) ;
  return 0 ;
}
```

Idem pour la différence des adresses !

(d) Pour la suite, on suppose que les variables sont déclarées dans l'ordre `x0`, `x1`, `x2`. Suit on suit la logique de la question précédente, l'adresse de `x2` est l'adresse de `x0` plus 2, et celle de `x1` est celle de `x0` plus 1. Donc en utilisant `*`, essayez de changer la valeur des variables `x1` et `x2` juste en utilisant l'adresse de `x0` (Il faut initialiser toutes les variables à zéro pour que le compilateur soit satisfait).

(e) Changez la valeur à l'adresse de `x` moins 2 et affichez `argc`.

```
#include <stdio.h>

int
main ( int argc , char * argv[] )
{
  int x0 = 0 ;
  int x1 = 0 ;
  int x2 = 0 ;
  printf ( "l'adresse de x0 est %p.\n" , & x0 ) ;
  printf ( "l'adresse de x1 est %p.\n" , & x1 ) ;
  printf ( "l'adresse de x2 est %p.\n" , & x2 ) ;
  printf ( "l'adresse de argc est %p.\n" , & argc ) ;
  printf ( "& x1 - & x0 = %ld\n" , &y - &x ) ;
  printf ( "& x2 - & x1 = %ld\n" , &z - &y ) ;
  printf ( "& x2 - & x0 = %ld\n" , &z - &x ) ;
  * ( & x0 - 2 ) = 0 ;
  * ( & x0 + 1 ) = 1 ;
  * ( & x0 + 2 ) = 2 ;
  * ( & x0 + 3 ) = 3 ;
  printf ( "x0 = %d, x1 = %d, x2 = %d\n" , x0 , x1 , x2 ) ;
  printf ( "argc = %d\n" , argc ) ;
  return 0 ;
}
```

Morale : quand on fait des opérations sur les adresses, on ne sait pas forcément ce qu'on modifie.

(f) Changez les types de `x0`, `x1`, et `x2` en `char` au lieu d'`int`. Que remarque-t'on ?

De prime abord, c'est n'importe quoi :
— Maintenant, la différence entre les adresses réelles est de 1 (et plus 4) ;
— Mais la soustraction entre les adresses et l'addition d'un entier à une adresse continuent de fonctionner correctement.

L'explication est simple :
— Pour stocker un entier de type `int`, il faut 4 cases de type `char` ;
— L'adresse d'une case est l'adresse de la première case de type `char` qu'elle contient ;
— Donc quand le compilateur met les variables de type `int` les unes après les autres, leur adresse réelle varie de 4 en 4. Et quand ce sont des adresses de type `char`, leur adresse réelle varie de 1 en 1 ;
— Par contre, la soustraction entre adresses renvoie le nombre de cases du même type (de la même taille) entre les 2 adresses, et l'addition renvoie l'adresse de la *i*ème case du type après la case courante.

(g) Pour conclure, `sizeof` calcule le nombre de cases qu'il faut pour stocker une variable. Affichez ce nombre pour les types suivants :

`char`, `short int` , `int`, `long int`, `float`, `double`, `int *`, `char *`

III Exercice : *Tableaux*

Ce qu'on a vu lors de l'exercice précédent n'est pas "normal'' : on a utilisé notre connaissance du fonctionnement de gcc pour savoir quelle était l'adresse d'autres variables à partir d'une adresse connue. La méthode *standard* pour déclarer plusieurs variables du même type en C est de déclarer un tableau.

(a) Dans un nouveau fichier `exercice.c` et :

1. remplacer les 3 déclarations de variables par :

```
int x[3] ;
```

2. remplacer partout `x0` par `x[0]`, `x1` par `x[1]`, et `x2` par `x[2]`.

Compilez et exécuter le programme.

On n'a pas initialisé les variables du tableau, donc x[0] a une valeur quelconque, mais pour le reste, on obtient exactement les mêmes résultats qu'avant.

La déclaration `int x[3]` déclare une suite (un tableau) de 3 variables `x[0]`, `x[1]`, et `x[2]`.
`x` n'est pas une variable !

(b) Affichez la valeur de `x` en utilisant la directive des adresses. Que remarque-t'on ?
La *valeur* de `x` est l'*adresse* de `x[0]`.

(c) Affichez l'adresse de `x`. Affichez les tailles de `x` et de `& x`. Que remarque-t'on ?

```
#include <stdio.h>

int
main ( int argc , char * argv[] )
{
  int x[3] ;
```

```
  printf ( "l'adresse de x[0] est %p.\n" , & x[0] ) ;
  printf ( "l'adresse de x[1] est %p.\n" , & x[1] ) ;
  printf ( "l'adresse de x[2] est %p.\n" , & x[2] ) ;
  printf ( "la valeur de x est %p.\n" , x ) ;
  printf ( "l'adresse de x est %p.\n" , & x ) ;
  printf ( "& x[1] - & x[0] = %ld\n" , & x[1] - & x[0] ) ;
  printf ( "& x[2] - & x[1] = %ld\n" , & x[2] - & x[1] ) ;
  printf ( "& x[2] - & x[0 = %ld\n" , & x[2] - & x[0] ) ;
  * ( & x[0] - 2 ) = 0 ;
  * ( & x[0] + 1 ) = 1 ;
  * ( & x[0] + 2 ) = 2 ;
  * ( & x[0] + 3 ) = 3 ;
  printf ( "x[0] = %d, x[1] = %d, x[2] = %d\n" , x[0] , x[1] , x[2] ) ;
  printf ( "sizeof ( x ) = %ld\n" , sizeof ( x ) ) ;
  printf ( "sizeof ( & x ) = %ld\n" , sizeof ( & x ) ) ;
  return 0 ;
}
```

C'est la même adresse que celle de x[0]. Tout se passe comme si on avait déclaré une très grande case pouvant contenir 3 entier :

- La valeur de `x` est l'adresse de la première case `x[0]` du tableau ;
- La taille de `x` est la somme des tailles des cases ;
- La valeur de `& x` est l'adresse du début de cette grande case ;
- La taille de `& x` est la taille normale des adresses.

Les tableaux ont les propriétés suivantes :

- `x[i] = * ( x + i )`
- `& x[i] = & * ( x + i ) = x + i`

(d) Vérifiez ces égalités en simplifiant le programme, et en vérifiant que les résultats sont les mêmes.

```
#include <stdio.h>

int
main ( int argc , char * argv[] )
{
  int x[3] ;
  printf ( "l'adresse de x[0] est %p.\n" , x ) ;
  printf ( "l'adresse de x[1] est %p.\n" , x + 1 ) ;
  printf ( "l'adresse de x[2] est %p.\n" , x + 2 ) ;
  printf ( "la valeur de x est %p.\n" , x ) ;
  printf ( "l'adresse de x est %p.\n" , & x ) ;
  printf ( "& x[1] - & x[0] = %ld\n" , ( x + 1 ) - ( x + 0 ) ) ; // = 1
  printf ( "& x[2] - & x[1] = %ld\n" , ( x + 2 ) - ( x + 1 ) ) ; // =1
  printf ( "& x[2] - & x[0 = %ld\n" , ( x + 2 ) - ( x + 0 ) ) ; // = 2
  x[-2] = 0 ;
  x[0] = 1 ;
  x[1] = 2 ;
  x[2] = 3 ;
  printf ( "x[0] = %d, x[1] = %d, x[2] = %d\n" , x[0] , x[1] , x[2] ) ;
  printf ( "sizeof ( x ) = %ld\n" , sizeof ( x ) ) ;
```

```
    printf ( "sizeof␣(␣&␣x␣)␣=␣%ld\n" , sizeof ( & x ) ) ;
    return 0 ;
}
```