Développement C **L2 CUPGE 2020–2021**

Travaux pratiques Types

1 Cours

Organisation de la mémoire. On va créer des programmes qui traitent des données. Pour chaque programme :

- Les données sont conservées avec des 0 et des 1 dans la *mémoire* (et ce sont des *bits*) ;
- La mémoire accessible est organisée en *cases atomiques* qui ont chacune une adresse entre 0 et $2^{31} - 1$ ou $2^{63} - 1$;
- Les cases atomiques peuvent être regroupées en cases plus grandes, dont l'adresse est l'adresse de la première case atomique.

Cela sera vu en détail en cours. Le but en TP n'est pas d'aller vite, mais de commencer à comprendre comment ça marche.

Types. Un *type* en C est défini par le nombre de cases atomiques qu'il faut pour stocker les valeurs de ce type et les opérations que le processeur peut faire sur les valeurs de ce type.

Exemples :

- Le type `char` est celui des entiers qui peuvent être stockés sur une seule case atomique (et chaque case atomique contient 8 bits) ;
- Sur les ordinateurs courants, le type `int` est celui des entiers qui peuvent être stockés sur 4 cases atomiques (32 bits) ;
- Sur tous les ordinateurs, le type `float` est celui des nombres à virgule qui peuvent être stockés sur 4 cases atomiques (32 bits) ;
- Attention, pour le processeur, la multiplication de 2 entiers n'a rien à voir avec la multiplication de 2 nombres à virgule (le nombre de cases ne suffit pas).

Valeurs. Une *valeur* est une suite de bits qui a un type. On peut changer le type d'une valeur en mettant le nouveau type entre parenthèses devant.

Exemples :

- `'a'`, `'3'`, 35, -128 sont des valeurs de type `char`, qui sont des entiers entre -128 et 127 (la valeur de `'a'` est 97 et celle de `'3'` est 51 dans le code ASCII)
- $1.0/3.0 = 0.33\ldots$, mais $1/3 = 0$ (la division sur des entiers est comme la division euclidienne)
- `( float ) 'a'` est le nombre à virgule 51.0 ;
- `( int ) 51.01` et `( int ) 51.99` sont tous les deux l'entier 51 (arrondi à l'entier inférieur).

Variables. Une *variable* est une case d'un certain type. La *valeur* d'une variable est la suite de bits qu'elle contient avec le même type que celui de la variable.

— Les programmes qu'on écrit en C sont *compilés* : un programme traduit ce qu'on écrit en instructions pour le processeur ;
— C'est ce programme qui choisira l'adresse où les variables seront stockées ;
— Le programmeur doit juste donner un nom (avec un type) pour prévenir qu'il a besoin d'une variable.

I Exercice : *Adresses*

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

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.

(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`.

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

(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 `x0` moins 2 ou `x0` moins 4, et affichez `argc`.

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

(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 *`

II 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 `exercice2.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.

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 ?

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

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.

III Exercice : *Allocation automatique de mémoire*

Pour les types de base et pour les utilisations normales, le compilateur réserve suffisament de mémoire auprès du système d'exploitation pour que toutes les variables qu'on déclare puissent être stockées. Autrement dit, on n'a pas besoin, en tant que programmeur, de demander explicitement de la mémoire au système d'exploitation. Le but de cet exercice est de trouver cette limite. Attention, à cause de tout ce qui est ajouté lors de la compilation, la limite trouvée dépend du programme exact.

(a) Tapez et compilez le programme suivant :

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

int main ( int argc , char * argv[] )
{
        int n ;
        n = atoi ( argv[1] ) ;
        char t[n] ;
        t[0] = 0 ;
        return 0 ;
}
```

(b) Exécutez-le en lui donnant un entier en argument.

Lorsque l'argument est petit (moins de 1000000), le programme ne fait rien. Lorsqu'il est grand (plus de 10000000), le programme fait `segmentation fault` *car on utilise un tableau qui ne tient plus dans la mémoire allouée automatiquement.*

(c) Trouvez le nombre exact pour votre programme.

IV Exercice : *Affichage d'un code*

Le but de cet exercice est de créer un programme `../bin/codage` qu'on peut utiliser pour afficher le code d'un `float` tel qu'on le calcul, et le vrai. Par exemple :

Code bash 1 – Terminal

```
../bin/codage 3.14159
01000000010010010000111111001111
01000000010010010000111111010000
```

Pour cela, on utilise un tableau de 32 `char` qui vont stocker les bits, et calculer les bits les uns après les autres. Toutes les fonctions utiliseront celle de la première question, qui remplit k cases d'un tableau de char t avec les bits (en partant du plus grand) d'un entier n.

(a) Écrivez la fonction :

```
int
int_to_bits (
              unsigned int n ,
              char * bits ,
              int longueur ) ;
```

On veut, lorsque la longueur est 4 :

- si $n = 11 = (1011)_2$, alors la fonction met 1 dans la case à l'adresse *bit*, 0 à l'adresse *bit*+1, 1 à l'adresse *bit*+2, et 1 à l'adresse *bit*+3. Comme il n'y a plus de chiffres dans n, la fonction renvoie la valeur 0 ;
- si $n = 22 = (10110)_2$, alors la fonction met 1 dans la case à l'adresse *bit*, 0 à l'adresse *bit*+1, 1 à l'adresse *bit*+2, et 1 à l'adresse *bit*+3. Comme il reste un chiffre dans n, la fonction renvoie une valeur différente de 0 ;

Pour savoir s'il faut mettre un bit à 0 ou à 1, on remarque que si $n < 2 \times 2^i$, alors le $i$ème bit en partant est à 1 si et seulement si $n \geq 2^i$. En reprenant l'exemple de 22 :

- on commence par calculer la puissance de 2, ici 16, qui satisfait $16 \geq 22 < 32$;
- En partant de 16 :
  - 16 est inférieur ou égal à 22, donc on met le premier bit à 1 ;
  - $8 = 16/2$ est strictement supérieur à $6 = 22 - 16$, donc on met le bit suivant à 0 ;
  - $4 = 8/2$ est inférieur ou égal à 6, donc on met le bit suivant à 1 ;
  - $2 = 4/2$ est inférieur ou égal à $2 = 6 - 4$, donc on met le bit suivant à 1 ;

  On a 4 bits, donc on s'arrête. La puissance n'est pas 1, donc il reste des chiffres.

(b) Écrivez une fonction :

```
int
calcule_signe ( double * f , int * s ) ;
```

qui met à l'adresse s 1 si le `double` à l'adresse f est négatif, et 0 sinon. La fonction doit aussi changer le `double` à l'adresse f pour qu'il soit maintenant positif.

(c) Écrivez une fonction :

```
int
calcule_signe ( double * f , int * e ) ;
```

qui met à l'adresse e l'exposant biaisé, calculé avec des multiplications ou des divisions successives, du `double` à l'adresse f. Si ce `double` n'est pas positif, la fonction renvoie 1. Si ce `double` est 0, la fonction met 0 à l'adresse e. À la fin de la fonction, le `double` à l'adresse f doit

être dans l'intervalle $[1, 2[$. Si l'exposant biaisé est inférieur à 0 ou supérieur à 255, le nombre ne peut pas être codé en float, et la fonction doit renvoyer 1. S'il n'y a pas eu d'erreurs, elle renvoie 0.

(d) Écrivez une fonction :

```
int
calcule_mantisse ( double * f , int * m ) ;
```

qui calcule la mantisse de f en faisant des multiplications par 16 (suffisament), et met le résultat dans m. Cette fonction suppose que le `double` à l'adresse f est entre 1 et 2 (n'oubliez pas d'enlever 1 !).

(e) On peut maintenant écrire la fonction `main`. L'argument est dans `argv[1]`. Il faut le transformer en `double` avec la fonction `atof`, et calculer le triplet (s, e, m) de son codage en `float`. Cette fonction déclare aussi un tableau *bits* de 32 `char`. Utilisez la fonction `int_to_bits` pour remplir ce tableau à partir du triplet. On peut aussi obtenir plus directement le codage d'un `float` dans la variable f avec :

```
int_to_bits ( * ( unsigned int * ) & f , bits , 32 ) ;
```