Programmation impérative en C

Cours 3. Codage et mémoire

Olivier Baldellon

Courriel : prénom.nom@univ-cotedazur.fr

Page professionnelle : https://upinfo.univ-cotedazur.fr/~obaldellon/

Licence 2 — Faculté des sciences et ingénierie de Nice — Université Côte d'Azur

Sommaire

La fonction lire entier

- Comment lire un entier proprement.

*.c
```c
int lire_entier() {
    int n = 0;
    char c;
    printf("entier> ");
    while ( (c = getchar()) != '\n' && c != EOF) {
        if ('0' <= c && c <= '9') {
            n = n*10 + (c-'0');
        } else {
          while ( (c = getchar()) != '\n' && c != EOF);
          return -1;
        }
    }
    return n;
}
```

- Remarques : `scanf` rencontre exactement les mêmes problèmes.
- On peut définir une petit fonction auxilliaire.

*.c
```c
void vider_buffer() {
  while ( (c = getchar()) != '\n' && c != EOF);;
}
```

La mémoire est en accès libre

*.c

```c
void f(void) {
    int tab[] = {0};
    int i = 0;;
    while (tab[i] != 777) {
        printf("t[%.2d] = %i\n",
                        i, tab[i]);
        i++;
    }
    printf("t[%.2d] = %i\n",
                    i, tab[i]);
    tab[i]=666;
}

void fonction(void) {
    f();
}

int main() {
    int variable = 777;
    fonction();
    printf("%d\n", variable);
    return 0;
}
```

- En C, chaque fonction peut accéder à-peu-près toute la mémoire allouée.

stdout

```
t[00] = 0
t[01] = 1
t[02] = 671837296
t[03] = 32764
t[04] = -1972874112
t[05] = 22038
t[06] = 671837328
t[07] = 32764
t[08] = -1972874089
t[09] = 22038
t[10] = 671837568
t[11] = 32764
t[12] = 0
t[13] = 777
666
```

- *Undefined Behavior*
- On peut même la modifier ! (dangereux !)

Afficher en binaire

- On range les ordis et on sort une feuille INTERRO SURPRISE ! ! !
- Écrire une fonction qui affiche sur une ligne la représentation binaire d'un octet.
  - interdiction d'utiliser le modulo
  - ou la division entière par deux
  - Les bits doivent être affichés dans le bon ordre.
- Exemple :
  - pour 2 on affichera : `00000010`
  - pour 131 on affichera : `10000011` (131 = 128 + 2 + 1)

- Pour nous aider à comprendre la mémoire, on va écrire une fonction affichant proprement 8 octets successifs en mémoire.
  - en décimal
  - en héxa
  - en binaire

- Affichage voulu :

*.c

```
+--------+--------+--------+--------+--------+--------+--------+--------+
|   85   |    2   |   10   |    0   |    0   |    0   |    0   |    0   |
|   55   |   02   |   0A   |   00   |   00   |   00   |   00   |   00   |
|01010101|00000010|00001010|00000000|00000000|00000000|00000000|00000000|
+--------+--------+--------+--------+--------+--------+--------+--------+
```

- À faire chez vous en exercice

Sommaire

- Comment savoir ce qu'écrit C lorsqu'on stocke une donnée en mémoire ?

*.c
```c
typedef struct { char r, g, b;} rgb;

typedef struct { char c;
                 short s; } alignement1;

typedef struct { unsigned char petit1;
                 unsigned short grand;
                 unsigned char petit2; } alignement2;

union mon_type {
  long long_s; unsigned long long_u;
  int  int_s;  unsigned int  int_u;
  float float_t;

  alignement1 alignement1_t;
  alignement2 alignement2_t;
  unsigned char tableau[8];
};
```

- Méthode
  - on stocke un type en mémoire : par exemple `x.long_s = 17`
  - on affiche le tableau `x.tableau` pour voir ce qui est codé.

- Comment est stocké un entier ?

*.c
```
x.long_s = 655957
+--------+--------+--------+--------+--------+--------+--------+--------+
|   85   |    2   |   10   |    0   |    0   |    0   |    0   |    0   |
|   55   |   02   |   0A   |   00   |   00   |   00   |   00   |   00   |
|01010101|00000010|00001010|00000000|00000000|00000000|00000000|00000000|
+--------+--------+--------+--------+--------+--------+--------+--------+
```

- Pas facile de comprendre ce qu'il se passe...
- Pour comprendre, choisissons un entier en héxa.

*.c
```
x.long_s = 0xCAFE0230
+--------+--------+--------+--------+--------+--------+--------+--------+
|   48   |    2   |  254   |  202   |    0   |    0   |    0   |    0   |
|   30   |   02   |   FE   |   CA   |   00   |   00   |   00   |   00   |
|00110000|00000010|11111110|11001010|00000000|00000000|00000000|00000000|
+--------+--------+--------+--------+--------+--------+--------+--------+
```

- Comment écrire `0x1234CAFE` (en hexa) dans la mémoire ?
  - Gros-Boutisme : dans le bon ordre (naturel pour nous autres humains)
  - Gros-Boutisme : on commence par les bits de poids fort
  - Petit-Boutisme : à priori ce que votre machine utilise (ARM, x86, x86-64)
  - Petit-Boutisme : on commence par les bits de poids faibles

Le Gros-Boutisme *big-endian*

Le Petit-Boutisme *little-endian*

- Intérêt de petit boutisme ?
  - L'entier 17 est codé pareil quelque soit son type : `int`, `short`, etc.
  - Plus rapide pour les calculs (qui commencent souvent avec les bits de poids faibles)

- Ne pas tenter de lire un `int` dans la mémoire octet par octet
  - sauf pour la curiosité : risqué et non portable
  - les opérations bits à bits marchent indépendemment du boutisme.
  - utilisez-les si besoin de décomposer un nombre (tp2).

Nombres négatifs

*.c

```
x.int_s = 1
+--------+--------+--------+--------+--------+--------+--------+--------+
|    1   |    0   |    0   |    0   |    0   |    0   |    0   |    0   |
|   01   |   00   |   00   |   00   |   00   |   00   |   00   |   00   |
|00000001|00000000|00000000|00000000|00000000|00000000|00000000|00000000|
+--------+--------+--------+--------+--------+--------+--------+--------+

x.long_s = 1
+--------+--------+--------+--------+--------+--------+--------+--------+
|    1   |    0   |    0   |    0   |    0   |    0   |    0   |    0   |
|   01   |   00   |   00   |   00   |   00   |   00   |   00   |   00   |
|00000001|00000000|00000000|00000000|00000000|00000000|00000000|00000000|
+--------+--------+--------+--------+--------+--------+--------+--------+

x.int_s = -1 ou x.int_u = 4294967295
+--------+--------+--------+--------+--------+--------+--------+--------+
|  255   |  255   |  255   |  255   |    0   |    0   |    0   |    0   |
|   FF   |   FF   |   FF   |   FF   |   00   |   00   |   00   |   00   |
|11111111|11111111|11111111|11111111|00000000|00000000|00000000|00000000|
+--------+--------+--------+--------+--------+--------+--------+--------+

x.long_u = -1
+--------+--------+--------+--------+--------+--------+--------+--------+
|  255   |  255   |  255   |  255   |  255   |  255   |  255   |  255   |
|   FF   |   FF   |   FF   |   FF   |   FF   |   FF   |   FF   |   FF   |
|11111111|11111111|11111111|11111111|11111111|11111111|11111111|11111111|
+--------+--------+--------+--------+--------+--------+--------+--------+
```

- Comment sont codé les flottants ? Prenons l'exemple du nombre 5.

*.c
```
x.float_t = 5.0;
+--------+--------+--------+--------+--------+--------+--------+--------+
|    0   |    0   |  160   |   64   |    0   |    0   |    0   |    0   |
|   00   |   00   |   A0   |   40   |   00   |   00   |   00   |   00   |
|00000000|00000000|10100000|01000000|00000000|00000000|00000000|00000000|
+--------+--------+--------+--------+--------+--------+--------+--------+
```

- Quel est le rapport avec le nombre 5 ? (car oui, il y en a un !)
  - Conservons seulement 32 bits et remettons les dans l'ordre.
  - 01000000 10100000 00000000 00000000
    - 1 bit pour le signe : (0 pour +1 et 1 pour −1)
    - 8 bits pour l'exposant biaisé : ici 1000000 1 donne 129.
    - 24 bits pour la mantisse : ici 0100000 00000000 00000000 ce qui donne 2097152.

- Bref, rien de bien compliqué...
  - des questions ?

- Comment obtenir 5 à partir de ces trois valeurs.

$$signe \times \left(1 + \frac{mantisse}{2^{23}}\right) \times 2^{exposant - 127}$$

- De même que :
  - en décimal $524 = 5,24 \times 10^2$
  - en binaire cinq vaut $\overline{101} = \overline{1,01} \times 2^2$

$$5 = +1 \times \overline{1,01} \times 2^{129-127}$$

- Pourquoi $1 + mantisse \times 2^{-23}$?
  - En binaire le premier chiffre vaut toujours 1 : inutile de le stocker.
  - On conserve les 23 premiers bits après la virgule : c'est la mantisse.
  - $1 + mantisse \times 2^{-23} = 1.\texttt{0100000 00000000 00000000}$

- C'est la norme IEEE 754 utilisée dans presque tous les langages.

*.c
```
typedef struct{ char r, g, b;} rgb;

typedef struct  {
    char c;
    short s;
} alignement1;
```

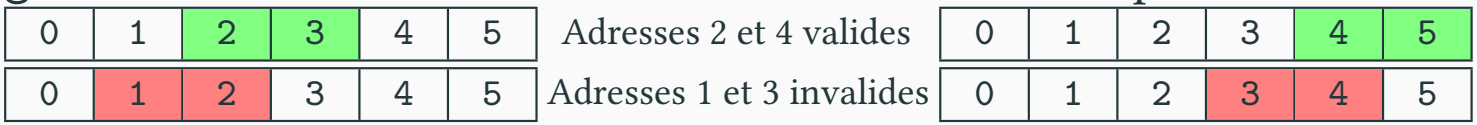
- Les structures sont codées comme des tableaux

*.c
```
x.rgb_t : r=11 g=12 et b=13
+--------+--------+--------+--------+--------+--------+--------+--------+
|   11   |   12   |   13   |    0   |    0   |    0   |    0   |    0   |
|   0B   |   0C   |   0D   |   00   |   00   |   00   |   00   |   00   |
|00001011|00001100|00001101|00000000|00000000|00000000|00000000|00000000|
+--------+--------+--------+--------+--------+--------+--------+--------+
```

- Avec des cases inutilisées parfois ! Pourquoi ?

*.c
```
x.alignement1_t  c=5 et s=0x0706
+--------+--------+--------+--------+--------+--------+--------+--------+
|    5   |    0   |    6   |    7   |    0   |    0   |    0   |    0   |
|   05   |   00   |   06   |   07   |   00   |   00   |   00   |   00   |
|00000101|00000000|00000110|00000111|00000000|00000000|00000000|00000000|
+--------+--------+--------+--------+--------+--------+--------+--------+
```

*.c
```
typedef struct { unsigned char petit1;
                 unsigned short grand;
                 unsigned char petit2; } alignement2;
```

- Un champs de longueur n doit être sur une adresse multiple de n
  - `grand` est un `short` sur deux octets : l'adresse doit être paire.
  - 

- La taille de la structure doit être un multiple de la taille de ces champs.
  - La structure ne peut pas être de taille 5
  - Le compilateur rajoute une case après (et y met n'importe quoi : 77)

*.c
```
x.alignement2_t petit1=0xFF grand=0xCAFE petit2=0xAA
+--------+--------+--------+--------+--------+--------+--------+--------+
|  255   |    0   |  254   |  202   |  170   |   77   |    0   |    0   |
|   FF   |   00   |   FE   |   CA   |   AA   |   4D   |   00   |   00   |
|11111111|00000000|11111110|11001010|10101010|01001101|00000000|00000000|
+--------+--------+--------+--------+--------+--------+--------+--------+
```

- Le compilateur fait ce travail pour vous, mais comment connaître la taille ?

- Pour connaître la taille des données en mémoire on utilise `sizeof`
  - l'unité est le *byte*, donc concrètement l'octet.
- On peut l'utiliser avec un type : `sizeof(int)`
- On peut l'utiliser avec une variable : `sizeof(variable)`

*.c

```c
typedef struct{
    unsigned char petit1;
    unsigned short grand;
    unsigned char petit2;
} alignement2;

int main() {
    ma_struct s;
    /* Affiche 1 2 1 : logique */
    printf("%ld %ld %ld\n",sizeof(s.petit1)
                          ,sizeof(s.grand)
                          ,sizeof(s.petit2));

    /* Affiche 6 : Pourquoi ? */
    printf("%ld\n", sizeof(s));
    printf("%ld\n", sizeof(alignement2));
    return 0;
}
```

Merci pour votre attention
Questions

?

Cours 3 — Codage et mémoire