Programmation C - Bases (1)

Laurent Signac – CC-BY-SA – 05-12-22 1451 7b38e63f42777e0ca08f

1 Instructions

On a trois types d'instructions :

- instructions simples `a=5;`
- instructions de structuration :
  `if (a==5) printf("Voilà"); else {printf("Eh non...");}`
- blocs (délimités pas des accolades) `{a=5; b=4;}`

La notion d'instruction est différente de la notion d'expression.

- Une instruction *fait* quelque chose alors qu'une expression *vaut* quelquechose
- En C, les instructions sont souvent aussi des expressions : L'instruction simple `a=5;` fait quelque chose (elle affecte `5` à `a`), mais elle vaut aussi quelque chose, la valeur affectée (5). C'est pourquoi on peut écrire `b=a=5` (lire `b=(a=5)`).

2 Types de données scalaires

En C, on déclare les variables avant leur utilisation (contrairement à ce qu'on fait en Python). Le type d'une variable ne peut ensuite plus changer.

2.1 Entiers

Il y a plusieurs longueurs de codage (`int`, `short`, `long`, `long long`) et les entiers peuvent être signés ou non (`signed`, `unsigned`).

```c
#include<stdio.h>
int main(void) {
    printf("%ld %ld %ld %ld\n",sizeof(short),
            sizeof(int),sizeof(long),sizeof(long long));
    return 0;
  }
```

Sur la machine de test, le programme précédent affiche par exemple `2 4 8 8`. On peut consulter les tailles possibles des entiers sur la page Wikibook/Programmation C/Types de base[1]

Les constantes littérales entières peuvent être données dans trois bases : décimal `43`, hexadécimal `0x2B`, binaire `0b101011`.

2.2 Nombres à virgule flottante

Deux précisions disponibles : `float` et `double` (32 ou 64 bits). Les `double` du C sont exactement les `float` de Python. Les constantes littérales peuvent être entrée de plusieurs façons : `2.45e-5`, `0.00023`, `42.0`,`42.` (mais pas `42`).

2.3 Caractères

Un caractère est codé sur un octet. Il y a équivalence entre un caractère et son code Ascii. C'est la manière d'afficher (`%i` ou `%c`) qui fera apparaître un nombre ou un caractère. Les constantes littérales peuvent être entrées de plusieurs façons :

- Caractère entre guillemets simples : `'A'` ou `'b'` (uniquement des guillemets simples, qui sont différents des guillemets doubles, contrairement à ce qui se passe avec Python)

[1] http://fr.wikibooks.org/wiki/Programmation_C/Types_de_base

- Caractères spéciaux `'\n'` ou code Ascii `'\65'` (en octal !) éventuellement en hexadécimal `'\x41'`

3 Conversions, casts

3.1 Cast

L'opérateur de *cast* permet de convertir un type en un autre : `(type) var` convertit la variable `var` dans le type `type`.

```
int a;
a=(int) 5.6;
```

3.2 Conversions implicites

Certaines conversions sont automatiques, notamment dans les opérations arithmétiques, où toutes les opérandes doivent être de même type. Les conversions s'opèrent alors dans ce sens :

`int` → `long` → `float` → `double`

Des conversions sont aussi réalisées implicitement durant les affectations :

```
int a = 7.5;
```

4 Opérateurs

4.1 Opérateurs arithmétiques

`+`	addition
`-`	soustraction
`*`	multiplication
`/`	division
`%`	reste

4.2 Opérateurs sur les bits

`&`	et
`|`	ou inclusif
`^`	ou exclusif
`<<`	décalage à gauche
`>>`	décalage à droite
`~`	complement à 1

Exemples :

- `12 & 5` donne 4 (12 s'écrit 1100 et 5 s'écrit 0101, la somme des bits communs vaut 4).
- `12 | 5` vaut 13 (bit 0, 2 et 3 à 1 dans l'un ou l'autre des nombres)
- `12 ^ 5` vaut 9 (bit 0 et 3 à 1 dans **un seul** des deux nombres)
- `12 << 1` vaut 24 (décalage des bits à gauche ajout d'un 0 à droite revient à multiplier par 2)
- `12 >> 2` vaut 3 (décalage de 2 range à droie revient à diviser par 4)

Les opérateurs sur les bits sont souvent utilisés comme masques. Lors de la programmation sur microcontrôleurs, on a parfois besoin de manipuler, plutôt que des nombres, des bits précis.

Par exemple, régler la broche 2 du port 6 en sortie ⇔ mettre à 1 le bit 2 à une adresse particulière (port 6).

Les valeurs numériques, adresses, variables, sont définies dans des fichiers en-tête.

Sur un MSP 430, le port 6 est à l'adresse `0x36`, associée à la variable `P6DIR`, définie dans un fichier d'en-tête.

Régler la broche 2 du port 6 en sortie ⇔ `P6DIR = P6DIR | 0b00000100`

On trouve aussi comme écritures :

```
P6DIR = P6DIR | 4
P6DIR = P6DIR | 0x04
P6DIR = P6DIR | BIT2
```

4.3 Affectation élargie

Il n'est pas utile d'utiliser les opérateurs d'affectation élargis dès le début, on peut s'en passer. Mais vous allez très certainement rapidement les rencontrer. Le tableau en contient certains.

`i++`	`i=i+1`
`i--`	`i=i-1`
`i+=b`	`i=i+b`
`i-=b`	`i=i-b`
`i*=b`	`i=i*b`
…	…

5 Entrées sorties

- Nécessité dès les premiers progs (débuggage, affichage des résultats), car on n'a pas de shell ou d'invite de commande.
- Affichage avec la fonction `printf`.

```
float a;
char c;
int i;
printf("Bonjour à tous\n");
printf("Bonjour, vive %i\n",42);
a=42.;
i=42;
printf("Attention, %i n'est pas %f\n",i,a);
printf("Un petit 42 scientifique : %e\n",a);
c='\x42';
printf("Mais 42, c'est aussi (en héxa) %x, ou %c\n",c,c);
```

5.1 Format usuels

Format	Signification
`%i`	entier signé
`%u`	entier non signé
`%d`	entier décimal
`%ld`	entier long décimal
`%lu`	entier long non signé
`%09d`	entier décimal sur 9 chiffres complété par des 0 à gauche
`%f`	nombre à virgule flottante
`%.3f`	nombre à virgule avec 3 chiffres après la virgule
`%e`	nombre à virgule flottante en notation scientifique
`%s`	chaîne de caractères (délimitée par un espace)
`%c`	caractère

Format	Signification
%x	nombre hexadécimal

5.2 Entrées `scanf`

- `scanf` marque une pause
- L'entrée est interprétée selon un certain format

```
int n;
float v;
printf("Entrez un nombre entier puis un nombre à virgule ");
scanf("%i%f",&n,&v);
printf("Vous avez entré %i et %f\n",n,v);
```

La fonction `scanf` ne permet pas de saisir simplement une chaîne de caractères contenant des espaces (voir `fgets`).

5.3 Fonction `fgets`

```
char texte[256];
printf("Entrez une phrase (255 caractères max)\n");
fgets(texte,256,stdin);
printf("Merci d'avoir dit :\n%s\n",texte);
```

6 Retour sur les instructions et les expressions

- les instructions du C sont aussi des expressions (elles ont une valeur)
- `i = 5` **vaut** 5
- Affectation élargie : à la fois instruction et expression. Attention `++i` est différent de `i++` au niveau *expression*. Dans le premier (resp. second) cas l'expression vaut la valeur de `i` après (resp. avant) l'incrémentation.

  ```
  i=5;
  j=i++;
  printf("%d %d\n",i,j);
  ```

- La `,` sépare une instruction simple en expressions. La valeur prise par l'instruction est celle de la dernière expression.

Exemples à décortiquer :

```
int i, j, k;
i = j = 0;
k = ++i, --j;
printf("%d %d %d\n", i, j, k);
```

et

```
int i, j, k;
i = j = 0;
k = (++i, --j);
printf("%d %d %d\n", i, j, k);
```

6.1 Instruction

- Instructions simples :
  - `a = 3;` ou `i = collatz(27);`
  - se terminent nécessairement pas un `;`(une instruction simple peut être composée, et les morceaux qui la composent sont séparés par des `,` comme dans: `i = 4, b = 3*i+1;`
- Instructions de structuration (tests, boucles…) : contiennent souvent un ou plusieurs blocs
- Blocs, délimités par des accolades, pouvant contenir des instructions simples, des blocs et des instructions structurées (il peut aussi être vide…).

Plus c'est simple, mieux c'est : un programmeur, amateur ou non, n'a **jamais intérêt** à écrire un code compliqué.

7 Tests

Dans les tests on utilise des opérateurs de comparaison et des connecteurs logiques. En voici la liste.

7.1 Opérateurs de comparaison

`>`	strictement supérieur
`<`	strictement inférieur
`>=`	supérieur ou égal
`<=`	inférieur ou égal
`==`	égal
`!=`	différent

Le résultat d'un opérateur de comparaison est 0 (faux) ou 1 (vrai). Ce résultat peut être utilisé dans un calcul.

7.2 Opérateurs logiques

`&&`	et
`||`	ou
`!`	non

Les connecteurs logiques manipulent les valeurs 0 et 1

Sans surprise, les instructions qui permettent de faire des tests sont `if` ou `if/else`.

```
if (condition)
    instruction
```

```
if (condition)
    instruction 1
else
    instruction 2
```

Exemple :

```
if (c % 3 == 0)
  printf("%d est un triple\n", c);
else {
  printf("%d...", c);
  printf("n'est pas un triple\n");
}
```

7.3 `switch`

- accroît la lisibilité du code (pas de `elif` en C).
- branchement au **premier `case` qui convient**, et exécution jusqu'au bout (`break`)
- série d'instructions (et non un bloc)

```
switch (expression) {
 case constante_0 :
     instructions...
 case constante_1 :
     instructions...
 ...
 default :
     instructions...
}
```

Exemple :

```
scanf("%d",&a);
switch(a%3) {
  case 0 :
    printf("a est un multiple de 3\n");
    break;
  case 1 :
    printf("a est congru à 1 modulo 3\n");
    break;
  default :
    printf("a n'est pas congru à 1 ou 0 modulo 3\n");
    break;
}
```

7.4 Opérateur ternaire

L'opérateur ternaire permet de remplacer des tests et d'écrire un code plus concis (pas toujours très lisible).

L'expression : `condition ? expr1 : expr2` vaut `expr1` si `condition` est vraie et `expr2` sinon.

Exemple (minimum) : `c = a<b?a:b`

Exemple (suite de Collatz)

- $u_{n+1} = u_n/2$ si u_n est pair
- $u_{n+1} = 3 \times u_n + 1$ si u_n est impair

```
u = u%2 ? 3*u+1 : u/2;
```

8 Précédence des opérateurs

Nous avons vu les opérateurs d'affectation, d'affectation élargie, l'opérateur ternaire, l'arithmétique, les comparaisons… Que se passe-t-il si on mélange tout ça sur la même ligne et sans parenthèse ? Il y a des règles de priorité strictes, résumées dans ce tableau. Mais n'oubliez pas : dans le doute, mettez des parenthèses, c'est bien plus simple. Les opérateurs les plus prioritaires sont en haut du tableau.

op. unaires : `+`, `-`, `++` `--` `!` `~`, cast
`*` `/` `%`
`+` `-`
`<<` `>>`
`<` `>` `<=` `>=`

`==` `!=`
`&`
`^`
`|`
`&&`
`||`
`? :` (opérateur ternaire)
`=` `+=` `-=` `*=` `/=`
`,`

Les parenthèses permettent de modifier l'ordre des opérations. Les opérateurs de la première ligne sont associatifs à droite (`-++m` vaut `-(++m)`).