Langage C

Wieslaw Zielonka
zielonka@irif.fr

Ouvrages

- Kernighan, Ritchie - Le langage C. Norme ANSI. 2e édition, DUNOD
  le livre des concepteurs du langage, lecture aride

- J-P Braquelaire - Méthodologie de la programmation en C. Norme C99 - API POSIX, 4e édition, DUNOD
  pas vraiment pour les débutants, mais utile en complément. Un peu daté.

- Peter van der Linden - Expert C programming. Deep C secrets. Prentice Hall, 1994.
  Mon livre préféré, daté mais toujours une lecture agréable, le niveau juste ce qu'il faut pour quelqu'un qui sait programmer mais ne connaît pas C.

- Ben Klemens, 21st Century C, O'Reilly

- https://www.rocq.inria.fr/secret/Anne.Canteaut/COURS_C/

Les normes du langage C

- 1989 ANSI C, ratifié comme ISO standard en 1990
- 1999 ISO C standard
- 2011 ISO C standard (l'option -std=c11 du compilateur gcc)

```c
/* fichiers en-tête qui contiennent les déclarations
(prototypes)
 * de fonctions utilisées dans le programme */
#include <stdio.h>

int main(void){
  int tab[]={-5, 8, 12, 9, -4, 21, -31};
  double s = 0;
  /* calculer le nombre d'elements de tab */
  int taille = sizeof(tab)/sizeof(tab[0]);
  /* calculer la somme de tous les éléments */
  for(int i=0 ; i <  taille ; i++){
     s += tab[i];
  }
  s/=taille; /* la moyenne */
  printf("somme=%8.3f\n",s);/*imprimer le resultat*/
  return 0; /*main doit retourner un int*/
}
```

Explications

- L'exécution de programme commence toujours par la fonction main:

  int main(void){ }

- main() doit retourner un entier, la valeur 0 indique terminaison correcte du programme, une valeur > 0 un terminaison incorrecte

- La fonction printf() est définie dans le fichier en-tête stdio.h sert à faire afficher les valeurs d'expressions

Compiler le programme C

Préparer le le fichier Makefile dans le même répertoire que le fichier source prog1 :

```
CC=gcc

CFLAGS=-g   -Wall

ALL=prog1

all : $(ALL)

prog1 : prog1.c

clean:
	rm -rf  *~  $(ALL)
```

le caractère TAB

CC CFLAGS - les variables de make

gcc - le nom de compilateur

Valeur de CFLAGS : les options de compilation

-Wall : Affichage de tous les avertissements.

-g   : produire le code compatible avec le debogueur

prog1.c   le nom du fichier source, à remplacer avec le nom de fichier contenant votre programme

prog1 le nom de fichier exécutable créé par le compilateur

prog1: prog1.c   est une ligne de dépendances, elle indique que pour obtenir prog1; il faut compiler prog1.c

les lignes indentation (rm) commencent par le caractère TAB et non pas par des espaces.

Terminer la dernière ligne par le retour à la ligne.

Makefile et la commande make

Une fois Makefile préparé on appelle la commande

make

make lit le fichier Makefile est exécute les commandes de compilation.

La commande

make clean

provoque l'exécution de la commande rm qui cuit la dépendance clean dans Makefile.

Exécuter la commande de compilation directement est découragé.

Makefile pour plusieurs programmes

```
CC=gcc
CFLAGS=-g   -Wall
ALL=prog1   prog2 prog3
all : $(ALL)
prog1: prog1.c
prog2: prog2.c
prog3: prog3.c
clean:
        rm -rf  *~  $(ALL)
```

Maintenant

make

compilera trois programmes qui se trouve dans le fichiers prog1, prog2 et prog3

make prog2

demande la compilation de prog2 uniquement

Types entiers

- Types entiers signés :
  signed char
  short (short int)
  int
  long   (long int)
  long long   (long long int)

- Types entiers non-signés :
  unsigned char
  unsigned short
  unsigned int
  unsigned long
  unsigned long long

Types entiers

le standard C garantie que

1 = sizeof(signed char) <= sizeof(short) <=
sizeof(int) <= sizeof(long) <= sizeof(long long)

où l'opérateur sizeof(…) retourne le nombre de bytes (octets) de mémoire occupés par une donnée d'un type.

Sur mon MacBook :

type	nombre de bytes
signed char	1
short	2
int	4
long	8
long long	8

Types de base réels

- float (à ne pas utiliser)
- double
- long double

Expressions arithmétiques

Division entière et modulo :

Si a/b == d et a%b == r alors

- |r| < |b| et
- a == b*d + r

Par exemple :

(-17)%(-5) == 3 et (-17)%(-5) == -2

(-17) / 5 == -3 et (-17)%5 == -2

17/(-5) == -3 et 17%(-5) == 2

17/5 == 3 et 17%3 == 2

Comment C calcule la valeur d'une expression arithmétique ?

signed char a = 2;
short b = -2;
Comment C calcule a*b ?

Si tous les arguments sont de types entiers signés alors

- si tous les éléments sont de types signed char, short, int alors chaque élément est transformé en int et le résultat est de type int

- sinon si le plus grand type est long alors chaque élément est transformé en long et le résultat est long

- sinon si le plus grand type est long long alors chaque élément est transformé en long long et le résultat est long long

Comment C calcule la valeur d'une expression arithmétique ?

Si on mélange les réels et entiers alors tous les arguments sont transformés en réels appropriés et le résultat est réel.

Avertissement :

Les règles de calcule sont peu intuitives si une expression mélange les entiers signés et non-signés.

La règle de bon sens fortement recommandée:

- ne mélangez jamais les entiers signés et non-signés ni dans les expressions ni dans les relations (ou faites un cast sur les entiers non-signés)

- utilisez les non-signés uniquement pour les opérations bit à bit et pour rien d'autre

Exemple où mélanger les entiers signés et non-signés donne des résultats bizarres

```
int a = -1;
unsigned int b = 1;


if(a < b)
  printf("a<b \n");
else
  printf("a>=b \n");


if( a < (int) b )
   printf("a < (int)b \n");
else
   printf("a >= (int)b \n");
```

Exemple où mélanger les entiers signés et non-signés donne des résultats bizarres

```
int a = -1;
unsigned int b = 1;


if(a < b)
  printf("a<b \n");
else
  printf("a>=b \n");
```

affiche
a>=b

```


if( a < (int) b )
   printf("a< (int)b \n");
else
   printf("a>=(int)b \n");
```

affiche
a<(int)b

Expressions arithmétiques comme conditions logiques

C traite une expression arithmétique dont la valeur est différente de 0 comme la valeur logique VRAI et une expression arithmétique dont la valeur est 0 comme FAUX.

```
int a = 5, b=-4;
if( a*b ) then {
   printf("VRAI");
}else{
   printf("FAUX");
}
```

affiche VRAI puisque a*b est différent de 0.

Relations

a, b -- les valeurs numériques.

- a < b
- a <= b
- a > b
- a >= b
- a == b
- a != b

Ces expressions valent soit 1 (VRAI) si la relation est vraie soit 0 (FAUX) si la relation est fausse.

En C la valeur d'une relation n'est pas un booléen mais un entier, soit 1 (vrai) soit 0 (faux).

Opérations logiques

- exp1 && exp2

  AND logique, vrai (1) si et seulement si exp1 et exp2 sont vraies (différentes de 0). Si exp1 est faux alors exp2 ne sera pas évalué.

- exp1 || exp2

  OR logique, vrai (1) si au moins une de deux est vraie (différente de 0). Si exp1 est vrai alors exp2 ne sera pas évalué.

- ! exp
  négation, vrai(1) si et seulement si exp est faux (0)

Evaluation paresseuse des opérations logiques (rappel)

- exp1 && exp2
  exp2 est évaluée uniquement quand exp1 est vrai

```
int tab[10];
int i;
…..
for(i = 0; i < 10 && tab[i] != 0; i++)
        ; /*boucle vide*/
```

- chercher le premier élément de `tab[]` égal 0.
  Quel est le problème si on remplace la condition par
  `tab[i]!=0 && i < 10`
  Qu'est-ce qui se passe quand `i` vaut `10` ?

- exp1 || exp2
  exp2 est évalué uniquement si exp1 est faux. Si exp1 est vrai le résultat de l'expression est vrai sans que exp2 soit évalué.

if

```
if( condition ){
    instructions
}
```

```
if( condition ){
    instructionsA
}
else{
    instructionsB
}
```

if

```
if( condition1 ){
   instructions1
}
else if( condition2 ){
   instructions2
}
…
else{
   instructions_n
}
```

while

```
while( condition ){
   instructions
}
```

Remarque : si le corps de la boucle est vide alors on écrit

```
while( tab[i++] )
    ; /* boucle vide */
```

(On suppose que tab[] contient au moins un élément 0.)

~~`while( tab[i++] ) ;`~~

do while

```
do {
     instructions
}while( condition );
```

(1) on exécute les instructions

(2) on vérifie la condition, si satisfaite alors on revient à (1)
sinon on termine la boucle.

La condition vérifiée après chaque exécution de la boucle.
La boucle exécutée au moins une fois.

for

```
for( initialisation ; condition ;incrémentation ){

}
```

`initialisation` exécutée une fois, avant l'entrée dans la boucle

`condition` est vérifiée au début de la boucle et si vraie alors la boucle est exécutée

`incrémentation` évaluée à la fin de chaque boucle et on revient au début, à la vérification de la condition

```
for( i = 0, j=99;  i < j ;  i++, j--){
  if( tab[i] == tab[j] )
       break;
}
```

Notez , (virgule) qui permet de connecter les expressions.

for

Chaque partie de `for()` peut être vide, donc

```
for(  ;  ; ){

}
```

est une boucle infinie, comme d'ailleurs

```
while( 1 ){

}
```

break et continue

`break` provoque la sortie de la boucle, le saut vers la première instruction après la boucle

`continue` termine l'itération courante de la boucle, on passe à l'itération suivante (on revient au début de la boucle et on refait le test de la terminaison)

int s = 0;

```
for(int i=0; i < 100; i++){

    if( tab[i] <= 0 ) /* rien à faire pour les elements <=0
                       * On passe au suivant */
        continue;

   s += tab[i];

 }
```

Faire la somme des tous les éléments positif de tab.

affectation

```
int a,b,c;

a = b * c;

Rappel :

a *= b;   ->  a = a * b;

même chose pour +  /  -

a /= b;    -> a = a / b;

a += b;    ->  a = a + b;

a -= b;    ->  a = a - b;

mais

        a = expr;

est aussi une expression dont la valeur est égale à la valeur de expr

a = b = c*d;    est la même chose que a = ( b = c*d ) ;
```

= ou == ?

```
int a;

a = expr;
```

c'est aussi une expression dont la valeur est égale à la valeur de l'expression expr.

```
a = 0;
if( a = 5 ){
    printf("egal\n");
}else{
    printf("different\n");
}
```

a = 5 vaut 5, c'est différent de 0 donc la condition de if satisfaite, impression "egal"

un compilateur moderne comme gcc émet un warning quand il détecte = où on attend plutôt ==

switch

```
switch( expression )
{
   case expr-const : instructions ;

   case expr-const : instructions ;

   default : instructions ;

}
```

N'oubliez pas break après les instructions de chaque case (sauf si la liste d'instructions est vide).

switch

```
int tab[]={-1,-1,2,-1,-2,4,-7,0,-2,2,4};
int nb = sizeof(tab)/sizeof(tab[0]);
int c1=0, c2, autre;

for(int i=0; i < nb; i++){
   switch(tab[i]){
      case 1:
      case -1:
         c1++;
         break;
      case 2:
      case -2:
         c2++;
         break;
      default:
         autre ++;
         break;   /* ce break ne sert à rien mais recommandé */
}
```

calculer le nombre d'éléments dont la valeur absolue est 1 (c1),

le nombre d'éléments de tab dont la valeur absolue est 2 (c2), et le nombre de tous autres éléments.

vecteurs

```
int tab[5];
int s=0;

for(int i = 0; i < 5 ; i++ ){
    s += tab[i];
}
```

5 - écrire les constantes en dur (constantes magiques) est à proscrire. Difficile à maintenir.

vecteurs

```
#define  NB_ELEM  5

/* définition d'une constantes symbolique avec la
directive define */

int tab[NB_ELEM];
int s=0;

for(int i = 0; i < NB_ELEM ; i++ ){
   s+=tab[i];
}
```

il suffit de changer la valeur de NB_ELEM, pas la peine de chercher toutes les occurrences de 5 dans le code.

vecteurs

```
double tab[] = {-4.8, 6.1, 57.0, 23.99, -11.32, 4.5};
int nb_elem, i;
double s = 0;

nb_elem = sizeof(tab)/sizeof(tab[0]);

for(i=0 ; i < nb_elem; i++)
   s += tab[i];
```

`sizeof()` n'est pas une fonction mais un opérateur (pas besoin d'include) évalué par le preprocesseur C.

`sizeof(nom de vecteur)` == le nombre d'octets de mémoire occupé par le vecteur

`sizeof(tab[0])` == le nombre d'octets de mémoire occupés par l'élément `tab[0]`

vecteur comme paramètre de fonction

```
double somme(int nb_elem, double t[]){
  int i;
  double s;
  s=0;
  int nb = sizeof(t)/sizeof(t[1]);

  for(i=0 ; i < nb_elem; i++)
    s += t[i];
  return s;
}
```

sizeof(t) ne donne pas la taille de vecteur
t en octet, t n'est plus vraiment un vecteur
même s'il est utilisé comme vecteur
à l'intérieur de la fonction somme() !

```
int main(void){
  double tab[] = {-4.8, 6.1, 57.0, 23.99, -11.32, 4.5};
  int n = sizeof(tab)/sizeof(tab[i]); /* OK, tab[] n'est pas
                                         paramètre de main */
  double s = somme(n, tab);

}
```

Quand un vecteur est un paramètre d'une fonction il est nécessaire de passer le nombre d'éléments du vecteur comme un autre paramètre de la fonction.
C est incapable de trouver le nombre d'éléments d'un vecteur obtenu comme paramètre de fonction.

Fonctions - définition versus déclaration

```
#include <stdio.h>

double somme(int nb, double tab[]);

int main(void){
  double tab[] = {-4.8, 6.1, 57.0, 23.99, -11.32, 4.5};
  int n = sizeof(tab)/sizeof(tab[i]);
  double s = somme(n, tab);
  printf("somme = %f\n",s);
}
```

déclaration ou prototype de la fonction somme( )
pas de corps de fonction,
juste les types de paramètres.

définition de la fonction somme()

```
double somme(int nb_elem, double t[]){
  double s=0;
  for(int i=0 ; i < nb_elem; i++)
    s += t[i];
  return s;
}
```

Fonctions - déclarations de fonctions

Quand on déclare une fonction les noms de paramètres sont optionnels :

```
double somme(int, double[]);

double somme(int nb_elem_t, double t[]) ;
```

Les noms de paramètres peuvent être quelconques (pas forcement les mêmes que dans la définition de la fonction).

Chaque fonction doit être soit déclarée soit définie avant qu'elle soit appelée.

Fonctions - passage de paramètres par valeur

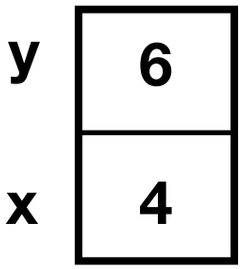
```
void echange(int a, int b){
   int c;
   c = a;
   a = b;
   b = c;
}

int main(void){
   int x = 4;
   int y = 6;
   echange(x,y);
   printf("x=%d y=%d\n", x, y);
   return 0;
}
```

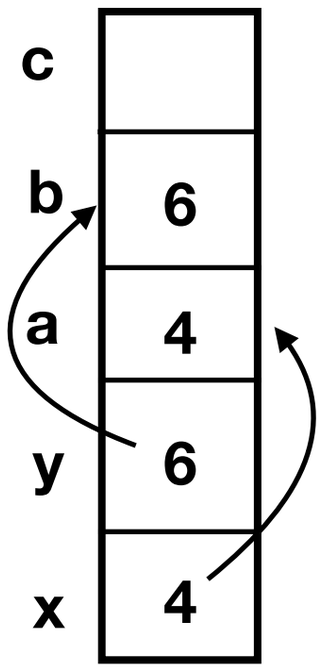
Quelles sont les valeurs de x, y après l'appel à echange() ?

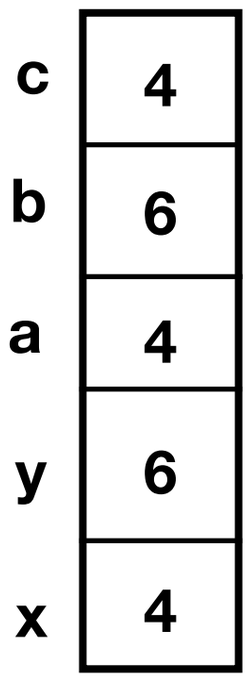
Fonctions - passage de paramètres (par valeur)

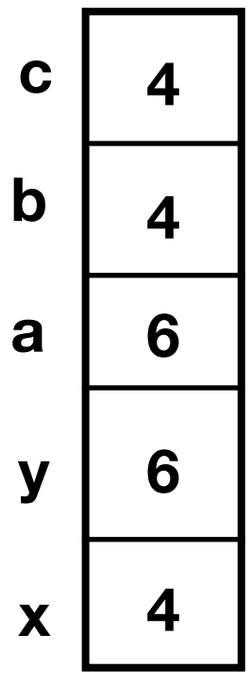
appel à échange



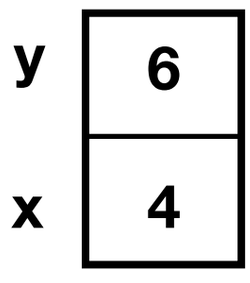
l'exécution echange()







retour de la fonction echange()




```
void echange(int a, int b){
   int c;
   c = a;
   a = b;
   b = c;
}

int main(void){
   int x = 4;
   int y = 6;
   echange(x,y);
   printf("x=%d y=%d\n", x, y);
   return 0;
}
```

a, b, c nouvelles variables, locales à la fonction échange

a et b sont initialisées avec les valeurs obtenues
en évaluant les arguments de l'appel de change(),
c n'est pas initialisé.