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

- Peter van der Linden - Expert C programming. Deep C secrets. Prentice Hall, 1994.
  Daté mais toujours agréable à lire, 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 très bien mais trop gros et détaillé pour un débutant.

- 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)
- 2017 ISO C standard (corrections de bugs de c11)

```c
/* fichier 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, même chose que : s = s/taille ; */
  printf("somme=%8.3f\n",s);/*imprimer le resultat*/
  return 0; /* main() doit retourner un int */
}
```

Explications

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

  int main(void){ }

ou int main( ) { }

- main() doit retourner un entier, il existe deux constantes entières :

  EXIT_SUCCESS et EXIT_FAILURE

  définies dans le fichier en-tête stdlib.h qui peuvent être utilisées comme les valeurs de retour de main() pour indiquer l'exécution correcte et l'exécution qui n'arrive pas à calculer les résultats. Donc je peux réécrire le programme précédent:

```
#include <stdlib.h>
#include <stdio.h>
int main( ){
    .......
    return EXIT_SUCCESS;
}
```

- La fonction printf() est définie dans le fichier en-tête stdio.h

Compiler le programme C

compiler un programme qui se trouve dans fichier hello.c sur un terminal :

```
gcc –Wall hello.c –o hello
```

gcc - le compilateur C

`–Wall` une option de compilateur (obligatoire). Permet de voir tous les avertissements (warning). Compilation avec des avertissements signifie que le programme est presque sûrement incorrect (mais si la compilation réussie).

`hello.c` le fichier text qui contient votre code

`hello` le fichier exécutable produit par le compilateur

Exécuter le programma compilé depuis le terminal :

```
./hello
```

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 en paramètre.

Sur mon MacBook :

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

Types pour les nombres 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 ?

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

Comment C calcule la valeur d'une expression arithmétique ?

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 retypage (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");
```

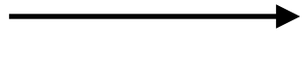
Exemple où mélanger les entiers signés et non-signés donne des résultats bizarres

```
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 (true) et une expression arithmétique dont la valeur est 0 comme FAUX (false).

```
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).
  Evaluation paresseuse : 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).
  Evaluation paresseuse : 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 évaluée.

if

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

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

if

```
if( condition1 ){
   instructions1
}
else if( condition2 ){
   instructions2
}
else if( conditions3 ){
   instructions3
}
…
else{
   instructions_n
}
```

while

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

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

```
while( i < 10 && tab[ i++ ] )
    ; /* boucle vide */
```

```
while( i < 10 && 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 chaque exécution de la boucle et si vraie alors la boucle est exécutée

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

```
int tab[100];
.........


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 la boucle

```
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 condition de terminaison)

int s = 0;

int tab[100];

..... /* ici on mets des valeurs dans tab */

```
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 positifs 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;
```

affectation

En C

```
a = expr;
```

est une expression dont la valeur est égale à la valeur de expr. Cette expression a un effet de bord qui consiste à affecter une nouvelle valeur à la variable a.

```
a = b = c*d;
```

est la même chose que

```
a = ( b = c*d ) ;
```

digression

```
int a = 1, b = 2, c = 3, d;

d = b * ( a = c * c );

/* correcte en C? Quelle valeur de d? */
```

digression

```
int a = 1, b = 2, c = 3, d;

d = b * ( a = c * c );
```

Correcte en C ? Oui.

Même si correcte en C il ne faut coder de cette façon sauf si vous participez à The International Obfuscated C Code Contest :

```
http://www.ioccc.org

https://fr.wikipedia.org/wiki/
International_Obfuscated_C_Code_Contest
```

= 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");
}
```

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

les compilateurs modernes, comme gcc, émettent un warning quand ils détectent = là 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;
int c2 = 0;
int autre = 0;

for(int i=0; i < nb; i++){
   switch(tab[i]){
      case  1:
      case -1:
         c1++;  /* compte les occurrences de 1 et -1 */
         break;
      case  2:
      case -2:
         c2++;  /*compte les occurrences de 2 et -2 */
         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. Le code difficile à maintenir.

vecteurs

```c
#define  NB_ELEM  5

/* définition d'une constante symbolique avec la
directive define, ce n'est pas une instruction,
pas de ; à la fin */

int tab[ NB_ELEM ];
int s=0;

/* ici vient le code pour remplir tab */

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

Pour changer le nombre d'éléments dans tab[] il suffit de changer la définition de la macro-constante `NB_ELEM`, pas la peine de chercher toutes les occurrences de 5 dans le code.

vecteurs

```
/* déclarer et initialiser un vecteur */
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;
}
```

si t un paramètre de fonction alors `sizeof( t )` ne donne plus 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 qui passe 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

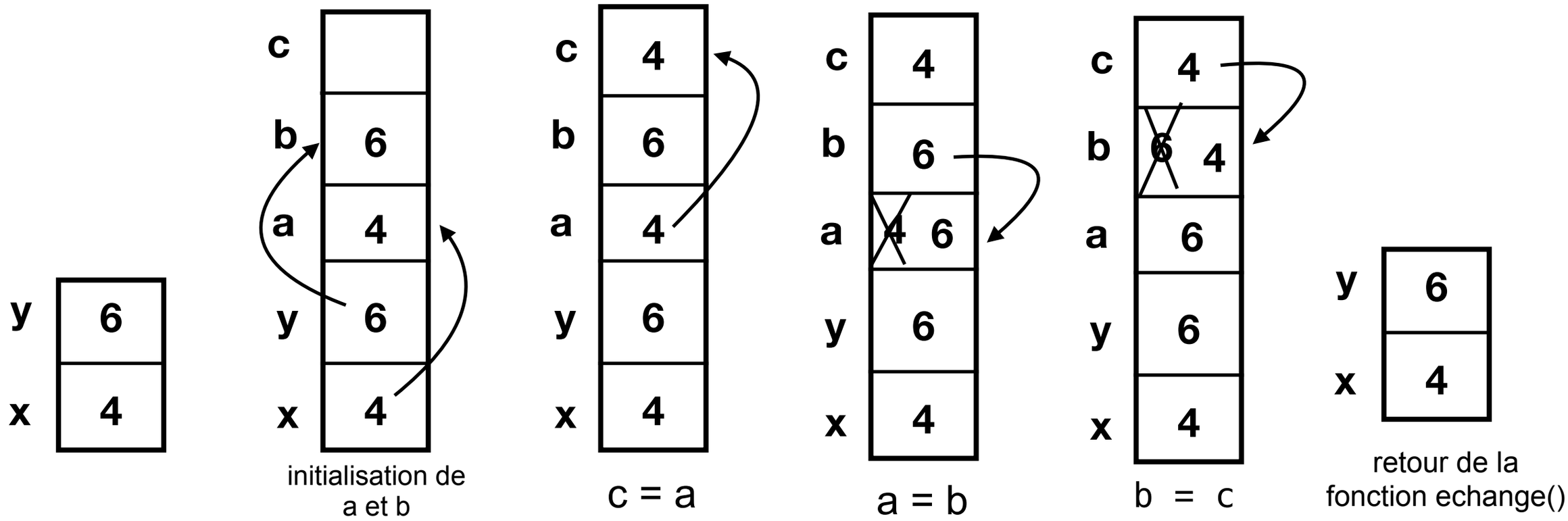
```
void echange(int a, int b){
   int c;
   c = a;
   a = b;
   b = c;
}

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)

l'exécution echange(x,y)




```
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.