Programmation C

Tableaux, entrées/sorties de bases et structures de contrôle

ESIPE Info 1ère année

Université de Marne-la-Vallée

Les tableaux

- éléments contigus de même type
- déclaration : `type nom[taille];`
  ```
  int t[N];
  double cosinus[360];
  ```
- avec initialisation :
  ```
  char vowel[6] = {'a','e','i','o','u','y'};
  ```
- La taille devient optionnelle :
  ```
  char* name[] = {"Smith", "Doe", "X"};
  ```

Les tableaux

- type des éléments quelconque
- numérotation de `0` à `taille-1`

```
1  void init(int value[]){
2    int i;
3    for (i = 0 ; i < N ; i++){
4      value[i] = i;
5    }
6  }
```

Les erreurs sur les indices de tableaux restent la source de fautes de segmentation la plus courante en Licence...

Taille

- un tableau ne connaît pas sa taille !
- l'opérateur `sizeof` ne fonctionne que pour les tableaux dont la taille est connue par la fonction à la compilation.

```
1  void test_sizeof(int t[]){
2    printf("sizeof(t) from function = %lu bytes\n", sizeof(t));
3  }
4
5  int main(int argc, char* argv[]){
6    int m[60];
7    test_sizeof(m);
8    printf("sizeof(m) from main = %lu bytes\n", sizeof(m));
9    return 0;
10 }
```

donne

```
sizeof(t) from function = 8 bytes
sizeof(m) from main = 240 bytes
```

Taille

3 solutions :

- connaître la taille grâce à une constante
  ```
  #define SIZE_TAB 256
  static const int SIZE_TAB=256;
  ```
  (la deuxième solution fabrique une constante visible par le débogueur)

- passer la taille en paramètre
  ```
  void affiche_tableau(int T[], int n);
  ```

- utiliser un élément marqueur comme `'\0'`
  comme par exemple pour ce qui est fait pour les chaînes de caractères

Débordement

- aucun contrôle de débordement
- attention aux surprises !

```
1  #define N 10
2  void foo(int A[], int B[]){
3    int i;
4    for (i = 0 ; i < 20 ; i++){
5      B[i] = 33;
6    }
7    for (i = 0 ; i < N ; i++){
8      printf("%d %d\n", A[i], B[i]);
9    }
10 }
11
12 int main(int argc, char* argv[]){
13   int B[N];
14   int A[N];
15   foo(A, B);
16   return 0;
17 }
```

Débordement

produit...

```
nborie@perceval:~> gcc -o test test2.c -Wall -ansi
nborie@perceval:~> ./test
33 33
33 33
33 33
33 33
33 33
33 33
33 33
33 33
33 33
1362109184 33
32767 33
```

deux 33 sont dans la nature... ou dans la ram

Tableaux à *n* dimensions

- `int t[100][16][45];`
- chaque `int t[i][j]` est un tableau de 45 `int`
- Quand on passe un tableau (avec notations crochets) à une fonction, on doit mettre toutes les dimensions sauf la première.

  `void foo(int t[][]);` ⇒ Non !
  `void foo(int t[][M]);` ⇒ Ok !

  Si on connaît toutes les dimensions via des macros ou autre, autant les mettre...

Tableaux à *n* dimensions

- attention à l'ordre de parcours !!! C alloue les tableaux à deux dimensions par lignes (alors que `Fortran` fait de l'allocation par colonnes)

`int T[2][3];` donne en mémoire
pour @ l'adresse de T[0][0] :

T[0][0]	T[0][1]	T[0][2]	T[1][0]	T[1][1]	T[1][2]
@+0	@+4	@+8	@+12	@+16	@+20

$\Rightarrow$ on boucle d'abord sur la première dimension !

Benchmark pour le parcours

```
 1  const int N=40000;
 2  char T[N][N];
 3
 4  int main(int argc, char* argv[]){
 5    int i,j;
 6    time_t before = time(0);
 7    for (i=0 ; i<N ; i++){
 8      for (j=0 ; j<N ; j++)
 9        T[i][j] = 1;
10    }
11    time_t middle = time(0);
12    printf("%d seconds\n", (int)(middle-before));
13    for (j=0 ; j<N ; j++){
14      for (i=0 ; i<N ; i++)
15        T[i][j] = 1;
16    }
17    time_t end = time(0);
18    printf("%d seconds\n", (int)(end-middle));
19    return 0;
20  }
```

donne

```
3 seconds
24 seconds
```

Passage d'adresse

- les tableaux sont implicitement passés par adresse
- tableau = adresse de son premier élément

```
1  /* equivalent a :
2  void foo(int* t) */
3  void foo(int t[]){
4    t[1] = 42;
5  }
6
7  int main(int argc, char* argv[]){
8    int t[3];
9    t[0] = t[1] = t[2] = 0;
10   foo(t);
11   printf("%d\n",t[1]);
12   return 0;
13 }
```

donne bien 42 si vous en doutiez...

Passage par adresse

- pas de différence formelle entre un tableau de `truc` et un `truc` passé par adresse.
- pour le compilateur, les 3 prototypes suivants sont les mêmes !

```
void foo(char s[]);
void foo(char* s);
void foo(char *s);
```

- pour s'y retrouver, on peut noter par convention :
  `float s[]` : `s` est un tableau de `float`
  `float *s` : `*s` est un `float` passé par adresse

Retourner une adresse

- **ne jamais retourner l'adresse d'une variable locale**

  les données du tableau sur la pile en sortie de fonction se feront écrasées par le code qui suit...

  $\Rightarrow$ on ne peut pas retourner l'adresse d'un tableau créé localement dans une fonction !

- on peut retourner l'adresse d'une variable allouée dynamiquement

  avec les listes chaînées, on a souvent une fonction du type:

```
Cellule* alloue_cellule(int n);
```

Tableaux et pile

- les tableaux en variables locales = place mémoire sur la pile d'exécution
- attention aux explosions !

```
1  const int N=2000;
2
3  void test_matrix(){
4    int t[N][N]; /* gros gros sur la pile... */
5    t[N-1][N-1] = 42;
6    printf("%d\n", t[N-1][N-1]);
7  }
8
9  int main(int argc, char* argv[]){
10   test_matrix();
11   return 0;
12 }
```

donne une erreur de segmentation...(N = 1000 passe encore sur ma machine)

printf

- fonction d'affichage (`stdio.h`)

  ```
  printf("format", ...);
  ```

- variables indiquées avec :

  `%d` : entier `%f` : réel
  `%c` : caractère `%s` : chaîne
  + `%%` pour le caractère `'%'`

- exemple :

  ```
  printf("moyenne(%d, %d) = %f\n", a, b, moy);
  ```

- voir `man 3 printf` pour plus d'options

printf

- les fonctions ayant un nombre variable d'argument s'appellent les fonctions variadiques
- il doit y avoir autant de variables que de `%...`
- ne pas afficher une chaîne directement

```
1  int main(int argc, char* argv[]){
2    char* insult = "*!%s#\n";
3    printf(insult);
4    return 0;
5  }
```

donne

```
nborie@perceval:~> gcc -o test test6.c -Wall -ansi
test6.c: In function ‘main’:
test6.c:5:3:attention:le format n’est pas une chaîne littérale
[-Wformat-security]
nborie@perceval:~> ./test
*!..Z..^?#                #<-- deux caracteres non affichable dans ce cours
nborie@perceval:~>
```

printf

- l'affichage est bufferisé
  - l'affichage se déclenche quand il y a un `'\n'`
  - l'affichage se déclenche quand le buffer est plein
  - la sortie d'erreur standard n'est pas bufferisée

```
1  int main(int argc, char* argv[]){
2    char* nom = "Jean";
3
4    printf("Ou se trouve l'erreur de segmentation");
5    nom[34] = 'y';
6    return 0;
7  }
```

Le programme n'affiche rien. Une erreur de segmentation arrête son exécution.

Messages d'erreurs

- utiliser la sortie d'erreur avec :
  `fprintf(stderr, "format", ...);`

- même fonctionnement que `printf` mais non bufferisée !

- exemples :

  `perror` pour les erreurs systèmes classiques, voir le `man` !

  Pour `fopen`, `if (file == NULL) ...`
  ```
  fprintf(stderr, "Echec de l'ouverture du fichier %s \n",
  file);
  ```

  Pour `malloc`, `if ((C = (Cellule*)malloc()) == NULL) ...`
  ```
  fprintf(stderr, "Problème d'allocation mémoire\n");
  ```

  Pour l'utilisation des programmes, `if (argc <= 3) ...`
  ```
  fprintf(stderr, "Le programme nécessite 3 arguments :
  ...\n");
  ```

scanf

- fonction de saisie au clavier (`stdio.h`)
- ressemble à `printf`, mais:
  - que des variables, pas de constantes
  - `&` devant les variables, sauf pour les chaînes
  - la chaîne de format n'est pas affichée

```
int a,b; char s[64];
scanf("%d %d %s", &a, &b, s);
```

scanf

- espaces ignorés : "%d %d" ⇔ "%d%d"
- saisie bufferisée par ligne:

```
1  const int N=8;
2
3  void get\_int\_array(int t[]){
4    int i;
5    for (i=0 ; i<N ; i++){
6      scanf("%d", &t[i]);
7    }
8    printf("done.\n");
9  }
```

s'utilise comme cela :

```
$>./a.out
1 2 3 4 5 6 7 8
done.
$>
```

scanf

- les chaînes sont délimitées par les espaces, les tabulations et les sauts de ligne

```
1  int main(int argc, char* argv[]){
2    int a;
3    char s1[32];
4    char s2[32];
5    scanf("%d␣%s␣%s", &a, s1, s2);
6    printf("%d␣%s␣%s", a, s1, s2);
7    return 0;
8  }
```

donne avec les entrées suivantes:

```
$>./a.out
5  hello 32   abc
5 hello 32
$>
```

scanf

- `scanf` renvoie le nombre de variables saisies correctement
- en cas d'erreur:

```
1  int main(int argc, char* argv[]){
2     int a,b,c,n;
3     n = scanf("%d␣%d␣%d",&a,&b,&c);
4     printf("%d␣%d␣%d␣%d\n",n,a,b,c%);
5  }
```

donne pour les entrées suivantes:

```
$>./a.out
4 8 hello
2 4 8 4198592
$>
```

scanf

- lecture de caractères

  attention à ce qu'il reste dans le buffer !!!

```
1  #define YES 1
2  #define NO 0
3
4  int yes_or_no(){
5    char c;
6    do{
7      printf("y/n ? ");
8      scanf("%c", &c);
9    }while (c != 'y' && c != 'n');
10   if (c == 'y')
11     return YES;
12   return NO;
13 }
```

donne avec les entrées suivantes :

```
nborie@perceval:~> ./test
y/n ? u
y/n ? y/n ? y
nborie@perceval:~>
```

if... else...

- `if (condition) instruction1 else instruction2`
  ou `if (condition) instruction`
- opérateurs de comparaisons :
  - `a < b` `a <= b` `a > b` `a >= b` `a != b`
  - `a == b` à ne pas confondre avec `a = b`
- opérateurs sur les entiers et les réels
- ne marche pas sur les chaînes

Portée du `else`

- en cas d'ambiguïté, le `else` se rapporte au `if` le plus proche

- pour éviter les mauvaises surprises et faciliter la lisibilité, bien utiliser l'indentation et les blocs

```
1  if (a == 0)
2    if (b == 1)
3      /* ... */
4    else
5      /* ... */
```

```
1  if (a == 0){
2    if (b == 1){
3      /* ... */
4    } else{
5      /* ... */
6    }
7  }
```

opérateurs logiques

- convention du langage C : 0 = faux et ≠0 = vrai
- a&&b a||b !a

```
1  int main(int argc, char* argv[]){
2    printf("4 AND 6\t = %d\n"
3           "4 AND 0\t = %d\n"
4           "4 OR 6\t = %d\n"
5           "0 OR 0\t = %d\n"
6           "NOT 17\t = %d\n"
7           "NOT 0\t = %d\n",
8           4&&6, 4&&0, 4||6, 0||0, !17, !0);
9    return 0;
10 }
```

done

```
4 AND 6  = 1
4 AND 0  = 0
4 OR 6   = 1
0 OR 0   = 0
NOT 17   = 0
NOT 0    = 1
```

Opérateurs logiques

- exemple:

  Sachant que les lettres sont rangées dans l'ordre alphabétique dans la table `ascii`...

```
1  int is_letter(char c){
2    if (( c >= 'a' && c <= 'z' )
3        || ( c >= 'A' && c<= 'Z' )){
4      return 1;
5    }
6    return 0;
7  }
```

Conditions simplifiées

- condition ⇔ expression
- `if (condition) ...` = `if (expression!=0) ...`
- écritures simplifiées

  `if (value!=0) ...` = `if (value) ...`

  `if (value==0) ...` = `if (!value) ...`

Évaluation paresseuse

- opérateurs `&&` et `||` paresseux
- évaluation de gauche à droite
- s'arrête dès que possible
  - `(a=0 && b=c)` ⇒ `b` n'est jamais affecté
  - `(a=1 || b=c)` ⇒ `b` n'est jamais affecté

Priorités

- du plus au moins prioritaire

  `()`

  `--` `++` `!` `-` unaire

  `*` `/` `%`

  `+` `-`

  `>` `>=` `<` `<=`

  `==` `!=`

  `&&` `||`

  `=`

- dans le doute, on met des parenthèses !!!

Fautes de style

- un bon code ne contient pas de bloc vide !
- tests inutiles : on ne fait des tests dont on est sûr du résultat !

```
1  if (n>0){
2     /* .. */
3  }
4  if (n<0){
5     /* .. */
6  }
7  if (n==0){
8     /* .. */
9  }
```

```
1  if (n>0){
2     /* .. */
3  }
4  else if (n<0){
5     /* .. */
6  }
7  else{
8     /* .. */
9  }
```

Expression conditionnelle

- `if (condition) a = expr1; else a = expr2;`
  $\Leftrightarrow$ `a = (condition)?expr1:expr2;`
- pratique pour les petits réglages
- ne pas utiliser si son utilisation réduit la lisibilité

Boucle `for`

- `for (instr1;condition;instr2) instr3`

  - faire `instr1`

  - tant que `condition` est vraie, faire :

    - `instr3`
    - `instr2`

classique

```
1  int i;
2  if (i=0 ; i<10 ; i++){
3     printf("i = %d\n", i);
4  }
```

double initialisation

```
1  int i,n;
2  if (i=0,n=get_max() ; i<n ; i++){
3     printf("i = %d\n", i);
4  }
```

Quand la boucle est vide

- mettre alors un bloc vide (cela insiste sur le fait que l'algorithmique se trouve dans la condition et les instructions du `for`)

```
1  /* Returns the last position of a given integer
2  in the given array, or -1 if not found */
3  int find_n(int t[], int n){
4    int i;
5    for (i=N-1 ; (i>=0) && (t[i]!=n), i--) {}
6    return i;
7  }
```

Quand la condition est vide

- la condition vide est toujours vraie
- `for (instr1;;instr2) ...`
  = boucle infinie

```
1  /* Returns sum of integers read from standard input */
2  int scan_and_sum(){
3     int sum=0;
4     int n;
5     for (;;){
6        if (scanf("%d",&n) != 1) return sum;
7        sum = sum + n;
8     }
9  }
```

Attention à la condition

- La condition est évaluée à chaque tour !
- attention aux complexités cachées

```
1  void spell(char* s){
2    int i;
3    for (i=0 ; i<strlen(s) ; i++){
4      printf("s[%d] = %c", i, s[i]);
5    }
6  }
```

est quadratique alors qu'on peut faire du linéaire

```
1  void spell(char* s){
2    int i, n;
3    for (i=0,n=strlen(s) ; i<n ; i++){
4      printf("s[%d] = %c", i, s[i]);
5    }
6  }
```

Boucle `while`

- `while (condition) instruction`
- `while` VS `for` : question de lisibilité

```
1  int scan_and_sum(){
2    int sum=0;
3    int n;
4    for (;;){
5      if (scanf("%d",&n) != 1) return sum;
6      sum = sum + n;
7    }
8  }
```

```
1  int scan_and_sum(){
2    int sum=0;
3    int n;
4    while (scanf("%d",&n) == 1){
5      sum = sum + n;
6    }
7    return sum;
8  }
```

Boucle `do...while...`

- ça existe... mais à éviter
- `do instruction while (condition);`
- ”utile” quand on doit passer au moins une fois dans la boucle
- peut éviter de devoir initialiser des variables...

```
1  int yes_or_no(){
2    char c;
3    do{
4      printf("y/n ? ");
5      scanf("%c", &c);
6    }while (c != 'y' && c != 'n');
7    if (c == 'y')
8      return YES;
9    return NO;
10 }
```

switch

- choix entre plusieurs constantes **entières** !

```
switch (expression) {
 case const1:  instruction break;
 case const2:  instruction break;
 case const3:  instruction break;
 ...
 default:  instruction;
}
```

switch

- plus élégant que plein de `if ... else ...`
- possibilité d'optimisation par le compilateur
- ne pas oublier le `default`

```
1  int get_base(char choice){
2    int base;
3    switch(choice){
4      case 'b': base=2; break;
5      case 'o': base=8; break;
6      case 'd': base=10; break;
7      case 'h': base=16; break;
8    }
9    return base;
10 }
```

donne avec le choix suivant

```
$>./a.out
B
4572656
$>
```

`switch`

- sans l'utilisation de `break`, l'exécution continue
- l'utilisation de `default` permet de factoriser les erreurs

  pour tous les cas non prévus, on envoie un message sur `stderr`

break

- peut servir à sortir d'une boucle:
  - dans certains cas d'erreurs
  - si le résultat d'un calcul est connu avant la fin

```
1  int add_product(int t1[], int t2[]){
2    int i, p1=1, p2=1;
3    for (i=0 ; i<N ; i++){
4      p1 = p1*t1[i];
5      if (p1 == 0) break;
6    }
7    for (i=0 ; i<N ; i++){
8      p2 = p2*t1[i];
9      if (p2 == 0) break;
10   }
11   return p1+p2;
12 }
```

`break`

- à utiliser avec parcimonie et bon escient
- à éviter en cas de boucles imbriquées
- à éviter si on peut quitter la fonction (préférer `return`)
- utile lorsque la sortie de boucle diminue la complexité

continue

- sert à sauter un tour de boucle
- éviter un niveau d'indentation ⇒ lisibilité (surtout si le `else` est gros)

```
1  void print_free_seats(){
2    int i,j;
3    for (i=1 ; i<NB_ROWS ; i++){
4      if (i == 13){
5        /* No seats number 13 because
6        of stupid superstitious people  */
7        continue;
8      }
9      for (j=0 ; j<N_SEATS ; j++){
10       if (seat[i][j]) printf("%2d%c ",i,j+'A');
11       else printf("    ");
12     }
13     printf("\n");
14   }
15 }
```

code équilibré

On préféra toujours un `for` (bien lisible) à un `while` (moins lisible) ou encore un `do ... while` (pénible).

Un dessin vaut mieux qu'un long discours, ainsi, un code équilibré a la silhouette suivante :