Cours de C

Tableaux, entrées/sorties de base et structures de contrôle

Sébastien Paumier

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` ⚡

```
void init(int value[]) {
    int i;
    for (i=0;i<N;i++) {
        value[i]=i;
    }
}
```

Taille

- un tableau ne connaît pas sa taille
- l'opérateur **sizeof** ne marche que pour les tableaux dont la taille est connue à la compilation

```
void test_sizeof(int t[]) {
printf("sizeof(t)=%d bytes\n",sizeof(t));
}

int main(int argc,char* argv[]) {
int m[15];
test_sizeof(m);
printf("m: %d bytes\n",sizeof(m));
return 0;
}
```

```
$>./a.out
sizeof(t)=4 bytes
m: 60 bytes
```

Taille

- 3 solutions:
  - connaître la taille grâce à une constante
  - passer la taille en paramètre
  - utiliser un élément marqueur comme le `'\0'` pour les chaînes

```c
void print1(int t[]) {
int i;
for (i=0;i<N;i++) {
   printf("%d\n",t[i]);
}
}
```

```c
void print2(int t[],
            int n) {
int i;
for (i=0;i<n;i++) {
   printf("%d\n",t[i]);
}
}
```

```c
void print3(int t[]) {
int i;
for (i=0;t[i]!=Z;i++) {
   printf("%d\n",t[i]);
}
}
```

Débordement

- aucun contrôle de débordement
- attention aux surprises!

```
#define N 10

void foo(int A[],int B[]) {
   int i;
   for (i=0;i<20;i++) {
      B[i]=33;
   }
   for (i=0;i<N;i++) {
      printf("%d %d\n",A[i],B[i]);
   }
}
```

```
$>./a.out
33 33
33 33
33 33
33 33
33 33
33 33
33 33
33 33
2293600 33
2009000225 33
```

Il y a deux 33 qui se promènent dans la nature!

Tableaux à *n* dimensions

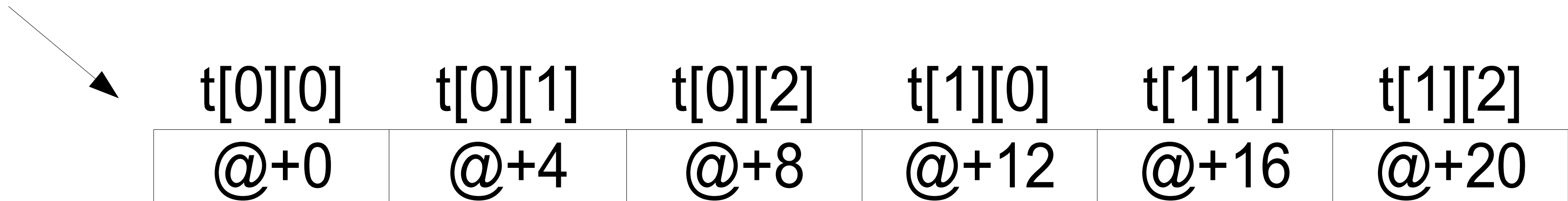
- `int t[100][16][45];`
- chaque `t[i][j]` est un tableau de 45 int
- quand on passe un tableau à une fonction, on doit mettre toutes les dimensions sauf la première :

```
void foo(int t[][M]) {
   /*
     ...
   */
}
```

```
void foo(int t[][]) {
   /*
     ...
   */
}
```

Tableaux à *n* dimensions

- `int t[2][3];`



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

- attention à l'ordre de parcours
  - pareil en théorie, pas en pratique:

```
for (i=0;i<2;i++)
  for (j=0;j<3;j++)
     ...t[i][j]...
```

```
for (j=0;j<3;j++)
   for (i=0;i<2;i++)
      ...t[i][j]...
```

Tableaux à *n* dimensions

```
#define N 15000
char t[N][N];

int main(int argc,char* argv[]) {
   int i,j;
   time_t before=time(0);
   for (i=0;i<N;i++) {
      for (j=0;j<N;j++) {
         t[i][j]=1;
      }
   }
   time_t middle=time(0);
   printf("%d seconds\n",middle-before);
   for (j=0;j<N;j++) {
      for (i=0;i<N;i++) {
         t[i][j]=1;
      }
   }
   time_t end=time(0);
   printf("%d seconds\n",end-middle);
   return 0;
}
```

```
$>./a.out
4 seconds
32 seconds
```

Passage d'adresse

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

```
/* équivalent à:
void foo(int* t) */
void foo(int t[]) {
   t[1]=666;
}

int main(int argc,char* argv[]) {
   int t[3];
   t[0]=t[1]=t[2]=0;
   foo(t);
   printf("%d\n",t[1]);
   return 0;
}
```

```
$>./a.out
666
```

Passage par adresse

- pas de ≠ formelle entre un *tableau de trucs* et un *truc* passé par adresse
- pour le compilateur, ces 3 fonctions ont le même prototype:

```
void foo(char s[]) {
   printf("(%s)\n",s);
}
```

```
void foo(char* s) {
   printf("(%s)\n",s);
}
```

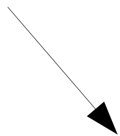
```
void foo(char *c) {
   *c='$';
}
```

- une notation pour s'y retrouver:

  `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 

```
char* uppercase(char s[]) {
    char tmp[MAX];
    int i=-1;
    do {
        i++;
        tmp[i]=toupper(s[i]);
    } while (tmp[i]!='\0');
    return tmp;
}

int main(int argc,char* argv[]) {
    char* x=uppercase("abc");
    printf("%s\n",x);
    return 0;
}
```

```
$>gcc -Wall -ansi var.c
var.c: Dans la fonction « uppercase »:
var.c:14: AVERTISSEMENT: fonction
retourne l'adresse d'une variable
locale
$> ./a.out
¨÷ÿ¿Õí·
```

ABC est sur la pile, et se fait écraser lors de l'appel à **printf**

Retourner une adresse

- conséquence:
  - on ne peut pas retourner un tableau
  - il faudra utiliser de l'allocation dynamique

Tableaux et pile

- tableau en variable locale = place occupée sur la pile
- attention aux explosions:

```
#define N 10000

void test_matrix() {
   int t[N][N];
   t[N-1][N-1]=17;
   printf("%d\n",t[N-1][N-1]);
}

int main(int argc,char* argv[]) {
   test_matrix();
   return 0;
}
```

→

```
$>./a.out
Segmentation fault
```

printf

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

```
printf(chaîne de format,arguments);
```

- variables indiquées avec:

  `%d` : entier   `%f` : réel

  `%c` : caractère   `%s` : chaîne de caractères

  \+ `%%` : le caractère `'%'`

- exemple:

```
printf("moyenne(%d,%d)=%f\n",i,j,moy);
```

- voir `man printf` pour les autres options

printf

- il doit y avoir autant de variables que de `%...`
- ne pas afficher une chaîne directement

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

```
$>./a.out
*!*!%s#
#
```

printf

- affichage bufferisé
  - quand il y a un `'\n'`
  - quand le buffer est plein

```
int main(int argc,char* argv[]) {
   char* msg="Computing...";
   printf("%s",msg);
   int i;
   float f=0;
   for (i=0;i<200000000;i++) {
      f=f+0.1;
   }
   printf("\nf=%f\n",f);
   return 0;
}
```

→

```
$>./a.out
```

3 secondes d'attente

```
Computing...
f=2097152.000000
```

Messages d'erreur

- utiliser la sortie d'erreur avec `fprintf(stderr,"...",...);`
- même fonctionnement que `printf`

```
int sum(int t[],int size) {
   int i,sum=0;
   if (size<0) {
      fprintf(stderr,"Invalid size: %d\n",size);
   }
   for (i=0;i<size;i++) {
      sum=sum+t[i];
   }
   return sum;
}
```

scanf

- fonction de saisie au clavier (`stdio.h`)
- ressemble à `printf`, mais:
  - que des variables, pas de constantes
  - `&` devant les variables, sauf 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:

```c
#define N 8

void get_int_array(int t[]) {
   int i;
   for (i=0;i<N;i++) {
      scanf("%d",&t[i]);
   }
   printf("Done.\n");
}
```

→

```
$>./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

```
int main(int argc,char* argv[]) {
   int a;
   char s1[32];
   char s2[32];
   scanf("%d %s %s",&a,s1,s2);
   printf("%d %s %s\n",a,s1,s2);
   return 0;
}
```

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

scanf

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

```
int main(int argc,char* argv[]) {
   int a,b,c,n;
   n=scanf("%d %d %d",&a,&b,&c);
   printf("%d %d %d %d\n",n,a,b,c);
   return 0;
}
```

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

scanf

- lecture de caractères
  - attention à ce qui reste dans le buffer

```
#define NO 0
#define YES 1

int yes_or_no() {
   char c;
   do {
      printf("y/n ? ");
      scanf("%c",&c);
   } while (c!='y' && c!='n');
   if (c=='y') return YES;
   return NO;
}
```

```
$>./a.out
y/n ? u
y/n ? y/n ? y
```

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 marchent pas sur les chaînes

Portée du else

- en cas d'ambiguïté, le **else** se rapporte au **if** le plus proche
- éviter les mauvaises surprises en utilisant des blocs

```
if (a==0)
   if (b==1)
      /* ... */
   else
      /* ... */
```

```
if (a==0) {
   if (b==1) {
      /* ... */
   } else {
      /* ... */
   }
}
```

Erreur classique

- pas de point-virgule après la condition ⚡

```
int odd(int n) {
   if (n%2==1);
      return 1;
   return 0;
}

int main(int argc,char* argv[]) {
   int i;
   for (i=0;i<6;i++) {
      printf("odd(%d)=%d\n",i,odd(i));
   }
   return 0;
}
```

```
$>./a.out
odd(0)=1
odd(1)=1
odd(2)=1
odd(3)=1
odd(4)=1
odd(5)=1
odd(6)=1
```

Opérateurs logiques

- convention C: `0`=faux `≠0`=vrai
- **`a&&b      a||b      !a`**

```c
int main(int argc,char* argv[]) {
   printf("4 AND 6\t= %d\n"
          "4 AND 0\t= %d\n"
          "4 OR 6\t= %d\n"
          "0 OR 0\t= %d\n"
          "NOT 17\t= %d\n"
          "NOT 0\t= %d\n",
          4&&6, 4&&0,
          4||6, 0||0,
          !17, !0);
   return 0;
}
```

→

```
$>./a.out
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:

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

Négation

- penser aux formules de Morgan:

$$\overline{A} \wedge \overline{B} = \overline{A \vee B} \qquad \overline{A} \vee \overline{B} = \overline{A \wedge B}$$

```
while ((c=fgetc(f))!=EOF &&
        !(c==' ' || c=='\t'
        || c=='\n' || c=='\r')) {
    /*
    ...
    */
}
```

```
while ((c=fgetc(f))!=EOF
        && c!=' '  && c!='\t'
        && c!='\n' && c!='\r') {
    /*
    ...
    */
}
```

Conditions simplifiées

- condition ⇔ expression
- `if (condition)...` = `if (expression!=0)...`
- écriture simplifiée:

```
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

```c
/* Returns 1 if the given string is not NULL and starts with a
 * character that is not a digit; 0 otherwise. */
int no_digit_prefix(char* s) {
   if (s!=NULL && (s[0]<'0' || s[0]>'9'))
      return 1;
   return 0;
}
```

Priorités

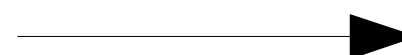
- du plus au moins prioritaire:

```
()
--   ++   !   - (moins unaire)
*    /    %
+    -
<    <=   >   >=
==   !=
&&   ||
=
```

- dans le doute, on parenthèse

Fautes de style à éviter

```
if (condition) {
   /* bloc vide */
} else {
   /*
     ...
    */
}
```



```
if (!condition) {
   /*
     ...
    */
}
```

```
if (a!=valeur_speciale) {
   s=a;
} else {
   s=valeur_speciale;
}
```

```
s=a;
```

Tests inutiles

```
if (n>0) {
   /* ... */
}
if (n<0) {
   /* ... */
}
if (n==0) {
   /* ... */
}
```

1) on évite de faire des tests dont on est sûr qu'ils seront faux

```
if (n>0) {
   /* ... */
}
else if (n<0) {
   /* ... */
}
else if (n==0) {
   /* ... */
}
```

2) on évite de faire des tests dont on est sûr qu'ils seront vrais

```
if (n>0) {
   /* ... */
}
else if (n<0) {
   /* ... */
}
else {
   /* ... */
}
```

Expression conditionnelle

- `if (condition) a=exp1; else a=exp2;`

⇔ `a=(condition)?exp1:exp2;`

- pratique pour des petits réglages:

```
int main(int argc,char* argv[]) {
   char* s="supercalifragilisticexpialidocious";
   int n=count_vowels(s);
   printf("%d vowel%s\n",n,(n>1)?"s":"");
   return 0;
}
```

Boucle for

- `for (instr1;condition;instr2) instr3` :
  - faire `instr1`
  - tant que `condition` est vraie, faire
    - `instr3`
    - `instr2`

classique:

```
int i;
for (i=0;i<10;i++) {
   printf("i=%d\n",i);
}
```

variante avec double initialisation:

```
int i,n;
for (i=0,n=get_max();i<n;i++) {
   printf("i=%d\n",i);
}
```

Quand la boucle est vide

- mettre un bloc vide { }

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

Quand la condition est vide

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

```c
/**
 * Returns the sum of integers read from
 * the standard input. The function stops
 * when there are no more integers to read.
 */
int scan_and_sum() {
   int sum=0;
   int n;
   for (;;) {
      if (scanf("%d",&n)!=1) return sum;
      sum=sum+n;
   }
}
```

Attention à la condition

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

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

→ complexité: *longueur(s)*^{2}

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

→ complexité: *longueur(s)*

Boucle while

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

```
int scan_and_sum() {
    int sum=0;
    int n;
    for (;1==scanf("%d",&n);) {
        sum=sum+n;
    }
    return sum;
}
```

```
int scan_and_sum() {
    int sum=0;
    int n;
    while (1==scanf("%d",&n)) {
        sum=sum+n;
    }
    return sum;
}
```

Boucle do...while...

- `do instruction while (condition);`
- à utiliser quand on doit passer au moins une fois dans la boucle

```
int scan_positive_integer() {
    int n=-1;
    char foo;
    do {
        if (1!=scanf("%d",&n)) {
            scanf("%c",&foo);
        }
    } while (n<0);
    return n;
}
```

```
$>./a.out
dff d-34 T_56op
56
```

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**:

```
int get_base(char choice) {
    int base;
    switch (choice) {
        case 'b': base=2; break;
        case 'o': base=8; break;
        case 'd': base=10; break;
        case 'h': base=16; break;
    }
    return base;
}
```

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

switch

- sans break, l'exécution continue

```
void print_note(char us_note) {
    switch(us_note) {
        case 'a': printf("la\n");
        case 'b': printf("si\n");
        case 'c': printf("do\n");
        case 'd': printf("re\n");
        case 'e': printf("mi\n");
        case 'f': printf("fa\n");
        case 'g': printf("sol\n");
        default: fprintf(stderr,"Error\n");
    }
}
```

```
$>./a.out
d
re
mi
fa
sol
Error
```

switch

- possibilité de factoriser des valeurs

```
/**
 * Prints the latin name of the given
 * note in US notation.
 */
void print_note(char us_note) {
   switch(us_note) {
      case 'a': case 'A': printf("la\n"); break;
      case 'b': case 'B': printf("si\n"); break;
      case 'c': case 'C': printf("do\n"); break;
      case 'd': case 'D': printf("re\n"); break;
      case 'e': case 'E': printf("mi\n"); break;
      case 'f': case 'F': printf("fa\n"); break;
      case 'g': case 'G': printf("sol\n"); break;
      default: fprintf(stderr,"Error: '%c' is not a US note\n",us_note);
   }
}
```

break

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

```
int add_products(int t[],int t2[]) {
   int i,p=1,p2=1;
   for (i=0;i<N;i++) {
      p=p*t[i];
      if (p==0) break;
   }
   for (i=0;i<N;i++) {
      p2=p2*t2[i];
      if (p2==0) break;
   }
   return p+p2;
}
```

break

- à utiliser avec parcimonie
- à éviter en cas de boucles imbriquées
- à éviter si on peut quitter la fonction

```
int product(int m[][N]) {
   int i,j,p=1;
   for (i=0;i<N;i++) {
      for (j=0;j<N;j++) {
         p=p*m[i][j];
         if (p==0) return 0;
      }
   }
   return p;
}
```

continue

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

```c
void print_free_seats() {
   int i,j;
   for (i=1;i<=N_ROWS;i++) {
      if (i==13) {
         /* No seat #13 because of stupid
          * superstitious people */
         continue;
      }
      for (j=0;j<N_SEATS;j++) {
         if (seat[i][j]) printf("%2d%c ",i,j+'A');
         else printf("    ");
         if (j==2) printf("  ");
      }
      printf("\n");
   }
}
```

```
$>./a.out
 1A  1B  1C    1D  1E  1F
 2A  2B  2C    2D  2E  2F
 3A  3B  3C        3E  3F
 4A      4C    4D  4E  4F
 5A  5B  5C    5D  5E  5F
 6A  6B  6C    6D
 7A  7B  7C    7D  7E  7F
 8A  8B  8C    8D  8E  8F
 9A  9B  9C    9D  9E  9F
10A 10B 10C   10D 10E 10F
11A     11C   11D 11E
    12B 12C   12D 12E 12F
14A 14B 14C       14E 14F
```