Langage C

Wieslaw Zielonka
zielonka@irif.fr

Fonctions et vecteurs

```
void echange(unsigned int n, int t[],
             unsigned int i, int j){
   if( i >= n || j >= n )
       return;
   int c;
   c = t[i];
   t[i] = t[j];
   t[j] = c;
}
```

```
int main(void){
   int tab[] = {4, 6, 8, -11, -8, 2};
   int n = sizeof( tab )/ sizeof( tab[0]);
   echange(n, t, 1, 3);
   printf("t[1]=%d t[3]=%d\n", t[1], t[3]);
   return 0;
}
Avant l'appel à echange() :  tab[1]=6, tab[3]=-11
Après l'appel à echange() :  tab[1]=-11, tab[3]=-6
```

préprocesseur -> compilation -> linkage

Quand on lance gcc il y a derrière trois programmes différents qui s'exécutent à tour de rôle :

1. Le **préprocesseurs** transforme le texte du programme en utilisant les directives de préprocesseur commençant par #. Le résultat c'est un fichier texte qui contient le texte de programme mais sans directives de preprocesseur.

2. Le **compilateur** traduit le texte du programme vers un code binaire qui n'est pas encore exécutable. Il manque, par exemple, les fonctions de la bibliothèque standard, comme printf().

3. Le linker rassemble les différentes parties du code binaire, ajoute les références vers les fonctions des bibliothèques. Le résultat est un code binaire exécutable sur la machine.

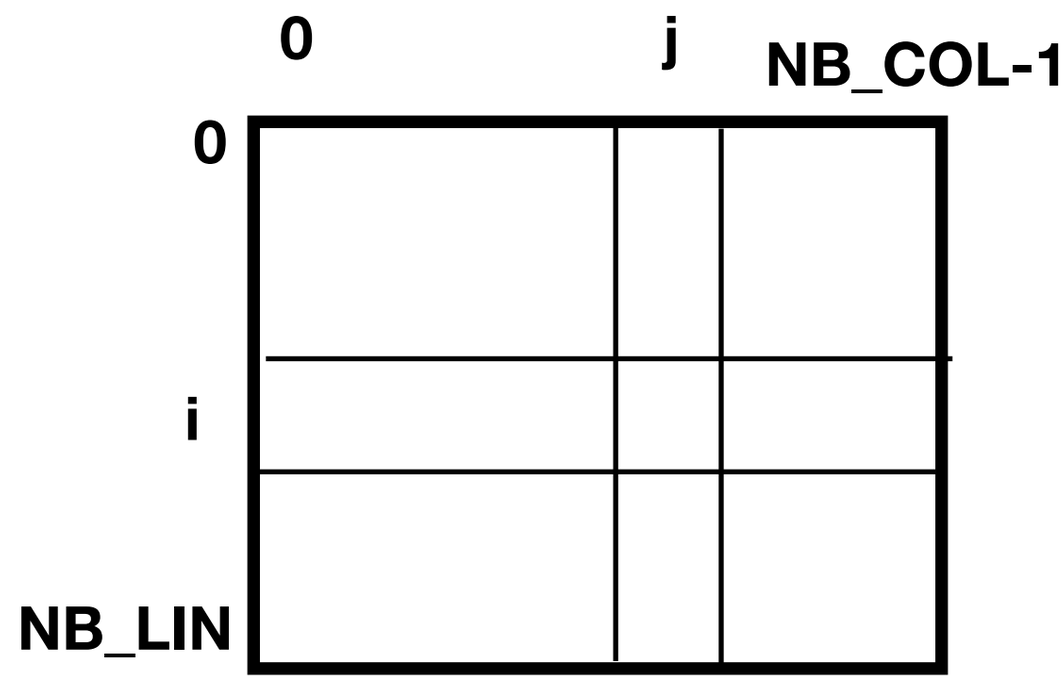
code source

```c
#include <stdio.h>
#define NB_LIN 10
#define NB_COL 20

int fun(int,int);


int main(void){
   double tab[ NB_LIN * NB_COL ];
   int i, j;
   for(int i=0; i < NB_LIN ; i++){
      for( int j = 0; j < NB_COL; j++){
          tab[i * NB_COL + j] = fun(i,j);
      }
   }

.....
```



le fichier texte obtenu à la sorite du préprocesseur

```
#include <stdio.h> remplacé par  le contenu du fichier
stdio.h (plusieurs centaines de lignes).

int main(void){
   double tab[10*20];
   int i, j;
   for(int i=0; i < 10*20; i++){
       tab[i]=0;
   }
   .....
   tab[i * 20 + j] = 20*20 - 10;
```

Le préprocesseur remplace toutes les occurrences des macro-constantes NB_LIN par 10 et les occurrences de NB_COL par 20.

Formatage du code

une déclaration par ligne

~~`int x; int y =1;`~~

```
int  x;
int y = 1;
```

une instruction élémentaire par ligne

~~`i=0; j=10;`~~

```
i=0;
j=10;
```

accolade ouvrante termine la ligne

~~`if(x<y){ t=x; x=y; y=t;}`~~

```
if(x<y){
  t =x;
  x=y;
  y=t;
}
```

```
if(x<y)
{
 t =x;
  x=y;
  y=t;
}
```

Formatage du code

Une seule accolade fermante sur une ligne :

```
}}
```

```
    }
}
```

l'étiquette seule sur une ligne

```
erreur: free(p);
        return 0;
```

```
erreur :
    free( p);
    return 0;
```

`if` et `else` sur la même colonne:

```
if()

else
    if
```

```
if()

else if()
```

Formatage du code

boucle vide, virgule sur la ligne suivante (et commentaire)

```
while( tab[i++] > 0 )
        ; /* boucle vide */

for( i=0; i < NB_ELEM && tab[i]>0 ; i++)
            ; /* boucle vide */
```

Pour obtenir les indentations correctes sous emacs :

- choisir la partie à formater :
    - ctrl-espace pour marquer le début et déplacer le cursor à la fin du code sélectionné
- taper TAB pour formater

les opérateurs à effet de bord

```
int x = 7, y ;

y = ++ x;

printf("x=%d y=%d\n") ; /* affiche x=8 y=8 */
```

L'effet de bord de ++ : incrémentation de x

La valeur de l'expression ++x : 8 (la valeur de x **après** l'incrémentation).

```
x = 7;

y = x ++;

printf("x=%d y=%d\n") ; /* affiche x=8 y=7 */
```

l'effet de bord de ++ : incrémentation de x

la valeur de l'expression x++ : 7 (la valeur de x **avant l'**incrémentation)

expression ternaire

```
(condition) ? val1 : val 2
```

Si condition est vraie la valeur de l'expression est val1 sinon val2

```
int a, b, c;

c = (a < b)? a-1 : b+2;
```

c reçoit la valeur de a-1 si a < b et la valeur de b+2 sinon.

```
printf("%d\n", (a < b)? a : b );
```

affiche plus petit de deux nombres a, b

vecteurs de longueur variable (variable length arrays VLA - optionnel en C11)

C99 a introduit variable length arrays mais C11 a rétrogradé cette possibilité en option. Donc un compilateur conforme à C11 n'est pas obligé d'implementer VLA.

gcc et beaucoup d'autres compilateurs acceptent VLA.

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

Le paramètre qui indiquent la dimension du vecteur doit précéder le vecteur sur la liste de paramètres.

vecteur de longueur variable (VLA - optionnel en C11)

```
double fun(unsigned int n){

   unsigned int k;

   k = 100;

   double tab[ k * n * 2  + 10 ];

}
```

Le nombre d'éléments de tab dépend de k et de n.

Structures

Structures

Vecteurs : pour agréger plusieurs éléments du même type

structures : pour agréger des éléments de types différents.

```
#define LEN 20
struct personne{
   char sex;
   unsigned int annee; /* annee de naissance */
   char  nom[ LEN ];
   char  prenom[ LEN ];
};
```

Notez ; (point-virgule) après chaque champs, y compris le dernier.

```
struct personne beta; /* déclarer la variable beta de type struct
                         personne */

/* déclarer et initialiser une variable de type struct */
struct personne robert = { .sex = 'm',  .nom ="Tituti",
                           .annee = 1956, .prenom = "Robert"};
```

Structures

```
struct point{
   int x;
   int y;
};
struct point p1,p2,p3;

p1.x = 2;
p1.y = -5;
```

Définition de la structure struct point
et la déclaration de trois variables de type struct point.

```
p2.x = - p1.x - 1;

p3 = p1;   /* l'affection est possible entre deux variables
              * de type struct de même type  */
```

Structures

```
struct point{
   int x;
   int y;
};
```

Définition de la structure struct point
et la déclaration de trois variables de type struct point.

```
/* déclaration avec initialisation */
struct point p1 = { .y = 5 , .x = -4 };

struct point p2, p3;

p3 = p1;   /* l'affection est possible entre deux variables
              * de type struct de même type  */

p2 = { .x =-9,   .y = 22 };        /* incorrect */

p2 = (struct point) { .x = -9, .y = 22 }; /* OK*/
```

Structures

```
struct point{
  int x;
  int y;
};
struct point p1,p2,p3;

p1.x = 2;
p1.y = -5;
p2.x = - p1.x - 1;
p2.y = p1.x + p1.y ;
```

```
p3 = p1;
```

Impossible de comparer les valeur de deux variables de type structure à l'aide de ==

~~`if( p1 == p3 )`~~

```
if ( p1x  ==  p3.x && p1.y  ==  p3.y )
```

Structures

```
#define LEN 20
struct personne{
   char sex;
   unsigned int annee; /* annee de naissance */
   char  nom[ LEN ];
   char  prenom[ LEN ];
};

struct personne delta ;

delta = { .sex = 'm',  .nom ="Tituti", .annee = 1956,
          .prenom =  "Vlad"};

delta = (struct personne){ .sex = 'm',  .nom ="Smith", .annee = 1933,

                           .prenom = "Jack"};  /*  OK */
```

Une fois la variable de type struct déclarée on peut changer les valeurs de chaque champs séparément :

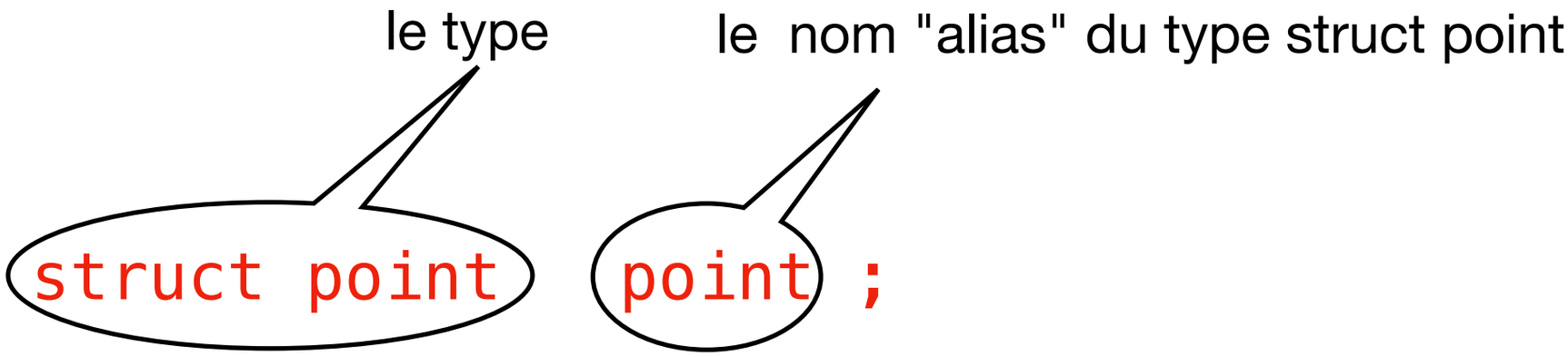
```
delta.sex = 'm';

delta.annee = 1995;

delta.nom[0]='T'; delta.nom[1]='i'; delta.nom[2]='t'; delta.nom[3]='u';
delta.nom[4]='t'; delta.nom[5]='i'; delta.nom[6]='\0';
```

Marre de taper struct? Utilisez typedef

```
struct point{
   int x;
   int y;
} ;
```




```
point p3 = {.x = 3, .y = -7};
```

Possible de définir une structure et le nom alias en même temps:

```
typedef struct point{
int x;
int y;
} point;
```

Marre de taper struct? Utilisez typedef

Et même définir une structure sans nom et le type en même temps :

```
typedef struct{
   int x;
   int y;
} point;
```

```
point p4;
point p5 = {.x = 3, .y = -7};
```

~~`struct point p5;`~~ `/* il n'y a pas de type * struct point */`

explication :

structure anonyme :

`struct { int x; int y };`

`point` est le nom alias de cette structure anonyme

Vecteur de structures

```
typedef struct point{
   int x;
   int y;
} point ;

#define NB_EL  10
point tab_points[ NB_EL ]; /* vecteur de
                              structures */

tab_points[0].x = 2;
tab_points[0].y = tab_points[0].x – 20;

tab_points[ NB_EL –1 ].x = tab_points[ NB_EL – 1] = NB_EL ;
```

Structures dans les structures

```
typedef struct point{
   int x;
   int y;
} point ;

typedef struct rectangle{
     point pa;
     point pb;
} rectangle;

rectangle r = { .pa = { .x = 1, .y = 1 },
                .pb = { .x = 2, .y = 3} };

rectangle d;

d.pa.x = 5; d.pa.y = 5;
d.pb.x = 8; d.pb.y = 23;
```

Structures dans les structures

Deux définitions de types avec les structures anonymes :

```
typedef struct{
   int x;
   int y;
} point ;
```

```
typedef struct{
    point pa;
    point pos;
} rectangle;

/* declarer les variables */
rectangle r;

point p;
```

vecteurs dans les structures

```
typedef struct{
  int x;
  int y;
} point ;
#define NB_MAX 20

typedef struct{
   unsigned int nb_sommets;
   point sommets[NB_MAX];
} polygone;
```

`polygone` utilisé pour mémoriser les sommets d'un polygone. nb_sommets : le nombre de sommets, au maximum 20.

```
polygone triangle, tr;

triangle.nb_sommets = 3;

triangle.sommets[0].x = -1; triangle.sommets[0].y = 0;
triangle.sommets[1].x = 1; triangle.sommets[1].y = 0;
triangle.sommets[2].x = 1; triangle.sommets[2].y = 1;

tr = triangle;

/* OK, on peut faire une affectation entre deux variables de type
structure qui contiennent des vecteurs même si on ne peut pas faire
affectation entre deux vecteurs ! */
```

Structures comme paramètres de fonctions

```
typedef struct{
  int x;
  int y;
} point;

void mirror(point p){
  p.x = -p.x;
  p.y = -p.y;
}

int main(void){
  point p = { .x = 9, .y = -11 };
  printf("p.x=%d p.y=%d\n", p.x, p.y);
  mirror(p);
  printf("p.x=%d p.y=%d\n", p.x, p.y);
  return 0;
}
```

Quelles valeurs affichées par chaque printf()?

Structures comme paramètres de fonctions

```
typedef struct{
  int x;
  int y;
} point;
void mirror(point p){
  p.x = -p.x;
  p.y = -p.y;
}
int main(void){
  point q = { .x = 9, .y = -11 };
  printf("q.x=%d q.y=%d\n", q.x, q.y);
  mirror(q);
  printf("q.x=%d q.y=%d\n", q.x, q.y);
  return 0;
}
```

Quelles valeurs affichées par chaque printf()?

```
q.x=9 q.y=-11
q.x=9 q.y=-11
```

Le paramètre `p` de la fonction `mirror()` est initialisé avec les valeurs de champs qui viennent de la variable `q.` `q` dans `main()` ne sera pas modifié par la fonction `mirror()`.

Structure comme valeur de retour de fonction

```
typedef struct{
  int x;
  int y;
} point;

/* une fonction peut retourner une structure */
point inverser(point p){
  point q = { .x = p.y, .y = p.x };
  return q;
}
```

```
int main(void){
  point p = { .x = 9, .y = -11 };
  printf("p.x = %d p.y = %d\n", p.x, p.y);
  point a = inverser(p);
  printf("a.x = %d a.y = %d\n", a.x, a.y);
  return 0;
}
```

```
p.x = 9 p.y = -11
a.x = -11 a.y = 9
```

Structures et fonctions

```
#include <stdio.h>
#include <math.h>

typedef struct{
  double x;
  double y;
} point;

#define NB_MAX 20
typedef struct{
  unsigned int nb_sommets;        /* le nombre de sommets */
  point sommets[NB_MAX];          /* le tableau de sommets de polygone,
}polygone ;                        *  NB_MAX le nombre maximal de sommets */

double distance(point a, point b){
  return sqrt( (a.x-b.x)*(a.x-b.x)+ (a.y-b.y)*(a.y-b.y) );
}

double perimetre(polygone poly){
  double s = 0;
  for(int i = 0; i < poly.nb_sommets-1; i++){
    s += distance(poly.sommets[i], poly.sommets[i+1]);
  }
  s += distance(poly.sommets[poly.nb_sommets-1], poly.sommets[0]);
  return s;
}
```

Structures et fonctions (cont)

```
int main(void){

  polygone triangle = {
        sommets. = { { .x = -1.0, .y = 0.0 },
          { .x = 1.0, .y = 0.0 },
          { .x = 0.0, .y = 5.0 } },
        .nb_sommets = 3
 };

  double p = perimetre(triangle);
  printf("perimetre = %f\n",p);
  return 0;
}
```

Structures et fonctions

- une structure peut être utilisée comme un paramètre d'une fonction,

- comme pour d'autres paramètres, le paramètre de type structure est un variable locale à la fonction initialisée à l'appel de fonction

- une fonction peut retourner une structure

- en particulier, une fonction peut retourner une structure qui contient un tableau (même quand c'est le seul champ de la structure) alors qu'en C les fonction ne peuvent pas retourner un tableau

Fonctions mathématiques

Les fonctions mathématiques, comme `sqrt()`, sont déclarées dans le fichier en-tête

`math.h`

Mais

```
#include <math.h>
```

n'est pas suffisant.

Il faut ajouter une option -lm :

```
gcc  –Wall  –lm  prog.c – o prog
```

Les fonctions mathématiques se trouvent dans la bibliothèque libm.

-Wall option de compilateur

-lm option de linker (le programme qui ajoute les bibliothèques)

Toutes les autres fonctions du C standard se trouvent dans la bibliothèque glibc qui est automatiquement recherchée par le linker.

les fonctions mathématiques

```
#include <math.h>
```

quelques exemples de fonctions mathématiques :

`sin()`, `cos()`, `asin()`, `sqrt()`, `log()`, `exp()` etc

la page man de math :
man math.h sous linux
man math sous MacOS

enumeration

une enumeration définit un type composé de constantes numériques:

```
enum color{ BLUE , RED , GREEN };

enum color feu; /* déclarer une variable feu de type enum color*/

enum  color autres_feu;

feu = RED ;


if( feu == GREEN ){
}
```

Par défaut les valeurs de constantes dans la liste sont 0,1,2,3 etc. c'est-à-dire

BLUE == 0, RED == 1, GREEN == 2.

La variable `feu` est une variable qui peut prendre une de trois valeurs entières.

Il est possible de spécifier explicitement les valeurs des constantes:

```
enum color{ BLUE = 1 , RED = 2, GREEN = 4 };
```

enumeration

Comme pour structures nous pouvons définir un nom alias

```
enum color{ BLUE =1 , RED = 2, GREEN =
4 } ;
```

```
typedef enum color color;

color c = GREEN;
```

goto

```
#include <stdio.h>
#include <limits.h>
int somme_ligne( int nb_l, int nb_c, int tab[nb_l][nb_c] ){
  int s = 0;
  int i;
  for( i = 0; i < nb_l; i++ ){
    for( int j = 0 ; j < nb_c ; j++ ){
      if( tab[i][j] < 0 )
       goto et;
    }
  }
  return INT_MAX;

 et:
  for( int j = 0 ; j < nb_c ; j++ ){
    s += tab[i][j];
  }
  return  s;
}
```

dans un tableau à 2 dimensions calculer la somme d'éléments de la première ligne qui contient au moins un élément négatif.

goto sert à sortir d'une boucle double

goto

La seule utilisation de goto tolérée dans ce cours c'est pour sortir d'une boucle imbriquée.

digression : tableau à plusieurs dimensions

```
int main(void){
  int  t[][4] = { {1,2,3,11}, {4,-5,6,12}, {-7,-8,-9,13}};
  int r = somme_ligne( 3, 4, t );
  if( r == INT_MAX )
    printf( "lignes nin negatives ?\n");
  else
    printf("somme=%d\n",r);
  return 0;
}
```

```
int  t[ ][ ] = { {1,2,3,11}, {4,-5,6,12}, {-7,-8,-9,13}};
```

incorrect en C

digression : tableau à plusieurs dimensions

```
int  t[][4] = { {1,2,3,11}, {4,-5,6,12},
{-7,-8,-9,13}};
```

Dans ce cours j'utilise uniquement de tableaux à une dimension (les vecteurs).

```
int vect_t[] = {1, 2, 3, 11, 4, -5, 6, 12, -7, -8,
-9, 13 };

/* 3 lignes et 4 colonnes */

int nb_col = 4

vect_t[ i *  nb_col  + j ] équivalent à t[i][j]
```

#include <limits.h>

Le fichier en tête limits.h définit plusieurs constantes symboliques utiles :

SHRT_MAX SHRT_MIN
INT_MAX INT_MIN
LONG_MAX LONG_MIN
UINT_MAX
ULONG_MAX

etc.

#include <stdint.h>

Le fichier en tête stdint.h définit plusieurs types entiers avec le nombre de bits fixe et indépendant de l'architecture :

int8_t

int16_t

int32_t

uint8_t

uint16_t

uint_32_t