Langage C

Wieslaw Zielonka
zielonka@irif.fr

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).

Pour la même valeur de x les valeurs d'expressions ++x et x++ sont différente mais l'effet de bord est le même : incrémentation de x.

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;

d = 2 * ((a<b)? a : b );
```

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

d reçoit plus petite de deux valeurs 2*a ou 2*b

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

`affiche` plus petit de deux nombres a, b

vecteurs en C

En C traditionellement le vecteur avaient la taille fixe (je ne parle pas ici de vecteur comme paramètres de fonctions !):

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

vecteur tab non-initialisé

```
double dt[] = {1.33,  4.9, 5.0, -11.5 };
```

dt : vecteur de 4 éléments double

```
int tp[ 10 ] = {1, 2, 3};
```

tp : vecteur de 10 éléments int dont les trois premiers sont initialisés avec les valeurs indiquées et le 7 suivants sont initialisés à 0.

vecteurs en C

```
#define N 100

int vec[N] = { [6]= 5, [11]= -22, [55] = 4};

Le vecteur vec : les éléments avec les
indices 5, 11, 55 sont initialisés avec les
valeurs respectivement 5,-22,4 , tous les
autres éléments sont initialisés à 0.
```

vecteur de longueur variable (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.

```
double fun(unsigned int n){

    double tab[ 2 * n   + 10 ];



}
```

Le nombre d'éléments de tab dépend de k et de n. tab est un tableau de longueur variable. Il est recommander de ne pas utiliser les VLA.

vecteur et fonctions

Une fonction C ne peut pas retourner un vecteur.

type bool

Valeurs booléennes

Rappel : traditionnellement en C pas de valeurs booléennes :

- une valeur numérique différente de 0 correspond à **true**
- une valeur numérique égale 0 correspond à **false**

```
while( 1 ){
    if(....)
        break;
}
```

Valeurs booléennes

C moderne ajoute le type **bool** avec les valeurs **true** et **false**

```
#include <stdbool.h>

bool b = true;

while( b ){

  ....

  if( ... )

     b = false;

}
```

Mais stdbool.h définit
true comme 1 et false comme 0 donc
les valeurs booléennes sont superflues.

Structures

Structures

Vecteurs : pour agréger plusieurs données de même type

structures : pour agréger des données 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 champ, y compris le dernier et point-virgule qui termine la définition.

```
struct personne eleve; /* déclarer la variable non-initialisée eleve de type
                          struct personne */

eleve.sex = 'F';

eleve.année = 2011;

eleve.nom = "Lopez";

eleve.prenom = "Igor";
```

Structures

Vecteurs : pour agréger plusieurs données de même type

structures : pour agréger des données 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 champ, y compris le dernier et point-virgule qui termine la définition.

```
struct personne eleve; /* déclarer la variable eleve 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 };

/* déclaration de variables p2, p3 non-
initialisées*/
struct point p2, p3;

p2 = { .x =-9,  .y = 22 };      /* incorrect,
l'expression à droite à utiliser uniquement pendant
la déclaration de variable de type struct, comme
pour la variable p1 */

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

Structures : affectation entre les variables

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

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

```
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  */
```

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 ;
```

Pour accéder à un champ d'une variable de type struct on utilise la notation

`variable.champ`

```
p3 = p1;
```

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

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

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

Structures

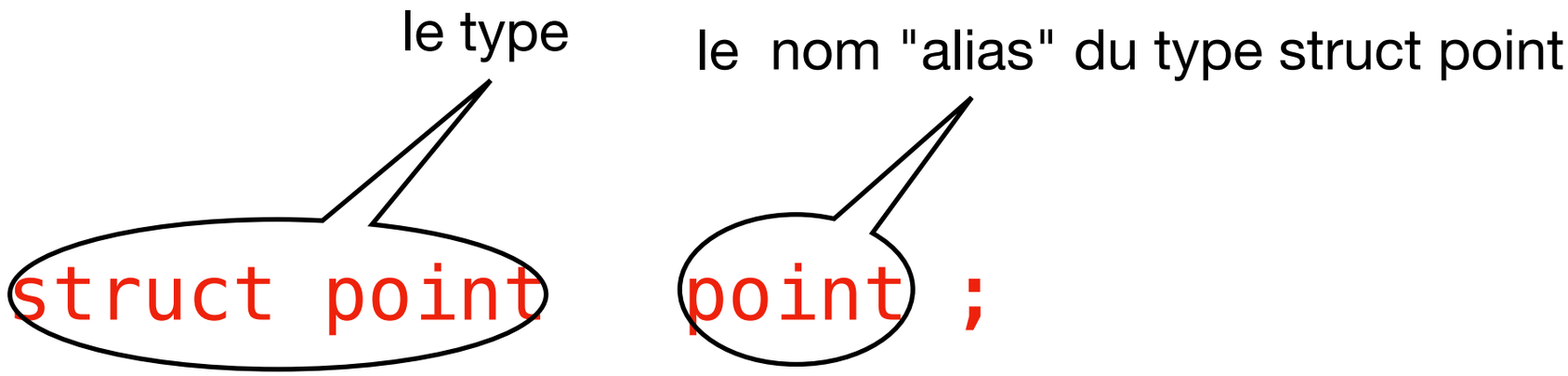
```
#define LEN 20
struct personne{
   char sex;
   unsigned int annee; /* annee de naissance */
   char  nom[ LEN ];
   char  prenom[ LEN ];
};

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 champ 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;
} ;
```




```
typedef struct point   point ;
```

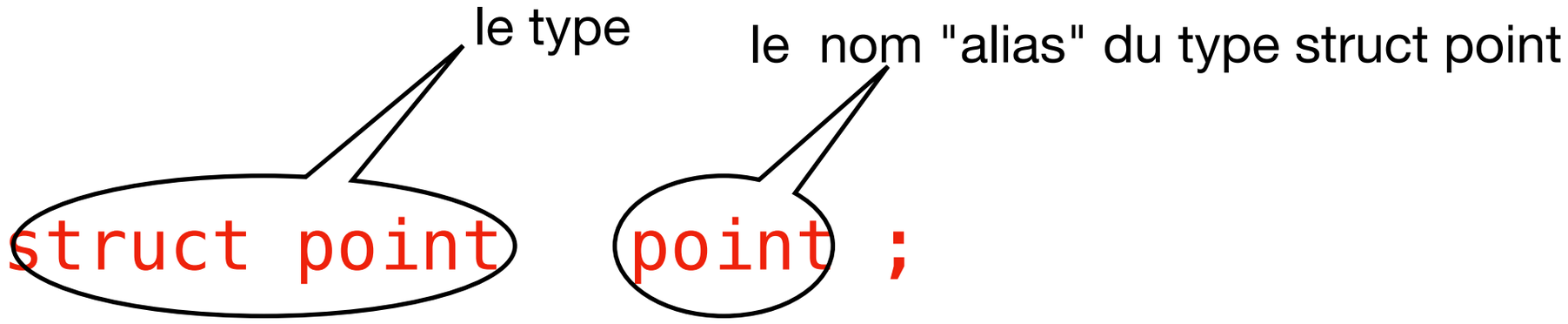
```
point p3 = {.x = 3, .y = -7};
```

`typedef` définit le nom alias "point" pour le type de données "struct point". A partir de ce moment on peut écrire "point" à la place de "struct point".

Le nom alias peut être différent de tag utilisé avec struct.

Marre de taper struct? Utilisez typedef

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




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

struct point p4 = p3;
```

p4 et p3 sont de même type : struct point

Marre de taper struct? Utilisez typedef

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

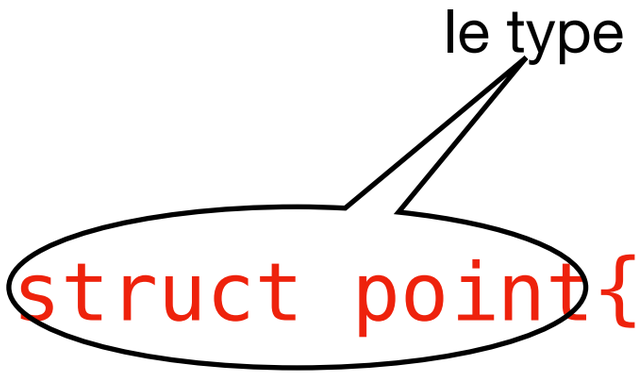
le nom alias peut être différent de tag :

```
typedef struct point  punkt ;

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

struct point p4 = p3;
```

p4 et p3 sont toujours de même type : struct point

Marre de taper struct? Utilisez typedef

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




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

le nom "alias" du type struct point

```
struct point pa;

point pb = pa;
```

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 :
typedef ci-dessus définit le nom alias **`point`** pour une structure anonyme :
`struct { int x; int y; } ;`

Vecteur de structures

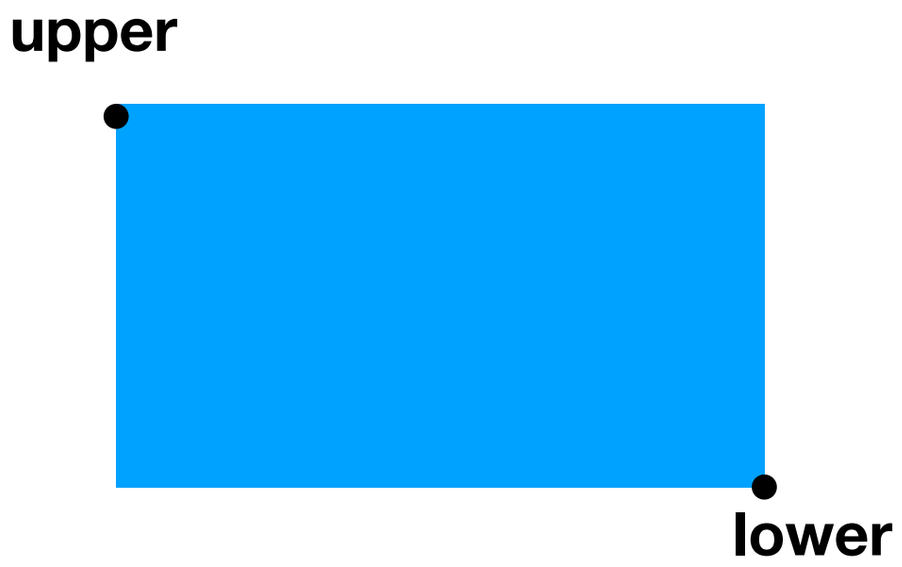
```
typedef struct point{
   int x;
   int y;
} point ;

#define NB_EL  10
point tab_points[ NB_EL ]; /* vecteur de 20
                              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].y
                         = 20 ;
```

Vecteur de structures

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

/* declarer et initialiser un vecteur de
trois structures */
point tab_points[ ] = { { .x=5, .y=-2 },
                        { .x=-4, .y=11 },
                        { .x =-1, .y =9 }
                      }
tab_points[0].y = tab_points[0].x - 20;

tab_points[ NB_EL -1 ].x = tab_points[ NB_EL - 1].y
                         = 20 ;
```

Structures dans les structures

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

typedef struct rectangle{
     point upper;
     point lower;
} rectangle;

rectangle r = { .upper = { .x = 1, .y = 1 },
                .lower = { .x = 2, .y = 3} };

rectangle d;

d.upper.x = 5; d.upper.y = 5;
d.lower.x = 8; d.lower.y = 23;
```



Structures dans les structures

Deux définitions de types avec les structures anonymes :

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

```
typedef struct{
    point upper;
    point lower;
} 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. Les vecteurs comme les champs de structures doivent être de taille fixe.

```
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 ! */
```

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 quadrilatere = { .nb_sommets = 4,

   .sommets = { {.x=1, .y=1}, {.x=1, .y=12},

                { .x=4, .y =1}, {.x=4, .y=15 }

              }

 };  /*  le 4 premiers éléments de vecteur sommets sont

      *   initialisés */
```

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()?

```
premier printf :  q.x=9 q.y=-11
deuxième printf : q.x=9 q.y=-11
```

Le paramètre `p` de la fonction `mirror()` est une variable locale initialisée avec la valeur de q.
La variable `q` dans `main()` ne sera pas modifiée par la fonction.

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;
}
affiche :
p.x == 9 p.y == -11
a.x == -11 a.y == 9
```

```
/* fonction renverser sans variable
 * auxiliaire q */
point inverser(point p){
  return (point){ .x = p.y, .y = p.x };
}
```

Structure comme valeur de retour de fonction

Les fonctions peuvent retourner les structures, même les structures contenant de tableaux.

```
#define MAX_DEG 100000
struct poly{
    unsigned int n;
    double coeff[MAX_DEG];
};
```

```
struct poly creer( unsigned int d ){
    struct poly p = { .n = d };
    for( unsigned int i = 0; i < d; i++ )
        p.coeff[i] = 1;
    return p;
}
```

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 (résumé)

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

- comme pour d'autres paramètres, le paramètre de type structure est une variable locale de la fonction initialisée au moment de 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

enumeration

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 des entiers 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 les structures nous pouvons définir un nom alias

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

typedef enum color color;

color c = GREEN;

enum color d = RED ;

if( d == c ){

}
```

goto

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;
}

int main(void){
```

dans un tableau à 2 dimensions calculer la somme d'éléments de la première ligne qui contient au moins un élément négatif ou retourner INT_MAX si pas de ligne avec 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 non 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]
```

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 à la compilation :

```
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