SCIENCES ET MONTAGNE
UNIVERSITÉ SAVOIE MONT BLANC

UNIVERSITÉ SAVOIE MONT BLANC

INFO 505 - Programmation C II
L3 – 2024-25

CM4 : Structures++, Pointeurs de fonction, compilation avancée

J.Y. RAMEL

Bureau XXX - Polytech'Savoie - Bourget du Lac

Retour sur les structures → Grpmt de bits

Structure classique :

- Type de donnée, contenant un ou plusieurs objets pouvant être de type différent, regroupées au sein d'une même entité.
- Les objets contenus dans la structure sont désignés par champs (de la structure).

Si nécessité d'accéder à des groupements de bits sous la forme de champs nommés.

Utilisation de structures dons la taille des champs :

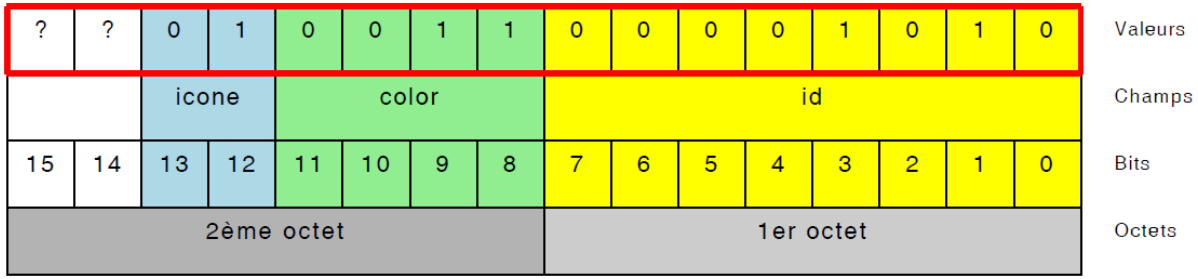
- est définie par l'utilisateur en nombre de bits
- Le type associé est int ou unsigned int (C-ANSI)
- la représentation unsigned étant à privilégier

- occupation memoire = $\left\lceil \frac{\text{nombre total de bits des champs}}{8} \right\rceil$

```
struct MaStructure
{
   signed int id : 8;
   unsigned short color : 4;
   unsigned int icone : 2;
} MaVariable;
MaVariable.icone = 1;
MaVariable.color = 3;
MaVariable.id = 10;
```

Restrictions :

Un champ de bits n'est pas une variable (l'opérateur & n'est pas applicable).



Type énuméré : enum

Type de données permettant d'attribuer des noms à des constantes entières.
Permettre de définir une liste de constantes entières énumérées.

Syntaxe :

```
enum Id_enum {Cte1, Cte2, Cte3 = val, Cte4};
enum {Cte1, Cte2, Cte3 = val, Cte4};
```

Remarques :

Constante entière de type int ;
Par défaut : première valeur = 0 ;
Par défaut : valeur suivante = valeur précédente + 1 ;
Pour chaque constante, il est possible de préciser la valeur par *Id = valeur* ;

Type énuméré : enum

Mise en œuvre :

Identificateur ⇔ Nom du type

enum JOUR {Lundi, Mardi, Mercredi, Jeudi, Vendredi, Samedi, Dimanche};

Constantes (entières) énumérées

```
enum JOUR j;
j = Mardi;
printf("%d\n", j);
```

for (j = Lundi; j <= Dimanche; j++) ici le *code itéré*

Une constante énumérée ne peut être définie qu'une seule fois.

Type énuméré : enum

Redéfinition de type et enum

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

enum color chosenColor = RED;
```

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

color chosenColor = RED;
```

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

typedef enum color color;

color chosenColor  = RED;
enum color defaultColor = BLUE;
```

Exemple Enum + Pointeurs void

Pointeur de type void - Exemple

```
void ajouter( enum Type type, void * result, void * source, int length)
{
  int * pint1, * pint2;
  double * pdouble1, * pdouble2;
  switch (type)
  {
  case Entier : {
                  pint1 = (int*)result;
                  pint2 = (int*)source;
                  while (length--> 0)
                    *pint1++ += *pint2++;
                }
  case Reel :   {
                  pdouble1 = (double*)result;
                  pdouble2 = (double*)source;
                  while (length--> 0)
                    *pdouble1++ += *pdouble2++;
                }
    default : exit(1);
  }
}
```

Type Union

Union

Caractérise une même zone mémoire partagée par des champs de type différent.

Occupation mémoire en nombre d'octets : max {sizeof(champs)}

Syntaxe :

```
union Id_union{
    type1 Id_1;
    type2 Id_2;
    ….
};

union Id_union var1, var2;
Union Id_union var = {valeur de Id_1};
```

```
union RecordType
{
    char   ch;
    int    i;
    long   l;
    double d;
} t;

t.i = 5; // t contient un int
t.d = 7.25; // t contient un double
```

Variables et paramètres constants

const : mot réservé.
Permet de spécifier au moment de la compilation que le contenu d'une variable ne pourra pas être modifié en cours d'exécution du programme.
=> réservation mémoire pour stocker la variable

Notion différente de celle de constante symbolique définie par #define
=> pas de réservation mémoire

Usage 1 : modificateur de type de variable

```
const char c1;              // Le caractère ne peut être modifié
const char * c2;            // Le caractère pointé ne peut être modifié
char * const c3;            // Le pointeur vers le caractère ne peut être modifié
const char * const c4;      // pointeur et caractère pointé ne peuvent être modifiés
```

Usage 2 : modificateur sur paramètre de fonction
Permettre de s'assurer que le contenu d'une variable passée en paramètre d'une fonction ne sera pas modifié au sein de la fonction

Variables et paramètres constants

Exemple 1 : const sur une variable

Erreur de compilation / warning

const int * ptr ⇔ ptr représente le pointeur vers une valeur entière qui après affectation ne pourra plus être modifiée.

En revanche, l'adresse contenu dans la variable p peut être modifiée.

Modificateur_const_1\main.c

```c
/**
 * @brief Usage du modificateur const
 * Erreur d'utilisation
 *  Modification du contenu d'une variable aprés affectation
 */

/* C program to demonstrate that the pointer to point to
    any other integer variable, but the value of the object
   (entity) pointed can not be changed
*/

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int i = 10;
    int j = 20;

    const int* ptr = &i; // ptr is pointer to constant
    printf("ptr: %d\n", *ptr);

    /* error: object pointed cannot be modified
    using the pointer ptr */
    *ptr = 100;

    ptr = &j; /* valid */
    printf("ptr: %d\n", *ptr);

   return EXIT_SUCCESS;
}
```

Variables et paramètres constants

Exemple 2 : paramètre de fonction

Erreur de compilation / warning

Const_Modificateur\main.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/**
 * @brief Utilisation du modificateur const
 *  Erreur de compilation
*/

void fct (const int n);
void length_str (const char *s);

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

void length_str (const char *s){
    s[0] = 'A';
    printf("%lld\n", strlen (s));
}

int main(void) {
    int n = 10;
    char s[] = "Une chaine";

    fct (n);
    length_str (s);
    return EXIT_SUCCESS;
}
```

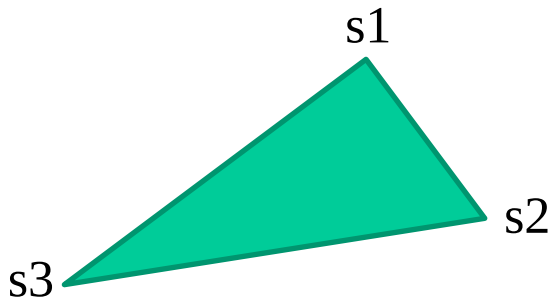
Tableaux et structures

Rappel

- Tableau => ensemble d'emplacements mémoire contigus, portant le même nom, et contenant le même type de données

- Conséquence : les éléments d'un tableau peuvent (aussi) être des structures

Exemple :
Gestion de 4 triangles à l'aide d'un tableau



3 sommets : s1, s2, s3;
1. chaque sommet : défini par ses coordonnées 2D (ou 3D);
2. triangle : collection de trois sommets;
3. un point d'entrée unique pour gérer 4 triangles : tableau

Tableaux et structures

```
struct point_2D {
   double x;
   double y;
};
```

```
struct triangle {
   struct point_2D sommet1;
   struct point_2D sommet2;
   struct point_2D sommet3;
} ;
```

Un tableau pour manipuler 4 triangles
`struct triangle Ensemble_tgl[4];`

```
Ensemble_tgl[0].sommet1.x = 1.0;
Ensemble_tgl[0].sommet1.y = 1.0;
Ensemble_tgl[0].sommet2.x = 2.0;
Ensemble_tgl[0].sommet2.y = 2.0;
Ensemble_tgl[0].sommet3.x = 3.0;
Ensemble_tgl[0].sommet3.y = 3.0;
```

Mais aussi :

```
struct triangle tab_triangle[4] = {{{1.0, 1.0}, {2.0, 2.0}, {3.0, 3.0}},
                                   {{4.0, 4.0}, {5.0, 5.0}, {6.0, 6.0}},
                                   {{7.0, 7.0}, {8.0, 8.0}, {9.0, 9.0}},
                                   {{10.0, 10.0}, {11.0, 11.0}, {12.0, 12.0}}};
```

Tableaux 2D

Déclaration

type Id_tableau[nb_ligne][nb_colonne];

type : le type de données de tous les éléments du tableau
Id_tableau : nom du tableau (identificateur au sens du langage C)
nb_ligne : nombre de lignes du tableau (constante littérale ou symbolique de type entier)
nb_colonne : nombre de colonnes du tableau (constante littérale ou symbolique de type entier)

Un tableau de dimension 2 correspond à une architecture de **matrice**

Exemple : tableau de 3 lignes et 4 colonnes, contenant des réels

float tab_reel [3][4] ;

Tableaux 2D

Occupation mémoire

- type Id_tableau[nb_ligne][nb_colonne];

- sizeof (type) * nb_ligne * nb_colonne
- sizeof (Id_tableau)

Accès à l'élément d'un tableau 2D

- Id_tableau[indice_ligne][indice_colonne]

- Indices du premier élément : toujours [0][0] ;
- Indices du dernier élément : toujours [nb_ligne – 1][nb_colonne – 1]

Tableaux 2D

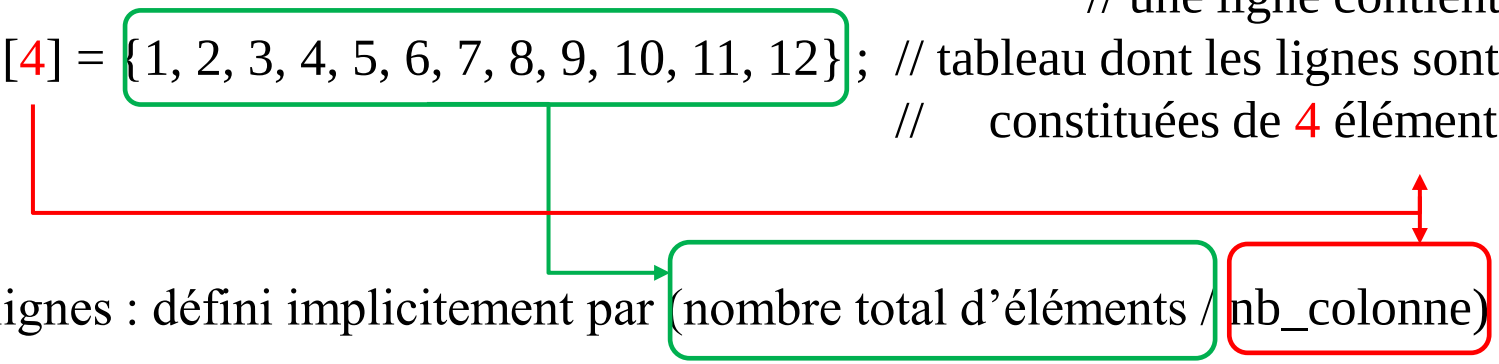
initialisation des éléments

type Id_tableau[nb_ligne][nb_colonne];

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12

float tab_reel [3][4] ;
tab_reel [1][2] = 3.14 /*l'élément de la 2ème ligne et de la 3éme colonne reçoit la valeur 3.14 */

int tab_int [3][4] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12} ; // tableau de 12 éléments
int tab_int [3][4] = {{1, 2, 3, 4}, {5, 6, 7, 8, 9}, {10, 11, 12}} ; // tableau de 3 lignes
// une ligne contient 4 éléments
int tab_int [][4] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}; // tableau dont les lignes sont
// constituées de 4 éléments



Nombre de lignes : défini implicitement par (nombre total d'éléments / nb_colonne)

Tableaux 2D

Tableau 2D et pointeurs

type Id_tableau[nb_ligne][nb_colonne];

Un tableau n'est pas un pointeur !
- L'identificateur d'un tableau *Id_tableau* correspond au pointeur sur le premier élément du tableau dont les éléments sont du type *type*

Conséquence 1 :
- Id_tableau ⇔ &Id_tableau [0] [0]

- Id_tableau représente l'adresse à laquelle le premier élément du tableau est stocké

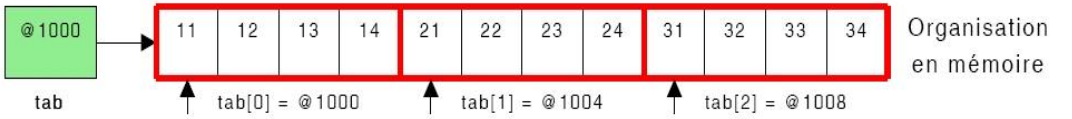
Tableaux 2D

Tableau 2D et pointeurs

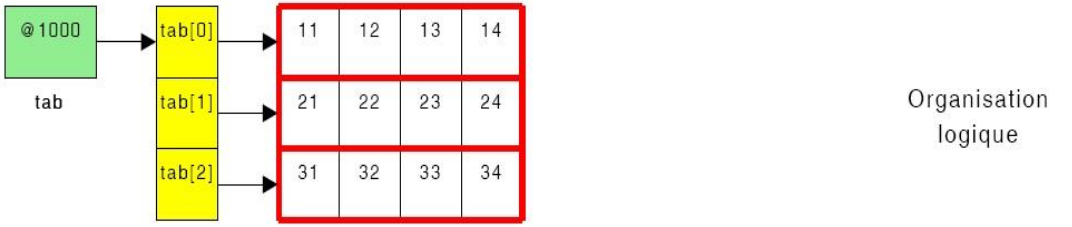
```
char tab[3][4] = {11, 12, 13, 14, 21, 22, 23, 24, 31, 32, 33, 34};
```

	0	1	2	3
0	11	12	13	14
1	21	22	23	24
2	31	32	33	34

Un tableau 2D est une collection d'objets identiques, occupant des emplacement mémoire adjacents.



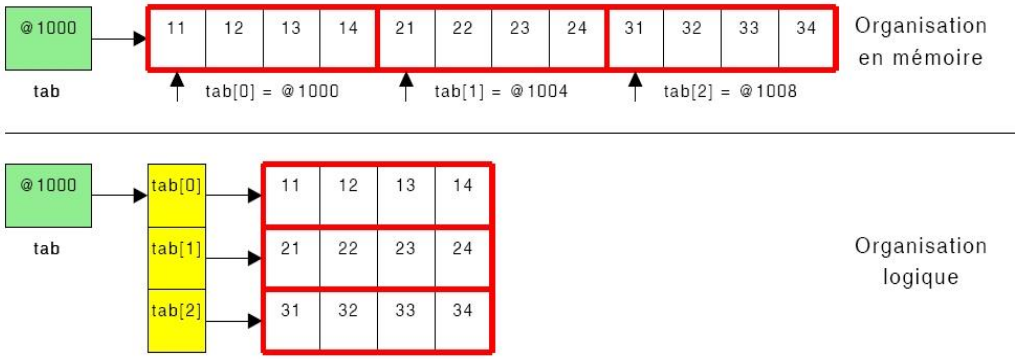
Un tableau 2D est un tableau 1D dont les éléments sont eux-mêmes des tableaux 1D d'objets identiques, occupant des emplacement mémoire adjacents.



Tableaux 2D

Tableau 2D et pointeurs

char tab[3][4] = {11, 12, 13, 14, 21, 22, 23, 24, 31, 32, 33, 34};



Conséquence 2 :

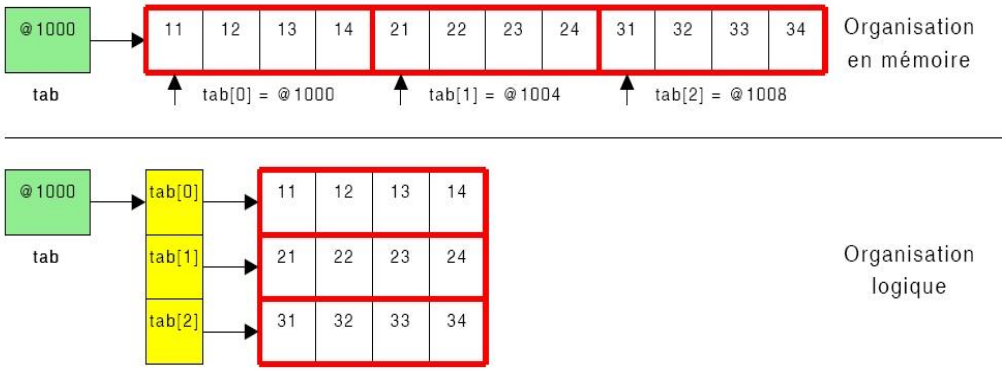
- tab ⇔ &tab[0] ⇔ &tab[0][0] // Adresse du premier élément du tableau

- tab[0][0] ⇔*tab[0] ⇔ * * tab // Valeur du premier élément du tableau

Tableaux 2D

Tableau 2D et arithmétique des pointeurs

char tab[3][4] = {11, 12, 13, 14, 21, 22, 23, 24, 31, 32, 33, 34};



Accéder à tab[i][j]

tab + i /* pointeur sur le premier octet de la (i+1)ème ligne du tableau tab */

(tab+i) / valeur du premier élément de la (i+1)ème ligne du tableau tab */

tab[i][j] ⇔ *(tab[i] + j) ⇔ *(*(tab +i) + j)

Tableaux 2D

Tableau 2D et allocation dynamique

Création en cours d'exécution d'un programme d'un tableau 2D de *nb_ligne* lignes et *nb_colonne* colonnes, et contenant des données de type *type*.

=> type Id_tableau[nb_ligne][nb_colonne];

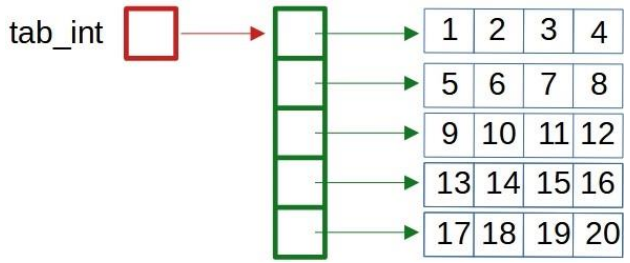
Approche 1 : allouer (nb_ligne * nb_colonne * sizeof (type)) octets

```
type **tab_2D;
tab_2d = (type **) malloc (nb_ligne * nb_colonne * sizeof (type);
if (tab_2D == NULL) exit (-1);
Tab_2D[i][j]          // accéder à l'élément d'indice [i][j] du tableau Tab_2D
```

Approche 2 :

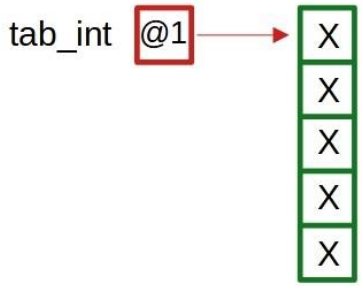
1. allouer un tableau 1D de nb_ligne pointeurs sur *type*;
2. Pour chaque ligne, allouer un tableau de nb_colonnes éléments sur *type*

Exercice : créer un tableau 2D de 5 lignes et 4 colonnes contenant des réels de type double



Tableaux 2D

1 Allouer un tableau 1D de nb_ligne pointeurs sur des *int*;

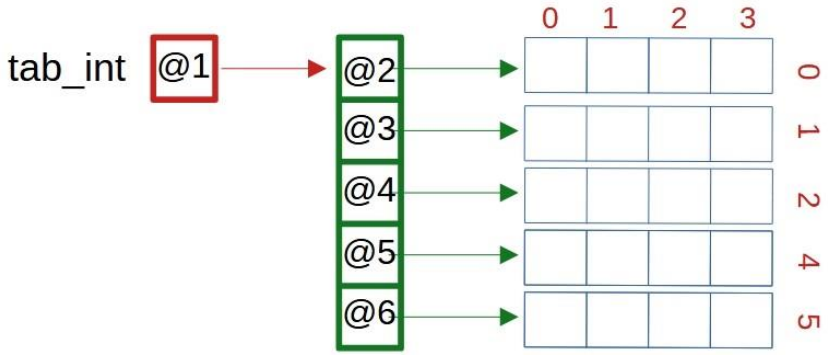



```
int ** ptr_tab;
ptr_tab = (int **) malloc (nb_lig * sizeof (int *));
if (ptr_tab == NULL) exit (-1);
```

2 Pour chaque ligne, allouer un tableau de nb_colonnes éléments sur *type*

Pour i variant de 0 à nb_lig – 1 :

```
ptr_tab[i] = (int *) malloc (nb_colonnes * sizeof (type));
If (ptr_tab[i] == NULL) exit (-2);
```



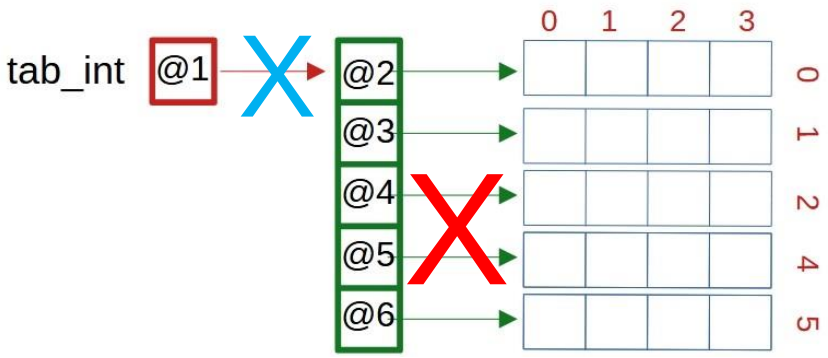
Tableaux 2D

int **tab_int[nb_ligne][nb_colonne];**

3 Libération de la place mémoire (si l'ensemble des allocations effectuées avec malloc se sont exécutées correctement)

Pour i variant de 0 à nb_lig – 1 :
free (ptr_tab[i]);

4 free ptr_tab;

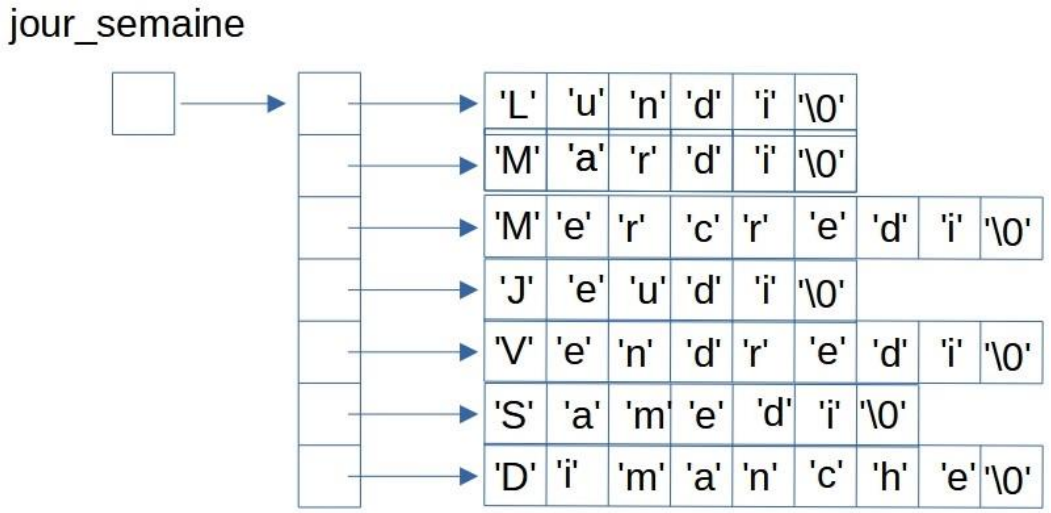


Tableaux 2D

Tableau 2D et chaines de caractères

Illustration :

```
char *jour_semaine [7] = {
  ' ' Lundi  ' ',
  ' ' Mardi' ',
  ' ' Mercredi' ',
  ' ' Jeudi' ',
  ' ' Vendredi' ',
  ' ' Samedi' ',
  ' ' Dimanche' ',
};
```



jour_semaine[2] ⇔ ' ' Mercredi' '
jour_semaine[2][3] ⇔’c’

Pointeurs de fonction

Objectif

– appel de fonctions plus fléxible
– pouvoir transmettre le nom d'une fonction à appeler à une autre fonction

Règle générale

Déclaration similaire à un **pointeur** sur variable mais **entourée de parenthèses**
Le nom d'une fonction constitue le pointeur sur la fonction (principe similaire aux tableaux)

Déclaration

```
TypeRetourné (*NomDeLaVariable)(ListeDesParamètres);
```

Typedef possible

```
typedef TypeRetourné (*NomDuType)(ListeDesParamètres);
```

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

int calculer1()
{
  return 0;
}

int calculer2()
{
  return rand();
}

void afficher1( int x )
{
  printf( "Tatam ! x=%d\n", x );
}

void afficher2( int x )
{
  printf( "*** %d ***\n", x );
}

int main( void )
{
  int (*sansargument)( void );
  void (*avecargument)( int x );

  sansargument = calculer1;
  avecargument = afficher2;

  avevargument(sansargument());

  return 0;
}
```

Pointeurs de fonction

```c
#include <stdio.h>
#include <stdlib.h>

typedef int (*Traitement)(int index, int v, void * data);

int appliquer( int * liste, int N, Traitement t, void * data )
{
  int res = 0, i;
  for( i = 0; i < N; ++i )
    if (t(i, liste[i], data))
      res = 1;
  return res;
}

int plusPetit( int index, int v, void * data )
{
  if (index == 0)
    *(int*)data = v;
  else
    if (*(int*)data > v )
      *(int*)data = v;
  return 1;
}

int existe( int index, int v, void * data )
{
  if (*(int*)data == v)
    return 1;
  else
    return 0;
}

int afficher( int index, int v, void * data )
{
  printf( "%d => %d\n", index, v );
}
```

```c
{
  int liste[100];
  int i, min, v;

  for( i = 0; i < 100; ++i )
    liste[i] = rand();

  if (appliquer(liste, 100, plusPetit, &min))
    printf( "Le plus petit élément est : %d\n", min );

  v = 25;
  if (appliquer(liste, 100, existe, &v))
    puts( "25 est dans la liste\n");
  else
    puts( "25 n'est pas dans la liste\n");

  appliquer(liste, 100, afficher, NULL);

  return 0;
}
```

Fonctions variadiques

Fonctions à nombre d'arguments variables

Rem : printf n'a pas toujours le même nombre d'arguments → C'est une fonction variadique !

Contraintes

– La fonction doit avoir au moins 1 argument fixe
– Il faut utiliser le fichier d'entête **stdarg.h** pour :
- Déclarer une variable **X** du type → **va_list X;**
– Initialiser cette variable X avec **va_start (X , NomDernierParam)**
- Récupérer l'argument courant du type *TypeArgument* et se déplacer sur l'argument suivant
avec **maval = va_arg (X,** *TypeArgument* **);**
– Terminer le traitement avec **va_end(X);**

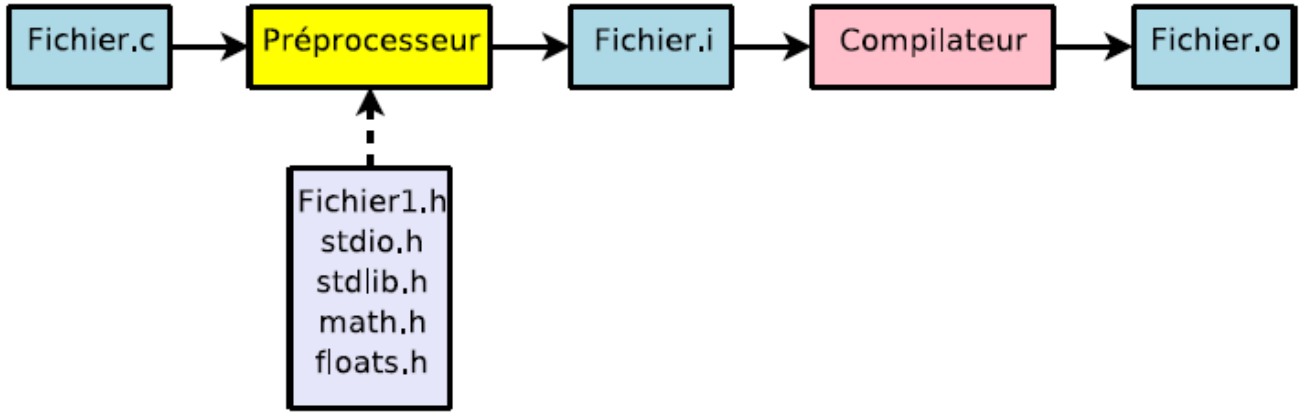
Remarques importantes

– Les paramètres optionnels (...) sont promus automatiquement
– Les entiers sont systématiquement promus en int sauf si la taille du type est plus grande (⇒pas de convertion)
– Les réels sont systématiquement promus en double sauf si la taille du type est plus grande (⇒ pas de convertion)
– char, short, float ne peuvent donc pas être utilisé avec va_arg

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

void AfficherArg( int p1, int p2, ...)
{
  int arg, count = 0;
  va_list ap;
  printf( "%d => %d\n", 1, p1 );
  printf( "%d => %d\n", 2, p2 );
  count = 3;
  va_start(ap, p2);
  while ((arg = va_arg(ap, int)) != -1)
    printf( "%d => %d\n", count++, arg );
  va_end(ap);
}
int main(void)
{
  AfficherArg( 1, 2, 3, 4, 5, -1 );
  AfficherArg( 6, 2, 1, -2, -1 );
  return 0;
}
```

Structuration du code avec le préprocesseur



#include → Inclure un fichier de déclaration (un fichier .h).

Deux syntaxes supportées :

Pour les fichiers d'en-tête standard (stdio.h,…) situés dans le répertoire des Include par défaut :

#include <stdio.h>

Pour les fichiers d'en-tête utilisateur, placés dans le projet :

#include '' Nom_Fichier.h ''

Structuration du code avec le préprocesseur

#define → permet de définir des constantes ou des macros

#define TAILLE_TABLEAU 100
#define __DEBUG__
Permettre de définir un symbole sans lui attribuer de valeur
Utiliser dans le cadre de la compilation conditionnelle

Il existe des constantes prédéfinies.

Nom	Valeur de la **macro**	Forme syntaxique
__LINE__	numéro de la ligne courante du programme source	entier
__FILE__	nom du fichier source en cours de compilation	chaîne
__DATE__	la date de la compilation	chaîne
__TIME__	l'heure de la compilation	chaîne

Structuration du code avec le préprocesseur

#define (instruction du préprocesseur)

```
#define NomMacro( Liste des paramètres) Expression
```

Exemples

Attention

Pour éviter les effets indésirables, les paramètres doivent être placés entre parenthèses dans l'expression

```
#define Moitier( x ) (x)/2
#define Carre( x ) (x)*(x)
...
resultat = Moitier( 10 );
resultat = Moitier( x[1] + x[2] );
resultat = Carre( x+y );
```

```
#define PlusGrand( x, y ) ( (x)>(y)?(x):(y) )
```

```
#define ABS( valeur) valeur<0?-valeur:valeur
...
x = ABS(10-20);
```

Structuration du code avec le préprocesseur

#define (instruction du préprocesseur)

Macro paramétrées.

```
#define BONJOUR puts("Bonjour !");

#define MIN (X, Y)  ((X) < (Y) ? (X) : (Y))
#define MAX (X, Y) ((X) > (Y) ? (X) : (Y))

 x = min(a, b);          →  x = ((a) < (b) ? (a) : (b));
 y = min(1, 2);          →  y = ((1) < (2) ? (1) : (2));
 z = min(a + 28, *p);    →  z = ((a + 28) < (*p) ? (a + 28) : (*p));
```

Rq : attention à la priorité des opérateurs ainsi qu'aux effets de bord

Structuration du code avec le préprocesseur

Test\main.c

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define T_STRING 128

int main(void)
{
    char s_file[T_STRING], s_date[T_STRING];
    char s_time[T_STRING];

    memset(s_file, 0, sizeof(char));
    memset(s_date, 0, sizeof(char));
    memset(s_time, 0, sizeof(char));

    strcat (s_file, __FILE__);
    strcat (s_date, __DATE__);
    strcat (s_time, __TIME__);

    printf("l : %d  ", __LINE__); printf ("s_file : %s\n", s_file);
    printf("l : %d  ", __LINE__); printf ("s_date : %s\n", s_date);
    printf("l : %d  ", __LINE__);printf ("s_time : %s\n", s_time);


    printf("\nhello world\n");
    return EXIT_SUCCESS;
}
```

```
d:\Documents\ProgrammationLangageC\build-Debug\bin\Test.exe
l : 19  s_file : d:/Documents/ProgrammationLangageC/Test/main.c
l : 20  s_date : Jul 10 2024
l : 21  s_time : 14:36:33

hello world

==== Program exited with exit code: 0 ====
Time elapsed: 000:00.031 (MM:SS.MS)
Press any key to continue...
```

Structuration du code avec le préprocesseur

Compilation conditionnelle

#if, #if defined, #ifdef, #if not defined, #ifndef, #endif

Permet d'effectuer de la compilation conditionnelle selon qu'une macro est définie ou non.

Compilation conditionnelle : permettre de choisir quels seront les blocs de code qui seront considérés lors de la compilation.

#if defined et #ifdef sont équivalents
#if not defined et #ifndef sont équivalents

#else et #elsif permettent d'implémenter des directives du préprocesseur *si alors sinon*

Toutes directives #if se termine toujours par #endif

```
#if GB == 1
  #include "GB.h"
#elif FR == 1
  #include "FR.h"
#else
  #include "US.h"
#endif
```

```
#if TAILLEINT == 32
  typedef short int16;
  typedef int int32;
#else
  typedef int int16;
  typedef long int32;
#endif
```

```
#if defined(DEBUG)
  printf( "Debug : x = %d \n", x );
#endif
#ifdef DEBUG
  printf( "Debug : x = %d \n", x );
#endif
```

Pour activer ces instructions : **#define** DEBUG

Structuration du code avec le préprocesseur

Structuration du code → header guard

Cas de la compilation séparée de fichiers source (.c) contenant des directives multiples #include d'un même fichier d'en-tête (.h)

Erreur de compilation possible : redéfinition d'un type de données

header guard – include guard (cf. https://fr.wikipedia.org/wiki/Include_guard)

« un #include guard, parfois appelé macro guard ou encore garde-fou, est une construction utilisée afin d'éviter le problème de la *double inclusion* pouvant apparaitre avec l'utilisation des directives d'inclusion.

Structuration du code avec le préprocesseur

Structuration du code → header guard

Mise en œuvre :

- directive #ifndef
- directive #define
- directive #endif

au sein de chaque fichier d'en-tête .h

```
ifndef _MYHEADER_
#define _MYHEADER_

… code…

#endif
```

Gestion des erreurs

Erreurs de la bibliothèque C standard

extern int errno

- Utilisent un mécanisme simple pour indiquer des erreurs.

– Deux cas de figure :

– La fonction retourne une valeur indiquant qu'une erreur est survenue. La variable errno donne le détail de l'erreur.

– La fonction ne peut pas indiquer qu'une erreur est survenue en retournant une valeur. Elle utilise néanmoins errno pour indiquer qu'il y a eu une erreur.

Remarque : La variable errno n'est jamais mise à 0. Il est donc nécessaire de la mettre à zeno soit même.

Affichage message d'erreur

- 2 fonctions :

```
char *strerror (int errnum);
void perror (const char *s);
```

- strerror() renvoie une chaîne de caractères décrivant l'erreur associée au code d'erreur
- perror() affiche la chaîne s suivie de : puis le message d'erreur associé à errno.

Gestion des erreurs

```c
#include <stdio.h>
#include <errno.h>   // Pour errno
#include <string.h>  // Pour strerror()

int main() {
    FILE *file;

    // Tentative d'ouverture d'un fichier qui n'existe pas
    file = fopen("nonexistent_file.txt", "r");

    // Vérification si l'ouverture a échoué
    if (file == NULL) {
        // Affiche le code d'erreur et le message correspondant
        printf("Erreur lors de l'ouverture du fichier. Code d'erreur: %d\n", errno);
        printf("Message d'erreur: %s\n", strerror(errno));

        // Ou utilisation de perror qui combine les deux
        perror("Erreur avec fopen");

        return 1;  // Indiquer que le programme s'est terminé avec une erreur
    }

    // Si le fichier est ouvert avec succès (ce qui ne sera pas le cas ici)
    fclose(file);

    return 0;  // Succès
}
```