École Centrale
de Marseille

Programmation Structurée en Langage C - 1re année

Stéphane Derrode, Bureau 206 (Institut Fresnel),
stephane.derrode@ec-marseille.fr

Pôle Mathématique et Informatique

Sommaire

Le langage C

- B.W. Kernighan et D.M. Ritchie : *Le Langage C ANSI*, 1977 ;
- Langage « universel » : disponible sur toutes les plate-formes (Unix, Linux, Windows, ...) ;
- Langage de « 2e génération », comme Pascal / Delphi ;
- Langage structuré (modulaire) et impératif :
  - Types de variables : Nombres entiers, nombres réels, caractères
  - Structures de contrôle : Si ... alors, boucles, séquences
  - Construction de types : Tableaux, structures, enregistrements
  - Sous-programmes : Fonctions
- Langage compilé (≠ interprété) ;
- Successeur : langage orienté objet C++ (OG de 2e A).

Premiers programmes

1 – Ne fait rien

```c
/*
   Programme minimum
   ne faisant rien
*/

int main()
  {
      return 0;
  }
```

2 – « Hello World ! »

```c
#include <stdio.h>

int main()
  {
    // Affiche le message
    printf("\nHello World!");

    return 0;
  }
```

démo 1

Ligne de compilation : cf. Tr. 105

Principales bases

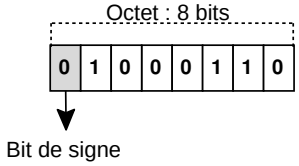
Bases	Notations	Symboles
2	binaire	0, 1
8	octale	0, 1, ..., 7
10	décimale	0, 1, ..., 9
16	hexadécimal	0, 1, ..., 9, a, b, c, d, e, f

décimale	binaire	octale	hexadécimal
70	1000110	106	46
511	111111111	777	1FF

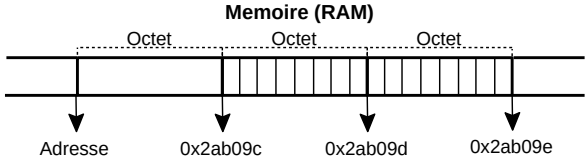
$$70 = 0 * 2^0 + 1 * 2^1 + 1 * 2^2 + ... + 1 * 2^6, 70 = 6 * 16^0 + 4 * 16^1$$

Élément mémoire : bit et octet

- Un octet (ou « byte ») est composé de 8 « bits »,
- Un bit est un élément mémoire de base : valeurs 0 ou 1.
- Un octet : $2^8 = 256$ possibilités, valeurs de -128 à +127.



- Mémoire (RAM) : suite d'octets adressés.



Les différents types de variables

➽ En informatique, on fait la distinction entre les variables contenant des entiers, des nombres réels, des tableaux (vecteurs) de nombres, ...

➽ On peut distinguer 5 types de variables :

- **Types entiers** : nombres entiers et caractères - Tr. 8.
- **Types flottants** : nombre réels - Tr. 10.
- **Types dérivés** : tableaux (chaînes de caractères) - Tr. 21.
- **Types pointeurs** : adresse mémoire - Tr. 43.
- **Types complexes** : structures (unions et énumérations) - Tr. 66.

Types entiers

➽ **3 types entiers**

- `char` : `'a'` (1 octet), de `-128` à `+127`. ➪ Tr. 9
- `short int` : `-10814` (2 octets), de `-32768` à `+32767`.
- `int`, `long int` : `158141234` (4 octets), de ... à ...

Qualificatifs : `signed` (défaut) ou `unsigned`.

➽ **Déclaration, initialisation de variables**

Déclaration	Initialisation	Les deux
`int i ;`	`i=-5 ;`	`unsigned int k=5, j=2 ;`
`unsigned char c ;`	`c='s' ;` (`c=115 ;`)	`char v = 168 ;`

➽ **Constantes** : `const unsigned char c='S' ; const int dix=10 ;`

Type char

- Entre côtes : `'a'`, `'z'`, `'B'`
  - Table des codes ASCII (annexe du poly. de langage C).
- Valeur de type entier
  - `'A'` à `'Z'` : caractères majuscule → code ASCII : `65` à `90`
  - `'a'` à `'z'` : caractères minuscule → code ASCII : `97` à `122`
  - `'0'` à `'9'` : caractères chiffres → code ASCII : `48` à `57`
- Caractères spéciaux :
  - `'\''` : apostrophe → code ASCII : `39`
  - `'\\'` : antislash → code ASCII : `92`
  - `'\t'` : tabulation → code ASCII : `9`
  - `'\n'` : saut de ligne → code ASCII : `10`
  - `'\0'` : terminaison de chaîne → code ASCII : `0`

Types flottants

- **3 types flottants**
  - `float` : `125.46` (4 octets), de -10^{38} à $+10^{38}$.
  - `double`,`long double` : `1.55e-6`, (8 octets), de -10^{308} à $+10^{308}$.

  Qualificatifs `signed` ou `unsigned` **interdits** !!!
- **Déclaration, initialisation de variables**

Déclaration	Initialisation	Les deux ensemble
`float v ;`	`v=-5.8975 ;`	`float w=4.25, z=-5E-5 ;`
`double x ;`	`x=+5.23e-12 ;`	`double y = -168E2 ;`

- **Variable à valeur constante** : `const double PI=3.14 ;`

Lire / Écrire (Input/Output)

Librairie standard : `#include<stdio.h>`

`printf()` : Le programme affiche des résultats sur la console.

- `int i=5 ; printf("La valeur de i est %d", i) ;`
- `char c='A' ;`
  `printf("Caractere : %c, code ASCII : %d", c, c) ;`
- `char ch[9]="hello" ;`
  `printf("message : %s", ch) ;`

`scanf()` : L'utilisateur donne des informations au programme.

- `float reel ; scanf("%f", &reel) ;`
- `float r1,r2,r3 ;`
  `printf("Entrez trois nombres réels :") ;`
  `scanf("%f%f%f", &r1, &r2, &r3) ;`

Exemples : cf. poly Langage C, Chapitre 3. démo 2

Déclaration	Lecture	Écriture	Format externe
`int i ;`	`scanf("%d",&i) ;`	`printf("%d",i) ;`	décimal
`int i ;`	`scanf("%o",&i) ;`	`printf("%o",i) ;`	octal
`int i ;`	`scanf("%x",&i) ;`	`printf("%x",i) ;`	hexadécimal
`float m ;`	`scanf("%f",&m) ;`	`printf("%f",m) ;`	point décimal
`double m ;`	`scanf("%lf",&m) ;`	`printf("%lf",m) ;`	point décimal
`double m ;`	`scanf("%le",&m) ;`	`printf("%le",m) ;`	exponentielle
`double m ;`	`scanf("%lg",&m) ;`	`printf("%lg",m) ;`	la + courte
`char o ;`	`scanf("%c",&o) ;`	`printf("%c",o) ;`	caractère
`char p[10] ;`	`scanf("%s",p) ;`	`printf("%s",p) ;`	chaîne de caractères

Exemples de printf() et scanf()

```c
#include <stdio.h>
int main() {
    int a = 5;
    int b = a;

    a = 8;
    printf("a = %d, b = %d", a, b);
    return 0;
}
```


démo 3

```c
#include <stdio.h>
int main() {
    char tt[80];    // Tableau de 80 caractères
    printf("Donnez une chaine de caracteres : ");
    scanf("%s", tt);
    printf("\nLa chaine entree est : %s\n", tt);
    return 0;
}
```

```c
#include <stdio.h>
int main() {
  int i=10;
  float l=3.14159;
  char p[50]="Bonjour";

  printf("Avant lecture au clavier : %d %f %s\n",i,l,p);
  scanf("%d%f%s",&i,&l,p);
  printf("Apres lecture au clavier : %d %f %s\n",i,l,p);
  return 0;
}
```


Opérateurs arithmétiques

Par ordre de priorité croissante

- `-`, `+` : unaire (inverse/confirme le signe). Ex : `-a`, `+5.6`
- `+`, `-` : addition et soustraction. Ex : `6+5`, `a-5.6`
- `*`, `/` : multiplication et division. Ex : `6*5`, `a/5.6`
- `%` : reste de la division entière. Ex `7%2` donne `1` ($7 = 3*2+1$)

démo 4

```c
#include <stdio.h>
int main() {
  int i=10, j=-3;
  int k = i-j;
  printf("\nk=%d, calcul=%d\n", k, (i*i)%(-j));
  return 0;
}
```

Librairie mathématique : `#include<math.h>` : cos, sin, tan, ... démo 5

```
#include <stdio.h>
#include <math.h>
int main() {
   double a;
   printf("\nEntrez un angle : ");
   scanf("%lf", &a);

   printf("Le cos de l'angle vaut : %lf\n", cos(a));
   printf("Son logarithme vaut    : %lf\n", log10(a));
   return 0;
}
```

Attention : Conversion implicite et explicite (≪ cast ≫)

- `float f = 2 / 4;` ➠ `f = 0.0`
- `float f = 2.0 / 4;` ➠ `f = 0.5`
- `int i = 2.0 / 4;` ➠ `i = 0`
- `float j = (float) 3/2;` ➠ `j = 1.5`

Incrémentation et décrémentation

Incrémentation

- `i++;` ou `++i;` signifie `i = i + 1;`
- `i += n;` signifie `i = i + n;`
- Même chose avec `i--`, `--i` et `i -= n`

Pré-incrémentation

- `int i = 5; int a = ++i;` donne `i=6` et `a=6`
- `int i = 5; int a = --i;` donne `i=4` et `a=4`

Post-incrémentation

- `int i = 5; int a = i++;` donne `i=6` et `a=5`
- `int i = 5; int a = i--;` donne `i=4` et `a=5`

Opérateurs relationnels et logiques

➽ **Opérateurs relationnels** (`true` ou `false`)

- `==`, `!=` : identité, différent . Ex : `A == 5`
- `>`, `<`, `>=`, `<=` : supérieur, inférieur

Exemples `int A = 10, B = -5 ;`

- `A == B` ➽ `0` : `false`
- `A >= B` ➽ `1` : `true`

Attention Les deux opérandes doivent avoir le même type arithmétique, sinon une conversion de types est effectuée.

➽ **Opérateurs logiques** :

- `!` : négation unaire d'une valeur logique. Ex : `!true == false`
- `&&` : « ET » de deux valeurs logiques. Ex : `true && false == false`
- `||` : « OU » de deux valeurs logiques. Ex : `true || false == true`

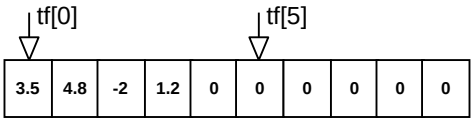
Exemple de combinaison d'opérateurs : `(K>5) && (L<10)`.

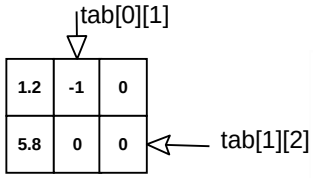
Priorité des opérateurs (ordre décroissant)

Classe d'opérateur	Opérateur(s)	Associativité
Parenthèsage	()	de gauche à droite
Suffixes	[] -> . ++ --	de gauche à droite
Unaires	& * + -! sizeof ~	de droite à gauche
Changement de type	(type)	de droite à gauche
Multiplicatifs	* / %	de gauche à droite
Additifs	+ -	de gauche à droite
Décalage	<< >>	de gauche à droite
Comparaisons	< <= > >=	de gauche à droite
Égalités	== !=	de gauche à droite
ET bit à bit	&	de gauche à droite
OU exclusif bit à bit	^	de gauche à droite
OU bit à bit	|	de gauche à droite
ET logique	&&	de gauche à droite
OU logique	||	de gauche à droite
Condition	? :	de droite à gauche
Affectations	= += -= *= /= &= |= ^= <<= >>=	de gauche à droite
Succession	,	de gauche à droite

Types dérivés (1) : tableaux

- Déclaration : `int tab[5] ;`
- Initialisation :
  - `double tf[10]={3.5, 4.8, -2, 1.2} ;`
  - `float tab[2][3]={{1.2, -1}, {5.8}} ;`





- Affectation :
  - `tf[5] = 3.8 ;`
  - `tab[0][2] = -5.0 ;`

démo 6

```c
#include <stdio.h>
int main() {

    double tf[5] = {1.2, -4.7, 3.4};
    printf("\ntf[0]= %lf", tf[0]);

    double t = tf[2];
    printf("\nt    = %lf", t);

    tf[3] = -8.789;
    printf("\ntf[3]= %lf", tf[3]);

    printf("\ntf[5]= %lf", tf[5]);
    tf[5] = 1.21;
    printf("tf[5]= %lf", tf[5]);

    return 0;
}
```

Types dérivés (2) : chaînes de caractères

- Déclaration : `char chaine[20];`
- Initialisation :
  - `char hello1[13] = {'b','o','n','j','o','u','r'};`
  - `char hello2[13] = "bonjour";`

b	o	n	j	o	u	r	\0					

  - `char ch[] = "message";`

m	e	s	s	a	g	e	\0

- `ch[4] = 'b'` : remplace la lettre `a` par la lettre de `b`.
- Pensez à ajouter un espace mémoire pour le caractère de terminaison de chaîne \0.
- La fonction `strlen()` (bibliothèque `#include <string.h>`) ne s'applique qu'aux chaînes de caractère (et non aux tableaux de nombres !).

démo 7

```c
#include <stdio.h>
#include <string.h>
int main() {
   char ch1[50] = "Bonjour !";
   printf("\nch1=%s; ch1[3]=%c; ch1[20]=%c", ch1, ch1[3], ch1[20]);

   char ch2[120];
   printf("\nEntrez une chaine : ");
   scanf("%s", ch2);

   ch2[3] = 'Z';
   printf("\nch2=%s; ch2[3]=%c; ch2[20]=%c", ch2, ch2[3], ch2[20]);

   int taille = strlen(ch2);
   printf("\nTaille de ch2 = %d", taille);

   return 0;
}
```

Instructions de contrôle

- Instructions conditionnelles :
  - `if` : test. Tr. 26
  - `switch` : table de branchement. Tr. 28
- Instructions répétitives :
  - `while`. Tr. 30
  - `for`. Tr. 32
  - `do ... while`. Tr. 31
- Rupture de séquence :
  - `continue`. Tr. 35
  - `break`, `return`. Tr. 36

Inst. cond. : `if`

1- `if` (expression)
   instruction
2- `if` (expression)
   instruction1
   `else`
   instruction2

```
if (fin == 1)
    printf("Au revoir !");

if ( (n==1) && (m>=10)) {
    printf("Bonjour");
    fin = 1;
}
else
    printf("Au revoir");
```

```c
int main() {

    printf("Voulez vous imprimer le mode d'emploi ?\n");
    char c;
    scanf("%c", &c);

    if (c=='o')
        printf("\nJe lance le processus d'impression");
    else {
        if (c=='n')
            printf("\nJe ne fais rien");
        else
            printf("Tapez o ou n");
    }

    return 0;
}
```


Inst. cond. : `switch`

`switch`	(expression)	{
	`case` value1 :	bloc inst 1
		break ;
	`case` value2 :	bloc inst 2
		break ;
	`case` valueN :	bloc inst N
		break ;
	`default` :	bloc inst d
		break ;
	}	

Inst. cond. : switch

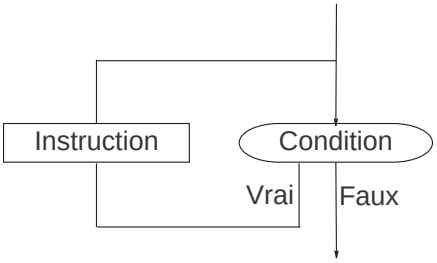
```c
char c;
scanf("%c", &c);
switch(c)
  {
    case 'a' :
      printf("Choix 'a'");
      break;
    case 'A' :
      printf("Choix 'A'");
      break;
    case 'b' :
      printf("Choix 'b'");
      break;
    case 'B' :
      printf("Choix 'B'");
      break;
    default :
      printf("Mauvais choix !");
      break;
  }
```

démo 8

```c
int i;
scanf("%d", &i);
switch(i)
  {
    case 1 :
    case 2 :
      printf("Choix 1 ou 2");
      break;
    case 3 :
    case 4 :
      printf("Choix 3 ou 4");
      break;
    default :
      printf("Mauvais choix !");
      break;
  }
```


Inst. répétitive : while

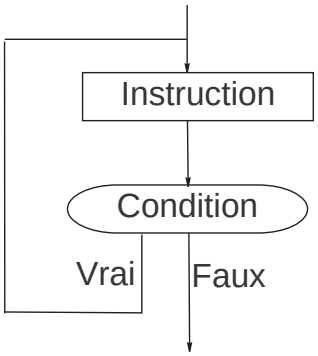
`while` (expression de fin) instruction




```c
int nb;
printf("\nEntrez un nombre : " );
scanf("%d", &nb);

int i = 0;
while (i<nb)
{
  printf("Iteration : %d", i);
  i++;
}
```

Inst. répétitive : do ... while

`do` instruction `while` (expression de fin) ;

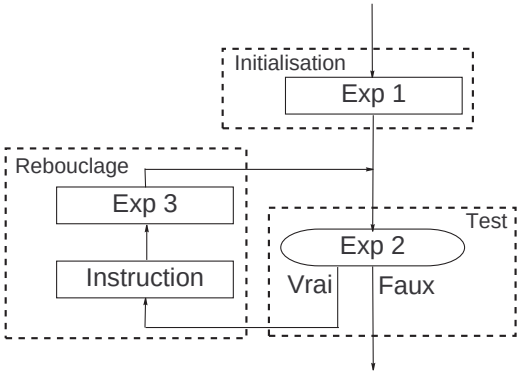


démo 9

```
int s[10], i=0, n=5634;
do {
  s[i] = n % 10;
  i++;
  n /= 10;
} while ( n > 0 );
```


Inst. répétitive : for

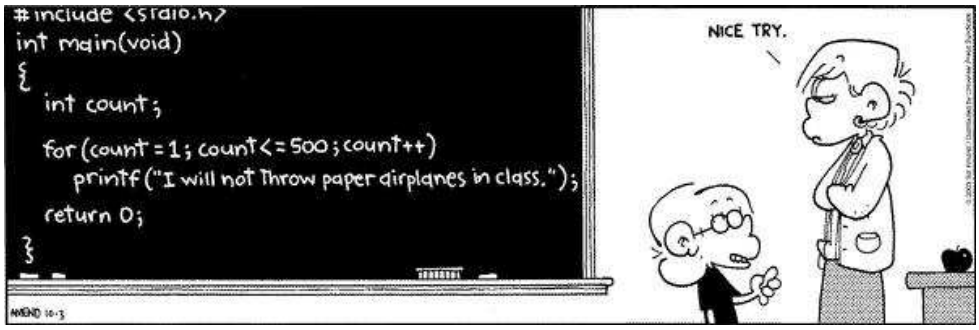
`for` (exp1 ; exp2 ; exp3) instruction



```c
double tab[4] = {3.2, 5.1, 9, -1.8};
int i;
for (i=0; i<=3; i++)
    printf("\nValeur: %lf", tab[i]);

int nb;
printf("Entrez un nombre : ");
scanf("%d", &nb);

for (int i=nb; i>0; i--)
  printf("Iter numero: %d", i);
```

démo 10

```c
char ch1[]="Coucou !";
char ch2[50];
int i;
for (i=0; i<=strlen(ch1);
      i++)
  ch2[i] = ch1[i];
printf("Copie=%s", ch2);
```

Rupture : `continue`

```
int  i, j = 0;
char c=0;
for (i = 0; c != '\n'; i++){
    scanf("%c", &c);
    if ( (c == ' ' || (c == '\t') || (c == '\n'))
        continue;
    j++;
}
```

démo 11

Le programme compte le nombre de caractères non blancs entrés au clavier et le nombre total de caractères (jusqu'à un Retour-chariot).

- `i` contient le nombre total de caractères qui ont été tapés au clavier ;
- `j` contient le nombre de caractères non blancs ;

Rupture : `break`

```
int  i, j=0;
char c=0;
for( i=0; ; i++ ) {
    scanf("%c",&c);
    if (c == '\n') break;
    if ( (c == ' ') || (c == '\t')) continue;
    j++ ;
}
```

- La boucle se termine lorsque l'utilisateur tape un retour-chariot.

Fonctions : rôles et intérêts

- **Objectifs** : Découper un programme en petites entités :
  - Indépendantes ;
  - Réutilisables ;
  - Plus lisibles.
- **Où** : Les fonctions peuvent être définies :
  - Dans le fichier principal ;
  - Dans d'autres fichiers source (compilation séparée - Tr. 54) ;
- **Syntaxe d'une fonction**

```
type_de_retour nom_fonction ( arguments ) {
    declarations des variables locales ;
    instructions ;
}
```


Fonction sans argument

```c
// Déclaration de la fonction
void Affiche() {
    printf("Message");
}

// fonction principale
int main() {
    // Appel de la fct
    Affiche();
    return 0;
}
```

```c
// En-tête de la fonction
void Affiche();

// fonction principale
int main() {
    // 1ppel de la fct
    Affiche();
    return 0;
}

// Déclaration de la fonction
void Affiche() {
    printf("Message");
}
```

Fonction : valeur de retour

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

// Déclaration de la fonction
double Sinus() {
    double angle;
    scanf("%lf", &angle);
    return sin(angle);
}

int main() {
    double res;
    // Appel de la fonction
    res = Sinus();
    printf("\nResultat = %lf", res);
    return 0;
}
```


Fonction : passage d'argument par valeur

démo 13

```c
#include<stdio.h>

// Déclaration de la fonction
int Add(int val1, int val2) {
    int result;              // dec var locale
    result = val1 + val2;    // calcul
    return result;           // retour
}

int main() {
    int res, a = 5, b = 8;
    //Appels de la fonction
    res = Add(a, b);         printf("\nResultat = %d", res);
    res = Add(-8, 15);       printf("\nResultat = %d", res);
    res = Add(-8*2-5, 6*a); printf("\nResultat = %d", res);

    return 0;
}
```

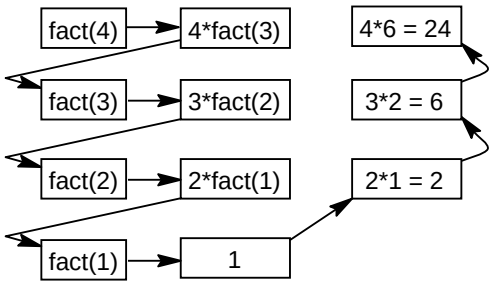
➥ Les arguments transmis sont recopiés dans les variables locales de la fonction.

Fonctions récursives

➪ La fonction s'appelle elle-même ! ➪ Exemple : $4! = 4 * 3!$.

démo 16

```c
#include<stdio.h>

// Déclaration de la fonction
int fact(int n) {
    if (n<=1)
        return 1;
    else
        return n*fact(n-1);
}

int main() {
    // Appel de la fonction
    printf("\nFactoriel 4 = %d", fact(4));
    return 0;
}
```

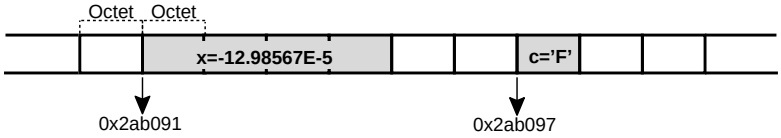


Attention au test de fin de récursivité !

Pointeurs et adresses

Type pointeur

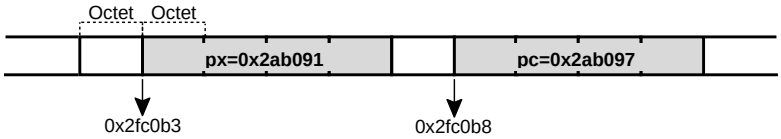
- **Variables :** `float x=-12.98587E-5 ; char c='F' ;`



- **Pointeurs :**

  Déclaration : `float* px ;` et `char* pc ;`

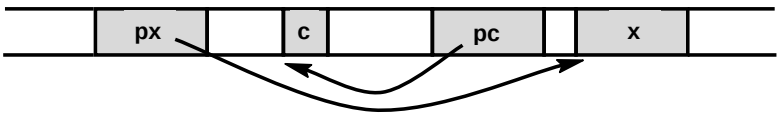
  Initialisation : `px = &x ;` et `pc = &c ;`



- **Remarque :** La taille d'un pointeur, c'est à dire la quantité de mémoire réservée au stockage d'une adresse, est de 4 octets (quel que soit le type sur lequel il pointe).

  `sizeof(char*)==sizeof(double*)==4` `sizeof(char)==1 ; sizeof(double)==8`

Schéma global :



Autre exemple :

```
int x=1, y=2;
int* px;     // pi pointe sur un int
px  = &x;    // pi pointe sur x
y   = *px;   // y reçoit la valeur 1 (x)
*px = 0;     // x vaut 0
```

Remarques :

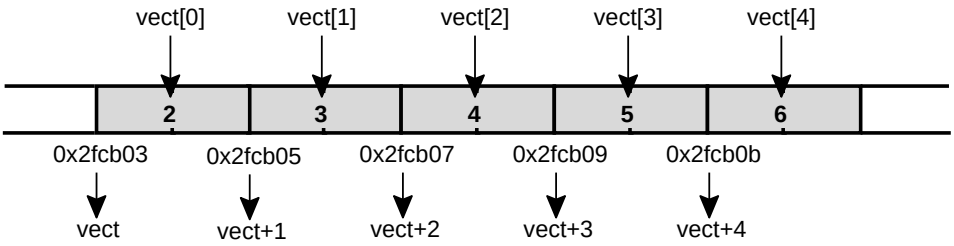
- `&*p` est identique à `p` (pour `p` un pointeur)
- Opérateurs sur les pointeurs : `+`, `-`, `++`, `--`, `+=`, `-=`, `==`, `!=` ...
- `p=NULL` : identique à l'initialisation à `0` d'une variable entière ou flottante.

```c
#include <stdio.h>
void main( ) {
  int i = 0;
  int *p;
  float x = 3.14;
  float *f;

  p = &i;
  *f = 666;
  f = &x;
  *f = *p;
  *p = 34;
  p = f;
  *p = *p + 1;

  printf("\ni=%d; *f=%f", i, *f);
}
```

Supprimer les lignes invalides - Que s'affiche-t'il ?

Pointeurs et Tableaux

➪ **Tableau 1D (Vecteur) :** `int vect[5] = {2, 3, 4, 5, 6};`



- ➪ `&vect[0] == vect` : Adresse du 1er élément de `vect`.
- ➪ `&vect[i] == vect+i` : Adresse du ième élément de `vect`.
- ➪ `vect[i] == *(vect+i)` : Valeur du ième élément de `vect`.

➽ C'est la fin du mystère `scanf("%s", chaine);` !

➽ **À vous de deviner ... :**

```
float *p, *q;
float tab[5] = {1, 2, 3, 4, 5};

p = &tab[2];
p = tab+2;
*p= 0.0;
q = p++;          \\Post-incrémentation !
*(q+2) = -10;
```

Supprimer les lignes invalides - État du vecteur ?

Allocation dynamique de mémoire

➽ **Types d'allocation :**

➽ Allocation statique :
- ➪ Pour créer et supprimer des ≪ objets ≫ lors de la compilation.
- ➪ Utilise la pile (limitée en taille).
- ➪ La taille de la mémoire à allouer doit être connue à la compilation.

➽ Allocation dynamique :
- ➪ Pour créer et supprimer des ≪ objets ≫ lors de l'execution.
- ➪ Utilise le tas (limitée par la mémoire de votre PC).
- ➪ La taille de la mémoire à allouer peut être connue lors de l'éxecution.

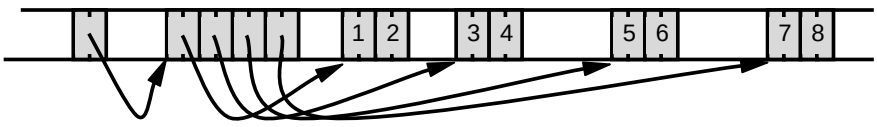
➽ **Allocation dynamique :** librairie `#include<stdlib.h>`
- ➪ `malloc()` alloue de la mémoire
- ➪ `free()` libère de la mémoire

démo 12

```c
#include<stdio.h>
#include<stdlib.h>
int main() {

  int taille, i;
  printf("Entrez la taille du vecteur : ");
  scanf("%d", &taille);

  double *tab;
  tab = (double *) malloc(taille * sizeof(double));
  for(i=0; i<taille; i++) tab[i] = 8.0*i;
  free(tab);

  return 0;
}
```


Allocation dynamique de mémoire : Exo

➫ **Tableau 2D (Matrice) :** Écrire un programme qui alloue dynamiquement la mémoire d'un tableau à deux dimensions, selon le schéma suivant (exemple : 4 lignes, 2 colonnes) :



démo 12bis

Passage d'arguments par adresse dans les fonctions

démo 14

```c
#include<stdio.h>

void Swap(int* pt1, int* pt2) {
    int aux = *pt1;
    *pt1 = *pt2;
    *pt2 = aux;
}

int main() {
    int a = 5, b = 8;

    printf("\na=%d, b=%d", a, b);
    Swap(&a, &b);
    printf("\na=%d, b=%d", a, b);

    return 0;
}
```

➪ Les arguments transmis sont les adresses des variables. Les valeurs des variables de la fonction appellante sont modifiées.

Calcul de la longueur d'une chaîne

démo 15

```c
int strlen1(char* s) {
  int n;
  for(n=0;*s != '\0'; n++) s++;
  return n;
}
int strlen2(char* s) {
   char* p=s;
   while ( *p != '\0' )  p++;
   return p-s; // Conv. héxa -> déc.
}

int main() {
  int taille1, taille2;
  char chaine[]="La pie chante.";
  taille1 = strlen1(chaine);
  taille2 = strlen2(chaine);
  printf("\ntaille1=%d, taille2=%d", taille1, taille2);
  return 0;
}
```

Concaténation de deux tableaux

```c
#include<stdio.h>
#include<stdlib.h>
const int TAILLE=3;

int* ConcatVect(int* Vect1, int* Vect2) {
  int* NewVect;
  NewVect = (int* ) malloc(2*TAILLE*sizeof(int));

  for(int k=0; k<TAILLE; k++) {
    NewVect[k]        = Vect1[k];
    NewVect[k+TAILLE] = Vect2[k];
  }

  return NewVect;
}

int main() {
  int tab1[TAILLE] = {1,2,3};
  int tab2[TAILLE] = {4,5,6};
  int* tab12;
  tab12 = ConcatVect(tab1, tab2);
  ...
  free(tab12); // libération mémoire
  return 0;
}
```

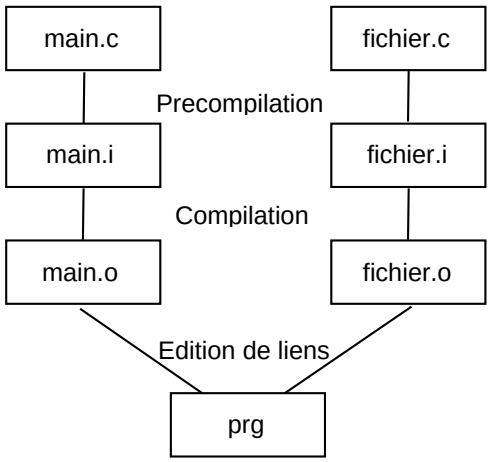

Compilation séparée

- Compilation d'un programme **dans un seul fichier**
  - 1 : `gcc main.c -o prg` ⇨ fichiers `main.o` et `prg`

Equivalent à

  - 1 : Compilation `gcc -c main.c` ⇨ fichier `main.o`
  - 2 : Ed. de liens `gcc main.o -o prg` ⇨ éxecutable `prg`

- Compilation d'un programme **dans plusieurs fichiers**
  - Les fonctions sont réparties dans plusieurs fichiers.
  - Les fonctions non déclarées dans le fichier principal (celui contenant la fonction `int main() ...`) sont appelées *fonctions externes*.



- 1 : Compilation `gcc -c main.c` fichier `main.o`
- 2 : Compilation `gcc -c fichier.c` fichier `fichier.o`
- 3 : Ed. de liens `gcc main.o fichier.o -o prg` éxecutable `prg`


```c
// main.c

// Prototype des fonctions
extern int plus(int a, int b);
extern int mult(int a, int b);

// Programme principal
int main() {
   int a = 5, b = 8;
   int result1 = plus(a, b);
   int result2 = mult(a, b);
   return 0;
}
```

```c
// fichier.c

int plus(int a, int b) {
   int c;
   c = a + b;
   return c;
}

int mult(int a, int b) {
   int c;
   c = a * b;
   return c;
}
```

On délocalise les en-têtes des fonctions dans un fichier « header file », que l'on appelle généralement `fichier.h` (extension `.h`). Dans ce fichier, on y ajoute la définition des constantes, des structures, les macro-fonctions ...

```c
// main.c
#include "fichier.h"

int main() {
    int a = 5, b = 8;
    int result1 = plus(a, b);
    int result2 = mult(a, b);
    return 0;
}
```

```c
// fichier.h
extern int plus(int a, int b);
extern int mult(int a, int b);
```

```c
// fichier.c
int plus(int a, int b) {
    int c = a + b;
    return c;
}

int mult(int a, int b) {
    int c = a * b;
    return c;
}
```

démo 17 à modifier

Makefile : pour simplifier la compilation

- Principe du Makefile
  - Au lieu de taper toutes les commandes de compilation et d'édition de liens, on peut les regrouper dans un fichier (généralement appelé `Makefile` ou `makefile`).
  - Les `Makefile` doivent respecter une syntaxe très particulière.
  - On lance la compilation grâce à l'utilitaire `make`, qu'il suffit d'exécuter avec `make` (ou `make -f Makefile`).
  - La compilation par `make` est intelligente, au sens où n'est recompilé que ce qui a été modifié.

➪ Règle d'écriture d'un `Makefile` Le principe pour « rédiger » un `Makefile` est de décrire les dépendances entre les fichiers selon le principe suivant :

```
fichier_cible : liste_fichiers_requis
<tabulation> commande_associee
```

➽ `fichier_cible` : Nom du fichier qui doit être mis à jour.

➽ `liste_fichiers_requis` : Liste des fichiers (séparés par des espaces) dont dépend `fichier_cible`. Ces fichiers sont nécessaires à la création de `fichier_cible`.

➽ `commande_associée` : Ligne de commande permettant la mise à jour de `fichier_cible`.

➪ Exemple de Makefile

```
# Création de l'exécutable par édition de liens.
prg: main.o fichier.o
    gcc -o prg main.o fichier.o

# Création de main.o par compilation.
main.o: main.c fichier.h
    gcc -c main.c

# Création de fichier.o par compilation.
fichier.o: fichier.c fichier.h
    gcc -c fichier.c
```

Préprocesseur

➽ Le préprocesseur intervient juste avant la phase de compilation. Il prend en compte :

- ➪ Les commentaires (suppression),
- ➪ Les commandes commençant par `#`.

➽ 4 types de *directives* commençant par `#`

- ➪ Inclusion des fichiers d'en-tête (`#include`)
- ➪ Définition de variables du pré-processeur (`#define` et `#undef`)
- ➪ Les macro-fonctions (`#define`)
- ➪ La sélection de codes en fonction des variables du pré-processeur (`#if`, `#else`, `#elif` et `#end`)

Variables du pré-processeur

➽ Constante de pré-compilation

Sans constante de pré-compilation	Avec constante de pré-compilation

`int tab[20] ;`
`for( i=0 ; i<20 ; i++ )`
`  ...`
	`#define LG 20`
	`int tab[LG]`
	`for( i=0 ; i<LG ; i++ )`
	`  ...`

➽ Si on souhaite changer la taille du tableau, il suffit de changer la variable (lisibilité, gain de temps, ...).

➽ Pour supprimer une variable : `#undef`.

Macro-expression ou macro-fonction

- Directive : `#define`.
- Exemple : `#define add(x1,x2) ((x1) += (x2))` .
- `add(a,b)` est remplacé par `((a) += (b))`
- Attention au parenthésage : Il faut toujours encadrer les pseudo-variables par des parenthèses.

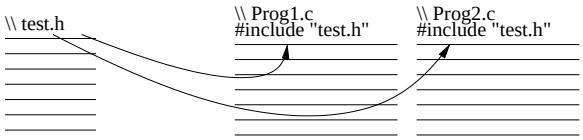
Évaluation de macro-expressions

démo 18

Avant pré-compilation	Après pré-compilation	Bon/Faux
`#define m(x) 128*x+342*x*x` `#define y(x) 128*(x)+342*(x)*(x)`		
`int a,b,d,e,f,g;` `a = b = 1;`	`int a,b,d,e,f,g;` `a = b = 1;`	
`d = m(a);` `e = y(a);`	`d = 128*a+342*a*a;` `e = 128*(a)+342*(a)*(a);`	bon bon

`f = m(a+b);`
`g = y(a+b);`
	`d = 128*a+b+342*a+b*a+b;`
	`d = 128*(a+b)+342*(a+b)`
	`*(a+b);`
		faux
		bon

Protection contre l'inclusion multiple




```c
// Test.h
#ifndef TEST_H
#define TEST_H

extern int add(int C, int D);

int VarGlobale;

#endif
```


Structures, unions et énumérations

- **Types de base :** char, short, int, float, double + pointeurs.
- **Types dérivés :** Tableaux de nombres ou de caractères.
- **Types complexes (objets) :**
  - **Énumération :** Sert à offrir la possibilité de gérer des constantes numériques, Tr. 103.
  - **Structure :** Objets composé de plusieurs champs. Exemple : Voiture = (marque, num. d'immatriculation, année, ...)
    - Définition et exemples, Tr. 67
    - Structure et pointeurs, Tr. 70
    - Structure et fonctions, Tr. 71
    - Tableau statique de structures, Tr. 74
    - Structure et alloc. dyn. de mémoire, Tr. 75
    - Structures auto-référentielles, Tr. 77

Structures : définition et utilisation

➽ **Structures :** Agrégat de plusieurs objets de types quelconques (appelés *champs* de la structure).

➽ **Syntaxe :**

```
struct nom_structure {
    type1 variable1 ;
    type2 variable2 ;
    ...
} ;
```

➽ **Exemple :** Point du plan

```
struct Point2D {
    float x ;
    float y ;
} ;
```


➽ Une structure est un type et **non** une variable.

➽ Pour déclarer une variable du type Point2D :

- `struct Point2D coord ;`
- `struct Point2D TabCoord[3] ;`

➽ On peut déclarer et initialiser les champs d'une structure :

- `struct Point2D coord = {2.8, 9.8} ;`
- `struct Point2D TabCoord[3] = {{2.8, -0.8}, {4.2, -9.8}, {-1.0, 1.0}} ;`

➽ Pour accéder à un champ de la structure :

- `printf("%f",coord.x) ;`
- `coord.x = 5.0 ;`
- `double root = coord.x*coord.x+coord.y*coord.y ;`
- `printf("%f", TabCoord[1].y) ;`

➽ L'affectation entre deux structures de même type est permise :

```
struct Point2D coord1 = {3.0, 8.5}, coord2 ;
coord2 = coord1 ;
```

➽ Par contre, il n'est pas possible de tester l'égalité, la supériorité, ... de deux structures. Il faut comparer individuellement chaque champs :

```
if ((coord1.x==coord2.x) && (coord1.y==coord2.y) )... ;
```

➽ Structure incorporant des structures. Exemple :

```
struct Rect {
    struct Point2D pt1 ;
    struct Point2D pt2 ;
} ;
```

Déclaration et init. : `struct Rect carre = {{1.0,2.0}, {3.0,4.0}} ;`

Affectation : `carre.pt1.x=3.2 ; carre.pt1.y=5.8 ;`

Structures et pointeurs

➽ Déclaration d'un pointeur sur une structure :

```
struct Point2D* p ;
```

➽ Initialisation d'un pointeur sur une structure :

```
struct Point2D coord1 ;
p = &coord1 ;
```

➽ Comment accéder au contenu de la valeur pointée ?

```
(*p).x == coord1.x
```

Remarque : Plutôt que `(*p).x`, on préfèrera écrire `p->x`.

Exemple : `printf("%f, %f, %f", (*p).x, p->x, coord1.x) ;`
affiche 3 fois la même valeur.

Structures et fonctions

démo 19

```c
#include<stdio.h>
struct Point2D {
  double x;
  double y;
};

struct Point2D InitPoint (double val1, double val2) {
  struct Point2D temp;
  temp.x = val1;  temp.y = val2;
  return temp;
}

int main() {
  struct Point2D point;
  double a,b;

  printf("Entrez deux valeurs :");
  scanf("%lf%lf", &a, &b);
  point = InitPoint(a, b);
  printf("x=%lf, y=%lf", point.x, point.y);

  return 0;
}
```


démo 20

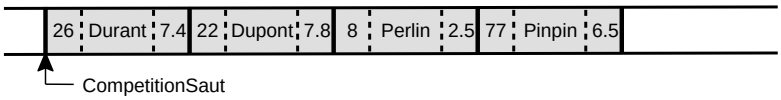
```c
#include<stdio.h>
struct Point2D {
  double x;
  double y;
};

int ComparePoint (struct Point2D pt1, struct Point2D pt2) {
  return (pt1.x == pt2.x) && (pt1.y == pt2.y) ;
}

int main() {

  struct Point2D point1 = {2.0,-9.5};
  struct Point2D point2 = {-1.0, 2.0};

  int ok = ComparePoint(point1, point2);
  if (ok == 1)
    printf("Les deux points sont identiques");
  else
    printf("Les deux points sont differents");

  return 0;
}
```

démo 21

```c
#include<stdio.h>
struct Point2D {
  double x;
  double y;
};

void AjoutePoint (struct Point2D* pt1, struct Point2D pt2) {
  pt1->x += pt2.x;
  pt1->y += pt2.y;
}

int main() {

  struct Point2D point1 = {2.0,-9.5};
  struct Point2D point2 = {-1.0, 2.0};

  printf("\nAvant : point1(%f,%f), point2(%f,%f)", point1.x,
       point1.y, point2.x, point2.y);
  AjoutePoint(&point1, point2);
  printf("\nApres : point1(%f,%f), point2(%f,%f)\n", point1.x,
        point1.y, point2.x, point2.y);

  return 0;
}
```


Tableau statique de structures

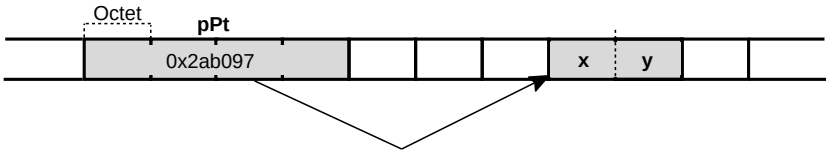
```c
#include <string.h>
struct athlete {
   int    Age;
   char   Nom[15];
   double BestJump;
};
void main() {
   struct athlete CompetitionSaut[10];
   CompetitionSaut[0].BestJump =  7.4;
   CompetitionSaut[0].Age      = 26;
   strcpy(CompetitionSaut[0].Nom, "Durant");
}
```



Structure et allocation dynamique de mémoire

Exemple 1 : une structure

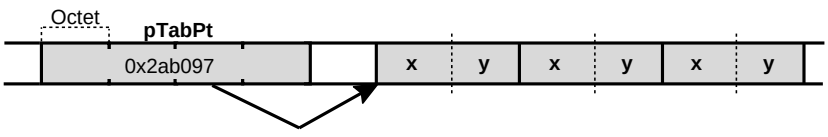
```
struct Point2D* pPt;
pPt = (struct Point2D* ) malloc( sizeof(struct Point2D) );
...
free(pPt); pPt = NULL;
```



Exemple 2 : un tableau de structures

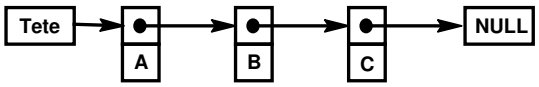
démo 22

```
int cpt;
printf("Dimension du tableau : "); scanf("%d", &cpt);

struct Point2D* pTabPt;
pTabPt = (struct Point2D* ) malloc (cpt * sizeof(struct Point2D));
...
free(pTabPt);
pTabPt = NULL;
```



Structures auto-référentielles : Listes

➽ **Liste** : Ensemble d'objets *chainés* et de *même type*. Les objets sont appelés *maillons*.




```
struct maillon {
   char           symb;
   struct maillon* suiv;
};
typedef struct maillon M;
```

démo 23

- ➪ Queue ou file d'attente : Liste FIFO ≪ First In, First Out ≫ (Ex : File d'attente à un guichet)
- ➪ Pile : Liste LIFO ≪ Last In, First Out ≫
- ➪ Listes circulaires
- ➪ Listes symétriques

➽ Opérations classiques sur les listes :

- ➪ Ajouter un objet (allocation mémoire),
- ➪ Supprimer un objet (libération mémoire),
- ➪ Tester si la liste est vide (`NULL`),
- ➪ Supprimer toute la liste (libération mémoire),
- ➪ Parcourir la liste : pour retrouver ou afficher un objet,
- ➪ Trier une liste,
- ➪ ...

Les fichiers

➽ Rangement des fichiers : Le système de répertoires est un outil de classement organisé sous forme arborescente :

- ➪ Il existe un répertoire racine (*root*) à partir duquel s'organise l'arborescence.
- ➪ Chaque répertoire (à l'exception du répertoire racine) possède un répertoire père.
- ➪ Un répertoire peut contenir à la fois des fichiers et d'autres répertoires (qui sont ses répertoires fils).
- ➪ L'organisation sous forme de répertoires permet de positionner les différents fichiers du disque. La position d'un fichier au sein de l'arborescence est donnée par un chemin (*path*) qui part de la racine jusqu'au répertoire contenant le fichier.

➽ Un fichier est doté :

- ➪ d'un nom,
- ➪ d'une position (son chemin dans le système de répertoires)
- ➪ d'un certain nombre de droits (son ou ses propriétaires)
- ➪ d'une taille (correspondant à un certain nombre d'emplacements réservés sur l'espace de stockage)
- ➪ d'une date de création, de modification ...

➽ Descripteur de fichiers : Un descripteur de fichiers sert à désigner une connexion avec un fichier (ou un périphérique d'E/S comme l'écran, l'imprimante, le clavier, ...). Un fichier est représenté par une structure prédéfinie `FILE` (en majuscule !).

```
FILE* f ;
```

Le pointeur `f` désigne le flux (canal de communication) établi avec le fichier.

- Descripteurs pré-définis :
  - `stdin` : flux d'entrée standard
  - `stdout` : flux de sortie standard
  - `stderr` : flux de sortie des messages d'erreur

```
do {
  fprintf (stdout, "Voulez vous continuer (o/n)?\n");
  fscanf(stdin, " %c", &rep);
} while (rep=='o');
```

Est équivalent à :

```
do {
   printf ("Voulez vous continuer (o/n)?\n");
   scanf(" %c", &rep);
} while (rep=='o');
```


Modes et fonctions d'ouverture

- l'appel à la fonction `fopen` s'écrit :

```
g = fopen (nom, mode) ;
```

- l'opération d'ouverture correspond :
  - à associer à `g` un fichier décrit par son nom. On dit pour simplifier que `g` « pointe sur le fichier ».
  - au positionnement de la tête de lecture sur la première ligne de ce fichier.
- Fermeture du flux associé au pointeur sur fichier.
  - En cas de mode d'utilisation en écriture, les données écrites sont transférées sur le disque dur.
  - Exemple : `fclose(g) ;`

➽ Modes d'ouverture (`FILE *f ;`)

- En lecture :
  ```
  g = fopen ("entree.txt", "r") ;
  ```
- En écriture (avec écrasement !)
  ```
  g = fopen ("resultat.txt", "w") ;
  ```
- En écriture/ajout (sans écrasement)
  ```
  g = fopen ("resultat.txt", "a") ;
  ```
- Lecture/Ecriture (sans écrasement, position fin) :
  ```
  g = fopen ("resultat.txt", "r+") ;
  ```
- Lecture/Ecriture (avec écrasement) :
  ```
  g = fopen ("resultat.txt", "w+") ;
  ```
- Lecture/Ecriture (sans écrasement, position début) :
  ```
  g = fopen ("resultat.txt", "a+") ;
  ```

À la seconde chaîne, on peut ajouter le caractère `"b"`, pour signifier que l'enregistrement sera codé en **binaire**. Par défaut, le codage est `"t"` pour **texte**.

➽ Test à l'ouverture Selon les droits disponibles, il arrive souvent que l'ouverture d'un fichier se passe « mal ». Il faut traiter les erreurs d'ouverture pour éviter des erreurs ultérieures.

```
f = fopen ("monfichier.txt","r");
if (f==NULL) {
   printf("Impossible d'ouvrir !\n");
   exit(0);
}
```

- La valeur NULL pour le pointeur `f` indique une erreur d'ouverture
- La fonction `exit` termine l'exécution du programme.

Les fichiers texte

➽ Organisation d'un fichier texte : Une séquence de chaînes de caractères séparées par sauts à la ligne.

	col 1	col 2	col 3	col 4	...							
ligne 1	b	l	a	b	l	a	\n					
ligne 2	\n											
ligne 3	b	l	a	b	l	a	b	l	a	b	l	a	\n
...	b	l	a	\n								
	b	l	a	b	l	a	b	l	a	\n		
	b	l	a	\n								

➽ Organisation d'un fichier de données texte : Dans un fichier de données texte, les données sont structurées sous forme de tableau, où les champs sont en général séparés par des tabulations.

	champ 1		champ 2		champ 3	
ligne 1	info11	\t	info12	\t	info13	\n
ligne 2	info21	\t	...	\t	...	\n
ligne 3	info31	\t	...	\t	...	\n
	...	\t	...	\t	...	\n
	...	\t	...	\t	...	\n
	...	\t	...	\t	...	\n

➽ Parcours (lecture) d'un fichier texte : Fonction `fscanf`

- Chaque appel à cette fonction permet de poursuivre la lecture, c'est à dire de lire l'élément qui suit l'élément précédemment lu.
- À chaque nouvel appel à `fscanf`, la tête de lecture est déplacée sur l'élément suivant.
- `fscanf` retourne le nombre d'éléments lus
- Exemple :

```
FILE *f = fopen("data.txt", "r");
char mot[50];
fscanf(f,"%s",mot);
printf("voici le mot : %s", mot);
```


➽ Test de fin de fichier : Fonction `feof`

- A chaque lecture, la tête de lecture se déplace
- La fonction feof indique si la fin de fichier est atteinte (valeur 1)
- Exemple :

```
char p;
while (feof(f)==0) {
  fscanf(f, "%c", &p);
  printf("%c", p);
}
```

➽ Exemple récapitulatif :

```
#include<stdio.h>
int main(){
  FILE *f;
  char p;
  f = fopen ("texte.txt","r");
  if (f==NULL) {
    printf("Impossible d'ouvrir !\n");
    exit(0);
  }
  while(feof(f)==0){
    fscanf(f, "%c", &p);
    printf("%c", p);
  }
  fclose(f);
}
```


➽ Écriture dans un fichier texte :

➪ Mode d'ouverture de `fopen` : `w` - Exemple :

```
FILE *h;
h = fopen("resultat.txt", "w");
```

➪ Utilisation de la fonction `fprintf` : comme `printf`. Exemple :

```
char mot[20] = "Bonjour";
fprintf(h, "%s", mot);
```

➪ Les données sont écrites physiquement au moment de la fermeture : `fclose(h);`

➽ Accès séquentiel par caractère et par ligne

par caractère :

- `int fgetc(FILE* f)` : Lecture d'un caractère à partir du fichier pointé par `f`. Ex : `char c = fgetc(ficIn);`.
- `int fputc(char c, FILE* f)` : Écrit le caractère `c` dans le fichier pointé par `f`. Ex : `char c = 'o'; fputc(c, ficIn);`.

par ligne :

- `char* fgets(char* ch, int n, FILE* f)` : Lecture de `n-1` caractères sur le fichier pointé par `f`. Les caractères lus sont rangés dans `ch`.
- `int fputs(char* ch, FILE* f)` : Écrit la chaîne de caractères `ch` dans le fichier pointé par `f`.

Exemple : Recopie d'un fichier

démo 24

```
#include <stdio.h>
#include <stdlib.h>
void main() {
   char  c;

   FILE* ficIn = fopen("..\original.txt", "r");
   if (ficIn == NULL) exit(0);
   FILE* ficOut = fopen(".\temp\copie.txt", "w");
   if (ficOut == NULL) exit(0);

   while ( feof(ficIn) == 0 ) {
       c = fgetc(ficIn);
       fputc(c, ficOut);
   }

   fclose(ficIn);
   fclose(ficOut);
}
```

Les fichiers binaires

- Ouverture : Exemple `g = fopen("fichier.bin", "wb") ;`
- Fermeture : `fclose(g) ;`
- Lecture :
  `int fread(void* p, int s, int nb, FILE* f) ;`.
  Cette fonction transfert, depuis le fichier associé à `f`, `nb` éléments de taille `s` octets et dont le type est celui de `*p`. La fonction retourne le nombre d'octets lus.
  Exemple :

```
char mot[20];
fread( mot, 1, 10, g);
```


- Écriture :
  `int fwrite(void* p, int s, int nb, FILE* f) ;`.
  Cette fonction transfert, vers le fichier associé à `f`, `nb` éléments de taille `s` octets et dont le type est celui de `*p`. La fonction retourne le nombre d'octets écrits.
  Exemple :

```
char mot[20] = "Bonjour";
fwrite( mot, 1, strlen(mot), g);
```

```c
#include <stdio.h>
#include <stdlib.h>
struct individu {
   char nom[15];
   int  age;
};

void main() {
   struct individu eleve;
   int taille = sizeof (struct individu);

   FILE* ficIn = fopen("c:\\temp\\original.bin", "rb");
   if (ficIn == NULL) exit(0);
   FILE* ficOut = fopen("c:\\temp\\copie.bin", "wb");
   if (ficOut == NULL) exit(0);

   // Boucle de recopie
   while ( feof(ficIn) == 0 ) {
       fread (&eleve, taille, 1, ficIn);
       fwrite(&eleve, taille, 1, ficOut);
   }

   fclose(ficIn);
   fclose(ficOut);
}
```


Autres petites choses ...

Les arguments de `int main(...)`

```
int main (int argc, char **argv) { ... }
```

➽ Explication des arguments (variables pré-définies)

- `int argc` : nombre d'argument transmis au prg principal
- `char **argv` : tableau de chaînes de caractères contenant les arguments transmis

➽ Fonction avec un nombre d'arguments variable

- `argv[0]` : chaîne contenant le nom du programme
- `argv[argc]` : contient \0
- Le premier argument est donc `argv[1]` et le dernier `argv[argc-1]`

➽ Exemple

```c
#include "stdio.h"

void main(int argc, char **argv) {
  int i;
  double d;

  printf("\nNombre d'arguments : %d :\n", argc);
  for(i=0;i<argc;i++)
    printf("  Arg #%d : %s\n", i, argv[i]);
}
```

démo 25

➽ Si un argument attendu est un nombre entier ou réel, on peut utiliser les fonctions `int atoi(char *s)`, `long atoi(char *s)` et `double atoi(char *s)` de `stdlib.h`.

Exercice : écrire un programme `Add` qui fait la somme de tous les nombres entiers qui lui sont transmis. Ainsi l'appel `./Add 1 2 3` doit afficher 6.

```
#include "stdio.h"
void main(int argc, char **argv) {
  int i, res = 0;
  for(i=1; i<argc; i++)
    res += atoi(argv[i]);
  printf("\nResultat = %d", res);
}
```


Pointeur de fonction

➽ Pointeur pointant vers une fonction (et non plus sur un nombre) ! Il s'agit donc d'une variable qui peut contenir l'adresse d'une fonction.

```
void AjouteUn(int *x) {
  *x += 1;
}

void  (*inc)(int *z); //déclaration
inc = AjouteUn;       //initialisation

int a = 10;
inc(&a);              //incrémente a de 1
```

➽ Exemple

Sans pointeur de fonction

```
#include "stdio.h"
double fct_1(double x) { return 3.0*x; }
double fct_2(double x) { return x/3.0; }
int main() {
    int     f;
    printf("\nFonction 1 ou 2 : "); scanf("%d", &f);
    if (f==1)
        printf("\n fct(%lf) = %lf", 9.0, fct_1(9.0));
    else
        printf("\n fct(%lf) = %lf", 9.0, fct_2(9.0));
    if (f==1)
        printf("\n fct(%lf) = %lf", 12.0, fct_1(12.0));
    else
        printf("\n fct(%lf) = %lf", 12.0, fct_2(12.0));
    return 0;
}
```


Avec pointeur de fonction : démo 26

```
#include "stdio.h"
double fct_1(double x) { return 3.0*x; }
double fct_2(double x) { return x/3.0; }

int main() {
    int     f;
    printf("\nQuelle fonction (1 ou 2) : "); scanf("%d", &f);

    double (*pfct)(double z);
    if (f==1)
        pfct = fct_1;
    else
        pfct = fct_2;

    printf("\n fct(%lf) = %lf", 9.0, pfct(9.0));
    printf("\n fct(%lf) = %lf", 12.0, pfct(12.0));
    return 0;
}
```

Les énumérations

➽ Sert à définir des variables entières qui ne peuvent prendre que certaines valeurs précises :

➽ Exemple

```
enum color {noir, bleu, vert, cyan, rouge, magenta, blanc};
typedef enum color couleur;
couleur c1 = magenta;
printf("\nCouleur = %d", c1); // affiche 5
```

➽ Remarque : Identifié comme des entiers

```
enum bascule{ON, OFF,  NON = 0, OUI};
```

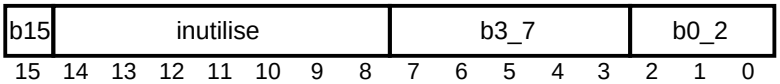
équivaut à `enum bascule{ON=0, OFF=0, NON=0, OUI=1}`

➽ Remarque : Alternative à `#define`

Les champs de bits

➽ **Exemple :**

```
struct chps {
  unsigned int b0_2 : 3;
  signed int b3_7   : 5;
  unsigned int      : 7;
  unsigned int b15  : 1;
}
```



➽ Remarques :

- ⇨ Les type sont soit `int` (équiv. `signed int`), soit `unsigned int`.
- ⇨ Ils peuvent entraîner des problèmes de portabilité entre machines (« Low Indian » et « Big Indian »)

UNIX/Linux, commandes élémentaires

Description	**Commande**	**Anglais**
Affichage du répertoire courant	`pwd`	
Création d'un nouveau rép. `monRep`	`mkdir monRep`	*make directory*
Changement de répertoire	`cd monRep`	*change directory*
Retour au répertoire précédent	`cd ..`	
Liste des fichiers du rép. courant	`ls`	*list*
Création d'un fichier vide appelé `monFic.c`	`touch monFic.c`	
Déplacement d'un fichier vers un répertoire `monRep`	`mv monFic.c monRep`	*move*
Copie de `fichier1` dans `fichier2`	`cp fichier1 fichier2`	*copy*
Renommage de `fichier1` en `fichier3`	`mv fichier1 fichier3`	*move*
Suppression d'un fichier	`rm monFic`	*remove*
Suppression d'un répertoire complet	`rm -r monRep`	
Edition d'un fichier	`nedit monFic &`	`nedit` : editeur
Compilation de `monFic.c`	`gcc monFic.c` `gcc monFic.c -o monFic`	exécutable `a.out` exécutable `monFic`
Exécution du programme `monFic`	`./monFic`	

F I N