Introduction au langage C

29 août 2012

Origine et caractéristiques du C

- utilité : langage d'assemblage portable
- pour noyau de système (unix)
- opérations de bas niveau (champs de bits)
- manipulation fine de mémoire : pointeurs de n'importe quoi
- orientation objet rajoutée après coup : C++
- années 1970, époque FORTRAN /PASCAL
- 1970 : aider le compilateur à optimiser (+=)
- des défauts sur modularité/compilation séparée
  - module = fichier,
  - pas d'espaces de noms par module
  - exportation par défaut

- Typage laxiste, constructions syntaxiquement correctes, sémantiquement problématiques genre "implementation dépendent" (++)
- Portabilité sur toutes machines, toutes tailles : fonctionnalités absentes (exceptions) ou gérées en bibliothèque (chaînes de caractères, allocation dynamique)
- Vérifications dynamiques à l'exécution absentes (bornes d'un tableau).
- Pièges syntaxiques (affectation et comparaisons, break dans switch)
- Si syntaxiquement correct, sera compilé, même si çà n'a pas du tout le sens que pensait le programmeur.

Plan de progression

1. Introduction aux premiers éléments de syntaxe : variables, boucles, fonctions.

2. Retour sur les types : tailles, constantes, notion d'adresse/pointeur, tableaux, classes de stockage et allocation dynamique de mémoire.

3. Chaînes de caractères, paramètres pointeurs et tableaux, structures, déclaration et paramètres de main, accès aux fichiers.

Déclaration de variables

attributs typevar nom_variable ;
attributs typevar nom_variable = valeur_initiale ;

Indique au compilateur que nom_variable est de type typevar :

- sa taille en octets est sizeof(typevar) : pour allouer la mémoire de stockage, pour lire ou modifier le contenu
- manière d'interpréter des bits/octets du contenu (entier relatif, entier naturel, nombre à virgule ?)

Demande au compilateur d'allouer un emplacement de stockage

```
int x;
long l1, l2;  /* déclare 2 var d'un coup (éviter) */
float racine2 = 1.414;
```

Opérateur d'affectation

Syntaxe : gauche_contenant = droit_expression ;

Sémantique : gauche_contenant $\underleftarrow{ppnc}$ valeur(droit_expression)

g $\underleftarrow{ppnc}$ d : g **p**rend **p**our **n**ouveau **c**ontenu valeur(d)

Exemple : y = 3*x - y + 2 ;

y $\underleftarrow{ppnc}$ 3*contenu(x) - contenu_actuel(y) + 2

A noter : ce n'est **pas** := comme en ADA ou CAML !

Formes abrégées d'affectation

```
                    /* code équivalent */
x -= 3              x = x - 3;
y += 2              y = y + 2;

/* à droite : ancienne valeur */
z = x++             z = x;
                    x = x + 1;

/* à gauche : nouvelle valeur */
y = --x;            x = x - 1;
                    z = x;
```

Opérateurs de calcul arithmétique et comparaisons

arithmétique

- +, -, * : addition, soustraction/opposé, multiplication
- / : division (entière ou flottante selon type opérandes)
- % : modulo (reste de la division entière)

comparaisons

- <,<=, >, >= : inférieur/supérieur avec/sans égalité
- != : non égalité
- == : égalité (**double égal !**)

ATTENTION : égalité ADA/CAML =, égalité C ==

En C, l'affectation est une expression légale

Une affectation est une expression au même titre qu'une addition

Sa valeur est égale à celle de son membre droit

Effet de bord : valeur de droite stockée dans membre gauche

```
/* Code horrible mais légal (à éviter !) */
y = 4 * (z = 3 * x + 1) + 5;

/* code équivalent */
z = 3 * x + 1:
y = 4 * z + 5;
```

Types d'instruction

- expression suivi d'un point-virgule
- bloc d'instructions groupées par des accolades
- appel de fonction
- construction conditionnelle (if, while, for, switch)

```
y = x + 1;    /* instr classique : affectation */
x + 2     ;   /* instr ne modifiant rien */
          ;   /* instr vide */

{              /* bloc d'instructions avec */
int var_loc;   /* déclaration de variable locale */
instr1;
instr2;
...
instrn;
}
```

Instruction if

if (condition) instr // pas de then !

if (condition) instr_alors **else** instr_sinon

```
if (v<10)
   printf ("v inferieur a 10 \n");

if ((x > 0) && ((int)log10(x) < 3))
   {
   printf ("x dans l'intervalle ]0, 1000[\n");
   x = x / 2.5;
   }
else
   printf ("pas de nombre <= 0 ou >= 1000!\n");
```

If : else ambigu associé à if le + proche

```
if (c1)
  {
  if (c2)
    printf ("cas c1 et c2\n");
  }
else
  printf ("cas non c1\n);

if (c1)
  if (c2)
    printf ("cas c1 et c2\n");
  else
    printf ("cas c1 et non c2\n);
```

Instruction while

Répétition de l'exécution en boucle d'une instruction

while (cond) instruction_corps

cond : condition d'exécution du corps

sortie de boucle lorsque **cond fausse**

Attention : **pas de ;** après la condition

Affectation légale dans condition : **vérifier** == et non =

```
//     v !!
if (x = y)
  printf ("egalité\n);
```

```
/* code équivalent */
x = y;
if (y != 0)
    printf ("egalité\n);
```

```
while ((c = f()) != d)
  {
  g ();
  }
```

```
c = f ();
while (c != d)
  {
  g (); c = f();
  }
```

```
//              v !!
while (x != y);
  x = x + 1;
```

```
while (x != y)
  { /* corps vide ! */ }
x = x + 1;
```

Boucle for

for (initialiser ; condition ; mettre_à_jour)

for (i=0 ; i<=N-1 ; i=i+1)

```
/* Parcourir [0, N-1] : for ou while */

                          i = 0;                  /* init  */
for (i=0; i<N; i++)       while (i<N)             /* cond  */
  {                         {
  x = y + 1;                x = y + 1;            /* corps */
  logt = log (t);           logt = log (t);
  }                         i++;                  /* maj   */
                            }
```

Champs vides

On peut laisser vide :

- l'initialisation (variables déjà initialisées)
- la condition (toujours vraie $\rightarrow$ boucle infinie)
- la mise à jour (réalisée par effet de bord dans le corps)

Attention à ne pas utiliser , (groupement d'expressions comme une seule) à la place de ; (délimiteur).

Définition des fonctions et procédures

Prototype : déclare le type d'une fonction

- spécifie type du résultat + types {et noms} des paramètres
- $type_{ret}$ nom_fonc ($type_1$ nom_1, ..., $type_n$ nom_n) ;
- $type_{ret}$ nom_fonc (**void**) ;
- **void** nom_proc ($type_1$ nom_1, ..., $type_n$ nom_n) ;
- (<C99) : sans $type_{ret}$, implicitement $type_{ret}$ = int.

```
float fmaxf (floatx, float y);
double fmax(double x, double y);
void exit (int status);                    /* procédure */
int getchar (void)                  /* pas de paramètre */
```

Définition : prototype suivi du corps

- prototype **avec** noms des paramètres.
- corps : bloc d'instructions avec/sans déclarations de variables locales au début. La définition **alloue de la mémoire** pour le stockage des instructions du corps.
- dernière instruction du corps d'une fonction : **return**, suivie de la valeur retournée à l'appelante
- dernière instruction du corps d'une procédure : **return** facultatif.
- instruction return (+ valeur retournée si fonction) au milieu du corps : retour immédiat à l'appelante.

Appel d'une fonction

Paramètres :

- formels : dans le prototype de la fonction
- effectifs/réels ou arguments d'appel : expressions dans l'instruction d'appel

Passage **par valeur** :

- évaluation paramètres effectifs : valeurs $\rightarrow$ paramètres formels
- exécution du corps de la fonction
- fonction appelée **ne peut pas modifier** la valeur d'une **variable** passée par l'**appelante**.

Si expression argument est juste une variable :
copie de sa valeur $\rightarrow$ paramètre formel

```
int max2 (int, int);  /* prototype sans noms arg */

/* prototype de max2 doit précéder */
int max_pondere_3 (int a, int b, int c)
{
  int resultat;
  resultat = max2 (max2 (a,2*b), 3*c);
  return résultat;
}

int max2 (int x, int y) /* vraie definition */
{
  if (x <= y)
     return y;
  else
     return x;
}
```

```
long var = 5;
long triple;

long xtrois (long x, long y)
{
  long res;
  res = 3*x;
  y = 1;  /* copie_var_dans_y=1 et non var=1! */
  return res;
}

void main (void)
{
  /* avant appel : var=5 */
  triple = xtrois (2L,var);
  /* après appel : var=5, triple=6 */
}
```

Analyse par cas (selon) : switch/case

Analyse par cas : comparer expr à une liste de constantes

```
switch   expr {
         case c_1 :   instructions_cas_c_1
                      break ;                    /* optionel */
         ...
         case c_j :   instructions_cas_c_j
                      break ;
         ...
         case c_n :   instructions_cas_c_n
                      break ;

         default :    instructions_autres_cas
         }
```

Sans break en fin de cas **enchaîne** avec **cas suivant**

```
switch  lu {
        case '0' :  | lu -= '0' ;  |  exécuté si '0'
        case 0 :    | break ;      |  termine exec switch

        case '1' :  | lu -= '0' ;  |  exécuté si '1'
        case 1 :    | valeur ++ ;  |  exécuté si '1' ou 1
                    | break ;      |  terminé pour 1 ou '1'
        case '+' :
        case '-' :  | break ;      |  rien si '+' ou '-'

        default :   | ok = 0 ;     |  exécuté autre cas
        }
```

Plan de progression

1. Introduction aux premiers éléments de syntaxe : variables, boucles, fonctions.

2. Retour sur les types : tailles, constantes, notion d'adresse/pointeur, tableaux, classes de stockage et allocation dynamique de mémoire.

3. Chaînes de caractères, paramètres pointeurs et tableaux, structures, déclaration et paramètres de main, accès aux fichiers.

Ecriture des entiers en base B

poids forts poids faibles

$$x_{n-1}\ x_{n-2}\ \ldots\ x_i\ \ldots\ x_2\ x_1\ x_0\ \_B$$

écriture en base de numération B (10 par défaut)

de l'entier $\sum_{i=0}^{n-1} x_i B^i$

$$236 = 236\_10 = 2x10^2 + 3x10^1 + 6$$

$$236\_10 = 354\_8 = 3x8^2 + 5x8^1 + 4$$

écriture à B chiffres

base 10 : 0 à 9 (décimal)

base 10 : 0 à 9 et a à f (hexadécimal)

base 2 : 0 et 1 (binaire)

Tailles et unités de stockage

Mémoire des ordinateurs : représentation des entiers en base 2

Unité : le bit (**bi**nary digi**t** : 0 ou 1)

- bit = chiffre d'un entier en base 2 (entier 0, entier 1)
- bit = élément booléen (algèbre de Boole : 0/faux, 1/vrai)

Octet (byte en anglais) : paquet de 8 bits

Ne confondez pas bit et byte !

Taille minimale d'accès/**unité d'allocation mémoire : octet**

Identification d'emplacement : adresse = numéro (1er) octet

Opérateur **sizeof**(T) : taille en **octets** du type T

Types C : les entiers (cf table en début de doc)

Type de base **int** : entier de taille standard

Naturel/relatif = interprétation des n bits de contenu :

- **unsigned int** : bits représentant 1 **naturel** ($[0, 2^n - 1]$).
- **int** : bits représentant 1 **relatif** ($[-2^{n-1}, +2^{n-1} - 1]$)

Attributs optionnel de taille : short/long/long long

Type spécial d'entier court sur 8 bits : char

Deux manières de définir la taille du type entier :

- **octets** (allocation de mémoire) : **sizeof**(int) $\in \{4,8\}$
- **bits** (nombre de chiffres binaires) : **8*sizeof**(int) $\in \{32,64\}$

Constantes : #define et const

- Directive #**define** : effectue une **substitution de texte** (comme chercher/remplacer dans éditeur de fichiers) → pas d'adresse.
- Atrribut **const** : le contenu de la variable ne doit pas être modifié (à stocker dans une section protégée contre écriture). On peut prendre l'adresse de cette "variable à contenu constant".
- Tailles de tableau : pas de const, toujours #define

```
const float rac2 = 1.414; // ptr=&rac2 autorisé
#define PI       3.14     // @@ pas de ; en fin de ligne
float aire = PI*r*r;;     // --> float aire = 3.14*r*r;
```

Mieux que #define : définir un type enuméré

- ```
  #define NOIR  0
  #define ROUGE 1
  #define VERT  2
  #define JAUNE 4
  #define BLANC 7
  #define BLEU  8
  #define ORANGE 13
  int ma_couleur;
  ```
- ```
  enum couleur {NOIR, ROUGE, VERT, JAUNE,
                BLANC = 7, BLEU, ORANGE=13};
  enum couleur ma_couleur;
  ```
- ```
  ma_couleur = VERT;   // dans les 2 methodes
  ```
  ```

Types C : renommage de type avec typedef

Syntaxe : typedef type_existant nouveau_type

typedef unsigned long size_t ;

size_t devient synonyme du type unsigned long

Application : typage du paramètre de malloc pour toute machine

```
/* mon_prog.c */
#include <stdlib.h>
...
size_t taille;
taille = n * sizeof(int);
nouveau = malloc (taille);
```

Types C : exemple d'utilisation de typedef

```
/* Fichier /usr/include/stdlib.h */

typedef unsigned long size_t;
                                /* ou  bien */
typedef unsigned long long size_t;
                                /* ou  bien */
typedef unsigned int size_t;
                           /* selon  machine */

/* prototype de malloc : paramètre size_t */
void *malloc (size_t taille);
```

Interprétation d'un entier comme un booléen

Un entier peut être interprété comme une valeur booléenne

0 : faux autre que 0 : vrai

Opérateurs booléens et comparaisons retournent entier 0 ou 1

ET puis OU alors NOT : **&&** **||** **!**

```
// Opérande de droite évalué seulement si nécessaire
if ((x != 0) && (y/x > 2))
   ...
```

Attention : ne pas oublier de doubler **&** et | :

&& **||** **|** : un booléen = l'entier

& **|** **~** : 8*sizeof(int) booléens = chaque bit de l'entier

Types C : nombre à virgule et cas particuliers

- **float** et **double** : nombres à virgule flottante (2 tailles).
- **char** : type spécial d'entier court pour la représentation des codes des caractères sur 1 octet (ASCII/ISO-latin1).
- **void** : indication d'absence de type (procédures et pointeurs).
- chaîne de caractères : tableau **char[ ]** et marque de fin (0)
- **pas de bool** : renommer éventuellement bool un type entier pour clarifier l'usage (mais ne crée pas un contrôle de type fort)

```
enum bool {false=0, true=1};  // ou typedef int bool
bool terminer;
terminer = 2;   // pas rejeté par le compilateur
```

Conversion ou forçage de type

Syntaxe : (nouveau_type) expr

- expr convertie en nouveau_type ou
- expr interprétée comme de nouveau_type

```
int x = 3;
int y = 4;
int divent;
float divfloat;

divent = y/x;                 /* 4/3 = 1              */
divfloat =
   (float) (y/x);             /* 4/3 = 1 --> 1.0      */
divfloat =
  (float) y / (float) x;      /* 4.0 / 3.0 = 1.25     */
```

Caractères : propriété du codage ASCII

L'ordre des codes réflère l'ordre alphabétique des lettres et l'ordre des valeurs entières associées aux chiffres :

- 'A' à 'Z' : codes consécutifs [65,90] ([0x41,0x5A])
- 'a' à 'z' : codes consécutifs [97,122] ([0x61,0x7A])
- '0' à '9' : codes consécutifs [48,57] ([0x30,0x39])

- ASCII('c') = ASCII('a') + 2 = 0x63
- ASCII('0') = 48 = 0x30 $\neq$ **0**
- ASCII('3') = ASCII('0') + 3 = 0x33 $\neq$ **3**
- ASCII('a') = ASCII('A') + 0x20
- Code ASCII 0 : pseudo-caractère non imprimable **nul**. (Rien à transmettre/erreur de perforation de la carte).

Typage des caractères en C : variante d'entier

- pas de vrai type caractère ($\neq$ entier) à la ADA/CAML
- le type **char** est un type entier court comme short int
- mais (sizeof(char) = 1) : un octet suffit pour stocker un code ASCII/iso-8859-x
- conversion char $\leftrightarrow$ int inutile : 'A' **est** l'entier 65
- 3 notations synonymes du même entier : 'A', 65, 0x41
- rang(x) dans l'alphabet : x - 'a' + 1
- entier x $\in$ [0,9] $\rightarrow$ caractère chiffre $\in$ ['0','9'] : x + '0'
- caractère chiffre c $\in$ ['0','9'] $\rightarrow$ entier $\in$ [0,9] : c - '0'

Directive #include

Les fichiers **bibliotheque_xxx.h** contiennent les informations sur les fonctions de bibliothèques standard. On y trouve :

- des **définitions de types** (enum, typedef) utilisées dans les fonctions,
- des **définitions** par #define de **constantes** utilisées par les fonctions,
- les **prototypes** (sans corps) des **fonctions** : indique la manière dont elles doivent être appelées.

La directive #**include <en_tete.h>** indique au compilateur d'**insérer une copie** du contenu du fichier en_tete.h dans le code C au moment de la compilation.

La directive #**include <stdio.h>** permet de récupérer la description des fonctions d'entrée/sortie standard de la bibliothèque C (printf, scanf, etc).

Opérateurs unaires & et *

Variable de type T : occupe sizeof(T) octets consécutifs.

Opérateur unaire &var : adresse de (1er octet de) var
Opérateur unaire *adr : emplacement mémoire d'adresse adr

```
short x;
short y;
x = y;      /* peut aussi s'écrire */      *&x = *&y;
```

Mem[&x] $\xleftarrow{ppnc\_16\ bits}$ Mem[&y]

1 accès mémoire, sizeof(short)=2 copies d'octet en // :

Mem[&x] $\xleftarrow{ppnc\_8bits}$ Mem[&y]
Mem[&x+1] $\xleftarrow{ppnc\_8bits}$ Mem[&y+1]

Tableaux : déclaration et réservation de place

Pas de vrai type tableau

- indiçage toujours à partir de 0 : tab [0] à tab [TAILLE-1]
- nom du tableau = adresse 1er élément : tab est synonyme de &(tab[0])
- définition = déclarer type + allouer TAILLE éléments contigus en mémoire
- (hors fonctions :) spécification de valeur initiale possible

- type_elem montab [N] occupe N*sizeof(type_elem) octets.
- La taille n'est pas stockée en mémoire avec les éléments.
- Pas de contrôle à l'exécution : indice hors bornes non détecté.

Exemples de déclaration et initialisation

```
#define NB_ELEM 4    /* pas de const --> #define */

int montabint [NB_ELEM];

/* avec valeurs initiales */
/*                vect3[0]  vect3[1]  vect3[2] */
float vect3 [3] = {1.414, 2.718,    3.14};

/* avec premières valeurs initiales */
/* t[2] et t[3] : vi = 0 par défaut */
int t [NB_ELEM] = {-3,2};

/* Nb_elem=7 déduit de initialisateur */
int tab [ ] = {1,2,4,8,16,32,64};
```

Calcul des adresses : sans vérification

Allocation contigüe en mémoire

tab[0]	est à l'adresse	tab (adresse de début du tableau)
tab[1]	est à l'adresse	tab + taille d'un élément (en octets)
tab[2]	est à l'adresse	tab + 2*taille d'un élément
tab[j]	est à l'adresse	tab + j * taille d'un élément

Pas de vérification à l'exécution

```
int x = 4;
int t[3] = {1,2,3};          /* t[0], t[1] et t[2] */
int y = 5;
t[-1] = 0;  /* illégal : effet probable x = 0; */
t[3] = 4;   /* illégal : effet probable y = 4; */
```

Boucle de parcours par indice

```
#define N 5

short int ts [N] = {1,2,4,8,16};

int indice;

for (indice=0; indice<N; indice++)
    ts[indice] = 3*indice;
```

Variables de type pointeur

Variable pointeur de T :
stockage de l'adresse de quelquechose (variable) de type T

.

Comment pourrait-on noter le type
"constante adresse ou variable pointeur de T" ?

1. & T : pas retenu en C (cf référence à un objet en C++)
2. T * : retenu en C $\rightarrow$ appliquer l'opérateur * à un (T *) donne un (quelquechose de type) T.

Syntaxe de déclaration = syntaxe d'utilisation.

A noter : $\forall$ T, sizeof(T *) = taille d'une adresse

Pourquoi préciser le type d'objet reperé ?

- Cohérence : * adr_de_short utilisé comme un float ?
- Combien d'octets lire ou écrire avec * ?
- Comment interpréter le contenu repéré ?

Pointeur de type **void *** :

- Accepte tout type d'adresse
- Conversion obligatoire en pointeur de ($\neq$ void)
  - avec forceur de type (T *)
  - avant application de * sur le pointeur
  - var_de_type_T = * (T *) ptr_de_void ;

NULL : adresse illégale

- Pointeur ne repère rien : ***NULL** $\rightarrow$ erreur "seg. fault"
- NULL correspond souvent à **(void *) 0L**
- Valeur initiale par défaut d'un T * (hors fonction).

Taille pointeur $\neq$ taille élément pointé

- Stockage d'un pointeur = stockage d'une adresse
- Ne dépend pas du type d'objet repéré
- sizeof (long) $\geq$ sizeof (short) $\geq$ sizeof (char)
- sizeof (int *) = sizeof (long *) = sizeof (short *) = sizeof (void *)

```
int x;
int y;

int *p = &x; /* p déclaré initialisé     */
int *q;      /* q déclaré non initialisé */

x = 5;       /* signifie *&x = 5 */
*p = 3;      /* p=&x d'où *&x=3 : x=3 */

p = &y;      /* p repère maintenant y  */
*p += 1;     /* *&y += 1, d'où y += 1 */

void *pvoid; /* pointeur de n'importe quoi   */
pvoid = &y;  /* repère y, mais sans type...  */
             /* *pvoid = 3 est  illégal      */
/*  utiliser pvoid comme si déclaré (int *)  */
* (int *) pvoid = 3;                  /* y = 3 */
```

Plan de progression

1. Introduction aux premiers éléments de syntaxe : variables, boucles, fonctions.

2. Retour sur les types : tailles, constantes, notion d'adresse/pointeur, tableaux, classes de stockage et allocation dynamique de mémoire.

3. Chaînes de caractères, paramètres pointeurs et tableaux, structures, déclaration et paramètres de main, accès aux fichiers.

Classes de stockage des variables

Variables globales ($\notin$ bloc instructions/fonction) :

- un seul exemplaire, **allocation statique**
- adresse fixe définie à la compilation
- mémoire allouée et initialisée en début d'exécution du programme
- libérée en fin d'exécution du programme
- valeurs initiales explicites : sections du fichier exécutable pour constantes (text, rodata) ou variables (data).
- sans valeurs initiales (bss) : initialisation implicite à 0.

Dimensionnement figé à la compilation

Variables locales (fonctions, bloc d'instructions)

- plusieurs exemplaires si récursion
- **allocation automatique** dans la pile au début et pour la durée de l'exécution du bloc
- **adresse varie** suivant ordre des appels de fonction
- **libération automatique** de l'espace de stockage en fin d'exécution du bloc $\rightarrow$ réallocation possible à autre chose dès prochain appel d'une fonction
- **pas d'initialisation** à 0 **par défaut**
- initialisateur = instructions d'initialisation ajoutées au prologue de la fonction

Dimensionnement à l'exécution **mais**

- la variable **n'existe plus après le retour** à l'appelante
- **ne pas retourner** à l'appelante/affecter à un pointeur global l'**adresse** d'une variable **locale**

Allocation **dynamique** explicite

- **allocation explicite** par le programmeur (**malloc** ou **calloc**)
- dans le tas (extension dynamique zone var. glob./bss)
- **libération explicite** par le programmeur (**free**)
- **pas de libération automatique** des blocs orphelins par "ramasse-miettes" (pas de "garbage collector")
- initialisation à 0 à l'allocation : calloc/oui malloc/non
- déclarer un pointeur **n'alloue pas** de mémoire pour le stockage pour la donnée repérée

```
montype *p_var, *p_tab;
// allocation : variable +  tableau de NB_ELEMENTS
p_var = (montype *) malloc (sizeof(montype));
p_tab = (montype *) calloc (NB_ELEMENTS,sizeof(montype));
```

- **vérifier** adresse retournée par malloc $\neq$ NULL
- oubli de free après utilisation = risque de "fuite" de mémoire
- free plusieurs fois pour le même malloc = erreur (corruption des structures de données de bibliothèque d'allocation dynamique)
- gérer correctement les copies des adresses d'éléments alloués dynamiquement : surveiller
  - pointeurs locaux n'existant que pour la durée de l'appel
  - free sur éléments alloués dynamiquement et contenant des pointeurs vers d'autres éléments alloués dynamiquement
  - réaffectation de pointeurs

Chaînes de caractères

- sous forme de tableaux de char (ASCII, ISO-Latin1)
- nom du tableau ne donne pas de taille : chaîne terminée par le caractère **nul** de code 0 (qu'on peut aussi noter '\ 0').
- "fou" synonyme de {'f','o','u','\0'} ou {'f','o','u',0}
- "fou" : chaîne de 3 caractères, occupe 4 éléments char.
- allocation dynamique : ptr = (char *) malloc(strlen(chaine)+1)
- tableau chaîne = un seul paramètre (marque de fin de chaîne dispense de passer la taille)

Quelques fonctions sur les chaînes

Paramètre **const char *** ou **const char [ ]** : la fonction ne modifie pas la chaîne de caractères.

- size_t **strlen** (const char *s) : longueur de la chaîne sans la marque de fin de chaîne.
- char ***strncpy** (char *dest, const char *src, size_t n) recopie de chaîne dans tableau dest existant de taille n.
- int **strcmp** (const char *s1, const char *s2) : comparaison de chaînes
  - 0 : contenus de s1 et s2 identiques
  - < 0 : s1[j] $<$ s2[j] avec j rang 1^{er} caractère $\neq$ (s1 avant s2)
  - > 0 : s1[j] $>$ s2[j] avec j rang 1^{er} caractère $\neq$ (s1 après s2)
- liste des fonctions : **man 3 string**

Exemple

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

#define LONGUEUR_NOM_MAX 75   // max 74 caractères

char nom [LONGUEUR_NOM_MAX];

const char coucou [] = "bonjour";
const char oui [] = {'o','u','i',0};
char *ch_dyn;
...
ch_dyn = (char *) malloc (strlen(coucou) + 1);
if (ch_dyn == NULL) { /* traiter erreur */}
strcpy (ch_dyn, coucou);   /* dupliquer la chaîne */
```

```
void main (void)
{
   printf ("Quel est votre nom ?\n");
   if ((fgets (nom,LONGUEUR_NOM_MAX,stdin)!=NULL) &&
      (strlen(nom) >0))
     {
     // attention : strcmp retourne 0 sur égalité
     if (!strcmp (nom, "Duconlajoie"))
       printf ("Sacré farceur !\n");
     else
       printf ("Bonjour Monsieur %s\n",nom);
     }
}
```

Chaînes : divers

Egalité des contenants $\neq$ égalité des contenus :

- ptr_s1 et s2 désignent le même contenant (même adresse) : contenu unique sous deux noms différents
  - affectation ptr_s1 = s2
  - test ptr_s1 == s2
- ptr_s1 et s2 désignent deux contenants différents dans lequels la chaîne est dupliquée
  - affectation strcpy(ptr_s1,s2),
  - test !strcmp(ptr_s1,s2)

Attention à l'ordre alphabétique pour les caractères accentués (ISO-latin1) : 'd' $<$ 'e' $<$ 'f', mais 'é' $\notin$ ['a','z']

```
printf ("bonjour"\n");
/* equivalent à */
const char tab_bonjour[] = "bonjour\n";
printf (tab_bonjour);
```

A éviter : lecture de chaînes de taille non bornée au clavier $\rightarrow$ impossible de prévoir un tableau destination de taille suffisante.

- scanf ("%s", tab_chaine_a_remplir) ; // NE PAS UTILISER
- gets(tab_chaine_a_remplir) ; // NE PAS UTILISER
- préférer strncpy, strncat à strcpy, strcat

Oublier de terminer par 0 : boucle $\pm$ infinie

Paramètre résultat (équiv. OUT/INOUT/VAR)

Passage par valeur de pointeur = passage par adresse
= possibilité de modifier variable repérée

```
int vglob;

/* par fonction
int plus1 (int x)
{
   return x+1;
}

void main (void)
{
   vglob =
      plus1 (vglob);   }
}
```

```
/* var à incrementer */

 par procédure */
void incrementer (int *v)
{
  (*v) = (*v) + 1;
}

void main (void)
{
  incrementer (&vglob);
```

Tableaux : passage en paramètre

Pour pouvoir traiter des tableaux de toutes tailles, il faut déclarer **deux** paramètres :

- nombre d'éléments du tableau
- adresse 1er élément (nom du tableau **sans & devant**)

1. void tabx2 (unsigned int nb, long t[])
2. void tabx2 (unsigned int nb, long t[nb]) /* C99 */
3. void tabx2 (unsigned int nb, long *t)

Version 3 moins précise :

- t repère (1er élément de) tout un tableau ?
- t repère une simple variable à modifier ?

```
/* fichier tab.h */
#define TAILLE_MONTAB 10
extern int montab[];

/* fichier tab.c */
int montab[TAILLE_MONTAB];

void  tabx2 (unsigned int nb, long t[])
{
   for (i=0; i<nb_elements;i++)
      t[i] = 2*t[i];
}

void main (void)
{
   tabx2 (TAILLE_MONTAB, montab);
}
```

Précautions avec sizeof(tableau)

Deux manières de calculer la taille de stockage d'un tableau t :

1. toujours correcte : nb_élém * sizeof (type_élém) ;
2. à manier avec précaution : sizeof(t) ;

Type dépend de taille entre crochets :

- int tab[6] : tableau, sizeof(tab) = 6 * sizeof(int)
- int tab[] : pointeur, sizeof(tab) = taille d'une adresse

```
madelbrot> ./sizeof
main : taille long : 4
main : taille t    : 24
afficher : taille de t = 4
```

```c
#include <stdio.h>

long tab [6] = {0,1,2,3,4,5};
void afficher (long t[])  /* oubli de passer taille */
{
  int i;
  printf ("afficher : taille de t = %d\n", sizeof(t));
}
int main(void)
{
  printf ("main : taille long : %d\n", sizeof(long));
  printf ("main : taille t    : %d\n", sizeof(tab));
  afficher (tab);
  return 0;
}
```

Printf et scanf

nb_car = printf ("format",e1, e2, . . . , en) ;

- retourne le nombre de caractères écrits (0 → erreur !)
- souvent utilisé comme une procédure (nb_car ignoré)
- texte seul : printf ("Affiche juste un texte\n") ;
- printf ("expc=**%c** et expf=**%f**\n", **expc**, **expf**) ;
  %c et %f remplacés par valeurs de expc (caractère) et expf (float)
- '\n' : caractère "line feed" (saut à la ligne)

nb_var_lues = scanf ("%d",&varint) ;

- retourne nb valeurs lues → **vérifier == 1** !
- modifie variable → passer adresse : **&**varint
- éviter lecture de chaînes avec (%s) : taille non bornée !

Structures (record ADA, n-uplet CAML)

- groupement d'éléments de types différents
- sizeof (type_struct) $\geq \Sigma sizeof(membre)$: alignement de chaque membre sur une adresse multiple de sa taille (et entre structures dans un tableau)
  $\rightarrow$ des octets non utilisés entre membres.
- définition du type struct xxx
- création éventuelle d'un type synonyme avec typedef
- déclaration de variables ou de pointeur de struct xxx
- accès à un membre : var_struct.membre
- accès à un membre par pointeur : **ptr -> membre** synonyme de (*ptr).membre

```
/* renommage du type : évite répétition de struct */
typedef struct _coord_pol coord_pol;

/* définition du type struct _coord_pool */
struct _coord_pol
  {
  float r;
  float theta;
  };

/* déclaration de variables */
coord_pol point1;
struct _coord_pol point2 = {2.5, 1.57};
coord_pol carre[2];

/* un pointeur de structure */
coord_pol *pointe_point = &point1;
```

```
coord_pol tourner (coord_pol point)
{
   coord_pol nouveau_point;
   nouveau_point=point;
   nouveau_point.theta += 3.14/4;
   return nouveau_point;
}

void main (void)
{
   point2.r = 0.0;
   point1 = point2;
   *pointe_point = (coord_pol) {1.0,0.0};
   point2.theta = pointe_point -> theta;
   carre[1] = tourner (point1);
   printf ("rayon(point1) = %f\n", point1.r);
}
```

Fichiers : identification des fichiers ouverts

- pointeur d'élément de table des fichiers ouverts
- table gérée par la bibliothèque C : tableau de FILE
- type FILE défini dans stdio.h : struct { //infos sur fichier ... }

- **stdin** : entrée standard prévouverte (par défaut clavier)
- **stdout** : sortie standard prévouverte (par défaut écran)
- **stderr** : sortie prévouverte (messages d'erreur : par défaut écran)

$$
\begin{aligned}
\text{scanf("\%d", \&var)} &\rightarrow \text{fscanf(stdin, "\%d", \&var)} \\
\text{printf ("\%lu",val)} &\rightarrow \text{fprintf(stdout,"\%lu", val)} \\
\text{putchar('Z')} &\rightarrow \text{fputc('Z',stdout)} \\
\text{car=getchar()} &\rightarrow \text{car = fgetc(stdin)}
\end{aligned}
$$

Fichiers : ouverture et fermeture

FILE *fopen(const char *path, const char *mode) ;

- path : chaîne = chemin d'accès (nom) du fichier à ouvrir
- mode : chaînes parmi (entre autres)
  - "r" : lecture
  - "w" : écriture (tronque contenu si existe, crée sinon)
  - "a" : écriture en ajout en fin de fichier
- retourne un identificateur FILE *, NULL si erreur → **tester !**

int fclose(FILE *fp) ;

- termine écritures différées (vidade des tampons)
- retourne 0 si OK, EOF si erreur (rarement testé !)

Fichiers : lecture et écriture simples

size_t f...(void *ptr, size_t s, size_t nmemb, FILE *f) ;

- fread et fwrite : lecture/écriture de tableaux de nmemb éléments
- ptr : adresse du tableau (de type const void * pour fwrite)
- s : sizeof(type_élément_de_tableau)
- retournent le nombre d'éléments lus ou écrits $\rightarrow$ **vérifier !**

int fputc(int c, FILE *f) ; **int fgetc(FILE *f) ;**

- Le caractère est codé sur un int :
  - dont l'octet de poids faible = code du caractère
  - ou **EOF** (end of file), souvent défini comme -1
- fputc : écriture d'un caractère, retourne le caractère écrit ou EOF
- fgetc : lecture d'un caractère : retourne le caractère lu ou EOF

Fichiers : autres fonctions

Fprintf et fscanf : même principe que printf et scanf

Entrées/Sorties avec tampon en mémoire :

- prélecture de caractères à l'avance dans le tampon
- écriture retardée : dépôt dans le tampon et vidage ultérieur du tampon dans le fichier.
- **int fflush(FILE *f) ;**
  - force explicitement le vidage des tampons
  - retourne 0 ou EOF si erreur
- printf : pas de vidage implicite du tampon sans \n
- **int feof(FILE *stream) ;** : retourne $\neq 0$ (vrai) si fin de fichier

int main (int argc, char*argv[ ], char *envp[ ])

- Main est une fonction retournant un code de terminaison
- 0 : terminaison correcte, $\neq 0$: nature du problème
- Peut être déclaré void $\rightarrow$ warning "main devrait être int"

mandelbrot> gcc -Wall f.c -o f

- argc : nombre de mots de la ligne de commande : 5
- argv : tableau de argc pointeurs vers les mots (chaînes) de la ligne de commande : "gcc" "-Wall" "f.c" "-o" "f"
- envp : tableau de pointeurs de chaînes : environnement "nom_de_var_d'env=texte_valeur_de_la_variable" pas de taille : dernier élément de envp est NULL

Argv[0] est le nom de la commande elle-même ("gcc")