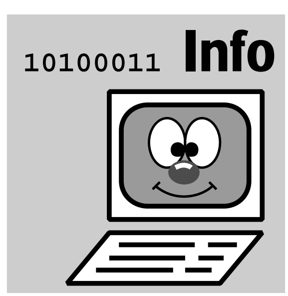


Informatique

Le Langage C

Cours de référence

Agnès Priou
IUT de Cachan
Dpt Génie Electrique et Informatique Industrielle

Avril 2012

Nom de l'étudiant :

Table des matières

Introduction **5**

1 - Premier programme en C **7**

1.1. Respectons la tradition… *Bonjour !* 7

1.2. Un premier programme sans entrées/sorties 8

2 - Un exemple de programme plus évolué **9**

3 - Variables-constantes-affectation **13**

3.1. Les variables en bref 13

3.2. A quel endroit définir une variable ? 14

3.3. Quel nom donner à une variable ? 14

3.4. Comment créer (définir/déclarer) une variable ? 14

3.5. Les différents types de variables scalaires : entiers, réels 15

3.6. Affectation d'une variable *(« reçoit »)* 17

3.7. Et si j'ai besoin d'une constante ? Valeurs entières/réelles, *#define, const* ... 18

4 - Opérateurs - Conversions **21**

4.1. Quelles sont les priorités quand on mélange des opérateurs ? 21

4.2. Les opérateurs arithmétiques : + - * / % (modulo) 22

4.3. L'opérateur d'affectation = (« *reçoit* ») 23

4.4. Les conversions de type : implicites et explicites (*cast*) 23

4.5. Les opérateurs relationnels *: inférieur, supérieur, égal, différent…* 25

4.6. Les opérateurs logiques : *ET OU NON* 25

4.7. Opérateurs de manipulation de bits – masques (ET bit à bit, décalage…) 26

4.8. L'opérateur d'adresse & 29

4.9. Les opérateurs d'incrémentation et de décrémentation ++ -- 29

4.10. L'opérateur *sizeof* (taille en octets) 30

5 - Les structures de contrôle **31**

5.1. Les répétitions *: for, while, do…while, continue, break* 32

5.2. Les exécutions conditionnelles : if…else, switch, ?: 37

6 - Les entrées/sorties conversationnelles (clavier/écran) **43**

6.1. Affichage à l'aide de la fonction *printf* 43

6.2. Lecture au clavier à l'aide de la fonction *scanf* 45

6.3. D'autres fonctions d'entrées/sorties *: getchar, putchar, GetKey…* 49

7 - Utilisation de fonctions **51**

7.1. Un peu de vocabulaire 52

7.2. Le mode d'emploi d'une fonction : le *prototype* (ou *déclaration*) 52

7.3. L'utilisation de la fonction : l'*appel de fonction* 53

7.4. Plus d'informations sur les prototypes de fonction 54

7.5. Récapitulation : prototypes et appels de fonction en une page 54

8 - La bibliothèque de fonctions mathématiques *(sinus, exp, valeur absolue…)* **.........57**

9 - Définition de fonction 59

9.1. Pourquoi créer des fonctions dans mon programme ?.................................. 59

9.2. Dessinons : boîtes noires - découpage fonctionnel du programme 59

9.3. Premier avertissement : déclaration ≠ définition de fonction........................ 60

9.4. En détails : définition, déclaration, utilisation d'une fonction 61

9.5. Quelques exemples de fonctions simples 63

9.6. Variables locales et variables globales 65

10 - La compilation séparée (multi-fichiers) 67

10.1. Fabrication du programme : compilation (préprocesseur), édition de lien..... 67

10.2. Conséquences sur l'utilisation de fonctions... 68

10.3. Conséquence sur les variables globales : la déclaration *extern* 68

10.4. Exemple de programmation multi-fichiers (sans fic en-tête) 69

10.5. Ecriture d'un fichier en-tête – Exemple complet 70

11 - Les tableaux 73

11.1. Premier exemple de programme avec un tableau.. 73

11.2. Définition d'un tableau à une dimension 74

11.3. Accès aux éléments d'un tableau : tab[i] 75

11.4. Initialisation d'un tableau, totale ou partielle 75

11.5. Copie d'un tableau : *for* ou *memmove* 76

11.6. Transmission d'un tableau en paramètre d'une fonction 76

11.7. Tableau multidimensionnel (matrice, …) 78

12 - Les chaînes de caractères 83

12.1. Définition et initialisation d'une chaîne 83

12.2. Ecriture à l'écran d'une chaîne (utilité : ***) 84

12.3. Lecture d'une chaîne au clavier.. 85

12.4. Quelques fonctions de traitement de chaînes de caractères 86

12.5. Les tableaux de chaîne de caractères 90

13 - Les pointeurs 91

13.1. Définition et affectation d'un pointeur 91

13.2. Arithmétique des pointeurs 94

13.3. Application des pointeurs au passage en paramètre..................................... 95

13.4. Application des pointeurs : allocation dynamique *malloc, free*....................... 97

13.5. Pointeurs sur une fonction 100

14 - Pointeurs et tableaux à une dimension 105

14.1. Relation nom de tableau - pointeur 105

14.2. Transmission d'un tableau en paramètre d'une fonction............................. 106

15 - Pointeurs et chaînes de caractères 107

15.1. Pointeur et constante chaîne de caractères 107

15.2. Retour sur les tableaux de caractères 107

15.3. Les tableaux de pointeurs 108

16 - Les structures **109**
16.1. Modèle de structure 109
16.2. Définition d'une variable de type structuré 110
16.3. Initialisation d'une variable structurée 111
16.4. Accès aux champs d'une variable structurée 112
16.5. La structure en tant que paramètre 113
16.6. Bilan sur les ressemblances tableau - variable structurée 115
16.7. Structures imbriquées 115
16.8. Listes chaînées : application des pointeurs 116
16.9. Allocation dynamique de structures 117
16.10. Un exemple en plusieurs fichiers avec fichier en-tête 118
17 - Les fichiers **121**
17.1. Contenu d'un fichier binaire et de son équivalent texte 121
17.2. Ouverture et fermeture d'un fichier : *fopen, fclose* 123
17.3. Entrées/sorties en accès binaire 125
17.4. Entrées/sorties formatées en mode texte 129
17.5. Lecture en accès binaire d'un fichier texte 134
17.6. Un fichier d'échange entre programmes : le fichier *csv* 135
18 - Les simplifications d'écriture **137**
18.1. Définition de nouveaux noms de types : *typedef* 137
18.2. Les types énumérés : *enum* 139
19 - Les classes d'allocation mémoire **141**
19.1. Les 3 sortes de variables : fichier, bloc, paramètre formel 141
19.2. Les variables de fichier 142
19.3. Les variables de bloc 143
19.4. Les paramètres formels 144
19.5. Initialisation d'une variable 144
19.6. Syntaxe complète d'une définition de variable 146
19.7. Variables « statiques » et « automatiques » 146
20 - Etes-vous un « bon » programmeur ? **147**
Annexe A. Table des codes ASCII **149**
Annexe B. Débogage d'un programme **150**
Liste des tableaux et des figures **151**
Index **152**

Introduction

Pourquoi le Langage C ?

Incontournable pour l'informatique industrielle

Autrefois programmables uniquement en Assembleur, les processeurs utilisés en informatique industrielle (microcontrôleurs, DSP) sont à présent tous fournis avec un compilateur C. Le Langage C est "plus proche de l'Assembleur" que d'autres langages, donc convient bien aux applications d'informatique industrielle.

Pour le meilleur et pour le pire

Le Langage C est un langage **puissant** et **permissif** : qualités ou défauts selon le niveau du programmeur, ces deux caractéristiques liées n'en font malheureusement pas l'ami des débutants, à qui un cadre plus rigide comme celui du Pascal convient mieux.

Des codes sources portables… ou presque

Le Langage C est **portable** sous certaines conditions : « portable » signifie qu'un même **fichier source** peut être compilé avec des compilateurs différents, pratiquement sans modifications. Pour être portable, un fichier source doit être écrit en C normalisé ("C ANSI" ou "ISO").

Mais cette portabilité a ses limites :

- chaque IDE (Environnement de Développement Intégré) ajoute aux instructions et fonctions C de base, appartenant au Langage C normalisé, des fonctions supplémentaires spécifiques qui sont parfois bien pratiques : primitives d'entrées-sorties, graphiques, sonores, primitives utiles en informatique industrielle... Il faut éviter l'usage de ces primitives pour que les programmes sources obtenus soient du « C normalisé pur », c'est-à-dire soient portables sous un autre compilateur et/ou sur un autre type de machine.
- le Langage C des microcontrôleurs est un dialecte plutôt qu'un langage : certaines instructions spécifiques (en particulier l'accès aux ports d'entrée/sortie) dépendent entièrement du compilateur utilisé.

La portabilité complète est difficile à atteindre. Quand on ne peut/veut pas éviter d'utiliser quelques primitives spécifiques de l'IDE, ou quand on utilise des instructions spécifique d'un microcontrôleur, la solution est alors de **les isoler dans des zones (fonctions) bien particulières du programme**. Seules ces parties de code, non portables, seront à réécrire en cas de migration vers un autre compilateur. L'exemple le plus classique est celui des entrées/sorties du programme (interface graphique, accès à des ports de microcontrôleur, etc.) : il ne faut pas les mélanger à des parties facilement portables comme les calculs et autres traitements décisionnels.

Quelques (rares) exemples de ce document contiennent des fonctions spécifiques à l'environnement de développement (fonctions non portables). Elles seront toujours signalées par un commentaire.

C ou C++ ?

Les logiciels de développement en Langage C sur PC sont souvent des logiciels de C++ (Langage C « Orienté Objet »). Mais ils conviennent parfaitement pour faire du "C pur" : le C++ contient un noyau de C et des couches supplémentaires pour programmer en "Orienté Objet" ("++"). Nous ne parlerons ici que de C.

Help !

Dans l'environnement CVI :
Pour obtenir des informations sur une fonction, placez le curseur sur le nom de la fonction (avec parenthèses, même vides) et tapez ***<CTRL> P***. Vous obtenez ainsi le "panneau de fonction", où le clic droit de souris vous sera très utile.
F1 fournit l'aide générale.

Les bonus : débogueur et règles de style

Outre la présentation du Langage C proprement dit, vous trouverez en annexe dans ce document les commandes de base du "**débogueur**". Cet outil pratique et très simple à utiliser nous aide à trouver les erreurs lors de l'exécution du programme : exécution pas à pas, visualisation des variables…(*debug*)
Les commandes de *debug*, qu'on retrouve dans tous les logiciels de développement (sous des noms parfois un peu différents), sont rapidement indispensables quand la taille du programme augmente... mais aussi quand on veut éviter de fastidieuses entrées/sorties au début des tests (lecture du clavier, affichage à l'écran).

Vous trouverez aussi dans ce document des "**règles de style**". Ce sont des conseils destinés à vous faire programmer de façon **propre et lisible**. Bien que ces règles de style ne soient pas une obligation imposée par le compilateur, il est important de les respecter pour que vos programmes soient clairs et évolutifs.

Ce document n'est donc pas un simple catalogue des instructions disponibles en Langage C. Il vous donne aussi des conseils de programmation qui permettent de bien programmer.

Ce document pour qui ?

J'enseigne dans les domaines de l'électronique et de l'informatique industrielle à l'IUT de Cachan. L'approche de ce document est donc parfois celle de l'« électronicien / informaticien industriel », qui navigue entre PC et microcontrôleurs sous des environnements variés, plutôt que celle de l'« informaticien pur », qui ne programme que sur PC ou équivalent.

Par exemple, le type entier *int* sera en général remplacé par un type dérivé plus précis comme *short* ou *long int* (car *int* n'a en général pas la même taille sur un PC et sur un microcontrôleur). Certains chapitres, comme les opérateurs de manipulation de bit très utiles en Informatique Industrielle, sont très développés.

Cette approche « Informatique Industrielle », parfois plus exigeante, est tout à fait compatible avec celle de l'informaticien pur.

Les symboles de ce document

Les remarques, les règles de style et les points dangereux seront signalés par les symboles suivants :

☞ Ceci est une remarque

☺ Ceci est une règle de style : elle n'est pas obligatoire pour le compilateur, mais il faut suivre cette règle pour écrire un beau programme, lisible et évolutif.

Ceci est un point dangereux

J'ai besoin d'un environnement de développement en C !

De nombreux environnements intégrés gratuits sont téléchargeables sur Internet, comme ***CodeBlocks*** ou *DevCPP*. On peut leur adjoindre des bibliothèques spécifiques, comme ***SDL*** pour le graphisme.
Les programmes de ce document ont été écrits sous *CVI*, un excellent IDE de *National Instruments* (payant).

1 - Premier programme en C

En général, un ouvrage d'informatique commence par un programme qui affiche « Hello World ». Ce n'est pas forcément un exemple judicieux : les entrées/sorties clavier/écran sont des notions à éviter pour le débutant, puisque l'utilisation du débogueur permet de se consacrer à des notions plus amusantes et plus portables. Sans parler du cas de l'informaticien industriel : pour lui qui ne dispose que d'un microcontrôleur entouré de quelques boutons-poussoirs et autres LED, cet exemple classique relève de l'impossible !

1.1. Respectons la tradition… *Bonjour !*

Voici un programme C très simple qui permet d'afficher à l'écran dans une « fenêtre DOS » le message :

`Bonjour monde !`

```
/* Version 1 */
#include <stdio.h>
void main(void)      /*point d'entrée du progr. */
       {
       printf("Bonjour monde !");
       }
```

```
/* Version 2 */
#include <stdio.h>
int main(void)      /*point d'entrée du progr. */
       {
       printf("Bonjour monde !");
       return 0 ;
       }
```

Certains compilateurs préfèrent la version 2 de droite et génèrent un warning sans gravité avec celle de gauche, plus simple. Tous les exemples de ce document utilisent la version 1 plus simple.

La ligne **`void main(void)`** (ou **`int main(void)`**)est une ligne d'**en-tête** qui déclare que ce qui suit constitue le **programme principal** : l'exécution de tout programme en Langage C commence toujours à partir de cette ligne. En fait, *main* est une **fonction** reconnaissable à la présence des parenthèses ( ) après son nom.

Le terme fonction apparaîtra souvent dans ce document : en Langage C, le mot fonction désigne tous les sous-programmes.

Les accolades ouvrante **{** et fermante **}** contiennent le corps de la fonction *main*, c'est-à-dire les définitions de variables et les instructions de main. On dit qu'elles délimitent un **bloc**.

Le corps de *main* ne contient pour l'instant qu'une ligne qui fait appel à un « sous-programme » d'affichage de la bibliothèque standard du Langage C : *printf* est une fonction (présence des parenthèses) qui affiche la chaîne de caractères comprise entre les guillemets " ".

A la fin de chaque ligne se trouve un **point-virgule** (**;**). Ce délimiteur termine obligatoirement en C chaque déclaration de variable et chaque instruction simple.

Un **commentaire** (ici sur la ligne d'en-tête de *main*) peut être inséré sur une ligne, seul ou derrière une instruction. Il commence par **/***, termine par ***/**, et occupe autant de place qu'on le désire. Les commentaires imbriqués sont interdits. Beaucoup d'environnements acceptent qu'un commentaire commence par **//**, il s'achève alors automatiquement à la fin de la ligne.

Le Langage *C* fait la **distinction entre minuscules et majuscules**. Toutes les instructions du *C* sont écrites en **minuscules**.

La directive **`#include`** au début du programme n'est pas une instruction exécutable. C'est un ordre destiné au préprocesseur (étape précédant la compilation), qui demande l'insertion du **fichier en-tête** `stdio.h`. Un fichier en-tête contient les informations nécessaires au compilateur pour vérifier la bonne utilisation des fonctions d'entrées/sorties comme *printf* (stdio = STandarD Input/Output). Toute utilisation d'une fonction du Langage C devra être précédée par l'insertion (par la directive **`#include`**) du fichier en-tête associé : par exemple, *math.h* pour les fonctions mathématiques, *string.h* pour les fonctions de manipulation de chaînes de caractères, *stdio.h* pour les fonctions d'entrées/sorties ...

Dans la suite de ce document, afin d'alléger l'écriture des programmes, nous n'écrirons pas les directives *#include* nécessaires au bon fonctionnement des exemples. Cet "oubli" sera corrigé par le compilateur (CVI) ou donnera lieu à un message du compilateur si celui-ci est correctement configuré (du genre "Call to function without declaration/prototype").

Utilisez l'aide en ligne pour savoir quel fichier en-tête d'extension .h est associé à une fonction.

1.2. Un premier programme sans entrées/sorties

Voici un programme en Langage C, toujours très simple, qui calcule la plus grande valeur réelle parmi deux possibles :

```
void main (void)
      {
      /* définitions des variables du programme */
      double reel1=5.7 , reel2=-12.8 ;     /* données d'entrée */
      double max ;                         /* donnée de sortie */

      /* traitement */
      if ( reel1>reel2 )  max = reel1 ;
      else                max = reel2 ;

      /* vérification du résultat au débogueur au lieu d'un affichage */
      }
```

Cette fois, notre programme manipule des **données** stockées dans des **variables**. Il s'agit ici de trois variables réelles (de type *double*), dont deux sont initialisées pour éviter une saisie au clavier (ce sont les données d'entrée du traitement). La troisième variable *max* est destinée à contenir le résultat du traitement.

Le traitement proprement dit consiste à comparer les valeurs des variables d'entrée et à recopier dans *max* la valeur de la plus grande. L'instruction de contrôle *if* permet de réaliser une exécution conditionnelle.

Ce programme ne contient pas d'« entrées/sorties conversationnelles » au clavier et à l'écran : **l'initialisation des données d'entrée permet d'éviter la saisie au clavier (toujours fastidieuse) et l'affichage des variables avec le débogueur en fin de traitement évite un affichage à l'écran**. Il est conseillé de tester chaque programme de cette façon avant d'ajouter les entrées/sorties conversationnelles.

Pour afficher les variables du programme en fin de traitement, il faut mettre un point d'arrêt à la fin de *main* (sur l'accolade fermante par exemple) et demander l'affichage des variables. La mise en œuvre exacte du débogueur dépend de votre IDE, mais elle est toujours simple.

2 - Un exemple de programme plus évolué

La programmation moderne est une programmation **MODULAIRE**, c'est-à-dire composée de modules (sous-programmes) appelés **fonctions** par le Langage C.
Nous avons vu qu'il existe toujours au moins une fonction principale appelée *main*. En pratique, un "bon" programme (c'est-à-dire suffisamment modulaire) comporte de nombreuses fonctions qui dépassent rarement une dizaine de lignes. C'est pourquoi ce chapitre se propose de vous présenter un exemple de programme modulaire avec plusieurs fonctions.

En fait, cet exemple a des objectifs multiples, puisqu'il est destiné à :

- vous présenter la modularité, avec le découpage du programme en fonctions ;
- vous habituer à la forme générale d'un programme en C ;
- vous permettre de situer l'emplacement des instructions détaillées dans les chapitres suivants.

Cet exemple est une version simple du calcul des racines réelles d'un polynôme du second degré ; les coefficients de l'équation sont réels et imposés.

☞ Ne vous inquiétez pas si vous ne comprenez pas tout, cela viendra en temps utile...

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

/***************************************************************************
Résolution simplifiée d'une équation du second degré   6/03/08
L'équation est imposée : Ax^2+Bx+C=0
***************************************************************************/

/* inclusion des fichiers en-tête nécessaires */
#include <math.h>          /* contient la déclaration de sqrt */
#include <stdio.h>

/* Définition des constantes symboliques */
#define EPS 1e-10          /* constante "très petite" pour la comparaison de delta avec 0. */

/* Déclaration des fonctions utilisées (« prototypes » = mode d'emploi) : */
double calculer_delta( double a, double b, double c ) ;
int    calculer_nb_solutions( double delta ) ;
void   calculer_afficher_solutions( double a, double b, double c, int nb_sol,
                                    double delta ) ;
void   afficher_coeff( double a, double b, double c ) ;
double saisir_reel( char * message );
```

```
/*------------------------------------------------------------------------
  Programme principal : fonction main
------------------------------------------------------------------------*/
void main(void)
   {
  /* définition des variables locales de main */
   double a, b, c ;            /* coefficients réels de l'équation */
   double delta ;
   int nb_sol_reelles ;

  /* Première partie du programme, réalisée et testée avant de poursuivre : */
  /* l' équation est imposée par l'initialisation des données de test (sans saisie au clavier) : */
   a=-1.0 ;  b=-0.5 ; c=-5.0 ;      /* on modifie ensuite ces valeurs pour effectuer d'autres tests */
   afficher_coeff ( a, b, c ) ;
   delta = calculer_delta( a, b, c ) ;
   nb_sol_reelles = calculer_nb_solutions( delta ) ;
   calculer_afficher_solutions ( a, b, c, nb_sol_reelles, delta );

  /* Suite du programme (après test  complet de la première partie) : */
  /* l' équation est choisie par l'utilisateur  avec une saisie au clavier : */
   a = saisir_reel("\n\t\t Nouvelle equation :\n\nTapez a :");
   b = saisir_reel("Tapez b :");
   c = saisir_reel("Tapez c :");
   delta = calculer_delta( a, b, c ) ;
   nb_sol_reelles = calculer_nb_solutions( delta ) ;
   calculer_afficher_solutions ( a,b, c, nb_sol_reelles, delta );
   }

/*------------------------------------------------------------------------
  Fonction calculer_delta
------------------------------------------------------------------------*/
double calculer_delta( double a, double b, double c )
   {
   return b*b - 4.*a*c ;
   }
/*------------------------------------------------------------------------
  Fonction calculer_nb_solutions
------------------------------------------------------------------------*/
int calculer_nb_solutions( double d )
   {
   int nb_sol ;
   if ( d > EPS )           nb_sol = 2 ;         /* deux solutions reelles */
   else if ( d < -EPS )     nb_sol = 0 ;         /* pas de solutions reelles */
   else                     nb_sol = 1 ;         /* une solution reelle double */
   return nb_sol ;
   }
/*------------------------------------------------------------------------
  Fonction afficher_coeff
------------------------------------------------------------------------*/
void afficher_coeff( double a, double b, double c )
   {
   printf("\t\t  Resolution simplifiee d'une equation du second degre :\n");
   printf("\t\t Coefficients : a = %lf - b = %lf - c = %lf \n\n", a, b, c );
   }
```

```
/*-----------------------------------------------------------------------
 Fonction calculer_afficher_solutions
------------------------------------------------------------------------*/
void calculer_afficher_solutions( double a, double b, double c, int nb_sol,
double delta )
   {
   double rac, x1, x2 ;             /*  variables locales de la fonction */

   switch (nb_sol)            /* instruction qui permet le choix multiple */
         {
         case 0 :
                    printf(" \n Pas de solutions reelles.\n") ;
                    break ;
         case 1 :
                    printf("\n Une solution double: %lf\n", -b/(2.*a) ) ;
                    break ;
         case 2 :
                    rac = sqrt(delta) ;
                    x1 = (-b-rac)/(2.*a);
                    x2 = (-b-rac)/(2.*a);
                    printf("\n Solution 1 : %lf", x1 ) ;
                    printf("\n Solution 2 : %lf\n", x2 ) ;
                    break ;
         }
   }
/*-----------------------------------------------------------------------
 Fonction saisir_reel
------------------------------------------------------------------------*/
double saisir_reel( char * message )
    {
        double reel = 1.0 ;

        printf( "%s ", message );
        scanf("%lf", &reel );
        return reel;
    }
```

Outre la manipulation de **fonctions**, ce programme simple fait apparaître la nécessité de créer des données (**variables** et **constantes)** qu'on manipule à l'aide d'**opérateurs** et d'**instructions de contrôle**.
Des blocs d'instructions sont isolés sous forme de **fonctions** (= sous-programme en Langage C) afin d’obtenir une programmation modulaire.
Toutes ces notions seront développées dans les chapitres qui suivent.

Ce programme peut bien sûr être amélioré : séparation complète du calcul et des affichage, gestion complète du cas *delta*<0, test des coefficients pour détecter les cas particuliers tels que a=0, etc... Mais un programme "complet" serait trop "effrayant" dans le cadre de cet exemple d'introduction !

Notez bien la structure générale de ce programme :

- les insertions de fichiers en-tête (#include) et les constantes symboliques (#define) figurent en début de fichier : ce sont des **directives** destinées au **préprocesseur**, repérables grâce au caractère **#** qui les commence (et par l'absence de point-virgule : ce ne sont pas des instructions). Pour plus d'information sur le préprocesseur et sur les étapes du développement d'un programme en C, on peut se reporter au début du chapitre "La compilation séparée".

- derrière les directives apparaissent les **prototypes** (modes d'emploi) des fonctions utilisées qui sont définies par le programmeur. Les prototypes des fonctions fournies par le Langage C, quant à elles, sont contenues dans les fichiers en-tête insérés par *#include*.

  Les prototypes de fonction permettent au compilateur de vérifier la bonne utilisation des fonctions (nombre et type des paramètres), ainsi que d'effectuer les éventuelles conversions nécessaires. Les prototypes sont indispensables et leur absence provoque normalement une erreur de compilation ("F*unction should have a prototype*" ou équivalent) ou une proposition d'insertion faite par le compilateur (environnement *CVI*).
- Ensuite seulement viennent les **instructions exécutables**, situées à l'intérieur des fonctions. La fonction *main* a été placée en première position parce que c'est sa place logique. Mais le compilateur ne l'exige pas : si tous les prototypes de fonction sont présents en début de fichier, l'ordre d'apparition des **définitions** de fonction est quelconque.

Ce court programme est destiné à vous familiariser avec la structure générale d'un fichier source C. Les chapitres qui suivent vont expliquer en détail :

- comment **définir des variables** simples ou composées
- comment **effectuer des actions** sur ces variables (opérateurs)
- comment regrouper des blocs d'instructions sous forme de **fonctions**
- etc ...

3 - Variables-constantes-affectation

Toutes les **informations utiles** au programme (« **données** ») sont mémorisées dans des **VARIABLES**, c'est-à-dire des **emplacements mémoire** accessibles en lecture et en écriture. L'emplacement mémoire d'une variable est créé (alloué) lors de la **définition** (parfois appelée **déclaration)** de la variable.

Dans le programme C donné au chapitre précédent, chaque fonction définit ses propres variables au début de son code interne : par exemple, *a*, *b*, *c* et *delta* pour la fonction *main*, *rac* pour la fonction *calculer_afficher_solutions*, etc.

3.1. Les variables en bref

Où définir une variable ?

En général, au début d'une fonction, juste derrière les accolades ouvrantes (voir exemples des chapitres précédents). La variable est alors dite **variable locale** de la fonction.

Quelles sont les caractéristiques d'une variable locale ?

C'est une variable qui appartient en propre à une fonction. Elle n'est **utilisable que par cette fonction** : on dit que sa visibilité, c'est-à-dire son domaine d'utilisation, est limitée à la fonction.

Par défaut (variable locale "automatique", définie en pile), elle est **créée au début de chaque exécution de la fonction et elle est détruite à la fin de l'exécution de la fonction** : on dit que sa durée de vie est celle de la fonction.

Peut-on créer deux variables locales ayant le même nom ?

Oui, pourvu qu'elles appartiennent à deux fonctions différentes. Chaque fonction ne peut accéder qu'à ses variables locales, il n'y a donc aucune confusion possible.

> *De même, deux personnes qui portent un prénom identique, mais pas le même nom, sont différentes. Le nom de la variable locale joue le rôle du prénom, alors que le nom de la fonction joue le rôle du nom de famille.*

Quels sont les types possibles pour une variable ?

Une variable peut être de type simple, c'est-à-dire principalement entière ou réelle : elle occupe alors "une case mémoire". Elle peut aussi être de type composé (tableau ou structure) : elle occupe alors une "suite de cases mémoire" consécutives.
Les variables composées, indispensables en programmation, seront vues dans les chapitres sur les tableaux et les structures.

Qu'est-ce qu'une variable locale "static" ?

C'est une variable locale qui n'est pas détruite à la fin de l'exécution de la fonction : elle **conserve sa valeur** d'un appel à l'autre de la fonction. Sa durée de vie devient celle du programme et elle n'est plus créée en pile.
Mais sa visibilité reste limitée à la fonction (la variable reste « locale »).

3.2. A quel endroit définir une variable ?

L'emplacement dans le fichier source de la déclaration de la variable est très important : il indique dans quelle partie du programme la variable peut être utilisée.
Dans le cas le plus fréquent, la définition de variable apparaît **au début du corps de la fonction qui en a besoin**, juste après l'accolade ouvrante (voir la définition de *delta* au début de *main*, chapitre précédent). La variable est alors dite **locale à la fonction**, elle n'est connue et utilisable qu'à l'intérieur de cette fonction.

3.3. Quel nom donner à une variable ?

Les identificateurs sont des noms qui permettent de désigner un objet manipulé dans le programme : variables et fonctions principalement.
Un identificateur est formé d'une **suite de lettres** (a, ..., z, A, ..., Z), de **chiffres** (0, ..., 9) et du signe de **soulignement** (_). Il commence obligatoirement par une lettre ou par _. Il ne peut comporter aucun espace, car celui-ci est un séparateur en C. Les 32 premiers caractères sont seuls significatifs.

En Langage C, les minuscules et les majuscules ne sont pas équivalentes et constituent des caractères différents. Par ailleurs, les caractères accentués sont interdits.

Exemples d'identificateurs (tous différents) : moy_geom, MoyArith, resultat_1, data_4octets, somme, Somme, Prix_Article, masse_proton, …

☺ Ne pas donner à une variable un nom entièrement écrit en majuscules : **les noms en majuscules sont réservés aux constantes.**

☺ **Choisir des noms "expressifs", donc suffisamment longs**. Il faut utiliser le caractère souligné ou les majuscules pour les noms composés (exemple : moy_geom ou MoyGeom).

3.4. Comment créer (définir/déclarer) une variable ?

Une variable est caractérisée par :

- son **identificateur**, c'est-à-dire son nom ;
- son **type**, qui indique l'ensemble des valeurs qui peuvent être attribuées à la variable et les opérations possibles sur la variable ;
- sa **valeur initiale**, qui peut éventuellement être indéfinie ;
- sa classe d'allocation mémoire, qui ne sera abordée qu'au chapitre correspondant. Sauf indication contraire, les variables seront toutes **locales** (règle de style). Une variable locale peut être automatique (cas par défaut) ou statique (mot-clé *static*).

Toute variable doit être préalablement définie avant d'être utilisée pour la première fois. La **définition** (ou **déclaration**) d'une variable permet de r**éserver un emplacement mémoire** pour la variable ; cet espace mémoire est repéré par l'identificateur de la variable et sa taille dépend du type de la variable.
La déclaration d'une variable précise son nom et son type, parfois sa valeur initiale. Le compilateur lui réservera le nombre d'octets nécessaires en mémoire : c'est « l' allocation mémoire ».

La syntaxe d'une définition (ou déclaration) de variable est : **type nom_var ;**

Exemple 1. Définitions de variables :

```
short int nb_mesures ;          /* entier (court) */
double pression ;               /* réel */
unsigned char data_8bits ;      /* entier sur un octet */
/* On peut définir plusieurs variables de même type dans une seule définition :  */
long int longueur_mm, largeur_mm, hauteur_mm ;
```

L'**initialisation** de la variable, c'est-à-dire l'attribution d'une valeur dès la réservation de l'emplacement mémoire, peut avoir lieu en même temps que sa définition :

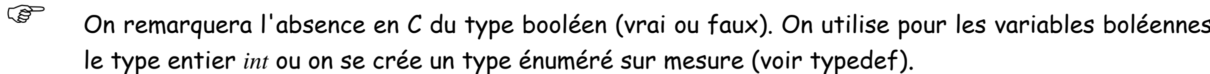
Exemple 2.   Définition et initialisation de variables :

```
short int nb_iteration=12, note_max=100 ;           /* variables entières (sur 16 bits) */
double charge_electrique = -1.6e-19 ;                /* variable réelle */
char Lettre='A', Chiffre='9', data_8bits=0xFF ;     /* variables entières (un octet) */
```

3.5. Les différents types de variables scalaires : entiers, réels

En Langage C existent trois familles de types de base :

- les types **entiers** (bâtis autour du mot-clé ***int***) permettent de représenter les nombres entiers ;
- les types **réels** (mot-clé *float* ou ***double***) permettent de représenter les nombres réels (parfois appelés « flottants » en informatique) ;
- le type « **octet** » (mot-clé ***char***) permet de représenter les variables occupant un seul octet, entre particulier les caractères ; il s'agit en réalité d'un type entier.


☞ On remarquera l'absence en C du type booléen (vrai ou faux). On utilise pour les variables boléennes le type entier *int* ou on se crée un type énuméré sur mesure (voir typedef).

Par défaut, tous les types sont considérés comme **signés** (*signed*), sauf le type *char*. Mais on peut rajouter le mot-clé *unsigned* devant le type de la variable si nécessaire.

3.5.1 Les variables de type entier

Définition	Valeurs possibles	Place occupée
int	dépend du logiciel et du processeur. Très utilisé par les informaticiens « purs », il est déconseillé en informatique industrielle.	2 ou 4 octets
short int ou ***short***	-32768 à +32767	2 octets
unsigned short int ou ***unsigned short***	0 à 65535	2 octets
long int ou ***long***	-2147483648 à 2147483647	4 octets
unsigned long int ou ***unsigned long***	0 à 4294967295	4 octets
char Très utile en II[1] : ***unsigned char***	0 à 255 ou -128 à +127 (selon le compilateur)	1 octet

Les définitions de type entier (les types en gris sont les plus utilisés)

Les variables entières sont définies à partir du mot-clé ***int*** : ***short int*** (en abrégé ***short***), ***long int*** (en abrégé ***long***), ***unsigned short int*** (en abrégé ***unsigned short***), etc.

En effet, on peut ajouter à la déclaration *int* des « attributs » qui agissent sur la taille de l'emplacement mémoire (mots-clés ***short*** ou ***long***) ou sur le mode de représentation, signé ou non signé (mots-clés ***unsigned*** ou par défaut ***signed***). L'attribut *unsigned* sera le plus souvent réservé à *char* : *unsigned char* est très utilisé en informatique industrielle, car les ports d'entrées/sorties sont souvent sur 8 bits.
Ces attributs sont indispensables avec *int* si on veut rendre le programme portable. En effet, le type *int* est le seul type non portable : le nombre d'octets occupés par une variable *int* dépend du logiciel et du processeur. L'intervalle des valeurs que peut prendre une variable *int* fluctue donc selon la cible et le logiciel : sa valeur absolue sera inférieure à 32 768 pour un *int* sur 2 octets, et inférieure à environ 2 milliards sur 4 octets...
Par facilité d'écriture, on emploie souvent le type *int* sans préciser *short* ou *long* : on utilise ainsi la taille par défaut (*short* ou *long*) du logiciel. Mais le programme n'est plus portable ! Si on veut alléger les écritures, mieux vaut supprimer le mot-clé *int* qui est optionnel derrière *short* ou *long*.

☞ Le type *int* (sans attribut) peut être utilisé pour des variables particulières, comme les indices de tableau, les codes d'erreur renvoyés par certaines fonctions et les booléens.

[1] I.I.= abréviation pour Informatique Industrielle, souvent synonyme de programmation sur microcontrôleur (µC).

Exemple 3. Déclaration et initialisation de variables entières :

```
short int somme ;                     /* le mot-clé int est optionnel derrière short ou long */
short int largeur = 10, hauteur = 20 ;
long int nb_de_Francais = 55000000 ;        /* L est optionnel */
unsigned short int Masque =0xFF00 ;
```

Les valeurs numériques sont exprimées :

- en **base 10** (par défaut) : par exemple 12 ou -5740 pour des entiers courts (*short int*), 23L ou -154678L (le L est optionnel) pour un entier long (*long int*).
- en notation **hexadécimale** (base 16) : elles commencent alors par **0x** ou **0X**. Exemples : 0x002A, 0x6B ou 0XFFFF.

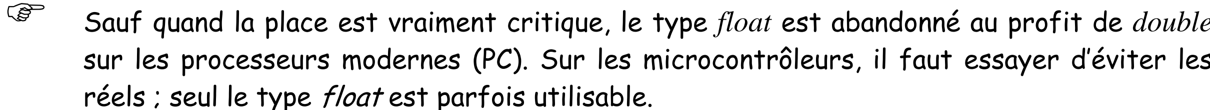
3.5.2 Les variables de type réel

Définition	Gamme de valeurs	Précision	Place occupée	Exemples de constantes	commentaires
double	$+2{,}23.10^{-308}$ à $+1{,}79.1^{308}$ et 0 $-2{,}23.10^{-308}$ à $-1{,}79.10^{308}$	$\mathbf{10^{-15}}$	8 octets	**-1.6e-19** **3.14159**	à choisir sur un PC
float (obsolète sur PC)	$+1{,}21.10^{-38}$ à $+3{,}4.10^{38}$ et 0 $-1{,}21.10^{-38}$ à $-3{,}4.10^{38}$	$\mathbf{10^{-6}}$	4 octets	**-1.6e-19f** **3.14159f**	**à éviter** sur un PC sauf si la place est comptée. Certains µC le tolèrent (mais coûteux !).

Les définitions de réels (le type en gris est le plus utilisé)

La variable réelle **double précision** (à utiliser par défaut) est définie à l'aide du mot-clé ***double***. Elle occupe 8 octets en mémoire. La taille de la mantisse (52 bits) garantit une précision relative de $\mathbf{10^{-15}}$ dans les calculs : c'est le principal critère de choix d'un type réel.

La variable réelle **simple précision** est définie à l'aide du mot-clé ***float***. Elle occupe 4 octets en mémoire. La taille de la mantisse (23 bits) garantit une précision relative de 10^{-6} (ce n'est pas beaucoup !).



☞ Sauf quand la place est vraiment critique, le type *float* est abandonné au profit de *double* sur les processeurs modernes (PC). Sur les microcontrôleurs, il faut essayer d'éviter les réels ; seul le type *float* est parfois utilisable.

La valeur d'un nombre réel peut s'écrire de deux façons en Langage C :

- sous forme **décimale** [-]ddd.dddddd : par exemple 23.145, 1.0 ou -345.09
- sous forme **scientifique** (qui s'écrit en mathématiques $M.10^x$ avec exposant x et mantisse M). En C, cette notation exponentielle s'écrit [-]d.dddddd**E**[-]ddd (la lettre e ou E sert à introduire l'exposant entier) : par exemple 4.19E-3 (qui signifie $4{,}19.10^{-3}$) ou -3e20 (qui signifie -3.10^{20}) ;

☞ Par défaut, les valeurs numériques sont de type *double*. Pour indiquer une valeur *float*, on peut les terminer par la lettre f (exemple 1.56f ou -5.7e4f). Mais le compilateur sait effectuer les conversions implicites qui conviennent si le f est absent.

Exemple 4. Déclaration et initialisation de variables réelles :

```
double temp1=0.0, temp2=-100.0 ;       /* le 0 derrière le point est mis pour la visibilité */
double masse_electron = 9.1E-31 ;      /* valeur en  notation scientifique */
float longueur=10.57f ;               /* la lettre optionnelle f indique une constante de type float */
```

3.5.3 Les variables de type caractère

Le mot-clé ***char*** permet de déclarer une variable qui occupe **un seul octet** en mémoire. **Il est très utilisé en informatique industrielle sous sa forme *unsigned char***.

Dans d'autres langages, ce type s'appelle BYTE (octet), nom moins réducteur que *char* qui laisse faussement penser que ce type n'est utilisé que pour stocker un caractère.

Quand une variable *char* est utilisée pour stocker un caractère, elle peut prendre toutes les valeurs du code ASCII (soit 256 valeurs de 0 à 0xFF). La façon la plus simple de lui donner une valeur consiste à placer **entre apostrophes** (' ') le caractère voulu, ou si c'est impossible, son code ASCII hexadécimal.

Exemple 5. Déclaration et initialisation de variables sur un octet :

```
char Voyelle='a', Consonne='B', Ponctuation='?' ;
unsigned char masque = 0xF3 ;      /* ici, ce n'est sans doute pas un caractère */
unsigned char port = 134 ;
char LigneSuivante = 0x0A ;      /* code ASCII hexadécimal du saut de ligne; identique à '\n' */
```

Le type char est en fait un type **entier codé sur un octet** : en plus des caractères proprement dits, il peut servir à représenter toute variable qui occupe un octet en mémoire. C'est le cas de la variable *masque* de l'exemple précédent, initialisée avec la valeur hexadécimale 0xF3 (ou la valeur décimale équivalente 243).

En informatique industrielle, on utilise beaucoup le type *unsigned char*, car les ports et les registres des microcontrôleurs sont souvent sur 8 bits.

La table des codes ASCII des caractères se trouve en annexe en cas de besoin. Les représentations usuelles ('\n', '\r' ...) de certains caractères seront données au chapitre « Entrées/sorties conversationnelles ».

Suivant les compilateurs, le type *char* est par défaut non signé (valeurs entre 0 et 255) ou signé (valeurs entre -128 et +127). Il faut préciser *signed* ou *unsigned* si on a besoin de lever l'ambiguïté, c'est-à-dire si on sort de la gamme commune [0, 127].

3.6. Affectation d'une variable *(« reçoit »)*

L' affectation d'une variable consiste à lui attribuer une valeur au cours de l'exécution du programme à l'aide de l'**opérateur d'affectation =**. Elle suppose que la définition de la variable (la réservation de son emplacement en mémoire) a déjà été effectuée.
Rappel : l'initialisation d'une variable consiste à lui donner une valeur **au moment de sa définition**.

Exemple 6. Définition et affectation de variables :

```
void main(void)
     {
     short int nb_points ;      /* définitions */
     double Resultat ;
     long int nombre = 3 ;      /* définition avec initialisation  */
     …
     nb_points = 5 ;            /* affectations */
     Resultat = -5.67e-8 ;
     Resultat = 6.347 ;
     }
```

3.7. Et si j'ai besoin d'une constante ? Valeurs entières/réelles, *#define, const*

Définir une constante, c'est associer un nom symbolique **bien choisi,** écrit en **majuscules**, à une valeur numérique pas forcément explicite et qui peut être utilisée en plusieurs endroits du code source. Les constantes sont destinées à :

- augmenter la **lisibilité** : un nom bien choisi est plus expressif qu'une valeur numérique ;
- rendre le programme **plus évolutif**. On pourra facilement changer la valeur de la constante lors d'une prochaine compilation, sans avoir à parcourir tout le code source.

☺ Un bon programmeur définit beaucoup de constantes. A chaque fois que vous utilisez une valeur numérique, posez-vous la question : « Si je créais une constante pour cette valeur ? »

En Langage C, il existe deux façons de définir des constantes : les constantes symboliques (qui sont remplacées par la valeur numérique associée juste avant la compilation) et les « variables constantes » (qui sont des variables protégées en écriture).

3.7.1 Comment écrire la valeur associée à une constante ?

Les valeurs constantes peuvent être de tous les types précédemment définis :

- constantes **entières** : par défaut, elles sont de type *short int*

notation décimale :	321 -34 0	(constante *short int*)	
	-6L 100000L	(constante *long int*)	*le* L *est optionnel*
notation hexadécimale :	0xFFFF 0X3A 0x56 0xD36E		
code d'un caractère :	'e' 'Y' '7' '!' '\n' '\x007'		

- constantes **réelles** : par défaut, elles sont de type *double*

notation décimale :	1. -10. -34.894	(constante *double*)	
	655.3f -56.f	(constante *float*)	*le* f *est optionnel*
notation scientifique :	**-9.99E+23 4.456e2 -7E-4**	(constante *double*),	
	1.87E-12f 0.3e1f	(constante *float*)	*le* f *est optionnel*

3.7.2 Définition d'une constante symbolique

Une constante symbolique est définie par la directive **#define**. Elle peut être placée :

- **en début du fichier source** (comme dans l'exemple du chapitre "Programme évolué en C") ;
- **dans un fichier en-tête *.h** (par exemple la constante PI est définie dans *math.h* dans certains IDE).

☺ Il faut toujours écrire les constantes symboliques en **MAJUSCULES** : NB_ELEMENTS, PI, INDICE_MAX, CHARGE_ELECTRON, MIN_MESURES, … Inversement, aucune variable ne doit être écrite en majuscules seulement.

La directive *#define* substitue littéralement un texte à un autre avant la compilation (le compilateur ne connaîtra donc pas le symbole). Il n'y a pas de place allouée en mémoire pour une constante symbolique.
Ainsi, la directive :

```
#define PI 3.14159
```

demande au préprocesseur de remplacer littéralement le symbole PI par le texte 3.14159 (de type *double* par défaut), chaque fois que ce symbole apparaît dans le fichier source.

Une des utilisations les plus courantes est l'indication du **nombre d'éléments d'un tableau**. En effet, seule la constante symbolique convient à cet usage.

Les détracteurs des constantes symboliques lui reprochent de ne pas avoir de type explicite : il convient donc de bien choisir la valeur numérique associée à la constante.

Exemple 7. Définition d'une constante symbolique avec la directive #define :

```
#define MAX 100                          /* constante entière (short int par défaut) */
#define CHARGE_ELECTRON -1.6e-19         /* constante réelle  (double par défaut )*/
        /* Notez l'absence de point-virgule ! La définition d'une constante n'est pas une instruction,
           mais une directive (#...) donnée au préprocesseur . */

void main(void)
  {
  double charge = 5. ;

  printf("Entrez un entier compris entre %ld et %ld :", MAX/2, MAX) ;
  res = res * CHARGE_ELECTRON ;
  }
```

3.7.3 Définition d'une constante par le mot réservé *const*

On ne peut pas utiliser une constante définie par *const* comme dimension d'un tableau.

Le mot réservé ***const*** permet de définir une constante qui est en réalité une variable protégée par le compilateur. Contrairement à une constante symbolique, il est donc obligatoire de préciser la valeur de la constante au moment de sa définition : c'est un avantage par rapport aux constantes symboliques.

La constante PI peut ainsi être définie par :

const float PI = 3.14159f

Le compilateur vérifie que la variable n'est pas modifiée dans le code, ce qui est particulièrement intéressant en cas d'appel de fonction avec passage de paramètre par adresse.

Contrairement au cas des constantes symboliques, une place en mémoire est allouée par le compilateur pour la constante définie par *const*.

Certains prototypes de fonctions standards du Langage C utilisent des constantes symboliques pour protéger le paramètre d'appel. Ainsi, le prototype de la fonction *strcpy* est le suivant : `char* strcpy( char* destination, const char* source ) ;`

4 - Opérateurs - Conversions

Les opérateurs permettent de **fabriquer des expressions** et d'agir sur le contenu de variables. Ils effectuent un calcul numérique (opérateurs mathématiques), fournissent une condition logique pour un test (opérateurs logiques et relationnels), agissent sur certains bits d'une variable (opérateurs de manipulation de bits) ou réalisent des opérations spécifiques (affectation, conversions, obtention d'une adresse, etc).

4.1. Quelles sont les priorités quand on mélange des opérateurs ?

Lors d'un mélange (fréquent) entre opérateurs, deux règles s'appliquent principalement :

- L'évaluation d'une expression « simple » situé à gauche d'une affectation = se fait **de la gauche vers la droite** quand les priorités des opérateurs concernés sont les mêmes.
- Des **règles de priorité** s'appliquent entre opérateurs (voir tableau des priorités). La plupart sont évidentes (car inspirées des mathématiques ou du bon sens), d'autres moins : l'utilisation de parenthèses est une solution prudente en cas de doute.

☞ *Certains informaticiens qui connaissent par cœur le tableau des priorités d'opérateurs ci-dessous n'utilisent les parenthèses que si elles sont indispensables. Personnellement, je trouve que les parenthèses, utilisées à dose raisonnable, ajoutent à la lisibilité en cas d'expressions un peu compliquées… Dans le doute : mettre des parenthèses !*

Voici la priorité des opérateurs en commençant par les plus prioritaires :

<table>
<tr><th>Opérateurs</th><th>Rôle</th><th>Exemples</th></tr>
<tr><td>appel de fonction
accès aux éléments d'un tableau ou d'une structure</td><td></td><td>fct()
tab[i] toto.champ</td></tr>
<tr><td>! ~ ++ -- * &(adr.) (cast) sizeof -(unaire)</td><td>divers op. unaires</td><td>&toto i++ -a (double) *ptr</td></tr>
<tr><td>* / %</td><td rowspan="2">op. arithmétiques (binaires)</td><td>a%b</td></tr>
<tr><td>+ -</td><td>a+b</td></tr>
<tr><td><< >></td><td>op. bit de décalage</td><td>1<<5</td></tr>
<tr><td>< <= > >=</td><td rowspan="2">op. relationnels</td><td>a<b</td></tr>
<tr><td>== !=</td><td>a != b</td></tr>
<tr><td>&</td><td rowspan="3">op. bit (binaires)</td><td>a & 0xF0</td></tr>
<tr><td>^</td><td>a ^ 0xF0</td></tr>
<tr><td>|</td><td>a | 0xF0</td></tr>
<tr><td>&&</td><td rowspan="2">op. logiques (binaires)</td><td>cond1 && cond2</td></tr>
<tr><td>||</td><td>cond1 || cond2</td></tr>
<tr><td>? :</td><td>affectation condit.</td><td>res= a>0 ? 1 : 0</td></tr>
<tr><td>= += -+ *= /= %= &= |= <<= >>=</td><td>affectation</td><td>res = 67</td></tr>
</table>

Priorité des opérateurs dans l'ordre décroissant

☞ Les opérateurs « unaires » n'ont qu'un seul opérande : & (op. d'adresse), - (« opposé de »), opérateurs ! et ~ (NON), ++ (incrémentation), etc.

4.2. Les opérateurs arithmétiques : + - * / % (modulo)

Ils s'appliquent à tous les types numériques (*int*, *double*...), à l'exception de l'opérateur **modulo** qui ne concerne que les entiers.

Un opérateur ne fournit pas le même résultat s'il est appliqué à des entiers ou à des réels ! C'est en particulier le piège des divisions entière/réelle avec l'opérateur *quotient*.

Opérateur	Rôle	Exemples
+	addition	2+3 vaut 5
-	soustraction	2-3 vaut -1
*	produit	2*3 vaut 6
/	quotient **(entier ou réel !)**	11./2. vaut 5.5 (division réelle) mais 7/3 vaut 2 (division entière) ! ☠
%	**modulo** (reste de la division entière)	11%3 vaut 2 24%8 vaut 0 *Très utile en informatique !*

Les opérateurs mathématiques

A ces opérateurs binaires, il convient d'ajouter les deux opérateurs unaires (un seul opérande) qui sont l'opposé - et l'identité +.

La division entière fournit **deux** résultats : le quotient (opérateur /) et le reste (opérateur modulo, % en Langage C). Tous deux sont très utilisés en informatique…

Attention : les opérateurs binaires, c'est-à-dire agissant sur deux opérandes, ne sont a priori définis que pour des opérandes de même type et ils fournissent un résultat de ce type.
Par exemple, 5./2. est le quotient de deux valeurs de type *double* et l'opérateur quotient / fournit le résultat 2.5 de type *double*. Par contre, 5/2 est le quotient de deux entiers et le résultat est l'entier 2 !

Cet exemple a priori évident peut avoir des effets surprenants : une mise à l'échelle par une simple règle de trois (formule du style n/NMAX*100) fournit un résultat presque toujours nul si elle est effectuée sur des opérandes entiers sans précautions ! La solution est un *cast* (conversion explicite).

Quand les deux opérandes ne sont pas du même type, une opération de conversion implicite est mise en oeuvre par le compilateur afin que le calcul soit fait dans le **type dominant**. La hiérarchie des types est :

char < short int < long int < float < double

Quand plusieurs opérateurs apparaissent dans une même expression, les règles traditionnelles de **priorité** de l'algèbre s'appliquent (voir tableau des priorités) : d'abord les opérateurs unaires + et -, puis les opérateurs *, /, et %, puis enfin les opérateurs binaires + et -. Des parenthèses permettent de s'affranchir des priorités.

Exemple 8. Opérateurs mathématiques

Expression	Commentaire
res = 5+9/4	9/4 qui vaut 2 est ajouté à 5 → res vaut 7 au final.
res = (5+9)/4	5+9 qui vaut 14 est divisé par 4 (division entière) → res vaut 3 au final.
res = (5+9.)/4	5+9. qui vaut 14. (conversion implicite en *double* de 5 et résultat *double*) est divisé par 4 (division réelle) → res vaut 3.5 au final.
res = 4*2+9%4	4*2 qui vaut 8 est ajouté à 9%4 qui vaut 1 → res vaut 9 au final.
i = (i+1)%10	permet d'incrémenter i « modulo 10 » : i prend successivement les valeurs 0, 1, 2,…8, 9, 0, 1... Très utile pour effectuer automatiquement la remise à zéro de i quand il arrive à sa valeur maximale.

4.3. L'opérateur d'affectation = (« *reçoit* »)

L'affectation permet de « ranger » une valeur dans un emplacement mémoire (en général une variable) appelée *Leftvalue*.

Mais l'affectation n'est pas seulement un traitement à effectuer : c'est aussi une **expression** qui prend comme valeur la valeur affectée. Cela permet d'utiliser l'affectation comme opérande d'une autre expression. En voici deux exemples :

```
min = max = 0 ;
if ( (val=getchar()) =='q' ) ...        /* ici les parenthèses sont indispensables */
```

Pour comprendre ce qui se passe, commençons par deux cas simples.

L'instruction **nombre=4** est une expression qui :

- place la valeur 4 dans la variable *nombre* : c'est l'affectation de *nombre* ;
- prend elle-même la valeur 4.

De même, l'écriture **max=5*2+3** est une expression qui :

- évalue l'expression à droite (13) et donne à la variable *max* la valeur 13 ;
- prend elle-même la valeur 13 ;
- L'expression max=5*2+3 peut alors se placer à droite d'une autre affectation. Par exemple, on peut écrire : min=max=5*2+3 → les variables *min* et *max* auront toutes deux la valeur 13.

Remarquons que pour une fois, l'associativité se fait **de la droite vers la gauche**. Ainsi lorsqu'on écrit a=b=c=0 , l'expression la plus à droite est évaluée (elle vaut 0), puis sa valeur est affectée à la variable c ; en même temps, l'expression (c=0) prend la valeur 0, qui est ensuite affectée à la variable b ; l'expression (b=c=0) prend alors la valeur 0, etc ...

☞ A gauche de l'opérateur d'affectation = ne peut se trouver qu'une variable ("***LeftValue***").

On peut ainsi comprendre l'exemple **if((val=getchar())=='q')** ... L'expression val=getchar() prend comme valeur la valeur du caractère saisi au clavier par la fonction *getchar* (valeur aussi rangée dans *val*) et peut donc être comparée au caractère 'q'.

4.4. Les conversions de type : implicites et explicites (*cast*)

Le Langage C est très (trop) tolérant en ce qui concerne les mélanges de types dans une expression. C'est au programmeur de vérifier que les **conversions implicites** réalisées par le compilateur ont le sens désiré... Cet allègre mélange des types est un des plus gros pièges pour le débutant, à qui un langage fortement typé convient mieux. En l'absence de cadre rigoureux, il est facile de réaliser sans s'en douter des opérations douteuses… et fausses.
Le programmeur confirmé, lui, utilisera comme un atout ce mélange possible.

4.4.1 Les conversions implicites

Les conversions implicites sont effectuées par le compilateur pour l'évaluation d'une expression.
Prenons le cas de l'affectation suivante :

```
var_destination = expression ;
```

où *expression* peut comporter un mélange de variables de types différents et d'opérateurs. Par exemple,

```
var_double = var_int * var_float ;
```

Les règles sont les suivantes :

- Le type de la variable de destination (*Leftvalue*) n'intervient pas pendant le calcul de l'expression située à droite de l'opérateur =. Ce n'est qu'**après** le calcul de celle-ci que la valeur est éventuellement convertie pour s'exprimer selon le type de la variable de destination.

- L'expression à droite de l'opérateur = est évaluée par défaut **de la gauche vers la droite** en respectant les **priorités** des opérateurs rencontrés.
- Afin de fournir deux opérandes de même type à l'opérateur qui va être appliqué, le compilateur convertit si nécessaire l'opérande la plus « faible » dans le type de la variable occupant le plus de place en mémoire. Il existe donc une hiérarchie pour les conversions :

char < short int < int < long int < float < double

Exemple 9. ***Conversions implicites***

```
var_double = var_int * var_float ;
```

Le produit est effectué dans le type dominant *float* (*var_int* est pour cela convertie en *float* par le compilateur). Puis son résultat est converti en *double* au moment de l'affectation à la variable *var_double* (sans perte d'information dans ce sens).

```
var_float = 45 + var_int * var_double ;
```

On commence par calculer le produit (prioritaire) qui est effectué dans le type dominant *double*. Puis son résultat (de type *double*) est ajouté (addition réelle) au *double* résultant de la conversion de l'entier 45. Enfin, le résultat *double* de l'addition est converti (c'est-à-dire tronqué) en *float* au moment de l'affectation à la variable *var_float*, avec perte d'information.

4.4.2 Les conversions explicites : l'opérateur de *cast*

La conversion implicite effectuée par le compilateur, ou le type utilisé pour un calcul, ne conviennent pas toujours au programmeur : celui-ci peut forcer la conversion d'une expression quelconque **dans le type de son choix** grâce à l'opérateur unaire de conversion appelé **cast**. C'est par exemple très utile pour obliger un calcul à se faire en réel bien que ses opérandes soient des entiers (piège de la division entière…).

Sa syntaxe d'un *cast* est la suivante : `(type) expression`

Exemple 10. ***Conversions explicites par « cast »***

```
entier = (long int)1000*200/4 ;
```

Le cast s'applique à la valeur 1000 qui est converti en *long* → cela permet au reste du calcul de se faire en *long* et ainsi d'**éviter un dépassement** lors de la multiplication. Le résultat de la division finale sera converti si nécessaire lors de l'affectation dans le type de la Leftvalue *entier*.

```
entier_a_l_echelle = (double)n/NMAX*100 ;
```

Nous avons ici un cas classique de **règle de trois** pour une mise à l'échelle entre 0 et 100 d'un entier compris à l'origine entre 0 et NMAX. Le cast de l'opérande *n* en *double* permet de faire **tout le calcul en réel**. Le résultat sera converti en entier lors de l'affectation finale (voir exemple suivant).

```
partie_entiere = (short int)var_double ;
```

Ce cast permet de convertir la variable réelle *var_double* en entier en « tronquant" la partie décimale. Attention : ce n'est pas un arrondi à la valeur la plus proche : 3.99 sera converti en 3 (ce qui correspond bien à la définition de la partie entière en mathématiques).
Subtilité pour les nombres négatifs : -3.99 sera converti en -3 (alors que les mathématiques définissent la partie entière comme -4 dans ce cas).

```
arrondi = (short int)(var_double + 0.5) ;
```

Ce cast permet de réaliser un arrondi à la valeur la plus proche : 3.99 sera arrondi en 4 et 3.49 sera arrondi en 3. Attention : cela ne marche que pour les valeurs positives.

Terminons avec les pièges de la division entière.

Exemple 11. Division entière et conversion explicite par « cast »

Soient *n* et *p* deux variables **entières** valant 10 et 3. On cherche à effectuer leur division **réelle** et à stocker le résultat dans une variable réelle *res_reelle*.

Solution 1 (fausse) : `res_reel = (double)(n/p);`

l'expression entière n/p est ici convertie en *double* **après** division et *res_reelle* vaut 3.0. En raison des parenthèses, l'opérateur force la conversion du **résultat** de l'expression et non celle des valeurs qui la composent. La division reste une divison entière et notre but n'est pas atteint.
Notons que le cast explicite est superflu, car le compilateur réalise de toute façon cette conversion.

Solution 2 (juste) : `res_reel = (double)n/p ;`

le cast s'applique ici sur le premier opérande *n* et permet à la division de s'effectuer en *double* alors qu'elle s'effectuait précédemment en entier. Cette fois, nous n'avons plus une division entière, mais une division réelle. Au final, *res_reelle* vaut 3.3333.

☞ Un cast *(short int)* ou *(long int)* effectué sur un réel revient à prendre la partie entière de la valeur absolue, puis à « remettre le signe » (sous réserve que la valeur initiale reste dans les limites du nouveau type). C'est parfois bien utile.

☞ Un cast *(char)* sur un entier revient à prendre sa valeur modulo 256.

4.5. Les opérateurs relationnels *: inférieur, supérieur, égal, différent…*

Souvent associés aux opérateurs logiques, ils permettent de **comparer** des expressions pour effectuer des **tests**, fréquents en informatique.

Exemples : **a < 100** **a == b** **a >= b** **a != b** mais pas **1<a<10** (voir Exemple 14.)

Contrairement aux autres langages, le résultat en Langage C d'un opérateur relationnel ou logique n'est pas une valeur booléenne (vrai ou faux), mais un **entier** qui vaut :

- **0 si le résultat de la comparaison est faux** ;
- **1** si le résultat de la comparaison est **vrai**.

Opérateur	Signification
<	inférieur à
<=	inférieur ou égal à
>	supérieur à
>=	supérieur ou égal à
==	égal à ***ATTENTION à ne pas confondre avec = qui est l'affectation !***
!=	différent de

Les opérateurs relationnels

La notation == de l'opérateur d'égalité ne doit pas être confondu avec l'opérateur d'affectation =. En cas de confusion, certains compilateurs (pas tous) vous fournissent un *Warning* (qu'il faut toujours lire !).

4.6. Les opérateurs logiques : *ET OU NON*

Ces opérateurs logiques permettent de relier entre elles plusieurs conditions pour réaliser une condition **multiple**. Ils sont souvent utilisés en association avec les opérateurs relationnels. Opérateurs logiques et relationnels ont en commun de fournir un résultat sous la forme d'une **valeur entière (0 si faux, 1 si vrai).**

Le Langage C possède trois opérateurs logiques :

Opérateur	Signification
&&	**ET** logique (résultat 0 ou 1)
||	**OU** logique (résultat 0 ou 1)
!	**NON** logique (résultat 0 ou 1)

Les opérateurs logiques

Les opérateurs logiques acceptent comme opérandes toutes les valeurs numériques (réels compris !). Ils considèrent que :

- 0 correspond à *faux* ;
- toute valeur **non nulle** (et pas seulement 1) correspond à *vrai*. *-4.76* est considéré comme *vrai*…

Comment le compilateur évalue t-il une expression logique ?

- Les expressions reliées par des opérateurs logiques sont évaluées **de la gauche vers la droite**.
- L'évaluation prend fin quand le résultat d'une expression entraîne le résultat définitif pour l'expression globale ; les expressions situées plus à droite ne seront alors pas évaluées.

Exemple 12. Expressions avec opérateurs logiques et relationnels

`a<b && c<d` prend la valeur 1 (vrai) si les deux conditions a<b et c<d sont toutes les deux vraies (de valeur non nulle), la valeur 0 (faux) dans le cas contraire.

`a<b || c==0` prend la valeur 1 (vrai) si l'une au moins des deux expressions a<b et c==0 est vraie (de valeur non nulle), la valeur 0 (faux) dans le cas contraire.

☞ Dans les conditions ci-dessous, les parenthèses sont inutiles étant donné les règles de priorité. Mais dans le doute, mieux vaut mettre deux parenthèses de trop.

`! (a<b)` prend la valeur 1 (vrai) si la condition a<b est fausse (de valeur 0), la valeur 0 (faux) dans le cas contraire. Cette expression est équivalente à a>=b.

Exemple 13. Evaluation des expressions avec les opérateurs logiques :

`expr1 && expr2`	expr2 ne sera évaluée que si expr1 est vraie
`expr1 || expr2`	expr2 ne sera évaluée que si expr1 est fausse

Exemple 14. Exemple piège : test d'un encadrement

On cherche à savoir si un entier n est compris entre 10 et 20.

Solution du débutant (fausse) : `if ( 10<n<20 ) …`

l'expression n<10 vaut 0 ou 1 selon qu'elle est vraie ou fausse. C'est cette valeur 0 ou 1 qui est ensuite comparée à 20, avec un résultat évidemment vrai. Au final, la condition `10<n<20` est toujours vraie !

Solution juste : **`if ( n>10 && n<20 ) …`**

Il faut écrire deux conditions testées successivement (ET logique). Remarquons que la deuxième condition sera testée seulement si la première est vraie.

4.7. Opérateurs de manipulation de bits – masques (ET bit à bit, décalage…)

Quand on veut modifier ou connaître **certains bits d'une variable** au lieu de sa totalité, il faut utiliser les opérateurs de manipulation de bits. **Ces opérateurs sont très utiles en informatique industrielle.**

Une opération bit à bit s'applique à **chaque bit** de ses opérandes, lesquels sont des entiers.

Les opérateurs de manipulation de bits disponibles

Le Langage C offre la possibilité de réaliser des opérations bit à bit du type :

- **ET, OU, OU EXCLUSIF** (entre deux entiers) ;
- complément à 1 ;
- opérations de **décalage** à droite ou à gauche (sur un entier).

Les opérateurs C correspondants sont donnés dans le tableau suivant :

Opérateur	Signification	Exemple
&	ET bit à bit	a & b
|	OU bit à bit	a | b
^	OU EXCLUSIF bit à bit	a ^ b
>>	décalage à droite	a >> 2 (décalage à droite de 2 bits)
<<	décalage à gauche	a << 5 (décalage à gauche de 5 bits)
~	complément à 1 (NON bit à bit)	~a

Les opérateurs de manipulation de bit

Illustrons cette table avec des exemples d'utilisation :

Exemple 15. Opérateurs de manipulation de bits

Soient a et b deux entiers (signés ou non) qui valent en hexadécimal et en binaire :

a = 0x03EF = $(0000.0011.1110.1111)_2$
b = 0x8049 = $(1000.0000.0100.1001)_2$

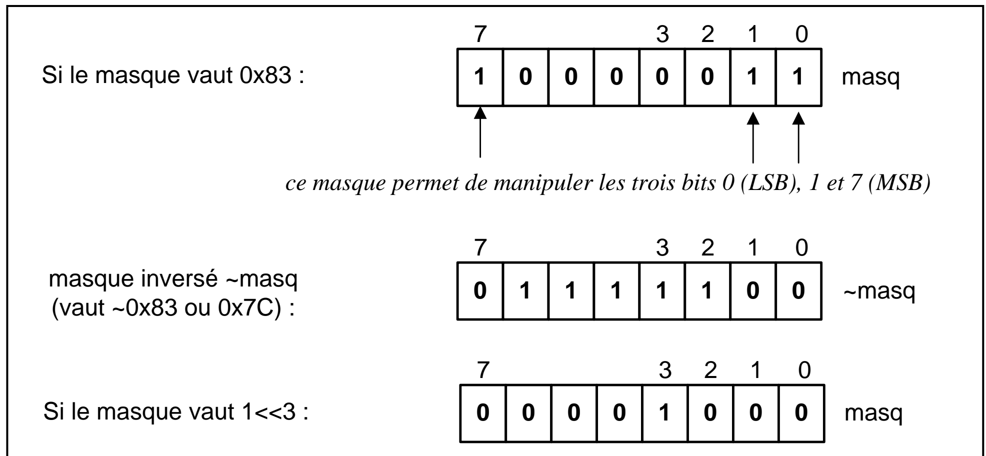
Alors :

a & b = 0x0049 = $(0000.0000.0100.1001)_2$
a | b = 0x83EF = $(1000.0011.1110.1111)_2$
a ^ b = 0x83A6 = $(1000.0011.1010.0110)_2$
~a = 0xFC10 = $(1111.1100.0001.0000)_2$
a>>3 = 0x007D = $(0000.0000.0111.1101)_2$
a<<4 = 0x3EF0 = $(0011.1110.1111.0000)_2$

Les opérateurs de décalage précisent le nombre de bits du décalage (a>>5 pour un décalage de 5 bits vers la droite, b<<1 pour un décalage de 1 bit vers la gauche).
Dans le décalage à gauche, les "bits de gauche" sont perdus et des **bits 0 apparaissent à droite**. Dans le décalage à droite, les "bits de droite" sont perdus et des **bits 0 ou 1 apparaissent à gauche** (0 si *a* est non signé ; dépendent de la machine si *a* est signé).

Une notion très utile : le masque.

Les opérateurs de bits sont souvent employés avec la notion de **masque**, en informatique industrielle notamment.
Un masque est une **succession binaire destinée à indiquer la position des bits concernés** par une opération (voir dessin). Un '1' dans le masque indique que le bit est concerné, un '0' qu'il ne l'est pas. On peut aussi utiliser le « masque inversé » pour certaines opérations comme la mise à 0 d'un bit.
Le masque s'exprime en général sous la forme d'une **constante hexadécimale** (0xF0, 0x03, 0x40...) ou peut être **fabriqué par décalage** s'il concerne un seul bit (**1<<6**, **~(1<<n)**).



Comment utiliser le masque ?

☞ **Une mise à 1 se fait avec l'opérateur OU, une mise à 0 par l'opérateur ET.**

Voici quelques applications très utiles des opérateurs de manipulation de bits (voir les dessins des exemples plus loin). Ils utilisent tous comme opérandes **une variable *a* et un masque** qui indique l'emplacement des bits concernés :

- **accès à certains bits d'une valeur en masquant les autres bits** (utilisation de l'opérateur ET) :

a & 0x0F permet d'isoler les 4 bits de poids faible de *a* (masque des 4 bits LSB : 0x0F).

a & (1<<n) pour connaître la valeur du $n^{ème}$ bit de *a* (expr. nulle si le bit est nul). Voir dessin. Si n=2, le masque **1<<n** vaut 1<<2 = $(00 ...00100)_2$. Notez que les parenthèses autour de 1<<n ne sont pas indispensables.

a & (1<<n) est nul si le $n^{ème}$ bit de *a* vaut 0, non nul s'il vaut 1.

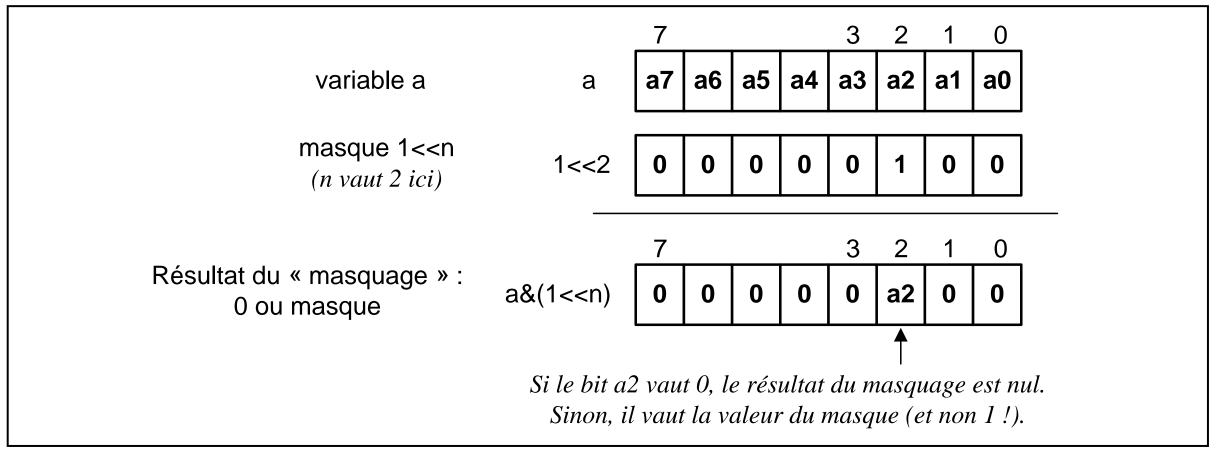
- **mise à 0 ou à 1 de certains bits d'une valeur** :

a = a | 0x0F met à 1 les 4 bits de poids faible de *a* ;

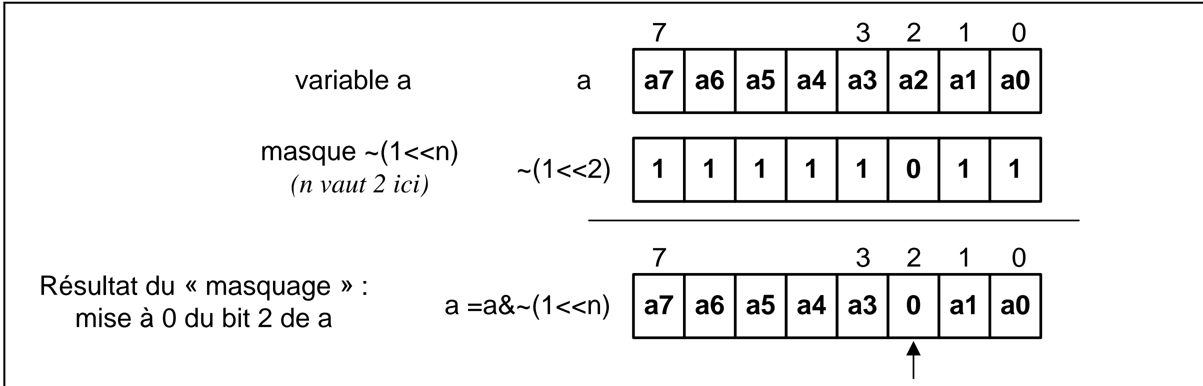
a = a & ~(1<<n) met à 0 le $n^{ème}$ bit de *a*. Voir dessin.

a = a | (1<<n) met à 1 le $n^{ème}$ bit de *a*.

Exemple 16. Lecture d'un bit par l'expression `a & (1<<n)`



Exemple 17. Mise à 0 d'un bit par l'expression `a = a & ~(1<<n)`



Exemple 18. Mise en œuvre d'un masque pour connaître la valeur d'un bit

```
/* Affiche le MSB (Most Significant Bit) d'un octet */
#define MASQ 0x80          /* masque hexa du MSB = (1000 0000) en binaire */
unsigned char port ;

/* on suppose que la variable port a une valeur (par exemple lue sur un port)... */
if ( (port & MASQ) != 0 )        printf("Le MSB du port est 1") ;
   /* Attention au test : port & MASQ ne vaut pas 0 ou 1, mais 0 ou MASQ !
Attention aussi à la présence des parenthèses, à cause de la priorité plus faible de & */
```

Exemple 19. Décalage des 4 bits MSB et mise à 1 de bits

```
/* Transforme l'octet {abcd xxxx} en l'octet {1111 abcd} */
#define MASQ 0xF0          /* masque hexa des 4 bits MSB = 1111 0000 */
unsigned char octet ;

 /* on suppose que la variable octet a une valeur (par exemple lue sur un port)... */
octet = octet >> 4 ;       /* décalage des 4 bits MSB */
octet = octet | MASQ ;     /* mise à 1 des 4 bits MSB */
```

4.8. L'opérateur d'adresse &

Le seul moyen de connaître l'adresse mémoire d'une variable est d'utiliser l'opérateur de calcul d'adresse **&** :

&var

représente l'adresse de la variable *var* (l'adresse du premier octet de la zone mémoire qu'elle occupe).

L'opérateur d'adresse est très utile comme argument d'appel pour les fonctions qui utilisent un passage en paramètre par adresse. Exemple : `scanf("%lf ", &toto) ;`

4.9. Les opérateurs d'incrémentation et de décrémentation ++ --

Ces opérateurs sont réservés aux entiers.

Deux opérations d'affectation courantes du type : i=i+1 (« incrémentation ») et j=j-1 (« décrémentation ») peuvent être remplacées par l'emploi des opérateurs d'incrémentation et de décrémentation **++** et **--**. On écrira alors :

i++ **incrémente la variable i de 1**

j-- **décrémente la variable j de 1**

Les opérateurs ++ et -- peuvent se placer avant ou après leur opérande, mais la valeur de l'expression obtenue (si elle est utilisée) n'est alors pas la même. Ainsi :

i++ incrémente la variable i de 1 → l'expression vaut la valeur de i **avant incrémentation**.
++i incrémente la variable i de 1 → l'expression vaut la valeur de i **après incrémentation.**

Ainsi, si x=5, alors :
y=x++ ; est équivalent à y=x; suivi de x++; donc y vaut 5 et x=6 à la fin de l'exécution de l'instruction.
y=++x ; est équivalent à x++; suivi de y=x; donc y vaut 6 et x=6 à la fin de l'exécution de l'instruction.

Exemple 20. Incrémentation et décrémentation

nb_trouv++ ;	utilisation la plus courante : l'incrémentation est seule sur sa ligne.
for (i=10 ; i>0 ; i--) …	++ et – sont très utilisés avec la boucle for.

Comme exercice, prévoir les résultats de ces deux programmes :

```
void main(void)
       {
       short int x=3, y ;
       y = x ++ ;
       printf("x=%hd - y=%hd", x, y);
       }
```

```
void main(void)
       {
       int x=3, y ;
       y = ++x ;
       printf("x=%hd - y=%hd", x, y);
       }
```

Solution : à la fin du programme, y vaut 3 à gauche, 4 à droite.

- ne pas abuser des ++ et -- au milieu d'expressions : la lisibilité est vite nulle ! Trois lignes claires sont préférables à une seule incompréhensible. De façon générale, la clarté d'un programme ne doit pas être sacrifiée à la concision.
- ne jamais utiliser les opérateurs ++ et -- dans des expressions qui figurent en paramètre d'un appel de fonction.

4.10. L'opérateur *sizeof* (taille en octets)

Pour rendre un programme portable et pour s'éviter des calculs fastidieux, il ne faut jamais écrire "en dur" le nombre d'octets occupés par une variable : l'opérateur ***sizeof*** est là pour ça.

sizeof fournit la taille (en octets) du type ou de la variable qui suit. Sa syntaxe est :

sizeof (type ou nom de variable)

Exemple 21. Opérateur sizeof

int n ;	**sizeof(n)** peut valoir 2 ou 4 selon l'ordinateur et le logiciel .
double x ;	**sizeof(x)** vaut 8
	sizeof(float) vaut 4
	sizeof(char) vaut 1

L'opérateur ***sizeof*** ne se limite pas aux types simples ; il est surtout utile pour les tableaux, les structures, les pointeurs... Il est très utile, car il rend les programmes plus portables.

5 - Les structures de contrôle

*Les exemples de ce chapitre utilisent souvent la fonction d'**affichage à l'écran** **printf**, ou plus rarement la fonction de saisie **scanf**. Elles seront décrites au chapitre suivant.*

Dans un programme, la plupart des instructions sont exécutées de façon **séquentielle**, c'est-à-dire **à la suite les unes des autres.** Mais pour « donner de l'intelligence » au programme, des instructions de contrôle sont nécessaires pour :

- effectuer des **choix** en fonction des circonstances (**exécutions conditionnelles**) ;
- effectuer des itérations (« boucles »), c'est-à-dire r**épéter un ensemble d'instructions** autant de fois que nécessaire.

L'utilisation de ces instructions de contrôle suppose que des **tests** soient effectués ; ceux-ci font appel aux opérateurs relationnels (>, <, ==, != ...) et logiques (&&, ||, !), vus au chapitre "Opérateurs".

Exemple 22. "Boucle" et exécution conditionnelle :

```
/* Ce programme calcule l'entier n tel que la somme  1+2+3+4+…+n  soit juste supérieure à 1000 */

void main(void)
      {
      long n, som ;
      n=1; som = 0 ;
      while (som<1000)              /* on sort de la boucle quand som dépasse 1000 */
            {
            som = som + n ;
            n++;                    /* « incrémentation ». Equivaut à    n=n+1;  */
            }
      n-- ;          /* on a incrémenté i une fois de trop, alors on compense en sortie de la boucle */
      /* affichage des résultats : */
      printf("Pour obtenir une somme des premiers entiers juste superieure
           a 1000, il faut sommer jusqu'a %ld (somme obtenue=%ld)", n, som);
      if ( som == n*(n+1)/2 )
            printf("\nLes maths confirment ce resultat.");
      }
```

DEUX POINTS INDISPENSABLES POUR LA LISIBILITE DE VOS PROGRAMMES :

☺ Les accolades ouvrante et fermante qui encadrent un bloc d'instructions doivent être **alignées verticalement**. *Pour cela, il suffit d'acquérir le réflexe de taper l'accolade fermante immédiatement après l'accolade ouvrante (avec frappe de « ENTREE » entre les deux).*

☺ Toutes les structures de contrôle nécessitent d'utiliser des « **indentations** » (décalage à droite du texte) pour les instructions qu'elles encadrent. L'indentation peut être faite avant ou après l'accolade ouvrante. Dans ce document, elle est toujours faite avant (choix).

5.1. Les répétitions *: for, while, do…while, continue, break*

On veut pouvoir répéter une action (c'est-à-dire une suite d'instructions) un certain nombre de fois. Deux cas se présentent :

- **le nombre de répétitions est prévisible** (c'est le cas le plus fréquent) ; on utilise alors de préférence l'instruction ***for*** ;
- la répétition de l'action doit avoir lieu tant qu'une condition est vraie, **sans qu'on puisse prévoir le nombre de répétitions qui seront nécessaires** ; on utilise alors de préférence les instructions ***while*** et ***do while***.

5.1.1 L'instruction *while*

L'itération ***while*** permet de **répéter une action tant qu'une condition est vraie**, sans connaître à l'avance le nombre de répétitions.

Elle réalise l'instruction en « pseudo-code » suivante :

tant que *condition vraie*
faire { ... }

Sa syntaxe en Langage C est :

```
while ( condition )
      {
      instructions ;
      }
```

Les instructions du bloc *instructions* sont **répétées** tant que l'expression ***condition*** **reste vraie** (non nulle). Cette expression est testée **avant l'exécution de l'itération** : si l'expression est fausse (nulle) dès le départ, les instructions ne seront jamais exécutées.

Pas de point-virgule sur la ligne *while* !

L'instruction à répéter peut être simple ou composée (c'est-à-dire formée d'une séquence d'instructions appelée bloc) ; le **bloc d'instructions** est entouré d'accolades { et }.

Exemple 23. Utilisation de while pour répéter une saisie
avec calcul par une boucle *for* de la factorielle n! = n*(n-1)* ...*2*1

```
void main(void)
{
short int n=1, copie_n ;
double fact ;               /* type réel pour éviter les dépassements dans le calcul */

printf("\n Entrez un entier (negatif si vous voulez sortir) : ") ;
scanf("%hd", &n) ;          /* saisie au clavier (non protégée) */
while (n>0)        /* pour éviter un dépassement lors de l'affichage : while (n>0 && n<17) */
      {
      fact=1.0 ;
      for (copie_n=n ; copie_n>1 ; copie_n-- )
            {
            fact = fact * copie_n ;
            }
      printf("\n\t La factorielle demandee est %ld (%.2le)",(long)fact,fact);
      printf("\n\n Entrez un entier (negatif si vous voulez sortir) : ") ;
      rewind(stdin) ;  scanf("%hd", &n) ;      /* saisie au clavier (non protégée) */
      }
}
```

Sauf cas particulier (par exemple en informatique industrielle sur un microcontrôleur), il faut absolument éviter les **boucles infinies** du style :

while(1) {…}

Une telle boucle est syntaxiquement correcte (1 est non nul, donc la condition est toujours vraie). Mais sur un PC, elle est douteuse du point de vue programmation et on peut toujours l'éviter.

Si on utilise ce genre de boucle, il faut prévoir une possibilité de sortie, par *return* (voir chapitre "Fonctions") ou par *break* (voir paragraphe correspondant) ou par la fonction *exit* (qui met fin à l'exécution du programme).

5.1.2 L'instruction *do while*

Comme *while*, l'itération ***do while*** permet de répéter une action tant qu'une condition est vraie. Mais la condition est cette fois testée **en fin d'itération**.
Elle réalise donc l'instruction en « pseudo-code » suivante :

répéter
{ … }
tant que condition vraie

Sa syntaxe est :

do
{
***instruction*(s) ;**
}
while (*condition*) ;

Le traitement est effectué une première fois, puis la condition est testée. Si elle est vraie, on répète le traitement ; sinon l'exécution passe à l'instruction suivante.

A la différence de *while*, la condition est testée **après** l'exécution de l'itération : le traitement a donc lieu **au moins une fois**, même si l'expression de la condition est fausse (nulle) dès le départ.

☞ Cette fois, il y a un point-virgule sur la ligne *while* : il s'agit en effet de la fin de l'instruction.

Exemple 24. Utilisation de do ... while

```
void main(void)
      {
      short int n=-1 ;     /* valeur par défaut pour prévenir certaines erreurs de saisie */
      do
            {
            printf("\n Entrez un entier positif : ") ;
            rewind(stdin) ;
            scanf("%hd", &n) ;
            }
      while (n<0) ;        /* reboucle aussi si on tape une lettre */

      printf("\n La racine carree de %hd est %lf", n, sqrt(n) ) ;
      }
```

☞ Une boucle *while* correctement initialisée peut souvent être utilisée à la place d'un *do… while*.

5.1.3 L'instruction *for*

C'est l'itération la plus utilisée, grâce en particulier à son utilité pour les tableaux. En général, elle est employée quand **<u>le nombre de répétitions est prévisible</u>** (dès la compilation ou bien à l'exécution).

Dans sa forme la plus utile et la plus simple, elle fait intervenir l'initialisation d'un **compteur**, son incrémentation et un test de fin de comptage.

Sa syntaxe générale est :

```
for ( [action initiale] ; [test de continuation] ; [action d'evolution] )
    {
    instructions ;
    }
```

Dans cette syntaxe figurent trois expressions séparées par des points virgule :

- **l'action initiale** est effectuée avant l'itération ; elle consiste souvent à initialiser le compteur de boucle ;
- le test de **<u>continuation</u>** de boucle (et non de fin de boucle !) est une expression qui est évaluée **à chaque <u>début</u> d'itération** : tant que l'expression est vraie (non nulle), les instructions sont exécutées ;
- **l'action d'évolution** modifie la valeur de la condition : elle consiste souvent à incrémenter le compteur de boucle. Elle est effectuée **<u>à chaque fin</u> d'itération** (après les instructions et avant le test de continuation).

Voici un premier exemple, le plus courant : celui où un compteur de boucle (*cpt*) permet de répéter une action un nombre de fois connu (ici, 100).

```
for ( cpt=1 ; cpt<=100 ; cpt++ )
    {
    Action_à_répéter_100_fois() ;
    }
```

Les trois expressions qui composent l'instruction *for* sont facultatives.

☞ Les instructions *for* et *while* sont en réalité équivalentes en Langage C. Mais *for* est plus compacte, donc plus utilisée..

En effet, l'instruction :

```
for ( expr1 ; expr2 ; expr3 )
   {
   instructions ;
   }
```

peut toujours être remplacé par :

```
expr1 ;              /* action initiale (avant l'itération) */
while( expr2 )       /* test de continuation (en début d'itération)*/
       {
       instructions ;
       expr3 ;       /* action d'évolution (effectuée en fin d'itération) /*
       }
```

Exemple 25. Utilisation de la boucle *for*

Exemple 1 : affiche les entiers de 1 à 10 inclus

```
void main(void)
    {
    short int i ;
    for (i=1 ; i<=10 ; i++)
          printf("\n i=%hd",i) ; /* accolades optionnelles car une seule instruction */
    /* attention : à la sortie de la boucle, i vaut maintenant 11 ! */
    }
```

Exemple 2 : calcul de la factorielle de n n! = n*(n-1)*(n-2)* ...*3*2*1

```
void main(void)
    {
    short int n=5, copie_n ;      /* n est initialisé à 5 pour le test */
    double fact ;                 /* réel pour éviter les dépassements dans le calcul */
    fact=1.0 ;
    for ( copie_n=n ; copie_n>1 ; copie_n-- )
          fact = fact * copie_n ; /* accolades optionnelles car une seule instruction */
    printf("\n La factorielle demandee est %12.5le", fact) ;
    }
```

Exemple 26. Boucles *for* imbriquées

```
void main(void)
       {
       short int lig, col ;
       for (lig=1 ; lig<=3 ; lig++)
             {
             printf("\n ligne=%hd", lig) ;
             for (col=3 ; col>0 ; col--)
                    printf("\n\t colonne=%hd",col) ;
             }
       printf("\n boucle terminee : ligne=%hd", lig) ;
       }
```

Ce programme affiche :

```
ligne=1
       colonne=3
       colonne=2
       colonne=1
ligne=2
       colonne=3
       colonne=2
       colonne=1
ligne=3
       colonne=3
       colonne=2
       colonne=1
boucle terminée : ligne=4          (eh oui...)
```

5.1.4 L'instruction *continue*

L'instruction ***continue*** insérée dans une boucle permet de **sauter les instructions restantes du traitement** en cours, puis de **continuer l'itération**. L'action d'évolution est exécutée dans le cas du *for*. Le test de continuation est effectué dans tous les cas.

☺ Comme pour l'instruction *break* qui suit, il faut éviter d'utiliser ce genre d'instructions de "rupture de boucle" (auxquelles appartient le célèbre et hideux *goto*, strictement prohibé). Le bon programmeur trouve généralement une autre solution…

☞ Les instructions de ruptures de séquence sont à réserver aux « sorties d'urgence » (quand le programme risque de « planter » si on continue). C'est le marteau brise-vitre des trains !

Exemple 27. Utilisation de *continue* dans une itération

Exemple syntaxiquement correct, mais douteux du point de vue de la qualité de programmation.

```
void main(void)                              /* EXEMPLE A NE PAS SUIVRE ! */
      {
      short int i ;
      for (i=1 ; i<5 ; i++)
            {
            printf("\n La valeur %hd de i vous convient-elle <o/n> ?", i) ;
            rewind(stdin);                   /* vide le tampon pour la lecture d'un caractère */
            if (getchar()=='n') continue ;   /* saut conditionnel à l'itération suivante */
            printf("\n La valeur i=%hd est retenue", i) ;
            }
      }
```

Cet exemple montre ce qu'il ne faut pas faire : répéter une action en sautant une partie de l'itération avec *continue*. Pour éviter d'exécuter la ligne `printf…` sans utiliser l'instruction *continue*, la bonne solution consiste à l'intégrer dans un *if* bien écrit (remarquez l'inversion de la condition…) :

Exemple 28. Remplacement de *continue* dans une itération *for*

```
void main(void)
      {
      short int i ;
      for (i=1 ; i<5 ; i++)
            {
            printf("\n La valeur %hd de i vous convient-elle <o/n> ?", i) ;
            rewind(stdin);            /* vide le tampon pour la lecture d'un caractère */
            if (getchar() != 'n')     /* la ligne suivante sera exécutée sous condition */
                  printf("\n La valeur i=%hd est retenue", i) ;
            }
      }
```

5.1.5 L'instruction *break*

L'instruction ***break*** permet de **sortir de toute itération** (*for*, *while* ...) qui la contient. Son effet est limité à un seul niveau d'imbrication.
Elle est surtout utilisée avec l'instruction **switch** qui sera étudiée au paragraphe correspondant.

☺ A l'exception de son utilisation dans *switch*, *break* doit être évitée au même titre que *continue* : les "ruptures de boucle" ne sont pas les bienvenues dans une programmation propre.

break permet de sortir de boucles apparemment infinies du type *while(1)*[2]. Mais aucune programmation propre (hors microcontrôleur) n'utilise ce genre de boucle infinie !

```
while(1)          /* on évite au maximum cette boucle infinie (sauf sur microcontrôleur)  */
        {
        ...
        if (expression)   break ;
        ...
        }
```

Dans ce cas, on ne connaît pas à l'avance le nombre d'itérations. Donc la bonne solution consiste à utiliser une boucle *while* avec une condition bien écrite.

5.2. Les exécutions conditionnelles : if…else, switch, ?:

5.2.1 L'alternative if – else et le test if

L'instruction *if* permet d'effectuer une action si et seulement si une certaine condition est satisfaite. Elle se complète éventuellement d'une partie **else**, optionnelle.

L'instruction *if-else* permet de réaliser l'alternative en « pseudo-code » suivante :

si *condition* alors faire { ... }
sinon faire { ... }

Sa syntaxe est :

```
if (condition)            /* condition : voir 4.5 et 4.6 */
    {
    suite d'instructions 1 ;
    }
else                          /* partie facultative */
    {
    suite d'instructions 2 ;
    }
```

Il n'y a pas de point-virgule sur les lignes *if* et *else*.

☺ Notez le **décalage à droite** (ou « indentation », obtenu par la touche **tabulation** du clavier) des instructions conditionnelles du *if* ou du *else*. Ce décalage est INDISPENSABLE.

L'instruction *else* est facultative, l'instruction *if* peut être utilisée seule. Par exemple :

```
if (a==b)          /* comparaison de a et b par l'opérateur « double égal » */
    {
    printf("a et b sont égaux" ) ;
    }       /* ici, accolades facultatives, car une seule instruction */
```

Si l'expression a==b est vraie (non nulle), l'instruction d'affichage *printf* est effectuée ; sinon, l'exécution reprend à la ligne suivante. C'est l'expression la plus simple de l'instruction *if*.

L'instruction *if* complétée avec *else* permet de choisir entre deux actions (alternative). Ainsi :

```
if (a>=b)    distance = a-b ;      /* accolades facultatives ici  */
else         distance = b-a ;
```

[2] Une telle boucle est fréquente en Informatique Industrielle (sur microcontrôleur), mais doit être évitée sur PC.

Si l'expression a>=b est vraie (non nulle), l'instruction distance = a-b est exécutée ; dans le cas contraire (expression nulle), c'est l'instruction distance = b-a qui est exécutée.

Exemple 29. Utilisation de if - else :

```
void main(void)              /* on suppose qu'une bibliothèque de fonctions est fournie */
  {
  short int code, BonCode = 4321 ;
  code = saisir_code_utilisateur() ;
  if (code == BonCode)          /* comparaison de a et b par l'opérateur « double égal » */
      {
      ouvrir_porte();
      }
  else
      {
      printf("\n Vous n'êtes pas autorisé à continuer!");
      exit(0) ; /* termine "brutalement" le programme */
      }
  /* suite du programme */
  }
```

Il est possible d'imbriquer les instructions *if*. Pour lever l'ambiguïté, en l'absence de parenthèses bien placées, C associe toujours le *else* au *if* sans *else* le plus proche.

☺ Un bon usage des tabulations est indispensable pour rendre le programme lisible, surtout avec des if-else imbriqués.

Exemple 30. Ecriture de *if-else* imbriqués (deux tabulations sont nécessaires) :

```
            if (expression1)
                  {
                  if (expression2)
                        { instruction2.1 ; }
                  else
                        { instruction2.2 ; }
                  }
            else
                  { instruction1 ; }
```

Certains *if-else* imbriqués traduisent un choix multiple qu'on ne peut pas réaliser avec *switch*, parce que le choix ne se fait pas en fonction d'une variable entière. Par exemple, lorsqu'on compare successivement une chaîne de caractères à plusieurs chaînes fixes. Il est alors bien pratique (et c'est la seule exception à la règle de style qui impose les tabulations) de les écrire au même niveau avec ***else if*** :

```
/* Choix multiple (quand switch est inutilisable) : */
        if (condition1)             { ... }
        else if (condition2)        { ... }
        else if (condition3)        { ... }
        else                        { ... }
```

Exemple 31. Utilisation de *if - else if - else if* :

```
short int n ;      /* valeur à saisir au clavier par exemple (non fait ici) */
if (n>0)           printf("\n Le nb n est strictement positif") ;
else if (n<0)      printf("\n Le nb n est strictement négatif") ;
else               printf("\n Le nb n est nul") ;
```

On prendra garde de ne pas masquer avec des tabulations à quel *if* se rapporte un *else*.

L'exemple suivant illustre le danger des instructions *if-else* imbriquées. On s'efforcera d'éviter de telles imbrications.

Exemple 32. Danger des if-else imbriqués :

```
if (expr1)
        {       /* ces accolades sont indispensables, sinon le else se rapportera au if(expr2)  */
        if (expr2)
                { instruction2 ; }
        }
else        /*ce else se rapporterait à if(expr2) en l'absence des accolades qui entourent if(expr2)*/
        {  instruction1 ; }
```

5.2.2 L'opérateur d'alternative

Un opérateur conditionnel permet de simplifier l'écriture de tests du type :

```
if (A>B) max=A ; else max=B ;
```

Avec l'opérateur conditionnel, ce traitement devient :

```
max = A>B ? A : B ;
```

L'opérateur comporte le symbole **?** pour *alors* et le symbole **:** pour *sinon*. L'expression A>B est testée : si elle est vraie, la deuxième expression A est évaluée et donne sa valeur à *max* ; sinon, la troisième expression B est évaluée et donne sa valeur à l'expression globale.

La syntaxe de cet opérateur conditionnel est :

expression 1 ? expresssion 2 : expression 3 ;

L'expression 1 est testée : si elle est vraie, l'expression 2 est évaluée et donne sa valeur à l'expression globale ; sinon, c'est l'expression 3 qui donne sa valeur à l'expression globale.

Exemple 33. Utilisation de l'opérateur conditionnel ?:

```
void main(void)
      {
      short int i, max=10 ;
      printf("Entrez i : ") ;
      scanf("%hd", &i) ;
      i>max ? i=0 : i++ ;
      printf("\n Nouveau i : %hd", i) ;
     }
```

5.2.3 Le choix multiple : l'instruction *switch*

Il est utile de pouvoir choisir un traitement parmi plusieurs en fonction de la valeur d'une variable entière. S'il n'existe que deux actions possibles, l'instruction *if ... else* convient très bien. Sinon, un aiguillage direct vers le traitement voulu peut souvent être réalisé à l'aide de l'instruction *switch*.

La différence avec *if… **else** **if**…else* est qu'avec *switch*, la condition d'aiguillage doit porter sur la comparaison d'une variable **entière** avec des valeurs constantes, et non sur une condition quelconque (l'aiguillage en fonction de conditions comme *delta>0, delta<0* ou *mesure==0.0*, doivent être réalisés avec *if…else if*).

La syntaxe de l'instruction *switch* est :

```
switch( expr )          Les instructions entre crochets sont optionnelles
    {
    case constante_1 : [suite d'instructions 1 ;]
                       [break ;]
    case constante_2 : [suite d'instructions 2 ;]
                       [break ;]
    ...
    case constante_n : [suite d'instructions n ;]
                       [break ;]
    [ default : suite d'instructions ]
    }
```

L'aiguillage se fait en fonction de la valeur de l'expression *expr* : celle-ci doit être de type entier ou caractère (rappel : un caractère est un cas particulier d'entier).

Notez que si le choix ne se fait pas en fonction d'une expression entière, il faut recourir au ***if - else if - else if*** vu précédemment.

Si la valeur entière de *expr* correspond à une des constantes entières *constante_i*, le traitement correspondant est exécuté. Sinon, le traitement par défaut ***default*** est utilisé.

☺ Effectuez l'aiguillage en fonction d'une variable entière plutôt que d'une expression : *switch(choix)* est préférable à *switch(2*k+1)*

Exemple 34. Utilisation de l'aiguillage *switch* :

```
void main(void)
{
char operation ;
short int a=5, b=-7 ;

printf( "Choisir une opération (+ ou - ou *) : " ) ;
rewind(stdin) ;          /* vide le tampon clavier */
operation = getchar() ;

switch(operation)
     {
     case '+':    printf("\n a + b = %hd", a+b) ;
                  break ;
     case '-':    printf("\n a - b = %hd", a-b) ;
                  break ;
     case '*':    printf("\n a * b = %hd", a*b) ;
                  break ;
     default :    printf("\n Mauvais choix !") ;
 }
}
```

Après un *case*, la suite d'instructions est optionnelle. Plusieurs valeurs distinctes peuvent ainsi donner accès à une **même suite d'instructions** :

Exemple 35. Réunir des traitements communs avec *switch*

```
switch (expr)
                {
                case 1 :     instructions 1 ;
                             break ;
                case 2 :
                case 4 :
                case 5 :     instructions 2 ;        /* s'applique à 3 cas */
                             break ;
                case 8 :     instructions 3 ;
                }
```

Dans cet exemple, *instructions 2* sera exécutée si *expr* vaut 2, 4 ou 5.

L'instruction *break* est le seul moyen de sortir du *switch* quand le traitement relatif à une valeur a été exécuté : *break* fait passer à l'instruction qui suit le *switch*. Sans un *break* après *instruction 1* dans l'exemple précédent, *instruction 2* aurait été exécutée à la suite d' *instruction 1*.

L'exemple suivant illustre une utilisation très fréquente de l'instruction *switch* : l'écriture de menus. L'itération do ... while permet de revenir au menu à la fin du traitement d'une option et gère les mauvaises réponses de l'utilisateur. (*debug*)

Exemple 36. Utilisation de *switch* pour écrire un menu :

```
void main(void)
{
char choix, choix_entier ;

do
     {
     printf("Choisir une option dans le menu suivant : \
            \n\t Visualisation <1><return> \
            \n\t Modification <2><return> \
            \n\t Suppression <3><return> \
            \n\t Quitter <q><return> ") ;
     printf("\n\t\t Indiquez votre choix : ") ;
     rewind(stdin) ;
     choix = getch() ;
     choix_entier = choix-'0';     /*permet de passer du caractère 'i' (code ASCII) à l'entier i */

     switch(choix_entier)
            {
            case 1 :      printf("Appel de l'affichage") ;
                          break ;
            case 2 :      printf("Appel de la modification") ;
                          break ;
            case 3 :      printf("Appel de la suppression") ;
                          break ;
            }
     }
while( choix!='q' ) ;
}
```

6 - Les entrées/sorties conversationnelles (clavier/écran)

Le Langage C dispose d'un grand nombre de fonctions, fournies dans une bibliothèque standard, qui sont destinées à afficher des informations à l'écran ou à lire des données tapées au clavier : ces activités sont appelées *entrées/sorties conversationnelles.*

Exemple 37. Premier exemple de saisie au clavier et d'affichage :

```
#define POURCENT_RED 20.0          /* réduction accordée (en pourcent) */

void main(void)
      {
      double prix, reduc ;

      Cls();                /* Cls = fonction spécifique CVI (clear_screen) = efface l'écran */
      printf("Tapez le prix (en euros) avant réduction : ") ;
      scanf("%lf", &prix) ;

      reduc = prix*REDUC/100.0 ;
      printf("\n Vous gagnez %8.2lf euros avec %2hd %% de réduction",
           reduc, POURCENT_RED ) ;
     /* Le curieux %% dans les guillemets de printf permet d'afficher le caractère % à l'écran */
      }
```

Si on tape 50, le programme affiche :

Vous gagnez 10.00 euros avec 20 % de réduction.

6.1. Affichage à l'aide de la fonction *printf*

La fonction *printf* permet d'afficher sur la fenêtre de l'écran un texte qui suit un format défini par le programmeur. Sa syntaxe générale est :

printf("format", arg_1, arg_2, ..., arg_n) ;

où *format* représente une chaîne de caractères (placée entre guillements "") qui contient :

- du texte à imprimer tel quel ;
- des spécifications de format (autant que de paramètres *arg_i*) qui indiquent comment afficher les variables *arg_1, ..., arg_n* fournies en paramètre. Ce sont des **codes formats**.

Chaque **code format** commence par le **symbole % suivi par une (ou deux) lettre(s)** indiquant le format d'affichage du paramètre *arg_i* correspondant (celui qui a le même rang dans la liste des arguments). **La valeur de *arg_i* remplacera son code format à l'affichage.**

La lettre placée après le symbole % dans le code format indique le **type du paramètre** associé au code format (voir tableau page suivante). Dans l'exemple précédent, %lf (et ses variantes comme %8.2lf) est le code format associé à un *double*.

Les paramètres de *printf* peuvent être des variables, des constantes, des expressions à calculer préalablement à l'affichage...

Exemple 38. *printf* et ses codes formats

```
long int n = 9 ;
printf("Le carré de %ld est %ld, son cube est %ld", n, n*n, n*n*n) ;
        /* affiche : Le carré de 9 est 81, son cube est 729 */
```

N'utilisez pas un code format choisi au hasard :

- le format utilisé doit correspondre au type de la variable utilisée comme argument ;
- le nombre de codes formats doit être égal au nombre d'arguments. Si ce n'est pas le cas, C choisit d'obéir au code format.

Code format	Type de l'argument à afficher	Format d'affichage et exemples
%c	caractère	Ex : a G ? +
%hd **%hu** **%hX**	entier *short int* (ou caractère) **signé** entier *short int* (ou caractère) **non signé** pour afficher en hexadécimal (*short* ou *char)*	Base 10. Ex : -12 Base 10. Ex : 463 Hexadécimal. Ex : 9A0F
%ld **%lu** **%lX**	entier *long int* **signé** entier *long int* **non signé** entier *long int* en hexadécimal	Base 10. Ex : -1289 Base 10. Ex : 46399 Hexadécimal. Ex : B4E98A0F
%d **%X**	entier *int* pour afficher un *int* en hexadécimal	Base 10. Ex : -546 Hexadécimal. Ex : 9A0F
%lf **%le**	Réel double précision (*double*)	Virgule flottante. Ex : -3.141592 Avec exposant. Ex : -1.450000e-7
%f **%e**	Réel simple précision (*float*)	Virgule flottante. Ex : -3.141592 Avec exposant. Ex : -1.450000e-7
%s	Chaîne de caractères	Ex : bonjour !

Les codes formats

Des options peuvent s'insérer entre le symbole % et la lettre indiquant le type, afin de spécifier de façon plus précise la présentation de l'affichage. Ces options sont :

- **un nombre** qui indique le nombre de positions sur lequel doit s'écrire la valeur de la variable, **cadrée à droite** (si ce nombre n'est pas suffisant, le C n'en tient pas compte) :

```
short int n=11 ;
printf("L'entier n vaut %6hd \n", n)
        /* affiche : L'entier n vaut     11, puis saute à la ligne */
```

- **deux nombres séparés par un point** (très utile pour les réels) ; le premier indique le nombre total de positions occupées (il peut être absent), le deuxième précise le nombre de chiffres à écrire après la virgule :

```
double prix = 32.5, charge = 1.6E-19 ;
printf("prix = %8.2lf Francs \n", prix) ;
        /* affiche : prix = ___32.50 Francs (puis saute ligne suivante avec \n) */
printf("charge = %12.4le ou %.2le ", charge, charge ) ;
        /* affiche : charge =   1.6000e-19 ou 1.60e-19 */
```

- le **signe -** devant le nombre précédent pour demander de **cadrer à gauche** :

```
short int d=-12 ;
printf("La distance vaut %-6hd metres", d);   /* affiche : La distance vaut -12    metres */
```

Dans le texte à imprimer peuvent être insérés des caractères spéciaux composés du symbole \ suivi d'une lettre ; par exemple, **\n** dans l'exemple précédent représente un changement de ligne, **\t** permet l'introduction d'une tabulation horizontale (voir tableau).

Bien que ça ne soit utile que pour certains caractères spéciaux, un caractère à imprimer peut également être désigné par son code ASCII hexadécimal ; par exemple **\x**041 permet d'afficher le caractère 'A'.

La table des codes ASCII est fournie en annexe.

Code format	Code ASCII	Caractère	Effet à l'écran
\n **\t** **\r**	10 9 13	*LF* *HT* *CR*	Changement de ligne Avance d'une tabulation horizontale Retour en début de ligne courante
%% **** **\"**		**%** **** **"**	Écrit le caractère à l'écran. Ces caractères ne sont pas utilisés comme les autres, car ils ont une signification particulière dans *printf*.
\x*HHH*	0xHHH	n'importe lequel	HHH = valeur hexa du code ASCII

Les représentations des codes ASCII usuels

La fonction *printf* renvoie un entier qui constitue un compte-rendu d'exécution : cette valeur retournée correspond au nombre de caractères affichés. Cette information n'est utile que pour régler des problèmes de mise en page.

☞ Pour continuer à la ligne suivante un texte trop long dans une instruction *printf*, il faut utiliser le caractère \ :

```
printf("Pour continuer une chaîne de car sur la ligne suivante, \
il faut utiliser l'antislash ou scinder en 2 printf");
```

6.2. Lecture au clavier à l'aide de la fonction *scanf*

6.2.1 Mise en oeuvre de *scanf*

La fonction *scanf* permet de lire les informations tapées au clavier par l'utilisateur selon un certain format. Les informations lues sont converties en caractères, entiers ou réels suivant le format attendu.

Il existe une différence fondamentale entre *printf* et *scanf*. Pour *printf*, c'est la **valeur** de la variable qui est passée en argument à la fonction. Dans le cas de *scanf*, cette valeur est inconnue : c'est elle qui sera lue au clavier, puis rangée dans un emplacement mémoire qui reste à spécifier. C'est donc **l'adresse** de cet emplacement qu'il faut fournir en argument à la fonction *scanf* pour qu'elle puisse y ranger la valeur lue au clavier. L'argument fourni à *scanf* est **l'adresse de la variable** où *scanf* doit ranger la valeur lue au clavier.

En C, l'adresse d'une variable se représente en faisant précéder le nom de la variable du symbole & :

l'adresse de la variable *toto* est *&toto*.

La syntaxe de la fonction *scanf* est :

nb_val_lues = scanf("format", arg_1, arg_2, ..., arg_n) ;

où *format* représente une suite de codes formats, construits comme avec *printf*, et *arg_1, ..., arg_n* sont les **adresses** des variables où doivent être rangées les valeurs lues par *scanf*. Par exemple :

`nb_val_lues = scanf("%ld", &toto);` si *toto* est une variable *long int*

La valeur renvoyée par *scanf* est le nombre de valeurs que *scanf* a réellement réussi à lire : elle sert à protéger les saisies contre les erreurs de l'utilisateur (voir paragraphe 6.2.3 *Valeur retournée par scanf*).

Exemple 39. Saisie (non protégée) par *scanf*

Un exemple de saisie "blindée" sera vue à l'exemple du paragraphe correspondant.

```
void main(void)
{
short int entier ;
double reel ;

Cls() ;              /* Cls = fonction spécifique CVI (clear_screen) = efface l'écran */

printf("Entrez un entier dans <-32768,+32767> : ") ;
scanf("%hd", &entier) ;
printf("\n Pour verification : l'entier lu est %hd", entier) ;

printf("\n Entrez un reel double : ") ;
scanf("%lf", &reel) ;
printf("\n Pour verification : le reel double lu est %lf", reel) ;
}
```

L'oubli de l'opérateur d'adresse & n'entraîne pas de message d'erreur de la part du compilateur. Mais la valeur lue sera écrite n'importe où en mémoire !

On peut lire plusieurs valeurs numériques dans une même instruction *scanf*. Les valeurs à lire doivent être séparées par un **séparateur** lors de la frappe au clavier : les caractères *ESPACE*, fin de ligne (touche *ENTRÉE*) ou tabulation sont des séparateurs. *scanf* saute tous les séparateurs avant de constituer une nouvelle valeur numérique.
Ainsi, en réponse à l'instruction

```
scanf("%hd%hd", &entier1, &entier2) ;
```

on peut taper indifféremment 526___12(*ENTRÉE*) ou __526(*ENTRÉE*) ou (*TAB*)_12_. *scanf* sautera les espaces (représentés par _), les fins de ligne (*ENTRÉE*) et les tabulations (*TAB*) avant de lire la valeur numérique suivante.

Les codes formats successifs dans une même instruction *scanf* doivent se suivre **sans le moindre espace** (scanf("**%hd%f%hd**",...). Si un espace ou toute autre chaîne de caractères séparent deux codes formats, ils devront être **exactement** retrouvés lors de la saisie des valeurs, ce qui laisse peu de marge à l'utilisateur.

Par exemple, si l'instruction est **scanf("%hd, %hd", &entier1, &entier2)** (les 2 codes formats sont séparés par une virgule et un espace), l'utilisateur devra impérativement taper une virgule et un espace entre les deux entiers (par exemple : 23,_6) sous peine de provoquer une mauvaise lecture.

C'est utile pour imposer le format d'une saisie, comme celle d'une date avec le format jj/mm/aa :

```
scanf("%hd/%hd/%hd ", &jour, &mois, &annee) ;
```

6.2.2 Gestion du tampon par *scanf*

Deux tampons pour la saisie

Le fonctionnement de *scanf* s'explique par l'existence de deux tampons entre le clavier (c'est-à-dire le système d'exploitation) et le programme : le tampon système et le tampon du C.
Les caractères entrés par l'utilisateur sont d'abord placés dans le tampon système ; ils ne sont copiés dans le tampon du C que lorsque l'utilisateur tape *ENTRÉE*. *scanf* va lire les informations qui lui sont nécessaires dans le tampon du C : les caractères sont retirés du tampon au fur et à mesure des conversions **réussies** vers les types de variables de destination.
Il y a mise en attente **seulement lorsque le tampon du C est vide** (curseur clignotant à l'écran).

Une **conversion non réussie** (type de la variable d'accueil non adapté à la donnée) provoque un mauvais fonctionnement des lectures **suivantes** : la donnée incorrecte est présentée en réponse à toutes les interrogations *scanf* qui suivent, jusqu'à ce qu'elle soit absorbée par l'une d'elles, libérant ainsi le tampon du C.

De même, un **excès de données** en entrée perturbe les lectures suivantes : les caractères excédentaires restent dans le tampon et sont proposés en réponse aux interrogations *scanf* qui suivent.

Saisie de caractères : attention danger !

A la fin de la dernière conversion effectuée par une instruction *scanf*, il reste dans le tampon du C au moins le dernier caractère fin de ligne (caractère '\n' généré par la frappe de la touche *ENTREE*). Ce reliquat du tampon sera présenté lors de la prochaine interrogation *scanf*. Si *scanf* cherche à lire une valeur numérique, '\n' sera considéré comme un séparateur que *scanf* sautera naturellement ; par contre, si la première donnée que cherche à lire *scanf* est un caractère, celui-ci sera automatiquement chargé avec '\n' et aucune interrogation ne sera affichée (pas de curseur clignotant)... Le programmeur doit donc particulièrement garde à la gestion du tampon lors de la lecture de caractères : la solution est *rewind*.

Nettoyer le tampon avant la saisie : *rewind*

Les explications précédentes montrent qu'il est important de s'assurer que le tampon du C est nettoyé avant d'utiliser *scanf* , surtout lors de la lecture de caractères. C'est pourquoi la bibliothèque standard du C met à notre disposition une fonction ***rewind*** qui permet de **vider le tampon du C**. Sa syntaxe est :

rewind(stdin) ; /* vide le tampon avant une lecture */

stdin est le flux standard d'entrée, en principe la console. De même, le flux standard en sortie *stdout* représente généralement l'écran.
La fonction *rewind* peut souvent être remplacée par la fonction *fflush* (qui s'utilise de la même façon que *rewind*), mais celle-ci n'est pas standard.

Exemple 40. Lectures incorrectes par *scanf* et utilisation de *rewind* :

```
void main(void)
 {
 short int i, j ;
 char carac ;

 printf("Tapez < 1& > (->faute) : ") ;
 scanf("%hd", &i) ;
/* à la suite de cette saisie de i apparement réussie, les caractères '&' et '\n' restent dans le tampon */

 printf("Entrez j : ") ;
 scanf("%hd", &j) ;
 /* '&', résidu de la saisie précédente, est incorrect pour j. Il n'y a pas d'interrogation du clavier et la
variable j ne reçoit aucune valeur ! */

 printf("Entrez un caractere : ") ;
 scanf("%c", &carac) ;
 /* '&', toujours présent dans le tampon C,  est absorbe par carac. Il n'y a toujours pas d'interrogation du
clavier et la variable carac reçoit comme valeur le résidu de la saisie précédente ! */

 rewind(stdin);  /* vide le tampon */
 printf("Entrez un caractere : ") ;
 scanf("%c", &carac) ;
 /* le tampon est vide, donc interrogation du clavier : tout va bien maintenant */
 }
```

6.2.3 Valeur retournée par *scanf* - Saisie protégée

Comme *printf*, la fonction *scanf* renvoie un entier qui permet de tester son bon déroulement : la valeur retournée est le **nombre de valeurs réellement lues**. Elle est indispensable pour effectuer des **saisies protégées** contre les erreurs de frappe de l'utilisateur ou pour vérifier que la saisie s'est bien passée.

Exemple 41. Saisie PROTEGEE (contre les erreurs de frappe) d'un entier :

/ Si on tape une lettre, scanf renvoie 0 et on reboucle tant que l'utilisateur ne tape pas un entier */*

```
short int entier ;
rewind(stdin);   /* vide le tampon de lecture */
while ( scanf("%hd", &entier) != 1)
      {
      printf(" Valeur incorrecte ! Tapez un entier : ") ;
      rewind(stdin) ;     /* "nettoie" le tampon de lecture */
      }
rewind(stdin);     /* prépare la saisie suivante */
```

Exemple 42. Boucle et saisie protégée

```
/* Ce programme affiche la racine carrée de x s'il est positif, puis recommence */
 void main (void)
      {
      double x ;

      printf("\n Tapez un reel (ou une lettre pour sortir) : ") ;
      rewind(stdin);        /* vide le tampon de lecture = tampon clavier */

      while (scanf("%lf",&x)==1)  /* saisie et test de la valeur renvoyée par scanf */
         {
         if ( x > 0. )
              printf("\n La racine carree de %lf est %lf", x, sqrt(x));
         else
              printf("\n %lf est negatif.", x ) ;

         printf("\n Tapez un reel (pour sortir, tapez une lettre) : ");
         rewind(stdin) ; /* nettoie le tampon de lecture en vue de la saisie suivante */
         }

      rewind(stdin);        /* prépare la saisie suivante en vidant le tampon clavier */
      . . . /* main peut se poursuivre si souhaité */
      }
```

6.3. D'autres fonctions d'entrées/sorties *: getchar, putchar, GetKey…*

Pour la lecture et l'écriture de caractères, le Langage C offre d'autres fonctions et macros-instructions que *printf* et *scanf*.

6.3.1 Les macro-instructions *putchar* et *getchar*

La macro *putchar* écrit le caractère passé en paramètre sur le flux standard de sortie, en principe l'écran. Elle est équivalente à l'instruction `printf("%c",c)`, mais plus rapide. Elle se trouve dans le fichier en-tête standard *<stdio.h>*, qui doit donc impérativement être inclus dans le fichier source à l'aide de la directive *#include*.

La syntaxe de *putchar* est :

```
#include <stdio.h>
 char carac = 'a' ;
 putchar(carac) ;          /* équivalent à printf("%c", carac) */
 putchar('A') ;
```

La macro *getchar* lit un caractère sur le flux standard d'entrée, en principe le clavier. Elle est équivalente à l'instruction `scanf("%c", &c)`, c'est-à-dire qu'elle effectue la lecture après la frappe de *ENTRÉE* et fait l'écho sur l'écran du caractère lu. Elle se trouve dans le fichier *<stdio.h>*.

La syntaxe de *getchar* est :

```
#include <stdio.h>
char carac ;
carac = getchar() ;   /* équivalent à scanf("%c", &carac) */
```

Cette macro-instruction *getchar* travaille avec le même tampon que *scanf*, c'est-à-dire le tampon du C.

Notez que la fonction *rewind* permettant de vider le tampon du C est équivalente à l'instruction :

```
while (getchar() != '\n') ;        "boucle" tant que le tampon n'est pas vide */
```

6.3.2 La fonction *getch* (spécifique à Borland) ou *GetKey* (spécifique à CVI)

Les fonctions dont nous allons parler ne sont pas portables : elles ne font pas partie du Langage C normalisé et dépendent du logiciel.

La fonction *getch* est réservée aux logiciels Borland et son équivalent *GetGey* à CVI. Elle ne figure pas dans la norme ANSI : elle n'est donc pas portable. Mais elle est très utile pour introduire une attente dans un programme (le programme s'arrête et ne se poursuit qu'après la frappe d'une touche).

La fonction *getch / GetKey* retourne un caractère depuis le clavier. Mais à la différence de *scanf* ou *getchar* :

- elle utilise le tampon du système au lieu du tampon du C : le caractère est donc lu **immédiatement** après qu'il ait été tapé par l'utilisateur, **sans attendre la frappe de *ENTRÉE*** ;
- elle ne fait pas l'écho du caractère lu, c'est-à-dire n'affiche pas le caractère sur l'écran.

Sa syntaxe générale est :

```
#include <conio.h>        /* pour Borland seulement */
char carac ;
carac = getch( ) ;          // ou avec CVI :     carac = GetKey( ) ;
```

Exemple 43. Utilisation de getch (Borland) / GetKey (CVI) :

```
#include <conio.h>                    /* avec Borland seulement */
...
printf( "Tapez 'q' pour sortir" );
if ( getch()=='q' ) exit(0) ;         /* exit termine "brutalement" le programme */
/* ou avec CVI :
if ( GetKey()=='q' )…  */
...
printf( "Tapez une touche pour continuer") ;
getch() ;                             /* pour attendre la frappe d'une touche */
/* ou avec CVI :   GetKey() ;  */
```

Remarquons l'existence chez Borland d'une fonction *getche* (*getch* avec écho) qui fonctionne comme *getch* (pas d'attente de ENTRÉE), mais fait l'écho à l'écran du caractère tapé.

7 - Utilisation de fonctions

C'est une notion essentielle pour tous, programmeurs débutants ou aguerris : l'appel de fonction, c'est-à-dire l'**utilisation de sous-programmes déjà écrits**, est un besoin qui se fait sentir rapidement dès que vous commencez à écrire vos premiers programmes.

Notez que la « **définition** de fonction », qui est l'écriture du **code interne** de la fonction, est une notion plus évoluée qui sera vue ultérieurement. Pour commencer, nous allons **utiliser** des fonctions **toutes faites**, disponibles dans une des nombreuses **bibliothèques** de fonction du Langage C, de l'environnement de déve-loppement (IDE) ou fournies par un autre programmeur.

Exemple 44. Appels de fonction, fichiers en-tête avec prototypes

```
    #include <math.h>      /* inclusion des fichiers en-tête des bibliothèques de fonction utilisées */
    #include "ma_biblio.h"      /* en-tête d'une bibliothèque située dans le dossier de travail */

void main(void)
    {
    double a, b, module, argument ;

    a =  saisir_reel( );        /* 1er appel de la fonction saisir_reel */
    b =  saisir_reel( );        /* 2ème appel de la fonction saisir_reel */

    module = sqrt( a*a+b*b ) ;   /* appels des fonctions sqrt et atan (cf biblio mathématique)*/
    argument = atan( b/a );

    afficher_2_val("Cx en polaire (R, Theta): ", module, argument );
    }
```

Extrait du fichier en-tête *math.h* (bibliothèque standard disponible dans tous les IDE)

```
extern double atan(double);       // extern est optionnel dans le prototype
extern double sqrt(double);
```

Extrait du fichier en-tête *ma_biblio.h* (bibliothèque fournie par un autre programmeur)

```
//------------------- saisie non protégée -----------------------------
double saisir_reel (void);

//--------- affichage de 2 valeurs précédées par un message------------
void   afficher_2_val (char *message, double x, double y) ;
    // message est la chaîne de caractères à afficher (cf exemple)
    // les valeurs x et y sont affichées derrière message, séparées entre elles par un espace.
    // exemple d'appel : afficher_2_val ("la moyenne et le max sont", (a+b)/2, max );
```

Les fichiers en-tête contiennent des **prototypes** de fonction (éventuellement commentés), qui sont des **modes d'emploi** pour le programmeur et le compilateur. Les lignes d'appel de fonction (ici, dans *main*) doivent suivre le mode d'emploi du prototype sous peine d'être refusés lors de la compilation.

7.1. Un peu de vocabulaire

Une fonction est repérée par son identificateur (son nom). On lui fournit des informations appelées **arguments d'appel** et elle peut **retourner une valeur** (cas le plus fréquent).

Une bonne partie de la programmation moderne consiste à réaliser un travail d'assemblage en utilisant des **bibliothèques**[3] de fonctions toutes faites. La **définition** de la fonction, c'est-à-dire l'écriture de son code interne, revient alors au concepteur de la bibliothèque ; les autres programmeurs se contentent d'exploiter les **déclarations** de fonction (**prototypes**), fournies dans le fichier **en-tête** de la bibliothèque.

L'utilisateur d'une fonction n'a pas besoin d'avoir accès au code interne de la fonction (sa définition) : il doit savoir comment l'utiliser correctement (rôle du prototype), connaître la nature du résultat renvoyé ou du travail réalisé (importance des commentaires et du choix judicieux du nom de la fonction), mais pas comment la fonction procède en interne. On peut utiliser, par exemple, les fonctions de la bibliothèque mathématique en ignorant comment elles sont écrites en interne.

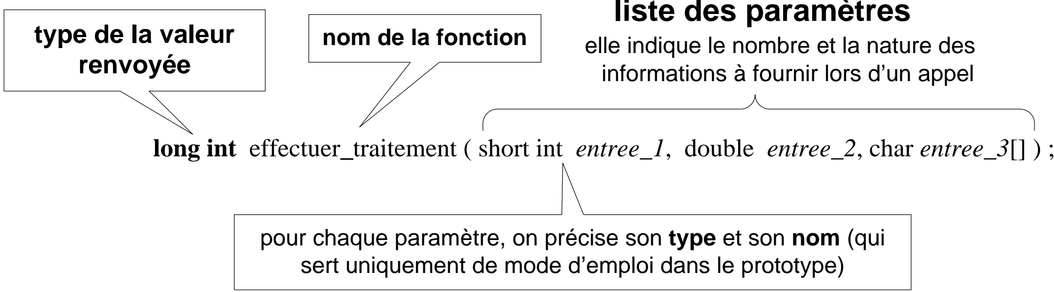
7.2. Le mode d'emploi d'une fonction : le *prototype* (ou *déclaration*)

Un prototype, ça sert à quoi ?

Le prototype est un **mode d'emploi** pour le programmeur qui veut utiliser une fonction : celui-ci connaît précisément le nombre des arguments d'appel à fournir, leur ordre, leur type, et la nature de l'éventuelle valeur renvoyée.

Mais les prototypes sont autant destinés au compilateur qu'au programmeur humain. Ils permettent au compilateur d'améliorer la qualité de son travail : grâce au prototype, il peut vérifier la validité d'un appel de fonction, mais aussi **forcer des conversions de type** si nécessaire.

Un prototype, ça ressemble à quoi ?



Regardez la page récapitulative en fin de chapitre : pour chaque fonction, vous y trouverez le dessin du **bloc fonctionnel**[4] (boîte noire), sa traduction en Langage C sous la forme d'un **prototype** et plusieurs exemples d'**appel** illustrant les arguments d'appel possibles.

Voir page récapitulative en fin de chapitre (principales fonctions simples rencontrées)

Exemple 45. Prototypes de fonction (les appels sont dans l'exemple suivant)

```
double calculer_delta( double a, double b, double c ) ;
void allumer_LED( short numero_LED ) ;
unsigned char lire_port_micro( void ) ;
void initialiser_micro( void ) ;
void afficher_message( char* msg ) ;
```

[3] En anglais, bibliothèque se dit *library*, d'où la mauvaise habitude répandue de parler de « librairie » de fonctions.

[4] Le bloc fonctionnel est la « **boîte noire** » de la fonction : c'est un rectangle avec des flèches rentrantes pour symboliser les données à fournir à la fonction et des flèches sortantes pour représenter les données récupérées en sortie.

De plus amples informations sur les prototypes seront fournies après avoir abordé la question « Comment écrire l'appel de fonction ? ».

Un prototype, ça se trouve où ?

Pour l'utilisateur d'une bibliothèque, les prototypes sont situés dans un fichier particulier appelé fichier **en-tête**, d'extension ***.h*** (*Header*). Ce fichier en-tête doit être inclus (recopié) **au sommet** du fichier source **.c** qui **utilise** la fonction, grâce à la directive :

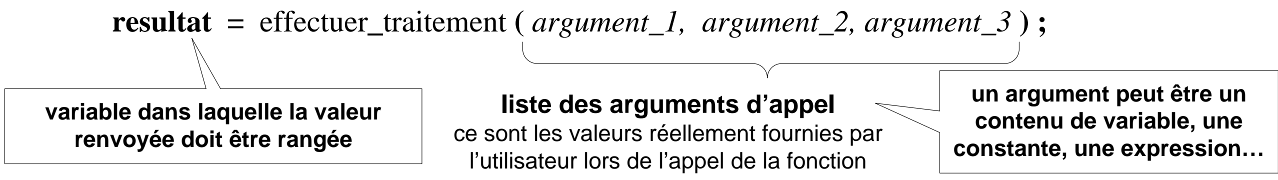
```
#include <fichier_entete_standard.h>      /* cf exemple du début de chapitre */
                Ou
#include "mon_fichier_entete.h"    /*si le fichier en-tête est situé dans le dossier courant */
```

Un prototype peut aussi être écrit au sommet du fichier source *.c* qui utilise la fonction. C'est surtout utile pour les fonctions que le programmeur définit pour son usage personnel (sans volonté de les diffuser), usage en général limité à un seul fichier.

7.3. L'utilisation de la fonction : l'*appel de fonction*

L'utilisation d'une fonction se fait sous la forme d'une ligne exécutable placée **dans le corps d'une autre fonction** : cet **appel de fonction** peut être répété aussi souvent que nécessaire, avec tous les arguments d'appel souhaités.

La syntaxe est proche de celle utilisée en mathématique :



La liste de valeurs entre parenthèses est la liste des **arguments d'appel**. Ce sont des valeurs numériques qui peuvent être des **contenus de variables** ou des **constantes,** ou même des expressions à calculer au préalable. Le nombre de paramètres, leur ordre et leurs types doivent être conformes au prototype, qui est le mode d'emploi pour écrire une ligne d'appel.

La valeur renvoyée par la fonction, si elle existe, est placée dans la variable *résultat* au retour de la fonction ; bien sûr, c'est vous qui choisissez la variable qui mémorise la valeur calculée par la fonction.
S'il n'y a pas de valeur renvoyée ou si celle-ci ne nous intéresse pas, l'appel de la fonction prend une forme simplifiée :

nom_fct (liste de valeurs) ;

Exemple 46. **Appels de fonction** (les prototypes sont dans l'exemple précédent)

```
discr = calculer_delta( coeff_X2, coeff_X, coeff_cst ) ;
discr = calculer_delta( -1.5, 6.0, 39. ) ;

allumer_LED( i_led+1 ) ;
allumer_LED( 3 ) ;

octet_lu = lire_port_micro( ) ;
initialiser_micro( ) ;

afficher_message( "coucou !" ) ;
afficher_message( phrase ) ;
```

Voir page récapitulative en fin de chapitre (principales fonctions simples rencontrées)

7.4. Plus d’informations sur les prototypes de fonction

Un prototype, c’est vraiment indispensable ?

Oui !!!
Si le compilateur "connaît" la fonction appelée (c'est-à-dire s'il a déjà compilé sa définition), le prototype est inutile pour lui. Mais l'utilisation d'une fonction a souvent lieu alors que la définition de la fonction n'est pas encore connue du compilateur : soit parce que la définition a lieu plus bas dans le même fichier source, soit parce que la définition est effectuée dans un autre fichier (cas très fréquent).

Grâce au prototype, le compilateur peut effectuer son travail lorsqu'il rencontre un appel de fonction :

- vérification du nombre d’arguments
- vérification de la concordance des types
- mise en place de conversions de type si elles sont nécessaires (et possibles).

En l'absence de prototype, le compilateur doit afficher un message (ou un *Warning*) d'erreur.

Si le compilateur est ancien ou mal configuré, il n’exige pas les prototypes. Il effectue donc moins de vérifications et, surtout, attribue à la fonction le **type par défaut *int*** : cela génère souvent des erreurs difficiles à détecter pendant l'exécution. Par exemple, la ligne `y=sin(x)`, en l’absence de prototype, aura pour effet de mettre à 0 la variable y ! (la valeur renvoyée par la fonction *sinus* est convertie en entier, soit presque toujours 0).

⇒ **Vérifiez que votre compilateur exige les prototypes** (cherchez le menu *Options du compilateur*).

Dans l'exemple du début de chapitre, sans la directive `#include <math.h>`, un compilateur qui accepte de travailler sans prototype va supposer par défaut que la fonction *sqrt* fournit un résultat de type **entier** (*int*) ; il mettra alors en place une conversion de la valeur de retour en *int*, ce qui conduira à un résultat faux…

Les lignes de prototype et les directives *#include* ne génèrent aucune instruction exécutable. Mieux vaut en mettre trop que pas assez.

Faut-il respecter à la lettre les types indiqués du prototype ?

Pas forcément !
Si le type de l'information transmise par l’utilisateur (cette information est appelée argument d’appel[5]) ne coïncide pas avec le type qui figure dans le prototype, le compilateur force la conversion de la valeur transmise pour qu'elle corresponde au prototype (à condition que ce soit possible !). Ces conversions sont particulièrement utiles dans le cas des fonctions mathématiques, dont les prototypes ne comportent que des types *double*, mais qui sont fréquemment appelées avec des arguments d’appel *int* ou *float* :

```
y = sin(5) ;   /* conversion en double faite par le compilateur */
```

Quel lien existe entre prototype et fichier en-tête ?

Les fichiers en-tête d'extension *.h* contiennent principalement des prototypes de fonctions. Ainsi, *math.h* contient les prototypes de toutes les fonctions mathématiques disponibles en bibliothèque, *stdio.h* contient les prototypes des fonctions d’entrée/sortie standard (clavier, écran, fichier…).

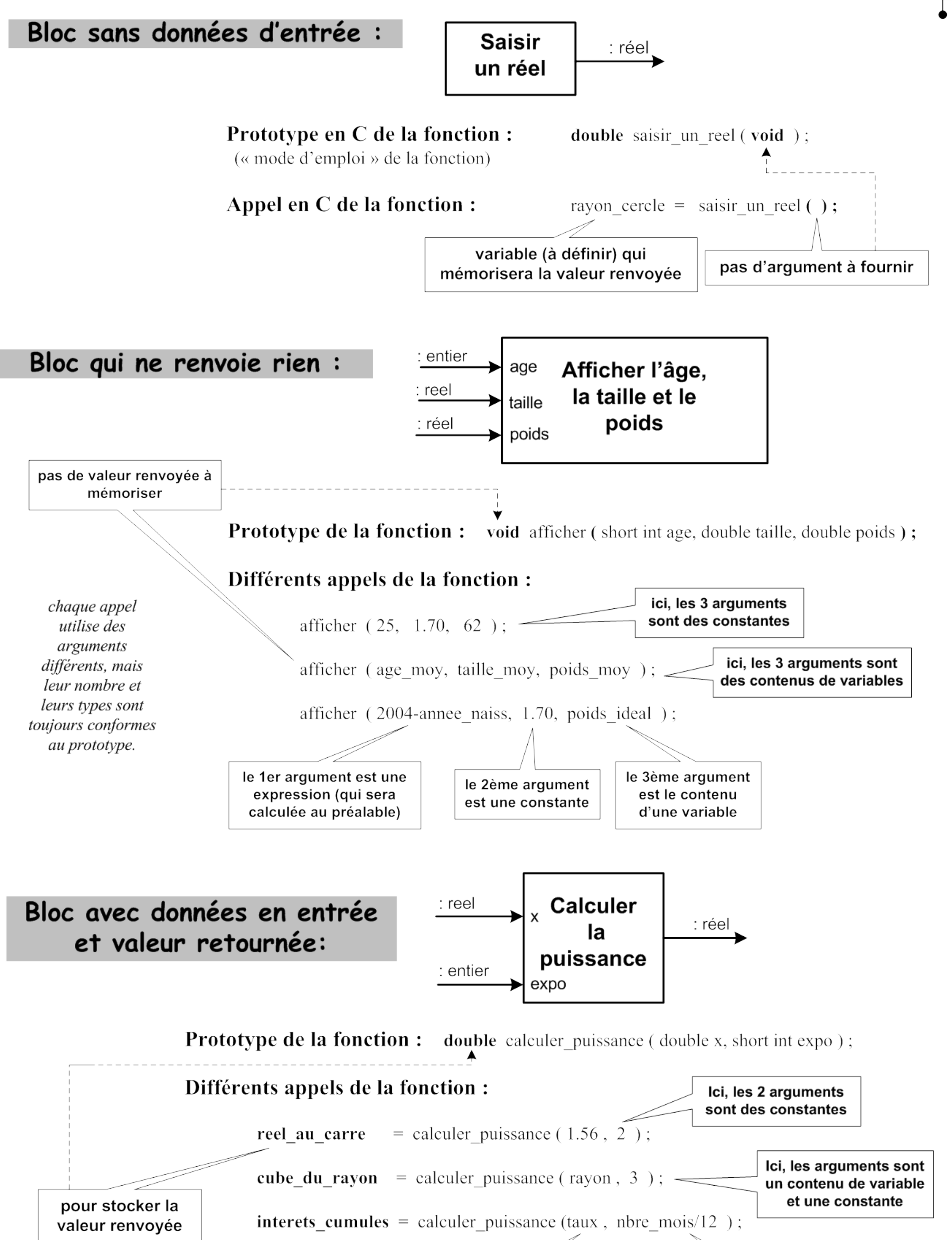
Toute utilisation d'une fonction doit être précédée de l'inclusion du fichier en-tête associé à la bibliothèque, pour que le compilateur dispose des prototypes :

```
#include < fichier_entete_standard.h>    (à placer au sommet du fichier utilisateur)
        ou
#include "mon_fichier_entete.h"    /*si le fichier en-tête est situé dans le dossier courant */
```

7.5. Récapitulation : prototypes et appels de fonction en une page

[5] Ou paramètre effectif (plus compliqué !)

Exemples de blocs fonctionnels simples
Comment les traduire et les utiliser en Langage C



Représentation d'un bloc fonctionnel simple

Du dessin du bloc à l'utilisation de la fonction en Langage C

Les noms de paramètres et les types indiqués sont des exemples.

type du paramètre

nom du paramètre **à l'intérieur** du bloc fonctionnel

bloc fonctionnel
ce dessin est un mode d'emploi pour l'utilisateur

Cas n°1 : la valeur retournée est une **valeur numérique unique calculée** par le bloc fonctionnel. Sa présence suppose que le traitement se déroule toujours sans problème.

Cas n°2 : la valeur retournée est un **code d'état** (on peut alors utiliser la flèche spécifique plus bas) qui indique comment s'est déroulé le traitement. Ce code d'état est indispensable dans le cas où le bloc fonctionnel peut rencontrer un problème.

: entier → entree_1
: reel → entree_2
: T_tableau → entree_3

effectuer un traitement

paramètres d'ENTREE

: entier → **valeur numérique calculée par le bloc (unique)**

une seule valeur retournée à choisir entre ces deux cas

int → **cas particulier où la valeur retournée est un code d'état (type *int* à utiliser en C)**

un paramètre d'entrée mémorise une **information fournie** au bloc. C'est une variable locale du bloc, qui n'est pas censé être modifiée.

La valeur retournée est ici un code d'état au lieu d'une valeur numérique : code<0 si problème, code>0 ou code=0 si traitement réussi sans problème.

(si nécessaire, le ou les résultats calculés doivent alors restitués à l'aide de paramètres de SORTIE. Mais ceci sort de ce cas simple.)

type de la valeur renvoyée

Traduction en Langage C de ce bloc fonctionnel : « prototype de fonction »

long int effectuer_traitement (long int *entree_1*, double *entree_2*, T_tableau *entree_3*) ;

ou

T_code_etat effectuer_traitement (long int *entree_1*, double *entree_2*, T_tableau *entree_3*) ;

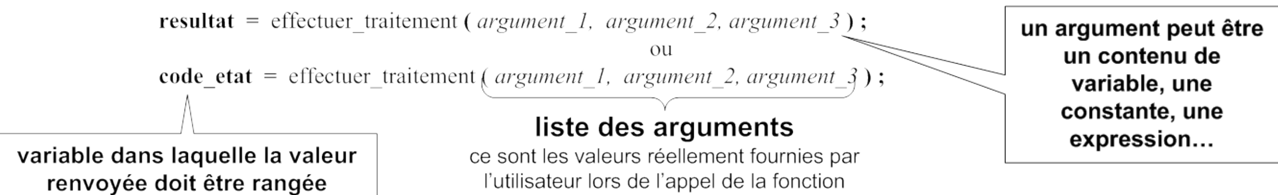
nom de la fonction

liste des paramètres
elle indique le nombre et la nature des informations à fournir lors d'un appel

pour chaque paramètre, on précise son **type** et son **nom** (celui qui sera utilisé **à l'intérieur** du bloc)

Utilisation en Langage C de ce bloc fonctionnel : « appel de fonction »

L'appel doit respecter le mode d'emploi que représente le prototype



8 - La bibliothèque de fonctions mathématiques *(sinus, exp, valeur absolue…)*

La bibliothèque[6] standard *math* du Langage C offre un certain nombre de fonctions mathématiques. Pour pouvoir les utiliser, il est nécessaire **d'inclure au début du fichier source la directive :**

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

Rappellons qu'une directive est un **ordre donné au préprocesseur** (le programme qui s'exécute automatiquement avant la compilation). La directive #include <math.h> insère (recopie) le texte contenu dans le fichier *math.h* à l'emplacement de la directive (c'est une substitution).

Le fichier *math.h* est un **fichier en-tête** (extension *h* pour *Header*), présent dans le dossier des bibliothèques standards du Langage C de votre IDE (dossier *include*). Il contient des prototypes de fonctions (mode d'emploi) et parfois des définitions de constantes symboliques (M_PI, M_E, M_SQRT_2, etc...)...

La liste suivante contient les fonctions les plus courantes. Le type de l'argument est indiqué entre parenthèses (`double x`) et le type de la valeur retournée est précisée devant le nom de la fonction : c'est toujours un réel *double*.

Prototype	Nom mathématique	Exemple
double sin(double x) ;	sinus	y = sin(x) ;
double cos(double x) ;	cosinus	
double tan(double x) ;	tangente	
double atan(double x) ;	arc tangente	
double exp(double x) ;	exponentielle	
double log(double x) ;	logarithme népérien	
double log10(double x) ;	logarithme décimal	
double pow(double x, double y) ;	x^y	puis = pow(x, 4) ;
double sqrt(double x) ;	racine carrée	
double fabs(double x) ;	valeur absolue réelle	

Les principales fonctions mathématiques de la bibliothèque standard

☞ Remarquez l'absence de la fonction « carré » x^2. Il est conseillé d'utiliser des expressions comme `x*x` ou `x*x*x` pour calculer x^2 ou x^3, plutôt que d'employer la fonction `pow(x,y)` (qui calcule $e^{y*\ln(x)}$ sur des réels double précision).

☞ On peut toujours appeler une fonction mathématique avec un argument entier ou *float*. Le compilateur effectue les conversions nécessaires.

💣 Les écritures françaises et anglo-saxonnes étant parfois contradictoires, il ne faut pas confondre le logarithme népérien (*log* en Langage C) avec le logarithme décimal (*log10* en Langage C… et non *log* !).

[6] En anglais, bibliothèque se dit *library*, d'où la mauvaise habitude répandue de parler de « librairie » de fonctions.

Pour une liste plus complète des fonctions et constantes disponibles, reportez-vous au manuel de référence du logiciel utilisé, ou ouvrez le fichier d'en-tête.

Exemple 47. Utilisation de la bibliothèque mathématique

```
#include <math.h>          /* fichier entête à inclure pour utiliser les fonctions mathématiques */

void main(void)
      {
      double x, y;
      double a=6.0, b=-1.5, module, argument ;

      module = sqrt( a*a+b*b ) ;
      argument = atan( b/a );

      for ( x=1.0 ; x < exp(5) ; x=x+1.0 )
            {
            printf( "Le log decimal de x=%lf est %lf", x, log10(x) );
            y = fabs(sin(x)) + exp(cos(x));
            }
      }
```

☞ Le fichier en-tête *math.h* de certains IDE inclut la définition de constantes mathématiques très utiles comme PI ou e. Mais pas tous !

9 - Définition de fonction

9.1. Pourquoi créer des fonctions dans mon programme ?

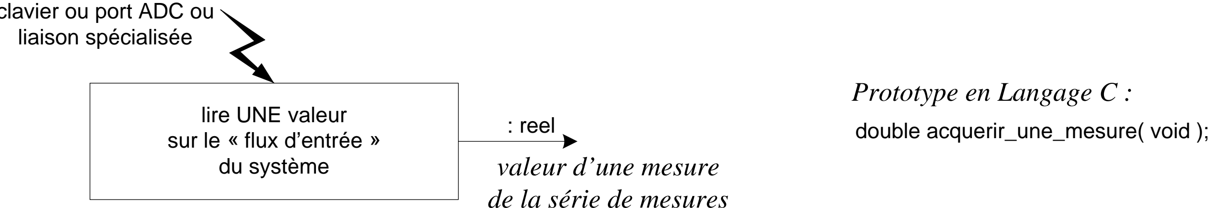
Plutôt que d'écrire sans arrêt les instructions constituant une action souvent répétée, il paraît logique de les regrouper sous la forme d'un **sous-programme**, qui se traduit en Langage C par la notion de **fonction**.

Mais c'est loin d'être le seul intérêt des fonctions : même s'il n'est exécuté qu'une seule fois, un bloc d'instructions peut être isolé sous la forme d'une fonction. L'objectif est alors de **rendre le programme modulaire** : cette notion clé des langages de haut niveau est indispensable pour qu'un programme soit **lisible**, **évolutif**, de maintenance aisée, développable par une équipe de personnes…

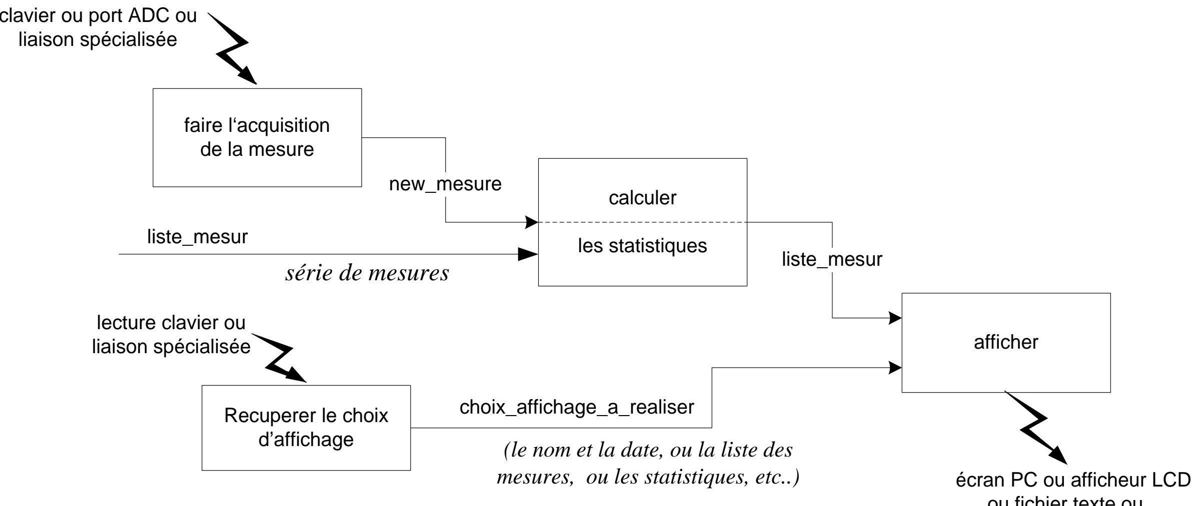
9.2. Dessinons : boîtes noires - découpage fonctionnel du programme

Un programme, ça commence par des dessins sur le papier !

Une étape essentielle de la conception d'un programme consiste à découper le programme en « sous-tâches » plus petites. Il est pratique de dessiner individuellement chaque module de programme (chaque future fonction) sous la forme d'une « **boîte noire »** (ou bloc fonctionnel), avec des flèches rentrantes pour symboliser les données à fournir au module et des flèches sortantes pour les données récupérées en sortie. Ce dessin de boîte noire sera traduit en Langage C sous la forme d'un prototype de fonction.



On représente la façon dont les différents modules sont reliés entre eux en dessinant les connexions (flèches) entre les différents boîtes noires : le dessin obtenu est le **découpage fonctionnel** du programme.



☺ Plus la complexité du programme s'accroît, plus il est indispensable de **dessiner** les boîtes noires et le découpage fonctionnel qu'on veut réaliser avant de penser "langage informatique" : un dessin est beaucoup plus parlant qu'une syntaxe textuelle rébarbative.

☞ Notez que cette démarche « diviser pour mieux régner » n'est pas spécifique à l'informatique : elle s'applique aussi à l'électronique, la mécanique etc.

Dessiner, d'accord… mais quel rapport avec le Langage C ?

La boîte noire, aussi appelée " bloc fonctionnel", est idéale à dessiner pendant la phase de conception sur papier. Mais il est ensuite nécessaire de la traduire sous forme textuelle dans un fichier source C (souvent un fichier en-tête) : ce sera le **prototype** de la fonction.

De même, le découpage fonctionnel du programme va se traduire en Langage C par une succession d'**appels** de fonction, par exemple dans la fonction *main*.

9.3. Premier avertissement : déclaration ≠ définition de fonction

En informatique, on distinguera bien les deux notions suivantes :

- **déclarer** une fonction, sous la forme d'un **prototype** : c'est la traduction en Langage C du dessin du bloc fonctionnel. Ce **mode d'emploi** de la fonction est destiné au programmeur humain et au compilateur.

Voir le chapitre « Utilisation de fonctions » pour plus de détails et des exemples.

Exemple de prototype tiré du chapitre « Un programme C plus évolué » :

```
double calculer_delta( double a, double b, double c ) ;
```

- **définir** une fonction : beaucoup plus complète que la déclaration qu'elle reprend dans sa ligne d'en-tête, la définition donne la suite des instructions qui composent la fonction, c'est-à-dire le **code interne** de la fonction.

Exemple de définition tiré du chapitre « Un programme C plus évolué » :

```
double = calculer_delta( double a, double b, double c )
   {
   return ( b*b - 4.*a*c ) ;
   }
```

☺ Une fonction réalise une **action** : il est judicieux de lui donner un nom composé d'un <u>**verbe**</u> suivi par un complément d'objet. Les appels de fonction seront alors plus lisibles et le programme pourra être compris dans ses grandes lignes par n'importe qui.

☺ Le nom d'un paramètre, qui apparaît dans la définition de la fonction et dans son prototype, doit être choisi avec soin pour que le prototype puisse jouer pleinement son rôle de "mode d'emploi pour le programmeur". Un nom de paramètre explicite remplacera avantageusement un commentaire.

Par exemple :

```
void afficher_statistiques( double moy, double ecart_type );
```

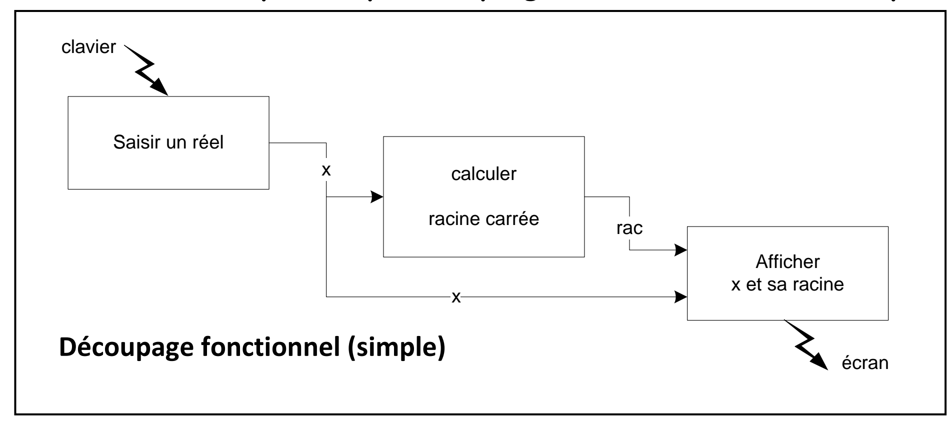
est beaucoup plus clair pour le programmeur que

```
void aff(double a, double b);
```

… qui convient pourtant au compilateur.

9.4. En détails : définition, déclaration, utilisation d'une fonction

Exemple 48. Un exemple complet de programmation modulaire simple




```
/******************** Déclarations des fonctions : ********************/
#include <math.h>              /* contient les prototypes des fonctions mathématiques */

double calculer_racine( double reel );   / prototypes des 3 fonctions définies dans ce fichier*/
double saisir_un_reel( void ) ;
void afficher_racine( double racine, double x ) ;

/********************* Définitions des fonctions : ********************/
void main(void)
{
double x, rac ;                      /* variables locales de main */
x = saisir_un_reel( ) ;              /* appel de la fonction de saisie */
rac = calculer_racine( x );          /* appel de la fonction de traitement */
afficher_racine( rac, x );           /* appel de la fonction d'affichage */
}
//-------------------------------------
double saisir_un_reel(void)          /* définition de la fonction de saisie */
{
double reel ;                        /* variable locale de la fonction saisir_un_reel */
printf("Entrez un reel : ") ;
scanf("%lf", &reel) ;
return reel ;
}
//-------------------------------------
double calculer_racine(double reel)     /* définition de la fonction de calcul */
{
double res ;                  /* variable locale de la fonction calculer_racine */
if (reel>=0)  res = sqrt(reel) ;
else          res = -1. ;
return res ;
}
//-------------------------------------
void afficher_racine( double rac, double x )   /* définition de la fonction afficher ...*/
{
if (rac==-1.)  printf( "Votre réel %lf est négatif!", x ) ;
else           printf( "La racine carrée de %lf est %lf",  x, rac);
}
```

L'exemple ci-dessus fait apparaître les trois phases qui interviennent dans la manipulation de fonction : déclaration (prototype), utilisation et définition. Nous allons détailler ces trois phases dans la suite de ce paragraphe, avant d'étudier le mécanisme du passage en paramètre à partir d'autres exemples.

9.4.1 Déclaration (prototype) d'une fonction

Nous avons déjà expliqué que le prototype est un **mode d'emploi** :

- pour le programmeur qui veut utiliser une fonction : il connaît alors précisément le nombre des arguments à fournir, leur ordre, leur type, et la nature de l'éventuelle valeur renvoyée.
- Pour le compilateur, qui effectue des vérifications et procède aux conversions nécessaires.

Pour fabriquer un prototype, il suffit souvent de **recopier l'en-tête** de chaque fonction, **terminé par un point-virgule**, au début du fichier. Le prototype (ou déclaration) de la fonction a pour syntaxe générale :

type_fonction nom_fonction(type par1, type par2 ...) ;

Si le type de l'information transmise par la fonction appelante (appelée argument d'appel ou encore paramètre effectif) ne coïncide pas avec le type qui figure dans le prototype, le compilateur forcera la conversion de la valeur transmise pour qu'elle corresponde au prototype (si c'est possible !).

☺ Il faut placer en tête d'un fichier les prototypes de toutes les fonctions qui y sont utilisées, même si leur définition suit. Cela permet de ne pas se soucier de l'ordre dans lequel se succèdent les définitions de fonction.

☺ Une meilleure solution consiste à regrouper les prototypes dans un ou plusieurs fichiers en-tête, qu'il suffit alors d'inclure au sommet du fichier utilisateur des fonctions.

Quel lien existe entre prototype et fichier en-tête ?

Les fichiers en-tête d'extension *.h* contiennent (entre autres) des prototypes de fonctions. Ainsi, *math.h* contient les prototypes de toutes les fonctions mathématiques disponibles en bibliothèque.

Vous serez amené rapidement à écrire vos propres fichiers en-tête, qui contiennent les prototypes des fonctions dont vous êtes l'auteur (voir chapitre sur la compilation séparée). Bien sûr, vous utilisez déjà les fichiers en-tête des bibliothèques de fonction fournies avec votre environnement de développement.

Voir le chapitre « Utilisation de fonctions » pour plus de détails et des exemples d'appel.

9.4.2 Définition d'une fonction

La syntaxe générale de la définition d'une fonction est la suivante :

```
type_fct nom_fonction(type par1, type par2…)  /* en-tête de la fonction */
      {
      type variable1 ;                        /* définition des variables locales */
      type variable2 ;
       ...
      instruction1 ;
      instruction2 ;
       ...
      return valeur_a_renvoyer ;              /* obligatoire si la fonction n'est pas void */
      }
```

L'en-tête de fonction

La première ligne est **l'en-tête** de la fonction, suivie par les accolades qui encadrent le **corps** de la fonction. Du point de vue syntaxique, **cet en-tête est identique, au point-virgule final près, au prototype de la fonction.**

Le type de la fonction, indiqué au début de l'en-tête avant le nom de la fonction, est le type du **résultat** que la fonction retourne à la fonction appelante : *long/short int, char, double*... Une fonction qui ne renvoie pas de résultat est de type ***void*** (vide).

Outre le nom et le type de la fonction, figure dans l'en-tête la liste des paramètres (on dit aussi paramètres formels). Cette liste, placée entre parenthèses derrière le nom de la fonction, précise le type et le nom des **variables dans lesquelles vont être mémorisées les informations fournies** lors d'un appel.

Il s'agit là, dans la ligne d'en-tête de fonction, d'une **définition de variables** (c'est-à-dire d'une réservation en mémoire) : les paramètres formels, qui sont des variables locales un peu particulières de la fonction, sont **créés** (alloués) sur la ligne d'en-tête de la fonction.

L'instruction *return*

L'instruction ***return*** permet de préciser quelle valeur doit être retournée à la fonction appelante. Cette instruction termine l'exécution de la fonction et redonne le contrôle à la fonction appelante. Sa syntaxe est :

return *expression_ou_variable* **;**

En théorie, l'instruction *return* peut figurer n'importe où dans le corps de la fonction, voire apparaître plusieurs fois. En pratique (règle de style), c'est **la dernière instruction** avant l'accolade fermante de la définition de fonction.

☺ Une fonction bien écrite contient **une seule** instruction *return*, placée sur la **dernière** ligne de la fonction. Sauf cas particulier (la gestion d'erreur), il faut éviter les instructions *return* en plusieurs exemplaires, dispersées dans le code au sein d'instructions comme *if-else*. Voir la fonction *calculer_racine* de l'Exemple 48.

Le compilateur rajoute automatiquement l'instruction *return* à la fin du texte d'une fonction s'il ne trouve pas cette instruction ; la valeur retournée est alors indéfinie.

9.5. Quelques exemples de fonctions simples

Exemple 49. Fonction sans paramètres qui ne renvoie rien

```
void afficher_accueil(void) ;      /* prototype */

void main(void)
      {
      afficher_accueil() ;          /* appel de la fonction (pas d'argument)*/
      … /* suite du main */
      }

void afficher_accueil(void)         /* définition de la fonction */
      {
      clrscr() ;         /* fonction non portable pour effacer l'écran chez Borland. En CVI : Cls */
      printf("Bonjour !") ;
      }
```

On voit apparaître dans cet exemple la définition d'une fonction appelée *afficher_accueil*. **L'en-tête** `void afficher_accueil(void)` précise le nom de la fonction, l'absence de paramètres (second *void*) et de valeur retournée (premier *void*). Après cet en-tête vient le **corps** de la fonction constitué d'un bloc entouré d'accolades { et }.

Remarquons que le programme principal apparaît sous la forme d'une fonction de nom *main* imposé. Par convention, l'exécution d'un programme en C commence toujours par cette fonction *main*.
Plusieurs en-têtes existent pour *main* : certains IDE (C++), qui fournissent un squelette pour *main*, proposent en général un en-tête qui commence par `int main(…)`, ce qui oblige à terminer par `return 0;` ou équivalent.

Exemple 50. Fonction sans paramètres qui renvoie une valeur

```
short int saisir_un_entier(void) ;      /* prototype de la fonction */

void main(void)
     {
     short int a, b ;

     a = saisir_un_entier( ) ;          /* appels de la fonction */
     b = saisir_un_entier( ) ;
     printf( "\n La somme de a et b est %hd", a+b ) ;
     }

short int saisir_un_entier(void)        /* définition de la fonction */
     {
     short int entier ;

     printf("\n Tapez un entier naturel entre -32768 et +32767 : ") ;
     scanf("%hd", &entier) ;

     return entier ;
     }
```

L'en-tête `short int saisir_un_entier(void)` précise cette fois que la fonction sans paramètres d'entrée *saisir_un_entier* fournit un résultat de type *short int*. L'instruction *return entier ;* indique quelle valeur est à renvoyer à la fonction appelante.

Attention : quel que soit l'endroit où est placée l'instruction *return*, son exécution termine toujours la fonction et provoque le retour à la fonction appelante.

Exemple 51. Fonction avec paramètres qui renvoie une valeur

```
double moyenne(double a, double b) ;   /* prototype de la fonction */

void main(void)
     {
     double mes1, mes2, resultat ;             /* variables LOCALES à main */

     printf(" Entrez deux reels (mesures) : ") ;
     scanf("%lf%lf", &mes1, &mes2) ;

     resultat = calculer_moyenne(mes1, mes2) ;
     printf( "\n La moyenne des mesures est %lf", resultat ) ;
     }

double calculer_moyenne(double a, double b)   /* définition de la fonction */
    {
    return (a+b)/2.0 ;
    }
```

La fonction appelante (ici, *main*) fournit la valeur des deux variables *mes1* et *mes2* dont elle veut la moyenne. *mes1* et *mes2* sont des **arguments d'appel**[7], alors que les variables *a* et *b* qui apparaissent dans l'en-tête de la définition de la fonction *calculer_moyenne* sont appelés paramètres formels ou **paramètres** tout court, par opposition aux arguments d'appel.

[7] Aussi appelés « paramètres effectifs »

Lors de l'appel de *calculer_moyenne*, les arguments *mes1* et *mes2* sont **copiés** dans les paramètres formels *a* et *b*. La fonction *calculer_moyenne* manipule ensuite exclusivement ses paramètres *a* et *b*, sans que les arguments *mes1* et *mes2* n'en soient affectés.

Les paramètres sont définis sur la ligne d'en-tête de fonction. Ils sont alloués au début de l'exécution de la fonction *calculer_moyenne* dans une zone de mémoire spéciale appelée la **pile**. Ils disparaissent dès que la fonction rend fin avec *return*.
Les arguments d'appel peuvent être des contenus de variables, mais aussi des **expressions** ou des **constantes** numériques, ou un mélange des trois :

```
resultat = calculer_moyenne ( n, p/3.+5.0 ) ;
resultat = calculer_moyenne ( 6.3, -23.1 ) ;
```

Les variable, expressions ou valeurs numériques *n*, p/3.+5.0, 6.3 et -23.1 sont alors les arguments d'appel (ou paramètres effectifs). Pour d'autres exemples d'appel, voir le chapitre « Utilisation de fonctions ».

Ce mode de transmission des paramètres est appelé **passage en paramètre par valeu**r. La fonction appelée n'utilise que des **copies** (paramètres, ici nommés *a* et *b*) des arguments d'appel, si ceux-ci sont des variables : il n'est pas possible de modifier dans la fonction appelée une variable de la fonction appelante.

Le nombre d'arguments d'appel fournis par la fonction appelante doit évidemment correspondre au nombre des paramètres dont la liste figure dans l'en-tête de la définition de la fonction. Cependant, le type des arguments peut différer : si c'est possible, le compilateur force la conversion des valeurs transmises pour qu'elles correspondent aux types figurant dans l'en-tête de la fonction.

Remarquons que les variables *mes1* et *mes2* sont définies à l'intérieur du bloc de la fonction *main*. Elles sont dites **variables locales** à *main* et sont inconnues pour toute autre fonction que *main*. Si la fonction *calculer_moyenne* décide d'appeler ses paramètres *mes1* et *mes2* au lieu de *a* et *b*, ce seront des variables différentes de celles de *main*.

9.6. Variables locales et variables globales

9.6.1 Les variables locales

En Langage C, toute variable définie à l'intérieur du corps d'une fonction est dite **locale** à cette fonction, c'est-à-dire qu'elle n'est connue que de cette fonction. Les autres fonctions ne peuvent jamais l'utiliser.

```
short int fonction1(void)
      {
      double x, y ;              /* x,y et n sont locales à fonction1 */
      short int n=12 ;
      ...
      }
void fonction2(short int n)      /* n est un paramètre (cas particulier de variable locale) */
      {
      double z=8.24 ;            /* z est une variable locale à fonction2 */
      ...
      }
void main(void)
      {
      short int i, j=2, n=30 ;   /* i, j et n sont locales à main */
      i = fonction1() ;
      fonction2(j) ;
      ...
      }
```

Ici, les deux variables locales *n* et le paramètre (formel) *n* des trois fonctions *fonction1*, *fonction2* et *main* n'ont aucun lien entre eux à part un nom identique ; il s'agit en fait de trois variables **différentes**, correspondant à trois emplacements mémoire distincts. A chaque exécution de l'appel d'une fonction, la variable locale ou le paramètre *n* sont redéfinis, et ils sont détruits à la fin de l'exécution de la fonction.

Les variables locales sont définies dans la zone particulière de la mémoire appelée la **pile**, tout comme les paramètres (formels) qui accueillent les valeurs transmises en argument lors d'un appel.

Les paramètres formels qui sont **créés dans l'en-tête** d'une fonction ont toutes les caractéristiques des variables locales : création à l'entrée dans la fonction, visibilité limitée à la fonction, destruction à la fin de l'exécution de la fonction. La seule différence entre paramètre formel et variable locale réside dans leur valeur initiale : le paramètre formel est toujours créé avec une valeur initiale (l'argument d'appel fourni par la fonction appelante), alors que l'initialisation de la variable locale est laissée à l'appréciation du programmeur. A ce détail près, **le terme "variable locale" peut parfaitement englober les paramètres qui apparaissent dans l'en-tête de fonction.**

La valeur d'une variable locale ne se retrouve pas d'une exécution à l'autre de la fonction. Si ce doit être le cas, il faut la définir comme « variable locale statique » avec le mot-clé *static* (voir le chapitre "Classes d'allocation mémoire", variable de bloc statique). Elle est alors définie en dehors de la pile et sa durée de vie devient permanente. Mais sa visibilité reste limitée à la fonction où elle est définie.

En résumé, retenons qu'une variable locale "normale" a une portée limitée à la fonction où elle est définie et que sa valeur n'est pas conservée après l'exécution de la fonction.

9.6.2 Les variables globales

☺ On évite au maximum d'utiliser des variables globales. On peut en général les remplacer, quand elles "paraissent" utiles, par le passage en paramètre par adresse (cf Chap. Pointeurs).

Les variables globales sont à utiliser de façon exceptionnelle : sauf exception particulière (indiquée ci-dessous), on peut toujours s'en passer.

A quoi reconnaît-on une variable globale ? Quelles sont ses caractéristiques ?

Une variable devient globale dès qu'elle est définie **en dehors de toute fonction**, en général au début du fichier. Elle est utilisable par toutes les fonctions du fichier. Elle existe pendant toute la durée du programme. Son emplacement mémoire est connu dès l'édition de liens (juste après la compilation).

Une variable globale est une variable "**publique**" utilisable par tout le monde. Cette apparente simplicité est trompeuse : le programme est beaucoup plus difficile à lire, on ignore tout du passé et de l'avenir de la variable si on n'épluche pas tout le programme (aucune traçabilité), et la programmation à plusieurs devient risquée, car des variables globales et locales peuvent porter par erreur le même nom[8]. C'est une source d'erreurs fréquentes qui sont difficile à trouver !

Les variables globales ne sont globales que dans la partie du fichier qui suit leur définition. Elles doivent être placées en tête du fichier pour la lisibilité. En compilation séparée, il est conseillé de **placer la définition des variables globales au sommet du fichier du *main*** ; elles sont alors simples à trouver.

Insistons : évitez les variables globales. Nous verrons dans le chapitre sur les pointeurs que le passage par adresse des paramètres permet d'éviter l'utilisation de variables globales dans la quasi-totalité des cas.

Dans quel cas je peux utiliser une variable globale ?

De façon général, dès que le passage en paramètre n'est pas possible. C'est le cas de toutes les fonctions qui ont un prototype figé : faute de pouvoir disposer de paramètres pour dialoguer avec elles, il faut utiliser des variables globales qui matérialisent des données ou des *flags* (drapeaux de signalisation). Exemples :

- en programmation Windows, pour échanger des données entre fonctions *Callback* ou *Threads*.
- pour dialoguer avec une routine d'interruption.

Exceptionnellement, dans un programme qui utilise tout le temps une « grosse » variable « centrale » (toujours un tableau ou une structure), on peut tolérer de rendre cette variable globale.

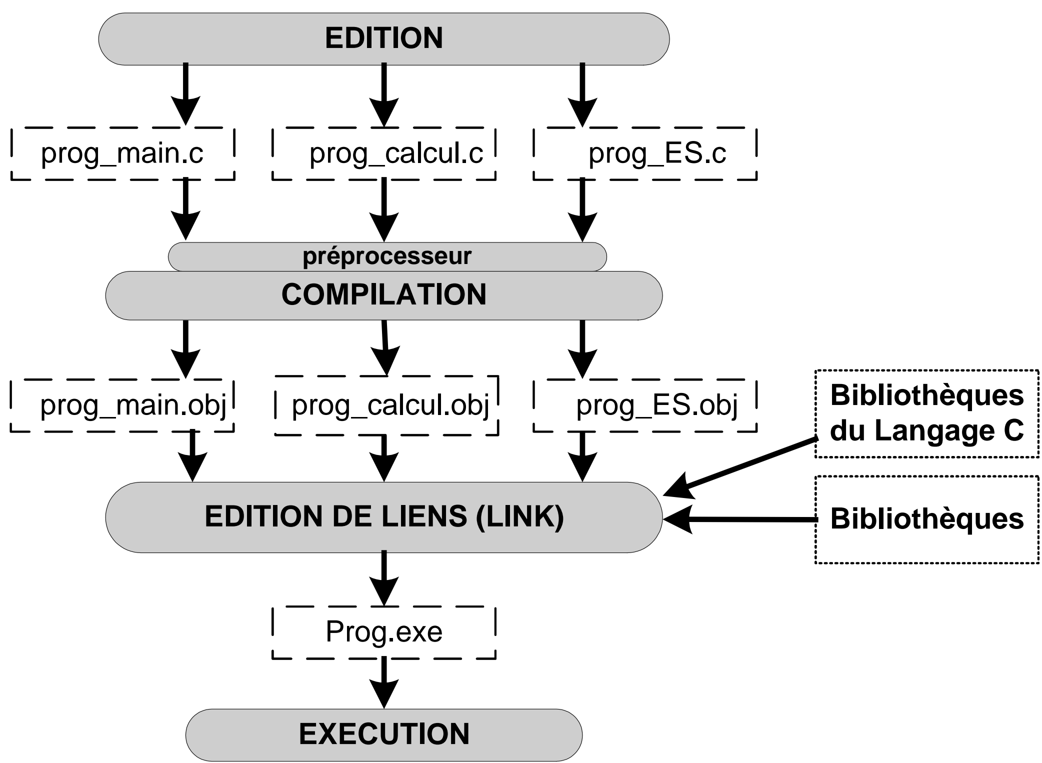
☺ Pour éviter les problèmes, donnez à vos variables globales des noms « compliqués », de préférence qui commencent par g. Exemples : gflag_dessin, ghandle_canal, gdata_a_emettre.

---

[8] C'est alors la variable locale qui est utilisée, mais cette situation est explosive !

10 - La compilation séparée (multi-fichiers)

10.1. Fabrication du programme : compilation (préprocesseur), édition de lien



Le Langage C offre la possibilité de répartir les différents éléments d'un programme dans **plusieurs** fichiers sources, qui portent l'extension ***.c*** ou *.cpp*, et parfois ***.h*** (*Header*). Ces fichiers sources sont écrits au cours de la phase d'édition à l'aide d'un éditeur de texte, le plus souvent celui de l'IDE[9].

Les fichiers sources sont compilés séparément pour donner autant de fichiers **objets**. Ce travail est réalisé par le compilateur C (*Compiler*).
Celui-ci est automatiquement précédé par l'exécution d'un programme appelé **préprocesseur**, dont le rôle est de traduire les **directives** figurant dans les fichiers sources. Ces directives sont des ordres destinés au préprocesseur, qui commencent toujours par **#** : les directives les plus fréquentes sont ***#define*** ou ***#include***. Le préprocesseur fournit au compilateur des fichiers qui sont encore écrits en Langage C.

A l'issue de la compilation de chaque fichier source, le texte C a été traduit en **langage machine** pour donner un fichier **objet** d'extension **.obj** (sur PC). Le programme comporte à présent autant de fichiers objets qu'il avait de fichiers sources. Mais il « manque encore des bouts » pour que l'exécutable soit complet.

[9] *Integrated Development Environment* = environnement de développement intégré

Les différents modules objets sont réunis en un seul module **exécutable** par l'**éditeur de liens** (***Linker***), à qui on fournit la liste des fichiers objets à assembler ; cette liste est contenu dans le **projet**. L'éditeur de liens complète les modules objets de l'utilisateur en allant chercher dans la bibliothèque standard du C ou les bibliothèques de l'IDE les blocs qui correspondent aux fonctions utilisées (*printf, scanf, sin, exp*, ...).

Le résultat de l'édition de liens est un programme exécutable, écrit en langage machine et autonome, qui est stocké sous la forme d'un fichier d'extension **.exe** (sur PC).

La bibliothèque standard du Langage C et les bibliothèques supplémentaires spécifiques de l'IDE fournissent deux types de fichiers :

- des fichiers **en-tête** tels que *stdio.h*, *math.h*, etc. Ce sont des fichiers sources qu'on peut consulter et qui contiennent des prototypes de fonctions (ex. *printf*), des macro-instructions (ex. *getchar*), des constantes symboliques (ex. *NULL*, *M_PI*)... Ces fichiers sont utilisés par le préprocesseur pour compléter les fichiers sources écrits par l'utilisateur : le préprocesseur remplace littéralement la directive `#include <fic.h>` par le texte contenu dans le fichier *fic.h*.
- des modules objets qui contiennent le code machine correspondant aux fonctions de la bibliothèque (*printf*, *scanf*, *log*, *sqrt*,etc.), dont les prototypes sont donnés dans les fichiers en-tête *.h*. Ces modules objets sont utilisés par l'éditeur de liens pour l'élaboration de l'exécutable.

10.2. Conséquences sur l'utilisation de fonctions

L'utilisation d'une fonction que le compilateur ne connaît pas (parce qu'elle est définie plus bas dans le même fichier source ou dans un autre fichier) doit obligatoirement être précédée de la **déclaration** de cette fonction :

- soit sous la forme d'un prototype tapé directement :

type nom_fonction(type par1, type par2, ...) ;

Les prototypes sont à regrouper **en début de fichier**, derrière les directives **#**.

- soit en insérant le fichier en-tête où se trouve le prototype voulu, avec la directive *#include* :

`#include "nom_fichier_entete.h"` (à placer au sommet du fichier utilisateur)

On peut préciser en commentaire le nom du fichier où est définie la fonction. Mais le compilateur et le *linker* n'en ont pas besoin : l'éditeur de liens utilisera la liste de fichiers du **projet** pour retrouver la fonction.

10.3. Conséquence sur les variables globales : la déclaration *extern*

Rappelons que l'usage des variables globales est soumis à de strictes limitations. Voir paragraphe 9.6.2.

Une variable globale est accessible par toutes les fonctions écrites en-dessous de sa définition dans le même fichier source. Mais une fonction écrite dans un autre fichier ne peut pas se servir de cette variable globale... C'est pourtant indispensable si on désire diviser le programme en plusieurs fichiers sources qui se partagent des variables globales.

Pour résoudre ce problème, le Langage C prévoit une déclaration qui précise que la variable globale est définie dans un autre fichier. C'est la déclaration ***extern***, dont la syntaxe est :

```
extern type nom_var ;     /* à placer au sommet du fichier utilisateur ou dans un fichier en-tête*/
```

Cette déclaration indique au compilateur qu'il ne s'agit pas d'une nouvelle variable et qu'il ne doit pas réserver de place en mémoire. La déclaration *extern* fournit le type de la variable et précise qu'elle est déjà définie ailleurs. L'éditeur de liens la trouvera à condition qu'elle ait été définie en variable globale dans un, et **un seul**, des fichiers à lier : celui où se trouve *main*, de préférence (règle de style).

Attention : il n'est pas question d'initialiser la variable globale dans cette déclaration *extern*.

10.4. Exemple de programmation multi-fichiers (sans fic en-tête)

Attention : dans cet exemple, il n'est pas fait usage de fichiers en-tête créés par le programmeur. Ceux-ci sont très utiles en compilation séparée : on peut donc directement passer à l'exemple 2 si on veut utiliser un fichier en-tête.

Cet exemple ne comporte pas non plus de variable globales, conformément à une bonne philosophie de programmation.

Exemple 52. Compilation séparée sans fichier en-tête

Fichier 1 :

```
#include <stdio.h>                /* contient le prototype de printf */
#define VALEUR_CODE 'H'           /* création d'une constante symbolique */
short int TesterSecu(char code);  /* prototype de TesterSecu (définie dans fichier2) */
void main(void)
      {
      char code = VALEUR_CODE ;
      if (TesterSecu(code)==0)
            printf("Vous n'êtes pas autorisé à poursuivre!");
      else  printf("Vous pouvez continuer.") ;
      }
```

Fichier 2 :

```
#include <stdio.h>  /* contient le prototype de printf */
#include <xxx.h>    /* contient le prototype de GetKey (non portable hors CVI !)*/
short int TesterSecu(char code)  /* définition de la fonction TesterSecu */
      {
      printf("Entrez le code de sécurite: ");
      return( GetKey()==code );     /* attention : GetKey n'est pas portable hors CVI ! */
      }
```

La fonction *TesterSecu* retourne le résultat du test `GetKey()==Code`, qui vaut 0 ou 1. Ce test compare le caractère entré par l'utilisateur avec la valeur du code transmise en paramètre.

☞ *GetKey* est une fonction non portable de CVI (elle s'appelle *getch* chez Borland). Elle permet la saisie d'un caractère au clavier, sans attendre la frappe de la touche *Entrée*. On peut la remplacer par **getchar**, qui fait partie de la bibliothèque standard, donc est portable (mais qui nécessite la frappe de la touche *Entrée*).

10.5. Ecriture d'un fichier en-tête – Exemple complet

Reprenons l'exemple précédent pour illustrer l'utilisation du fichier en-tête *global.h* créé par le programmeur.

Exemple 53. Compilation séparée avec fichier en-tête

Fichier en-tête global.h

```
/* Ce fichier en-tête contient tout ce qu'on veut rendre "public" : les constantes symboliques, les proto-
types de fonctions, les déclarations de variables globales. Mais aussi, le cas échéant, des modèles de struc-
tures et des créations de type par typedef. Voir exemple plus complet à la fin du chapitre Structures */
/* Création de constantes symboliques : */
#define VALEUR_CODE 'H'
/* Création de types (structurés ou autres) : ici, aucun. */
/* Prototypes des fonctions (ici TesterSecu définie dans fichier2) : */
short int TesterSecu(char code);
/* On ajoute ici les déclarations des variables globales si elles existent */
```

Fichier 1 :

```
#include <stdio.h>                  /* contient le prototype de printf */
#include "global.h"                 /* fichier en-tête décrivant constantes, prototypes... */
void main(void)
      {
      char code = VALEUR_CODE ;
      if (TesterSecu(code)==0)
            printf("Vous n'êtes pas autorisé à poursuivre!");
      else  printf("Vous pouvez continuer.") ;
      }
```

Fichier 2 :

```
#include <stdio.h>                  /* contient le prototype de printf */
#include <xxx.h>                    /* contient le prototype de GetKey (non portable hors CVI !)*/
#include "global.h"                 /* fichier en-tête décrivant constantes, prototypes... */
 short int TesterSecu(char code)          /* définition de la fonction TesterSecu */
      {
      printf("Entrez le code de sécurite: ");
      return( GetKey()==code );           /* attention : GetKey n'est pas portable hors CVI ! */
      /* résultat de la comparaison par == :  0 ("faux") si la valeur saisie (renvoyée par GetKey) diffère de code,
                                              1 ("vrai") si les deux valeurs sont identiques. */
      }
```

Cet exemple appelle certaines précisions.

1. La directive **#include** possède deux syntaxes :
    - la syntaxe **`#include <...>`** incite le préprocesseur à chercher le fichier à inclure dans un dossier défini par l'environnement de travail (par exemple, *C:\...\CVI\include*). C'est la syntaxe employée pour inclure les fichiers en-tête de la bibliothèque standard.
    - la syntaxe **`#include "..."`** (guillemets au lieu de < >) indique au préprocesseur que le fichier à inclure se trouve dans le dossier de travail courant. C'est la syntaxe généralement utilisée avec les fichiers en-tête créés par le programmeur. On peut aussi préciser le chemin complet du fichier (déconseillé !).

2. Si dans un même fichier source se trouvent simultanément la définition et la déclaration d'une variable ou d'une fonction, le compilateur ignore la déclaration pour ne tenir compte que de la définition.

3. Un fichier en-tête d'extension *.h* peut contenir (dans l'ordre) :

- des constantes symboliques définies par *#define*, ce qui évite de répéter le *#define* dans tous les fichiers où la constante symbolique est employée. Plus généralement, on pourra y définir des macros-instructions à l'aide de *#define* ;
- des définitions de types synonymes par *typedef* ;
- des modèles de structures (voir chapitre sur les structures) ;
- les déclarations des fonctions (prototypes) et des variables globales.

Par contre, un fichier en-tête ne peut contenir aucune définition, donc **aucune initialisation**, ni **aucun code exécutable**.

Attention : les prototypes de fonction utilisent en général les types créés par le programmeur, qui eux-même peuvent utiliser les constantes symboliques. L'ordre d'écriture d'un fichier en-tête est donc le suivant : **constantes symboliques,** puis **créations de type** (modèles de structures en particulier), et enfin les **prototypes de fonctions et les déclarations de variables globales**.

11 - Les tableaux

Le tableau est une structure de données très utilisée en informatique. Il permet de ranger en mémoire **une liste de valeurs**, puis de manipuler facilement ces valeurs à l'aide de l'instruction *for*.

Un tableau est un ensemble fini **d'éléments <u>de même type</u>** placés consécutivement en mémoire et repérés par un même identificateur (le nom du tableau).

Un numéro d'ordre appelé **indice** permet de repérer chacun des éléments du tableau.

Le tableau « à une dimension » matérialise en informatique la notion de liste. A deux dimensions, le tableau matérialise la notion de matrice, bien utile en mathématique.

11.1. Premier exemple de programme avec un tableau

Le tableau devient indispensable quand on désire stocker et manipuler **un ensemble** de variables de **même type** dont le nombre exclut la possibilité de les définir et de les traiter individuellement (cas fréquent en informatique !).

Exemple 54. Tableau à une dimension utilisé pour le stockage de notes :

```
#define NB_ELEVES 10                    /* nombre d'élèves de la classe */

void main(void)
    {
    short int i ;
    double moy = 0.;
    static double note[NB_ELEVES] = { 12, 5, 20, 10, 15, 18, 7, 13, 16, 8 };
        /*définition d'un tableau initialisé de NB_ELEVES éléments réels (l'intialisation sert pour le test) */

    /* ------- saisie (seulement quand le programme fonctionne : A EVITER EN PHASE DE TEST !! ------*/
        for (i=0 ; i<NB_ELEVES ; i++)
            {
            printf("\n Entrez la note %hd :", i+1);
            scanf("%lf", &note[i]);
            }

    /* ------------- calculs (partie à écrire et à tester avant les saisies/affichages) -------------- */
    for (i=0 ; i<NB_ELEVES ; i++)
        {
        moy = moy + note[i];        /* on calcule la somme des éléments du tableau */
        }
    moy = moy/NB_ELEVES ;


    /* ------ affichage (à éviter en phase de test : utiliser le debugger pour afficher le contenu des variables) ----*/
    printf("\n\n La moyenne de la classe est %5.2lf", moy );
    printf("\n\n La liste des notes sous la moyenne est :");
    for (i=0 ; i<NB_ELEVES ; i++)
        if (note[i]<moy)   printf("\n\t eleve %hd : %lf", i+1, note[i]);
    }
```

La définition `static double note[NB_ELEVES]` réserve de la place en mémoire pour NB_ELEVES éléments réels (type *double*). NB_ELEVES est une constante définie par la directive *#define* qui vaut ici 10 et peut être **facilement modifiée**, en particulier pour faciliter le test.

L'élément d'indice *i* du tableau *note* est désigné par la notation **note[i]**. En Langage C, **tous les tableaux commencent par l'indice 0**. Les indices utilisables vont donc **de 0 à (NB_ELEVES-1)**.

La notation ***¬e[i]*** désigne **l'adresse** de l'élément d'indice *i* (utilisée par exemple dans *scanf*).

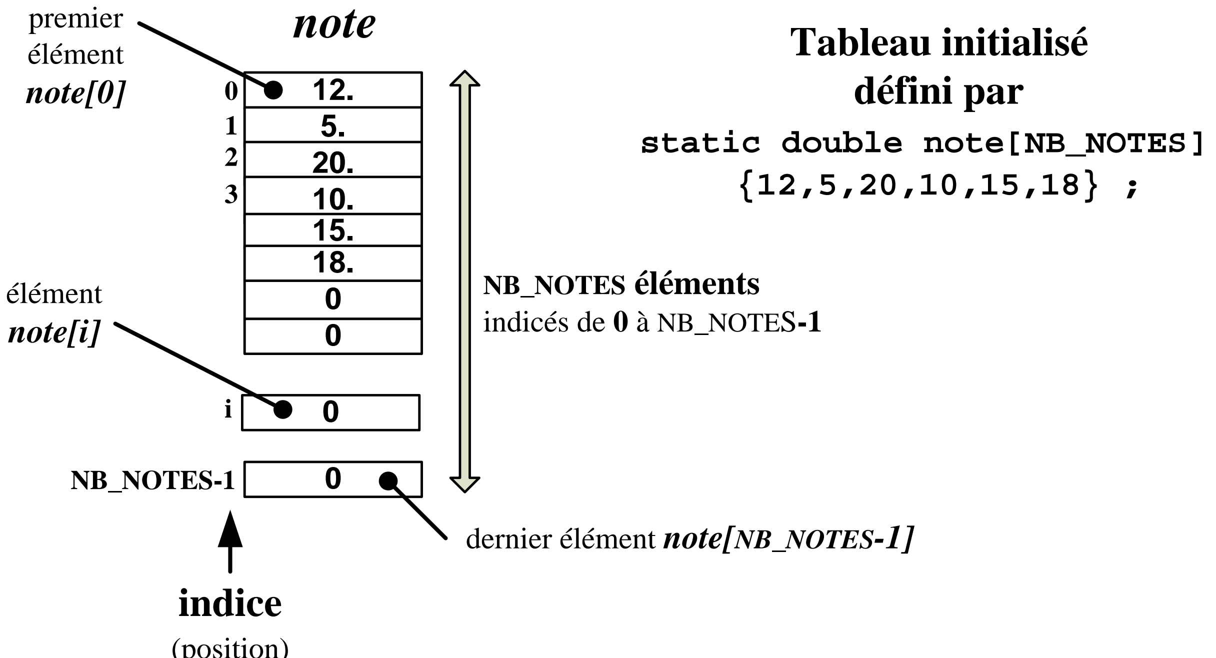


Figure 11 --1 : Un tableau en mémoire

11.2. Définition d'un tableau à une dimension

La syntaxe d'une définition de tableau est la suivante :

static *type_d_un_element nom_tableau*[*NB_ELEMENTS*] **;**

Cette définition réserve NB_ELEMENTS emplacements consécutifs en mémoire permettant d'accueillir NB_ELEMENTS valeurs de type *type_d_un_element*.

Le mot-clé optionnel ***static*** demande au compilateur d'allouer le tableau dans la mémoire statique, au lieu de la pile qui est utilisée par défaut pour une variable locale. Cela évite de faire « exploser la pile » et a pour effet secondaire d'initialiser le tableau avec des zéros. Pour préserver la pile, l'utilisation de *static* est conseillé avec toutes les « grosses » variables, comme les tableaux ou les structures.

Sans *static* (déconseillé), le tableau est défini en pile et ses éléments n'ont pas de valeur par défaut.

NB_ELEMENTS doit être une **constante** (par exemple 100) ou une **constante symbolique** définie par *#define*. Mais NB_ELEMENTS ne peut pas être une variable ou une constante définie avec le mot-clé *const*.

☺ Utilisez une constante symbolique (définie par *#define*) pour indiquer le nombre d'éléments d'un tableau : le programme devient plus lisible et plus évolutif.

Par exemple, la définition **`short int tab[5]`** réserve de la place mémoire pour 5 entiers courts :

- tab[**0**] est le **premier** élément du tableau *tab* ; on dit que c'est l'**élément d'indice 0**.
- **tab[i]** est le (i+1)ème élément du tableau (c'est l'**élément d'indice i**) ;
- tab[4] est le 5ème et **dernier** élément du tableau (attention !).
- les valeurs des éléments sont indéfinies (quelconques).

Exemple 55. Définition de tableaux (*static* est obligatoire pour les plus gros)

```
#define NB_PTS 100                    /* pour dimensionner un tableau */
static double x[NB_PTS], y[NB_PTS];
static short int entier[25];       /* des constantes symboliques seraient souhaitables ici ! */
static char Ligne[80] ;
static long gros_tableau[10*NB_PTS];   /* static obligatoire pour sortir le tableau de la pile*/
```

Insistons : le mot-clé *static* dans une définition de tableau oblige le compilateur à définir le tableau (qu'on souhaite voir local à une fonction) dans une autre zone mémoire que la pile. La pile, utilisée pour les variables locales, a une taille limitée qui rend dangereuse l'allocation de variables importantes. Une solution consiste à utiliser *static*, qui ne modifie pas la visibilité de la variable (mais rend sa durée de vie permanente et l'initialise à 0).

11.3. Accès aux éléments d'un tableau : tab[i]

Il faut retenir que les tableaux en Langage C **commencent toujours avec l'indice 0** : *tab**[0]*** est le premier élément d'un tableau défini par `double tab[10]` et *tab[9]* est son dernier élément.

Les éléments se manipulent comme des variables normales : ***tab[i]*** permet d'accéder à l'**élément d'indice *i*** du tableau *tab*, ***&tab[i]*** représente son adresse. L'indice *i* est forcément un entier.

Exemple 56. Manipulations variées sur les éléments de tableau

```
#define N  5
for (i=0 ; i<N ; i++)      tab1[i] = 2*tab2[i] ;
printf("%c", text[i]) ;
tab[i]++ ;                 /* incrémente l'élément tab[i] */
text[3] = 'a' ;
for (i=N ; i>0 ; i--)      tab[i] = 2*i+1 ;
scanf("%lf", &tab_reel[i]) ;
```

Attention : il n'y a **aucun contrôle de dépassement des indices** de la part du compilateur. Un tel dépassement peut conduire à l'écrasement de données en mémoire. Vérifiez toujours les limites de vos boucles et n'oubliez pas que les indices commencent à 0, donc finissent à **N-1**…

La boucle *for* est très utilisée pour « balayer » tous les éléments d'un tableau. Pour éviter des dépassements aux effets imprévisibles, la syntaxe employée pour la boucle *for* doit être

```
for ( i=0 ; i < nb_elts ; i++)
```

et non `for (i=0 ; i <= nb_elts-1 ; i++)`

11.4. Initialisation d'un tableau, totale ou partielle

Comme toute variable, un tableau peut être initialisé lors de sa définition. Par exemple :

static short int table[5] = { 100, 101, 102, 103, 104 } ;

permet de réserver et d'initialiser un tableau de 5 entiers. Si la taille du tableau (ici 5) n'est pas précisée, le compilateur la calcule lui-même en comptant le nombre d'éléments entre accolades (déconseillé) :

```
static short int table[ ] = {100, 101, 102, 103, 104 } ;
```

On peut initialiser **une partie** du tableau seulement (forcément les premiers éléments). **Les éléments restants sont alors automatiquement mis à 0 par le compilateur.** Voir figure page précédente.

Exemple 57. Initialisation de tableaux

```
#define NB_MESUR 100          /* pour dimensionner les tableaux */
#define NB_NOTES 20

static double mesure[NB_MESUR] = {1.0,-2.6,2e-3};      /* 3 premiers éléments seuls non nuls*/
static short note[NB_NOTES] = { 20, 10, 5, 15, 18 }; /* 5 premiers éléments non nuls*/
static char Lettre[4] = {'A', 'B', 'C', 'D' } ;
```

11.5. Copie d'un tableau : *for* ou *memmove*

Le nom d'un tableau représente **son adresse** (celle du premier élément). Il n'est donc pas possible de réaliser des affectations ou des copies de tableaux par l'écriture :

`tab2 = tab1;` écriture interdite, car sans signification !

C'est pourquoi on réalise souvent l'affectation ou la copie élément par élément :

for (i=0 ; i<N ; i++) tab_dest[i] = tab_srce[i] ;

Une solution plus rapide et élégante utilise la fonction de recopie mémoire *memmove* :

memmove (tab_dest, tab_srce, sizeof(tab_srce)) ;

Le mode d'emploi de *memmove*, qui peut être utilisée pour copier n'importe quel bloc d'octets (et pas seulement un tableau), est le suivant :

memmove (adresse_destination, adresse_source, nombre_d'octet_a_copier) ;

L'utilisation de l'opérateur ***sizeof*** pour calculer le nombre d'octets à copier est en général indispensable :

`sizeof(tab)` fournit la taille en octets d'un tableau
`n*sizeof(double)` fournit le nombre d'octets occupés par *n* cases mémoire de type *double*.

Exemple 58. *Copie d'un petit tableau dans un gros (memmove, sizeof)*

```
/* on veut ici, en vue d'un test, copier les éléments d'un petit tableau dans les premiers éléments d'un gros tableau */
#define N_PT        5
#define N_GROS      100

static long tab_test[N_PT]= { 4,-5,10,-12,9 };   /* petit tableau de test entièrement initialisé */
static long gros_tab[N_GROS];                     /* gros tableau à initialiser */

for (i=0; i< N_GROS; i++)  gros_tab[i] = 0;       /* inutile si gros_tab est static */

memmove(gros_tab, tab_test, sizeof(tab_test) );
/* recopie les N_PT éléments de tab au début de gros_tab. Les autres éléments (hors les N_PT premiers) seront à 0 */
```

11.6. Transmission d'un tableau en paramètre d'une fonction

Le passage d'un tableau comme paramètre d'une fonction est impossible en tant que valeur : la recopie du tableau prendrait trop de temps et de place. On passe donc à la fonction **l'adresse du tableau**, ce qui permettra à la fonction d'effectuer des lectures et des écritures DIRECTEMENT DANS LE TABLEAU.

Important : **l'identificateur du tableau (son nom) représente l'adresse du début du tableau**. La notation **tab** équivaut à *&tab[0].*

Pour transmettre l'adresse du tableau à une fonction, il suffit de lui donner **le nom du tableau** :

```
nom_fct( nom_tableau, ... ) ;     /* appel de fonction avec argument tableau */
```

Exemple 59. Passage en paramètre d'un tableau (rempli aléatoirement)

```
#include <stdlib.h>                /* pour rand et srand */
#include <time.h>                  /* pour time_t et time */
#define NB_MAX_ELT 10              /* nombre d'éléments du tableau */

 /* -------------------- prototypes des fonctions -------------------- */
 void remplir_aleatoirt_tableau( double tab[], short int nb_elt );
 void afficher_le_tableau(double tab[], short int nb_elt);
 short chercher_1er_element_sup_a_1000(double tab[], short int nb_elt);
//-------------------------------------------------------------------------
 void main(void)
      {
      static double reel[NB_MAX_ELT];      /* c'est le seul endroit où apparaît le mot-clé static */
      short int i_sup1000 ;
      time_t temps ;
      /* Initialisation du générateur aléatoire pour obtenir  une séquence aléatoire différente à chaque lancement du prog : */
      srand( (unsigned int)time(&temps) );
      remplir_aleatoirt_tableau( reel, NB_MAX_ELT );   /* appel avec argument tableau */
      i_sup1000 = chercher_1er_element_sup_a_1000( reel, NB_MAX_ELT );
      afficher_le_tableau( reel, NB_MAX_ELT );
      }
//-------------------------------------------------------------------------
  void remplir_aleatoirt_tableau( double tab[], short int nb_elt )
             /* en paramètres : le tableau et le nombre d'éléments à utiliser (pas forcément tout le tableau…) */
      {
      short int i ;
      for (i=0 ; i<nb_elt ; i++)
             tab[i] = rand() % 2000 ;      /* pour obtenir des valeurs comprises entre 0 et 1999 */
      }
//-------------------------------------------------------------------------
short chercher_1er_element_sup_a_1000( double tab[], short int nb_elt )
      {
      short int i ;
      for (i=0 ; i<nb_elt && tab[i]<1000. ; i++) ;     /* aucun traitement, c'est normal ! */
         /* on sort de la boucle for quand on trouve un élément qui convient, ou si on atteint la fin du tableau */
      if (i==nb_elt)   i=-1;              /* si aucune valeur supérieure à 1000 n'a été trouvée */
      return i ;
      }
//-------------------------------------------------------------------------
 void afficher_le_tableau( double tab[], short int nb_elt )
      {
      short int i ;
      printf("Le tableau possède %hd éléments :", nb_elt) ;
      for (i=0 ; i<nb_elt ; i++)
             printf("\n élément %hd : %lf", i, tab[i]) ;
      }
```

☞ La fonction *rand* fournit une valeur aléatoire entre 0 et RAND_MAX. Attention : la séquence pseudo-aléatoire est toujours la même si on n'utilise pas la fonction d'initialisation pseudo-aléatoire ***srand*** (ici, au début de *main*) avec un argument d'appel (*seed*) qui conditionne le démarrage de la séquence (ici, l'horloge du PC obtenue avec *time*).

```
void srand (unsigned int Seed);          int rand (void);
```

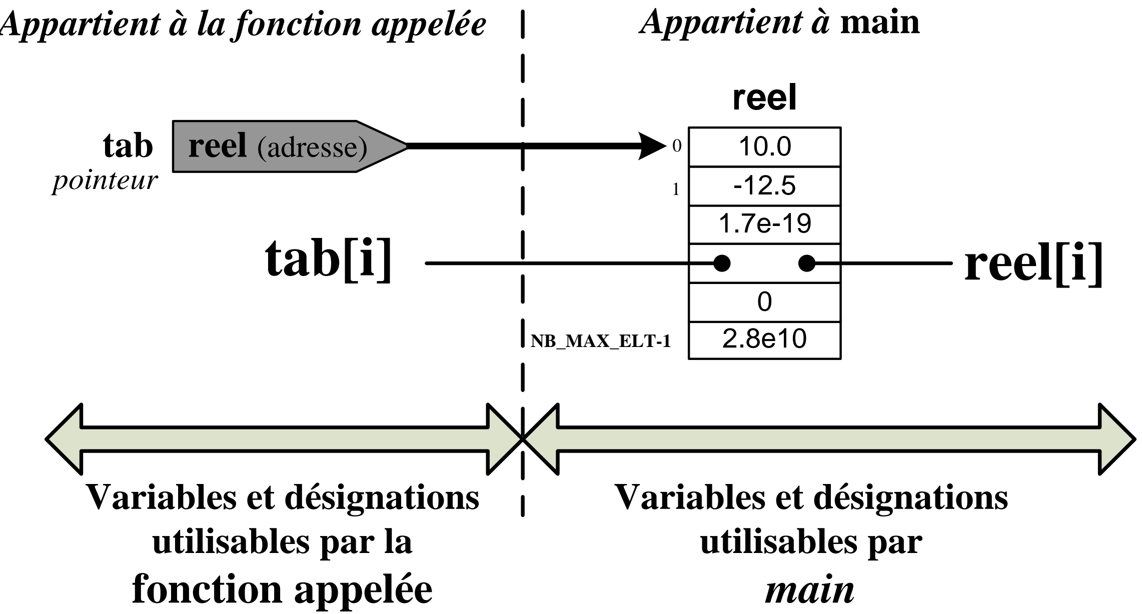


Figure 11 --2 : Un tableau en paramètre

Notez le prototype, très fréquent, d'une fonction qui doit manipuler un tableau fourni en argument :

void traiter_tableau(double tab[ ], short int nb_elt) ;

Les fonctions n'utilisent pas la constante NB_MAX_ELT, car la taille du tableau leur est fournie en argument d'appel avec *nb_elt*. Faire figurer le nombre d'éléments à traiter dans la liste des paramètres n'est pas obligatoire, mais fortement conseillé pour rendre les fonctions « plus universelles ». On peut ainsi traiter des tableaux de tailles différentes avec la même fonction, ou traiter une partie du tableau seulement.

L'en-tête de la définition et le prototype des fonctions ne précisent pas la taille du tableau : le compilateur n'a besoin que de l'adresse de début du tableau et de l'indice d'un élément pour calculer l'adresse de cet élément (vrai seulement pour les tableaux à une dimension). Cela permet d'écrire des fonctions qui peuvent manipuler des tableaux de tailles différentes, à condition de leur transmettre la taille du tableau.

Par exemple, la fonction *afficher_le_tableau* permet facilement d'afficher une sous-partie d'un tableau. Il suffit de lui transmettre l'adresse du premier élément à afficher (au lieu de l'adresse du tableau, qui est par définition l'adresse du premier élément du tableau), **ainsi que le nombre d'éléments à traiter**. Par exemple, l'affichage des trois éléments d'indices 5, 6 et 7 d'un tableau *reel* se fait avec l'appel suivant :

afficher_le_tableau(&reel[5], 3); */* 3 éléments affichés à partir de l'élément d'indice 5*/*

11.7. Tableau multidimensionnel (matrice, …)

11.7.1 Définition d'un tableau à plusieurs dimensions

Il suffit de faire figurer autant d'indices entre crochets [ ] qu'on désire de dimensions.

Exemple 60. Définition de tableaux multidimensionnels

```
#define NB_LIG 100          /* exemples de constantes pour dimensionner les tableaux */
#define NB_COL 10

static short int mat[NB_LIG][NB_COL];       /* la matrice est initialisée à 0 grâce à static */
static double reel[5][3] ;                  /* 5 lignes, 3 colonnes */
static char Ecran[25][80] ;                 /* 25 lignes, 80 colonnes */
static float courbe[10][10][10] ;           /*tableau à 3 dimensions */
```

Dans la définition d'un tableau à deux dimensions, la première paire de crochets contient le **nombre de lignes**, la deuxième paire de crochets précise le **nombre de colonnes**.
Par exemple, le tableau *Ecran* (défini par `char Ecran[25][80]`) est formé de 25 lignes constituées chacune de 80 caractères. Cette définition réserve une zone mémoire pour 25*80 caractères.

Un tableau à deux dimensions équivaut à une **matrice** :

- le premier indice représente le numéro de ligne ;
- le second indice le numéro de colonne.

La notation **a[i][j]** remplace alors la notation habituelle des matrices a_{ij}. Seule différence : les indices des tableaux du Langage C commencent toujours **avec la valeur 0**. Mais on n'est pas obligé d'utiliser l'élément d'indice 0 ! Il faut alors surdimensionner le tableau, en ajoutant un élément dans chaque sens :

static short int mat[NB_LIG+1][NB_COL+1]; /* pour ne pas utiliser les indices 0 */

Les tableaux à 3 ou 4 dimensions sont beaucoup plus rares, mais le principe reste le même.

11.7.2 Initialisation, stockage en mémoire

L'initialisation d'un tableau à deux dimensions s'effectue en mettant en évidence les deux « sous-tableaux » qui représentent les lignes :

static short int mat[3][4] = { { 0,1,2,3 }, { 4,5,6,7 }, { 8,9,10,11 } } ;

ce qui équivaut à l'écriture plus lisible :

Exemple 61. Initialisation d'une matrice

```
static short int mat[2][4] =  {
                              { 0, 1, 2, 3 },
                              { 4, 5, 6, 7 }
                              } ;
```

Cette initialisation ligne par ligne correspond au stockage en mémoire **linéaire** suivant : 0, 1, 2, …, 6, 7

En Langage C, les éléments d'un tableau sont rangés dans l'ordre obtenu en faisant varier le **dernier indice** en premier (celui de la colonne). On obtient donc la succession en mémoire suivante :

mat[0][0]
mat[0][1]   mat[**ligne**][**colonne**]
mat[0][2]
mat[0][3]
mat[1][0]
mat[1][1]
mat[1][2]
mat[1][3]

Cette disposition mémoire correspond à la disposition matricielle habituelle, sauf que la notion de « retour à la ligne » n'existe pas en mémoire :

mat[0][0]   mat[0][1]   mat[0][2]   mat[0][3]   */* ligne 0 */*
mat[1][0]   mat[1][1]   mat[1][2]   mat[1][3]   */* ligne 1 */*

11.7.3 **Accès à un élément** : *mat[lig][col]*

Un élément du tableau *mat* à deux dimensions est manipulé à l'aide de l'écriture

mat[lig][col]

où *lig* et *col* sont des indices qui commencent à 0.
lig (1er indice) représente la ligne et *col* (2ème indice) la colonne de l'élément.

☺ *mat[lig][col]* est une notation claire que vous devez utiliser au lieu de *mat[i][j]*, trop répandu. Dès qu'on atteint deux dimensions, **les indices doivent avoir des noms évocateurs** : *lig, col, i_eleve, i_point, i_sommet, i_mesure…*

💣 Attention : comme pour le tableau à une dimension, il n'y a aucun contrôle de dépassement des indices. Ne pas oublier que tous les indices commencent à zéro : le dernier élément d'un tableau défini par *short int mat[5][3]* est *mat[4][2]*.

☞ On peut ne pas utiliser les indices 0 à condition de surdimensionner de 1 chaque dimension concernée.

Exemple 62. Un programme avec un tableau à deux dimensions

```
#define NBLIG 3             /* nombre de lignes de la matrice */
#define NBCOL 4             /* nombre de colonnes */

void main(void)
{
short int lig,col ;
static short int mat[NBLIG][NBCOL] ;

/* Saisie de la matrice : */
for (lig=0 ; lig<NBLIG ; lig++)
      for (col=0 ; col<NBCOL ; col++)
            {
            printf("\n Entrez l'élément (%hd,%hd) : ", lig+1,col+1) ;
            scanf("%hd", &mat[lig][col]) ;
            }

/* Affichage de la matrice : */
printf("\n\n La matrice saisie est :\n") ;
for (lig=0 ; lig<NBLIG ; lig++)
      {
      for (col=0 ; col<NBCOL ; col++)
            printf("\t %hd ", mat[lig][col]) ;
      printf("\n") ;
      }
}
```

Si **mat[lig][col]** représente l'élément de ligne *lig* et de colonne *col*, la notation **mat[lig]** désigne **l'adresse <u>de la ligne</u> *lig*** c'est-à-dire l'adresse du premier élément de la ligne *lig* :

mat[lig] = & mat[lig][0].

De plus, comme dans le cas monodimensionnel, le nom *mat* du tableau représente l'adresse du tableau, c'est-à-dire l'adresse de son premier élément : les valeurs numériques de *mat* et de *&mat[0][0]* sont égales, bien que leurs types soient différents (*mat* est un pointeur « constant » sur un tableau, alors que *&mat[0][0]* est un pointeur « constant » sur un élément).

11.7.4 Transmission d'un tableau multidimensionnel en paramètre

La transmission d'un tableau à plusieurs dimensions en paramètre d'une fonction s'effectue comme pour les tableaux à une dimension, à une exception près. La seule différence est que **seule la dimension la plus à gauche peut être omise dans l'en-tête ou le prototype de la fonction**.

Une explication peut être la suivante : pour le compilateur, un tableau multidimentionnel est un tableau unidimentionnel de variables, elles-mêmes de type tableau ; c'est ce premier tableau qui est passé en paramètre et le type des variables qui le composent doit être connu.

☞ Pour les tableaux à deux dimensions et plus, le programmeur non confirmé a intérêt à écrire **toutes** les dimensions dans le prototype et l'en-tête, sans se poser de questions.

Exemple 63. Transmission en paramètre d'un tableau à deux dimensions

```
#define NBLIG 5
#define NBCOL 3
/* prototype de fonction avec tableau 2D en paramètre : */
  void traiter (short int mat[NBLIG][NBCOL] );
ou
  void traiter (short int mat[ ][NBCOL] );
        /* ATTENTION : les CROCHETS VIDES sont possibles seulement en 1ère position !! */

//-----------------------------------------------------------------------
  void main(void)
      {
      static short int note[NBLIG][NBCOL] ;     /* définition du tableau 2D */

      traiter(note);          /* appel de fonction avec un argument tableau 2D */

      }
//-----------------------------------------------------------------------
/* fonction avec tableau 2D en paramètre : */
  void traiter (short int mat[NBLIG][NBCOL] )
ou
  void traiter (short int mat[ ][NBCOL])    /*ATTENTION AUX CROCHETS NON VIDES!!*/

      {
      short int lig,col ;

      for (lig=0 ; lig<NBLIG ; lig++)
             {
             for (col=0 ; col<NBCOL ; col++)
                    mat[lig][col] = …
             }
      ...
      }
```

12 - Les chaînes de caractères

Une chaîne de caractères est utilisée en informatique pour **stocker du texte** (mot, phrase…). Outre la manipulation de données textuelles, elle est très utilisée pour effectuer des entrées/sorties : affichage à l'écran ou saisie au clavier, lecture ou écriture dans un fichier texte.

Une chaîne de caractères (*string*) est une suite de caractères alphanumériques (***char***) terminée par le **caractère nul** (c'est-à-dire de code ASCII 0). Contrairement à d'autres langages, il n'existe pas en Langage C de type chaîne (*string*) : une chaîne de caractères est mémorisée dans un **tableau de caractères à une dimension**.

Exemple 64. Manipulation de chaînes :

```
void main(void)
        {
        char nom[20], prenom[20], ligne[80];     /* définition de 3 chaînes */
        printf( "Entrez votre nom et votre prenom : ") ;
        scanf("%s%s", nom, prenom);       /* saisie de 2 chaînes (mots) au clavier */
        /* fabrication d'une nouvelle chaîne en mettant bout à bout  prénom et nom : */
        sprintf( ligne, "%s %s", prenom, nom ) ;
        printf("\n Bonjour, %s !", ligne );     /* affichage */
        printf( "\n Votre nom a %d caracteres", strlen(nom) );
        printf( "Entrez votre message : ") ;
        gets( ligne);                  /* saisie d'une phrase au clavier */
        puts( ligne);                  /* affichage à l'écran */
        }
```

12.1. Définition et initialisation d'une chaîne

On veut créer une chaîne de caractères pour stocker un message prédéfini qui ne changera pas (par exemple : *Bonjour !*). Les définitions (presque) équivalentes :

char texte[9] = "bonjour!" ; /* 9 éléments sont réservés pour le tableau de *char* */

permettent de créer un tableau de 9 éléments *char*, dans lequel sont stockés les 8 caractères 'b', 'o', 'n', 'j', 'o', 'u', 'r', '!' suivis par le "**caractère nul**" **0** (parfois noté '\0', c'est-à-dire dont le code ASCII est 0).
L'élément *texte[0]* contient le caractère 'b', *texte[7]* contient '!' et le dernier élément *texte[8]* contient le **caractère nul 0**, **ajouté par le compilateur**. Voir dessin page suivante.

L'écriture **entre guillemets** de la valeur de la chaîne "bonjour!" est une notation qui permet de simplifier l'initialisation classique des tableaux :

```
char texte[ ] = { 'b','o','n','j','o','u','r','!', 0 };     /* juste, mais à éviter !! */
```

La taille de la chaîne est en général supérieure à la taille de la valeur initiale, pour permettre de modifier celle-ci par la suite. Par exemple :

char texte[50] = "bonjour!" ; /* 50 élements réservés, 9 utilisés pour l'instant */

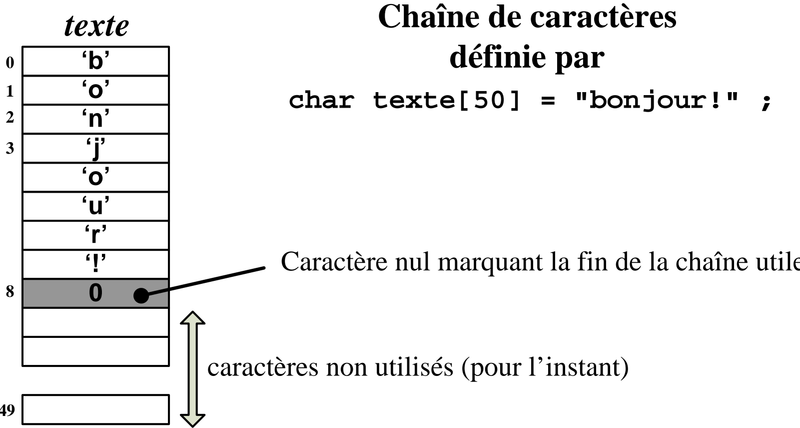


Figure 12 --3 : Allocation mémoire d'une chaîne de caractères

Les règles pour la définition et l'initialisation des chaînes de caractères sont les suivantes :

- la « constante chaîne de caractères », utilisée pour initialiser le tableau de *char*, se représente entre **guillemets " "**. Ne pas confondre avec les apostrophes **' '** qui entourent un caractère ;
- toute chaîne de caractères est terminée par le **caractère nul 0**. Celui-ci est placé automatiquement **par le compilateur** . Au-delà du caractère nul se trouvent les éléments non utilisés du tableau de *char* ;
- le compilateur peut calculer lui-même la taille de la chaîne (nombre de caractères+1) si celle-ci n'est pas précisée entre crochets. Mais il faut réserver cette facilité aux **chaînes constantes,** c'est-à-dire initialisées à la définition et jamais modifiées ensuite.

Exemple 65. Définition et initialisation de chaînes :

```
static char Identite[30] ="nom prenom" ;         /* la zone allouée est initialisée partiellement */
static char ponctuation[ ] = ".? ;, :!" ;        /* réserve (6+1) = 7 octets (chaîne constante */
static char message[50] = "Comment allez-vous? \n\n Très bien, merci!" ;
```

Rappel : le **nom** d'un tableau (ici, le nom de la chaîne) représente l'**adresse du tableau** (c'est-à-dire l'adresse de son premier élément) : *texte* équivaut à *&texte[0]*. Il est inutile de faire précéder le nom du tableau de l'opérateur &.

12.2. Ecriture à l'écran d'une chaîne (utilité : ***)

On peut utiliser la fonction **printf** avec le code format **%s** (*s=String*) réservé aux chaînes de caractères :

printf("%s", text) ;

affiche toute la chaîne *text* jusqu'à la rencontre du caractère nul 0.

Exemple 66. Utilisation de *printf* avec le code format %s

```
char phrase[50]="Je m'appelle Oscar" ;
printf("Message : %s", phrase ) ;            /* affiche les 18 caractères utiles de phrase */
```

On peut aussi utiliser la fonction **puts(texte)**, qui équivaut à **printf("%s\n", texte)**. Elle n'affiche qu'une chaîne de caractères à la fois et se termine par une **fin de ligne**.

Exemple 67. Utilisation de *puts* pour l'affichage d'une chaîne

```
char phrase[ ]="Je m'appelle Oscar…" ;
printf( "Voici un message : ") ;
puts( phrase ) ;          /* affiche : Voici un message : Je m'appelle Oscar… puis va à la ligne.
```

Plus long et toujours à déconseiller (les chaînes de caractères se manipulent avec des fonctions spécialisées) : en considérant la chaîne comme un tableau de caractères, on peut réaliser l'affichage caractère par caractère jusqu'au moment où le caractère nul est atteint. L'affichage des exemples précédents s'écrit alors (*i* est un entier initialisé à 0) :

```
while( phrase[i] )   printf("%c", phrase[i++]);
```

12.3. Lecture d'une chaîne au clavier

Pour une saisie au clavier (ou pour la lecture dans un fichier texte), on veut pouvoir lire un mot, une phrase, une ligne de texte. On utilise alors ***scanf*** ou ***gets*** selon qu'on veut lire un **mot** (séparateur = espace, tabulation, fin de ligne) ou une **phrase** (séparateur = fin de ligne seulement).

12.3.1 Lecture au clavier d'une chaîne par *gets* (utilité : ***)

La fonction **gets** permet de lire une phrase, c'est-à-dire une succession de caractères incluant les caractères *espace* ou *tabulation*, qui sont alors considérés comme les autres. Seul le caractère ***Entrée*** (fin de ligne) a droit à un traitement particulier, puisqu'il interrompt la lecture. Par exemple, les instructions

```
char texte[81] ;
gets(texte) ;
```

permettent la lecture au clavier d'une phrase complète, **la fin de la lecture n'étant provoquée que par la frappe de *Entrée*** (et non d'un autre séparateur). A noter : *Entrée* est éliminé du tampon clavier sans être mémorisé dans la chaîne (alors que *scanf* conserve le séparateur dans le tampon clavier).

Il est fréquent de devoir lire des chaînes de caractères qui comportent des espaces (phrases…). C'est donc *gets* (et ses variantes comme *fgets*) qui est la plus utile des fonctions de lecture de chaîne.

Exemple 68. Lecture d'une chaîne qui contient des espaces ou tabulations :

```
static char texte[81] ;
puts( "Tapez votre message : ") ;
gets( texte ) ;                      /* tapez par exemple : J'aime le langage C ! <Entrée> */
printf(" Vous avez écrit : %s", texte);     /* affiche :  J'aime le langage C ! */
```

12.3.2 Lecture au clavier d'une chaîne (mot) avec *scanf* (utilité : *)

Si on désire lire au clavier **un mot à la fois** pour le stocker dans une chaîne (ce n'est pas le cas le plus fréquent), la fonction **scanf** convient à condition de l'utiliser avec le code format **%s** ou **%N**s :

```
scanf( "%s", text ) ;
```

lit le clavier et remplit la chaîne *text* **jusqu'à la rencontre d'un séparateur** qui peut être **un espace, une tabulation ou un passage à la ligne (*Entrée*)**. La chaîne *text* est alors complétée par le caractère nul (de code ASCII 0). A noter : avec *scanf* (mais pas avec *gets*), le séparateur rencontré est laissé dans le tampon clavier.

L'utilisation de *scanf* interdit donc de lire en une fois une chaîne de caractères qui comporte des espaces ou des **tabulations**, puisque ceux-ci sont considérés comme des séparateurs par *scanf*.

Exemple 69. Lecture et écriture d'une chaîne (un mot à la fois) :

```
static char nom[21], prenom[21] ;                    /* 20 caractères utiles */
printf( "Entrez nom et prenom (avec tiret si nom composé): ") ;
scanf("%s%s", nom, prenom);      /* rappel : le nom d'une chaîne est déjà son adresse, donc pas de & */
printf("\n Bonjour, %s %s !", nom, prenom) ;         /* on peut taper : Jean-Marie Le-Pen */
Variante plus prudente :   scanf("%20s%20s", nom, prenom);
```

Attention à ne pas entrer plus de caractères que ne peut en contenir le tableau ! Etant donné l'absence de contrôle de dépassement du compilateur, il est prudent d'écrire `scanf("%Ns", ...)`, où N représente le nombre de caractères de la chaîne. Ici : **scanf("%20s%20s", nom, prenom);**

On remarquera aussi l'**absence de l'opérateur &** dans *scanf* , puisque le nom d'un tableau représente déjà son adresse.

12.4. Quelques fonctions de traitement de chaînes de caractères

Le Langage C fournit un grand nombre de fonctions de traitement de chaînes : copie, concaténation, recherches d'occurrence, conversion, initialisation .... Ces fonctions sont déclarées dans le fichier en-tête *string.h*, qu'il est conseillé de consulter avec l'aide en ligne.
Nous ne rappelons ici que quelques-unes des fonctions les plus utiles. Attention, aucun contrôle de dépassement de la taille des chaînes n'est en général effectué.
Voici un exemple qui sera suivi par la description rapide des fonctions utilisées. **UTILISEZ L'AIDE EN LIGNE POUR PLUS DE DETAILS** (fonctionnement, valeur renvoyée, exemples, autres fonctions, ...)

Exemple 70. Utilisation de quelques fonctions de traitement de chaînes

```
void main(void)
{
static char mot1[20]="Hello", mot2[20]="world" ;
static char phrase[50] = "" ;        /* chaîne vide */

strcpy( phrase, mot1 );              /* copie de la chaîne mot1 dans la chaîne phrase */
strcat( phrase, mot2 );              /* concaténation = « collage » de 2 chaînes */

printf( "%s", strcat(phrase, " !") );   /*concaténation avec une chaîne constante et affichage*/
printf( "\n Cette phrase a %d caractères ", strlen(phrase) ) ;

if ( strcmp(phrase,"Hello")==NULL )       /* comparaison de 2 chaînes */
     printf("Elle vaut Hello \n");
if ( strchr(phrase,'a')==NULL )          /* recherche d'un caractère dans une chaîne */
     printf("Elle ne contient pas la lettre 'a'");
}
```

Exemple 71. Fabrication et affichage (sous IDE CVI) de chaînes de caractères

```
void main(void)
{
static char libelle[81];
/* données de l'exemple (initialisées) */
char nom_article[] = "fer à repasser";  short i=24; double prix = 89.99 ;

sprintf( libelle," L'article n°%3hd du stock est %s.\n Il vaut %7.2lf euros",
        i, nom_article, prix);
MessagePopup("caractéristiques d'un article du stock", libelle );
}
```

caractéristiques d'un article du stock

L'article n° 24 du stock est fer à repasser.
Il vaut 89.99 euros

OK

12.4.1 **Fabrication d'une chaîne par *sprintf*** (utilité : ***)

Pour l'affichage (en particulier sous interface graphique), on a souvent besoin de fabriquer une chaîne de caractères **en assemblant du texte et des valeurs numériques**. La fonction ***sprintf***, une des fonctions les plus utiles, fournit un moyen très simple pour y arriver. Elle fonctionne exactement comme *printf*, sauf que le résultat est stocké dans une chaîne de caractères au lieu d'être affiché à l'écran.

☺ ***sprintf*** est une fonction « universelle » qui remplace avantageusement les fonctions de copie ou de concaténation.

Exemple 72. Fabrication d'une chaîne quelconque : voir *sprintf* dans l'Exemple 71.

L'utilisation de *sprintf* est souvent préalable à un affichage, en particulier dans le mode graphique pour lequel *printf* n'existe pas (fenêtre « Windows »), ou encore dans un fichier texte. Par exemple, la fonction *MessagePopup* de CVI affiche n'importe quoi dans une fenêtre *popup*, pourvu qu'on le lui donne sous forme de chaîne de caractères (avec un titre en option). Voir Exemple 71.
L'affichage de la chaîne de l'exemple ci-dessus peut ainsi s'écrire :

```
MessagePopup ("Combien gagnent-ils ?", phrase);   /* la chaîne constante est le titre */
```

Une autre application possible de *sprintf* est la **conversion de valeurs numériques** (entières ou réelles) **en chaînes de caractères.**

Exemple 73. *Conversion d'une valeur numérique en chaîne avec sprintf*

Pour convertir l'entier 9876543 en la chaîne "9876543", il suffit d'écrire :

sprintf(chaine, "%ld ", entier_long_a_convertir) ;

12.4.2 **Lecture formatée dans une chaîne avec *sscanf* (utilité : ***)**

Quand on dispose d'une phrase lue par **gets**, on peut y lire des informations par la fonction ***sscanf***. Elle fonctionne exactement comme *scanf*, sauf que le "flux d'entrée" est une chaîne de caractères au lieu d'être le tampon du Langage C.

Exemple 74. Utilisation de *sscanf* pour extraire des données d'un texte

☺ Pour pouvoir utiliser *sscanf* dans de bonnes conditions (sans rendre la lecture difficile, comme ci-dessous), il faut s'imposer un **format bien choisi (simple)** pour les données à lire.

Par exemple, on peut facilement extraire les 3 informations soulignées (nom, age et salaire) d'une chaîne *phrase* choisie de la forme « Sophie 20 ans 1000 euros ». Ce format, simple, permet de faire la lecture avec seulement 4 codes formats (dont un %*s pour sauter une information non utile) :

```
nb_val_lues = sscanf( phrase, "%s%hd%*s%lf", nom, &age, &salaire) ;      // %*s sert à sauter « ans »
if (nb_val_lues != 3)    printf ("problème de lecture") ;
```

Le code **%*s** permet de "sauter" un mot non significatif qu'on ne souhaite pas mémoriser (on l'utilise pour toutes les parties du texte qui ne sont pas à mémoriser) :

Voici un exemple de format mal choisi : la chaîne *phrase* est de la forme "Sophie (20 ans) gagne 1000.00 E par mois" ou "Max (30 ans) gagne 1650.00 E par mois". Ce format contient trop de caractères inutiles et il est difficile à lire. Il faut en effet écrire :

nb_val_lues = sscanf(phrase, "%s (%hd ans) gagne %lf", nom, &age, &salaire) ;

On peut aussi utiliser le code **%*s** qui permet de "sauter" un mot non significatif :

nb_val_lues =sscanf(phrase, "%s (%hd%*s%*s%lf", nom, &age, &salaire) ;

/* les codes format %*s permettent de sauter les deux sous-chaînes « ans) » et « gagne » */

La manipulation de *sscanf*, comme celle de *scanf*, est délicate. En particulier, tout caractère qui figure dans les guillemets en compagnie des codes formats (*%xx*) doit impérativement figurer dans la chaîne à lire. Dans notre exemple, le texte entre %s et %hd (espace puis parenthèse) doit figurer tel qu'il est écrit dans la chaîne à lire.

12.4.3 Longueur d'une chaîne avec ***strlen*** (utilité : **)

La fonction ***strlen*** fournit la longueur **utile** d'une chaîne de caractères (c'est-à-dire le nombre de caractères réellement mémorisés ; le caractère nul de fin de chaîne n'est pas compté). Son prototype est :

int strlen(const char *chaine) ; /* pour l'utilisation, voir Exemple 70. */

12.4.4 Copie d'une chaîne (peut être effectuée aussi par ***sprintf***) (utilité : **)

La fonction ***strcpy*** permet de copier une chaîne *source* dans une chaîne *destination* (y compris le caractère nul de fin de chaîne). La fonction renvoie l'adresse de la chaîne *destination*. Son prototype est :

char *strncpy(char *destination, char *source) ; /* voir Exemple 70. */

La variante ***strncpy*** permet un contrôle supplémentaire, car elle copie au plus *maxlen* caractères de la chaîne *source* dans la chaîne *destination*. Son prototype est :

char *strncpy(char *destination, char *source, int maxlen) ;

12.4.5 Concaténation de deux chaînes (peut être effectuée par ***sprintf***) (utilité : *)

sprintf permet de faire de l'assemblage de chaînes, mais il existe aussi une fonction spécifique : la fonction ***strcat*** permet d'ajouter une chaîne *source* à la fin d'une chaîne *destination* (c'est la "concaténation"). La fonction renvoie l'adresse de la chaîne *destination*. Son prototype est :

char *strcat(char *destination, char *source) ; /* voir Exemple 70. */

La variante ***strncat*** permet un contrôle supplémentaire, car elle ajoute au plus *maxlen* caractères de la chaîne *source* à la fin de la chaîne *destination* :

char *strncat(char *destination, char *source, int maxlen) ;

12.4.6 Comparaison de deux chaînes (utilité : **)

La fonction normalisée ISO ***strcmp*** permet de comparer une chaîne *chaine1* à une chaîne *chaine2*. La comparaison s'arrête quand deux caractères sont différents ou quand une chaîne est terminée. La fonction *strcmp* **renvoie l'entier 0 si les deux chaînes sont identiques**. Son prototype est ;

int strcmp(char *chaine1, char *chaine2) ; /* voir Exemple 70. */

Il existe de multiples variantes, parmi lesquelles :

- ***strncmp*** : elle compare deux chaînes en se limitant aux *maxlen* premiers caractères
  ```
  int strncmp( char *chaine1, char *chaine2, int maxlen );
  ```
- ***stricmp*** et ***strnicmp*** : variantes des deux précédentes, elles comparent deux chaînes sans faire de distinction entre majuscules et minuscules.

12.4.7 Recherches dans une chaîne (utilité : **)

Beaucoup de fonctions existent. En voici une liste non complète (voir aide en ligne de votre IDE) :

- La fonction ***strchr*** permet de rechercher la première apparition d'un caractère dans une chaîne. Voir Exemple 70.
- La fonction ***strstr*** recherche la première apparition d'une chaîne dans une autre chaîne.

- La fonction ***strtok*** divise une chaîne en sous-chaînes, en utilisant pour les séparer une liste de délimiteurs définis par l'utilisateur. D'utilisation un peu délicate (elle utilise un pointeur et son premier appel a un rôle particulier : voir Exemple 75. et l'aide en ligne du logiciel), elle est un outil efficace et puissant quand on sait l'utiliser.

L'aide en ligne est indispensable pour la manipulation des chaînes de caractères.

Exemple 75. Découpage d'une chaîne en sous-chaînes par *strtok*

```
static char texte[81] =
"salut: 1 2 3 partez! , 78.56 , comment allez-vous ? bien , -6541 , , fin";
char *ptr_chaine ;            /* pointeur pour  balayer les sous-chaînes obtenues */

printf("valeur initiale de la chaine : \n %s \n", texte );
ptr_chaine = strtok (texte, ",");      /* appel d'initialisation de strtok. Séparateur = ',' */

while ( ptr_chaine != NULL )
 {        /* on affiche la sous-chaîne courante avant de fabriquer la suivante : */
 printf("\n Chaine lue a la position courante de ptr_chaine :%s", ptr_chaine);
 ptr_chaine = strtok (NULL, ",?");  /* remplace le prochain séparateur trouvé par 0,
                         puis renvoie l'adresse de la chaîne ainsi obtenue. Séparateurs = ',' ou ' ?' */
 }
```

Voici le résultat obtenu :

```
Standard I/O
valeur initiale de la chaine :
 salut: 1 2 3 partez! , 78.56 , comment allez-vous ? bien , -6541 , , fin

 Chaine lue a la position courante de ptr_chaine: salut: 1 2 3 partez!
 Chaine lue a la position courante de ptr_chaine:  78.56
 Chaine lue a la position courante de ptr_chaine:  comment allez-vous
 Chaine lue a la position courante de ptr_chaine:  bien
 Chaine lue a la position courante de ptr_chaine:  -6541
 Chaine lue a la position courante de ptr_chaine:
 Chaine lue a la position courante de ptr_chaine:  fin_
```

Si on affiche les adresses successives des chaînes à l'aide d'une variante de ce programme, on obtient :

```
Standard I/O
Chaine pointee par texte=0x4280AC :
salut: 1 2 3 partez! , 78.56 , comment ca va ? bien , -6541 ,  , fin
Chaine pointee par prt_chaine=0x4280AC :        salut: 1 2 3 partez!
Chaine pointee par prt_chaine=0x4280C2 :         78.56
Chaine pointee par prt_chaine=0x4280CA :         comment ca va
Chaine pointee par prt_chaine=0x4280DA :         bien
Chaine pointee par prt_chaine=0x4280E1 :         -6541
Chaine pointee par prt_chaine=0x4280E9 :
Chaine pointee par prt_chaine=0x4280EC :         fin
```

12.4.8 Conversions d'une chaîne en valeur numérique et réciproquement

On peut convertir une chaîne en un entier ou en un réel, et réciproquement :

- d'une chaîne vers une valeur numérique (entière, réelle) : *atof*, *strtod* (String TO Double), *atoi*, *strtol*… On peut aussi utiliser ***sscanf*** (voir exemples *sscanf*), fonction « universelle » très utile dans la manipulation de chaînes.
- d'une valeur numérique (entière, réelle) vers une chaîne : utilisez ***sprintf*** (voir exemples *sprintf*).

Vous consulterez l'aide en ligne de votre logiciel pour plus de détails.

12.5. Les tableaux de chaîne de caractères

On a souvent besoin de manipuler un texte composé de lignes successives, donc une suite de chaînes.
On peut aussi vouloir stocker les différents intitulés d'un menu, ou de façon plus générale, les textes de différents affichages prédéfinis : il s'agit ici de chaînes « constantes », c'est-à-dire initialisées et jamais modifiées ensuite.
Enfin, on peut avoir besoin d'un tableau à deux dimensions de caractères, par exemple pour dessiner une image sur une fenêtre en mode texte : là encore, un tableau de chaînes sera bien pratique pour l'affichage.

Pour traiter un tel ensemble de chaînes, on définit un **tableau de chaînes de caractères**, c'est-à-dire un **tableau de caractères à deux dimensions**, avec un **caractère nul à la fin de chaque ligne**. Exemples :

> **static char** ligne**[NB_LIGNES][31] ;** /* définit NB_LIGNES chaînes de 30 caractères utiles */
>
> **static char** menu**[NB_OPTIONS][15] = { "memoriser", "continuer", "quitter", "autre option" } ;**

Pour manipuler la chaîne *i* du tableau de chaînes *ligne* (ou *menu*), il sufit d'utiliser la variable *ligne[i]* (ou *menu[i]*) qui représente l'adresse de la $i^{ème}$ chaîne.

Lors du remplissage d'un tableau de chaînes, la fonction *gets* et ses dérivées (*fgets*…) seront particulièrement utiles, car elles permettent la saisie de phrases.

Exemple 76. Définition, saisie et affichage de tableau de chaînes de caractères

```
static char jour[7][10] = { "lundi", "mardi", "mercredi", "jeudi",
      "vendredi","samedi", "dimanche" } ;       /* 7 chaînes constantes (de tailles diverses) */
static char day[7][12] ;                        /* 7 chaînes vierges, de 11 caractères utiles */
short int i ;

/* on effectue ici la saisie au clavier du nom anglais des 7 jours de la semaine, après l'affichage du nom français : */
for ( i=0 ; i<7 ; i++ )
   {
   printf( "En français, le jour n°%hd de la semaine est %s ", i+1 , jour[i] );
   printf( "\n Tapez le nom de ce jour en anglais : " ) ;
   gets( day[i] ) ;
   }
```

jour *tableau (à une dimension) de chaînes*

jour[0]	**"lundi"**
	"mardi"
jour[i]	**"jeudi"** ● —— *chaîne jour[i]*
jour[7]	**"dimanche"**

Exemple 77. Définition et affichage de tableau de chaînes de caractères :

```
#define NB_LIG 3
static char dessin[NB_LIG][20] =
                            {
                            "---(O---O)---",     /* chaînes constantes */
                            "   ----I----   ",
                            "     (===)     "
                            } ;
short int lig ;
/* affichage du dessin ligne par ligne : */
for ( lig=0 ; lig< NB_LIG ; lig++ )    puts( dessin[lig] ) ;
```

13 - Les pointeurs

On a vu jusqu'à présent comment accéder à une variable par **adressage direct** : on emploie le nom de la variable dans la partie du programme où celui-ci est connu et autorisé, c'est-à-dire, pour une variable locale, dans la fonction qui a défini la variable.

Il existe une autre façon d'accéder à une variable quand l'adressage direct n'est pas possible : on emploie **l'adresse** de la variable au lieu de son nom. C'est l'**adressage indirect**, qu'on utilise en particulier pour manipuler la variable dans une autre fonction que celle où elle est définie.

En informatique, l'adressage indirect est réalisé au moyen d'un **pointeur**. Un pointeur contient **l'adresse d'une variable** ; on dit qu'il « **pointe sur la variable** ». Il permet ainsi d'y accéder par adressage indirect.

Les pointeurs sont indispensables dans les applications suivantes :

- pour permettre à une fonction de modifier la variable locale d'une autre fonction. C'est le passage en paramètre par adresse. Cela inclut les fonctions (répandues) de **saisie** et **d'initialisation.**
- pour toutes les fonctions qui doivent calculer deux résultats ou plus (rappelons qu'il est alors impossible d'utiliser *return* !).
- pour créer des données pendant l'exécution du programme. C'est l'**allocation dynamique**, qui complète les allocations « statiques » faites pendant les phases de compilation/édition de liens. Les données créées dynamiquement pendant l'exécution sont accédées au moyen d'un pointeur.

13.1. Définition et affectation d'un pointeur

13.1.1 Définition d'un pointeur

Considérons la définition : `short int n = 3 ;`
n représente une variable du type entier, c'est-à-dire une adresse en mémoire dont le contenu est un entier de valeur 3. Cette définition a pour but de demander au compilateur/*linker* de réserver un emplacement mémoire, à une certaine adresse, pour la variable *n*.

Le contenu d'une variable peut être un entier, un réel, un caractère. Mais, c'est nouveau, ce contenu peut aussi être une **adresse**. Considérons les définitions suivantes :

short int* AdrEntier ;
double* AdrReel ; ou **double *AdrReel ;**

Cette écriture **`short int*`** signifie que *AdrEntier* est une variable dont le contenu est **l'adresse** d'une variable entière, tandis que *AdrReel* contient l'adresse d'une variable réelle.

On dit que *AdrEntier* est un **pointeur sur un entier** et *AdrReel* un **pointeur sur un réel**.

Ces définitions réservent la place mémoire nécessaire pour écrire deux adresses (une dans *AdrEntier*, l'autre dans *AdrReel*), mais elles ne réservent pas de place en mémoire pour les variables pointées (qui doivent être allouées séparément). De plus, *AdrEntier* et *AdrReel*, comme toute variable, ne sont pas initialisées a priori.

13.1.2 **Affectation d'un pointeur**

Soit un pointeur *ptr* et une variable *var* définis par :

```
short int var ;
short int* ptr ;
```

☞ L'étoile * peut être collée à un des deux mots qui l'entourent ou laissée isolée. Personnellement, je trouve *double* ptr* plus clair que *double * ptr*, car le type de *ptr* est bien *double* *.

Pour que *ptr* pointe sur *var*, il faut que le contenu de *ptr* soit **l'adresse** de *var*. Il suffit donc d'utiliser l'**opérateur d'adresse &** pour réaliser l'affectation :

ptr = &var ; /* à présent, *ptr* pointe sur *var* */

On peut ensuite manipuler la valeur de *var* à l'aide du pointeur sur *var*, par adressage indirect. L'instruction :

***ptr = 13 ;**

signifie : mettre la valeur 13 dans la « **variable pointée par *ptr*** » (c'est-à-dire *var*).

<u>*ptr désigne le contenu de l'adresse *ptr*</u>. ***** est ici l'opérateur d'indirection.

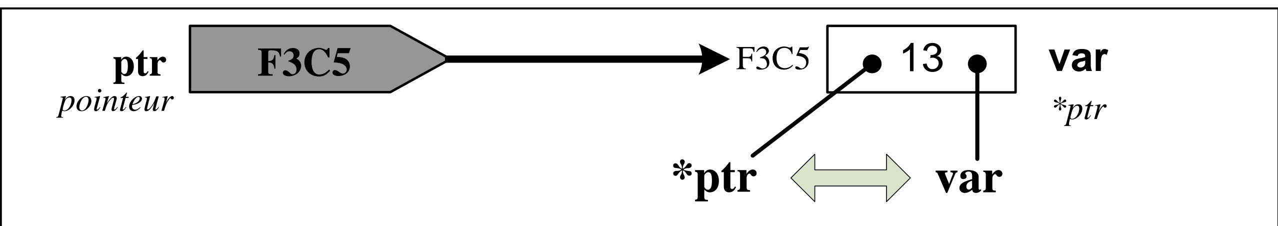
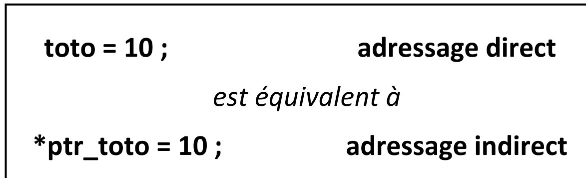


Figure 13 --4 : Pointeur et variable pointée



toto = 10 ;	**adressage direct**
est équivalent à	
***ptr_toto = 10 ;**	**adressage indirect**

En pratique, on dispose à l'intérieur d'une fonction, soit de la **variable** *toto* (adressage direct), soit de l'accès par **pointeur** **ptr_toto* (adressage indirect). Mais jamais des deux à la fois ! Il ne s'agit donc pas de créer des pointeurs juste pour le plaisir de compliquer les choses… Le pointeur est un moyen de manipuler une donnée **qui n'est pas accessible autrement** : parce qu'elle est définie dans une autre fonction ou parce qu'elle est créée par allocation dynamique.

☺ Le programmeur débutant doit utiliser pour un pointeur un nom qui commence par *p_* ou *ptr_*, afin de ne pas oublier la nature particulière de cette variable et la nécessité d'utiliser l'étoile * pour accéder à la variable pointée.
Seule exception à cette règle de prudence : quand le pointeur est utilisé pour pointer sur un tableau. Nous verrons en effet que les crochets contiennent déjà « l'indirection », et que l'étoile * ne sera pas utilisée pour accéder à la variable pointée quand il s'agit d'un élement de tableau. L'écriture *tab[i]* sera alors plus facile à utiliser que *ptr_tab[i]*.

L'affectation *ptr=&var* est une première façon (rarement utilisée sous cette forme) de donner une valeur à un pointeur. Il en existe d'autres, en particulier l'allocation dynamique : celle-ci consiste à réserver de la place en mémoire au cours de l'exécution et à placer l'adresse de cette zone mémoire dans un pointeur, dont la valeur n'est donc connue qu'à l'exécution.

L'allocation dynamique sera étudiée en détail au paragraphe 13.4, mais nous résumons ici son principe. Elle s'effectue à l'aide de l'instruction *malloc* dont la syntaxe est :

```
#include <stdlib.h>                    /*contient le protype de malloc */
type_var* ptr ;
ptr = (type_var*) malloc( nb_octets ) ;
```

L'instruction **`ptr=( ...)malloc(nb_octets)`** réserve de la place en mémoire pour *nb_octets* octets et place l'adresse correspondante dans le pointeur *ptr*. Cette allocation mémoire a lieu au cours de l'exécution, c'est pourquoi on dit qu'elle est **dynamique**.

Contrairement à `ptr=&a`, qui place dans le pointeur l'adresse d'une variable connue du compilateur, l'affectation du pointeur par *malloc* utilise l'adresse d'un « objet » qui n'existe pas encore à la compilation, et qui sera créé à l'exécution.

L'allocation mémoire sera étudiée plus en détail au paragraphe 13.4.

Exemple 78. Adresse et contenu de variables (intérêt pédagogique seulement)

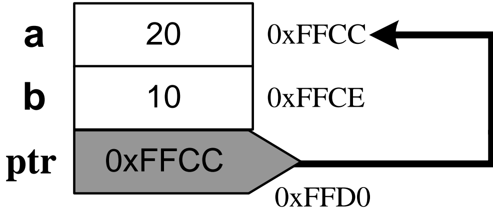
```
void main(void)
{
short int a,b ;
short int* ptr ;

ptr = &a ;          /* ptr pointe sur a */
a = 10 ;
b = *ptr ;          /* met dans b le contenu de l'adresse ptr (10) */
*ptr= b*2 ;         /* met la valeur b*2 à l'adresse pointée par ptr (attention aux 2 sens différents de *)  */

printf(" adresse de a : %p (%u en décimal)", &a, &a) ;
printf("\n adresse de b : %p (%u en décimal)", &b, &b) ;
printf("\n adresse de ptr : %p (%u en décimal)", &ptr, &ptr) ;
printf("\n\n contenu de ptr : %p", ptr) ;

printf("\n\n contenu de a : %hd \n contenu de b : %hd", a, b) ;
printf("\n contenu de la variable pointée par ptr : %hd", *ptr) ;
getch() ; /* attend la frappe d'une touche */
}
```

A la fin de ce programme, voici le contenu de la mémoire :



Ce programme affiche par exemple (avec l'allocation mémoire du schéma ci-dessus) :

```
adresse de a : FFCC (65484 en décimal)
adresse de b : FFCE (65486 en décimal)          --> un entier court occupe 2 octets
adresse de ptr : FFD0 (65488 en décimal)

contenu de ptr : FFCC                            --> on voit bien que ptr pointe sur a

contenu de a : 20
contenu de b : 10
contenu de la variable pointée par ptr : 20
```

13.1.3 Quelques pièges

On prendra garde à ne pas confondre un pointeur (*ptr*) avec l'objet qu'il pointe (**ptr*). L'exemple suivant, qui n'a qu'un intérêt pédagogique, illustre quelques manipulations courantes de pointeurs.

Exemple 79. **Jeu avec des pointeurs** (cet exemple ne sert qu'à illustrer l'utilisation de *)

```
void main(void)
{
short int a=12, b ;
short int * ptr1, * ptr2 ;

ptr1 = &a ;          /* ptr1 pointe sur a (qui vaut 12) */
ptr2 = &b ;          /* ptr2 pointe sur b */
*ptr2 = *ptr1 ;      /* la valeur 12 de a est copiée dans b */
ptr2= ptr1 ;         /* ptr2 pointe à présent aussi sur a */
}
```

Il ne faut pas confondre les écritures suivantes :

- `ptr1 = ptr2` est une affectation entre pointeurs : *ptr1* et *ptr2* contiennent désormais la même adresse (ils pointent sur la même variable) ;
- `*ptr1 = *ptr2` copie le contenu de l'emplacement mémoire pointé par *ptr2* dans celui pointé par *ptr1*. Les deux pointeurs ne contiennent pas les mêmes adresses ;
- `*ptr1=0` met à 0 le contenu de l'adresse pointée par *ptr1* ;
- `ptr1=0` ou mieux `ptr1=NULL` signifie par convention que *ptr1* **ne pointe sur rien**. Il est donc interdit de l'utiliser tant qu'il n'a pas reçu de valeur.

13.2. Arithmétique des pointeurs

13.2.1 Type d'un pointeur

Les notations **`char* ptr1`** et **`float* ptr2`** définissent deux variables *ptr1* et *ptr2* qui ont la même taille mémoire (car leur contenu est une adresse), mais se comportent de manière différente lors d'opérations arithmétiques telles que l'addition. Cela signifie donc que *ptr1* et *ptr2* sont de types différents.

Un pointeur a donc (comme toute variable) un type : le type ***int**** pour un pointeur sur un entier, le type ***double**** ou *float** pour un pointeur sur un réel, etc.

Ce type peut être utilisé dans des opérations de *cast* (conversion) avec l'opérateur *sizeof*.

13.2.2 Affectation de pointeur

Un pointeur est une variable qui contient une adresse. Les affectations suivantes peuvent être réalisées :

```
short int a = 12 ;    /* définition de variables scalaire ou tableau */
double table[10] ;

short int* ptr1 ;      /* définition de 3 pointeurs */
double* ptr2 ;
double * ptr3 ;

ptr1 = &a ;
ptr2 = table ;         /* qui équivaut à : ptr2 = &table[0] */
ptr3 = (double *)malloc ( 15 * sizeof(double) ) ;
    /* réserve le nombre d'octets nécessaire pour 15 float et place l'adresse du 1er octet dans ptr3 */
```

Remarquons la possibilité d'affecter à un pointeur la valeur 0 représentée par la constante *NULL* :

ptr = NULL ;

Par convention, un pointeur de valeur *NULL* ne pointe sur rien.

Attention : le compilateur ne vérifie pas si le pointeur que vous utilisez a été affecté avec une valeur non nulle et correcte. En cas d'oubli, un message peut apparaître en fin d'exécution, du type "Null pointer assignment". *C* emessage est l'indication d'un problème grave qui doit être résolu.

13.2.3 **Opérations sur les pointeurs**

Elles seront très utiles pour les chaînes de caractères ou pour les variables structurées telles que tableaux ou structures.

Les principales opérations sur les pointeurs sont l'incrémentation et la décrémentation à l'aide des opérateurs ++ et --. Voir l'exemple du paragraphe 14.1.

Soient ptrI et ptrR deux pointeurs définis par :

```
short int* ptrI ;
double* ptrR ;
```

Les instructions ptrI ++ et ptrR ++ provoquent l'incrémentation des deux pointeurs, mais l'opération n'est pas la même dans les deux cas :

- **`ptrI ++`** incrémente *ptrI* du nombre d'octets correspondant à la taille d'un entier : *ptrI* pointe donc ensuite **sur l'entier suivant**.
- **`ptrR ++`** incrémente *ptrR* du nombre d'octets correspondant à la taille d'un réel *double* : *ptrR* pointe donc **sur le réel *double* suivant**.

On peut aussi additionner un entier *n* à un pointeur : le pointeur **ptr+n** pointe sur le n[ème] objet qui suit celui pointé par *ptr*.

```
char text[10] ;
char *ptr1, *ptr2 ;
ptr1 = text ;          /* ptr1 pointe sur l'élément 0 du tableau */
ptr2 = ptr1 + 5 ;  /* ptr2 pointe l'élément 5 du tableau */
/*    *(ptr1+i) est synonyme de text[i] ou ptr[i], beaucoup plus lisible */
```

On peut appliquer les opérateurs relationnels (>, <, != etc ...) à des pointeurs. Voir exemple du paragraphe 14.1.

On utilise souvent la comparaison d'un pointeur avec la valeur *NULL* :

```
if ( ptr == NULL )  exit(0) ;     /* termine le programme "brutalement" */
```

13.3. Application des pointeurs au passage en paramètre

Le passage des paramètres par valeur empêche la modification des variables de départ, et une fonction ne peut retourner qu'une seule valeur par l'instruction *return*. Comment écrire une fonction qui modifie plusieurs variables de la fonction appelante sans que celles-ci soient définies comme globales ?

L'utilisation des pointeurs permet de répondre à cette question. Une fonction peut modifier autant de variables d'une uatre fonction qu'on le désire : il suffit de lui transmettre **l'adresse** d'une variable à modifier, au lieu de sa valeur. La fonction mémorise cette adresse dans un **pointeur** et peut alors accéder à la variable par adressage **indirect**.

Le passage en paramètre par adresse est très utile pour les fonctions **d'initialisation**, de **saisie** (*scanf* en est un bon exemple) et pour toutes les fonctions qui doivent calculer **deux** résultats ou plus (impossible de les renvoyer par *return*). Voici un exemple illustrant ce dernier cas.

Exemple 80. Calcul de DEUX résultats grâce au passage par adresse :

```
void calculer_puissances(float x, float* ptrCarre, float* ptrCube) ;

 void main(void)
      {
      double a = 6.3, carre_a, cube_a ;
      calculer_puissances( a, &carre_a, &cube_a ) ;
      printf("\n a=%lf - a au carré=%lf - a au cube=%lf", a, carre_a, cube_a);
      }

 void calculer_puissances(double x, double* ptrCarre, double* ptrCube)
      {
      *ptrCarre = x * x ;
      *ptrCube = x * x * x ;
      }
```

La fonction appelante fournit en paramètre **l'adresse** des variables « à remplir » (*&carre*, *&cube*) à l'aide de l'**opérateur d'adresse &**. La variable *a* est passée « normalement » par valeur, puisque sa valeur suffit à la fonction (qui ne doit pas modifier *a*).

La fonction appelée reçoit les adresses dans des pointeurs et utilise l'**opérateur d'indirection *** pour accéder aux variables par adressage indirect : **ptrCube* désigne le contenu de l'adresse placée dans *ptrCube* et permet de modifier la variable *cube* par adressage indirect (*cube* est locale à *main*, donc inutilisable directement).

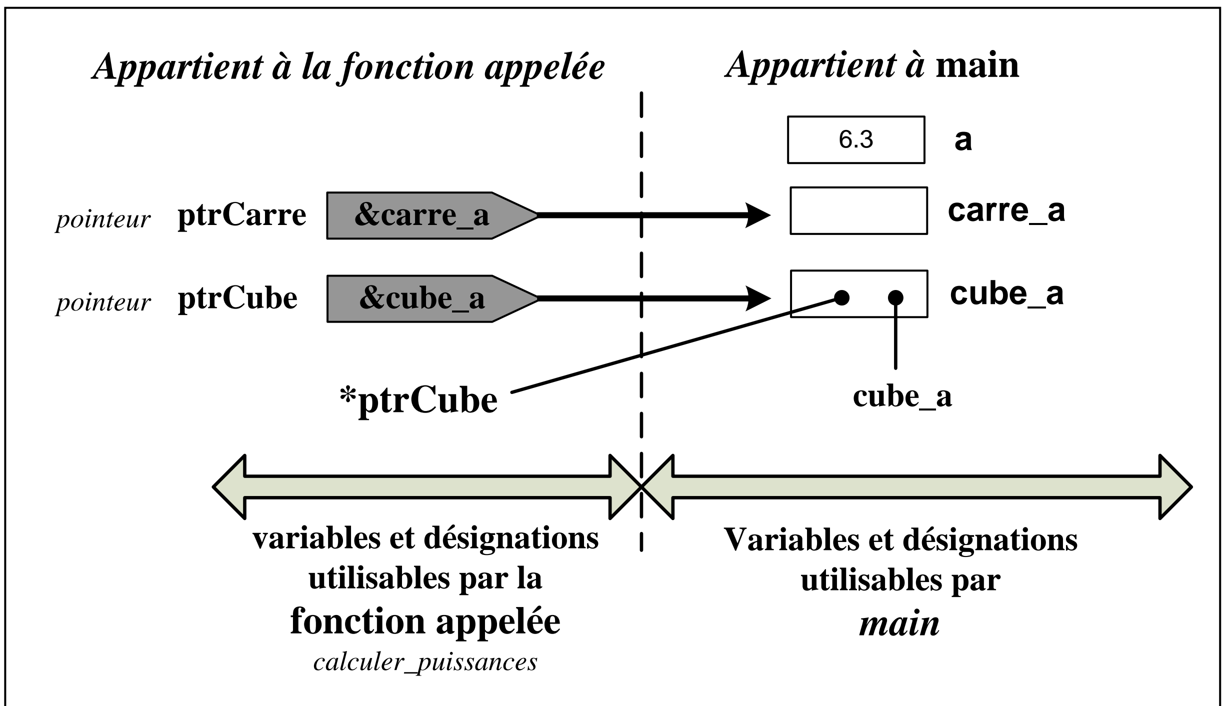


Figure 13 --5 : Passage en paramètre par adresse (ici pour calculer deux résultats)

L'exemple précédent pourrait se résoudre de façon correcte en utilisant deux fonctions au lieu d'une. Mais ce n'est pas toujours possible de se passer de pointeur... Un exemple classique (mais moins utilisé que le cas précédent), où les pointeurs sont indispensables, est celui de la **permutation** de deux variables à l'aide d'une fonction de permutation :

Exemple 81. Permutation de variables grâce au passage par adresse :

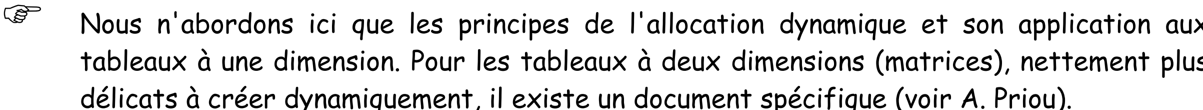
```
void permute(double* ptrX, double* ptrY); /*prototype de la fonction*/

 void main(void)
      {
      double a=10., b=20. ;
      printf(" a=%lf et b=%lf avant la permutation", a, b ) ;
      permute( &a, &b ) ;
      printf("\n a=%lf et b=%lf apres la permutation", a, b ) ;
      }
void permute( double* ptrX, double* ptrY)
      {
      double tampon ;
      tampon = *ptrX ;
      *ptrX = *ptrY ;
      *ptrY = tampon ;
      }
```

Ce programme affiche:

```
a=10 et b=20 avant la permutation
a=20 et b=10 apres la permutation
```

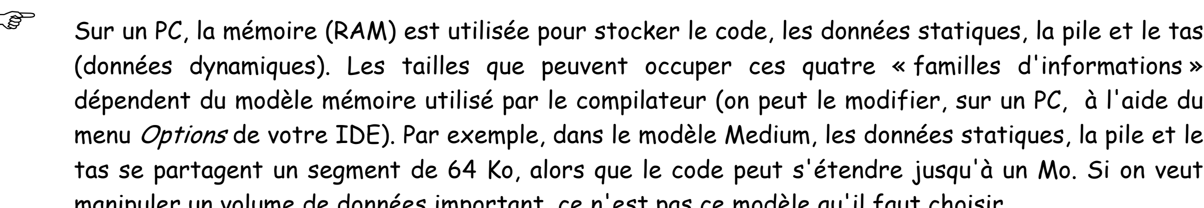
13.4. Application des pointeurs : allocation dynamique *malloc, free*


Nous n'abordons ici que les principes de l'allocation dynamique et son application aux tableaux à une dimension. Pour les tableaux à deux dimensions (matrices), nettement plus délicats à créer dynamiquement, il existe un document spécifique (voir A. Priou).

Nous avons vu qu'il existe :

- des variables **statiques**, occupant un emplacement mémoire déterminé à la compilation (exemple : variable locale *static* et variable globale) ;
- des variables automatiques, créées en pile et détruites au fur et à mesure de l'exécution des fonctions où elles sont définies (exemple : variable locale « normale »).

Il existe un troisième type de variables : les variables **dynamiques**, dont la création et la destruction dépendent de demandes explicites faites par le programme **au cours de l'exécution**. Elles sont situées dans une zone de la mémoire appelée le **tas** (*heap*).


Sur un PC, la mémoire (RAM) est utilisée pour stocker le code, les données statiques, la pile et le tas (données dynamiques). Les tailles que peuvent occuper ces quatre « familles d'informations » dépendent du modèle mémoire utilisé par le compilateur (on peut le modifier, sur un PC, à l'aide du menu *Options* de votre IDE). Par exemple, dans le modèle Medium, les données statiques, la pile et le tas se partagent un segment de 64 Ko, alors que le code peut s'étendre jusqu'à un Mo. Si on veut manipuler un volume de données important, ce n'est pas ce modèle qu'il faut choisir.

Le principal intérêt des variables dynamiques est de définir de « grosses » structures de données dont la taille ou le nombre ne sont connus qu'à l'exécution : par exemple, **tableaux de dimensions variables**, listes chaînées.

La gestion dynamique de données nécessite deux étapes :

- l'allocation de la mémoire, réalisée par la fonction ***malloc*** ;
- la libération de la mémoire, réalisée par la fonction ***free*** ; elle permettra de réutiliser la mémoire pour créer d'autres variables dynamiques.

Les prototypes des fonctions de gestion dynamique de la mémoire se trouvent dans le fichier en-tête *stdlib.h* de la bibliothèque standard du C.

13.4.1 Demander de la place en mémoire : la fonction *malloc*

La fonction *malloc* permet de réserver un emplacement mémoire dans le tas. La taille en octets de cet emplacement doit lui être précisée, en général avec *sizeof*.

L'utilisation de *malloc* est réalisée selon la syntaxe suivante (voir plus loin une version avec *sizeof*) :

```
type_var* ptr ;
ptr = (type_var*)malloc( nb_octets );    /*nb_octets est à calculer avec sizeof */
if (ptr==NULL) ...
```

A l'exécution, *malloc* réserve une zone mémoire de *nb_octets* octets et renvoie l'adresse du premier octet de la zone réservée (ou *NULL* si la mémoire disponible est insuffisante).

L'affectation *ptr=* permet ensuite de ranger cette adresse dans le pointeur *ptr*. Il faut convertir la valeur renvoyée par *malloc* avec le *cast* adéquat.

malloc renvoie le pointeur *NULL* si la mémoire qui reste dans le tas est insuffisante pour satisfaire la demande (ou si *nb_octets*=0). Le test de la valeur renvoyée par *malloc* est donc indispensable.

L'utilisation de l'opérateur *sizeof* est recommandée pour éviter le calcul « à la main » de *nb_octets*. L'utilisation de *malloc* qui suit est donc préférable :

```
ptr = (type_var *)malloc( sizeof(type_var) * nb_var );
if (ptr==NULL) ...
```

Cet appel permet de réserver le nombre d'octets nécessaires pour *nb_var* variables de type *type_var*.

13.4.2 Libérer de la place en mémoire : la fonction *free*

Un des intérêts de l'allocation dynamique est de libérer l'espace mémoire alloué lorsqu'il n'est plus nécessaire. Cet espace pourra donc être réaffecté ultérieurement lors de nouvelles demandes dynamiques.

La libération de la mémoire allouée par :

```
ptr = ( ...) malloc( nb_octets) ;
```

s'effectue très simplement avec la fonction *free* selon la syntaxe suivante :

```
free( ptr ) ;
```

13.4.3 Exemples d'allocation dynamique

Le premier exemple est le plus fréquent en pratique : il montre comment allouer dynamiquement un **tableau dont la taille est connue seulement à l'exécution**. Notez bien qu'une fois la mémoire allouée, le tableau *note* se manipule **exactement comme un tableau normal** (alloué statiquement), à condition d'utiliser la notation avec crochets.

Le deuxième exemple est destiné à illustrer le mécanisme d'allocation/libération, à de seules fins pédagogiques.

Un autre exemple d'utilisation de l'allocation dynamique est constitué par les listes chaînées : voir la fin du chapitre "Structures" 16.8).

Exemple 82. *Création d'un tableau de taille variable*

```
#include <stdlib.h>                /* pour malloc et free  */

void main(void)
       {
       short int n, i ;
       double* note ;
       double moyenne = 0. ;

       printf("\n Entrez le nombre de notes : ") ;
       scanf("%hd", &n) ;

       /* réservation dynamique d'une zone de n double: */
       note = (double *)malloc(n*sizeof(double));      /* affectation de note */
       if (note==NULL) { printf("\n Pb allocation !"); exit(0) ; }

       for (i=0 ; i<n; i++)
              {
              printf("\n Tapez la note %hd : ", i) ;
              scanf("%lf", note+i) ;              /* note+i ou &note[i] */
              moyenne = moyenne + note[i] ;       /* note[i] équivaut à *(note+i) */
              }
       printf("\n moyenne = %lf", moyenne/n) ;

       free( note ) ;        /*la mémoire récupérée par cette libération pourra être réutilisée*/
       }
```

Exemple 83. Fonctionnement de malloc/free (illustration à des fins pédagogiques)

/* Pour gagner de la place, nous ne testons pas les valeurs renvoyées par *malloc* (il faut le faire !)*/

```
void main(void)
{
short int * ptr1, * ptr2 ;

ptr1 = (short int *)malloc( 16*sizeof(int) ) ;
printf(" \n Allocation de 16 entiers à l'adresse %p ", ptr1 ) ;
ptr2 = (short int *)malloc( 32*sizeof(int) ) ;
printf(" \n Allocation de 32 entiers à l'adresse %p ", ptr2 ) ;

free( ptr1 ) ;
printf(" \n Libération de 16 entiers en %p", ptr1) ;

ptr1 = (short int *)malloc( 10*sizeof(int) ) ;
printf(" \n Allocation de 10 entiers en %p", ptr1) ;

free(ptr1) ;
free(ptr2) ;
}
```

Ce programme affiche par exemple :

```
Allocation de 16 entiers à l'adresse 072A
Allocation de 32 entiers à l'adresse 0752
Libération de 16 entiers en 072A
Allocation de 10 entiers en 072A --> utilise la place liberée par free
```

13.5. Pointeurs sur une fonction

Le Langage C offre la possibilité de définir un **pointeur de fonction**, qui permet notamment de **transmettre une fonction en paramètre** à une autre fonction.

13.5.1 Simplification des écritures avec *typedef*

La définition de « types synonymes » par *typedef* (voir le chapitre "Simplifications d'écriture") est ici indispensable pour simplifier des écritures très lourdes et accroître la lisibilité du programme.

Pour les types définis par *typedef*, il est conseillé d'adopter un nom facilement reconnaissable. En particulier, il est très utile que le nom du type commence par **T_** (pour *Type_*). Voici la convention (personnelle) adoptée pour la suite : un nom de type commence par **T_** et sera noté le plus souvent en **majuscules**.

Exemples : types *T_FONCTION, T_ADRESSE, T_date*. Le but de cette convention est d'éviter la confusion fréquente entre le nom du type créé par l'utilisateur et le nom des variables définies avec ce type.

Pour définir des pointeurs de fonction, il convient d'abord de créer un type pour caractériser le genre de fonction pointée. Par exemple :

```
typedef short int T_FONCTION( short int n ) ;
```

permet de définir un type « fonction recevant un entier et renvoyant un entier » baptisé *T_FONCTION*.

Pour définir ensuite des fonctions de ce type (appelées *calculer_carre* et *calculer_cube*), il suffit (si votre compilateur l'accepte) d'écrire :

```
T_FONCTION calculer_carre
        {
        return n*n ;                /* ou tout autre contenu */
        }
T_FONCTION calculer_cube
        {
        return n*n*n ;
        }
```

Attention ! certains compilateurs refusent l'utilisation du type *T_FONCTION* pour la définition de fonction (mais ils l'acceptent pour le prototype ou le pointeur de fonction). Il faut alors écrire classiquement :

```
short int calculer_carre( short int n )
        {
        return n*n ;
        }
```

On va voir que la définition et l'affectation d'un pointeur se fait alors simplement :

```
T_FONCTION* ptr_fct ;          /* définition du pointeur de fonction */
ptr_fct = calculer_cube ;       /* affectation du pointeur de fonction */
```

13.5.2 Définition et utilisation d'un pointeur de fonction

On commence par créer un type *T_FONCTION* pour caractériser les fonctions « qui reçoivent deux paramètres *short int* et *double* et qui renvoient un résultat de type *short int »* :

```
typedef short int T_FONCTION( short int n, double x ) ;
```

Pour définir un pointeur sur une fonction de type T_FONCTION, il suffit d'écrire très classiquement :

```
T_FONCTION* ptr_fct ;
```

ptr_fct est alors un pointeur sur une « fonction à 2 paramètres *short* et *double* qui renvoie un entier ».

☞ Juste pour vous montrer l'intérêt de *typedef* pour simplifier les écritures avec les pointeurs de fonctions, voici la définition de *ptr_fct* qui n'utilise pas *typedef* :

short int (*ptr_fct)(short int a, float b) ; /* bof ! */

On rappelle que le nom d'une fonction représente **son adresse**. Le pointeur de fonction *ptr_fct* peut donc être affecté par :

ptr_fct = f1 ;
ptr_fct = f2 ;

si *f1* et *f2* sont deux fonctions déclarées par :

T_FONCTION f1 ; /*notez la simplicité des prototypes avec *typedef* !*/
T_FONCTION f2 ;

On peut donc faire varier la fonction appelée par le programme en modifiant le contenu du pointeur.

Une fois que *ptr_fct* « pointe » sur une fonction donnée (par exemple *f1* ou *f2*), on peut **appeler la fonction par l'intermédiaire du pointeur**.
L' appel de la fonction dont l'adresse figure dans le pointeur *ptr_fct* s'écrit alors :

(*ptr_fct) (5, 12.4f)

ou

res = (*ptr_fct)(5, 12.4f)

Exemple 84. Utilisation d'un pointeur de fonction

```
/* Création du type T_FONC_PUISS : */
 typedef double T_FONC_PUISS(double x); /* type "fonction qui recoit un réel et renvoie un réel"*/

 /* Déclaration des fonctions (prototypes) : */
 T_FONC_PUISS calculer_carre ;           /* typedef simplifie aussi prototypes et en-têtes*/
 T_FONC_PUISS calculer_cube ;

 void main(void)
      {
      T_FONC_PUISS* p_puis;              /* définition du pointeur de fonction */

      float x = 3.7, res ;
      short int choix ;

      printf("Tapez 1 pour avoir le carre de x=%f, 2 pour son cube:", x);
      scanf("%hd", &choix) ;

      if (choix==1)       p_puis = calculer_carre ;     /* affectation du pointeur */
      else                p_puis = calculer_cube ;

      res = (*p_puis)( x );                    /* appel de la fonction via le pointeur */

      printf("\n résultat demandé = %f ", res ) ;
      }

 /* Définition des fonctions (voir mise en garde deux pages plus haut) : */
 T_FONC_PUISS calculer_carre { return x*x ; }
 T_FONC_PUISS calculer_cube { return x*x*x ; }
```

13.5.3 Passage en paramètre de l'adresse d'une fonction

L'utilisation de pointeur de fonction permet de passer une fonction en paramètre d'une autre fonction. Par exemple, créons un type « fonction qui reçoit un double et renvoie un double » appelé *T_FONC* :

typedef double T_FONC(double x) ;

et supposons définies deux fonctions *fct1* et *fct2* dont les prototypes sont :

T_FONC fct1, fct2 ;

On peut transmettre en paramètre l'adresse de *fct1* ou *fct2*, par exemple à une fonction de calcul de dérivée. Cette fonction de calcul de dérivée, qu'on appelle *calculer_deriv*, peut avoir un prototype du genre :

double calculer_deriv (T_FONC* ptr_fct, ...) ;

calculer_deriv reçoit l'adresse de la fonction à dériver ainsi que d'autres paramètres, et renvoie le résultat *double* de son calcul.

La fonction *calculer_deriv* sera simplement appelée de la façon suivante :

```
res1 = calculer_deriv( fct1, ... ) ;
res2 = calculer_deriv( fct2, ... ) ;
```

La fonction dont on transmet l'adresse à *calculer_deriv* sera appelée à l'intérieur de *calculer_deriv* par :

(*ptr_fct)(x)

ou

res = (*ptr_fct)(x) ;

En résumé (voir l'exemple qui suit) :

```
typedef double T_FONC(double x) ;            /* création du type T_FONCTION */
double calculer_deriv ( T_FONC* f, ... ) ;   /* prototype de calculer_deriv */
T_FONC fct1, fct2 ;                          /* prototypes de fct1, fct2  */

res = calculer_deriv( fct1, ... ) ;          /* appel de calculer_deriv */
res = (*f)(x) ;            /*appel dans calculer_deriv de la fonction passée en
paramètre*/
```

Exemple 85.   Passage en paramètre d'une fonction

```
/* création d'un type "fonction" nommé T_FONC : */
 typedef double T_FONC ( double x );

/* prototypes des fonctions (à l'aide du nouveau type T_FONC)  */
T_FONC fct1, fct2;
double calculer_deriv( T_FONC* f, double x );

/*---------------------------------------------------------------------*/
void main(void)
      {
      double res1, res2, x = 5.4;

      res1 = calculer_deriv( fct1, x );
      res2 = calculer_deriv( fct2, x );

      printf("\n dérivée de fct1 en x=%lf : %lf", x, res1 );
      printf("\n dérivée de fct2 en x=%lf : %lf", x, res2 );
      }
 /*---------------------------------------------------------------------*/
 /* définition de la fonction calculer_deriv (avec un paramètre de type T_FONC*) : */
 double calculer_deriv( T_FONC* f, double x )
      {
      double dx, res;

      dx = 1e-8 * x;
      res = ( (*f)(x+dx) - (*f)(x) ) / dx ;

      return res;
      }

/*---------------------------------------------------------------------*/
/* définitions des fonctions de type T_FONC (voir mise en garde du paragraphe 13.5.1) : */
 double fct1( double x )
      {
      return (2*x*x - 1.) ;
      }

 double fct2( double x )
      {
      return 1./x ;
      }
```

13.5.4 **Tableau de pointeurs de fonction**

Appelons *T_FCT* le type « fonction qui reçoit un *float* et un *short int*, puis renvoie un *float* » :

typedef double T_FCT(float x, short int entier) ;

On peut employer ce type pour créer un tableau de pointeurs de fonctions *T_FCT* et l'utiliser :

Définition d'un tableau de 4 pointeurs de fonction :

```
T_FCT* fct[4] ;    /* fct est le nom du tableau */
```

Affectation d'un pointeur de fonction :

```
fct[2] = nom_fonction ;
```

Utilisation d'un pointeur de fonction :

```
res = (*fct[2]) (1.35f, 12) ;
```

Exemple 86. Utilisation d'un tableau de pointeurs de fonction :

```
#include <stdio.h>
#include <math.h>                    /* on va utiliser des fonctions en bibliothèque du C */

#define NB_FCT 4

/* création d'un type "fonction" : */
typedef double T_FONC (double x);        /* prototype fréquent dans math.h */

/* prototype de la fonction calculer_deriv (définie plus bas) : */
double calculer_deriv( T_FONC* f, double x ) ;

/*--------------------------------------------------------------------------*/
/* définition complète d'une fonction personnelle */
double ma_fonction(double x) { return 1+x*x*x; }

/*--------------------------------------------------------------------------*/
/* Exemple de main : calcul des dérivées en x=2 de quatre fonctions stockées dans un tableau de fonctions */
void main(void)
      {
      short int i ;
      double der[NB_FCT] ;          /* pour stocker les dérivées calculées en x */
      double x = 2. ;

      /* définition et initialisation du tableau de pointeurs de fonctions : */
      T_FONC* fct[NB_FCT] = { ma_fonction, sqrt, exp, sin } ;

      /* calcul des dérivées en x=2 des 4 fonctions : */
      for (i=0 ; i<NB_FCT ; i++) der[i]= calculer_deriv( fct[i], x );

      /* affichage des résultats : */
      for (i=0 ; i<NB_FCT ; i++)
            printf("\n dérivée de la fct %hd en x=%lf : %lf", i+1, x, der[i] );
      }
/*--------------------------------------------------------------------------*/
/* définition de la fonction qui calcule la dérivée d'une fonction fournie en paramètre : */
double calculer_deriv( T_FONC* f, double x )
      {
      double dx, res;

      dx = 1e-8 * x;
      res = ( (*f)(x+dx) - (*f)(x) ) / dx ;

      return res;
      }
```

14 - Pointeurs et tableaux à une dimension

14.1. Relation nom de tableau - pointeur

On rappelle que **le nom d'un tableau *tab* représente l'adresse du début du tableau** (il n'est pas utile de le faire précéder de &) : la notation *tab* équivaut à *&tab[0].*

En réalité, le Langage C considère l'identificateur du tableau comme une constante de type pointeur sur les éléments du tableau. Les notations suivantes sont alors équivalentes :

tab	est équivalent à	**&tab[0]**
tab+i	est équivalent à	**&tab[i]**
***tab**	est équivalent à	**tab[0]**
***(tab+i)**	est équivalent à	**tab[i]**

Examinons cet exemple d'initialisation de tableau qui utilise les deux syntaxes pointeur et tableau :

Exemple 87. Syntaxe pointeur et tableau pour l'accès aux éléments d'un tableau

```
#define N 10          /* nombre d'éléments du tableau */
void main(void)
    {
    short int i;
    short int Tab[N] ;
    short int* ptrTab ;
    /* initialisation "classique" (conseillée !) : syntaxe de type tableau : */
    for (i=0 ; i<N ; i ++) Tab[i] = 10 ;
    /* initialisation à l'aide de la syntaxe de type pointeur : */
    for ( i=0 ; i<N ; i ++) *(Tab+i) = 10 ;
    /* initialisation à l'aide d'un pointeur : */
    for ( ptrTab=Tab ; ptrTab<Tab+N ; ptrTab ++) *ptrTab = 10 ;
    }
```

Analyse de l'initialisation utilisant la syntaxe de type pointeur `*(Tab+i)=10`

- *Tab* représente une constante de type pointeur, on peut l'utiliser comme tout pointeur pour accéder aux éléments du tableau. Mais on ne peut pas modifier ce "pointeur constant" : toute écriture du type *Tab ++* est donc interdite.
- **(Tab+i)=10* permet d'écrire la valeur 10 dans l'élément pointé par (Tab+i). Le compilateur/*linker* cherche l'adresse contenue dans Tab, lui ajoute *i*sizeof(Tab[0])* et écrit la valeur 10 à l'adresse ainsi obtenue (rappel : *sizeof* fournit la taille en octets du type ou de la variable qui lui sert d'opérande). L'écriture **(Tab+i)* est plus lourde que *Tab[i]*, qui est à utiliser en priorité.

Analyse de l'initialisation utilisant un pointeur `for(ptrTab=Tab;…;ptrTab++)*ptrTab=10`

- *ptrTab=Tab* permet d'écrire dans le pointeur *ptrTab* l'adresse du début du tableau *Tab*. Au début de la boucle, *ptrTab* pointe l'élément 0 du tableau.
- la valeur de *ptrTab* n'est pas gardée constante (à la différence de celle du pointeur *Tab* qui ne peut pas être modifiée) : elle est incrémentée à chaque exécution de la boucle pour permettre à *ptrTab* de pointer l'élément de tableau suivant. *ptrTab* pointe donc successivement tous les éléments du tableau.

La seule différence entre tableau et pointeur est qu'un nom de tableau représente une adresse **constante** (allouée par le compilateur) qui ne peut pas être modifiée (on dit que c'est une constante de type pointeur).

Des instructions comme *tab++* ou *tab=ptr* sont donc interdites si *tab* est un nom de tableau, alors qu'on peut incrémenter ou affecter une valeur à un pointeur.

L'utilisation de la syntaxe pointeur est souvent utilisée dans le traitement des chaînes de caractères :

Exemple 88. Utilisation de la syntaxe pointeur pour les chaînes

```
void copier_chaine( char *pdest, char *psrce )
      { while ( *(pdest++) = *(psrce++) ); }      /*écriture très condensée expliquée plus bas */
void main(void)
      {
      char srce[10] = "Bonjour!", dest[10];

      copier_chaine( dest, srce );
      printf("chaine destination = %s", dest );
      }
```

La boucle *while* de la fonction *copier_chaine* est un exemple des écritures condensées que permet le Langage C (ce n'est pas forcément à imiter, car la lisibilité n'en est pas améliorée !). Cette boucle peut s'écrire de façon moins compacte et donc plus lisible pour le débutant :

```
void copier_chaine( char *pdest, char *psrce )
      {do
             {
             *pdest = *psrce ;            /* copie d'un caractère de la chaine */
             pdest ++ ; psrce ++ ;        /* passage au caractère suivant */
             } while (*psrce != 0);       /* on s'arrête quand le caractère nul est atteint */
      }
```

14.2. Transmission d'un tableau en paramètre d'une fonction

Nous avons déjà étudié comment transmettre un tableau en paramètre d'une fonction. Essayons maintenant de comprendre ce qui se passe à la lueur de notre connaissance des pointeurs.

Exemple 89. Transmission d'un tableau en paramètre :

```
void initialiser( short int* tab, short int nb_elt );        /* prototype de la fonction */
void main(void)
{
short int entier[N] ;
initialiser ( entier, N ) ;       /* argument d'appel tableau */
}
void initialiser (short int* tab, short int nb_elt) /* on peut aussi écrire: short int tab[ ] */
{
short int i ;
for (i=0 ; i<nb_elt ; i ++)
      tab[i] = 10 ;               /* on peut aussi écrire : *(tab+i) = 10 (beaucoup moins lisible !) */
}
```

Lors de l'appel de la fonction *initialiser*, le paramètre formel *tab* reçoit une adresse, celle du début du tableau *entier* transmise par la fonction appelante *main*. *tab* est donc un pointeur sur un élément de tableau *int*, d'où sa définition de paramètre formel *short int* tab*.

Pour permettre de conserver les notations des tableaux (*tab[i]* est plus pratique que **(tab+i)*), la Langage C autorise à « dissimuler » la nature de pointeur du paramètre formel *tab* en utilisant la notation avec crochets *short int tab**[ ]*** pour sa définition. Quand un nom de tableau est transmis en paramètre, on peut donc utiliser indifféremment dans la fonction la notation des tableaux ou celle des pointeurs. Bien sûr, celle avec les crochets est la plupart du temps préférable.

15 - Pointeurs et chaînes de caractères

15.1. Pointeur et constante chaîne de caractères

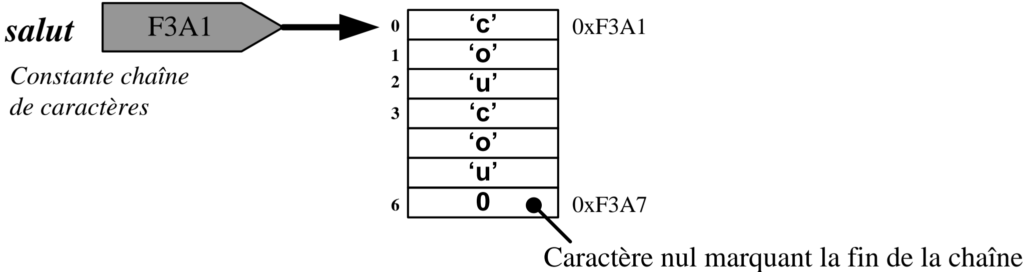
Etudions les instructions suivantes :

char* salut ;
salut = "coucou" ;

salut est un pointeur sur une variable de type *char*.

Le **compilateur** place la chaîne "coucou" dans une zone mémoire de 7 octets, c'est-à-dire un tableau de 7 caractères qui contient les 6 lettres du mot *coucou* et le caractère nul 0.

A l'exécution, l'affectation *salut*= met l'adresse de cette zone mémoire dans le pointeur *salut* : *salut* pointe donc sur le premier caractère de la chaîne "coucou".



La chaîne "coucou" est appelée **constante chaîne de caractères**. L'écriture "coucou" représente pour le compilateur, non pas la valeur de la chaîne elle-même, mais son **adresse**. D'où la possibilité d'affecter le pointeur par l'instruction *salut = "coucou"*.

Il est interdit de modifier à l'exécution la valeur de la chaîne pointée par *salut*, car c'est une constante.

15.2. Retour sur les tableaux de caractères

Considérons le tableau de caractères *text* défini par :

char text[20] ;

Il est interdit de réaliser une affectation directe du type : `text = "coucou" ;`

puisque *text* représente une adresse **constante** qui ne peut pas être modifiée (*text* est une constante de type pointeur). Seule l'initialisation **<u>en même temps que la définition</u>** est permise :

char text[20] = "coucou" ; /* conseillé */

Cette écriture est en fait une notation qui permet de simplifier l'écriture de la « véritable » initialisation :

char text[20] = { 'c', 'o', 'u', 'c', 'o', 'u', 0 } ; /* déconseillé */

Cette fois, on peut bien sûr modifier le contenu du tableau *text* comme on le souhaite.

15.3. Les tableaux de pointeurs

Rien n'empêche de combiner pointeurs et tableaux pour réaliser des tableaux de pointeurs. C'est surtout utile pour disposer d'un **tableau de chaînes de caractères**, très pratiques pour les affichages.

La définition :

char* jour[7] = { "lundi", "mardi", "mercredi", "jeudi", "vendredi", "samedi", "dimanche" } ;

permet de définir un tableau *jour* de 7 éléments, chaque élément étant de type pointeur de caractère *char**. Ces éléments sont initialisés avec les **adresses** des sept « constantes chaînes ».

Chaque élément *jour[i]* est un pointeur sur une chaîne, qui s'utilise comme tout nom de chaîne :

jour[6] = "Jour de Repos";
printf("\n Le jour %hd de la semaine est %s", i+1, jour[i]);

Exemple 90. Affichage d'un menu : tableau de chaînes de caractères

```
#include <stdio.h>

#define NB_OPTIONS 4

 void main(void)
   {
   short int i ;
   static char* menu[NB_OPTIONS] = {"Quitter", "Sauvegarder", "Save as", "Autre"};

   for (i=0 ; i<NB_OPTIONS ; i++)
        printf("\n %s :\t\t tapez %hd", menu[i], i) ;
   }
```

16 - Les structures

Une structure est une forme de données évoluée, qui permet d'effectuer un "rangement en mémoire" très propre ("structuré") et **personnalisé** (on crée la structure adaptée exactement au problème informatique à traiter). La structure est aux données informatiques ce que le dressing est aux affaires d'une chambre : un moyen efficace et personnalisé de tout ranger correctement...

Comme un tableau, la structure permet de ranger un ensemble d'informations. Tableau et structure ne sont pas concurrents, mais complémentaires :

- un tableau regroupe sous un même nom un ensemble de variables de **même type**, repérées par leur **indice**.
- la structure, elle, rassemble sous un même identificateur des variables de **types différents**. Chaque élément de la structure (appelé **champ**) est désigné par un nom qui permet d'y avoir accès.

Il est fréquent de manipuler un tableau de structures ou une structure qui contient des tableaux.

Le rôle d'une variable structurée est de ranger en mémoire, sous une forme **bien ordonnée** (bien "structurée"), toutes les informations se rapportant à un objet informatique. Grâce à un bon agencement des données, le programme gagne en **lisibilité** ; en outre, il devient beaucoup plus **évolutif**. Ces deux avantages font des structures un élément incontournable d'un bon programme.

☞ Dans ce chapitre, toutes les créations de types structurés utiliseront l'instruction *typedef*. Bien que non indispensable (voir remarque du §.1), l'utilisation de *typedef* simplifie l'écriture du programme et en accroît la lisibilité. Elle est fortement conseillée.

☺ Afin d'éviter la confusion fréquente entre le nom du **type** créé par l'utilisateur et le nom des **variables** définies avec ce type, le nom d'un type créé par *typedef* doit toujours commencer par T_ (T comme Type).

16.1. Modèle de structure

On souhaite par exemple caractériser des personnes à l'aide de leurs nom, prénom et année de naissance. On crée donc un type structuré *T_personne* décrit par **un modèle** qui permet de regrouper ces informations :

```
typedef struct
    {
    char nom[20] ;         /* premier des 3 champs de ce type structuré */
    char prenom[20] ;
    short int an_naiss ;
    }  T_personne ;        /* nom du type structuré */
```

champ

nom
prenom
an_naiss

Le mot-clé ***typedef*** permet de définir un **nouveau type** (d'autres exemples d'utilisation en sont donnés au chapitre *Simplifications d'écriture*). Ici, il s'agit d'un type structuré, comme l'indique le mot-clé ***struct*** qui commence la description de la structure. Celle-ci se termine par le nom du type structuré (ici *T_personne*). Les éléments entre accolades sont appelés les **champs** de la structure : ils sont caractérisé par leur type et leur nom.

☺ L'ordre des champs est quelconque, mais il est conseillé pour faciliter l'initialisation de mettre en premier les champs associés aux "données **d'entrée**", puis les champs des "données intermédiaires", enfin les champs associés aux "données **de sortie**" (ceux qui sont calculés par le programme).

Ce **modèle** n'est pas une définition de variable ; il se contente de **décrire** la structure, tout comme un **plan** de maison décrit la future construction. Il n'y a donc aucune réservation mémoire pour l'instant. Mais on dispose à présent d'un nouveau type qui s'ajoute aux types prédéfinis du Langage C (*int*, *double*...).

Ce modèle de structure est en général placé **au début du fichier**, en dehors de toute fonction, ou mieux **dans un fichier en-tête *.h**. Voir l'exemple à la fin du chapitre.

Voici un autre exemple, avec ce modèle de structure *T_adresse* qui permet de rassembler les éléments d'une adresse :

```
typedef struct
        {
        short int numero ;
        char rue[30] ;
        long int code_postal ;
        char ville[20] ;
        } T_adresse ;
```

champ
numero
rue
code_postal
ville

Voici enfin un exemple de structure qui possède un **champ de type tableau** (cas fréquent en informatique). Ce modèle de structure *T_enfants* permet de rassembler les informations concernant les enfants d'une famille :

```
typedef struct
   {
   short int nb_enfants ;          /*entier entre 0 et 20 (choix) */
   short int an_naiss_enf[20];  /*nombre d'enfants limité à 20 */
   char nom_enf[20][30] ;         /*tableau de 20 chaînes de 30 caractères*/
   } T_enfants ;
```

nb_enfants
0 1 2 *an_naiss_enf* (tableau de 20 entiers)
0 1 2 19 *nom_enf* (tableau de 20 chaînes)

Remarque :
On peut définir un type structuré sans utiliser *typedef*, mais ce n'est pas conseillé pour la légèreté des écritures. En effet, au lieu du nom *T_adresse*, le modèle de structure ci-dessous oblige à utiliser le nom composé *struct t_adresse* (les deux mots ne peuvent jamais être dissociés dans le nom du type !) :

```
struct t_adresse            /* exemple à DECONSEILLER, car on n'utilise pas typedef */
       {
        short int numero ;
        char rue[30] ;
        long int code_postal ;
        char ville[20] ;
        } ;                           /* aucun nom ici (il est défini plus haut) */
```

16.2. Définition d'une variable de type structuré

En définissant le modèle de la structure, on crée un **nouveau type** dit structuré. On peut donc définir des variables avec ce nouveau type comme avec n'importe quel autre type.
Ainsi, la définition :

static T_personne Oscar, Isidore ; /* *static* est conseillé */

permet de définir deux variables *Oscar* et *Isidore* qui sont de type structuré *T_personne*.

La place occupée par une variable structurée est la somme de la place occupée par chacun de ses champs (qui sont consécutifs). Elle peut être fournie par l'opérateur *sizeof* :

sizeof(T_personne) ou *sizeof(Oscar)* vaut 42 (octets)

On peut bien sûr utiliser le type structuré pour la définition de tableaux. La définition :

static T_personne eleve[30] ;

définit un tableau *eleve* de 30 éléments, chaque élément étant de type structuré *T_personne*. Ce tableau occupe 30*42 octets en mémoire, ce que confirme la valeur 1260 fournie par *sizeof(eleve)*.

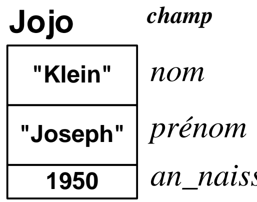
Une variable structurée, à plus forte raison un tableau de structures, occupe suffisamment de place en mémoire pour que le mot-clé *static* soit conseillé (rappelons que *static* demande que le tableau soit créé en mémoire statique plutôt qu'en pile : c'est indispensable pour les « grosses » variables).

Notons que si le modèle de structure n'a pas été défini en utilisant *typedef*, les écritures se compliquent beaucoup (il faut traîner le mot-clé *struct*) : `struct t_adresse mairie ;` A éviter donc !

Notons enfin qu'il est techniquement possible de définir simultanément (sur la même ligne) le type structuré (sans *typedef*) et des variables de ce type. C'est absolument déconseillé : la lisibilité est nulle et les variables ainsi créées sont globales, ce qui est à prohiber !

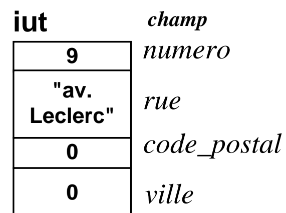
16.3. Initialisation d'une variable structurée

L'initialisation se fait avec la même syntaxe que pour les tableaux,
au moment de la définition :




```
static T_personne Jojo = { "Klein", "Joseph", 1950 } ;
```

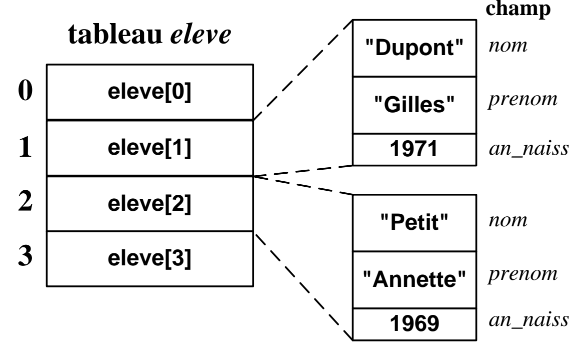
ou si on veut une initialisation **partielle** (des premiers champs obligatoirement) :




```
static T_adresse iut = { 9, "av. Leclerc" } ;
/* ici, les 2 derniers champs ne sont pas initialisés :
   ils seront mis à 0 automatiquement */
```

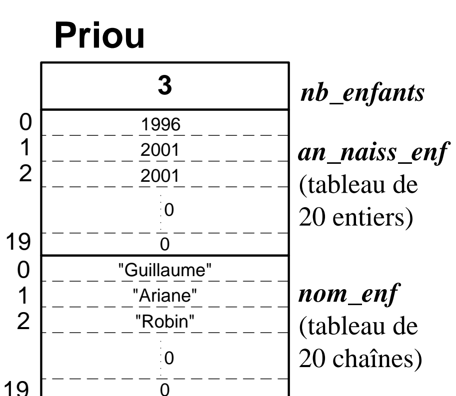
ou pour un **tableau de structures** :

```
static T_personne eleve[4] =
  {   /*chaque élément du tab. est entouré d'accolades*/
      { "Dupuis", "Claude", 1970 },
      { "Dupont, "Gilles", 1971 },
      { "Petit", "Annette", 1969 }
  } ;  // dernier élement laissé sans initialisation (0 partout)
```



ou pour une **structure qui contient un tableau** :

```
static T_enfants Priou =
   {
   3,                        /* champ nb_enfants */
   { 1996, 2001, 2001 },     /* champ an_naiss_enf (tableau) */
   {"Guillaume","Ariane","Robin"}   /* champ nom_enf (tab)*/
   } ;
```



16.4. Accès aux champs d'une variable structurée

Deux opérateurs spécifiques "." (point) et "->" (flèche) sont utilisés, selon qu'on dispose d'un adressage **direct** (par le nom) ou **indirect** (par un **pointeur**) à la variable structurée.

16.4.1 Opérateur "."

Pour accéder au champ *champ* de la variable structurée *nom_var*, il suffit de les relier par l'opérateur ".", qui a la priorité la plus élevée :

nom_var.champ

Grâce à cet opérateur, chaque champ se manipule **comme une variable du même type**.

Exemple 91. Utilisation des champs d'une structure (de différents types)

```
Isidore.an_naiss ++ ;                       /* incrémentation d'un champ entier */
scanf( "%ld", &iut.code_postal ) ;          /* saisie d'un champ long */
printf( "nom : %s", Oscar.nom ) ;           /* affichage d'un champ chaîne de caractères */
strcpy( iut.ville, "Sceaux" ) ;
eleve[i].an_naiss = 1980 ;                  /* accès à l'élément i d'un tableau de structures */
Priou.annee_naiss_enfant[2] = 2002 ;        /* accès à un champ de type tableau */
```

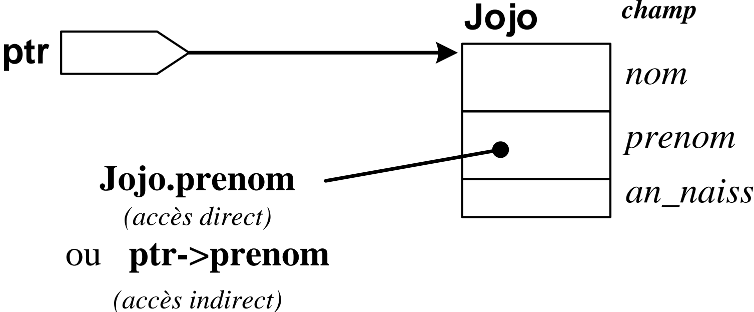


Figure 16 --6 : Structure en accès direct ou indirect

Pour l'accès aux champs d'un élément de tableau de type structuré (si on dispose, cas fréquent, d'un tableau de structures) :

eleve[i].champ (et non `eleve.champ[i]` qui a une signification différente)

permet d'accéder à un champ de l'élément *i* du tableau de structures *eleve*.

Inversement, quand le tableau est contenu DANS la variable structurée *toto* : on accède à l'élément *i* du champ tableau *tab* avec **toto.tab[i]** (cf dernière ligne des exemples ci-dessus).

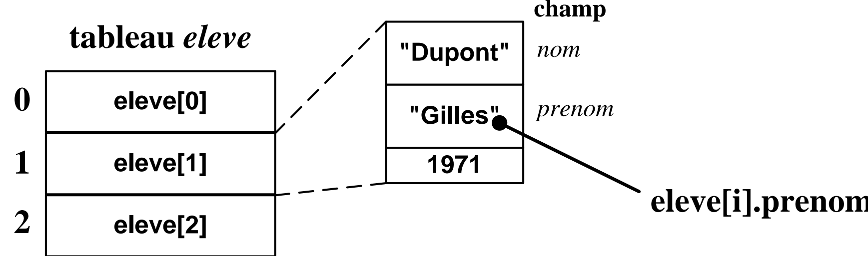


Figure 16 --7 : Tableau de structures

☞ La dernière normalisation a rendu possible l'**affectation de structures** :

Oscar = Jojo ; ou **Isidore = eleve[0] ;**

16.4.2 **Opérateur -> (flèche)**

Il est utilisé quand on dispose, au lieu d'une variable de type structuré, d'un **pointeur** sur une telle variable. Un pointeur *ptr* sur une variable structurée *nom_type* (type créé par *typedef*) se définit par :

nom_type *ptr ;

Soit *ptr* un pointeur sur une variable structurée et *champ* un champ de cette variable. La notation :

ptr->champ

remplace l'écriture plus lourde (mais correcte) : *(*ptr).champ*

Cet opérateur -> sera souvent utilisé pour manipuler une variable structurée passée en paramètre par adresse à une fonction. Voir le long exemple du chapitre suivant.

16.5. La structure en tant que paramètre

16.5.1 **Portée du modèle de structure**

Si on crée un modèle de structure à l'intérieur du bloc d'une fonction, ce type structuré n'est utilisable que dans cette fonction. Or, dans la plupart des cas, on désire que le type structuré soit connu de toutes les fonctions du programme. On rend donc le modèle « global » en le sortant de toute fonction. C'est pourquoi on trouve en général les modèles de structures **en début de fichier** ou mieux, **dans un fichier en-tête** (d'extension .h.)

Un exemple de fichier en-tête avec un type structuré est donné à la dernière page de ce chapitre. Cet exemple est **le modèle à suivre** pour écrire un programme en compilation séparé avec des types structurés.

16.5.2 **Passage en paramètre de variables structurées**

L'utilisation de variables structurées rend le programme **beaucoup plus évolutif** : on peut à tout moment ajouter un champ supplémentaire dans le modèle de structure, **sans modifier les appels, les prototypes et les en-tête de fonctions** ; seuls les corps des fonctions qui utilisent le champ rajouté seront à compléter.

La norme ISO actuelle autorise :

- la transmission en paramètre d'une variable structurée **par valeur** (la fonction ne manipule que la copie de la valeur) ou **par adresse** (la fonction peut modifier la variable originale en utilisant un pointeur) ;
- le renvoi par une fonction d'une valeur de type structurée.

Dans les anciens compilateurs, seule la transmission par adresse des structures était autorisée (comme pour les tableaux). Une valeur retournée de type structurée était aussi interdite.

☺ En dépit de la nouvelle norme ISO, il reste conseillé de transmettre **PAR ADRESSE** les variables structurées. Le programme est plus évolutif, puisque chaque fonction peut accéder en lecture comme en écriture aux variables structurées transmises par adresse. Evitez aussi d'utiliser une valeur de retour structurée.

Dans le cas d'un passage de paramètre par adresse, la fonction appelée récupère une adresse de variable structurée, c'est-à-dire un **pointeur sur une structure**. Un tel pointeur se définit par *nom_type_struct *ptr*. L'accès aux champs de la structure (voir dessin) se fait ensuite simplement à l'aide de l'opérateur flèche -> :

ptr->champ

Exemple 92. Passage en paramètre d'une structure (voir dessin page suivante et l'exemple complet avec fichier en-tête en fin de chapitre)

```
/* modèle de structure T_etud (utilisation de typedef) : */
 typedef struct
       {
       char nom[20] ;
       float moy[3] ;
       short int age ;          /* on peut ajouter des champs de sortie comme moy_generale ... */
       } T_etud ;

 /* prototypes des fonctions : */
 void saisir_donnees( T_etud *pEt ) ;                 /* paramètre = adresse (conseillé) */
 float calculer_moy_generale( T_etud *pEt ) ;
 void afficher( T_etud pers ) ;                       /* paramètre = valeur (déconseillé) */

/*---------------------------------------------------------------------------*/
 void main(void)
   {
   static T_etud etudiant ={"Toto", {10,10,10}, 20};    /*déf de la var. struct avec valeurs de test*/
   float moy_generale ;

   /* Appel des 3 fonctions avec transmission d'adresse ou de valeur de structure : */
   saisir_donnees( &etudiant ) ;                      /* passages en paramètre par adresse */
   moy_generale = calculer_moy_generale( &etudiant ) ;
   afficher( etudiant ) ;                             /* passage en paramètre par valeur (déconseillé)
*/
   }
 /*---------------------------------------------------------------------------*/
 void saisir_donnees( T_etud *pEt )         /* pEt est un pointeur sur la structure à traiter */
   {          /* attention : le nom d'une chaîne de car. est déjà une adresse (donc pas de & !) */
   printf("\n Nom et age de l'etudiant : ") ;
   scanf("%s%d", pEt->nom,  &pEt->age ) ;
   printf("\n Donnez les 3 moyennes de l'etudiant : ") ;
   scanf("%f%f%f", &pEt->moy[0] ,  &pEt->moy[1],  &pEt->moy[2] ) ;
   }
/*---------------------------------------------------------------------------*/
 float calculer_moy_generale( T_etud *pEt )      /* pEt est un pointeur sur la structure à traiter */
   {           /* on utilise exclusivement l'opérateur "flèche" pour accéder aux champs */
   float moy_gene ;
   moy_gene = (pEt->moy[0] + pEt->moy[1] + pEt->moy[2]) / 3 ;
   printf("\n Moyenne generale de %s : %5.2f",  pEt->nom, moy_gene) ;
   return moy_gene ;
  }
/*---------------------------------------------------------------------------*/
/* La fonction qui suit affiche par exemple :
                      Nom : Dupont
                      Age : 18 ans
                      Moyennes de l'etudiant : 12.50 8.00 10.25 */

 void afficher( T_etud pers )           /* pers est une copie de la variable etudiant de main */
   {           /* Le passage en paramètre par valeur est utilisé ici pour l'exemple (déconseillé en pratique) */
   printf("\n Nom : %s \n Age : %2d ans", pers.nom, pers.age ) ;
   printf("\n Moyennes de l'etudiant : %5.2f %5.2f %5.2f ",
                pers.moy[0], pers.moy[1], pers.moy[2]) ;
   }
```

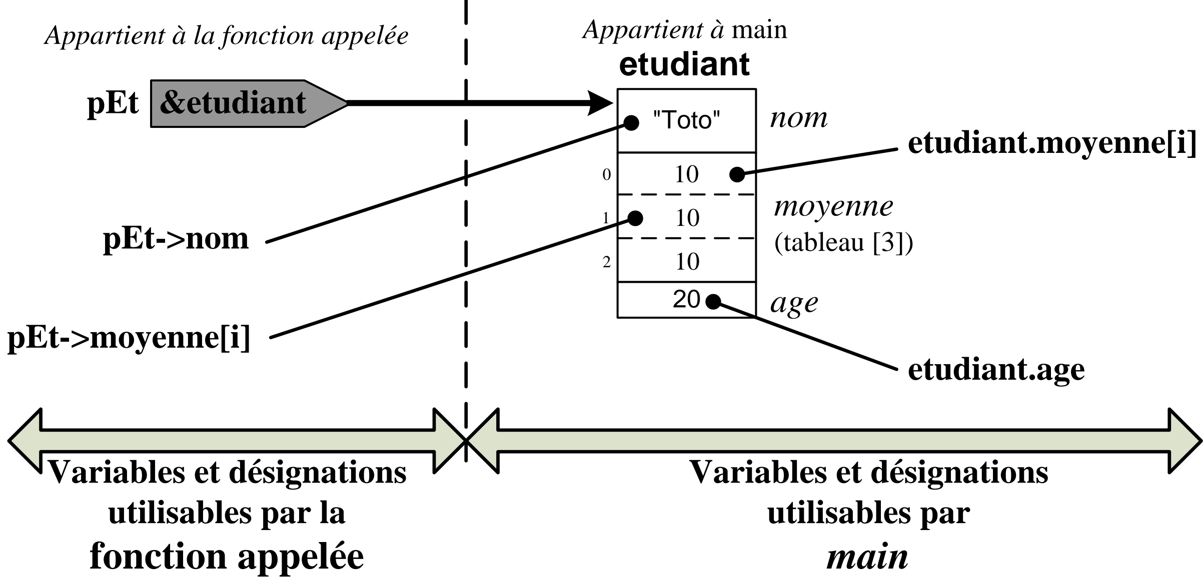


Figure 16 --8 : Accès à un champ de structure (direct ou indirect)

16.6. Bilan sur les ressemblances tableau - variable structurée

Une variable structurée partage un certain nombre de caractéristiques avec un tableau :

- elle est constituée de **plusieurs** « cases », appelées **champs** (au lieu d'« éléments », réservé à un tableau). Chaque case est repérée par son **nom de champ** au lieu d'un indice.
- elle est impossible à traiter dans son intégralité : il faut préciser le traitement à faire sur **chaque champ pris séparément** (affichage, saisie, test…). Une exception : l'affectation est possible entre variables structurées, alors qu'elle est interdite entre tableaux.
- Comme le tableau, une structure est en général transmise en argument à une fonction par l'intermédiaire de son **adresse**, et non par recopie de de sa valeur. La transmission par valeur est autorisée (elle est impossible pour un tableau), mais elle n'est pas conseillée.

16.7. Structures imbriquées

Les champs d'une structure peuvent être de tout type, y compris de type structuré. Voici par exemple un type structuré *T_date* qui est utilisé pour définir le champ *naissance* d'un autre type structuré *T_pers* :

```
/* Création de types structurés imbriqués :*/
 typedef struct      /* modèle de structure T_date */
       {
       short int jour ;
       short int mois ;
       short int annee ;
       } T_date ;

 typedef struct      /* modèle de structure T_pers (utilise un autre type structuré créé au préalable) */
       {
       char nom[20] ;
       char prenom[20] ;
       T_date naissance ; /* ce champ est de type structuré */
       } T_pers ;
```

jour
mois
annee

nom
prenom
naissance
jour
mois
annee

/* Définition de variables : */

```
static T_ pers etudiant[30] ;          /* tableau de structures T_pers */
static T_ pers prof = { "Pere", "Noel", { 24, 12, 1900 } } ;
                              /* notez la façon dont se fait l'initialisation d'un champ structuré */
```

L'accès à un champ de type structuré se fait grâce à une double utilisation de l'opérateur "." :

prof.naissance.annee = 1950 ;

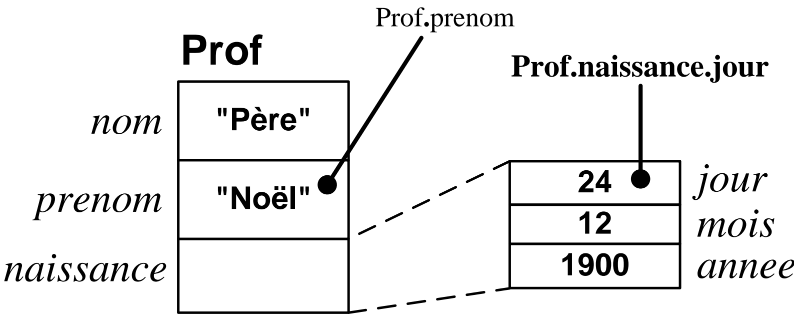


Figure 16 --9 : Structures imbriquées

Exemple 93. Accès aux champs d'un champ structuré :

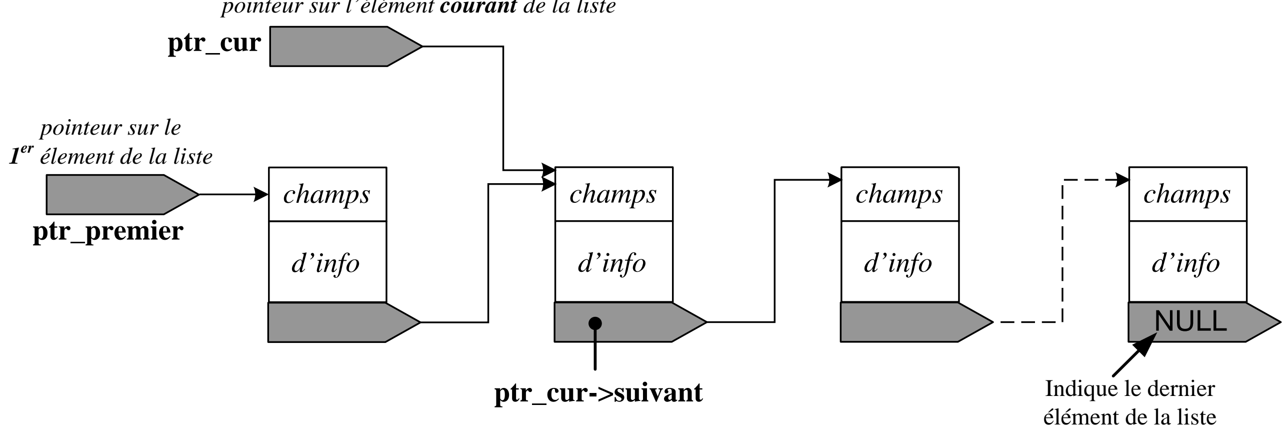
```
if ( etudiant[i].naissance.mois == 11 )
printf("L'etudiant %s %s est Scorpion", etudiant[i].nom, etudiant[i].prenom);
```

16.8. Listes chaînées : application des pointeurs

Parmi les champs d'une structure peut figurer un **pointeur**.

Dans une **liste chaînée** (concurrente du tableau de structures), une variable de type structuré contient, outre des informations diverses, **l'adresse d'une autre variable du même type structuré** sous la forme d'un **pointeur**. Le passage d'un élément à l'autre de la liste s'effectue donc à l'aide de ce pointeur. Au lieu de balayer les élements d'un tableau en incrémentant un indice, on se sert du « pointeur sur l'élément suivant » d'un élement de la liste.
Cette méthode, plus lourde à mettre en œuvre, permet d'ajuster en temps réel la taille de la liste, alors qu'elle est figée dans le cas d'un tableau.

Le mécanisme de fonctionnement des listes chaînées est illustré ci-dessous :



`ptr_cur = ptr_cur->suivant;` *permet de changer d'élément courant (de passer à l'élément suivant dans la liste)*

Mettons en évidence les adresses pour illustrer le fonctionnement des pointeurs :

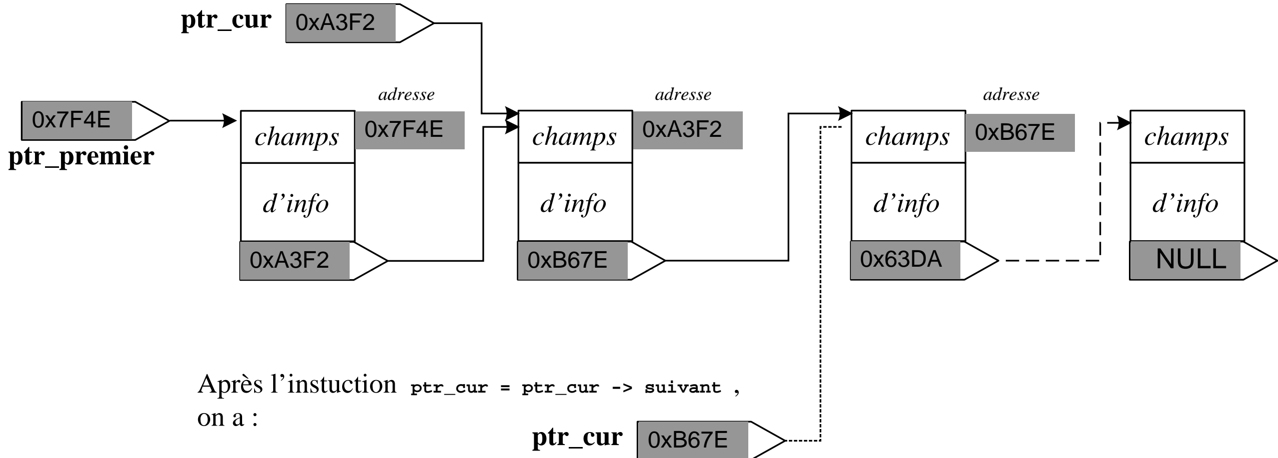


Figure 16 --10 : Liste chaînée

Dans cet exemple, la présence d'un seul pointeur (sur l'élément **suivant**) ne permet de parcourir la liste que dans un seul sens. Il suffit de rajouter un deuxième pointeur (sur l'élément **précédent**) pour pouvoir la parcourir dans les deux sens.

La liste chaînée est une application typique de l'allocation dynamique : la réservation des variables de la liste est réalisée au fur et à mesure des besoins. Les éléments de la liste ne sont pas consécutifs en mémoire.

Voici un modèle de structure qui permet de définir une liste chaînée du type "arbre généalogique" (remarquez l'auto-imbrication du type structuré : on dit que la structure est récursive) :

```
typedef struct t_enfant
        {
        char nom[20] ;
        char prenom[20] ;
        short int an_naiss ;
        struct t_enfant * Mere;
        struct t_enfant * Pere;
        } T_ENFANT ;
```

On remarque que le type structuré est repéré par deux noms différents :

- le nom *T_ENFANT* sera utilisé dans toute la suite du programme (notamment pour définir les variables) ;
- le nom ***struct t_enfant*** (synonyme de *T_ENFANT*) est créé seulement pour permettre la récursivité : on ne peut pas définir les champs *Pere* et *Mere* à partir du type *T_ENFANT*, car celui-ci ne sera connu du compilateur que quelques lignes plus loin ...

Cet exemple s'écarte du mécanisme de liste chaînée illustré par le schéma ci-dessus, dans la mesure où un élément de la liste permet d'obtenir **deux** autres éléments (ses parents). Il s'agit d'un **arbre binaire**.

16.9. Allocation dynamique de structures

L'allocation dynamique de variables structurées est utile :

- pour les listes chaînées ; on alloue une seule variable structurée à la fois.
- quand on veut créer un tableau de structures ; ceux-ci sont consommateurs en mémoire et il est utile d'optimiser leur taille en fonction de besoins qui ne sont souvent connus qu'à l'exécution.

Exemple 94. Allocation dynamique de structure / de tableau de structures

```
typedef struct      {
                    char nom[20];
                    float taille ;
                    short int age ;
                    } T_ETUD ;
//------------------------------------------------------------------------

void main(void)
 {
 T_ETUD* pUnEtudiant ;          /* pointeur pour la variable structurée */
 T_ETUD* ListeEtudiant ;        /* pointeur pour le TABLEAU de structures */
 short int i, nb_etud = 0 ;

 printf("Tapez le nombre d'étudiants :");
 scanf("%hd", &nb_etud );

/* ALLOCATION D'UNE VAR. STRUCTUREE : */
 pUnEtudiant = (T_ETUD*) malloc( sizeof(T_ETUD) );
 if (pUnEtudiant==NULL)    {
                           printf("\n Allocation dynamique impossible !");
                           exit(1) ;      // on quitte le programme
                           }

/* ALLOCATION D'UN TABLEAU DE STRUCTURES : */
 ListeEtudiant = (T_ETUD*) malloc( nb_etud * sizeof(T_ETUD) );
 if (ListeEtudiant==NULL) {
                           printf("\n Allocation dynamique impossible !");
                           exit(1) ;      // on quitte le programme
                           }

/* UTILISATION DE LA VARIABLE : via un pointeur uniquement */
 pUnEtudiant -> taille = 1.75 ;

 /* UTILISATION DU TABLEAU : comme tous les tableaux "normaux" */
 for (i=0 ; i<nb_etud ; i++)     ListeEtudiant[i].age = 18 ;

 /* LIBERATION MEMOIRE : */
 free(pUnEtudiant) ;
 free(ListeEtudiant);
 }
```

16.10. Un exemple en plusieurs fichiers avec fichier en-tête

Ou comment écrire un vrai programme…

FICHIER EN-TETE *GLOBAL.H*

```
// Fichier en-tête global.h
// (mettre ici la date, l'auteur, la description du contenu…)
/* Constantes : */
        #define NB_CAR_NOM      20
        #define ANNEE_COURANTE  2007
/* Création de types (structurés, énumérés...) : */
        typedef struct
                {
                char nom[NB_CAR_NOM] ;
                short int an_naiss ;      // etc.
                } T_perso ;
/* Prototypes de fonctions : */
        void remplir_personne( T_perso *ptoto ) ;
        short calculer_age( T_perso *ptoto ) ;
/* Déclaration var. globale (celles-ci sont à éviter !) : */
        extern char Bidon ;      // attention : l'initialisation ne se fait jamais dans le fichier en-tête !
```

FICHIER SOURCE *MAIN.C*

```
#include "global.h"

char Bidon = 'A' ;      // définition variable globale (si nécessaire) à faire ici dans main. Initialisation optionnelle.
//--------------------------------------------------------------------------
void main( void )
  {
  static T_perso Claude = { "Cloclo", -1 };   // initialisation de la structure avec valeurs de test

  remplir_personne( &Claude );     // passage par adresse toujours préférable avec une structure.
  age = calculer_age(&Claude );
  }
```

FICHIER SOURCE *FCT.C*

```
#include <stdio.h>
#include "global.h"
//--------------------------------------------------------------------------
void remplir_personne( T_perso *ptoto )      // le contenu de cette fonction est juste pour l'exemple.
  {
  ptoto->an_naiss = 2000 ;     // accès aux champs de la structure avec -> (car ptoto est un pointeur)
  scanf("%s",ptoto->nom );     // pas de & avec une ch. de caract. (le nom d'une chaîne est déjà une adresse…)
  }
//--------------------------------------------------------------------------
short calculer_age( T_perso *ptoto )
  {
  return (ptoto->an_naiss - ANNEE_COURANTE);     // accès aux champs de la structure avec ->
  }
```

17 - Les fichiers

Les fichiers permettent de stocker ou de lire des donnéees sur un support permanent tel que le disque dur. Les données sont mémorisées sous la forme d'une suite d'octets.

On peut choisir entre deux modes d'accès : binaire ou texte.

- **<u>fichier rempli en mode binaire</u>** : chaque information est stockée selon les règles de codage imposées par son type. Les données ne peuvent être **lues/écrites que par programme**. La taille du fichier est alors optimale et les données sont facilement lues/écrites en peu d'instructions. Mais on ne peut pas éditer le fichier pour en vérifier ou en modifier le contenu.

  Application : c'est le fichier le plus simple à utiliser pour faire des **sauvegardes de données** entre deux appels d'un programme. Il se prête aussi très bien à la gestion de bases de données.

  Pour lire/écrire dans un fichier binaire, il existe deux fonctions rapides à utiliser (*fread*, *fwrite*). On peut aussi se positionner n'importe où dans le fichier (*fseek*).

- **<u>fichier rempli en mode texte</u>** : chaque information est stockée sous la forme d'une succession de **codes ASCII**. Les données du fichier peuvent être **créées ou consultées par l'utilisateur à l'aide d'un éditeur de texte**, en plus de leur accès par programme. En contrepartie, elles occupent souvent plus de place et sont plus difficiles à manipuler par programme.

  Un fichier texte est un cas particulier de fichier binaire : on peut donc le manipuler en accès binaire, avec les fonctions de lecture/écriture binaires (*fread, fwrite*). Mais on utilise surtout des fonctions d'entrée/sortie dites "formatées" (*fprintf, fgets, fscanf*,...), qui ressemblent à celles dont on dispose pour écrire à l'écran ou lire au clavier.

  Application : le fichier texte est plus long et plus délicat à lire/écrire que le fichier binaire "pur". Mais il présente l'avantage d'être consultable (et modifiable) avec n'importe quel éditeur de texte, ce qui donne plus de marge de manoeuvre à l'utilisateur. C'est le fichier à employer quand l'utilisateur veut voir et manipuler les données du fichier avec un éditeur de texte.

Quel que soit le type d'accès envisagé, binaire ou texte, il faut suivre la même procédure :

- ouvrir le fichier (fonction ***fopen***) ; on indique le nom du fichier et les détails de l'accès envisagé.
- lire ou écrire dans le fichier : on utilise les fonctions autorisées par l'accès choisi (binaire ou texte).
- fermer le fichier (fonction ***fclose***).

17.1. Contenu d'un fichier binaire et de son équivalent texte

<u>Ce paragraphe peut être sauté dans un premier temps</u>. Il permet de mieux comprendre ce que contiennent un fichier binaire et un fichier texte utilisés pour **stocker les mêmes données** : ici un réel, un entier et une phrase.

Exemple 95. Contenu d'un fichier texte et binaire (avec des données identiques)

On veut stocker le réel *double* -1.60217e-19, l'entier *short* -25724 et la chaîne de 5 caractères "bof !", ces trois informations étant contenues dans une variable structurée *var_stru*.

Pour chaque mode d'accès (binaire ou texte), voici l'instruction C à utiliser ainsi que le contenu résultant pour les fichiers binaire et texte.

On rappelle qu'un caractère entre apostrophes est le **code ASCII** du caractère. Par exemple, 'a' est le code ASCII de la lettre *a*, c'est-à-dire l'entier 97 (ou 0x61).

Solution 1 : fichier en accès binaire

```
fwrite ( &var_stru, sizeof(var_stru), 1, fic );
```

<table>
<tr><td>xx</td><td>xx</td><td>xx</td><td>xx</td><td>xx</td><td>xx</td><td>xx</td><td>xx</td><td>yy</td><td>yy</td><td>'b'</td><td>'o'</td><td>'f'</td><td>' '</td><td>'!'</td><td>0</td></tr>
<tr><td colspan="8">Codage IEEE du réel double
(toujours 8 octets, avec une précision 10^{-15})</td><td colspan="2">Code C2 de l'entier (tjrs 2 octets)</td><td colspan="6">Codes ASCII des lettres de la chaîne, y compris le caractère nul de fin.</td></tr>
</table>

Solution 2 : fichier en accès texte

```
fprintf( fic, "%.5le %hd %s", var_stru.reel, var_stru.entier, var_stru.chaine);
```

<table>
<tr><td>'-'</td><td>'1'</td><td>'.'</td><td>'6'</td><td>'0'</td><td>'2'</td><td>'1'</td><td>'7'</td><td>'e'</td><td>'-'</td><td>'0'</td><td>'1'</td><td>'9'</td><td>' '</td><td>'-'</td><td>'2'</td><td>'5'</td><td>'7'</td><td>'2'</td><td>'4'</td><td>' '</td></tr>
<tr><td colspan="13">Codes ASCII de la succession des chiffres du réel (taille variable selon l'écriture du réel)</td><td>Espace rajouté</td><td colspan="6">Codes ASCII de la succession des chiffres de l'entier (taille variable)</td><td>Espace rajouté</td></tr>
</table>

Suite du fichier

<table>
<tr><td>'b'</td><td>'o'</td><td>'f'</td><td>' '</td><td>'!'</td></tr>
<tr><td colspan="5">Codes ASCII des lettres de la chaîne, sans le caractère nul</td></tr>
</table>

Taille des fichiers :

- La taille du fichier binaire de la solution 1 est toujours la même (8+2+6=16 octets), **quelles que soient les valeurs du réel et de l'entier**.
- la taille du fichier texte de la solution 2 dépend beaucoup du nombre de chiffres significatifs du réel et de l'entier, et donc des codes formats utilisés dans *fprintf*. Ici, on obtient 26 octets, y compris les deux espaces rajoutés pour séparer les valeurs.

Contenu affiché par un éditeur de texte :

Si on ouvre le fichier **avec un éditeur de texte** quelconque, on obtient bien sûr « n'importe quoi » pour le fichier 1 (il ne contient pas des codes ASCII), et le texte attendu pour le fichier 2 :

Fichier 1

```
bŒ¾Ó¤¼„›bof !
```

Fichier 2

```
-1.602170e-019 -25724 bof !
```

Contenu hexadécimal de chaque fichier

Les octets des deux fichiers valent les valeurs **hexadécimales** suivantes (obtenues par un programme) :

Fichier 1 (16 octets)

14 62 8C BE D3 A4 07 BC 84 9B **62 6F 66 20 21** 0

Fichier 2 (26 octets)

2D 31 2E 36 30 32 31 37 65 2D 30 31 39 20 2D 32 35 37 32 34 20 **62 6F 66 20 21**

☞ On reconnaît dans le fichier 2 les codes de la table des codes ASCII (fournie en annexe) : 0x2D est le code du signe -, 0x20 est le code de l'espace, 0x3*y* est le code du chiffre *y*…

☞ On reconnaît aussi la chaîne "*bof !*", qui est présente à la fin des deux fichiers. Notez que son caractère nul final a été supprimé par *sprintf*, mais pas par *fwrite* (pour *fwrite*, ce caractère nul n'a rien de spécial).

17.2. Ouverture et fermeture d'un fichier : *fopen, fclose*

17.2.1 Définition d'un pointeur de fichier FILE*

Tout fichier du disque dur sera associé, une fois ouvert, à une variable de type *FILE**, qui doit être préalablement définie par :

FILE* fic ;

La variable *fic* est un pointeur sur un objet de type *FILE* (le type *FILE* est un modèle de structure défini dans *stdio.h*) et on l'appelle le « **pointeur de fichier** ».
fic donne accès à diverses informations concernant le fichier, en particulier la position actuelle du « curseur de position » associé au fichier. Le programmeur n'a pas à connaître ces informations qui seront utilisées par les fonctions : il n'a qu'à fournir le pointeur de fichier aux fonctions de manipulation de fichier.

17.2.2 Ouverture d'un fichier

On va demander au programme d'associer le pointeur de fichier défini précédemment à un fichier physique. Pour cela, il faut indiquer le nom du fichier (tel qu'il apparaît sur le support physique), le **mode d'accès** choisi (binaire ou texte), et la **nature du travail** qu'on désire faire avec le fichier (lecture, écriture). Ces informations sont fournies à la fonction *fopen*, qui va ouvrir le fichier et renvoyer la valeur à ranger dans le pointeur de fichier.
Voici un exemple d'utilisation (incomplet) de *fopen* pour ouvrir en écriture binaire (*wb= Write Binary*) le fichier binaire *donnees.dat* et associer à ce fichier le pointeur de fichier *fic* :

fic = fopen("donnees.dat", "wb") ; /* le fichier est créé ou écrasé s'il existe */

Le prototype de *fopen* est :

FILE* fopen(char* filename, char* mode) ;

filename est le nom physique du fichier, chemin éventuellement inclus. Par exemple, "data.txt" ou "c:****tmp****data.txt". **<u>Notez le \\ dans le chemin</u> !**
mode est une chaîne de 2 ou 3 caractères (par exemple, "wb" ou "rt"), qui indique le mode d'accès choisi et le type de travail possible, choisis parmi :

- **r (*Read*)** : **lecture** seulement. Le curseur est positionné **au début du fichier**. Le fichier doit déjà exister ;
- **w (*Write*)** : **écriture** seulement. Le curseur est positionné **au début du fichier**. Celui-ci est **créé s'il n'existe pas, son ancien contenu est écrasé s'il existe** ;
- **a (*Append*)** : **écriture** seulement. Le curseur est positionné **à la fin du fichier**. Celui-ci est créé s'il n'existe pas. C'est le mode à utiliser pour compléter un fichier existant ;
- r+ : écriture et lecture. Le curseur est positionné au début du fichier. Le fichier doit déjà exister. Ce mode est peu utilisé.
- w+ : écriture et lecture. Le curseur est positionné au début du fichier. Le fichier est créé s'il n'existe pas, son ancien contenu est écrasé s'il existe. Ce mode est peu utilisé.
- a+ : écriture et lecture. Le curseur est positionné à la fin du fichier. Celui-ci est créé s'il n'existe pas. Ce mode est peu utilisé.
- **b (Binaire) ou t (Text)** : cette lettre placée en deuxième position indique si le fichier doit être accédé en mode texte ou en mode binaire.

mode vaudra par exemple **"wt"** (fichier texte créé avec accès en écriture seulement) ou **"a+b"** (fichier binaire avec accès en lecture et écriture).

La fonction *fopen* retourne l'adresse de la structure *FILE* associée au fichier. Elle renvoie *NULL* si elle ne parvient pas à ouvrir le fichier.

Le test de la valeur renvoyée par *fopen* est indispensable pour prévenir les erreurs : fichier inexistant, support physique défectueux ou saturé, nombre excessif de fichiers ouverts…

Exemple 96. Appel complet de la fonction *fopen*

```
fic = fopen( "data.txt", "rt" );          /*ouverture en lecture texte */
if (fic==NULL) { printf("Pb d'ouverture fichier !"); exit(0) ; }
```

Voici un exemple de création d'un fichier **texte en écriture** seulement (pour créer un fichier binaire, il suffit de remplacer "wt" par "wb" dans l'instruction *fopen*) :

Exemple 97. Ouverture d'un fichier texte (création)

```
void main(void)
{
char Nom[13];      /* taille à augmenter si on veut pouvoir mettre le chemin du fichier */
FILE* fic ;

printf("Entrez le nom du fichier (8 lettres maxi + extension): ") ;
scanf("%12s", Nom) ;
fic = fopen( Nom, "wt");       /* le fichier (texte) est créé en vue d'une écriture en accès texte */
if ( fic==NULL )
     { printf("\n\n Ouverture du fichier impossible !") ; exit(0) ; }
…
fclose( fic );                 /* fermeture du fichier quand il n'est plus utile */
}
```

17.2.3 Fermeture d'un fichier

Il est indispensable de fermer un fichier avant la fin du programme qui l'utilise pour éviter la perte de données. Cette fermeture se fait très simplement à l'aide de la fonction *fclose*, à laquelle on fournit le pointeur du fichier à fermer :

fclose(fic) ;

☺ Il est conseillé de ne pas laisser le fichier ouvert si on ne l'utilise plus pendant « un certain temps ». Il est plus prudent de le fermer et de le rouvrir ensuite.

17.3. Entrées/sorties en accès binaire

On peut toujours, quel que soit le type de fichier (binaire ou texte), utiliser l'accès en mode binaire. Il se fait par les fonctions ***fread*** et ***fwrite***, qui permettent de lire ou d'écrire des suites d'octets. Dans un accès non séquentiel (plus rare), on peut se positionner n'importe où dans le fichier avec la fonction ***fseek***.

Il existe aussi des macros **getc** et **putc** (ou des fonctions équivalentes **fgetc** et **fputc**) qui permettent de lire ou d'écrire un caractère dans un fichier, qu'il soit binaire ou texte. Cet accès octet par octet est toujours plus long : on privilégie donc l'accès "groupé" par *fread* et *fwrite*, quitte à créer un "buffer" de données comme dans Exemple 98. et Exemple 99.

17.3.1 Ecriture et lecture en accès binaire : *fwrite*, *fread*

Les fonctions *fread/fwrite* attendent, outre le pointeur du fichier, le nombre d'éléments à lire/écrire, la taille en octet d'un élément et l'adresse du « *buffer*[10] » (zone mémoire) où ranger/trouver les données à lire/écrire. Ce *buffer* est en général un tableau, mais il peut aussi être une variable simple ou structurée (voir **Erreur ! Source du renvoi introuvable.**), surtout en accès non séquentiel avec positionnement préalable par *fseek*.

Très important : *fread* et *fwrite* renvoient le **nombre d'éléments effectivement lus ou écrits**. Si la valeur renvoyée par *fread* diffère du nombre d'éléments à lire, c'est que **la fin du fichier a été rencontrée** lors de la lecture. On se sert donc de la valeur renvoyée par *fread* pour savoir si la fin du fichier est atteinte.

Les prototypes de *fread* et *fwrite* sont les suivants :

int fread(void* adr_buffer, int taille_element, int nb_elements, FILE* fic);
int fwrite(void* adr_buffer, int taille_element, int nb_elements, FILE* fic);

Voici un exemple d'écriture de données dans un fichier binaire (en l'occurence, un tableau de 5 entiers, écrit deux fois pour les besoins de l'exemple), et le programme de lecture correspondant :

Exemple 98. Sauvegarde de données dans un fichier binaire

/ La relecture du fichier sera effectuée dans le programme de l'exemple suivant. */*

```
#define NB_ELTS 5
void main (void)
{
FILE* fic ;
short int tablo[NB_ELTS] = {1,2,3,4,5 } ;
/* Ouverture du fichier (en écriture binaire) : */
fic = fopen( "exemple.dat", "wb") ;
if ( fic==NULL )  { printf("Ouverture du fichier impossible !"); exit(0); }
 /* Ecriture dans le fichier (ici, deux fois la même donnée, de deux façons différentes) : */
/* Voici 2 façons différentes de stocker un tableau (la 1ère est plus claire) : */
fwrite ( tablo, sizeof(short int), NB_ELTS, fic );
        /* on stocke NB_ELTS éléments de taille fournie par sizeof */
fwrite ( tablo, 1, sizeof(tablo), fic );
        /* on stocke un nombre d'octets égal à sizeof(tablo) */
 /* Fermeture du fichier : */
fclose( fic ) ;
}
```

[10] *buffer* = « tampon » en français = zone mémoire « à tout faire »

Exemple 99. Lecture (séquentielle) du fichier binaire précédent

/ On suppose ici que le nombre de valeurs à lire est inconnu. Sinon, le programme de lecture est l'exact* ***symétrique*** *du programme d'écriture précédent, avec* fread *au lieu de* fwrite *et ouverture par* fopen *en "rb" au lieu de "wb". */*

```
#define TAILLE_BUF 4          /* valeur quelconque (en général, beaucoup plus grande) */

void main (void)
 {
 FILE* fic ;
 short int buffer[TAILLE_BUF];      /* ce tableau mémorisera les valeurs lues dans le fichier */
 short int i, nb_val_lues = TAILLE_BUF ;

 /* Ouverture du fichier (en lecture binaire) : */
 fic = fopen( "exemple.dat", "rb") ;
 if ( fic==NULL ) { printf("Ouverture du fichier impossible !"); exit(0); }

/* Lecture dans le fichier : */
 printf("\n Liste des valeurs lues : \n");
 /*Remplissage du buffer et traitement, autant de fois que nécessaire jusqu'à la fin fichier : */
 while ( nb_val_lues == TAILLE_BUF )       /* vrai tant que fin du fichier non atteinte */
      {
      nb_val_lues = fread( buffer, sizeof(short int), TAILLE_BUF, fic);
      /* Traitement des valeurs stockées dans le buffer (ici, un simple affichage) : */
      for (i=0; i<nb_val_lues; i++)   printf( "%hd", buffer[i] );
      }
 /* Fermeture du fichier : */
 fclose( fic ) ;
 }
```

Ce programme affiche : 1234512345

17.3.2 Lecture non séquentielle : positionnement par *fseek*

Le programmeur qui veut lire les données d'un fichier séquentiellement, du premier au dernier octet, utilise la lecture séquentielle par *fread* seul. Mais s'il veut se positionner **en un point quelconque du fichier** sans lire ce qui précède, afin d'accéder à une donnée particulière, la lecture séquentielle ne convient plus : il faut un accès direct (ou non séquentiel). Celui-ci est obtenu en utilisant la fonction de positionnement ***fseek*** avant d'effectuer la lecture par ***fread***.

☺ La lecture non séquentielle doit être évitée autant que possible.

En accès non séquentiel, le programmeur utilise les deux fonctions ***fseek*** et ***ftell*** :

- ***fseek*** positionne le « curseur de fichier » à une distance **comptée en octets** à partir du début, de la fin ou de la position courante du fichier. *fseek* permet donc d'indiquer « où on veut aller ».
- ***ftell*** renvoie la position courante du « curseur de fichier », comptée en octets à partir du début du fichier. *ftell* permet donc de « savoir où on est ». Elle est beaucoup moins utile que *fseek*.

Le prototype de *fseek* est :

int fseek (FILE *Stream, long Offset, int Origin);

Offset est le nombre d'octets du déplacement, compté algébriquement à partir de *Origin*.
Origin est une constante qui vaut SEEK_SET (« à partir du début du fichier ») ou SEEK_END (« à partir de la fin du fichier ») ou, plus rarement, SEEK_CUR (« à partir de la position courante »).

Exemple 100. Calcul de la taille d'un fichier (supposé fermé) : fseek, ftell

```
long int taille_fichier( char* nom_fichier )
 {
 FILE* fic ;
 long int longueur_fichier ;

 /* Ouverture du fichier : */
 fic = fopen( nom_fichier, "rb") ;
 if ( fic==NULL ) { printf("Ouverture du fichier impossible !"); exit(0); }

 /* Calcul de la longueur du fichier : */
 fseek ( fic, 0, SEEK_END );          /* on se place à 0 octets de la fin du fichier (SEEK_END) */
 longueur_fichier = ftell (fic);      /* fournit la taille du fichier, car on est à la fin du fichier */

 /* Fermeture du fichier et renvoi de la valeur calculée: */
 fclose( fic ) ;

 return longueur_fichier ;
 }
```

Dans l'exemple suivant, le terme « enregistrement » désigne l'ensemble des informations relatives à un objet, en général (comme ici) une variable structurée complète. Le fichier ne contient que des enregistrements de formats identiques.

Exemple 101.   Accès non séquentiel à un fichier binaire (enregistrements)

```
/* Création d'un type structuré (ici pour le test), adapté aux données à créer/à lire : */
 typedef struct {  char nom[10]; short int an_nais;   } T_PERS ;

void main( void )
 {
 FILE* fic ;
 T_PERS pers ;
 T_PERS tab_pers[4]= {
      {"Luc", 1970}, {"Li", 1920}, {"Zoé", 1960}, {"Léa", 1975} };
 short int num=-1, err, nb_enreg;
 long int offset ;
 /* Création du fichier de données de test (il contient 7 enregistrements) : */
 fic = fopen( "exemple.dat, "wb") ;
 if (fic==NULL)  { printf("Ouverture fichier impossible !"); exit(0); }

 fwrite( tab_pers, sizeof(T_PERS), 4, fic); /* on crée 7 enregistrements (en 2 fois : 4+3) */
 fwrite( tab_pers, sizeof(T_PERS), 3, fic);
 fclose( fic ) ;
 /* Le fichier de données est maintenant enregistré sur le disque dur. */

/* Lecture d'un seul enregistrement "au milieu" du fichier : */

 /* On détermine d'abord le nombre total d'enregistrements stockés dans le fichier : */
 nb_enreg = taille_fichier("exemple.dat") / sizeof(T_PERS);
  /* la fonction taille_fichier est celle de l'exemple précédent */
  /* il faudrait vérifier que nb_enreg est non nul ...*/

 /* Ouverture du fichier : */
 fic = fopen( "exemple.dat, "rb") ;
 if (fic==NULL) { printf("Ouverture fichier impossible !"); exit(0); }

 /* On saisit au clavier le numéro de l'enregistrement à lire (saisie protégée) : */
 printf("Quel enregistrement (entre 0 et %hd) voulez-vous lire ? ", nb_enreg-1);
 while ( (scanf("%hd",&num)==0) || (num<0) || (num>=nb_enreg) )
      {
      printf("\n Tapez un numéro entre 0 et %hd ! ", nb_enreg-1);
      fflush(stdin);
      num=-1;
      }
 /* On positionne le pointeur de fichier sur l'enregistrement voulu (avec fseek) :  */
 offset = num * sizeof(T_PERS) ;
 err = fseek( fic, offset, SEEK_SET ) ;       /* SEEK_SET = à partir du début du fichier */
 if ( err != 0) { puts("Pb de positionnement par fseek !"); exit(0); }

 /* On lit un seul enregistrement, qu'on range dans la variable structurée pers : */
 if ( fread( &pers, sizeof(T_PERS), 1, fic) == 1 )
      printf("\n Enregistrement %hd : %s %hd\n", num, pers.nom, pers.an_nais );
 else
      { puts("Pb de lecture fichier !"); exit(0); }

 fclose( fic ) ;
 }
```

17.4. Entrées/sorties formatées en mode texte

Rappelons qu'un fichier texte peut être lu/écrit avec un éditeur (il est composé de caractères ASCII).

Les exemples qui suivent vont montrer qu'un programme qui manipule un fichier en accès texte est plus délicat et plus long qu'en accès binaire. Au lieu d'une ligne *fwrite* qui mémorise dans le fichier **tout un tableau** d'un coup (quelle que soit sa taille et la nature de ses éléments), il faut écrire **élément après élément**, voire champ après champ si les éléments sont de type structuré. C'est le prix à payer pour disposer d'un fichier consultable directement par l'utilisateur au moyen d'un éditeur...

Les fonctions d'entrées/sorties formatées dans un fichier sont identiques aux entrées/sorties conversationnelles à cette différence qu'elles possèdent un argument supplémentaire : le pointeur du fichier.

La position de lecture/écriture est la position courante du « curseur » dans le fichier. Ce curseur de fichier (ainsi nommé par analogie avec l'écran) se déplace automatiquement lors d'une lecture ou écriture.

Les principales fonctions disponibles sont les suivantes :

fprintf(fichier, codes formats et texte, valeurs);
fscanf (fichier, codes formats, adresses);

fgets(ptr_chaine, longueur_max, fichier);   /* lecture d'une chaîne */
fputs(ptr_chaine, fichier);   /* écriture d'une chaîne */

fgetc(fichier);   /* lecture d'un caractère */
fputc(fichier, caractere);   /* écriture d'un caractère */

Nous allons voir un exemple d'écriture/lecture avec *fprintf/fscanf*, puis un exemple de lecture avec *fgets*, plus simple et plus sûre.

☺ On a toujours intérêt à ramener le problème de lecture de fichier à un problème de traitement de chaîne de caractères (pour lequel nous disposons de nombreuses fonctions de recherche). C'est le rôle de la fonction ***fgets***.

Par conséquent, **on préfèrera toujours l'utilisation de *fgets*, suivie éventuellement par *sscanf*** (ou ***strtok*** pour la recherche), à celle de *fscanf* (plus délicate). En particulier, la gestion de la fin de fichier est beaucoup plus simple avec *fgets*.

17.4.1 Ecriture dans un fichier texte : *fprintf*

Exemple 102. Ecriture dans un fichier en mode texte

```
void main (void)
{
FILE* fic ;
short int i, tab[5] = {1,2,3,4,5 } ;
/* Ouverture du fichier (en écriture texte): */
fic = fopen( "resultat.dat", "wt");
if (fic==NULL)  {printf("\n\n Ouverture du fichier impossible !"); exit(0); }
/* Ecriture dans le fichier : */
fprintf( fic, "Nombre d'elements : %hd", 5 );
for (i=0; i<5; i ++)    fprintf( fic, "\n element %hd : %hd", i, tab[i] );
        /* la disposition dans le fichier texte sera comme à l'écran avec printf */
/* Fermeture du fichier : */
fclose( fic ) ;
}
```

17.4.2 Lecture dans un fichier texte par *fscanf* (déconseillé)

☞ Conseil (rappel) : <u>on préfèrera l'utilisation de **fgets**, suivie éventuellement par **sscanf**</u>, à celle de *fscanf* (plus délicate). A moins d'avoir absolument besoin de *fscanf*, vous pouvez donc passer au paragraphe suivant et ses exemples.

Pour la lecture avec *fscanf*, une précaution supplémentaire s'impose : il faut éviter de lire au-delà de la limite du fichier (même si on croit en connaître la taille !).

On utilise pour cela la fonction ***feof*** (*End Of File*), qui renvoie 0 tant que la fin du fichier n'est pas atteinte. Cette fonction permet de lire un fichier dont on ignore la taille.

Exemple 103. Lecture du fichier texte précédent avec *fscanf* (déconseillé)

```
void main (void)
{
FILE* fic ;
short int i, i_lu, n, tab[5];

/* Ouverture du fichier (en lecture texte) : */
fic = fopen( "resultat.dat", "rt") ;
if ( fic==NULL ) { printf("\n\n Ouverture fichier impossible !"); exit(0); }

/* Lecture du fichier (remarque : feof(fic) vaut 0 (faux) si fin de fichier non atteinte) : */
fscanf( fic, "Nombre d'elements : %hd", &n );
for ( i=0 ; (i<n) && (!feof(fic)) ; i ++ )
     fscanf( fic, "\n element %hd : %hd", &i_lu, &tab[i] );

/* Traitement des données lues : */
printf("\n\n Voici les valeurs lues dans le fichier :");
for (i=0; i<n; i ++)     printf( "\n element %hd : %hd", i, tab[i] );

/* Fermeture du fichier : */
fclose( fic ) ;
}
```

Pour *fscanf* comme pour *scanf*, si une chaîne de caractères figure entre les codes formats, elle doit être retrouvée très précisément à la lecture, sous peine de mauvais fonctionnement.

On peut aussi indiquer à *scanf* ou *fscanf* que des éléments non significatifs doivent être ignorés à la lecture grâce au symbole ***** placé entre le % et la lettre du format. Cela permet de « sauter » les chaîne de caractères non désirées, sans les mémoriser.

Par exemple, la lecture d'une ligne du fichier précédent, toujours de la forme `element 6 : 123`, devient si on veut mémoriser seulement la dernière valeur (123 ici) :

fscanf(fic, "%*s%*d%*s%hd", &tab[i]);

Rappelons que les espaces, les tabulations et les passages à la ligne sont considérés comme des séparateurs par *scanf* et *fscanf*, et qu'il sont **automatiquement sautés** lors de la lecture d'une valeur numérique ou d'une chaîne de caractères (mais pas d'un caractère !!).

Remarquez l'utilisation de la fonction *feof* dans l'exemple précédent : *feof(fic)* vaut 0 (faux) si la fin de fichier n'est pas atteinte. La lecture du fichier peut aussi s'écrire :

```
while( !feof(fic) )
{
fscanf( fic, "%*s%hd", &i );
fscanf( fic, "%*s%hd", &tab[i] );
}
```

17.4.3 Lecture d'un fichier texte par *fgets* combiné à *sscanf* (conseillé)

Quand on manipule un fichier texte, tapé par exemple avec un éditeur, il arrive souvent qu'on veuille analyser le contenu du fichier **mot après mot, ligne après ligne** (comme le fait un compilateur de langage quelconque). Il existe pour cela beaucoup de fonctions de traitement de chaînes très utiles, en particulier pour les recherches et les comparaisons (cf paragraphe 12.3.2 du chapitre "Chaînes de caractères"). Encore faut-il que le texte à analyser existe sous forme de chaînes de caractères. C'est l'application rêvée pour la fonction ***fgets***, qui **lit une ligne entière du fichier et la range dans une chaîne**.

Comme *fgets* lit une ligne entière, elle est aussi utile à ceux qui cherchent les fins de ligne. Mais de façon générale, si on la combine avec *sscanf* (ou *strtok* ou ...), elle se substitue complètement à *fscanf*, tout en étant plus simple.

La syntaxe d'utilisation de *fgets* est :

ptr = fgets(chaine, taille_chaine, ptr_fic **);**

ou plus souvent

while (**fgets(** chaine**,** taille_chaine, ptr_fic **)** != NULL)   { … }

fgets lit au maximum *taille_chaine-1* caractères de la ligne du fichier, et s'arrête au caractère fin de ligne s'il est rencontré (attention : il est aussi copié dans la chaîne). Elle renvoie *NULL* si la fin du fichier est rencontrée ou si un problème se produit. Le test de la valeur renvoyée est donc indispensable, ne serait-ce que pour savoir si on a atteint la fin du fichier.

Voici un exemple d'utilisation de *fgets*, qui copie à l'écran le contenu d'un fichier texte.

Exemple 104. Copie à l'écran d'un fichier texte (lecture des lignes par *fgets*)

```
char ligne[81] ;
/* le fichier est supposé ouvert en mode "rt" */
while ( fgets(ligne, 81, fichier) != NULL )          /*fin de fichier non atteinte*/
     printf("%s", ligne ) ; /*affiche la ligne lue à l'écran*/
```

☺ Conseil (rappel) : **<u>préférez toujours l'utilisation de *fgets*</u>**, suivie éventuellement par ***sscanf*** (ou *strtok*), à celle de *fscanf*. Entre autres avantages, la gestion de la fin de fichier est beaucoup plus simple avec *fgets*.

La fonction *fgets* peut être complétée par une fonction de **lecture dans une chaîne** nommée *sscanf* (*String scanf*), ou par des fonctions de recherche comme *strtok*. L'intérêt de *fgets* est justement de permettre l'utilisation des nombreuses fonctions de traitement de chaînes de caractères pour l'exploitation d'une ligne du fichier.

La combinaison de *fgets* avec *sscanf* remplace complètement *fscanf*, et permet de mieux gérer les problèmes de saisie. D'ailleurs, *sscanf* se combine aussi bien avec *fgets* qu'avec *gets* (lecture d'une ligne au clavier).
sscanf fait le même travail que *scanf* ou *fscanf*, mais au lieu de travailler sur le tampon de saisie du C ou sur un fichier, elle lit dans la chaîne de caractères qu'on lui fournit en paramètre. Ce paramètre supplémentaire est la seule différence avec *scanf*.

Pour la lecture de données dans un fichier :

- on lit une ligne du fichier avec *fgets* et on la mémorise dans une chaîne de caractères, qu'on appelle par exemple *ligne* ; on est ensuite ramené à un traitement de chaîne de caractères.
- on isole le contenu utile de *ligne* avec *sscanf*, en utilisant les codes formats qui conviennent (comme pour *scanf*). On peut aussi utiliser des fonctions de recherche comme *strtok* si la ligne comporte des séparateurs connus.

Exemple 105. Lecture de fichier texte avec *fgets* + analyse par *sscanf*

Chaque ligne du fichier de données *essai.dat* a le **format imposé** suivant :

un nom (**sans espaces**) suivi par une date de naissance de la forme « 12 janvier 1970 ».

La première ligne vaut par exemple : "**Anne-Laure 2 septembre 1980**".

On choisit ici de mémoriser les données lues dans le fichier dans une **variable structurée** (sinon, voir Exemple 106.).

```
/* Création d'un type structuré adapté aux données : */
 typedef struct     {
                    char nom[20];
                    short int jour;
                    char mois[10];
                    short int annee;
                    } T_ANNIV;

 void main(void)
 {
 FILE* fic ;
 char ligne[81];
 T_ANNIV pers;

 /* Ouverture du fichier (en lecture texte) :  */
 fic = fopen("essai.dat", "rt") ;
 if (fic==NULL) { puts("Pb d'ouverture du fichier essai.dat !"); exit(0); }

 /* Lecture du fichier par fgets+sscanf  (la fin de fichier est atteinte quand fgets renvoie NULL) : */
 while ( fgets( ligne, 81, fic) != NULL )
   {
   if (sscanf(ligne,"%s%hd%s%hd", pers.nom, &pers.jour, pers.mois,
      &pers.annee) == 4)              /* on s'assure de la validité des 4 données */
      {
      printf("\n nom: %s ", pers.nom ) ;     /* traitement = simple affichage ici */
      printf("\n date naiss: %2hd %s %hd\n",pers.jour, pers.mois, pers.annee);
      }
   else
      puts("Pb de lecture avec sscanf (format de ligne incorrect ?)") ;
   }
 fclose( fic ) ;
 }
```

On peut indiquer à *sscanf* que des éléments non significatifs doivent être ignorés à la lecture ; il suffit d'insérer le symbole ***** dans le code format, entre % et la lettre du format. Par exemple : **%*s** pour **ignorer <u>un</u> mot** (un seul !), **%*d** pour ignorer un entier. Cela permet de « sauter » les chaîne de caractères ou les valeurs non désirées, sans les mémoriser (et sans les comptabibliser dans la valeur renvoyée par *sscanf*).

Par exemple, supposons qu'une ligne du fichier précédent contienne des éléments non utiles et se présente comme suit : "**Anne-Laure ; née le 2 septembre 1980**". Le *sscanf* s'écrit alors :

sscanf(ligne,"%s%*s%*s%*s%hd%s%hd", pers.nom, &pers.jour, pers.mois,&pers.annee)

Rappelons que les espaces, les tabulations et les passages à la ligne sont considérés comme des séparateurs par *scanf* et ses dérivés, et qu'il sont **automatiquement sautés** lors de la lecture d'une valeur numérique ou d'une chaîne de caractères (mais pas d'un caractère !!).

☺ La lecture d'un fichier texte est délicate : pour la simplifier, il suffit souvent de s'imposer un **<u>format de ligne bien choisi</u>**, le plus simple possible.

Dans l'Exemple 106. suivant, le fichier texte à lire dispose d'une **en-tête de longueur variable** dont chaque ligne commence par $. Voici un exemple pour ce fichier :

```
$ Fichier TEXTE de test
$ pour lecture par fgets + sscanf

              Ariane  - age: 21 ans - taille: 1.61 m
                Chloé - age: 23 ans - taille: 1.63 m
                 Eve  - age: 25 ans - taille: 1.65 m
                 Iris - age: 29 ans - taille: 1.69 m
```

Et voici l'affichage qui sera obtenu à la fin du programme de l'Exemple 106. :

```
Standard I/O
(Donnees extraites Ligne  1)          Ariane  21   1.61
(Donnees extraites Ligne  2)           Chloe  23   1.63
(Donnees extraites Ligne  3)             Eve  25   1.65
(Donnees extraites Ligne  4)            Iris  29   1.69
```

Exemple 106. Lecture de fichier texte avec *fgets* + analyse par *sscanf*

```
void main(void)
  {
  FILE* fic;
  char ligne[81];                       // pour mémoriser la ligne lue dans le fichier
  short nb_lignes_lues, nb_val_lues ;   // compteurs pour le test
  char nom[15];  short age=21;  double taille=1.61;   // pour mémoriser les données lues

 // ouverture du fichier (qui doit exister) en accès BINAIRE et en lecture :
  fic = fopen("exemple_lecture.txt", "rt");
  if (fic==NULL) {  puts("Pb d'ouverture du fichier exemple.txt !");  exit(0) ;}

 /****************** on saute les lignes d'en-tête (qui commencent toutes par $) ****************/
  // Remarquez que le nombre de lignes de l'en-tête n'est pas connu à l'avance, ni imposé à l'utilisateur.
  while(fgets(ligne, 81, fic)!=0 && ligne[0]=='$');      //  NULL si fin fichier atteinte.

 /********************************** lecture des données ***************************************/
 // pour chaque ligne : on lit toute la ligne (fgets), puis on l'analyse (sscanf).
  nb_lignes_lues = 0 ;                          // pour l'affichage de test

  while (fgets (ligne, 81, fic) != NULL)    // on reboucle tant que la fin du fichier n'est pas atteinte
      {
      nb_lignes_lues++ ;

      // ANALYSE  avec sscanf de la ligne lue précedemment par fgets :
      nb_val_lues = sscanf (ligne, "%s%*s%*s%hd%*s%*s%*s%lf", nom, &age, &taille);
                 // %*s sert à ignorer UN MOT (= suite de caractères différents de Espace, tabulation, Entrée)

      if (nb_val_lues!=3)
         printf ("    !!!! PROBLEME Ligne %2hd !!!!\n", nb_lignes_lues);
                                  // on doit trouver 3 valeurs utiles par ligne

      else             // affichage POUR LE TEST :
         printf("(Donnees extraites Ligne %2hd) %15s  %2hd  %5.2lf\n",
                    nb_lignes_lues,nom, age, taille);
      }
  fclose( fic );
  }
```

Exemple 107. Lecture par *fgets* et comptage de mots avec *sscanf* :

```
char tampon[81], mot_bidon[81] ;
short int nb_mots, num_ligne=0 ;

/* le fichier est supposé ouvert en mode "rt" */
while ( fgets(tampon, 81, fichier) != NULL )        /*fin de fichier non atteinte*/
     {
     nb_mots = 0 ; num_ligne++ ;
     while (sscanf(tampon, "%s", mot_bidon)==1)    nb_mots ++ ;
     printf("\n La ligne %hd comporte %hd mots.", num_ligne, nb_mots ) ;
     }
```

17.5. Lecture en accès binaire d'un fichier texte

Quel que soit son format (texte ou autre), un fichier est stocké sous la forme d'une **succession d'octets** : on peut donc **toujours y accéder en utilisant l'accès binaire** par *fread/fwrite*. Si l'écriture binaire n'est pas toujours la meilleure solution pour un fichier texte (il vaut mieux utiliser *fprintf*), la lecture en accès binaire peut être très efficace.

17.6. Un fichier d'échange entre programmes : le fichier *csv*

Un fichier d'extension *.csv* (*Comma-Separated Values*) est un fichier **texte** qui sert de **fichier d'échange** entre logiciels différents, le plus souvent entre un tableur[11] et un autre programme. Il permet de s'affranchir des formats de fichier propriétaires, souvent complexes, et inconnus des autres logiciels. Il ne contient que l'essentiel (les données), mais aucune des informations de formatage autorisées par le format propriétaire.

Chaque ligne du fichier tableur correspond à une ligne du fichier texte *csv*, et les limites des cellules du fichier tableur sont matérialisées par un séparateur dans le fichier texte (virgule ou point-virgule souvent).

Voici par exemple un fichier *Excel* (sauvegardé au format *.xls*) et sa version *csv*, obtenue après sauvegarde sous *Excel* au format *csv* (avec séparateur point-virgule) et rouverture avec un **éditeur de texte** :

fichier *fic_data.xls*

	A	B	C	D
1	**hello**	-12	*blabla*	1.23
2	**bonjour**	-130	*blibli*	12.345
3	**GutenTag**	arghh!	*blublu*	123.456

fichier *fic_data.csv*

```
hello;-12;blabla;1.23
bonjour;-130;blibli;12.345
GutenTag;arghh!;blublu;123.456
```

Le fichier *csv* de droite présente l'avantage d'être au format texte, donc **lisible** par n'importe quel programme. Réciproquement, le fichier texte au format *csv* peut être **créé** par n'importe quel logiciel ou programme, puis importé dans un tableur où il ne reste plus qu'à refaire la mise en page.

17.6.1 Fabrication d'un fichier *csv* dans un programme en Langage C

C'est une application simple de la fonction *fprintf*. Voir les exemples d'écriture dans un fichier texte, comme l'Exemple 102.

17.6.2 Lecture d'un fichier *csv* dans un programme en Langage C

La lecture d'un fichier *csv* est celle d'un fichier texte, donc pas la plus simple, et elle est compliquée par la présence des séparateurs qui s'ajoutent ou se substituent aux espaces.
Les séparateurs les plus courants sont le **point-virgule ou la virgule**.

La lecture d'un fichier *csv* est le domaine d'application rêvé pour la fonction de **découpage en sous-chaînes *strtok***, vue dans le chapitre *Chaînes de caractères* (paragraphe 12.4.7). Elle doit être suivie par la fonction *sscanf*, qui lit et mémorise la donnée isolée par *strtok*. Comme d'habitude, on retrouve en préambule, pour lire une ligne dans le fichier, la fonction ***fgets***.

L'**Exemple 108.** est un programme qui lit et affiche les données du fichier *fic_data.csv* représenté ci-dessus, en ignorant la colonne 3 supposée sans intérêt. Voici la fenêtre d'exécution obtenue :

```
Ligne  1 :
        cellule 1 (chaine):       hello
        cellule 2 (entier):         -12
        cellule 4 (reel)  :       1.230
Ligne  2 :
        cellule 1 (chaine):     bonjour
        cellule 2 (entier):        -130
        cellule 4 (reel)  :      12.345
Ligne  3 :
Pb de lecture cellule 2 (entier)!

        cellule 1 (chaine):    GutenTag
        cellule 2 (entier):      -11111
        cellule 4 (reel)  :     123.456_
```

[11] Tableur = logiciel de calcul (*Microsoft Excel, OpenOffice* etc.)

Exemple 108. Lecture d'un fichier csv (voir contenu page précédente)

```
void main (void)
 {
 FILE* fic ;
 char ligne[81];
 char *ptr_chaine ;                  // pointeur pour  balayer les sous-chaînes obtenues
 short int num_ligne = 1 ;
 short int   data_entier;
 double      data_reel;
 char        data_chaine[11];

//---------------------- ouverture du fichier de données CSV -----------------------------

 fic = fopen( "fic_data.csv", "rt") ;
 if (fic==NULL)   { printf("Ouverture fichier impossible !"); exit(0); }

//---------------------- lecture du fichier de données CSV -----------------------------

 // on lit une ligne après l'autre jusqu'à la fin du fichier
 while ( fgets( ligne, 81, fic) != NULL )
       {
       printf("\n Ligne %2hd :", num_ligne );
       num_ligne++ ;

       ptr_chaine = strtok (ligne, ";");     // appel d'initialisation de strtok. Séparateur = ';'

       /* on lit une cellule (colonne) après l'autre jusqu'à la fin de la ligne. Notez que si les cellules contenaient
       des données de même type, on pourrait écrire une boucle while(ptr_chaine!=NULL){} */

       // cellule 1 :
       if (sscanf(ptr_chaine,"%s", data_chaine) != 1)     // verif. de la validité des données
              { puts("\nPb de lecture cellule 1 (chaine) !"); data_chaine[0]=0; }

       ptr_chaine = strtok (NULL, ";");           // remplace le prochain séparateur trouvé par 0,
                                                  // puis renvoie l'adresse de la chaîne ainsi obtenue. Séparateur = ';'

       // cellule 2 :
       if (sscanf(ptr_chaine,"%hd", &data_entier) != 1)
              { puts("\nPb de lecture cellule 2 (entier)!"); data_entier=-11111;
}
       ptr_chaine = strtok (NULL, ";");           // remplace le prochain séparateur trouvé par 0

       // cellule 3 :
       ptr_chaine = strtok (NULL, ";");           // on saute la cellule 3 supposée sans intérêt

       // cellule 4 :
       if (sscanf(ptr_chaine,"%lf", &data_reel) != 1)
              { puts("\nPb de lecture cellule 4 (reel)!"); data_reel=-111.111; }
       ptr_chaine = strtok (NULL, ";");           // non indispensable ici

       // affichages :
       printf("\n\t cellule 1 (chaine): %11s ", data_chaine ) ;
       printf("\n\t cellule 2 (entier): %11hd", data_entier);
       printf("\n\t cellule 4 (reel)  : %11.3lf", data_reel);

   }
fclose(fic);
}
```

18 - Les simplifications d'écriture

On a déjà dit, et répété, que pour qu'un programme soit le plus lisible possible, il est indispensable d'user, voire d'abuser, des commentaires ou de la directive *#define*.

Ce chapitre offre deux nouvelles possibilités pour augmenter la clarté d'un programme : la définition de nouveaux noms de types par ***typedef*** (déjà utilisé pour les types structurés et les pointeurs de fonctions) et la définition de types énumérés par ***enum***.

☺ Pour les types définis par *typedef*, il est conseillé d'adopter un nom facilement reconnaissable. Voici par exemple la convention (personnelle) adoptée dans ce document : le nom d'un type créé par *typedef* sera noté **en majuscules** et **commencera par T_** (pour *type_*) afin de le distinguer d'une constante symbolique (toujours écrite en majuscules, autre règle de style). Même si vous utilisez des minuscules, gardez impérativment le **T_** au début du type créé.

Exemples de noms de type : *T_FONCTION, T_COULEUR, T_date*.

Le but de cette convention est d'éviter la confusion fréquente entre le nom du type créé par l'utilisateur et le nom des variables définies avec ce type.

18.1. Définition de nouveaux noms de types : *typedef*

18.1.1 Présentation

Le mot-clé ***typedef*** permet de créer un **synonyme** pour un type existant, de façon à rendre le programme plus lisible et plus facilement modifiable (ce qui améliore la portabilité). Par exemple, la ligne :

```
typedef double T_REEL ;
```

permet d'utiliser le mot *T_REEL* à la place de *double* pour définir des variables dans tout le programme :

```
T_REEL x, y = 8.5 ;
T_REEL* ptr ;
```

La portabilité du programme est considérablement augmentée : il suffit de modifier la ligne qui contient *typedef* pour modifier le type *T_REEL*. Si on désire par exemple effectuer les calculs en simple précision au lieu de double précision, la ligne *typedef* précédente sera remplacée par :

```
typedef float T_REEL ;
```

Une ligne *typedef* peut être placée en début de fichier (à l'extérieur de toute fonction) ou, mieux, dans un **fichier en-tête d'extension .h**.

Remarquons que cette définition de synonyme ressemble à la directive *#define* dans les cas simples, mais elle est plus puissante, car traitée par le compilateur au lieu du préprocesseur.

18.1.2 Exemples d'utilisation de *typedef*

En informatique industrielle, on utilise beaucoup les types non signés (*unsigned* ...), en particulier le type *unsigned char*. Il est donc pratique de définir (dans un fichier en-tête) des synonymes plus courts pour les types non signés, par exemple *U8*, *U16* et *U32* :

```
typedef unsigned char U8  ;
typedef unsigned char U16  ;
typedef unsigned char U32 ;

U8 octet = 0xF3, port ;
U16 vitesse = 18000, position = 0x0A7D ;
```

Voici des exemples de types synonymes un peu plus compliqués (remarquez la syntaxe pour les tableaux) :

```
typedef short int* T_PTR_ENTIER ;
T_PTR_ENTIER p1, p2 ;
```

définit un type "pointeur d'entier" baptisé *T_PTR_ENTIER* et deux variables *p1* et *p2* appartenant à ce nouveau type.

```
typedef char T_CHAINE[81] ;
T_CHAINE phrase = "bonjour!" ;
```

définit un type "chaîne de 81 caractères" (tableau) baptisé *T_CHAINE* et une variable *phrase* initialisée.

```
typedef double T_MATRICE[4][4] ;
T_MATRICE matA, matB ;
```

définit un type "matrice 4*4 de réels double précision" (tableau 2 dimensions) baptisé *T_MATRICE* et deux variables *matA* et *matB*.

```
typedef short int T_FONCTION( short int n ) ;
        /* Le type T_FONCTION peut être refusé par certains compilateurs pour
        définir l'en-tête de fonction, mais ils l'acceptent toujours pour le proto-
        type ou le pointeur de fonction */

T_FONCTION calculer_carre, calculer_cube ;          /*prototypes */

T_FONCTION calculer_carre               /* refusé par certains compilateurs */
        { return n*n ; }
T_FONCTION calculer_cube                /* refusé par certains compilateurs */
        { return n*n*n ; }
```

définit un type "fonction recevant un entier n et renvoyant un entier" baptisé *T_FONCTION* et deux fonctions *calculer_carre* et *calculer_cube* de ce type.

18.1.3 Utilisation de *typedef* pour définir un type structuré

L'utilisation la plus fréquente de *typedef* concerne les structures et permet d'éviter l'écriture un peu lourde *struct toto*. On a déjà vu en détail comment employer *typedef* pour les types structurés dans le chapitre "Structures". Ce qui suit n'est qu'un résumé qui n'apporte rien de plus.

Prenons l'exemple d'une structure permettant de décrire une courbe caractérisée par son nom, son nombre de points et les ordonnées de ses points (les abscisses sont supposées réparties de façon équidistante sur un intervalle donné). On désire donc créer un type structuré *T_COURBE* avec *typedef* :

```
#define MAX_POINTS 10
typedef struct
        {
        char nom[20] ;
        char nb_points ;
        double y[MAX_POINTS] ;
        } T_COURBE ;
```

Pour créer des variables structurées de ce nouveau type *T_COURBE*, il suffit d'écrire :

```
T_COURBE parabole ;
T_COURBE sinus = { "sinusoide", 4, { 1., 0., -1., 0. } } ;
T_COURBE* ptr ;
```

L'écriture *T_COURBE* n'est plus possible dans le cas d'une **structure récursive** (c'est-à-dire un type structuré dont un des champs est du même type structuré), car le compilateur doit connaître tous les mots qu'il rencontre pendant sa lecture linéaire du fichier à compiler.

Voici un exemple de définition de structure récursive *T_INDIVIDU* :

```
typedef struct individu
        {
        char nom[20] ;
        char prenom[20] ;
        struct individu * pere ;
        struct individu * mere ;
        } T_INDIVIDU ;
```

Ici, le type structuré créé peut être désigné indifféremment par ses deux noms synonymes *struct individu* ou *T_INDIVIDU*. La deuxième appelation (plus courte) sera bien sûr toujours préférée, sauf à l'intérieur de la définition du type structuré où le mot *T_INDIVIDU* n'est pas encore connu du compilateur.

Pour définir des variables structurées de type *T_INDIVIDU*, il suffit d'écrire :

```
T_INDIVIDU Claude ;
T_INDIVIDU Jojo = { "Joseph", "Dupuis" } ;
```

18.2. Les types énumérés : *enum*

L'énumération définit la liste complète des valeurs (entières) qui peuvent être attribuées à une variable appartenant à ce type énuméré.

On peut définir un type énuméré *enum Couleur* par :

```
enum Couleur { BLEU, BLANC, VERT, ROUGE, VIOLET } ;
```

ou mieux, on utilise *typedef* pour définir le type *T_COULEUR* (c'est cette définition qui sera utilisée dans toute la suite) :

```
typedef enum { BLEU, BLANC, VERT, ROUGE, VIOLET } T_COULEUR ;
```

Des variables appartenant à ce type énuméré *T_COULEUR* peuvent ensuite être définies par :

```
T_COULEUR tapis ;
T_COULEUR canape = VERT ;
```

Ces variables ne peuvent prendre de valeur que dans la liste *BLEU, BLANC, VERT, ROUGE, VIOLET*.

Les valeurs de la liste sont considérés par le compilateur comme des **constantes entières ordonnées**, qui valent par défaut 0, 1, 2, 3 etc. (dans l'exemple ci-dessus, *BLEU* vaut 0, *BLANC* vaut 1 ... *VIOLET* vaut 4). Les variables de type énuméré appartiennent donc à un **sous-ensemble des entiers**. On peut les utiliser pour faire tout ce qui est autorisé sur un entier : indice de tableau, compteur de boucle *for*, etc.

Si on le souhaite, on peut forcer les valeurs entières associées par le compilateur aux noms symboliques de la liste :

typedef enum { BLEU=-1, BLANC, VERT=14, ROUGE, VIOLET } T_COULEUR ;

Dans ce cas, *BLEU* vaut -1, *BLANC* 0, *VERT* 14, *ROUGE* 15 et *VIOLET* 16.

L'utilisation des variables de type énuméré ne peut faire appel qu'aux valeurs de la liste, ce qui rend les programmes **très lisibles** (en particulier grâce à l'instruction *switch*).

```
if ( canape == ROUGE )    ...

tapis = VIOLET ;

switch (tapis)                 /* le plus utile */
        {
        case BLEU : ...
        case BLANC : ...
        ...
        }
```

Voici un dernier exemple qui permet de définir un type énuméré caractérisant les jours de la semaine :

```
typedef enum
        {
        LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE
        } T_ jour_semaine ;

T_ jour_semaine jour = MARDI ;

for (jour=LUNDI ; jour<=VENDREDI ; jour++)   ...
```

Le seul inconvénient des types énumérés se situe lors des entrées/sorties conversationnelles (*printf, scanf*) : il n'existe pas de codes formats pour ces types, en dehors des codes formats **entiers** bien sûr utilisables. Il est donc nécessaire, si on veut voir s'afficher « Vert » à l'écran au lieu de la valeur entière 14 associée à la valeur énumérée VERT , de créer des fonctions de lecture ou d'affichage qui convertissent une valeur du type énuméré en la chaîne de caractères correspondante et réciproquement (utilisation de *switch* ou d'un tableau de chaînes de caractères).

Exemple 109. Affichage des valeurs d'un type énuméré

```
//---- le type énuméré T_jour_semaine et son tableau de chaines de caractères associé ----
typedef enum
 { LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE
 } T_jour_semaine ;

static char nom_jour[7][10] = { "Lundi", "Mardi", "Mercredi",
"Jeudi","Vendredi", "Samedi", "Dimanche" };
//-----------------------------------------------------------
void main(void)
  {
  T_jour_semaine jour ;
  short int num_jour=1 ;

  for ( jour=LUNDI ; jour<=DIMANCHE ; jour++)
      printf("\n jour %hd : %s", num_jour++, nom_jour[jour] );
  }
```

Affichage obtenu :

```
jour 1 : Lundi
jour 2 : Mardi
jour 3 : Mercredi
jour 4 : Jeudi
jour 5 : Vendredi
jour 6 : Samedi
jour 7 : Dimanche
```

19 - Les classes d'allocation mémoire

19.1. Les 3 sortes de variables : fichier, bloc, paramètre formel

Examinons le programme suivant :

```
double nombre ;                  /* nombre est une variable de fichier globale */

double carre(double nbre )       /* nbre est un paramètre formel */
       {
       return( nbre * nbre ) ;
       }

 void main(void)
       {
       double reel ;             /* reel est une variable de bloc locale */

       reel = 5. ;
       nombre = 6. ;
       printf("Le carre de %f est %f\n", reel, carre(reel) ) ;
       printf("Le carre de %f est %f\n", nombre, carre(nombre) ) ;
       }
```

Ce programme fait apparaître les trois sortes de variables disponibles en Langage C :

- *nombre* est une variable **de fichier** ; elle est définie en dehors de toute fonction.
- *reel* est une variable **de bloc** ; elle est définie au début d'un bloc délimité par des accolades.
- *nbre* est un **paramètre formel** ; elle est définie dans la ligne d'en-tête d'une définition de fonction.

Suivant sa classe, une variable présente deux caractéristiques :

- sa **visibilité** : c'est la partie du programme où on peut utiliser la variable.
- sa **durée de vie** : c'est la durée pendant laquelle une place est réservée en mémoire pour la variable.

19.2. Les variables de fichier

19.2.1 Visibilité

```
/* Fichier 1 */
int reel ;
```

La variable *reel* définie dans le fichier 1 est visible **dans tout ce fichier** et peut être utilisée **dans d'autres fichiers** grâce à la déclaration *extern*. Cette variable est une variable **de fichier globale**.

La classe par défaut d'une variable de fichier est **globale**.

```
/* Fichier 2 */
extern int reel ;
```

La variable *reel*, déclarée *extern* dans le fichier 2, est visible dans tout le fichier 2 bien qu'elle soit **définie ailleurs**. Pour le fichier 2, cette variable est une variable **de fichier importée**.

Cette classe importée se caractérise par la déclaration ***extern*** :

extern *type nom_var* **;**

```
/* Fichier 3 */
static int reel ;
```

La variable *reel* du fichier 3, dont la définition est précédée du mot-clé *static*, n'est visible que **dans le seul fichier où elle est définie** : elle est alors dite de classe locale. Elle ne peut donc pas être utilisée dans un autre fichier.

La définition d'une variable de fichier de classe locale est complétée du mot-clé ***static*** :

static *type nom_var* ;

☞ On remarquera que ce que nous avons appelé jusqu'ici une variable globale est en réalité une variable de fichier de classe globale ou importée.

19.2.2 Durée de vie

La durée de vie d'une variable de fichier est celle du programme : c'est une variable **permanente**.
L'emplacement mémoire d'une variable de fichier est réservé (alloué) à la compilation, dans une zone mémoire dite zone de données statiques.

Classe de mémorisation	Commentaire	Exemple
durée de vie permanente quelle que soit la classe		
globale (ou externe)	peut être exportée dans un autre fichier	double reel ;
locale (ou statique)	visible uniquement dans le fichier où elle est définie	**static** double reel ;
importée (ou externe)	définie dans un autre fichier.	**extern** double reel ;

Figure 19 --11 : Les variables de fichier

19.3. Les variables de bloc

19.3.1 Visibilité

Une variable de bloc est visible seulement dans le bloc où elle est définie.

Exemple 110. Visibilité d'une variable de bloc :

```
double carre( double nbre )
        {
        double resultat ;              /* définition de la variable de bloc */
        resultat = nbre * nbre ;
        return resultat ;
        }
```

La variable *resultat* est visible seulement dans le bloc { } où elle est définie ; toute variable du fichier portant le même nom est distincte.

19.3.2 Durée de vie

Par défaut, une variable de bloc est allouée en cours d'exécution **à l'entrée dans le bloc** où elle est définie et elle est **détruite à la sortie du bloc**.

La durée de vie par défaut d'une variable de bloc est donc la durée pendant laquelle le bloc est actif.

On dit que la classe par défaut d'une variable de bloc est la classe **automatique**.

L'emplacement mémoire d'une variable de bloc de classe automatique est alloué **dans la pile** au **début** de l'exécution du bloc. Cet emplacement mémoire disparaît à la fin de l'exécution du bloc.

Cependant, une variable de bloc peut être **permanente**, c'est-à-dire que sa durée de vie devient celle du programme. Elle est alors dite de **classe permanente ou statique.**

Une variable de bloc de classe statique est définie à l'aide du mot-clé ***static***. Par exemple :

```
void fonction( void )
{
static type nom_var          /* définition de la variable de bloc statique */
…
}
```

Une variable de bloc de classe statique est allouée par le compilateur en zone de données statiques, comme une variable de fichier.
Sa durée de vie permanente ne l'empêche pas de rester **locale du point de vue de la visibilité**.

On remarquera que ce que nous avons appelé jusqu'à présent une variable locale est en réalité une variable de bloc de classe automatique.

19.4. Les paramètres formels

19.4.1 Visibilité

Un paramètre formel est visible **seulement dans la fonction où il est défini.**

19.4.2 Durée de vie

La durée de vie d'un paramètre formel est **la durée pendant laquelle la fonction s'exécute.**

Un paramètre formel est donc de classe **automatique**.

Il est créé en pile lors de l'appel de la fonction et disparaît à la fin de l'exécution de la fonction.

19.5. Initialisation d'une variable

Initialiser une variable consiste à lui affecter une valeur au moment de son allocation en mémoire.

19.5.1 Initialisation d'une variable de fichier

/* fichier principal */

```
int nombre = 5 ;

extern void affiche(void) ;

void main(void)
     {
     affiche() ;
     }
```

/* fichier fonction.c */

```
extern int nombre ;

void affiche(void)
     {
     printf("nombre=%d", nombre) ;
     }
```

La variable *nombre* est allouée et initialisée au cours de la compilation. Au lancement du programme, nombre a pour valeur 5.
Attention : dans le fichier *fonction.c,* il n'est pas permis d'initialiser *nombre*, car cette variable est définie ailleurs. La ligne `extern float nombre` est une déclaration et non une définition (elle ne réserve pas de place en mémoire).

19.5.2 **Initialisation d'une variable de bloc**

Initialisation d'une variable de bloc de classe **automatique**

Compiler et exécuter le programme suivant. Expliquer le résultat obtenu (le programme affiche toujours reel=5)

Exemple 111. Variable automatique non *static*

```
void incrementer( void )
      {
      short int entier = 5 ;
      printf( "entier=%hd \n", entier ) ;
      entier ++ ;
      }

 void main(void)
      {
      short int i ;
      for (i=1 ; i<=5 ; i++ ) incrementer() ;
      }
```

Initialisation d'une variable de bloc de classe **statique**

La variable de bloc utilisée est à présent de classe statique. Expliquer le fonctionnement des deux programme ci-dessous.

/* fichier var_st1.c */

```
void incrementer( void )
     {
     static short int entier = 5 ;
     printf( "entier=%hd \n", entier ) ;
     entier ++ ;
     }
void main(void)
     {
     short int i ;
     for (i=1 ; i<=5 ; i++ )  incrementer() ;
     }
```

/* fichier var_st2.c */

```
void incrementer( void )
     {
     static short int entier ;
     entier=5 ;
     printf( "entier=%hd \n", entier ) ;
     entier ++ ;
     }

void main(void)
     {
     short int i ;
     for (i=1 ; i<=5 ; i++ )    incrementer() ;
     }
```

Le premier programme affiche : entier=5, puis entier=6, puis entier=7 ... entier=9
Le second programme affiche : entier=5, puis entier=5, puis entier=5...

19.5.3 Initialisation d'un paramètre formel

Un paramètre formel est initialisé à l'appel de la fonction avec la valeur du paramètre effectif correspondant.

La valeur du paramètre effectif est copiée dans la pile à l'emplacement réservé au paramètre formel. La fonction vient chercher cette valeur en pile quand elle en a besoin.

19.6. Syntaxe complète d'une définition de variable

La définition d'une variable obéit donc à la syntaxe générale suivante :

[*classe*] <type> <identificateur> [= <valeur>] ;

Variable	Classe de mémorisation	Visibilité	Durée de vie	Exemple
de fichier	globale locale (ou statique) importée	globale locale au fichier -	permanente permanente permanente	*int entier ;* ***static** int entier ;* ***extern** int entier ;*
de bloc	automatique statique	locale au bloc locale au bloc	celle du bloc permanente	*int entier ;* ***static** int entier ;*
paramètre formel	automatique	locale à la fonction	celle de la fonction	*int entier ;*

Figure 19 --12 : Les classes d'allocation mémoire

19.7. Variables « statiques » et « automatiques »

Du point de vue de la localisation en mémoire, on voit apparaître deux types de variables :

- celles qui sont allouées par le compilateur dans la zone des données statiques (elles sont souvent qualifiées de **statiques**). Ce sont toutes les variables de fichier et les variables de bloc de classe statique. Elles ont une durée de vie **permanente** et sont **initialisées à 0** par défaut.
- celles qui sont allouées en pile (elles sont alors dites **automatiques**). Ce sont les variables de bloc de classe automatique et les paramètres formels. Elles ont une durée de vie limitée à celle du bloc oùelles sont définies et n'ont aucune initialisation par défaut.

Il existe des variables situées dans une troisième zone de la mémoire, appelée le **tas**. Ce sont les variables **dynamiques**. Elles sont utilisées pour l'allocation dynamique de la mémoire. Elles sont décrites dans le chapitre *Pointeurs*.

20 - Etes-vous un « bon » programmeur ?

Faites le test suivant. Si vous avez des réponses dans la colonne de gauche, vous n'êtes pas (encore) un programmeur confimé, raisonnable et efficace.
Refaites le test régulièrement : vous devez normalement finir par répondre toujours à droite ! Vous ne serez pas encore un programmeur efficace, mais vous serez sur le bon chemin…

Vous venez de finir la lecture de votre cahier de charge, assez complexe…

Vous lancez votre environnement de développement (IDE) et vous commencez à taper votre source.

Vous attrapez papier et crayon, et vous vous lancez dans plein de dessins avec des boîtes, des flèches, des listes...

En tapant votre source…

Pour gagner du temps, vous négligez les accolades et les indentations. Vous ferez ça plus tard quand ça marchera.
Les commentaires, ce sera vraiment pour la fin. De toute façon, ce n'est pas ça qui fera marcher le programme !

Vous ne savez pas écrire un programme sans indenter comme un maniaque.
Vous mettez des titres (commentaires) et des commentaires à toutes les lignes importantes.
D'ailleurs, vous êtes incapable de lire un code non indenté et sans commentaires !

Vous voulez aller le plus vite possible…

Vous écrivez tout votre programme dans le programme principal *main*. Vous découperez en fonctions quand ça marchera.

Vous écrivez une fonction, puis testez.
Puis vous recommencez avec la suivante.

Lisibilité ou facilité ?

Vous tapez les valeurs numériques directement « en dur ».
Pour les noms de variables, il n'y a qu'à piocher dans l'alphabet, on gagne du temps au moment de la frappe !

Vous créez plein de constantes.

Vous choisissez des noms de fonction et de variables explicites même si longs à écrire.

En tapant votre programme…

Vous faites plein de saisies au clavier et d'affichage à l'écran.
Un programme sans affichage à l'écran, ce n'est pas un beau programme !

Vous ne faites aucune entrée au clavier : vous utilisez des variables de test initialisées.
Vous ne faites pas d'affichage à l'écran : vous utiliserez le debugger pour afficher les variables.
Les entrées/sorties, ce sera pour la fin !

Vous cliquez sur l'icône « Compiler », le compilateur affiche 127 erreurs…

Vous partez en courant chercher quelqu'un de compétent.
Vous êtes traumatisé : « En plus, les messages d'erreur sont en anglais ! »
Il vous faut donc quelqu'un de compétent en informatique ET en anglais…

Vous prenez la chose avec philosophie : c'est probablement un point-virgule oublié qui a généré la première erreur. Et vous savez qu'une seule erreur peut générer des dizaines d'erreurs fictives…
Vous lisez attentivement chaque mot du premier message d'erreur du compilateur. Ce n'est pas de l'anglais très compliqué !

Vous cliquez sur l'icône « Fabriquer l'exécutable », tout se passe bien et un programme exécutable est disponible.

Vous êtes content : votre programme va marcher.

Vous savez que les choses sérieuses commencent.

Vous cliquez sur l'icône « Exécuter », le programme se lance et se termine sans planter…

Si le programme ne plante pas, c'est qu'il marche, non ?
Donc vous passez à l'exercice suivant.

Vous examinez tous les résultats avec attention, au debugger. Si ça marche, vous mettez au point des tests plus exigeants qui testent les limites du programme.

Quand vous lancez votre programme, il fournit de premiers résultats qui semblent assez corrects.

Vous passez à l'exercice suivant.
Votre programme, c'est comme votre enfant : vous n'allez pas chercher à le mettre en difficulté, non ?

Vous cherchez à mettre en défaut votre programme. Vu de dehors, cela peut apparaître comme la preuve d'un tempérament sadique : ce n'est que de la rigueur.

Il est deux heures du matin…

Vous y passerez la nuit s'il le faut, mais il va marcher, ce programme !

Vous êtes au lit depuis longtemps. Demain, vous aurez les idées plus claires et les erreurs vous paraîtront évidentes.

Annexe A. Table des codes ASCII

Dec	Hex	char		Dec	Hex	char	Dec	Hex	char	Dec	Hex	char
0	00	NUL	*caract. null*	32	20	espace	64	40	@	96	60	'
1	01	SOH		33	21	!	65	41	A	97	61	a
2	02	STX		34	22	"	66	42	B	98	62	b
3	03	ETX		35	23	#	67	43	C	99	63	c
4	04	EOT		36	24	$	68	44	D	100	64	d
5	05	ENQ		37	25	%	69	45	E	101	65	e
6	06	ACK		38	26	&	70	46	F	102	66	f
7	07	BEL	*bell*	39	27	‘	71	47	G	103	67	g
8	08	BS	*backspace*	40	28	(	72	48	H	104	68	h
9	09	TAB	*tabul. horiz*	41	29	)	73	49	I	105	69	i
10	0A	LF	*line feed*	42	2A	*	74	4A	J	106	6A	j
11	0B	VT	*Tabul. vertic*	43	2B	+	75	4B	K	107	6B	k
12	0C	FF		44	2C	,	76	4C	L	108	6C	l
13	0D	CR	*retour ligne*	45	2D	-	77	4D	M	109	6D	m
14	0E	SO		46	2E	.	78	4E	N	110	6E	n
15	0F	SI		47	2F	/	79	4F	O	111	6F	o
16	10	DLE		48	30	0	80	50	P	112	70	p
17	11	DC1		49	31	1	81	51	Q	113	71	q
18	12	DC2		50	32	2	82	52	R	114	72	r
19	13	DC3		51	33	3	83	53	S	115	73	s
20	14	DC4		52	34	4	84	54	T	116	74	t
21	15	NAK		53	35	5	85	55	U	117	75	u
22	16	SYN		54	36	6	86	56	V	118	76	v
23	17	ETB		55	37	7	87	57	W	119	77	w
24	18	CAN		56	38	8	88	58	X	120	78	x
25	19	EM		57	39	9	89	59	Y	121	79	y
26	1A	SUB		58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	*escape*	59	3B	;	91	5B	[	123	7B	{
28	1C	FS		60	3C	<	92	5C	\	124	7C	|
29	1D	GS		61	3D	=	93	5D	]	125	7D	}
30	1E	RS		62	3E	>	94	5E	^	126	7E	~
31	1F	US		63	3F	?	95	5F	_	127	7F	DEL

Annexe B. Débogage d'un programme

Les principes du débogage (*debug*) restent les mêmes sous n'importe quel environnement de développement (IDE), avec des noms proches de ceux donnés dans le tableau.

Tout environnement intégré de développement (IDE) dispose d'un débogueur (*debugger*) qui aide à trouver les erreurs (bogues en français, *bugs* en anglais). Il permet par exemple d'arrêter le programme aux endroits désirés, d'afficher le contenu de variables et d'exécuter des parties du programme en pas à pas.

Un débogueur offre toujours les possibilités suivantes :

- **arrêter l'exécution du programme sur une instruction précise**, indiquée par un **point d'arrêt** ou par la position du curseur ;
- **afficher le contenu de variables** ou la valeur d'expressions, pour suivre en temps réel leur évolution au cours de l'exécution ;
- **exécuter en pas à pas** certaines parties du code, soit en exécutant globalement les appels de fonctions (***Step Over***), soit en entrant en pas à pas dans les fonctions rencontrées (***Step Into***), à l'exception des fonctions de l'IDE.

Pour commencer l'exécution du programme en mode *debug*, on peut **placer un point d'arrêt** à l'endroit désiré (souvent avec un clic dans la marge), puis lancer la commande *Run*. Pour arrêter définitivement l'exécution d'un programme en mode *debug*, il existe toujours une commande spécifique *Reset*.

Un point d'arrêt reste en place jusqu'à ce qu'on l'enlève ; le programme s'arrête toujours lorsqu'il arrive à l'instruction où se trouve le point d'arrêt.

Une fenêtre spécifique (souvent appelée *Watch* ou *Variables*) peut présenter les valeurs des variables ou expressions dont on a demandé l'affichage. Elle est mise à jour au fur et à mesure de l'exécution du programme. Cette fenêtre est particulièrement utile pour afficher les contenus des tableaux ou des structures.

Les commandes de débogage sont en général réparties dans différents menus de l'IDE : souvent les menus *Run*, *Debug*, etc. C'est pourquoi le tableau ci-dessous n'indique que le nom approximatif des commandes de debug sans préciser le menu où elles se trouvent. Les noms peuvent légèrement différer d'un IDE à l'autre. Notez qu'il existe en général des touches de raccourcis très pratiques.

commande	Type de l'argument à afficher
Run ou **Go** **(Debug / non Debug)**	Exécute le programme jusqu'au prochain point d'arrêt ou jusqu'à la fin du programme. **Attention** : dans certains environnements (*CodeBlocks*, *DevCPP*), il y a un bouton « lancement en mode debug » distinct de celui du « lancement en mode normal » (qui ne tient pas compte des points d'arrêt).
Program Reset	Arrête définitivement l'exécution du programme en cours.
Toggle Breakpoint (souvent : clic dans la marge)	Place ou supprime un point d'arrêt.
Add Watch, Variables Windows…	Affiche une fenêtre avec les valeurs des variables (à choisir ou non).
Step Over	Exécute l'instruction suivante. S'il s'agit d'un appel de fonction, exécute **globalement** la fonction.
Step Into	Exécute l'instruction suivante. S'il s'agit d'un appel de fonction dont le source est disponible, **pénètre en pas à pas dans la fonction**.
Go to cursor	Exécute le programme jusqu'à la ligne du curseur.

Les commandes du débogueur

Liste des tableaux et des figures

Les définitions de type entier (les types en gris sont les plus utilisés) 15
Les définitions de réels (le type en gris est le plus utilisé)........ 16
Priorité des opérateurs dans l'ordre décroissant 21
Les opérateurs mathématiques........ 22
Les opérateurs relationnels 25
Les opérateurs logiques........ 26
Les opérateurs de manipulation de bit........ 27
Les codes formats 44
Les représentations des codes ASCII usuels 45
Les principales fonctions mathématiques de la bibliothèque standard........ 57
Figure 11 --1 : Un tableau en mémoire........ 74
Figure 11 --2 : Un tableau en paramètre 78
Figure 12 --3 : Allocation mémoire d'une chaîne de caractères........ 84
Figure 13 --4 : Pointeur et variable pointée........ 92
Figure 13 --5 : Passage en paramètre par adresse (ici pour calculer deux résultats)........ 96
Figure 16 --6 : Structure en accès direct ou indirect 112
Figure 16 --7 : Tableau de structures........ 112
Figure 16 --8 : Accès à un champ de structure (direct ou indirect) 115
Figure 16 --9 : Structures imbriquées 116
Figure 16 --10 : Liste chaînée........ 117
Figure 19 --11 : Les variables de fichier 142
Figure 19 --12 : Les classes d'allocation mémoire 146
Les commandes du débogueur........ 150

Index

!, *26*
!=, 25
#define Voir Constante symbolique
#include, *53*, *70*
% (modulo), *22*
%*s et autres %*, 130, 132
%... (code format), 44
& (op. adresse), *29*, *96*
& (op. bit à bit), *27*
&&, *26*
* (op. d'indirection, voir *Pointeur*), 92, 96
* (opérateur de multiplication) *Voir* chapitre Opérateurs arithmétiques
. (opérateur d'accès au champ d'une structure), 112
? (op.d'alternative), *39*
\ (antislash simple), 45
\\ (antislash double), *123*
^, *27*
|, *27*
||, *26*
~, *27*
++ et **--**, *29*
< ou >, 25
<= ou >=, 25
==, *25*
-> (opérateur d'accès aux champs d'une structure), 112, 113
>> ou **<<**, *27*
accès bit, 27
affectation =
  de pointeur, 92
  de variables scalaires, 17
  de variables structurées, 112
  opérateur, 23
aléatoire, 77
allocation dynamique, *97*, *98*, *117*
argument d'appel (de fonction), *53*, *54*, *62*, *64*
atan, *57*
bibliothèque mathématique du C, *57*
boîte noire (de fonction), *59*
boucle *Voir Répétition*
break, 36
buffer, 125
C++, 5
carré, cube, 57
cast, *24*
chaîne de caractères
  comparaison, 88
  constante chaîne, 107
  copie, concaténation, 88
  définition, 83
  écriture à l'écran, 84
  écriture formatée dans une chaîne, 87
  initialisation, 84
  lecture au clavier, 85
  lecture formatée dans une chaîne, 87
  longueur utile, 88
  recherche dans une chaîne, 88
  tableau de chaînes de caractères, 90
choix multiple (*switch*), *39*
code format, 44
commentaire, 7
compilateur, *54*, *67*
  configurer, *54*
condition
  double *Voir* opérateurs logiques/relationnels
  multiple, 25
  simple, 25, *Voir* opérateurs relationnels
const, *19*
constante, *18*
  chaîne de caractères, 107
  de type pointeur, 105, 106
  définie par *#define*, *18*
  définie par *const*, *19*
  exemples de valeurs entières ou réelles, *18*
constante symbolique, *18*
continue, *36*
conversion code ASCII-entier, 41
conversion valeur numérique<->chaîne de caractères, 87
conversions
  explicites (*cast*), *24*
  implicites, *23*
cosinus, 57
csv, 135
debug, 6, 150
décalage, *27*
découpage fonctionnel, *59*
différent de *Voir* opérateurs relationnels
directives (du préprocesseur), 11, 53, 67

- division entière, *22*, 24
- *do while*, *33*
- *double*, 16
- éditeur de liens, *68*
- effacer l'écran *Voir Cls* ou *clrscr* (non portable)
- égal à *Voir* opérateurs relationnels
- encadrement, *26*
- *enum*, 139
- exit, 124
- *exp*, *57*
- exponetielle, 57
- *extern*, *68*, *Voir* chapitre "Classes d'allocation"
- *fabs*, *57*
- *fclose*, 124
- *feof*, 130
- *fflush Voir rewind* (équivalent)
- *fgets*, 129, 131
- fichier
  - binaire (accès), 121
  - *csv*, 135
  - d'échange, 135
  - fermeture, 124
  - lecture/écriture binaire, 129
  - lecture/écriture texte, 129
  - longueur (exemple), 127
  - ouverture, 123, 127
  - texte (accès), 121
  - texte en lecture binaire, 134
- fichier *csv*, 135
- fichier en accès
  - binaire, 121, 125
  - texte, 121, 129
- fichier en-tête, *118*
- *FILE**, 123
- *float*, 16
- fonction
  - appel, *51*, *53*
  - arguments d'appel, *53*, *54*, *62*, *64*
  - définition, *62*
  - mode d'emploi *Voir* Prototype de fonction
  - pointeur de fonction, 100
  - prototype *Voir* Prototype de fonction
- *fopen*, 123
- *for*, *34*
- *fread*, 125
- *free*, 97, 98
- *fscanf*, 130
- *fseek*, 127
- *fwrite*, 125
- *getch* (spécifique à Borland), *49*, *69*
- *getchar*, *49*
- *GetKey* (spécifique à CVI), *49*, *69*
- *gets*, 85
- hasard, 77
- IDE, 5
- *if*, *37*
- *if – else*, *37*
- *if-else* imbriqués (*if- else if- else if*), *38*
- inférieur à *Voir* opérateurs relationnels
- initialisation
  - aléatoire, 77
  - d'un tableau, 75
  - d'une chaîne, 83
  - d'une structure, 111
  - d'une variable entière, 16
  - d'une variable réelle, 16
- *int*, 15
- itération *Voir* Répétition
- *LeftValue*, *23*
- liste chaînée, 116
- *log*, *57*
- logarithmes, 57
- *main*, 7, 63
- *malloc*, 97, 98
- masque (accès bit), 27
- masques, *27*
- *math.h*, *57*
- Mathématiques (fonctions), 57
- matrice *Voir* Tableau à plusieurs dimensions
- *memmove*, 76
- menu *Voir Switch*
- *MessagePopup* (CVI seulement), 86, 87
- minuscules et majuscules, 8
- modèle de structure, 109
- modèle mémoire, 97
- modulaire (programmation), 9, 59
- modulo, *22*
- NULL, 94, 95
- *null pointeur assigment*, 95
- opérateur d'adresse &, *29*, *96*
- opérateur d'alternative ?, *39*
- opérateur d'indirection, 92, 96
- opérateurs arithmétiques +, -, *, /, %, *22*
- opérateurs de manipulation de bit, 27
- opérateurs incrémentation ++/décrémentation --, *29*
- opérateurs logiques &&, ||, !, *25*
- opérateurs relationnels <, ==, >, !=, *25*
- Orienté Objet, 5
- paramètre formel/effectif, 62, 63, 64, *Voir aussi* chapitre "Classes d'allocation"
- passage en paramètre
  - d'un tableau, 76, 106
  - d'une fonction, 102
  - d'une variable structurée, 113
  - par adresse, 95
  - par valeur, 53, 62, 65
- pointeur, 91
  - affectation, 92
  - de fonction, 100, 102
  - définition, 91
  - sur une chaîne de caractères, 107
  - tableau de pointeurs, 108

utilisé pour l'allocation dynamique, 97
utilisé pour le passage en paramètre par adresse, 95
pointeur de fichier *Voir* FILE*
portabilité, 5
pour *Voir* for
pow, *57*
préprocesseur, *67*
printf, *43*
priorité des opérateurs, *21*
prototype de fonction, *52*, *54*, *60*, *68*
puissance *Voir* pow
putchar, *49*
racine carrée *Voir* sqrt
rand, 77
règles de style, 6
répéter *Voir* while et do...while
répétitions, *32*
rewind, *47*
Saisie *Voir scanf*
de caractères, *47*
scanf, *45*
valeur renvoyée, *48*
séparateur (dans un fichier csv), 135
séparateur (pour *scanf* et variantes), *46*
si...sinon *Voir* if...else
sin ou *cos*, *57*
sinus, 57
sizeof, *30*
sprintf, 87
sqrt, *57*
srand, 77
sscanf, 87, 131
static Voir chapitre "Classes d'allocation"
static (tableau ou structure), *74*, *110*
strchr, 88
strcmp, strncmp, 88
strcpy, *strncpy*, 88
strlen, 88
strstr, 88
strtok, 89, 135
struct, 109
structure
accès aux champs, 112
affectation, 112
comparaison avec tableau, 115
définition d'une variable structurée, 110
initialisation, 111
modèle, 109
passage en paramètre à une fonction, 113
récursive, 117, 139
structures imbriquées, 115, 139
supérieur à *Voir* opérateurs relationnels
switch, *39*, 140
tableau, *73*
à plusieurs dimensions, *78*
à une dimension, *74*
accès aux éléments, *75*, *80*
adresse, *76*
alloué dynamiquement, 98, 118
copie, 76
de caractères (chaîne) *Voir aussi* Chaînes de caractères
de pointeurs, 108
de pointeurs de fonction, 104
définition, *74*
initialisation, *75*, *79*
passage en paramètre à une fonction, 76, 81, 106
signification du nom, *76*
taille en octets d'une variable, *30*
tampons clavier et système, *46*
nettoyer avant saisie, *47*
tan, *57*
tangente, 57
tant que *Voir* while
tas, 97
test, 25
type
caractère ou octet, 17
entiers, 15
énuméré, 139
pointeur, 94
pointeur de fichier, 123
pointeur de fonction, 100
réels, 16
structuré, 109
synonyme *Voir Typedef*
tableau à une dimension, 74
tableau de plusieurs dimensions (matrices...), 78
typedef, 109, 138
valeur absolue, 57
variable, 13
définition, 14
emplacement de la définition, 13, 14
globales, *66*
identificateur, 14
initialisation, 15
locales, *65*
type caractère (char), *17*
type réel, 16
types entiers (*int* et variantes), 15
while, *32*

Table des matières simplifiée

Introduction 5
1 - Premier programme en C 7
2 - Un exemple de programme plus évolué 9
3 - Variables-constantes-affectation 13
4 - Opérateurs - Conversions 21
5 - Les structures de contrôle 31
6 - Les entrées/sorties conversationnelles (clavier/écran) 43
7 - Utilisation de fonctions 51
8 - La bibliothèque de fonctions mathématiques *(sinus, exp, valeur absolue…)* 57
9 - Définition de fonction 59
10 - La compilation séparée (multi-fichiers) 67
11 - Les tableaux 73
12 - Les chaînes de caractères 83
13 - Les pointeurs 91
14 - Pointeurs et tableaux à une dimension 105
15 - Pointeurs et chaînes de caractères 107
16 - Les structures 109
17 - Les fichiers 121
18 - Les simplifications d'écriture 137
19 - Les classes d'allocation mémoire 141
20 - Etes-vous un « bon » programmeur ? 147
Annexe A. Table des codes ASCII 149
Annexe B. Débogage d'un programme 150
Liste des tableaux et des figures 151
Index 152