LE LANGAGE C

TODO :

-

v1.1.11.0 – 16/02/2010
peignotc(at)arqendra(dot)net / peignotc(at)gmail(dot)com

TABLE DES MATIÈRES

TABLE DES ANNEXES

TABLE DES ILLUSTRATIONS

1 INTRODUCTION AU LANGAGE C

Le langage C est un langage de programmation qui a été inventé au début des années 70 par les créateurs du système d'exploitation Unix afin de faciliter le développement de celui-ci. À l'époque, l'assembleur était le langage le plus utilisé pour ce genre de programmation, mais par manque de fiabilité et de simplicité il fut abandonné au profit d'un nouveau langage, construit sur les bases des langages Algol et B (laboratoires Bell) [1].

Le C est un langage compilé, présentant les caractéristiques d'un langage de haut niveau (évolué et structuré), tout en offrant les mêmes possibilités que l'assembleur (proche du langage machine).

De par sa conception et sa très large diffusion, le C est un langage très ouvert, qui dispose d'un certain nombre de bibliothèques (graphiques, mathématiques, réseaux, etc.), et est capable de s'interfacer avec de nombreux autres langages, aussi bien des langages de plus bas niveau (assembleur) que de plus haut niveau (C++, Java, etc.).

Les avantages du langage C sont :

- fiable et rapide (à l'exécution) ;
- bonnes capacités d'accès aux ressources matérielles ;
- faible volume mémoire occupé par le noyau ;
- code généré optimisé (taille réduite).

Les inconvénients du langage C sont :

- langage vieillissant, inadapté aux méthodes de développement modernes (UML, objet) ;
- plus complexe que d'autres langages de haut niveau ;
- séquentiel donc inadéquat pour le développement multi-tâches.

Malgré sa conception assez ancienne, c'est un langage encore très utilisé dans les entreprises, à la fois par habitude, et à la fois par ses nombreux avantages ; cela reste notamment un choix de langage pertinent lorsque l'on veut travailler avec une partie opérative matérielle.

---

[1] A… B… C !!… ou « comment trouver un nom ».

2 NOTIONS DE BASE

2.1 DÉVELOPPEMENT EN LANGAGE C

2.1.1 Environnement de développement

Pour développer en langage C, un éditeur de texte associé à un compilateur C et à un éditeur de liens peut suffire. Mais on peut aussi utiliser des outils de développement, comme des EDI [1] (Code::Blocks, Dev-C++, BorlandC, C++ Builder, Visual C++, etc. [2]) qui incluent les outils de base nécessaires (éditeur de texte, compilateur et éditeur de liens) et proposent un certain nombre de fonctionnalités qui simplifient la construction d'applications (débogueur, etc.).

Le langage C est un langage compilé : à partir du fichier contenant le code source écrit en langage C, un fichier exécutable final est généré ; celui-ci pourra être exécuté par toute machine utilisant même processeur et même système d'exploitation que ceux de la machine utilisée lors du développement [3].

La création d'un fichier exécutable à partir d'un fichier source C suit les étapes suivantes [4] :

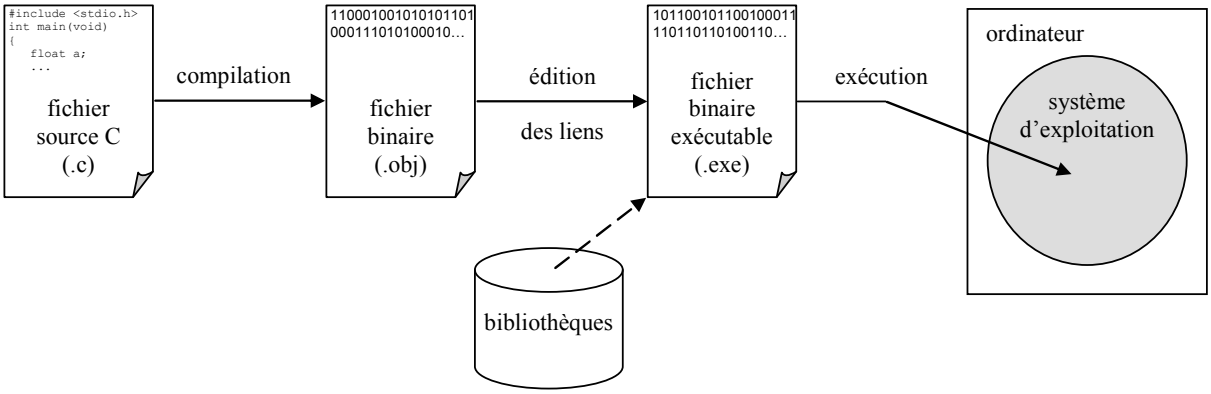

Figure 2.1 : environnement de développement en langage C

La **compilation** consiste à traduire le fichier source (*.c*) écrit en langage C (compréhensible par l'être humain) en code binaire (*.obj*) non exécutable (compréhensible par l'ordinateur, appelé *code machine*).
Cette compilation peut être décomposée elle-même en 3 phases :

- exécution des directives préprocesseur ;
- gestion des références de variables et de fonctions ;
- conversion du programme (sans les fonctions de bibliothèques) en code machine.

L'**édition des liens** (appelée aussi *link*) consiste à charger le code machine des fonctions de bibliothèques (*.obj*) et à l'associer avec le code machine du fichier source (*.obj*) afin de créer un fichier binaire exécutable (*.exe*).

L'**exécution** correspond au traitement, par le processeur, du fichier binaire exécutable qui est chargé au fur et à mesure en mémoire centrale (mémoire vive).

[1] Environnement de Développement Intégré.
[2] Code::Blocks et Dev-C++ (son prédécesseur) sont des EDI gratuits disponibles en téléchargement respectivement à l'adresse http://www.codeblocks.org/ et http://sourceforge.net/projects/dev-cpp/.
[3] Certains compilateurs proposent la possibilité de compiler pour une cible spécifique autre que celle utilisée pour la compilation (autre système d'exploitation par exemple).
[4] Appelée « chaîne de compilation ».

2.1.2 Structure d'un fichier source

En langage C, un fichier source doit avoir l'extension *.c*.

Dans ce fichier, la structure générale est la suivante :

- directives préprocesseur (inclusion de bibliothèques, définition de constantes, etc.) ;
- fonctions (/sous-programmes) ;
- fonction principale :
  - déclaration des variables ;
  - code de la fonction principale (instructions).

La structure d'un fichier source C est donc la suivante :

```
#include <nom_bibliotheque.h>   // nom d'une bibliothèque utilisée

/* autres directives préprocesseur */

/* fonctions */

int main(void)       // début de la fonction principale
{
   // déclaration des variables
   type_variable nom_variable;
   ...

   // instructions à réaliser par le programme (code du programme)
   instruction1(paramètres);
   instruction2();
   ...
}                    // fin de la fonction principale
```

Pour spécifier que plusieurs lignes de code appartiennent au même ensemble (fonction, groupe, etc.), on les encadre par les caractères '`{`' et '`}`' (accolades) ; un ensemble de lignes de code regroupées en utilisant les accolades constitue ce qu'on appelle un *bloc de code*, ou *bloc d'instructions*.

Toute ligne d'instruction située à l'intérieur d'un bloc de code [1] se termine par le caractère '`;`' (point-virgule) ; le retour à la ligne n'est pas interprété par le compilateur comme la fin de la ligne d'instruction [2].

Ex. : Un programme permettant de calculer le prix TTC à partir d'un prix HT saisi par l'utilisateur.

```
#include <stdio.h> // inclusion de la bibliothèque stdio.h (STanDard Input Ouput)
#define TVA 19.6   // définition d'une constante

int main(void)
{
   float HT, TTC;     // déclaration de deux variables

   printf("entrer le prix HT");
   scanf("%f", &HT);
   TTC = HT*(1+(TVA/100));                  } // instructions
   printf("prix TTC %f\n", TTC);
}
```

2.1.2.1 Les commentaires

Afin de faciliter la re-lecture et le débogage du programme, il est vivement conseillé d'insérer des **commentaires** dans son programme ; ces commentaires sont utiles uniquement au programmeur [3], car ils sont ignorés lors de la compilation.

[1] Que ce bloc de code corresponde à une fonction, à la fonction principale, ou aux instructions d'une boucle, d'un test, etc.

[2] Il faut donc être très attentif. Fort heureusement, le compilateur saura détecter les erreurs résultant de cet oubli ; par expérience, lorsqu'un programme, une fois compilé, contient beaucoup d'erreurs, c'est très souvent à cause d'un point-virgule oublié, car toutes les lignes suivant ce point-virgule deviennent alors incompréhensibles pour le compilateur.

[3] De manière générale, il convient de commenter avec pertinence en expliquant l'utilité d'une ou plusieurs lignes de code dans la finalité du programme, plutôt que de surcharger le code source avec des commentaires paraphrastiques. Exemple : pour `if (nbpersonnes < 10)`, on dira « on vérifie que le nombre de personnes ne dépasse pas le quota », plutôt que « on teste si nbpersonnes est inférieur à 10 ».

Deux styles de commentaires sont possibles :

- Mono-ligne [1] : avec la symbolique '`//`' (barre oblique (slash) x 2), suivant la syntaxe `//commentaire`. Le commentaire commence à la symbolique `//` et se termine à la fin de la ligne [2].
- Multi-lignes : avec les symboliques '`/*`' et '`*/`' (barre oblique+étoile et étoile+barre oblique), suivant la syntaxe `/*commentaire*/`. Le commentaire commence à la symbolique `/*` et se termine avec la symbolique `*/`.

2.1.2.2 Principe de « mot-clef »

L'objectif d'un langage de programmation est de proposer un ensemble de concepts et de structures afin de faciliter la mise en place de programmes. Dans ce sens, chaque langage de programmation propose des commandes fondamentales qui font référence à des actions de bas-niveau réalisables par le processeur [3].

Un **mot-clef** représente l'une de ces commandes fondamentales que l'on désire voir être exécutée par le processeur.

2.1.2.3 Principe d'« instruction »

Historiquement, on ne disposait pour programmer que d'un panel de commandes de base du processeur (mots-clefs) et de registres (zones mémoires temporaire) ; on appelait instruction une opération réalisée par le programme, c'est-à-dire l'appel d'une commande de base mettant en jeu des registres [4].

Par extension, le terme d'**instruction** s'applique en langage évolué à toute ligne de code complète (jusqu'au '`;`').

Chaque instruction est traitée l'une à la suite de l'autre, de manière séquentielle donc, telles qu'elles sont définies dans le programme source (du haut vers le bas).

Chaque instruction possède sa propre syntaxe, il convient donc de la respecter scrupuleusement.

2.1.2.4 Principe de « fonction »

En programmation, une **fonction** correspond à un traitement réalisé par le programme, un traitement étant composé de 1 ou plusieurs instructions.

Toute fonction possède sa propre syntaxe [5] et est définie dans une bibliothèque [6] laquelle doit être mentionnée dans l'en-tête du programme pour le bon déroulement de la phase d'édition des liens.

Toute fonction effectue son traitement en utilisant des informations qui lui sont propres et éventuellement aussi en utilisant des informations externes, appelées *paramètres d'entrée* ; ceux-ci permettent d'exécuter l'opération en utilisant des données variables et/ou fournies au moment de l'exécution seulement.

Ex. : Soit une fonction effectuant l'affichage de données à l'écran ; cette fonction a comme paramètres d'entrée les données que l'on désire afficher.

```
printf("ceci est un texte");       // fonction avec 1 paramètre
printf("ceci est le résultat : %d", var1);   // fonction avec 2 paramètres
```

Chaque fonction effectue un traitement qui lui est propre ; si le traitement produit un résultat de *sortie* sous forme alphanumérique, donc exploitable au niveau du code, celui-ci peut être récupéré et ré-utilisé dans le code source.

Pour récupérer le résultat d'une fonction, il faut, au niveau du code, assimiler celle-ci à son résultat ; c'est-à-dire qu'il faut considérer qu'à l'exécution, la fonction sera remplacée par son résultat. Celui-ci peut ainsi être ré-utilisé directement pour l'affectation d'une variable, comme paramètre d'une autre fonction, pour un calcul, etc.

Ex. : Soit une fonction permettant de générer un nombre aléatoire ; le résultat produit sera le nombre généré.

```
rand();            // génération d'un nombre entier aléatoire
nba = rand() + 1;    // utilisation d'un nombre aléatoire dans un calcul
printf("nombre aléatoire : %d", rand());  // affichage d'un nombre aléatoire
```

En résumé, une fonction est un module « opaque » qui effectue un traitement, en utilisant des paramètres d'*entrée* qu'on lui fournit, et qui produit un résultat de *sortie* symptomatique du traitement réalisé.

---

[1] Hérité de manière rétro-ascendante du langage C++ car, historiquement, le standard ANSI C n'autorise pas ce type de commentaires.
[2] Comme ceci est ignoré par le compilateur, le point-virgule final doit être placé en fin de ligne d'instruction, avant le début du commentaire.
[3] Commandes de bas-niveau parmi 5 types : arithmétique, branchement, entrées/sorties, logique ou transfert.
[4] Exemple en assembleur : `ADD AX,[0100]` réalise l'opération suivante : prendre le contenu stocké à l'adresse 0x100, l'ajouter au contenu du registre `AX`, et stocker le résultat dans le registre `AX`.
[5] Certaines fonctions admettent plusieurs syntaxes différentes, le choix adéquat étant laissé au développeur en fonction de ses besoins.
[6] Une bibliothèque permet de regrouper par thématique plusieurs fonctions (cf. 3.3).

Nb : Il est important, dans la syntaxe d'une fonction, de ne pas confondre l'*entrée* et la *sortie* de la fonction [1] :

```
résultat = nom_fonction(paramètres);    // sortie = nom_fonction(entrée)
```

2.1.2.5 Directives préprocesseurs

Une **directive préprocesseur** permet de préciser une règle à appliquer avant la compilation du programme ; elle est définie dans l'en-tête du code source du programme et commence par le symbole '`#`' (dièse).

Il existe divers types de directives, mais on en note deux principales :

- `#include <nom_bibliothèque.h>` : définit le nom d'une bibliothèque de fonctions utilisée ;
- `#define CONSTANTE valeur` : définit une constante ; chaque occurrence du texte `CONSTANTE` [2] dans le code sera substituée au tout début de la compilation par la valeur définie.

Nb : Une directive préprocesseur n'est pas une instruction, elle ne fait pas partie du code final généré. Donc la règle du point-virgule final ne s'applique pas.

2.1.2.6 Fonction principale

Tout programme écrit en langage C doit contenir une et une seule **fonction principale** appelée par convention `main()` [3]. Cette fonction est le point d'entrée du programme et permet ainsi de préciser « où » le programme doit commencer à être exécuté.

Le code de la fonction principale est défini à la suite du nom de la fonction, dans un bloc d'instructions. Sa déclaration est généralement `int main(void)` [4].

2.1.2.7 Principe de « variable »

Toute information utilisée dans un programme est symbolisée par une **variable**, c'est-à-dire un mnémonique alphanumérique [5] associé à un espace mémoire dédié et qui représente la valeur de l'information au moment-même de son interprétation.

Pour que le compilateur comprenne cette mnémonique [6], il faut, avant toute utilisation, la déclarer. La déclaration consiste à lister le nom de chaque variable qui sera utilisée dans le programme et à lui associer un type précis (entier, réel, caractère, etc.) afin de réserver une place en mémoire adaptée à l'information symbolisée.

2.1.2.8 Analyse d'un exemple

Reprenons l'exemple du calcul d'un prix TTC, et analysons chacune des lignes du code source :

```
#include <stdio.h> // utilisation d'une bibliothèque d'entrées/sortie standard
#define TVA 19.6  // définition de la constante TVA

int main(void)     // point d'entrée du programme
{
   float HT, TTC;             // déclaration de deux variables réelles
   printf("entrer le prix HT");    // affichage d'un texte
   scanf("%f", &HT);          // saisie au clavier du prix HT
   TTC = HT*(1+(TVA/100));      // calcul du prix TTC
   printf("prix TTC %f\n", TTC);   // affichage du résultat
}
```

2.2 LES VARIABLES

2.2.1 Définition

Une **variable** correspond à un espace mémoire réservé dont le contenu, c'est-à-dire la valeur de la variable, est susceptible de varier au cours de l'exécution du programme.

Lors de l'exécution, toute occurrence du nom de la variable est substituée par le processeur au contenu de l'espace mémoire qu'elle représente au moment-même où ce nom est « lu ».

---

[1] Cette syntaxe est similaire à celle d'une fonction en mathématiques : y = f(x), la sortie y est fonction de l'entrée x.
[2] Par habitude, le nom d'une constante est écrit en majuscule.
[3] Main (eng) ≡ principal (fr).
[4] cf. 3.2.2.
[5] À l'image de la notion d'inconnue en mathématiques (ex. : soit x le nombre de billes que Jean a gagné […]).
[6] Spécifique au programme, au programmeur, à l'environnement de développement, aux us et coutumes de l'entreprise, etc.

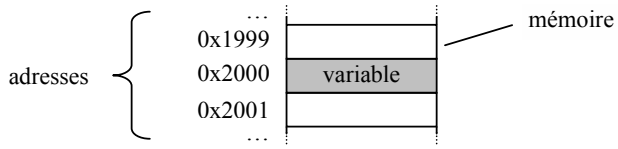

Figure 2.2 : espace mémoire réservé pour une variable

La taille de l'espace mémoire réservé est fonction du type de données que la variable représente. L'adresse d'une variable correspond à sa position dans l'espace mémoire total.

L'espace mémoire est segmenté en zones mémoire d'une taille fixe et qui est voulue minimale. Si une variable, pour être stockée, nécessite plus de place que ne le permet 1 seule zone mémoire, alors 1 ou plusieurs zones mémoire contiguës supplémentaires sont utilisées.

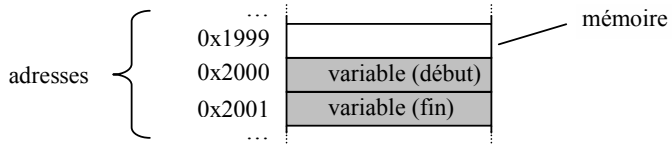

Figure 2.3 : espace mémoire réservé pour une variable nécessitant plusieurs zones mémoire

La notion de variable s'oppose à la notion de **littéral**, qui correspond à toute information exprimée directement : `4`, `-5.49`, `"texte"`, `'c'`, etc. Pour conserver la valeur d'un littéral, il suffit d'utiliser une variable.

2.2.2 Principes de mises en oeuvre

2.2.2.1 Déclaration

La **déclaration** d'une variable consiste à lui donner un nom (texte alphanumérique) ainsi que le type de données auquel elle correspond (entier, réel, caractère, etc.), en suivant la syntaxe `type_variable nom_variable;`.

Ex. : Soit une variable symbolisant l'information « nombre d'élèves » ; le nom choisi est `nbeleves`, et le type est un nombre entier (`int` : type *entier* (INTeger en anglais)).

```
int nbeleves;   // déclaration de la variable nbeleves
```

La déclaration d'une variable garantit que le compilateur lui réserve un espace mémoire spécifique, et ce pour toute la durée d'exécution du bloc de code dans lequel elle est déclarée.

Le nom d'une variable doit respecter les critères suivants :
- nom unique et distinct des mots-clefs du langage C qui sont réservés (cf. annexe A) ;
- nom uniquement composé de caractères alphanumériques ou du caractère souligné (underscore), pas de chiffre en début de nom ;
- longueur maximale de 32 caractères (standard C ANSI) ;
- casse distinguée (minuscules et majuscules différenciées) [1].

2.2.2.2 Affectation

L'**affectation** d'une variable consiste à lui donner une valeur, en utilisant l'opérateur '=' (égal), suivant la syntaxe `nom_variable = valeur;`.

Ex. : `nbeleves = 24;  // affectation de la valeur 24 à la variable nbeleves`

Nb : La valeur à affecter doit être du même type que celle de la variable, sinon le compilateur détectera une erreur [2].

2.2.2.3 Utilisation

L'**utilisation** d'une variable consiste à se servir de la valeur qu'elle contient. Pour cela, il suffit de mentionner son mnémonique à l'endroit du code où l'on désire utiliser sa valeur ; lors de l'exécution du programme, celui-ci sera alors remplacé par la valeur de la variable qu'il représente au moment même où il est interprété par le processeur.

[1] À proscrire cependant, afin d'éviter les confusions (ex. : `I` et `i`).
[2] Par exemple, il est impossible de stocker une valeur réelle dans une variable déclarée entière.

Ex. :
```
int demigroupe;           // déclaration d'une variable entière
demigroupe = nbeleves / 2;      // demigroupe vaut 12 lorsque nbeleves vaut 24
```

2.2.2.4 Initialisation

L'**initialisation** d'une variable consiste à effectuer une première affectation à cette variable ; en effet, toute variable doit être initialisée avant d'être utilisée [1]. On peut déclarer et initialiser une variable en même temps :

Ex. :
```
int nbeleves = 24;        // déclaration et initialisation de la variable nbeleves
int demigroupe = nbeleves / 2;  // déclaration et initialisation de demigroupe
```

2.2.3 Les variables numériques

Le langage C utilise divers types de **variables numériques** [2] :

nom	mot-clef	taille	valeurs extrêmes
entier (court)	`short`	2 octets	$[-2^{15} ; +2^{15}-1]$
entier non-signé (court)	`unsigned short`	2 octets	$[0 ; +2^{16}-1]$
entier	`int`	4 octets	$[-2^{31} ; +2^{31}-1]$
entier non-signé	`unsigned int`	4 octets	$[0 ; +2^{32}-1]$
nombre réel (flottant)	`float`	4 octets	$[\approx \pm 10^{-37} ; \approx \pm 10^{+38}]$
nombre réel double précision	`double`	8 octets	$[\approx \pm 10^{-307} ; \approx \pm 10^{+308}]$

Ex. :
```
int i, j = 3;       // déclaration de 2 variables entières
float x = 1.5;       // déclaration d'une variable réelle
int k = 0x11;      // déclaration d'une variable entière initialisée en
                   // hexadécimal (préfixe de 0x : 0x11 = 17 en décimal)
float y = 2.4e-3;    // déclaration d'une variable réelle initialisée en
                     // utilisant la notation exposant (2.4e-3 = 0.0024)
```

2.2.4 Les caractères

Pour définir une variable de type **caractère**, on utilise le type `char`. Pour l'affectation ou l'initialisation, on utilise le symbole '' (simple côte, nda : apostrophe sur le clavier) pour encadrer le caractère [3].

Ex. :
```
char c = 'A';       // déclaration d'une variable de type caractère
```

Le type `char` correspond en fait à un type *entier* codé sur 1 octet. 256 caractères différents peuvent donc être codés ; l'ensemble des caractères utilisables en langage C est défini par le standard ASCII [4] étendu.

Ex. :
```
char d;
d = 'B';   // d est affecté avec le caractère B (identique à d = 66;)
d = 67;    // d est affecté avec la valeur 67 (identique à d = 'C';)
```

Les 32 premiers caractères de la table ASCII sont des caractères de contrôle, donc non-affichables, mais que l'on peut préciser par une symbolique précise, commençant par le caractère spécial '\' (barre oblique inversée (antislash)). La plupart sont maintenant désuets mais d'autres restent encore très utiles.

caractère	description	valeur (en hexa)
`\n`	saut à la ligne	0x0a
`\r`	retour chariot	0x0d
`\t`	tabulation horizontale	0x09
`\b`	retour arrière	0x08
`\f`	saut de page	0x0c
`\a`	signal sonore	0x07

Ex. :
```
char c = '\n';      // déclaration d'une variable caractère 'saut à la ligne'
```

[1] Le compilateur peut être paramétré pour initialiser automatiquement les variables qui ont été oubliées, mais il est déconseillé de travailler ainsi.
[2] cf. annexe B.
[3] Afin d'éviter la confusion avec une éventuelle variable.
[4] ASCII : American Standard Code for Information Interchange (cf. annexe E).

Le caractère '\' est un caractère spécial en langage C ; il précise que le caractère qui le suit doit être interprété différemment d'un simple caractère alphanumérique. Un certain nombre de caractères ont ainsi une double fonction, selon qu'ils soient précédés de \ ou non.

Ex. :
```
char c = 'r';       // déclaration d'une variable caractère r minuscule
char d = '\r';     // déclaration d'une variable caractère 'retour chariot'
char e = '\\';      // déclaration d'une variable caractère \ (antislash) [1]
```

Pour faire référence directement à un code ASCII, on peut utiliser la syntaxe `\xHH` avec `HH` représentant la valeur hexadécimale du code ASCII du caractère.

2.2.5 Conversion de type

La **conversion de type**, appelée aussi transtypage, utilise l'opérateur cast : `( )` (parenthèse ouvrante et parenthèse fermante), suivant la syntaxe `(nouveau_type) expression_ou_variable.` ;

Ex. :
```
int i = 5, j = 2;
float x, y = 4.6;
x = (float) j;     // x vaut alors 2.0
j = (int) y;       // j vaut alors 4 (seule la partie entière de y est conservée)
y = (float) i/j;   // y vaut alors 1.25
```

2.2.6 Les opérateurs

Divers types d'**opérateurs** peuvent être utilisés en langage C [2].

2.2.6.1 Opérateurs arithmétiques

`+` (addition), `-` (soustraction), `/` (division), `*` (multiplication)
`%` (reste de la division entière, entre deux nombres entiers)

Ex. :
```
j = i * 4;
j = j + 2;
```

2.2.6.2 Opérateurs binaires

`&` (ET), `|` (OU), `!` (NON), `^` (OU exclusif)
`~` (complément à 1)
`<<` (décalage à gauche), `>>` (décalage à droite)
`sizeof(variable)` (renvoie la taille en octets de la variable)

2.2.6.3 Opérateurs combinés et notation condensée

`++` (incrémentation), `--` (décrémentation)
`+=` (ajout et affectation), `-=` (soustraction et affectation), …, `&=` (ET et affectation), etc.

Ex. :
```
i++;          // identique à (i = i + 1) et à (i += 1)
j-= 4;        // identique à (j = j - 4)
```

`var++` (post-incrémentation : exécution de la ligne de code puis incrémentation de la variable)
`++var` (pré-incrémentation : incrémentation de la variable puis exécution de la ligne de code)
`var--` (post-décrémentation), `--var` (pré-décrémentation)

Ex. :
```
i = 4;
j = 4;
k = i++ + 2;    // identique à (k = i + 2; i = i + 1;), k vaut 6
k = ++j + 2;    // identique à (j = j + 1; k = j + 2;), k vaut 7
```

[1] Réponse à la question : « Comment afficher le caractère '\' (antislash) puisque celui-ci est un caractère spécial ? ». Idem pour ''' (apostrophe) qui sert pour encadrer un caractère : `char f = '\'';` et '"' (guillemets) qui sert pour encadrer une chaîne de caractères : `char g = '\"';`.
[2] Liste complète en C.2.

2.2.6.4 Priorité des opérateurs

La priorité des opérateurs est la suivante (du plus prioritaire au moins prioritaire) [1] :

- parenthèses, crochets ;
- incrémentation, décrémentation ;
- multiplication, division, modulo ;
- addition, soustraction ;
- comparaison ;
- affectation.

2.3 AFFICHAGE ET SAISIE

2.3.1 Affichage de texte ou de variable à l'écran

Pour **afficher du texte** et/ou des valeurs de variables à l'écran, on utilise la fonction `printf()` de la bibliothèque *stdio.h* [2], suivant la syntaxe `printf(texte + format, variable1, variable2, ...)`.

Le premier paramètre correspond au texte à afficher ; on précise le format de chaque variable dont on veut afficher la valeur à l'emplacement exact qu'elle doit occuper dans le texte.

Les autres paramètres correspondent, dans l'ordre d'affichage, au nom de chacune des variables à afficher.

Ex. :
```
int cred = 1000, deb = 800;
printf("bilan:\n crédit = %d \ndébit = %d ", cred, deb);   // cred, deb : entiers
```

Résultat de l'exécution à l'écran :
```
bilan
crédit = 1000
débit = 800
```

Pour indiquer le format d'une variable à afficher, on utilise le symbole '`%`' (pourcent) suivi du code spécifique au format de la variable :

- `d` : décimal ;
- `x` : hexadécimal ;
- `u` : entier non signé ;
- `f` : réel ;
- `lf` : double ;
- `e` : réel en notation exposant ;
- `c` : caractère.

De plus, on peut dimensionner l'affichage de la valeur, en incluant certains codes entre le symbole '`%`' et la lettre du format de la variable :

- `X` (`X` nombre entier) : nombre total de symboles à afficher ;
- `0X` (`X` nombre entier) : nombre total de symboles à afficher en complétant à gauche avec des 0 non significatifs par rapport au nombre de chiffres maximal supporté par le format ;
- `.Y` (`Y` nombre entier) : nombre total de décimales à afficher ;
- `M.E` (`M` et `E` nombres entiers) : cas particulier du format d'affichage en notation exposant `%e` → `%M.Ee`) ;
- `+` : signe ('+' ou '–') [3] ;
- `<espace>` : signe (caractère espace pour valeur positive ou '–' pour valeur négative) ;
- `-` : cadrage à gauche.

Ex. :
```
float x = -24.142;
printf("valeur : %6.2f", x); // affichage à l'écran : -24.14 (6 : 6 chiffres
                             // en tout à afficher, .2 : 2 décimales)
```

Dans le cas où le dimensionnement est insuffisant, il est ignoré.

Nb : Pour afficher un caractère, on peut aussi utiliser la fonction `putchar()` / `putc()` suivant la syntaxe `putchar(c);` / `putc(c);` où `c` est une variable de type *caractère*.

[1] cf. C.3.
[2] À mentionner dans les directives préprocesseur donc.
[3] Par défaut, si une valeur positive est affichée, le signe '+' n'est pas précisé, alors que pour une valeur négative, le signe sera précisé.

2.3.2 Saisie de variable au clavier

Pour **saisir la valeur d'une variable**, on utilise la fonction `scanf()` de la bibliothèque *stdio.h*, suivant la syntaxe `scanf(format, &variable)` [1]. La saisie doit être validée par la touche <entrée>.

Le premier paramètre correspond au format de la variable devant être saisie [2] ; ce format doit être spécifié en suivant exactement la même syntaxe que pour la fonction `printf()`.
Le second paramètre correspond au nom de la variable dans laquelle doit être stockée la saisie au clavier ; ce nom de variable doit absolument être précédé du symbole '`&`' (ET commercial) [3].

Ex. :
```
int i;
printf("valeur = ");
scanf("%d", &i);      // i : entier
```

Résultat de l'exécution à l'écran : `valeur = ?`

La fonction `scanf()` est bloquante, c'est-à-dire que lors de l'exécution du programme, celle-ci sera stoppée jusqu'à ce que l'utilisateur ait validé la saisie par la touche <entrée>.

Pour indiquer le format d'une variable à saisir, on utilise les mêmes symboliques de format que pour l'affichage (`%d`, `%x`, `%f`, `%c`, etc.).

Nb : Pour saisir un caractère, on peut aussi utiliser la fonction `getchar()` (validation par <entrée>) / `getch()` (pas de validation) suivant la syntaxe `c = getchar();` / `c = getch();` où `c` est une variable de type *caractère*.

2.4 LES STRUCTURES DE CONTRÔLE

Les **structures de contrôles** permettent de « casser » la linéarité d'exécution d'un programme, en l'adaptant à des besoins spécifiques comme l'exécution conditionnelle d'instructions, la répétition d'instructions, etc.
Ces mécanismes s'appuient généralement sur des valeurs de variables qui ne sont connues que lors de l'exécution du programme ou qui évoluent tout au long de cette exécution ; il faut donc pouvoir tester leurs valeurs.
Pour tester la valeur d'une variable, on écrit une expression qui sera évaluée par le microprocesseur lors de l'exécution du programme ; en fonction du résultat de cette évaluation (*vrai* ou *faux*) l'exécution du programme sera alors modifiée en conséquence.

2.4.1 Les opérateurs d'évaluation d'expression

Pour évaluer une expression, on utilise un ou plusieurs **opérateurs**, parmi 2 types différents :

- opérateurs de tests :
  - `==` : égal [4] ;
  - `!=` : différent ;
  - `>` : strictement supérieur ;
  - `<` : strictement inférieur ;
  - `>=` : supérieur ou égal ;
  - `<=` : inférieur ou égal.
- opérateurs logiques :
  - `&&` : ET ;
  - `||` : OU ;
  - `!` : NON.

[1] Cette syntaxe est simplifiée ; la syntaxe réelle est identique à `printf()` : `scanf(format, &variable1, &variable2, ...);`. C'est-à-dire que l'on peut saisir plusieurs variables avec un seul appel à `scanf()`.

[2] En réalité, ce paramètre correspond à texte+format. Cependant, si l'on indique du texte en plus du format, la saisie devra alors inclure ce texte pour que le stockage de la variable s'opère correctement ; ex. : soit la saisie `scanf("i= %d", &i);` il faut que l'utilisateur tape exactement "`i= 4`" pour que la valeur 4 soit stockée dans la variable `i` ; si l'utilisateur saisit "`4`", cela stockera alors une valeur indéfinie dans la variable (la fonction cherche à lire exactement "`i=` " à partir du tampon d'entrée, puis lit une valeur entière (`%d`) qu'elle stocke dans la variable `i`).

[3] La fonction `scanf()` permet de stocker une valeur saisie directement dans un espace mémoire, et non via une variable ; la notation `&var` renvoie à l'espace mémoire réservé à la variable `var` en utilisant pour l'opérateur '`&`' – aucun lien avec l'opérateur binaire – (cf. chapitre 4).

[4] Attention : l'évaluation de « x vaut-il y ? » s'écrit donc `(x == y)` et pas `(x = y)`, qui est une affectation de la valeur de `y` dans `x`, et qui par définition, vaut *vraie* lorsque tout se passe bien (la valeur de `y` a bien été positionnée dans `x`) ; en résumé, l'« évaluation » `(x = y)` est toujours *vraie* !

Le résultat obtenu est de type booléen (0 si expression fausse, valeur autre que 0 si expression vraie).

Ex. :
```
(i == 4)      // pour vérifier que la variable i est égale à 4
(i != j)      // pour vérifier que la variable i est différente de la variable j
```

Lorsque l'expression doit prendre en compte plusieurs conditions, on utilise les opérateurs logiques, ainsi que les parenthèses pour assurer la priorité d'évaluation [1].

Ex. :
```
((i == 4) && (j > 6))      // vérifie que i vaut 4 et que j est supérieur à 6
```

Nb : Lorsqu'une expression de type *entier* est nulle, elle est considérée fausse ; si elle n'est pas nulle, elle est considérée vraie.

Ex. :
```
(i)           // vérifie que i est différent de 0, identique à (i != 0)
(!i)          // vérifie que i vaut 0, identique à (i == 0)
(i && j)      // vérifie que i et j sont différents de 0
```

2.4.2 Les tests

Les structures de tests permettent de réaliser des **exécutions conditionnelles** ; c'est-à-dire que sur la base de l'évaluation d'une expression (soit donc sur la base de la valeur de une ou plusieurs variables), certaines actions seront exécutées ou pas.

2.4.2.1 Test simple « si… sinon… »

La structure conditionnelle « si… sinon… » réalise un **test simple** : on évalue une expression ; en fonction du résultat de ce test, on exécute alors soit un bloc d'instructions, soit un autre.

L'organigramme associé au test simple est le suivant :

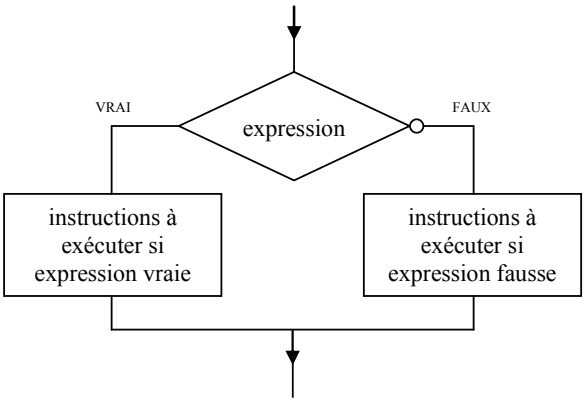

Figure 2.4 : organigramme de la structure conditionnelle « si… sinon »

Pour réaliser un test simple, on utilise l'instruction `if ()` éventuellement associée avec l'instruction `else`, suivant la syntaxe :

```
if (expression)
{
   /* instructions à exécuter si expression vraie */
}
else
{
   /* instructions à exécuter si expression fausse */
}
```

Chaque bloc d'instructions (*expression vraie* et *expression fausse*) doit être encadré par les caractères '`{`' et '`}`' (accolades).

[1] Il existe une priorité naturelle entre les différents opérateurs (cf. C.3).

Ex. : Tester la valeur de la variable `x` ; si `x` vaut 1, on affiche *gagné*, sinon on affiche *perdu*.

```
if (x == 1)
{
   printf("gagne\n");
}
else
{
   printf("perdu\n");
}
```

Lorsque le bloc d'instructions ne comprend qu'une seule instruction, les accolades peuvent être omises.

Ex. :
```
if (x == 1) printf("gagne\n");
else printf("perdu\n");
```

Dans le cas où aucune instruction ne doit être exécutée lorsque l'expression évaluée est fausse, l'instruction `else` n'est pas mentionnée.

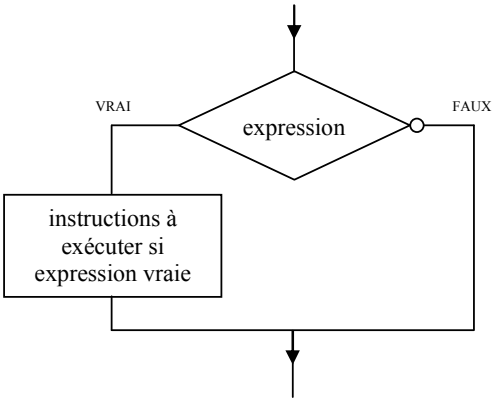

Figure 2.5 : organigramme de la structure conditionnelle « si… »

Ex. : Tester la valeur de la variable `x` ; si `x` vaut 2, on affiche *gagné*, sinon on ne fait rien.

```
if (x == 2)
{
   printf("gagne\n");
}
```

2.4.2.2 Test multiple « au cas où… faire… »

La structure conditionnelle « au cas où… faire… » réalise un **test multiple** : on évalue la valeur d'une variable ; selon certaines valeurs précises de cette variable, sera exécuté un bloc d'instructions spécifique à chaque cas ; lorsque la variable ne correspond à aucun des cas prévus, un bloc d'instructions par défaut peut être exécuté.

Pour réaliser un test multiple, on utilise les instructions `switch ()` et `case` éventuellement associées avec l'instruction `default`, suivant la syntaxe :

```
switch (variable)
{
   case valeur1 :  /* instructions si variable vaut valeur1 */
                   break;
   case valeur2 :  /* instructions si variable vaut valeur2 */
                   break;
   ...
   default :  /* instructions si variable ne vaut aucune des valeurs */
}
```

Nb : Seules des variables de type *entier* ou *caractère* peuvent être utilisées.

Ex. : Création d'un menu avec 2 options.

```
char choix;

printf("1) liste par groupe\n");
printf("2) liste par ordre alphabétique\n");
printf("votre choix ?");
choix = getchar();

switch (choix)
{
   case '1' : printf("liste par groupe");
              break;
   case '2' : printf("liste par ordre alphabétique");
              break;
   default :  printf("choix non prévu");
}
```

2.4.3 L'opérateur ternaire

L'opérateur ternaire peut être utilisé, à la place d'une structure de test à base de `if`, pour affecter rapidement une valeur à une variable.

La syntaxe est la suivante :

```
variable = (expression) ? valeur_si_expression_vraie : valeur_si_expression_fausse;
```

Ex. : `nb = (i > 5) ? 10 : 0;` ⇔

```
if (i > 5)
   nb = 10;
else
   nb = 0;
```

2.4.4 Les boucles

Les boucles permettent de réaliser des **répétitions d'exécution** ; c'est-à-dire que sur la base de l'évaluation d'une expression (soit donc sur la base de la valeur de une ou plusieurs variables), certaines actions seront répétées ou pas.
Une phase de répétition est appelée *itération*.

2.4.4.1 Boucle « tant que… faire… »

La boucle « tant que… faire… » réalise une **répétition avec aucune exécution au minimum** : on évalue une expression ; en fonction du résultat de cette évaluation, soit on exécute un bloc d'instructions, et on revient ensuite à l'évaluation de l'expression, etc., soit on quitte la boucle et on continue l'exécution.

L'organigramme associé à la répétition avec aucune exécution au minimum est le suivant :

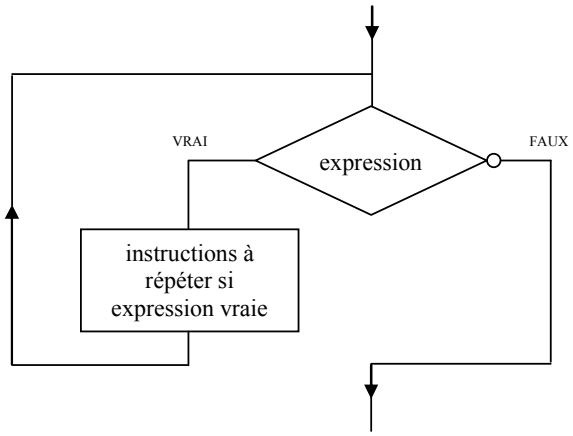

Figure 2.6 : organigramme de la répétition « tant que… faire… »

Pour réaliser une répétition avec aucune exécution au minimum, on utilise l'instruction `while ()`, suivant la syntaxe :

```
while (expression)
{
   /* instructions à répéter si expression vraie */
}
```

Ex. : On veut afficher 10 fois le message *gagné*.

```
int i = 0;
while (i < 10)
{
   printf("gagne\n");
   i++;            // modification de la valeur de i
}
```

Nb : Si l'expression évaluée est fausse avant le début de l'exécution de l'instruction `while ()`, le bloc d'instructions ne sera jamais exécuté, pas même une seule fois.

Attention : Lors de l'exécution d'une boucle, il faut s'assurer que les variables mises en œuvre dans l'expression évaluée et assurant ainsi la continuité ou l'arrêt de la boucle, soient modifiées dans le bloc d'instructions, sans quoi on risque de réaliser ce qu'on appelle une boucle infinie [1]. À moins que l'expression de continuité utilise des valeurs issues de capteurs [2], il faut donc prendre soin de vérifier qu'au moins l'une des variables de l'expression soit modifiée dans les instructions du bloc.

Ex. : Boucle infinie.

```
int i = 0;
while (i < 10)     // i n'est jamais modifié, donc i est toujours inférieur à 10
{
   printf("gagne\n");
}
```

Ex. : Boucle infinie minimale (parfois utile en mode « débogage à la main » [3]).

```
while (1);
```

2.4.4.2 Boucle « répéter… tant que… »

La boucle « répéter… tant que… » réalise une **répétition avec au moins 1 exécution** : on exécute un bloc d'instructions ; puis on évalue une expression ; en fonction du résultat de cette évaluation, soit on revient à l'exécution du bloc d'instructions, soit on quitte la boucle.

L'organigramme associé à la répétition avec au moins 1 exécution est le suivant :

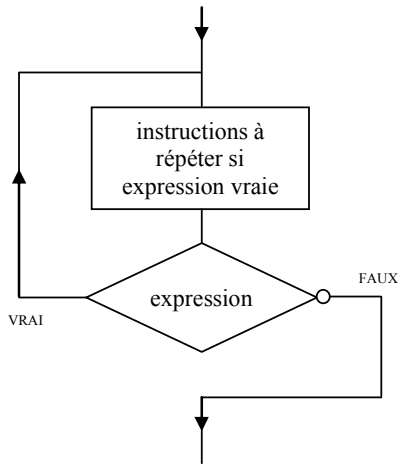

Figure 2.7 : organigramme de la répétition « répéter… tant que… »

[1] Ce qui bloque l'exécution de votre programme, et monopolise le processeur, puisque le même bloc d'instructions est répété à l'infini.
[2] Donc externes au programme lui-même, et susceptibles d'être modifiées même si le programme est « bloqué ».
[3] Permet de positionner un point d'arrêt manuel pour pouvoir visualiser le comportement du programme (affichage de valeurs de variables, etc.)

Pour réaliser une répétition avec au moins 1 exécution, on utilise les instructions `do` et `while ()`, suivant la syntaxe :

```
do
{
   /* instructions à répéter si expression vraie */
}
while (expression);
```

Ex. : On veut afficher 10 fois le message *gagné*.

```
int i = 0;
do
{
   printf("gagne\n");
   i++;            // modification de la valeur de i
}
while (i < 10);
```

Nb : Si l'expression évaluée est fausse avant le début de l'exécution des instructions `do while ()`, le bloc d'instructions sera quand même exécuté une fois [1].

Là encore, les risques de boucle infinie sont identiques à ceux de l'instruction `while ()`.

Ex. : `do { } while (1);  // boucle infinie minimale`

2.4.4.3 Boucle « pour… faire… »

La boucle « pour… faire… » réalise une **répétition contrôlée** : on effectue une initialisation de variables ; ensuite, on évalue une expression ; en fonction du résultat de cette évaluation, soit on exécute un bloc d'instructions, on modifie la valeur de variables, et on répète ensuite l'évaluation de l'expression, soit on quitte la boucle et on continue l'exécution.

L'organigramme associé à la répétition contrôlée est le suivant :

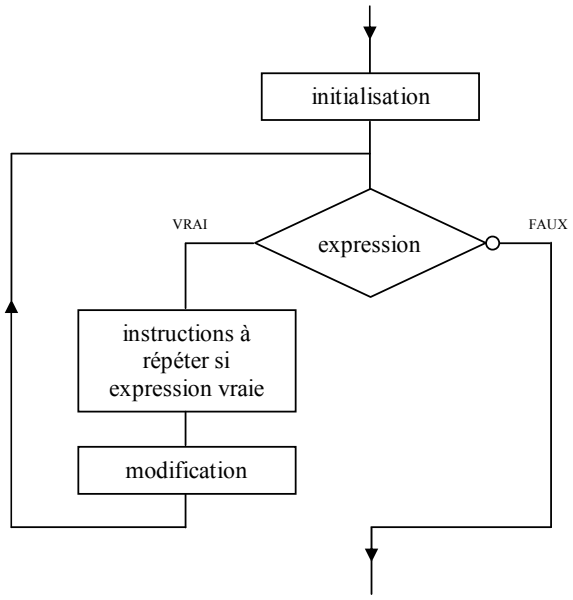

Figure 2.8 : organigramme de la répétition « pour… faire… »

Pour réaliser une répétition contrôlée, on utilise l'instruction `for ()`, suivant la syntaxe :

```
for (initialisation ; expression ; modification)
{
   /* instructions à répéter si expression vraie */
}
```

[1] À l'inverse de la boucle « tant que… faire ».

Ex. : On veut afficher 10 fois le message *gagné*.

```
int i;
for (i = 0 ; i < 10 ; i++)
{
   printf("gagne\n");
}
```

L'initialisation, l'expression à évaluer et/ou la modification peuvent être omises.

Ex. : Boucle infinie minimale.

```
for (;;);
```

2.4.5 Instructions de « déroutement »

2.4.5.1 Branchement à la fin du bloc de code

Pour réaliser un **branchement à la fin du bloc de code**, c'est-à-dire pour sortir immédiatement d'une boucle même si le test de continuité est vrai, ou d'un bloc de code en cours d'exécution, on utilise l'instruction `break`. Typiquement, on peut dire que celle-ci effectue un branchement direct à la prochaine accolade sortante existante.

Ex. : Boucle « presque » infinie.

```
while (1)
{
   if (getch() == '\n') break;  // saut à printf("fini\n"); si on tape <entrée>
   printf("gagne\n");
}
printf("fini\n");
```

2.4.5.2 Branchement au test de continuité

Pour réaliser un **branchement au test de continuité**, c'est-à-dire pour sauter les instructions restantes du bloc d'instructions mais en poursuivant tout de même l'exécution de la boucle (passage à l'itération suivante), on utilise l'instruction `continue`.

Ex. :
```
int i=0;
while (1)
{
   i++;
   if (i == 3) continue;              // saut à while (1) lorsque i vaut 3
   printf("gagne %d\n", i);           // ("gagne 3" ne sera pas affiché)
}
```

2.5 LES TABLEAUX

2.5.1 Définition

Un **tableau** permet de regrouper des données de même type dans un seul ensemble, tout en conservant chacune de ces données directement accessible. Ces différentes données sont stockées dans une zone de mémoire contiguë.

Nb : Les tableaux sont une manière de représenter en programmation les matrices, utilisées en calculs algébriques.

2.5.2 Principes de mises en œuvre des tableaux à 1 dimension

Un **tableau à 1 dimension** peut être considéré comme une liste de données du même type ; il est caractérisé par le nombre de données qu'il contient (appelé *taille* [1]), et le type de ces données.

donnée n°1	donnée n°2	donnée n°3	.....	donnée n°N

Figure 2.9 : représentation d'un tableau à 1 dimension

[1] Parfois aussi appelée *dimension*, à ne pas confondre avec l'appellation *tableau à 1 ou plusieurs dimensions*.

2.5.2.1 Déclaration

La **déclaration** d'un tableau se réalise comme une déclaration de variable, précisée de la taille du tableau en utilisant les symboles '`[`' et '`]`' (crochets), suivant la syntaxe `type_données nom_tableau[taille_tableau];`.

Ex. : Soit un ensemble de 10 variables de type *réel*.

```
float notes[10];  // déclaration d'un tableau de 10 réels
```

La déclaration d'un tableau garantit que le compilateur lui réserve un espace mémoire spécifique [1], et ce pour toute la durée d'exécution du bloc de code dans lequel il est déclaré.

Nb : La taille du tableau est nécessairement une valeur numérique, et ne peut être définie par une autre variable [2].

2.5.2.2 Affectation et utilisation

Pour l'**affectation** ou l'**utilisation** d'un des éléments du tableau, il faut indiquer sa position (appelée *indice* ou *index*) dans le tableau, suivant la syntaxe `nom_tableau[indice]`.

Ex. :
```
float notes[10];    // déclaration d'un tableau de 10 réels
notes[2] = 15.0;  // affectation de la valeur '15.0' à l'indice 2 du tableau
printf("%f", notes[2]);   // affichage de la valeur stockée à l'indice 2
scanf("%f", &notes[6]);   // saisie d'une valeur stockée à l'indice 6
```

Attention : En langage C, le premier indice est l'indice numéro 0 ; pour un tableau d'une taille N, l'indice maximal est donc N-1. Ceci est très important, d'autant plus qu'il n'y a aucune vérification lors de la compilation du programme que l'indice utilisé n'est pas hors des limites du tableau [3].

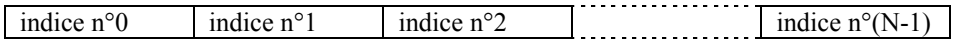

Figure 2.10 : représentation d'un tableau à 1 dimension et des indices

Ex. :
```
float notes[10];    // déclaration d'un tableau de 10 entiers
notes[0] = 15.0;  // affectation du premier élément du tableau
notes[9] = 3.5;   // affectation du dernier élément du tableau
notes[10] = 12.0; // indice hors limites -> erreur d'exécution ou comportement
                  // inattendu du programme
```

Si on ne peut pas utiliser une variable pour déclarer la taille d'un tableau, on peut en revanche utiliser une variable comme indice du tableau.

Ex. :
```
float notes[5];     // déclaration d'un tableau
int i;

for (i = 0 ; i < 5 ; i++) {  // répète pour autant d'éléments que la taille
   printf("valeur %d :\n", i);  // demande la valeur n°i
   scanf("%f", &notes[i]);      // fait saisir la valeur n°i
}
```

Nb : Il est important de savoir si la variable que l'on manipule est du type tableau ou variable simple ; en effet, dans le cas d'un tableau, les éléments sont accessibles avec la syntaxe `tableau[indice]` et la notation `tableau` correspond en fait à l'adresse en mémoire du début du tableau, usuellement appelée *pointeur* [4].

2.5.2.3 Initialisation

L'**initialisation** d'un tableau consiste à effectuer une première affectation de chacun de ses éléments, en utilisant la syntaxe suivante : `type_données nom_tableau[taille_tableau] = {val_0, ..., val_N-1};`.

Ex. : `float notes[5] = {4.0, 12.5, 15.0, 9.0, 16.5};  // initialisation du tableau`

[1] L'espace mémoire réservé correspond à la somme de l'espace mémoire nécessaire à chacune des variables ; dans l'exemple `float notes[10]`, mettant en jeu 10 variables réelles (4 octets), l'espace mémoire total réservé est donc de 10 x 4 = 40 octets.

[2] Néanmoins, en fixant une constante à l'aide de la directive préprocesseur `#define`, on peut définir la taille du tableau à l'aide de cette constante.

[3] L'erreur ne sera alors possiblement détectée qu'à l'exécution ; le risque de construire un exécutable buggé est donc très important.

[4] cf. chapitre 4.

2.5.3 Principes de mises en œuvre des tableaux à plusieurs dimensions

Les **tableaux à plusieurs dimensions** permettent de regrouper des données de même type avec un ordonnancement supérieur à celui des tableaux à 1 dimension ; c'est-à-dire que pour accéder aux valeurs du tableau on utilisera autant d'indices que le tableau a de dimensions.

Nb : De part l'abstraction nécessaire à la gestion des tableaux à plusieurs dimensions, on utilise de manière générale rarement des tableaux de dimension supérieure à 2, parfois 3.

Ex. : Soit un tableau à 2 dimensions ; on dispose donc de 2 indices pour référencer ses éléments (on parle alors généralement de *ligne* et *colonne* [1]) et on peut les représenter suivant une grille.

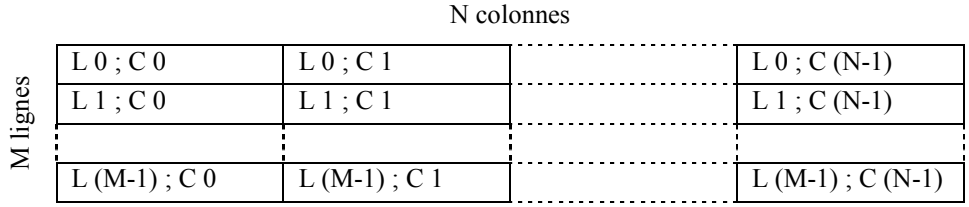

Figure 2.11 : représentation d'un tableau à 2 dimensions

La déclaration, l'affectation, l'utilisation et l'initialisation d'un tableau à plusieurs dimensions suivent les mêmes principes que les tableaux à 1 dimension.

2.5.3.1 Déclaration

La **déclaration** d'un tableau à plusieurs dimensions se fait en utilisant la syntaxe suivante : `type_données nom_tableau[taille_dim_1][taille_dim_2]...[taille_dim_max];`.

Ex. : Soit un tableau à 2 dimensions de taille 2 et 3, contenant des éléments de type *entier*.

```
int valeurs[2][3]; // déclaration d'un tableau d'entiers à 2 dimensions
```

2.5.3.2 Affectation et utilisation

Pour l'**affectation** ou l'**utilisation** d'un des éléments du tableau, il faut indiquer sa position dans le tableau, repérée par les indices, suivant la syntaxe `nom_tableau[indice_dim_1][indice_dim_2]...[indice_dim_max]`.

Ex. :
```
int valeurs[2][3];  // déclaration d'un tableau d'entiers à 2 dimensions
valeurs[1][0] = 15;  // affectation de la valeur '15' à l'indice [1;0]
printf("%d", valeurs[1][0]); // affichage de la valeur stockée à l'indice [1;0]
scanf("%d", &valeurs[0][2]); // saisie d'une valeur stockée à l'indice [0;2]
```

Nb : Là encore, pour chaque dimension, le premier indice est l'indice numéro 0.

2.5.3.3 Initialisation

L'**initialisation** consiste à effectuer une première affectation de chacun des éléments du tableau, suivant la syntaxe :
```
type_données nom_tableau[t_dim_1][t_dim_2]...[t_dim_max] =
{ {val_0_0, ..., val_0_N-1}, {val_1_0, ...}, ...};
```

Ex. :
```
int valeurs[2][3] = {{1,5,7}, {8,4,3}}; // initialisation 2 lignes 3 colonnes
                                        // soit donc d'un couple de triplets

int valeurs[2][3] = {{1,5}, {7,8}, {4,3}};   // erreur !!
```

[1] Même si dans le cas d'un tableau à 2 dimensions on parle de lignes et de colonnes, ce n'est qu'un moyen d'abstraction pour l'être humain. Ce qui veut dire que le premier indice n'est pas spécifiquement représentatif de la ligne ou de la colonne ; c'est au programmeur d'en décider à la déclaration du tableau, puis de s'y tenir tout au long de son programme. En effet, dans la mémoire de l'ordinateur, les informations ne sont pas stockées sous forme de lignes et de colonnes, mais sous forme de liste de valeurs (tableau à 1 dimension de N * M éléments).

2.6 LES CHAÎNES DE CARACTÈRES

2.6.1 Définition

Une **chaîne de caractères** est un ensemble de caractères représentant un mot ou une phrase.

En langage C, une chaîne de caractères est un tableau à 1 dimension de type *caractère*, dont la particularité est de posséder un caractère symbolisant la fin de la chaîne. Un tableau de N caractères ne pourra donc contenir au maximum qu'une chaîne de (N-1) caractères affichables ([0 ; N-2]), le caractère *fin de chaîne*, Nième caractère, se trouvant donc à l'indice (N-1) au maximum.

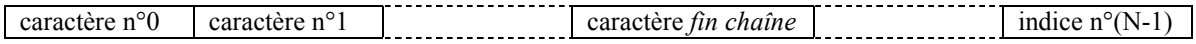

Figure 2.12 : représentation d'une chaîne de caractères

2.6.2 Principes de mise en œuvre

La déclaration, l'affectation, l'utilisation et l'initialisation d'une chaîne de caractères s'opèrent de la même manière que pour un tableau à 1 dimension.

2.6.2.1 Déclaration

La **déclaration** se fait en suivant la syntaxe `char nom_chaine[taille_max];`.

Ex. : Soit une chaîne de caractères de 10 caractères au maximum (soit 11 avec le caractère *fin de chaîne*).

```
char str[11];   // déclaration d'un tableau de 11 caractères
```

Nb : Cela n'est pas important que la chaîne soit plus « petite » que la taille du tableau qui la contient [1], car la fin de la chaîne est précisément connue grâce au caractère *fin de chaîne*, qui est le caractère NUL (noté '`\0`' ou bien `0x00`).

2.6.2.2 Affectation et utilisation

Pour l'**affectation** ou l'**utilisation** d'un des caractères de la chaîne, il faut indiquer sa position dans la chaîne, suivant la syntaxe `nom_chaine[indice]`.

Ex. :
```
char str[11];       // déclaration d'une chaine de 11 caractères
str[2] = 'a';   // affectation du caractère 'a' à l'indice 2 de la chaine
printf("%c", str[2]);   // affichage du caractère stocké à l'indice 2
scanf("%c", &str[6]);   // saisie d'un caractère stocké à l'indice 6
```

On peut afficher directement l'intégralité de la chaîne de caractères jusqu'au caractère *fin de chaîne*, en utilisant la fonction `printf()` associée au format d'affichage `%s`.

Ex. : `printf("%s", str);  // affichage de la chaine complète (jusqu'au caractère 0x00)`

Mais on peut aussi utiliser la fonction `puts()`, spécifique aux chaînes de caractères, suivant la syntaxe `puts(nom_chaine)`.

Ex. : `puts(str);   // identique à printf("%s\n", str);`

En revanche, il est impossible d'affecter directement une valeur à une chaîne, hormis en passant par la saisie ou l'initialisation.

Pour saisir une chaîne, on peut utiliser la fonction `scanf()`, associée au code de format `%s`.

Attention : Comme une chaîne est un tableau, le nom de la chaîne correspond à l'adresse [2] du début du tableau, il ne faut donc pas utiliser l'opérateur '`&`'.

Ex. : `scanf("%s", str);   // saisie d'une chaine`

[1] Cas où la chaîne de caractères n'est pas connue au lancement du programme, ou si elle est susceptible d'être modifiée en cours d'exécution ; il suffit juste de prévoir le dimensionnement en fonction de la taille maximale.

[2] cf. chapitre 4.

Cependant, l'utilisation de la fonction `scanf()` n'est pas recommandée car elle ne permet pas de saisir de chaîne comportant un espace, celui-ci étant considéré comme un délimiteur de fin de saisie.

Il est préférable d'utiliser la fonction `gets()`, suivant la syntaxe `gets(nom_chaine)`.

Ex. : `gets(str);  // saisie d'une chaine pouvant comporter des espaces`

2.6.2.3 Initialisation

L'**initialisation** d'une chaîne de caractères consiste à lui affecter une valeur de départ, c'est-à-dire effectuer une première affectation de chacun des caractères, en utilisant l'une ou l'autre des deux syntaxes suivantes :

- `char nom_chaine[taille_chaine] = "chaine";` ;
- `char nom_chaine[taille_chaine] = {car_0, ..., car_max, '\0'};`.

Ex. : Deux initialisations différentes de chaînes identiques.

```
char str1[10] = "hello";     // initialisation de la chaine
char str2[10] = {'h', 'e', 'l', 'l', 'o', '\0'};  // initialisation de la chaine
```

Nb : Même si une chaîne de caractères est représentée, en langage C, par un tableau de caractères, il ne faut pas confondre ; un tableau de caractères restera toujours un tableau de caractères, et il ne peut être considéré ou utilisé comme chaîne que s'il possède un caractère *fin de chaîne*.

3 LES FONCTIONS

3.1 GÉNÉRALITÉS

Une **fonction** correspond à un traitement réalisé par le programme ; un traitement étant composé de une ou plusieurs instructions. Une fonction est la forme généralisée d'un sous-programme ou procédure.

Toute fonction est définie dans une bibliothèque [1] qu'il faut mentionner dans l'en-tête du programme afin que la phase d'édition des liens puisse trouver le code pré-compilé correspondant.

Chaque fonction possède sa propre syntaxe, il convient donc de la respecter scrupuleusement ; certaines fonctions admettent d'ailleurs plusieurs syntaxes différentes, utilisables à volonté selon les besoins.

Toute fonction effectue son traitement en utilisant des informations internes (propres à la fonction) et éventuellement aussi en utilisant des informations externes, appelées *paramètres d'entrée* ou *arguments* ; ceux-ci permettent d'exécuter le traitement en utilisant des données variables et/ou disponibles seulement au moment de l'exécution.

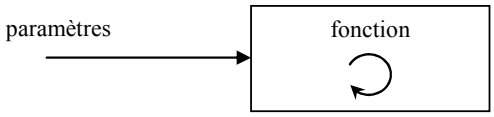

Figure 3.1 : exécution d'une fonction en utilisant des paramètres

En langage C, les paramètres d'entrée sont indiqués à la suite du nom de la fonction, entre parenthèses, séparés les uns des autres par une virgule :

```
nom_fonction(paramètre1, paramètre2, ...);      // exécution de la fonction
```

Ex. : Soit une fonction effectuant l'affichage de données à l'écran ; cette fonction a comme paramètres d'entrée les données que l'on désire afficher.

```
printf("ceci est un texte");       // fonction avec un paramètre
printf("ceci est le résultat : %d", var1);   // fonction avec deux paramètres
```

Chaque fonction effectue un traitement qui lui est propre ; si le traitement produit un résultat de *sortie* sous forme alphanumérique, donc exploitable au niveau du code, celui-ci peut être récupéré et ré-utilisé dans le code source.

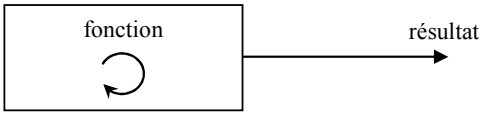

Figure 3.2 : exécution d'une fonction produisant un résultat

En langage C, pour récupérer le résultat d'une fonction, il faut, dans le code, assimiler celle-ci à son résultat ; c'est-à-dire qu'il faut considérer qu'à l'exécution, la fonction sera substituée par son résultat [2]. Le résultat peut ainsi être ré-utilisé directement, pour l'affectation d'une variable, comme paramètre d'une autre fonction, etc.

```
var = nom_fonction();   // exécution de la fonction et affectation du résultat
                        // dans une variable
```

[1] Une bibliothèque permet de regrouper par thématique plusieurs instructions (entrées/sorties, mathématiques, graphiques, …).
[2] Tout comme une variable est remplacée, à l'exécution, par sa valeur.

Ex. : Soit une fonction permettant de générer un nombre aléatoire ; le résultat produit sera le nombre généré.

```
rand();    // génération d'un nombre entier aléatoire
nba = rand() + 1;    // utilisation du résultat de la fonction dans un calcul
printf("nombre aléatoire : %d", rand());  // affichage du résultat de la fonction
```

En résumé, une fonction peut être assimilée à une *boîte noire* [1] qui effectue une opération, en utilisant des paramètres d'*entrée* qu'on lui fournit, et qui produit un résultat de *sortie* symptomatique de l'opération réalisée.

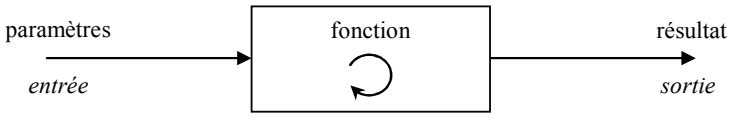

Figure 3.3 : exécution d'une fonction produisant un résultat en utilisant des paramètres

Nb : Il est important, dans la syntaxe d'une fonction, de ne pas confondre l'*entrée* et la *sortie* de la fonction :
`résultat = nom_fonction(paramètres);`

3.2 NOTION DE PROTOTYPE

3.2.1 Définition

Le **prototype** d'une fonction correspond à sa définition complète en tant que module *boîte noire*, c'est-à-dire qu'il faut la qualifier pleinement en spécifiant :

- son nom ;
- le nombre de paramètres d'entrée et le type de chacun d'entre eux ;
- le type de résultat en sortie.

Le nom permet d'appeler la fonction et de l'exécuter ; ce qui correspond à l'exécution du traitement qu'elle réalise. Ce nom suit les mêmes règles que celles liées aux noms de variables, mis à part qu'il sera toujours suivi de parenthèses : `nom_fonction()`.

Ex. : `ma_fonction()` `// fonction ma_fonction()`

Les paramètres d'entrée permettent à la fonction de réaliser un traitement adapté à différents cas de figure [2].
Il faut mentionner le type de chacun d'entre eux, en les énumérant dans l'ordre dans lequel ils devront être passés à l'appel, suivant la syntaxe `nom_fonction(paramètre_1, paramètre_2, ...)`.

```
Ex. : ma_fonction(int, char)     // le premier paramètre est du type int
                                 // le second paramètre est du type char
```

Le compilateur vérifiera que pour chaque appel de la fonction qui est faite, le type de chacun des paramètres est bien respecté. Dans le cas où la fonction n'a pas de paramètres d'entrée, on ne mentionne rien, ou bien `void`.

Le type de résultat en sortie spécifie le type de résultat que la fonction représente. Ce résultat est assimilable à une variable, et peut ainsi être ré-utilisé pour être affiché, pour affecter une variable existante, pour réaliser des calculs, comme paramètre d'entrée d'une autre fonction, etc.

On considère que le type du résultat renvoyé par la fonction correspond au type de la fonction ; et on dira *la fonction est du type ….* Par conséquent, une fonction possède toujours un type, que l'on spécifie suivant la syntaxe `type_résultat nom_fonction()`.
Si la fonction ne renvoie rien, on considère alors que cette fonction est du type `void` [3].

```
Ex. : int ma_fonction()   // fonction renvoyant un résultat de type entier
      void ma_fonction_2()    // fonction ne renvoyant rien
```

Le prototype complet est la synthèse de tous ces éléments : `type_résultat nom_fonction(paramètres)`.

Ex. : `int ma_fonction(int, char)`

[1] En informatique, les notions de *boîte blanche* et *boîte noire* définissent le type de débogage mis en œuvre : respectivement, en connaissant le fonctionnement interne du programme / du sous-programme / de la fonction que l'on teste, ou bien, sans connaître ce fonctionnement interne.
[2] Prémices du concept de ré-utilisabilité – concept extrêmement important en informatique, et très présent en programmation objet.
[3] Void (eng) ≡ vide (fr) ; type spécifique correspondant à « rien » utilisé exclusivement pour les fonctions.

3.2.2 Cas de la fonction principale main( )

Le prototype usuel et standard de la fonction principale `main()` est le suivant : `int main(void)`. Au-delà de son nom imposé [1], cette fonction est une fonction comme les autres.

Ce qui signifie donc que cette fonction n'a pas de paramètres d'entrée, et qu'elle renvoie un résultat entier en sortie, symptomatique, par convention, du bon déroulement du programme (en général `0` si tout s'est déroulé correctement).

En réalité un autre prototype existe : `int main(int argc, char *argv[])` ; ce qui permet, notamment d'inclure des paramètres d'entrée qui seront fournis à l'appel du programme exécutable sur la ligne de commande [2] : `monprog.exe param1 param2 ....`

3.3 FONCTIONS STANDARD ET BIBLIOTHÈQUES

Un certain nombre de fonctions d'utilisation extrêmement courante sont disponibles en C ; on les appelle les **fonctions standard**. Elles sont regroupées par thématique, dans des bibliothèques [3]. Parmi celles-ci, on peut noter :

- `stdio.h` : flux d'entrées/sorties standard ;
- `stdlib.h` : fonctions diverses d'usage très courant (conversion, tri, …) ;
- `conio.h` : flux d'entrées/sorties sur la console ;
- `malloc.h` : gestion mémoire ;
- …

Pour pouvoir utiliser une fonction d'une bibliothèque, l'éditeur de liens doit pouvoir accéder à son code afin de créer un fichier exécutable complet. Il faut donc mentionner, dans les directives préprocesseur, que la bibliothèque décrivant la fonction utilisée doit être importée, avec la syntaxe `#include <nom_bibliothèque.h>`.

Ex. : `#include <stdlib.h>  // importation de la bibliothèque stdlib`

Nb : Le fichier de bibliothèque *.h* [4] contient uniquement le prototype des fonctions ; le lien avec le fichier *.c* ou *.cpp* se fait automatiquement. Cela permet aussi, dès la phase de compilation, de vérifier la syntaxe des fonctions utilisées dans le programme.

3.4 FONCTIONS UTILISATEUR

3.4.1 Principes et écriture

À l'identique des fonctions standard, il est possible de créer ses propres fonctions dans son programme, dites **fonctions utilisateur**. L'intérêt est de décomposer le programme en modules [5], ceci afin de faciliter le développement, la lisibilité et/ou la relecture du programme.

Pour définir une fonction utilisateur, il faut donner son prototype, avec un nom donné à chaque paramètre d'entrée, ainsi que son code, suivant la syntaxe :

```
type nom_fonction(type_param_1 nom_param_1, type_param_2 nom_param_2, ...)
{
   /* code de la fonction */
}
```

Ex. : Soit une fonction permettant de calculer la moyenne de deux nombres réels passés en paramètres et renvoyant le résultat de ce calcul.

```
float moy2f(float nb1, float nb2)
{

}
```

[1] Permettant au compilateur de positionner le pointeur de programme au bon endroit dans la pile d'exécution, ceci afin que le processeur sache où le programme doit commencer à être exécuté.

[2] Les habitués du monde DOS ou UNIX utilisent cela tous les jours (ex. : `dir c:`, `dir a*`, etc.).

[3] Les bibliothèques standard ont été normalisées par le standard C ANSI (cf. annexe D). Lors du développement d'un programme, le respect de cette normalisation améliore la portabilité du code source.

[4] Appelés *fichiers d'en-tête* (voir le langage C++).

[5] Appelés aussi *sous-programmes*.

La spécification des paramètres d'entrée (types et noms) correspond en fait à la déclaration de variables internes à la fonction, qui prennent pour valeurs les valeurs respectives passées en paramètres à l'appel de la fonction. Il s'agit donc d'une recopie de valeur [1], et si celle-ci provient d'une variable, le paramètre demeure une variable bien distincte de la variable d'appel [2].

Ex. : La fonction précédente est appelée dans un programme principal.

```
int main(void)
{
   float x = 10.0;
   moy2f(1.0, 9.0);      // ici, dans moy2f(), nb1 vaudra 1.0 et nb2 9.0
   moy2f(6.0, x);     // ici, dans moy2f(), nb1 vaudra 6.0 et nb2 10.0 (recopie de
                      // la valeur de x)
}
```

Le code d'une fonction utilisateur suit exactement les mêmes règles que n'importe quel code source écrit en C. La déclaration de variables internes, l'appel de fonctions standard (ou d'autres fonctions utilisateurs), l'affichage, la saisie, etc. sont possibles.

Ex. :
```
float moy2f(float nb1, float nb2)
{
   float result;

   result = (nb1 + nb2) / 2;
}
```

Pour déterminer quelle est la valeur que doit retourner la fonction – ce qui correspondra au résultat de sortie –, il faut le spécifier dans le code de la fonction en utilisant l'instruction `return`, suivant la syntaxe `return valeur;`.

Le paramètre de cette instruction doit bien entendu être une valeur ou une variable du même type que celui de la fonction.

Ex. :
```
float moy2f(float nb1, float nb2)
{
   float result;

   result = (nb1 + nb2) / 2;
   return result;  // la fonction est du type float (float moy2f()), on
                   // renvoie un résultat du même type (float result)
}
```

Dans le programme appelant – le programme ayant appelé la fonction – le résultat de la fonction, soit donc la valeur associée au `return`, peut donc être récupérée directement en assimilant le nom de la fonction à ce résultat.

Ex. : La fonction est utilisée dans un programme principal.

```
int main(void)
{
   float x = 10.0;
   float var;
   var = moy2f(1.0, 9.0);     // var vaut le résultat renvoyé par moyenne avec
                              // comme paramètres 1.0 et 9.0
   printf("resultat: %f", moy2f(6.0, x));    // affichage du résultat renvoyé par
                                             // moyenne avec 6.0 et 10.0
}
```

3.4.2 Positionnement au sein du code source

Le code des fonctions utilisateur est écrit en général dans le même fichier que le code de la fonction principale, mais il doit cependant être positionné avant ce dernier ; en effet, lors de la compilation, le programme est compilé dans le sens de la lecture (du haut vers le bas), donc le compilateur doit « connaître » une fonction avant de pouvoir l'utiliser.

[1] On parle de « passage par valeur » – au contrario du passage par adresse, en utilisant les pointeurs (cf. chapitre 4).

[2] Et ce, même si le paramètre porte le même nom que la variable d'appel ; il s'agit de 2 variables distinctes, situées dans 2 blocs de code distincts, donc occupant 2 espaces mémoires bien distincts.

Ex. :
```
#include <stdlib.h>

float moy2f(float nb1, float nb2)
{
   float result;

   result = (nb1 + nb2) / 2;
   return result;
}

int main(void)
{
   float x = 10.0;

   printf("%f", moy2f(6.0, x));
}
```

Cette règle du « sens de lecture » à la compilation doit aussi être respectée lorsqu'une fonction utilisateur utilise une autre fonction utilisateur.

Une autre possibilité consiste à mentionner, en début de programme, les prototypes de toutes les fonctions utilisateur ; le compilateur « connaît » ainsi toutes les fonctions utilisateur avant de compiler quoi que ce soit [1].

Ex. :
```
void ma_fonction();
void ma_fonction_2();


void ma_fonction()
{
   ...
}

void ma_fonction_2()
{
   ...
}

int main(void)
{
   ...
}
```

Enfin, le code d'une fonction utilisateur peut être écrit dans un autre fichier *.c* que celui du programme principal ; pour pouvoir appeler la fonction à partir de ce dernier, il suffit de mentionner le chemin du fichier en tant que directive préprocesseur d'inclusion, suivant la syntaxe `#include "chemin_fichier.c"`.

Ex. :
```
#include "C:\\Mes_Progs\\ma_lib.c"      // double '\\' car caractère spécial
```

3.4.3 Variables locales et globales

On appelle **variable locale** une variable déclarée à l'intérieur d'un bloc de code, soit donc à l'intérieur d'une fonction. Son domaine d'existence est le bloc de code dans lequel elle est définie ; elle n'est connue que de celui-ci, et pas des autres blocs de code de même niveau ou de niveau supérieur. Un paramètre d'entrée est considéré comme une variable locale.

Inversement, on appelle **variable globale** une variable déclarée à l'extérieur de tout bloc de code – et de fait, de toute fonction. Son domaine d'existence est l'ensemble du programme ; elle est donc connue de toutes les fonctions [2].

Nb : Il est possible de déclarer une variable locale de même nom qu'une variable globale existante ; c'est alors la variable locale qui a « priorité ».

[1] Cette pratique a aussi l'avantage de présenter tous les modules du programme en début de fichier source.
[2] Au sens de lecture haut → bas près.

Ex. : `#include <stdlib.h>`

```
int i = 2, j = 4;    // déclaration de 2 variables globales

void ma_fonction(int m) // déclaration de 1 paramètre ≡ 1 variable locale
{
   int k = 5;      // déclaration de 1 variable locale
   float j = 6.2;  // déclaration de 1 variable locale de même nom qu'1 variable
                   // locale : les 2 variables restent différentes

   m = 7;
   i++;            // incrémentation de la variable globale i
   printf("%d", i);     // affichage : 3
   printf("%f", j);     // affichage : 6.2 (le 'j' local "masque" le 'j' global)
}

int main(void)
{
   int m = 6;

   ma_fonction(m);   // appel de ma_fonction() avec la valeur 6 comme paramètre
   printf("%d", i);     // affichage : 3 ('i' global, modifié par ma_fonction())
   printf("%d", j);     // affichage : 4 ('j' global)
   printf("%d", m);     // affichage : 6 ('m' local à main(), le 'm' local à
                        // ma_fonction() n'est pas accessible dans main())
   printf("%d", k);     // erreur de compilation : 'k' n'existe pas dans main()
}
```

Nb : On évite dans la mesure du possible de créer des variables globales ; en effet celles-ci sont extrêmement gourmandes en mémoire (la place mémoire est réservée du début jusqu'à la fin de l'exécution du programme, aucune libération de mémoire n'est possible en cours d'exécution).

De manière générale, même si cela pourrait paraître parfois plus pratique, il est déconseillé de déclarer dans un programme 2 variables portant le même nom :

- 1 paramètre d'une fonction / la variable d'appel associée ;
- 1 variable globale / 1 variable locale « cachant » la variable globale ;
- 2 variables de même nom mais distinctes car déclarées dans 2 fonctions différentes ;
- 1 variable / 1 autre variable avec le même nom mais avec une différence de casse ;
- …

Si cela n'a aucune conséquence sur la compilation et sur la qualité de la programmation, cela est source de grande confusion pour le programmeur et est néfaste pour la lisibilité du code source.

4 LES POINTEURS

4.1 INTRODUCTION

Toute information est stockée sous forme de variable ; c'est-à-dire qu'un espace mémoire est réservé pour stocker sa valeur. À cet espace mémoire correspondent donc un *contenu* (la valeur) et un *contenant* (l'adresse) lequel référence sa position dans l'espace mémoire total.

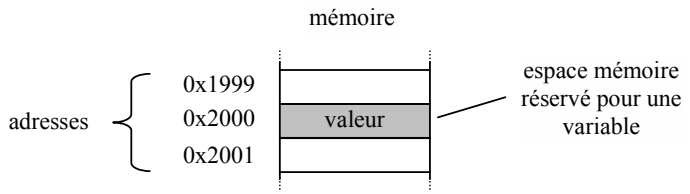

Figure 4.1 : espace mémoire occupé par une variable

En langage C, lors de l'utilisation d'une variable, cette gestion de la mémoire (les adresses utilisées par les variables) est transparente pour le programmeur [1]. Il est cependant possible de travailler directement avec des espaces mémoire, c'est-à-dire avec des adresses plutôt qu'avec des variables.

L'adresse utilisée par une variable est fixée par le système d'exploitation [2] et est identique tout au long de l'exécution du programme. Autant il est toujours possible de connaître l'adresse d'une variable, autant il est impossible de la modifier en cours d'exécution ; il est par contre possible de la fixer au départ, avant toute utilisation, mais cela est déconseillé sauf en cas de réel besoin [3].

4.2 GÉNÉRALITÉS

4.2.1 Les opérateurs

On peut prendre connaissance de l'adresse utilisée par n'importe quelle variable déclarée dans le programme, en utilisant l'**opérateur d'adresse** `&` [4] suivant la syntaxe `&variable`.

On fait ainsi référence au *contenant*, et non plus au *contenu*.

Ex. :
```
float va = 4;       // déclaration et initialisation d'une variable
printf("contenu de va: %f", va);  // affichage du contenu
printf("adresse de va: %d", &va); // affichage de l'adresse
printf("adresse de va: %p", &va); // affichage de l'adresse en hexa "pointeur"
```

Nb : L'adresse d'une variable, et ce quelque soit son type, est nécessairement une valeur de type entière (donc affichable avec les formats `%d`, `%x`, `%p`).

Inversement, à partir d'une adresse, on peut accéder à la valeur qui y est stockée, en utilisant l'**opérateur d'indirection** `*` [5] (opérateur de contenu) suivant la syntaxe `*adresse`.

On accède ainsi au *contenu* à partir du *contenant*.

[1] Ce n'est pas le cas de l'assembleur où l'on travaille principalement avec des espaces mémoires, et pas des variables.
[2] En fonction des autres programmes en cours d'exécution ; l'adresse est donc susceptible d'être différente entre 2 exécutions distinctes.
[3] Notamment lors d'accès à du matériel, afin d'effectuer des opérations d'entrées/sorties dans des registres positionnés à des adresses bien spécifiques ; l'inconvénient est que le code produit perd toute portabilité et doit être récrit pour être adapté à une nouvelle plate-forme.
[4] À ne pas confondre avec l'opérateur binaire `&` – ET logique – (cf. C.2).
[5] À ne pas confondre avec l'opérateur binaire `*` – multiplication – (cf. C.2).

Ex. :
```
float *pb;          // déclaration d'une adresse sur valeur réelle
printf("contenu de pb: %f", *pb); // affichage du contenu de l'adresse
printf("adresse de pb: %d", pb);     // affichage de l'adresse
printf("adresse de pb: %p", pb);     // affichage de l'adresse en hexa
```

Les opérateurs d'adresse et d'indirection réalisent donc 2 opérations inverses, et sont liés par la relation : si *adresse* = *&variable*, alors ***adresse* = *variable* (la réciproque est inexacte). De là, on en déduit que : `adresse = &*adresse`, et `variable = *&variable`.

4.2.2 Définition

On appelle **pointeur** une variable représentant une adresse mémoire ; on dit que le pointeur *pointe* sur cette adresse.

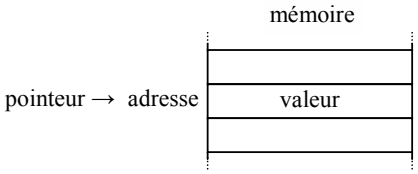

Figure 4.2 : valeur d'une variable dont l'adresse est pointée par un pointeur

Le principal intérêt des pointeurs est de travailler directement avec les adresses, c'est-à-dire avec la mémoire, avec la possibilité à loisir de lire ou écrire [1] n'importe quelle adresse. On s'affranchit ainsi des déclarations de variables figées et on peut alors gérer de manière dynamique le stockage des données utilisées lors de l'exécution d'un programme, aussi bien pour la réservation de l'espace mémoire nécessaire aux données que pour sa libération.

4.2.3 Principes de mise en œuvre

4.2.3.1 Déclaration

La **déclaration** d'un pointeur consiste à définir son nom ainsi que son type, qui correspond au type de la valeur pointée ; un type pointé est un type distinct d'un type classique et il est noté `type*` suivant la syntaxe `type *pointeur` ou `type* pointeur` [2]. On ne parle plus alors de *variable entière*, *réelle*, etc. mais de *pointeur d'entier*, de *réel*, etc. [3].

Ex. :
```
float *pf;    // déclaration d'un pointeur de type réel
int *pi;      // déclaration d'un pointeur de type entier
```

4.2.3.2 Affectation

L'**affectation d'une valeur** à un pointeur s'opère de la même manière que pour une variable, en faisant référence au contenu et en lui affectant une valeur, en suivant la syntaxe `*pointeur = valeur`.

Ex. :
```
float *pf;
int *pi, var=7;
*pf = 5.2;    // affectation de la valeur 5.2 à l'adresse pointée par pf
*pi = var;    // copie de la valeur de var à l'adresse pointée par pi (≠ pi=&var;)
```

L'**affectation d'une adresse** peut être réalisée de plusieurs manières :

- récupération de l'adresse d'une variable déjà déclarée, suivant la syntaxe `pointeur = &variable`, ou bien d'un pointeur déjà existant suivant la syntaxe `pointeurA = (type_pointeurA*) pointeurB` ;
- affectation directe de l'adresse, suivant la syntaxe `pointeur = (type_pointeur*) adresse`.

Ex. :
```
int *pj, *pk;              // déclaration de deux pointeurs de type entier
float *pg;           // déclaration d'un pointeur de type réel
pj = &var;           // pointeur pj pointe sur l'adresse de var (≠ *pj=var;)
pg = (float*) pj;       // pointeur pg pointe sur la même adresse que pj
pk = (int*) 0x3000;        // pointeur pk pointe sur l'adresse 0x3000
*pk = 13;            // affectation de la valeur 13 à l'adresse pk
```

[1] Avec les risques que cela suppose (ex. : on écrit dans une zone mémoire réservée à la gestion de la carte graphique → au mieux : erreur d'exécution et arrêt du programme incriminé, au pire : défaillance générale du système d'exploitation).

[2] `type* ≠ type`.

[3] En termes de *type*, un pointeur d'entier (`int*`) n'est pas du même type qu'un entier (`int`), idem pour un pointeur de réel et un réel, etc.

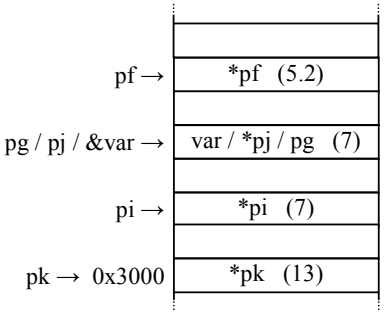

Figure 4.3 : pointeurs, adresses et contenus

Attention : La copie de l'adresse (`pointeur = &variable`) induit que la valeur pointée sera identique (`*pointeur = variable`) ; en revanche, la copie du contenu (`*pointeur = variable`) n'effectue aucune modification sur l'adresse.

4.2.3.3 Initialisation

L'une des principales caractéristiques des pointeurs est la possibilité de créer des variables dynamiquement lors de l'exécution. La conséquence directe est que, par défaut, aucun espace mémoire n'est réservé pour un pointeur.

Nb : Si aucun espace mémoire n'est spécifiquement réservé à un pointeur, alors son utilisation est très risquée, car on risque de travailler avec des zones mémoire utilisées par d'autres variables du programme, voire même des données du système d'exploitation [1], ou bien de travailler à un espace mémoire pour lequel il n'y a aucune garantie d'exclusivité, et qui peut donc être modifié à tout moment par une tierce application.

L'**initialisation** d'un pointeur consiste à lui affecter un espace mémoire réservé. Deux cas sont possibles :

- association du pointeur à un espace mémoire déjà réservé ; on utilise pour cela une variable (espace réservé automatiquement par le compilateur), ou bien un autre pointeur dont l'espace mémoire est déjà réservé ;

  Ex. :
  ```
  float *ph, nb=2.7;   // déclaration ph : aucun espace mémoire réservé
  int *pm;
  ph = &nb;          // ph pointe sur l'adresse de nb, espace mémoire déjà
                     // réservé lors de la déclaration "float nb"
  pm = (int*) ph;    // pm pointe sur l'adresse ph (espace mémoire déjà réservé)
  ```

- réservation explicite d'un nouvel espace mémoire ; on réalise pour cela une allocation de mémoire en utilisant la fonction `malloc()` de la bibliothèque *malloc.h* suivant la syntaxe `pointeur = (type_pointeur*) malloc(taille)`, où la taille est spécifiée en octets, et la valeur renvoyée est l'adresse à laquelle débute l'espace mémoire nouvellement réservé.

  Ex. :
  ```
  char *pn;              // déclaration d'1 pointeur sur char
  pn = (char*) malloc(sizeof(char));      // allocation d'espace mémoire
                                          //pour 1 char
  ```

  Nb : On utilise en général l'opérateur `sizeof()` afin de faciliter l'allocation de l'espace mémoire [2].

  Cette syntaxe, permet par ailleurs de réserver plusieurs zones mémoire successives, sans réelle limitation.

  Ex. :
  ```
  char *po;              déclaration d'1 pointeur sur char
  po = (char*) malloc(3*sizeof(char));    // allocation d'espace mémoire
                                          // pour 3 char
  ```

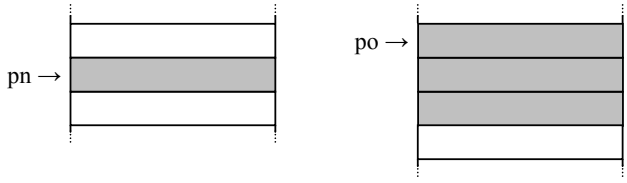

Figure 4.4 : réservation explicite d'un espace mémoire

[1] Ce qui se traduit dans le meilleur des cas par un programme buggé (lorsqu'on peut le déceler), ou dans le pire des cas par un dysfonctionnement du système d'exploitation.

[2] Et afin de soustraire aussi aux différences entre plates-formes ou compilateurs en ce qui concerne la taille allouée à chaque type de variable.

Lorsqu'un pointeur n'est pas initialisé, il vaut `NULL` [1].

Ex. :
```
int *pn;
if (pn == NULL)                          // allocation d'espace mémoire uniquement
   pn = (int*) malloc(sizeof(int));      // lorsque cela n'a pas déjà été fait
```

Tout espace mémoire réservé explicitement doit être libéré explicitement [2]. La libération de l'espace mémoire préalablement réservé à un pointeur se fait en utilisant la fonction `free()` de la bibliothèque *malloc.h* selon la syntaxe `free(nom_pointeur)`.

Ex. :
```
free(po);     // libération de l'espace mémoire réservé pour po
```

Nb : En réalité, la syntaxe `(type_pointeur*) malloc(taille)` (transtypage de la valeur d'adresse renvoyée par la fonction) peut être simplifiée en `malloc(taille)` (aucun transtypage) si on a prit la peine d'inclure la bibliothèque *stdlib.h* ; celle-ci permet en effet de définir la fonction `malloc()` comme étant une fonction de type `void*` et non `int*` [3].

4.2.4 Arithmétique des pointeurs

4.2.4.1 Principes

Comme avec les pointeurs on travaille directement avec les adresses, il est possible de modifier l'adresse que l'on souhaite pointer et accéder ainsi au contenu d'un autre espace mémoire, en se basant sur un pointeur déjà initialisé.

Un pointeur correspondant à une adresse, soit donc une valeur de type *entier* ; on peut à loisir l'incrémenter, le décrémenter, lui ajouter ou lui soustraire une valeur, etc., ceci afin de pointer sur une adresse mémoire différente.

Ex. :
```
char *pr, *ps;              // déclaration de deux pointeurs sur caractère
pr = (char*) 0x3000;    // pointeur pr pointe sur l'adresse 0x3000
ps = pr+1;           // pointeur ps pointe sur l'adresse pr+1 (0x3001)
printf("contenu de pr: %d", *pr); // affichage du contenu stocké à 0x3000
printf("contenu de ps: %d", *ps); // affichage du contenu stocké à 0x3001
printf("contenu de pr+2: %d", *(pr+2)); // affichage du contenu stocké à 0x3002
```

Attention : Il est important d'utiliser les parenthèses à bon escient – voire à chaque fois – dans l'écriture arithmétique des pointeurs. En effet, l'opérateur d'indirection (`*`) est prioritaire par rapport aux opérateurs arithmétiques, mais n'est pas prioritaire par rapport aux opérateurs d'incrémentation/décrémentation.

Ainsi `*pointeur+2` signifie `(*pointeur)+2` (ajout de 2 au contenu du pointeur) alors que `*pointeur++` signifie `*(pointeur++)` (accès au contenu de pointeur puis incrémentation de pointeur).

4.2.4.2 Application à chaque type de variable

L'espace mémoire utilisé par un programme est segmenté en zones mémoire d'une taille fixe et minimale. Comme chaque type de variable possède un espace mémoire d'une taille spécifique [4], il peut être nécessaire d'utiliser plusieurs zones mémoire contiguës pour stocker une seule et même variable.

Le plus « petit » type étant le type *caractère* [5], la taille d'une zone mémoire minimale est fixée sur ce type [6]. Une seule zone mémoire est donc utilisée pour stocker une variable de type *caractère* ; alors que plusieurs sont nécessaires pour les autres types.

nom	mot-clef	taille	nombre de zones mémoire
caractère	`char`	1 octet	1
entier (court)	`short`	2 octets	2
entier	`int`	4 octets	4
nombre réel	`float`	4 octets	4
nombre réel double précision	`double`	8 octets	8

[1] Mot-clef. Le pointeur `NULL` correspond en réalité à l'adresse 0x00, adresse jamais allouée et toujours valide.
[2] Convention de programmation : autant de `free()` que de `malloc()`, pas 1 de moins, pas 1 de plus.
[3] Le type `void*` est compatible avec tout type pointeur, au contrario du type `int*` qu'il faut donc transtyper pour initialiser convenablement l'adresse de début d'un pointeur autre que `int*`.
[4] cf. 2.2.
[5] Un caractère étant en fait un entier – codé sur 8 bits en C – (cf. 2.2.4).
[6] C'est souvent le cas dans les autres langages aussi, même si cela est bien souvent caché (Java, C#, …).

Lorsque le type d'une variable nécessite plusieurs zones mémoire, l'adresse de la variable correspond à la première adresse mémoire utilisée (la plus basse). Les octets constituant la variable, quant à eux, sont stockés selon les adresses croissantes suivant un ordre allant des poids faibles vers les poids forts (type *little-endian* [1]), ou bien selon un ordre allant des poids forts vers les poids faibles (type *big-endian* [2]), en fonction du type de processeur et/ou de système d'exploitation utilisé [3].

Nb : Dans la suite de notre étude, nous considérerons que c'est le modèle little-endian qui est utilisé, car il correspond au type de processeur utilisé le plus couramment dans la vie de tous les jours [4].

Ainsi 2 variables de même type stockées successivement en mémoire, sont séparées de X zones mémoire, où X représente la taille allouée pour le type de ces variables, et leurs adresses sont donc distantes de X octets.

En conséquence, lorsque l'on accède à un espace mémoire en se basant sur un pointeur déjà initialisé, cet accès est réalisé en concordance avec le type de variable pointé ; ceci afin d'accéder à un espace mémoire qui correspond à une variable complète du type pointé, et pas à 2 parties incomplètes de 2 variables différentes mais contiguës en mémoire.

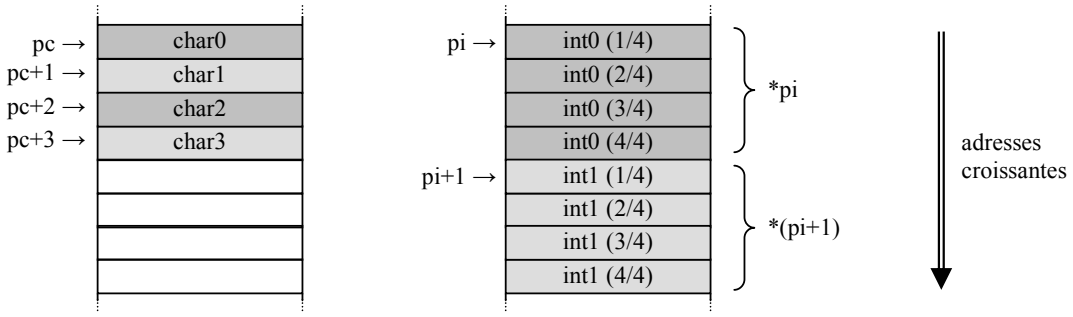

Figure 4.5 : arithmétique des pointeurs et zones mémoire par type de variable

Ex. : Trois pointeurs de différents types dont on modifie l'adresse pointée par incrémentation.

```
char *pc;       // déclaration d'un pointeur sur caractère
int *pi;                  // déclaration d'un pointeur sur entier
double *pdo;              // déclaration d'un pointeur sur nombre réel double

pc = (char*) malloc(2*sizeof(char));    // réservation espace mémoire pour 2 char
pi = (int*) malloc(2*sizeof(int));   // réservation espace mémoire pour 2 int
pdo = (double*) malloc(2*sizeof(double)); // réservation espace mémoire 2 double

printf("char0 : %p – char1 : %p", pc, pc+1);    // si pc vaut 0x2000,
                                                // alors (pc+1) vaut 0x2001
printf("int0 : %p – int1 : %p", pi, pi+1);      // si pi vaut 0x3000,
                                                // alors (pi+1) vaut 0x3004
printf("doub0 : %p – doub1 : %p", pdo, pdo+1); // si pdo vaut 0x4000,
                                                // alors (pdo+1) vaut 0x4008

free(pc);       // libération de l'espace mémoire réservé pour pc
free(pi);       // libération de l'espace mémoire réservé pour pi
free(pdo);   // libération de l'espace mémoire réservé pour pdo
```

4.2.4.3 Souplesse d'utilisation

Avec les pointeurs on manipule des adresses et les zones mémoire correspondant à ces adresses ; par conséquent, rien n'interdit d'utiliser des pointeurs différents pointant sur des zones mémoire communes.

[1] Les processeurs type Intel et Alpha, et les systèmes d'exploitation Windows et Linux sont du type little-endian.
[2] Les processeurs Sun et Motorola, et le système d'exploitation MacOS sont du type big-endian.
[3] Cela a pour conséquence qu'un entier de type `int` (4 octets = 32 bits), initialisé à 0x12345678 par un processeur/OS 32 bits de type little-endian, sera lu 0x78563412 par un processeur/OS 32 bits de type big-endian, et vice-versa.
[4] On pourrait alors dire que le LSB (Less Significant Byte – Octet le moins significatif) de la variable est stocké à la première adresse ; le MSB (Most Significant Byte) est stocké à la dernière adresse. S'il y a un bit de signe, celui-ci sera donc le dernier bit du dernier octet.

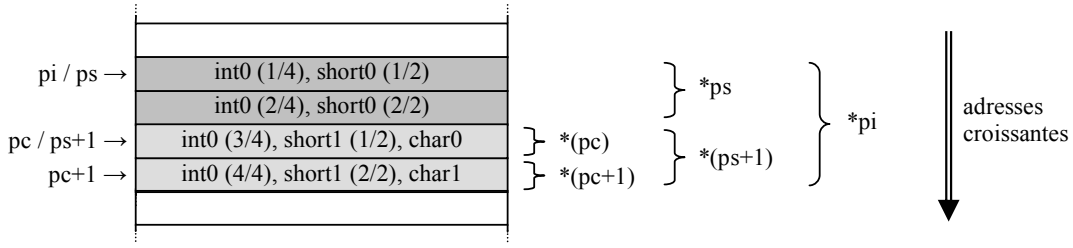

Figure 4.6 : pointeurs de différents types pointant sur des zones mémoire communes

Ex. : Trois pointeurs de différents types pointant sur des zones mémoire communes.

```
short *ps;   // déclaration d'un pointeur sur entier court
int *pi, i;          // déclaration d'un pointeur sur entier
unsigned char *pc;   // déclaration d'un pointeur sur caractère (non-signé)

ps = (short*) malloc(2*sizeof(short)); // réservation espace mémoire pour 2 short
pi = (int*) ps;              // pi pointe sur ps
pc = (char*) (ps+1);         // pc pointe sur (ps+1)

*ps = 0x4321;
*(ps+1) = 0x8765;

for (i=0 ; i<2 ; i++)
   printf("short%d : %x\n", i, *(ps+i));  // affichage de 0x4321, 0x8765

printf("int0 : %x\n", *pi);     // affichage de 0x87654321

for (i=0 ; i<2 ; i++)
   printf("char%d : %x\n", i, *(pc+i));    // affichage de 0x65, 0x87

free(ps);    // libération de l'espace mémoire réservé pour ps
```

Nb : Il faut noter que c'est une erreur (en plus d'être inutile) de vouloir libérer l'espace mémoire des pointeurs `pi` et `pc` ; en effet, aucune allocation spécifique n'a été faite (pas de `malloc()`) car ils pointent sur le même espace mémoire que `ps`. Donc lorsque l'espace mémoire réservé pour `ps` a été libéré, tous les espaces mémoire réservés explicitement ont été libérés ; libérer l'espace mémoire de `pi` ou `pc` ne sert à rien et peut même générer une erreur d'exécution.

4.3 POINTEURS ET TABLEAUX

4.3.1 Introduction

En utilisant les pointeurs, on a la possibilité d'utiliser des espaces mémoires contigus, sans limitation, ainsi que de réserver des zones mémoire contiguës spécifiques à notre programme. De plus, l'arithmétique des pointeurs permet de « naviguer » parmi les zones mémoire en corrélation directe avec le type de données que l'on manipule.

Il est donc possible d'utiliser les pointeurs pour gérer un ensemble de données de même type auxquelles on accède à partir d'une seule adresse que l'on « déplace » selon son besoin.

En conséquence, un pointeur constitue une solution alternative à l'utilisation d'un tableau.

Ex. : Un pointeur permettant d'accéder à 3 données réelles pour lesquelles on réserve l'espace mémoire nécessaire.

```
float *pf;         // déclaration d'un pointeur sur réel
int i;

pf = (float*) malloc(3*sizeof(float)); // réservation espace mémoire pour 3 réels
*pf = 5.3;
*(pf+1) = -98.12;
*(pf+2) = 636.932;

for (i=0 ; i<3 ; i++)
   printf("valeur no %d : %f\n", i, *(pf+i));   // affichage de chaque valeur

free(pf);          // libération de l'espace mémoire
```

4.3.2 Analogies

En réalité, un pointeur et un tableau sont très proches dans leur utilisation. On peut ainsi utiliser un pointeur pour accéder aux données d'un tableau ; la réciproque n'est pas toujours vraie.

De la même manière qu'il est possible d'accéder à l'adresse d'une variable à l'aide de l'opérateur `&`, il est possible de connaître l'adresse d'un tableau en se basant sur l'adresse de la première donnée du tableau, en utilisant la syntaxe `&tableau[0]`, ce qui est identique à l'écriture `tableau` [1].

L'adresse de chaque élément d'un tableau est alors obtenue en utilisant la syntaxe `&tableau[indice]`, et on en déduit que les écritures `*(pointeur+indice)` et `tableau[indice]` permettent ainsi d'accéder à la même zone mémoire [2].

Ex. : Un pointeur et un tableau permettant d'accéder à des données identiques.

```
float *pf;         // déclaration d'un pointeur sur réel
float tabf[3];     // déclaration d'un tableau (espace mémoire réservé)
int i;

pf = tabf;           // pf pointe sur l'adresse du début du tableau (→ pf=&tab[0])
tabf[0] = 5.3;
tabf[1] = -98.12;
tabf[2] = 636.932;

for (i=0 ; i<3 ; i++)
   printf("valeur no %d : tabf %f - pf %f\n", i, tabf[i], *(pf+i));
```

4.3.3 Tableaux dynamiques

Malgré la similitude, une grande différence reste cependant : un pointeur correspond à une adresse qui est variable, alors qu'un tableau est constant (aussi bien en positionnement au sein de la mémoire, qu'en termes de taille réservée).

L'utilisation d'un pointeur est donc la solution pour mettre en place des **tableaux dynamiques**, c'est-à-dire des tableaux dont la taille est variable et n'est fixée que lors de l'exécution du programme.

Alors que la taille d'un tableau est fixée à la déclaration (`type tableau[taille]`) et ne peut être basée sur une variable [3], la grande souplesse d'utilisation des pointeurs permet de déterminer l'espace mémoire à réserver en fonction d'une variable (`malloc(taille)`).

Ex. : Un pointeur permettant de mettre en œuvre un tableau dynamique.

```
float *pf;   // déclaration d'un pointeur sur réel
int nbv, i;

printf("nombre de valeurs ");
scanf("%d", &nbv);

pf = (float*) malloc(nbv*sizeof(float));  // réservation espace mémoire
                                          // pour nbv réels

for (i=0 ; i<nbv ; i++) {
   printf("valeur no %d ", i);
   scanf("%f", pf+i);
}

free(pf);
```

4.3.4 Tableaux dynamiques à plusieurs dimensions

Un pointeur pouvant se substituer à un tableau, il est tout à fait possible de gérer des **tableaux dynamiques multidimensionnels**.

[1] Pour un tableau, comme pour un pointeur, son adresse (l'adresse de début) correspond à l'adresse du premier élément.
[2] Noter la similitude de syntaxe.
[3] Au « mieux », on peut utiliser une constante pour définir la taille d'un tableau, cf 2.5.2.1.

4.3.4.1 Tableaux dynamiques à 2 dimensions

Pour définir un tableau dynamique à 2 dimensions, il suffit de déclarer un pointeur de pointeurs [1], suivant la syntaxe `type_pointeur **nom_pointeur` ou `type_pointeur** nom_pointeur`, le type du pointeur étant alors `type_pointeur**` [2].

Ex. : Un tableau dynamique à 2 dimensions.

```
float **mat; // déclaration d'un pointeur de pointeurs sur réels
int nbl, nbc, i, j;

printf("nombre de colonnes ");
scanf("%d", &nbc);
printf("nombre de lignes ");
scanf("%d", &nbl);

mat = (float**) malloc(nbl*sizeof(float*));     // allocation pour nbl lignes
                                                // de pointeurs
for (i=0 ; i<nbl ; i++)
   *(mat+i) = (float*) malloc(nbc*sizeof(float)); // allocation pour nbc réels
                                                  // par ligne
for (i=0 ; i<nbl ; i++)
   for (j=0 ; j<nbc ; j++) {
      printf("valeur [%d][%d] ", i, j);
      scanf("%f", *(mat+i)+j);
   }

for (i=0 ; i<nbl ; i++) {
   for (j=0 ; j<nbc ; j++) {
      printf("valeur [%d][%d] : %f\n", i, j, *(*(mat+i)+j));       // mat[i][j]
   }
}

for (i=0 ; i<nbl ; i++) free(*(mat+i));   // libération des espaces mémoire
free(mat);
```

4.3.4.2 Tableaux dynamiques de chaînes de caractères

On peut aussi utiliser les pointeurs pour créer des tableaux dynamiques de chaînes de caractères de taille dynamique.

Ex. : Un tableau dynamique de chaînes de caractères.

```
char **chaines; // déclaration d'un pointeur de pointeurs sur caractères
int nbc, lmc, i;

printf("nombre de chaines ");
scanf("%d", &nbc);
printf("longueur max des chaines ");
scanf("%d", &lmc);

chaines = (char**) malloc(nbc*sizeof(char*));     // allocation pour nbc chaines

for (i=0 ; i<nbc ; i++)
   *(chaines+i) = (char*) malloc(lmc*sizeof(char));  // allocation pour
                                                     // lmc caractères
for (i=0 ; i<nbc ; i++) {
   printf("chaine [%d] ", i);
   gets(*(chaines+i));
}

for (i=0 ; i<nbc ; i++) printf("%s\n", *(chaines+i));

for (i=0 ; i<nbc ; i++) free(*(chaines+i));
free(chaines);
```

[1] Tout comme un tableau à 2 dimensions n'est rien d'autre qu'un tableau de tableaux à 1 dimension.

[2] `type**` ≠ `type*` et `type**` ≠ `type`.

5 TYPES DE VARIABLES COMPLEXES

À l'opposé des types de variables simples (entier, réel, caractère) et de leurs déclinaisons, existent des types de variables complexes, définis par le programmeur.

5.1 LES STRUCTURES

5.1.1 Définition

Une **structure** permet de regrouper des variables de différents types en un seul ensemble. Chaque variable au sein de cette structure est appelée *champ*, et est accessible indépendamment des autres.

Ex. : Une structure, constituée de 2 champs, représentant une personne, nommée `personne`.

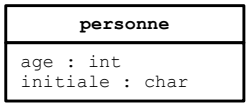

Figure 5.1 : exemple de structure

Lorsque l'on fait référence à l'un des champs d'une structure, il faut obligatoirement indiquer le nom de la structure à laquelle appartient le champ. Pour cela, on utilise l'opérateur '`.`' (point) qui permet de « naviguer » à travers la structure [1], suivant la syntaxe `variable_structurée.champ`.

Ex. : Soit `eleve` une variable structurée de type `personne` ; pour accéder à chacun des champs `age` et `initiale`, on écrit respectivement `eleve.age` et `eleve.initiale`.

5.1.2 Principes de mise en œuvre

5.1.2.1 Définition

Pour réaliser la **définition** d'une structure, on utilise le mot-clef `struct` suivant la syntaxe :

```
struct nom_structure {
   type_champ1 nom_champ1;
   type_champ2 nom_champ2;
   ...
};
```

Ex. : Une structure représentant une personne.

```
struct personne {
   int age;          // un champ 'age' du type entier
   char initiale;    // un champ 'initiale' du type caractère
};
```

Nb : La définition d'une structure est valide pour le bloc de code dans lequel elle est insérée ; un type structuré devant être utilisé dans l'ensemble du code source devra donc être défini à la suite des directives préprocesseur [2].

[1] Ce même opérateur est utilisé dans les langages objet pour marquer la relation *objet*.
[2] Telle une variable globale.

5.1.2.2 Déclaration

Une fois la structure définie, on peut utiliser directement son nom pour réaliser la **déclaration** d'une variable du type structuré, suivant la syntaxe `struct nom_structure nom_variable_structure`.

Ex. : Une variable du type structuré `personne`.

```
struct personne eleve1;
```

5.1.2.3 Affectation

L'**affectation d'un champ** d'une variable structure s'opère de la même manière que pour une variable simple, en accédant à chaque champ indépendamment.

Ex. :
```
eleve1.age = 20;          // affectation de 20 au champ age
eleve1.initiale = 'm';    // affectation de 'm' au champ initiale
```

L'**affectation d'une variable structure** en tant qu'entité à part entière ne peut être réalisée qu'en utilisant une autre variable structure de même type, déjà initialisée.

Ex. :
```
struct personne eleve2;   // déclaration d'une 2nde variable structure
eleve2 = eleve1;          // affectation globale
```

5.1.2.4 Utilisation

L'**utilisation d'un champ** d'une variable structure s'opère de la même manière que pour une variable simple.

Ex. :
```
int diffage;
struct personne eleve3;
eleve3.age = 18;
diffage = eleve1.age - eleve3.age;     // utilisation de champs
printf("%c - %d\n", eleve1.initiale, eleve1.age); // affichage des champs
```

L'**utilisation d'une variable structure** en tant qu'entité se réalise là aussi tout comme une variable simple.

Ex. :
```
void afficherPersonne(struct personne p)
{
   printf("age: %d\n", p.age);
   printf("initiale: %c\n", p.initiale);
}

int main(void)
{
   ...
   struct personne eleve4;
   ...
   eleve4 = eleve1;          // copie d'une variable structurée dans une autre
   afficherPersonne(eleve4);    // affichage des champs
}
```

Cependant, seules les opérations d'affectation ou de passage de paramètres ont un sens ; en effet, le nom de la variable structure est une référence à cette structure, et ne correspond à rien d'exploitable. De fait, une structure peut être le type de retour d'une fonction.

Ex. :
```
struct personne saisirPersonne()
{
   struct personne p;
   scanf("%d", &p.age);
   scanf("%c", &p.initiale);
   return p;
}

int main(void)
{
   struct personne eleve5;
   eleve5 = saisirPersonne();      // saisie des champs
   afficherPersonne(eleve5);       // affichage des champs
}
```

Une variable structure est passée par valeur lors de l'appel d'une fonction ; en conséquence, la modification dans une fonction d'un champ d'une variable structure passée en paramètre ne modifie pas le même champ de la variable structure dans le programme appelant.

Ceci peut cependant être réalisé, si on déclare un pointeur de structure plutôt qu'une simple variable structure. L'écriture `*pointeur_structure.champ` étant sujette à confusion [1], pour accéder au membre d'une variable structure pointée on utilise l'opérateur '->' (tiret + supérieur) suivant la syntaxe `pointeur_structure->champ`.

Ex. :
```
void modifierPersonne(struct personne *p)
{
   (p->age)++;
}

int main(void)
{
   ...
   modifierPersonne(&eleve5);      // modification d'un champ (on passe l'adresse)
   afficherPersonne(eleve5);          // affichage des champs
}
```

5.1.2.5 Initialisation

L'**initialisation** d'une variable structure consiste à initialiser chacun des champs qu'elle contient, dans l'ordre où ils sont donnés dans la définition de la structure, suivant la syntaxe :

```
struct nom_structure nom_variable_structure = {val_champ1, val_champ2, ...}
```

Ex. :
```
struct personne eleve6 = {21, 'n'};     //initialisation d'une variable structure
```

5.2 LES ÉNUMÉRATIONS

5.2.1 Définition

Une **énumération** permet de définir une liste de constantes symboliques de valeurs entières et consécutives commençant par défaut à la valeur 0.

5.2.2 Principes de mise en œuvre

5.2.2.1 Définition

Pour réaliser la **définition** d'une énumération, on utilise le mot-clef `enum` suivant la syntaxe :

```
enum nom_enumeration {nom_constante1, nom_ constante2, ...};
```

Ex. : Une énumération représentant les jours de la semaine.

```
enum semaine {lun, mar, mer, jeu, ven, sam, dim};
```

5.2.2.2 Déclaration

Une fois l'énumération définie, on peut utiliser directement son nom pour réaliser la **déclaration** d'une variable du type énuméré, suivant la syntaxe `enum nom_enumeration nom_variable_enumeration`.

Ex. : Une variable du type énuméré `semaine`.

```
enum semaine jour;
```

5.2.2.3 Affectation

L'**affectation d'une variable énumération** s'opère de la même manière que pour une variable simple, en utilisant l'une des valeurs de la liste de l'énumération ou bien une autre variable énumération de même type, déjà initialisée.

Ex. :
```
enum semaine jourRepos;          // déclaration d'une seconde variable énumérée
jour = mer;
jourRepos = jour;
```

[1] Est-ce la variable structure ou le champ qui est du type pointeur ?

Nb : Une énumération demeure une liste d'entiers, et on peut donc manipuler une variable énumérée comme tel.

Ex. :
```
jourRepos = 2;            // correspond à 'mer'
jour++;
```

5.2.2.4 Utilisation

L'**utilisation d'une constante** d'une variable énumération s'opère comme pour une variable simple.

Ex. :
```
int diffjour;
diffjour = dim - jour;       // utilisation directe d'une constante énumérée
printf("%d\n", diffjour);    // affichage: 4
```

5.2.2.5 Initialisation

L'**initialisation** d'une variable énumération est par défaut définie automatiquement en commençant par 0 et en affectant à chaque champ de la liste la valeur entière qui suit la valeur du champ précédent.

On peut cependant modifier la liste de valeurs en associant explicitement une valeur à un ou plusieurs champs, en utilisant la syntaxe `enum nom_enumeration {nom_champ1, ..., nom_champX = val_champX, ...}`.

Ex. : `enum jours {lun=1, mar, jeu=4, ven, sam};     // mar vaut 2, ven vaut 5, sam 6`

5.3 LES UNIONS

5.3.1 Définition

Une **union** permet de manipuler alternativement des variables de différents types et de différentes tailles stockées dans le même espace mémoire. Il s'agit en fait d'un type particulier de structure pour laquelle un seul et même espace mémoire est utilisé pour l'ensemble des champs ; sa taille correspond à la taille nécessaire pour stocker le plus grand des champs.

5.3.2 Principes de mise en œuvre

5.3.2.1 Définition

Pour réaliser la **définition** d'une union, on utilise le mot-clef `union` suivant la syntaxe :
```
union nom_union {
   type_champ1 nom_champ1;
   type_champ2 nom_champ2;
   ...
};
```

Ex. : Une union, définie par 2 champs, représentant un nombre.

```
union nombre {
   int i;       // un champ de type entier
   char c;      // un champ de type caractère
};
```

5.3.2.2 Déclaration

Une fois l'union définie, on peut utiliser directement son nom pour réaliser la **déclaration** d'une variable du type uni, suivant la syntaxe `union nom_union nom_variable_union`.

Ex. : Une variable du type uni `nombre`.

```
union nombre nb1;
```

5.3.2.3 Affectation

L'**affectation d'un champ** d'une variable union s'opère comme pour une structure, en accédant directement au champ dont on veut modifier la valeur.

Ex. : `nb1.i = 0x1234;     // affectation de 0x1234 au champ i`

L'**affectation d'une variable union** en tant qu'entité à part entière ne peut être réalisée qu'en utilisant une autre variable union de même type, déjà initialisée.

Ex. :
```
union nombre nb2;          // déclaration d'une 2nde variable union
nb2 = nb1;              // affectation globale
```

5.3.2.4 Utilisation

L'**utilisation d'un champ** d'une variable union s'opère comme pour une structure.

Ex. :
```
printf("%x\n", nb1.i);           // affichage de 0x1234
printf("%x\n", nb1.c);          // affichage de 0x34
```

L'**utilisation d'une variable union** en tant qu'entité se réalise exactement comme pour une structure.

5.3.2.5 Initialisation

L'**initialisation** d'une variable union consiste à initialiser sa valeur, suivant la syntaxe `union nom_union nom_variable_union = {valeur}`.

Ex. :
```
union nombre nb3 = {0x5678};     //initialisation d'une variable union
```

5.4 LES CHAMPS DE BITS

5.4.1 Définition

Un **champ de bits** est utilisé en conjonction avec un type de structure (ou d'union) constitué uniquement de champs de type entier dont on précise, pour chacun d'entre eux, le nombre de bits alloués dans la structure.

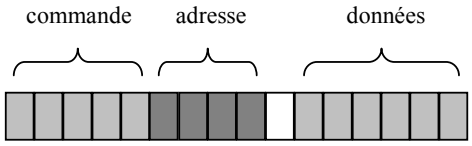

Figure 5.2 : exemple d'une structure constituée de champs de bits

Chaque champ de bits étant nécessairement de type entier, cela n'est pas obligatoire de le préciser. Il est possible aussi de laisser 1 ou plusieurs bits inutilisés.
La taille maximale de la structure ou de l'union ainsi utilisée est de 32 bits.

5.4.2 Principes de mise en œuvre

5.4.2.1 Définition

Pour réaliser la définition d'un champ de bits, il faut déclarer soit une structure soit une union, et préciser, lors de la déclaration du champ, le nombre de bits alloués, suivant la syntaxe `int nom_champ :nombre_bits_réservés`.

Ex. : Une structure contenant 3 champs de bits d'une longueur totale de 16 bits.

```
struct bufferIO {
   unsigned int commande :5;       // un champ de bits non-signé de 5 bits
   unsigned adresse :4;            // un champ de bits non-signé de 4 bits
   int :1                          // 1 bit inutilisé
   int donnees :6;                 // un champ de bits entier de 6 bits
};
```

5.4.2.2 Utilisation

L'**utilisation d'un champ de bits** d'une variable structure / union s'opère comme un champ « classique ».

Ex. : Initialisation d'un des champs de bits.

```
struct bufferIO registre;
registre.commande = 3;
registre.adresse = 20;          // 20 > (2^4 -1) = 15 !!
printf("commande: %d\n", registre.commande);   // affichage de 3
printf("adresse: %d\n", registre.adresse);        // affichage de 4 (20-16)
```

5.5 LES TYPES SYNONYMES

Sur la base d'un type existant, il est possible de créer un **type synonyme**, c'est-à-dire qui possède exactement les mêmes caractéristiques que le type que l'on désire copier, mais dont le nom est différent.

On utilise pour cela le mot-clef `typedef` suivant la syntaxe `typedef type_à_copier nouveau_type`.

Ex. :
```
typedef unsigned int compteur;   //définition d'un nouveau type entier non-signé
compteur cpt = 0;
printf("%d\n", ++cpt);

typedef float coordonnees[2];   // définition d'un nouveau type tableau de 2 réels
coordonnees pointA = {1, 1}, pointB;
pointB[0] = 2;
pointB[1] = 3;
printf("%f - %f\n", pointB[0], pointB[1]);
```

Les types synonymes peuvent être notamment utilisés avec les structures, et permettent de simplifier ainsi la déclaration de variables structure.

Ex. :
```
struct personne {   // définition d'une structure
   int age;
   char initiale;
};
typedef struct personne tPersonne;   // définition du nouveau type tPersonne
// on peut aussi écrire : typedef struct personne personne;

int main(void)
{
   tPersonne eleve7;    // tPersonne et struct personne sont utilisables
}
```

On peut appliquer directement la définition du nouveau type.

Ex. :
```
typedef struct {
   int age;
   char initiale;
} personne;     // définition d'un nouveau type à partir d'une structure anonyme

int main(void)
{
   personne eleve7;     // seul personne est un type utilisable
}
```

Nb : Il est possible de définir une structure ayant un champ du type de la structure elle-même.

Ex. :
```
typedef struct personne {
   int age;
   char initiale;
   struct personne parents[2];
} personne;
```

Un type de variable complexe constitue un nouveau type qui est utilisable exactement comme les types de variables simples connus, dans tous les cas où ceux-ci peuvent être mis en œuvre : déclaration de variable, de fonction, passage de paramètre, valeur renvoyée par une fonction au programme appelant, …

6 FLUX D'ENTRÉES/SORTIES SUR FICHIERS

La possibilité pour le langage C de manipuler des flux d'entrées/sorties sur fichiers permet de sauvegarder des données et de lire des données. Cela constitue ainsi un moyen pour un programme de communiquer avec le système d'exploitation sur lequel il est exécuté.

6.1 INTRODUCTION

Un **flux sur fichier** correspond à l'action de lire ou d'écrire un fichier sur une mémoire de masse (disque dur, disquette, cédérom, etc.).

On parle de *flux d'entrée*, lorsque les données « entrent » dans le programme, c'est-à-dire lorsque le programme lit un fichier ; on parle de *flux de sortie* lorsque les données « sortent » du programme, c'est-à-dire lorsque le programme écrit un fichier.

Accédé via un programme, un fichier est vu alors comme une suite d'octets, le dernier étant EOF [1].

On peut distinguer deux types de fichiers :

- fichier à accès séquentiel : les octets du fichier sont d'un seul bloc (pas d'octet vide ou appartenant à un autre fichier), on accède à n'importe quel octet du fichier en se basant sur le tout premier, on ne peut pas effacer un octet dans ce fichier, on peut en revanche tronquer la fin du fichier, ou bien rajouter un octet à la fin ;
- fichier à accès direct : les octets du fichiers peuvent être discontinus (octet vide possible), on peut accéder directement à n'importe quel octet du fichier, on peut modifier ou détruire n'importe quel octet.

Les fonctions standard du C permettant de manipuler et gérer des fichiers sont des fonctions à accès séquentiel.

6.2 GESTION DES FLUX

6.2.1 Déclaration

Pour gérer la suite d'octets que constitue un fichier, on les manipule en langage C à l'aide d'un pointeur ; on parle alors de *pointeur sur fichier*.

Pour pouvoir lire ou écrire un fichier, il faut donc déclarer un pointeur sur fichier avec le type déclaratif `FILE` [2], contenu dans la bibliothèque *stdio.h*, selon la syntaxe `FILE *nom_ptr_fichier`.

Ex. : `FILE *monfichier;  // déclaration d'un pointeur sur fichier`

6.2.2 Ouverture

L'**ouverture** du fichier, qui correspond à la création du flux, est nécessaire avant de pouvoir lire ou écrire un fichier.

Cette opération consiste à initialiser le pointeur sur le fichier désiré au sein du système d'exploitation, ainsi qu'à définir la manière dont on veut accéder à ce fichier (lecture / écriture / mise à jour / etc.).

Pour ouvrir un fichier et réserver l'espace mémoire nécessaire à son exploitation, on utilise la fonction `fopen()` [3] suivant la syntaxe `nom_ptr_fichier = fopen("nom_fichier", "mode_d_ouverture")`.

[1] EOF : End Of File (fin du fichier).
[2] Les majuscules sont importantes.
[3] Comportement identique à la fonction `malloc()`, chargée de réserver l'espace mémoire pour un pointeur (cf. 4.2.3.3).

Ex. : `monfichier = fopen("truc.txt", "r");  // ouverture en lecture seule`

Les différents modes d'ouverture sont :

symbole	accès	remarques
`r`	lecture seule	(1)
`w`	écriture seule	(1) (3) (4)
`a`	écriture	(2) (3) (5)
`r+`	lecture et écriture	(1)
`w+`	lecture et écriture	(1) (3) (4)
`a+`	lecture et écriture	(2) (3) (5)

(1) : pointeur positionné au début du fichier
(2) : pointeur positionné en fin de fichier
(3) : crée le fichier s'il n'existe pas
(4) : si le fichier existe, le contenu est écrasé
(5) : si le fichier existe, on écrit à la suite

6.2.3 Fermeture

La fermeture du fichier, nécessaire lorsque le fichier ne doit plus être utilisé, s'opère avec la fonction `fclose()` suivant la syntaxe `fclose(nom_ptr_fichier)`.

Ex. : `fclose(monfichier);`

Ainsi, l'espace mémoire précédemment réservé est libéré, et les données éventuellement stockées en mémoire tampon sont alors lues ou écrites physiquement.

6.3 LECTURE ET ÉCRITURE

6.3.1 Lecture dans un fichier

Pour effectuer une **lecture** dans un fichier, plusieurs fonctions, toutes incluses dans la bibliothèque *stdio.h*, peuvent être utilisées.
Le principe mis en œuvre dans ces diverses fonctions de lecture est toujours le même : à partir de la position courante du pointeur sur fichier, on lit un certain nombre d'octets, puis le pointeur est déplacé automatiquement du nombre d'octets qui a été lu ; celui-ci se trouve donc en position pour effectuer une nouvelle lecture/écriture.

Les différentes fonctions disponibles pour lire un fichier sont les suivantes :

- `getc()`, syntaxe `variable_entière = getc(nom_ptr_fichier)` : lit l'octet courant (qui peut être assimilé à un caractère pour les fichiers texte), et le retourne sous format entier ;
- `fgetc()` : identique à `getc()` ;
- `getw()`, syntaxe `variable_entière = getw(nom_ptr_fichier)` : lit l'entier courant, et le renvoie ;
- `fgets()`, syntaxe `fgets(chaine, nombre_caractères, nom_ptr_fichier)` : lit le nombre de caractères spécifié abaissé de 1 (caractère fin de chaîne), et les stocke dans une chaîne en ajoutant le caractère fin de chaîne final ;
- `fread()`, syntaxe `fread(ptr_dest, taille_bloc, nb_blocs, nom_ptr_fichier)` : lit `nb_blocs` de `taille_bloc` octets, et les stocke successivement dans le pointeur de destination ;
- `fscanf()`, syntaxe `fscanf(nom_ptr_fichier, format, &variable1, &variable2, ...)` : lit les différents formats successifs (qui peuvent se présenter sous la forme *texte+format*), et les stocke dans l'ordre dans les différentes variables spécifiées.

6.3.2 Écriture dans un fichier

Pour effectuer une **écriture** dans un fichier, plusieurs fonctions, toutes incluses dans la bibliothèque *stdio.h*, peuvent être utilisées.
Le principe mis en œuvre dans ces diverses fonctions d'écriture est le même que pour la lecture : à partir de la position courante du pointeur sur fichier, on écrit un certain nombre d'octets, puis le pointeur est déplacé automatiquement du nombre d'octets qui a été écrit ; celui-ci se trouve donc en position pour effectuer une nouvelle lecture/écriture.

Les différentes fonctions disponibles pour lire un fichier sont les suivantes :

- `putc()`, syntaxe `putc(caractère, nom_ptr_fichier)` : écrit un caractère ;
- `fputc` : identique à `putc()` ;
- `putw()`, syntaxe `putw(entier, nom_ptr_fichier)` : écrit un entier ;
- `fputs()`, syntaxe `fputs(chaine, nom_ptr_fichier)` : écrit une chaîne (le caractère fin de chaîne n'est pas écrit) ;
- `fwrite()`, syntaxe `fwrite(ptr_src, taille_bloc, nb_blocs, nom_ptr_fichier)` : écrit `nb_blocs` de `taille_bloc` octets lus à partir du pointeur source ;
- `fprintf()`, syntaxe `fprintf(nom_ptr_fichier, format, variable1, variable2, ...)` : écrit successivement les variables spécifiées selon le format (qui peut se présenté sous la forme *texte+format*).

6.3.3 Autres fonctions de gestion

D'autres fonctions permettant la gestion des flux peuvent s'avérer utiles :

- `fflush()`, syntaxe `fflush(nom_ptr_fichier)` : vide le buffer d'entrées/sorties en écrivant « physiquement » son contenu dans le fichier (flux de sortie), ou vide simplement le buffer (flux d'entrée) ;
- `fseek()`, syntaxe `fseek(nom_ptr_fichier, nb_octets, référence)` : déplace le pointeur du fichier du nombre d'octets indiqués à partir de la référence (0 : début du fichier, 1 : position courante, 2 : fin du fichier (retour en arrière)) ;
- `rewind()`, syntaxe `rewind(nom_ptr_fichier)` : positionne le pointeur du fichier au début de celui-ci ;
- `fgetpos()`, syntaxe `fgetpos(nom_ptr_fichier, ptr_backup)` : sauvegarde la position du pointeur courant (s'utilise exclusivement avec `fsetpos()`) ;
- `fsetpos()`, syntaxe `fsetpos(nom_ptr_fichier, ptr_backup)` : positionne le pointeur courant à la position sauvegardée(s'utilise exclusivement avec `fgetpos()`) ;
- `feof()`, syntaxe `feof(nom_ptr_fichier)` : renvoie une valeur non nulle si une lecture a été tentée au-delà de la fin du fichier (actif uniquement sur un flux d'entrée) ;
- `ferror()`, syntaxe `ferror(nom_ptr_fichier)` : appelé immédiatement après une opération d'entrées/sorties sur fichier, renvoie une valeur non nulle si une erreur s'est produite durant cette opération.

6.3.4 Gestion des erreurs

Les différentes fonctions de lecture/écriture décrites ci-dessus renvoient des valeurs permettant de déterminer si l'opération s'est bien déroulée ou bien si une erreur s'est produite :

- `fopen()` : renvoie le pointeur `NULL` en cas d'erreur ;
- `getc()` / `fgetc()` : renvoie `EOF` si on a atteint la fin du fichier ou si une erreur est survenue ;
- `fgets()` : renvoie le pointeur `NULL` en cas d'erreur ou si on a atteint la fin du fichier ;
- `fread()` : renvoie le nombre de blocs effectivement lus, ou `0` si on a atteint la fin du fichier avant la fin ;
- `fscanf()` : renvoie le nombre de variables effectivement écrites, ou `EOF` si on a atteint la fin du fichier ;
- `putc()` / `fputc()` : renvoie `EOF` en cas d'erreur ;
- `fputs()` : renvoie `EOF` en cas d'erreur ;
- `fwrite()` : renvoie le nombre de blocs effectivement écrits ;
- `fprintf()` : renvoie le nombre de variables effectivement écrits, ou un nombre négatif en cas d'erreur ;
- `fflush()` : renvoie `0` si le vidage s'est effectué correctement, ou `EOF` en cas d'erreur ;
- `fseek()` : renvoie `0` si le déplacement s'est effectué correctement, ou une valeur non nulle en cas d'erreur.

Les flux d'entrées/sorties peuvent être opérés sur d'autres éléments que des fichiers. Notamment, l'interfaçage de l'application avec le système d'exploitation utilise des flux d'entrées/sorties appelés **flux d'entrées/sorties standard** :

- entrée standard : par défaut le clavier, appelé *stdin* ;
- sortie standard : par défaut la console (l'écran), appelé *stdout*.

On peut ainsi noter que la fonction `fprintf()`, appliquée au flux de sortie standard (stdout), équivaut à la fonction `printf()` : `fprintf(stdout, format, variable1, variable2, ...)` ; on écrit les variables dans le flux de sortie, c'est-à-dire sur l'écran.

De la même manière, la fonction `fscanf()`, appliquée au flux d'entrée standard (stdin), équivaut à la fonction `scanf()` : `fscanf(stdin, format, &variable1, &variable2, ...)` ; on lit les variables à partir du flux d'entrée, c'est-à-dire le clavier.

A MOTS-CLEFS

A.1 ORDRE ALPHABÉTIQUE

A	`auto`					
B	`break`					
C	`case`	`char`	`const`	`continue`		
D	`default`	`do`	`double`			
E	`else`	`enum`	`extern`			
F	`FILE`	`float`	`for`			
G	`goto`					
I	`if`	`int`				
L	`long`					
N	`NULL`					
R	`register`	`return`				
S	`short`	`signed`	`sizeof`	`static`	`struct`	`switch`
T	`typedef`					
U	`union`	`unsigned`				
V	`void`	`volatile`				
W	`while`					

Les mots-clefs sont réservés, donc inutilisables comme noms de variable ou de fonction [1].

A.2 CATÉGORIES

A.2.1 Types de variables

`char`	Caractère sur 1 octet (standard ASCII étendu).
`double`	Réel sur 8 octets.
`FILE`	Pointeur sur fichier.
`float`	Réel sur 4 octets.
`int`	Entier sur 4 octets.
`short`	Entier sur 2 octets.
`void`	Type « vide » (utilisé pour les déclarations de fonctions uniquement).

A.2.2 Modificateurs de variables

`auto`	Variable locale, qui n'existe que durant l'exécution du bloc de code dans lequel elle est déclarée – *par défaut*.
`const`	Variable de valeur constante qui ne peut pas être modifiée lors de l'exécution.
`extern`	Variable déclarée ou susceptible d'être utilisée dans un autre module ou une autre bibliothèque ; sa durée de vie est celle d'une variable statique (cf. `static`).
`long`	Variable dont la taille est doublée (Nb : effet non garanti, ex. : sur les PCs, un `long int` a la même taille qu'un `int`, idem pour `long double`).

[1] Même si le langage C distingue la casse, il est déconseillé d'utiliser un nom, même avec une casse distincte, proche d'un mot-clef existant.

`register`	Variable stockée dans l'un des registres du processeur, et dont les accès sont ainsi optimisés (Nb : amélioration des performances de l'application non garantie).
`signed`	Type numérique signé, utilisant le bit de poids fort en guise de signe (0 : positif, 1 : négatif) ; échelle de valeurs centrée autour de 0 (types numériques uniquement, cf. `unsigned`) – *par défaut*.
`static`	Variable locale dont la valeur est conservée entre deux appels de la fonction (cf. `auto`).
`unsigned`	Type numérique non signé, utilisant tous les bits pour la valeur ; début de l'échelle de valeurs à partir de 0 (types numériques uniquement, cf. `signed`).
`volatile`	Variable sauvegardée uniquement en mémoire vive, et qui n'est jamais bufferisée ; sa valeur instantanée est ainsi assurée lors de l'accès par diverses fonctions, lorsque celles-ci sont susceptibles d'en modifier la valeur et que le programme ne peut le prévoir à la compilation.

A.2.3 Types de variables complexes

`enum`	Définition d'un type énuméré – liste de mnémoniques alphanumériques de valeur entière et successive (0, 1, 2, …).
`struct`	Définition d'un type structuré – regroupement de données de types différents (appelés *champs*) dans un seul ensemble.
`typedef`	Création d'un nouveau type sur la base d'une variable utilisateur (variable simple, tableau, pointeur, structure, etc.).
`union`	Définition d'un type structuré commun – mise en commun d'un espace mémoire accessible par des données de types différents (appelés *champs*).

A.2.4 Structures de contrôle

`break`	Sortie de boucle ou de bloc de code.
`case`	Test multiple – sélection (cf. `default`, `switch`).
`continue`	Poursuite de l'exécution de la boucle (branchement à l'itération suivante (test de continuité)).
`default`	Test multiple – sélection par défaut (cf. `case`, `switch`).
`do`	Boucle avec une itération minimale (cf. `while`).
`else`	Test simple – test conditionnel inverse (cf. `if`).
`for`	Boucle avec itération.
`if`	Test simple – test conditionnel (cf. `else`).
`return`	Sortie du bloc de code avec ou sans valeur de retour.
`switch`	Test multiple – test de sélection (cf. `case`, `default`).
`while`	Boucle avec une itération minimale (cf. `do`) ou aucune.

A.2.5 Autres

`goto`	Branchement à une étiquette (désuet et déconseillé).
`NULL`	Pointeur nul (pas d'espace mémoire réservé).
`sizeof`	Renvoi de la taille en octets occupée par un type donné, ou par une variable (i.e. le type auquel elle appartient).

B TYPES ET FORMATS DE VARIABLES

B.1 TYPES DE VARIABLES

nom	mot-clef	taille	valeurs extrêmes
entier (court)	`short`	2 octets	$[-2^{15} ; +2^{15}-1]$
entier non-signé (court)	`unsigned short`	2 octets	$[0 ; +2^{16}-1]$
entier	`int`	4 octets	$[-2^{31} ; +2^{31}-1]$
entier non-signé	`unsigned int`	4 octets	$[0 ; +2^{32}-1]$
nombre réel (flottant)	`float`	4 octets	$[\approx \pm 10^{-37} ; \approx \pm 10^{+38}]$ [1]
nombre réel double précision	`double`	8 octets	$[\approx \pm 10^{-307} ; \approx \pm 10^{+308}]$ [2]
caractère	`char`	1 octet	$[-2^{7} ; +2^{7}-1]$ ou ASCII
caractère non-signé	`unsigned char`	1 octet	$[0 ; +2^{8}-1]$ ou ASCII

B.2 FORMATS DE VARIABLES

- `d` : décimal ;
- `ld` : décimal long ;
- `x` : hexadécimal ;
- `o` : octal ;
- `u` : entier non signé ;
- `lu` : entier non signé long ;
- `f` : réel ;
- `lf` : double ;
- `e` : réel en notation scientifique (exposant) ;
- `c` : caractère ;
- `s` : chaîne de caractères ;
- `p` : pointeur.

---

[1] Pour un nombre réel noté ±x,y (x : partie entière, y : partie décimale) de 32 bits, celui-ci est ramené à une notation binaire en notation scientifique (exposant) sous la forme ±1,zE±t et on le décompose ainsi : signe 1 bit, mantisse 23 bits (z), exposant 8 bits (t) ; avec les valeurs d'exposants extrêmes réservées pour symboliser le nombre 0 (exposant = 0) et l'infini (exposant = 255), donc sur la plage [–126 ; +127], on trouve : $\pm 1x2^{-126} \approx \pm 1,17.10^{-38}$ (min) et $\pm(2x2^{+127}-1) \approx \pm 3,40.10^{+38}$ (max).

[2] 64 bits : signe 1 bit, mantisse 52 bits, exposant 11 bits ; avec les valeurs d'exposants extrêmes réservées pour symboliser 0 (exposant = 0) et l'infini (exposant = 2047), donc sur la plage [–1022 ; +1023], on trouve : $\pm 1x2^{-1022} \approx \pm 2,23.10^{-308}$ (min) et $\pm(2x2^{+1023}-1) \approx \pm 1,80.10^{+308}$ (max).

C DÉLIMITEURS ET OPÉRATEURS

C.1 DÉLIMITEURS

`;`	Marque la fin d'une ligne d'instruction ou d'une déclaration (typiquement, de toute ligne incluse dans un bloc de code).
`,`	Sépare deux éléments d'une liste.
`" "`	Délimite une chaîne de caractères.
`' '`	Encadre un caractère.
`( )`	Encadre une liste de paramètres.
`{ }`	Délimite un bloc d'instructions, ou une liste de valeurs d'initialisation.
`[ ]`	Encadre la taille ou l'indice d'un tableau.

C.2 OPÉRATEURS

On distingue trois types d'opérateurs :

- unaire : 1 argument ($_1$), syntaxe `opérateur opérande` ;
- binaire : 2 arguments ($_2$), syntaxe `opérande1 opérateur opérande2` ;
- ternaire : 3 arguments ($_3$), syntaxe `opérande1 opérateurA opérande2 opérateurB opérande3`.

C.2.1 Opérateurs d'affectation

`=`	Affectation. ($_2$)
`(op)=`	Opérateurs combinés – opération puis affectation (cf. 0). ($_2$)

C.2.2 Opérateurs arithmétiques

`+`	Addition. ($_2$)
`-`	Soustraction. ($_2$)
`*`	Multiplication. ($_2$)
`/`	Division. ($_2$)
`%`	Modulo – reste de la division entière (division euclidienne) entre deux entiers. ($_2$)
`-`	Opposé (positif/négatif). ($_1$)

C.2.3 Opérateurs binaires

`&`	ET logique (AND) – vrai si les deux sont vrais / faux si l'un des deux est faux. ($_2$)
`|`	OU logique (OR) – vrai si au moins l'un des deux est vrai / faux si les deux sont faux. ($_2$)
`!`	NON logique (NOT) – vrai si faux / faux si vrai. ($_1$)
`^`	OU logique exclusif (XOR) – vrai si l'un ou l'autre est vrai mais pas les deux / faux si les deux sont vrais ou si les deux sont faux. ($_2$)
`~`	Complément à un (inversion de tous les bits). ($_1$)
`<<`	Décalage à gauche (multiplication par 2) du nombre de bits spécifiés. ($_2$)
`>>`	Décalage à droite (division par 2) du nombre de bits spécifiés. ($_2$)

C.2.4 Opérateurs combinés

`++`	Incrémentation (pré- et post-). (1)
`--`	Décrémentation (pré- et post-). (1)
`+=`	Addition et affectation. (2)
`-=`	Soustraction et affectation. (2)
`*=`	Multiplication et affectation. (2)
`/=`	Division et affectation. (2)
`%=`	Modulo et affectation. (2)
`<<=`	Décalage à gauche et affectation. (2)
`>>=`	Décalage à droite et affectation. (2)
`&=`	ET logique bit-à-bit et affectation. (2)
`|=`	OU logique bit-à-bit et affectation. (2)
`^=`	OU logique exclusif bit-à-bit et affectation. (2)

C.2.5 Opérateurs d'évaluation d'expression

`==`	Égalité. (2)
`!=`	Différence. (2)
`<`	Infériorité stricte. (2)
`<=`	Infériorité ou égalité. (2)
`>`	Supériorité stricte. (2)
`>=`	Supériorité ou égalité. (2)
`&&`	ET logique (AND). (2)
`||`	OU logique (OR). (2)
`!`	NON logique (NOT). (1)

C.2.6 Opérateurs divers

`(type)`	Conversion de type (transtypage, cast). (1)
`*`	Opérateur d'indirection – accès au contenu. (1)
`&`	Opérateur d'adresse – accès au contenant. (1)
`.`	Accès à un champ d'une structure ou d'une union. (2)
`->`	Accès à un champ d'une structure ou d'une union pointée. (2)
`? :`	Opérateur ternaire (exécution conditionnelle simplifiée). (3)

C.3 PRIORITÉ DES OPÉRATEURS ET DÉLIMITEURS

Le niveau de priorité des opérateurs et délimiteurs est donné du plus prioritaire au moins prioritaire. Il n'y a pas de priorité particulière sur les opérateurs d'un même niveau ; en ce cas, il faut donc utiliser les parenthèses.

Légende : (1) : opérateur unaire, (2) : opérateur binaire, (3) : opérateur ternaire.

+ prioritaire →	`( )` `[ ]` `.` `->`
	`!` `~` `++` `--` `-` (1) `*` (1) `&` (1) `sizeof` `(type)`
	`*` (2) `/` `%`
	`+` `-` (2)
	`<<` `>>`
	`<` `<=` `>` `>=`
	`==` `!=`
	`&` (2)
	`^`
	`|`
	`&&`
	`||`
	`? :` (3)
← – prioritaire	`=` `*=` `/=` `%=` `+=` `-=` `<<=` `>>=` `&=` `^=` `|=`
	`,`

D RÉFÉRENCE DU LANGAGE

D.1 LA NORME C ANSI

La popularité grandissante et l'absence de standardisation du langage C sont à l'origine, durant les années 1970, d'une production, par diverses universités et organisations, de versions distinctes du C. Dans les années 1980, ces différentes implémentations du langage se sont révélées ne pas être en totalité compatibles entre elles.

En 1983, l'Institut National Américain des Standards (ANSI : American National Standard Institute) a formé un comité afin d'établir une spécification standard du langage C, en utilisant les différentes versions produites et les différents besoins ayant émergé depuis la naissance du langage (gestion des entrées/sorties par exemple). Le résultat de ces travaux a été rendu public en 1989, sous le nom de *norme C ANSI*, ou C89.

Une partie de cette spécification concerne les bibliothèques dites « standard » (comprendre *standard C ANSI*). Depuis 1989, un certain nombre de révisions sont venues étoffer la bibliothèque standard C ANSI, qui compte en tout [1] 24 bibliothèques différentes (accessibles grâce aux fichiers d'entête *.h*).

La dernière révision officielle du standard C est la version C99, datant de 1999.

D.2 BIBLIOTHÈQUES STANDARD

Le langage C, tel qu'il est défini, est extrêmement limité en « opération de base ». Ce sont les fonctions de bibliothèques qui apportent toute la richesse du langage [2].

`alloc.h`	Gestion de la mémoire.
`bios.h`	Routines BIOS.
`conio.h`	Flux d'entrées/sorties sur la console (utilisation de routines DOS).
`complex.h`	Manipulation des nombres complexes.
`ctype.h`	Manipulation de caractères.
`dir.h`	Gestion des répertoires.
`dos.h`	Appels DOS (80x86).
`errno.h`	Signification textuelle des codes d'erreurs renvoyés par les fonctions de bibliothèques.
`float.h`	Opérations en virgule flottantes.
`graphics.h`	Fonctions graphiques.
`io.h`	Entrées/sorties de bas niveau.
`locale.h`	Gestion des différentes conventions culturelles et géographiques : notation heure / date, monnaies, langues, etc.
`malloc.h`	Gestion de la mémoire.
`math.h`	Fonctions de calculs mathématiques.
`process.h`	Programmes multiples.
`stdio.h`	Flux d'entrées/sorties standard.
`stdlib.h`	Fonctions diverses d'usage courant.
`string.h`	Manipulation des chaînes de caractères.
`time.h`	Accès à l'horloge (date/heure) et conversions.

D.2.1 Fonctions d'entrées/sorties

Extrait de fonctions de la bibliothèque `stdio.h`.

[1] En janvier 2006.

[2] Cette remarque est aussi vraie pour d'autres langages, tels le C++ (la STL), le C# (le framework .Net), Java (API Java), etc..

`getchar()`	Saisie d'un caractère.
`gets()`	Saisie d'une chaîne de caractères.
`printf()`	Affichage formaté.
`putchar()`	Affichage d'un caractère.
`puts()`	Affichage d'une chaîne.
`scanf()`	Saisie formatée.

D.2.2 Fonctions de manipulations de caractères

Extrait de fonctions de la bibliothèque `ctype.h`.

`isalnum()`	Test alphanumérique (lettre ou chiffre).
`isalpha()`	Test lettre (minuscule ou majuscule).
`isascii()`	Test ASCII.
`iscntrl()`	Test caractère de contrôle.
`isdigit()`	Test chiffre décimal.
`islower()`	Test lettre minuscule.
`isprint()`	Test caractère imprimable.
`ispunct()`	Test caractère de ponctuation.
`isspace()`	Test caractère espace.
`isupper()`	Test lettre majuscule.
`isxdigit()`	Test chiffre hexadécimal.
`tiascii()`	Conversion numérique ASCII.
`toupper()`	Conversion en majuscule.
`tolower()`	Conversion en minuscule.

D.2.3 Fonctions de manipulations de chaînes de caractères

Extrait de fonctions de la bibliothèque `string.h`.

`sprintf()`	Copie formatée de chaîne de caractères.
`sscanf()`	Lecture formatée de chaîne de caractères.
`strcat()`	Concaténation de deux chaînes de caractères (cf. `strncat()`).
`strcmp()`	Comparaison lexicale de deux chaînes de caractères (cf. `strncmp()`).
`strcpy()`	Copie de chaîne de caractères (cf. `strncpy()`).
`strlen()`	Longueur d'une chaîne de caractères.
`strncat()`	Concaténation de caractères à une chaîne de caractères (cf. `strcat()`).
`strncmp()`	Comparaison lexicale des caractères de deux chaînes de caractères (cf. `strcmp()`).
`strncpy()`	Copie de caractères d'une chaîne de caractères (cf. `strcpy()`).

D.2.4 Fonctions de calculs mathématiques

Extrait de fonctions de la bibliothèque `math.h`.

`abs()`	Valeur absolue.
`acos()`	Arcosinus.
`asin()`	Arcsinus.
`atan()`	Arctangente.
`atan2()`	Arctangente X/Y.
`ceil()`	Arrondi par excès (cf. `floor()`).
`cos()`	Cosinus.
`cosh()`	Cosinus hyperbolique.
`exp()`	Exponentielle.
`fabs()`	Valeur absolue d'un nombre réel.
`floor()`	Arrondi par défaut (cf. `ceil()`).
`fmod()`	Calcul du reste réel d'une division de deux nombres réels avec quotient entier.
`frexp()`	Décomposition d'un nombre réel en mantisse et exposant selon la formule : réel = mantisse * $2^{exposant}$.
`ldexp()`	Recomposition d'un nombre réel à partir de la mantisse et de l'exposant.

`log()`	Logarithme népérien.
`log10()`	Logarithme décimal.
`modf()`	Décomposition d'un nombre réel en partie entière et en partie décimale.
`pow()`	Puissance.
`sin()`	Sinus.
`sinh()`	Sinus hyperbolique.
`sqrt()`	Racine carrée.
`tan()`	Tangente.
`tanh()`	Tangente hyperbolique.

D.2.5 Fonctions de gestion de fichiers

Extrait de fonctions de la bibliothèque `stdio.h`.

`fclose()`	Fermeture d'un fichier.
`feof()`	Test de fin de fichier.
`ferror()`	Test d'erreur.
`fgetc()`	Lecture d'un caractère (identique à `getc()`).
`fgets()`	Lecture d'une chaîne de caractères.
`fgetw()`	Lecture d'un entier (identique à `getw()`).
`fopen()`	Ouverture d'un fichier.
`fprintf()`	Écriture formatée.
`fputc()`	Écriture d'un caractère (identique à `putc()`).
`fputs()`	Écriture d'une chaîne de caractères.
`fputw()`	Écriture d'un entier (identique à `putw()`).
`fread()`	Lecture d'octets.
`fscanf()`	Lecture formatée.
`fseek()`	Déplacement dans un fichier.
`ftell()`	Renvoi de la position courante dans un fichier.
`fwrite()`	Écriture d'octets.
`getc()`	Lecture d'un caractère (identique à `fgetc()`).
`getw()`	Lecture d'un entier (identique à `fgetw()`).
`putc()`	Écriture d'un caractère (identique à `fputc()`).
`putw()`	Écriture d'un entier (identique à `fputw()`).
`rewind()`	Positionnement au début d'un fichier.

D.3 LE PRÉPROCESSEUR

Le **préprocesseur** est un programme spécifique fourni par le compilateur qui analyse le code source C et effectue des modifications sur son contenu avant qu'il soit compilé. Tous les ordres de modification du code, appelés *directives*, commencent par le symbole `#`.

`#`	Directive nulle.
`#define`	Définition d'une macro (constante ou fonction) (cf. `#undef`).
`#elif`	Compilation conditionnelle sur expression dans une compilation conditionnelle inversée – contraction de `#else` et de `#if` (cf. `#if`, `#else`, `#endif`, `#ifdef` et `#ifndef`).
`#else`	Compilation conditionnelle inversée (cf. `#if`, `#elif`, `#endif`, `#ifdef` et `#ifndef`).
`#endif`	Fin d'une compilation conditionnelle (cf. `#if`, `#elif`, `#else`, `#ifdef` et `#ifndef`).
`#error`	Émission d'un message d'erreur.
`#if`	Compilation conditionnelle sur expression (cf. `#elif`, `#else`, `#endif`, `#ifdef` et `#ifndef`).
`#ifdef`	Compilation conditionnelle sur existence de la définition d'une macro (cf. `#if`, `#elif`, `#else`, `#endif` et `#ifndef`).
`#ifndef`	Compilation conditionnelle sur absence de la définition d'une macro (cf. `#if`, `#elif`, `#else`, `#endif` et `#ifdef`).
`#include`	Inclusion du contenu d'un fichier (recopie) dans le fichier courant.
`#line`	Changement du numéro de la ligne courante.

`#pragma`	Non-prise en compte des directives précisées si le compilateur ne les comprend pas (plutôt que d'afficher des messages d'erreurs de compilation).
`#undef`	Annulation d'une définition de macro (cf. `#define`).
`defined()`	Opérateur de test de la définition d'une macro (s'utilise exclusivement avec `#if` et `#elif`).
`__DATE__`	Chaîne représentant la date courante (macro prédéfinie).
`__FILE__`	Nom du fichier source courant (macro prédéfinie).
`__LINE__`	Numéro de la ligne courante (macro prédéfinie).
`__STDC__`	Test des qualités ISO du compilateur (macro prédéfinie).
`__TIME__`	Chaîne représentant l'heure courante (macro prédéfinie).

Ex. : Avant l'analyse par le préprocesseur (code-source) :

```
#include <stdio.h>

#define PI 3.14
#define testrayon(r) (r>100 ? 0 : 1)

int main(void)
{
   float rayon;
   float circonference

   printf("entrer le rayon ");
   scanf("%f", &rayon);

   if (testrayon(rayon)) {
      circonference = 2 * PI * rayon;
      printf("circonference : %f\n", circonference);
   }
   else {
      printf("rayon trop grand\n");
   }

   return 0;
}
```

Après l'analyse par le préprocesseur (code réellement compilé) :

```
#include <stdio.h>

int main(void)
{
   float rayon;
   float circonference

   printf("entrer le rayon ");
   scanf("%f", &rayon);

   if (rayon>100 ? 0 : 1) { // macro remplacée par son code
      circonference = 2 * 3.14 * rayon; // constante remplacée par sa valeur
      printf("circonference : %f\n", circonference);
   }
   else {
      printf("rayon trop grand\n");
   }

   return 0;
}
```

L'usage du préprocesseur mérite énormément d'attention. Il constitue un outil puissant mais est une source d'erreurs difficiles à détecter, puisque les modifications sont réalisées avant la compilation et qu'elles ne subissent donc aucun contrôle.

E LE CODE ASCII

Le code ASCII, créé dans les années 60, permet de coder numériquement un caractère afin de pouvoir être manipulé par l'ordinateur. Il utilise dans sa version standard 7 bits, et permet donc de coder 128 caractères.

Les 32 premiers caractères sont des caractères de contrôle, donc non-affichables, qui servaient anciennement aux téléscripteurs comme codes de mise en page ; beaucoup sont maintenant désuets.

dec	hex	caractère	dec	hex	caractère	dec	hex	caractère	dec	hex	caractère
0	00	NUL (null)	32	20	<space>	64	40	@	96	60	`
1	01	SOH (start of heading)	33	21	!	65	41	A	97	61	a
2	02	STX (start of text)	34	22	"	66	42	B	98	62	b
3	03	ETX (end of text)	35	23	#	67	43	C	99	63	c
4	04	EOT (end of transmission)	36	24	$	68	44	D	100	64	d
5	05	ENQ (enquiry)	37	25	%	69	45	E	101	65	e
6	06	ACK (acknowledge)	38	26	&	70	46	F	102	66	f
7	07	BEL (bell), '\a'	39	27	'	71	47	G	103	67	g
8	08	BS (backspace), '\b'	40	28	(	72	48	H	104	68	h
9	09	TAB (horizontal tab), '\t'	41	29	)	73	49	I	105	69	i
10	0A	LF (line feed = new line), '\n'	42	2A	*	74	4A	J	106	6A	j
11	0B	VT (vertical tab)	43	2B	+	75	4B	K	107	6B	k
12	0C	FF (form feed), '\f'	44	2C	´	76	4C	L	108	6C	l
13	0D	CR (carriage return), '\r'	45	2D	–	77	4D	M	109	6D	m
14	0E	SO (shift out)	46	2E	.	78	4E	N	110	6E	n
15	0F	SI (shift in)	47	2F	/	79	4F	O	111	6F	o
16	10	DLE (data link escape)	48	30	0	80	50	P	112	70	p
17	11	DC1 (device control 1)	49	31	1	81	51	Q	113	71	q
18	12	DC2 (device control 2)	50	32	2	82	52	R	114	72	r
19	13	DC3 (device control 3)	51	33	3	83	53	S	115	73	s
20	14	DC4 (device control 4)	52	34	4	84	54	T	116	74	t
21	15	NAK (negativ aknowledge)	53	35	5	85	55	U	117	75	u
22	16	SYN (synchronous idle)	54	36	6	86	56	V	118	76	v
23	17	ETB (end of transmission bloc)	55	37	7	87	57	W	119	77	w
24	18	CAN (cancel)	56	38	8	88	58	X	120	78	x
25	19	EM (end of medium)	57	39	9	89	59	Y	121	79	y
26	1A	SUB (substitute)	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC (escape)	59	3B	;	91	5B	[	123	7B	{
28	1C	FS (file separator)	60	3C	<	92	5C	\	124	7C	|
29	1D	GS (group separator)	61	3D	=	93	5D	]	125	7D	}
30	1E	RS (record separator)	62	3E	>	94	5E	^	126	7E	~
31	1F	US (unit separator)	63	3F	?	95	5F	_	127	7F	DEL

Par la suite, on a créé le code ASCII étendu afin d'inclure les caractères accentués. On a pour cela utilisé un 8ème bit pour le codage, portant ainsi la table à 256 caractères différents [1].

Nb : Au début des années 90, le standard Unicode [2] a été créé. Reprenant les mêmes principes que le code ASCII, mais en utilisant 16 bits [3], ce code permet ainsi de pouvoir coder numériquement la quasi-totalité des alphabets existants, notamment les alphabets cyrillique, grec, arabe et hébreu. Il a été conçu avec une compatibilité totale avec le code ASCII existant, id est, les 256 premiers caractères Unicode sont les 256 caractères du code ASCII.

[1] L'usage des multiples de 8 bits s'étant démocratisé, l'appellation « code ASCII » fait aujourd'hui référence au code ASCII étendu à 8 bits.
[2] Listes des codes Unicode classées par thème disponible sur http://www.unicode.org/charts/.
[3] En janvier 2008, dans sa version 5.0.0, le standard Unicode utilise maintenant 20 bits.

F BIBLIOGRAPHIE

Maillefer Joëlle, *Cours/TP Langage C*, IUT Cachan, 1993 ;

Garreta Henri, *Le langage C*, Université de la Méditerranée – Faculté des sciences de Luminy, 2006 ;

Hérauville Stéphane, *Cours Langage C*, CFAI Marcel Sembat – Sotteville-lès-Rouen, 2002 ;

Infini-software.com, *Le langage C ANSI*,
http://www.infini-software.com/Encyclopedie/Developpement/C_Cpp/CAnsi/French/Index.wp, 2005 ;

Hardware.fr, *Forum C/C++*, http://forum.hardware.fr/hfr/Programmation/C++/liste_sujet-1.htm, 2006 ;

CommentÇaMarche.net, *Langage C*, http://www.commentcamarche.net/c/, 2006 ;

Letenneur Philippe, *Langage C* ;

Merouan Kaced sous tutorat de **Balmas Françoise**, *Little-endian et big-endian*,
http://www.ai.univ-paris8.fr/~fb/Cours/Exposes0405-1/little-big-endian.pdf,
Licence/maîtrise informatique – Labo LIASD, LEMA – Université Vincennes–Saint-Denis – Paris 8, 2005 ;

CommentÇaMarche.net, http://www.commentcamarche.net/, 2006 ;

Développez.com, *Langage C*, http://c.developpez.com/, 2006 ;

Cplusplus.com, *Library reference*, http://www.cplusplus.com/reference/, 2007 ;

Wikipedia – l'encyclopédie libre, *Bibliothèque standard du C* et *Préprocesseur*, http://fr.wikipedia.org/, 2006 ;

Labrousse Isabelle, *Langage C – Cross-compilateur C-68000, chaîne Whitesmith*, IUT Cachan, 1992.