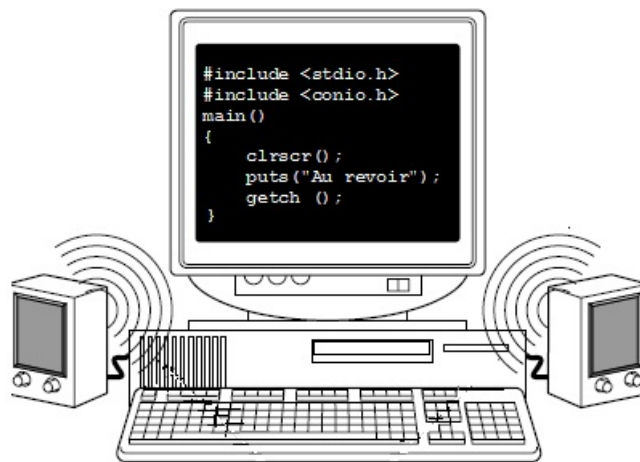




ENSET Libreville

Département informatique - Options SI & STI



Algorithmique et Programmation en langage C

Note de cours

William MOUCKEYTOU, enseignant E.N.S.E.T

2023

Table des matières

Objectif et plan du cours	7
1 La notion d'algorithme	8
1.1 Qu'est-ce que l'algorithmique ?	8
1.1.1 Définition	8
1.1.2 Historique	8
1.1.3 Double problématique de l'algorithmique	8
1.1.4 Différences entre algorithmes et programmes	9
1.1.5 Démarche de programmation et analyse descendante	9
1.1.6 Structure d'un algorithme	10
1.2 De l'algorithme au programme	10
1.3 Le pseudo-langage	10
1.4 Les variables	11
1.4.1 Déclaration des variables	11
1.4.1.1 Noms de variables	11
1.4.1.2 Types de variables	11
1.4.2 Les erreurs à éviter	12
1.5 L'entrée/sortie	12
1.5.1 Lecture	13
1.5.2 Écriture	13
1.6 Les constantes	13
1.7 L'affectation, l'expression et les opérateurs	13
1.7.1 Affectation	13
1.7.2 Expression	14
1.7.3 Opérateurs	14
1.7.4 Priorité des opérateurs	14
1.7.5 Opérateurs booléens	14
1.8 Les instructions	15
1.8.1 Instructions conditionnelles	15
1.8.1.1 Conditions simples	15
1.8.1.2 Conditions composées	16
1.8.1.3 Tests imbriqués	17
1.8.1.4 Instruction Cas	17
1.8.2 Instructions itératives : les boucles	17
1.8.2.1 Boucle TantQue	18
1.8.2.2 Boucle Répéter ... jusqu'à ...	19
1.8.2.3 Différences entre les boucles TantQue et Répéter ... jusqu'à	19
1.8.2.4 Boucle Pour ... jusqu'à ...	19
1.8.2.5 Lien entre Pour et TantQue	20
1.8.2.6 Boucles imbriquées	20
1.8.2.7 Choix d'un type de boucle	21
1.9 Procédures et fonctions	21
1.9.1 Signature d'une fonction	21

1.9.1.1	Erreurs à éviter	22
2	Les structures de données	23
2.1	Principe de base	23
2.2	Structures de données statiques : Tables ou Tableaux	24
2.2.1	Tableaux à une dimension	24
2.2.1.1	Accès et modification	24
2.2.1.2	Initialisation	25
2.2.2	Tableaux à deux dimensions	25
2.2.2.1	Lecture d'une matrice	26
2.2.2.2	Affichage d'une matrice	26
2.2.2.3	Initialisation d'une matrice	26
2.2.2.4	Quelques opérations sur les matrices	26
2.3	Enregistrement	27
2.3.1	Utilisation des enregistrements	28
3	Historique et présentation du langage	29
3.1	Historique du langage C	29
3.2	Présentation du langage	30
3.3	Pourquoi utiliser le langage C ?	30
4	Comment démarrer ?	32
4.1	Notion des langages informatiques	32
4.2	Avant de programmer	32
4.3	Cycle de développement du programme	32
5	Éléments de langage C	34
5.1	Structure d'un programme C	34
5.1.1	Exemple de programme	34
5.1.2	Structure du programme	35
5.1.2.1	La fonction main()	35
5.1.2.2	Appel d'un fichier #include	35
5.1.2.3	Définition de variable	35
5.1.2.4	Déclaration de fonction	35
5.1.2.5	Instructions	35
5.1.2.6	Définition de fonction	35
5.1.2.7	Commentaires du programme	36
5.1.2.8	Accolades	36
5.1.3	Remarques	36
5.1.4	Pour aller plus loin	36
5.2	Jeux de caractères	36
5.3	Identificateurs	37
5.4	Catégories et portée des identificateurs	38
6	Types élémentaires	39
6.1	Types entiers	39
6.1.1	Déclarations de variables	40
6.1.2	Mot-clé typedef	40
6.1.3	Initialisation des variables entières	41
6.2	Types flottants ou réels	41
6.3	Types caractères	42

7	Constantes	43
7.1	Constantes entières	43
7.2	Constantes flottantes	43
7.3	Constantes caractères	44
7.4	Constantes littérales et Constantes symboliques	45
7.4.1	Constantes littérales	45
7.5	Constantes symboliques	45
7.5.1	Fonctionnement de <code>#define</code>	45
7.5.2	Définition des constantes avec le mot-clé <code>const</code>	46
8	Expressions et opérateurs	47
8.1	Expressions	47
8.2	Opérateurs	47
8.2.1	Opérateurs arithmétiques	48
8.2.1.1	Opérateurs arithmétiques unaires	48
8.2.1.2	Opérateurs arithmétiques binaires	49
8.2.1.3	Opérateurs d'affectation	49
8.2.2	Opérateurs de comparaison	50
8.2.3	Opérateurs logiques	51
8.2.4	Opérateurs bit à bit	52
8.2.5	Opérateurs d'accès à la mémoire	52
8.2.6	Autres opérateurs	53
9	Conversions de type	55
9.1	Conversions automatiques de types	55
9.1.1	Promotion de type dans une expression	55
9.1.2	Conversion par affectation	56
9.2	Conversions explicites : l'opérateur de <code>cast</code>	56
9.2.1	Conversion par « <code>cast</code> » dans les expressions arithmétiques	56
9.2.2	Conversion par « <code>cast</code> » appliquée aux pointeurs	57
10	Structures de contrôle	58
10.1	Instructions	58
10.2	Blocs et expressions	58
10.3	Instructions de contrôle	59
10.3.1	Instruction <code>if</code>	59
10.3.2	Instruction <code>if ... else</code>	59
10.3.3	Instruction <code>switch</code>	60
10.3.4	Boucle <code>for</code>	62
10.3.5	Boucle <code>while</code>	63
10.3.6	Boucle <code>do-while</code>	65
10.3.7	Instruction <code>goto</code>	66
10.3.8	Instruction <code>continue</code>	66
10.3.9	Instruction <code>break</code>	66
10.3.10	Instruction <code>return</code>	67
11	Principes de base des entrées/sorties	68
11.1	Flots du C	68
11.1.1	Définition des entrées/sorties	68
11.1.2	Définition d'un flot	68
11.1.3	Flots de texte versus flots binaires	69
11.1.4	flots prédéfinis	69
11.2	Les fonctions d'entrées/sorties	69
11.3	Fonction <code>printf()</code>	69

11.3.1	Chaînes format de print()	70
11.3.2	Spécifications de conversion de printf()	70
11.3.3	Syntaxe de la fonction printf()	70
11.4	Fonction puts()	71
11.5	Lecture de données numériques avec scanf()	71
11.6	Lecture et écriture de caractères	72
11.6.1	Fonction getchar()	72
11.6.2	Fonction putchar()	73
11.6.3	Fonctions putc() et fputc()	73
11.7	Ouverture/fermeture de fichier	74
11.7.1	Ouverture	74
11.7.2	Fermeture	74
11.8	Lecture/écriture de fichier	74
11.8.1	Accès textuel	75
11.8.2	Accès binaire	75
12	Tableaux	76
12.1	Définition	76
12.1.1	Tableaux à une dimension	76
12.1.2	Tableaux à plusieurs dimensions	77
12.2	Nom et déclaration des tableaux	77
12.2.1	Initialisation de tableaux à une dimensions	78
12.2.2	Initialisation de tableaux à plusieurs dimensions	78
12.2.3	Taille maximale	78
13	Pointeurs	79
13.1	Définition	79
13.1.1	Mémoire de votre ordinateur	79
13.1.2	Création d'un pointeur	80
13.2	Pointeurs et variables simples	80
13.3	Déclaration	80
13.3.1	Initialisation	81
13.4	Pointeurs, mode d'emploi	81
13.5	Pointeurs et types de variables	82
13.6	Pointeurs et tableaux	82
13.6.1	Noms de tableau et pointeurs	83
13.6.2	Stockage des éléments d'un tableau	83
13.6.3	Pointeur arithmétique	84
13.6.3.1	Incrémenter les pointeurs	84
13.6.3.2	Décrémenter les pointeurs	84
13.6.3.3	Autre utilisation des pointeurs	84
13.7	Précautions d'emploi	85
13.8	Pointeurs et index de tableaux	85
14	Chaînes de caractères	86
14.1	Tableaux de caractères	86
14.2	Initialiser les tableaux de caractères	86
14.3	Chaînes et pointeurs	87
14.4	Chaînes sans tableaux	87
14.4.1	Allouer la mémoire nécessaire à la compilation	87
14.4.2	Fonction malloc()	87
14.4.2.1	Syntaxe de la fonction malloc()	88
14.4.2.2	Utilisation de malloc()	88
14.4.3	Fonction calloc	88

14.4.4	Fonction <code>realloc()</code>	89
14.4.5	Fonction <code>free()</code>	89
14.5	Affichage de chaînes et de caractères	89
14.6	Lecture des chaînes de caractères	89
14.6.1	Fonction <code>fgets()</code>	89
14.6.2	Fonctions <code>scanf()</code> et <code>fscanf()</code>	90
14.7	Manipulation de chaîne de caractères	90
15	Types évolués	94
15.1	Définition d'un nouveau type	94
15.1.1	Commande <code>typedef</code>	94
15.1.2	Types qui représentent des tableaux	94
15.1.3	<code>typedef</code> vs. <code>#define</code>	95
15.2	Structures	95
15.2.1	Structures simples	95
15.2.1.1	Définition et déclaration des structures	95
15.2.1.2	Accès aux membres d'une structure	96
15.2.2	Structures plus complexes	96
15.2.2.1	Structures contenant des structures	96
15.2.2.2	Tableaux membres de structures	97
15.2.3	Tableaux de structures	98
15.2.4	Initialisation des structures	98
15.2.5	Structures et pointeurs	99
15.2.5.1	Pointeurs membres d'une structure	99
15.2.5.2	Pointeurs vers des structures	99
15.2.5.3	Pointeurs et tableaux de structures	100
15.2.5.4	Passage de structures comme arguments de fonctions	101
15.3	Unions	102
15.3.1	Définition, déclaration et initialisation des unions	102
15.3.2	Accéder aux membres d'une union	102
15.4	Structures et <code>typedef</code>	102
16	Liste chaînées	104
16.1	Principes de base des listes chaînées	104
16.2	Utilisation de listes chaînées	105
16.2.1	Préliminaire	105
16.2.2	Ajout d'un maillon dans une liste	105
16.2.3	Ajout d'un maillon en queue de liste	106
16.2.4	Ajout d'un maillon au milieu	107
16.2.5	Suppression d'un maillon d'une liste	107
16.3	Un exemple de liste chaînée	108
17	Portée des variables	110
17.1	Définition de la portée et importance de cette notion	110
17.2	Variables externes	110
17.2.1	Portée des variables externes	110
17.2.2	Quand utiliser les variables externes	111
17.2.3	Mot clé <code>extern</code>	111
17.3	Variables locales	111
17.3.1	Variables statiques et automatiques	111
17.3.2	Portée des paramètres d'une fonction	111
17.3.3	Variables statiques externes	111
17.3.4	Variables de type <code>register</code>	112
17.4	Variables locales et la fonction <code>main()</code>	112

18 Retour sur les fonctions	113
18.1 Prototypes de fonctions	113
18.2 Définitions de fonctions	114
18.3 Appels de fonction	115
18.3.1 Passage de pointeurs à une fonction	115
18.3.2 Fonctions renvoyant un pointeur	116

Objectif et plan du cours

■ Objectif :

- Apprendre les concepts de base de l'algorithmique et de la programmation
- Être capable de mettre en œuvre ces concepts pour analyser des problèmes simples et écrire les programmes correspondants
- Apprendre le langage de programmation le plus utilisé en informatique industrielle

■ Plan :

- Introduction à l'algorithmique et à la programmation
 - Généralités sur l'algorithmique et les langages de programmation
 - Notion de variable, affectation, lecture et écriture
 - Instructions conditionnelles et instructions itératives
 - Les Tableaux, les fonctions et procédures, la récursivité
 - Structures de données
- Initiation au Langage C (Travaux pratiques)

Chapitre 1

La notion d'algorithme

L'art de programmer, c'est l'art de faire résoudre des problèmes par des machines.

Un ordinateur est dénué d'intelligence ; il ne peut donc résoudre que les problèmes pour lesquels existe une méthode de résolution algorithmique.

Ce chapitre étudie la démarche algorithmique qui part de l'analyse d'un problème pour déterminer un algorithme qui résout ce problème de manière effective. La première étape importante est celle d'écrire (ou de décrire) l'algorithme. Celui-ci peut être vu, dans de nombreux cas, comme une procédure de calcul que doit effectuer un ordinateur abstrait. L'étape suivante consiste à rendre l'algorithme opérationnel en le traduisant dans un langage de programmation approprié et directement exploitable par ordinateur. Des exemples simples illustreront la diversité des problèmes qui relèvent de méthodes algorithmiques.

1.1 Qu'est-ce que l'algorithmique ?

1.1.1 Définition

Un **algorithme** est suite finie d'opérations élémentaires constituant un schéma de calcul ou de résolution d'un problème.

Ainsi, l'**algorithmique** est la science des algorithmes.

1.1.2 Historique

Le mot « algorithme » provient de la forme latine (*Algorismus*) du nom du mathématicien arabe ALKHAREZMI ou AL-KHWĀRIZMĪ auteur – entre autres mais ce n'est pas le plus important – d'un manuel de vulgarisation sur le calcul décimal positionnel indien (v. 830) expliquant son utilisation et, surtout, la manipulation des différents algorithmes permettant de réaliser les opérations arithmétiques classiques (addition, soustraction, multiplication, division, extraction de racines carrées, règle de trois, etc.).

1.1.3 Double problématique de l'algorithmique

1. **Trouver une méthode** de résolution (exacte ou approchée) du problème.

- Soient trois nombres réels a , b et c , quelles sont les solutions de l'équation $ax^2 + bx + c = 0$? (Résultat bien connu.)
- Soient cinq nombres réels a , b , c , d et e , quelles sont les solutions de l'équation $ax^5 + bx^4 + cx^3 + dx^2 + ex + f = 0$? (Pas de méthode générale.)

2. Trouver une méthode **efficace**.

Savoir résoudre un problème est une chose, le résoudre efficacement en est une autre.

1.1.4 Différences entre algorithmes et programmes

L'élaboration d'un **algorithme** précède l'étape de **programmation**, **programmation** : ensemble des activités orientées vers la conception, la réalisation, le test et la maintenance de programmes.

Un **programme** est la réalisation (l'implémentation) d'un algorithme au moyen d'un **langage de programmation**, langage compris par l'ordinateur (sur une architecture donnée). Il s'agit de la mise en œuvre du principe. Par exemple, lors de la programmation on s'occupera parfois explicitement de la gestion de la mémoire (allocation dynamique en C) qui est un problème d'implémentation ignoré au niveau algorithmique.

Un **langage de programmation** est un code de communication, permettant à un être humain de dialoguer avec une machine en lui soumettant des instructions et en analysant les données matérielles fournies par le système.

Le **langage de programmation** est l'intermédiaire entre le programmeur et la machine.

Nous entendons par programmation, un ensemble des activités orientées vers la conception, la réalisation, le test et la maintenance de programmes.

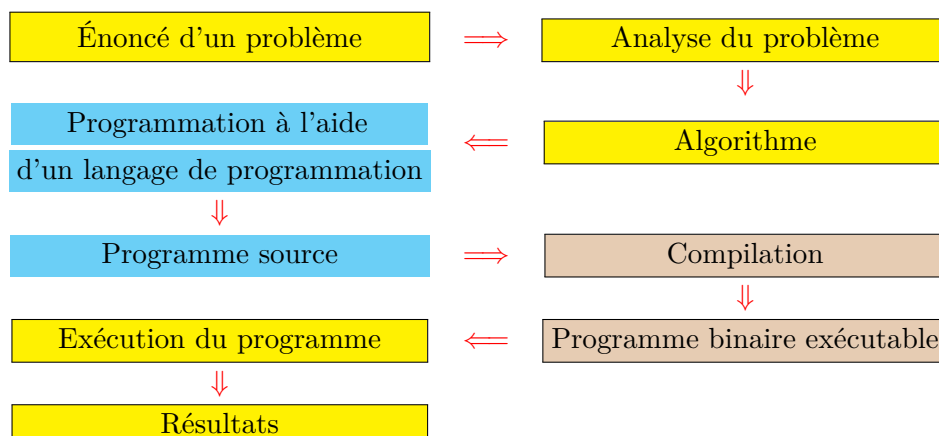
Il y a trois catégories de langage de programmations : les **langages interprétés** (Le langage HTML (les pages web), le langage Maple – calcul mathématique, Prolog – Intelligence artificielle, ..) et les **langages intermédiaires** (Java, Python,...) et les langages compilés (langage C, langage C++ – langage orienté objet, le Cobol, ..).

Le choix d'un langage de programmation n'est pas facile, chacun a ses spécificités et correspond mieux à certains types d'utilisations

1.1.5 Démarche de programmation et analyse descendante

La résolution d'un problème passe par toute une suite d'étapes :

1. Phase d'analyse et de réflexion (algorithmique)
2. Phase de programmation :
 - choisir un langage de programmation
 - traduction de l'algorithme en programme
 - programme (ou code) source
 - compilation : traduction du code source en code objet
 - traduction du code objet en code machine exécutable, compréhensible par l'ordinateur
3. Phase de test
4. Phase d'exécution



Le travail est ici surtout basé sur l'analyse du problème et l'écriture de l'**algorithme**. La réalisation d'un programme passe par l'**analyse descendante** du problème. Cette analyse consiste à décomposer un problème en sous-problèmes, et ainsi de suite, jusqu'à descendre au niveau des algorithmes connus. Durant l'écriture d'un programme, on peut être confronté à 2 types d'erreur :

- les erreurs syntaxiques : elles se remarquent à la compilation et sont le résultat d'une mauvaise écriture dans le langage de programmation.
- les erreurs sémantiques : elles se remarquent à l'exécution et sont le résultat d'une mauvaise analyse. Ces erreurs sont beaucoup plus graves car elles peuvent se déclencher en cours d'exploitation du programme.

1.1.6 Structure d'un algorithme

La structure d'un algorithme est divisée en deux parties :

- la première partie est une ligne qui indique le nom de l'algorithme.
- la deuxième partie, le traitement, situé entre les deux mots clés **Début** et **Fin**, contient le bloc d'instructions (voir §1.3.5).

Sa syntaxe :

```

Algorithme :   nom-de-l'algorithme           // partie en-tête
Variables :   indice : entier ;              // partie des déclarations des variables
Début
    {
        bloc d'instructions (voir §1.3.5)    // partie traitement
    }
Fin

```

1.2 De l'algorithme au programme

L'analyse du problème à résoudre, étape préliminaire dans la réalisation de l'algorithme, doit permettre tout d'abord d'identifier les données d'entrée de l'algorithme ainsi que les données de sortie qui fournissent la solution du problème. L'écriture d'un algorithme doit être concise et sans équivoque. Pour atteindre cet objectif, beaucoup d'ouvrages traitant d'algorithmes utilisent un pseudo-langage ou pseudo-code. Celui-ci a pour but de transcrire la solution du problème dans un langage destiné à une machine abstraite conçue comme un ordinateur virtuel constitué :

- d'une entrée au clavier
- d'une sortie (écran)
- d'une mémoire (dite centrale)
- d'un processeur, capable d'exécuter les instructions du pseudo-code
- éventuellement, d'autres composants plus spécifiques.

1.3 Le pseudo-langage

Un algorithme doit être lisible et compréhensible par plusieurs personnes. Il doit donc suivre des règles précises, il est composé d'une **entête** et d'un **corps** :

- l'entête, qui spécifie :
 - le nom de l'algorithme
 - les données *en entrée*, c.-à-d. les éléments qui sont indispensables à son bon fonctionnement
 - les données *en sortie*, c.-à-d. les éléments calculés, produits, par l'algorithme
 - les données locales à l'algorithmique qui lui sont indispensables
- le corps, qui est composé :
 - du mot clef **Début**
 - d'une suite d'**instructions** indentées
 - du mot clef **Fin**

Le plus important pour un algorithme sont les déclarations ainsi que les instructions qui constituent le corps de l'algorithme. Il existe des instructions qui ne servent qu'à la clarté de l'algorithme (l'ordinateur les ignore complètement), ce sont les **commentaires**.

Nous utiliserons un pseudo-langage (ou pseudo-code) proche du langage Pascal, mais délesté de toutes les contraintes syntaxiques inhérentes à un véritable langage de programmation. Cela a des avantages mais aussi des inconvénients, dont celui de ne pas écrire un code directement utilisable.

Nous distinguerons 6 concepts :

- les **variables**
- l'**entrée/sortie**
- les **constantes**
- l'**affectation, l'expression et les opérateurs**
- les **instructions**
- les **procédures et fonctions**

1.4 Les variables

Dans un programme informatique, on va avoir en permanence besoin de stocker provisoirement des valeurs. Il peut s'agir de données issues du disque dur, fournies par l'utilisateur (frappées au clavier). Ces données peuvent être de plusieurs types : elles peuvent être des nombres, du texte, le contenu d'un fichier, lui-même constitué de symboles, etc. Dès que l'on a besoin de stocker une information au cours d'un programme, on utilise une **variable**.

1.4.1 Déclaration des variables

La première chose à faire avant de pouvoir utiliser une variable est de **créer la boîte et de lui coller une étiquette**. Ceci se fait tout au début de l'algorithme, avant même les instructions proprement dites. C'est ce qu'on appelle la **déclaration des variables**. Une variable ne peut être utilisée que si elle est déclarée. La déclaration se fait par la donnée, du **nom** et du **type**, de la variable.

Exemples :

```
|| Variables i, j, k : entier
|| x, y : réel
|| OK : booléen
|| ch1, ch2 : chaîne de caractères
```

1.4.1.1 Noms de variables

Le nom de la variable (l'étiquette de la boîte) obéit à des règles qui changent selon le langage utilisé. Les principales règles à respecter sont :

- Le nom de variable peut comporter des lettres et des chiffres,
- On exclut la plupart des signes de ponctuation, en particulier les espaces.
- Un nom de variable doit commencer par une lettre.
- Le nombre maximal de caractères qui composent le nom d'une variable dépend du langage utilisé.
- Ne pas utiliser les mots clés du langage de programmation

1.4.1.2 Types de variables

Lorsqu'on déclare une variable, il ne suffit pas de créer une boîte (réserver un emplacement mémoire) ; il faut préciser ce que l'on voudra mettre dedans, car de cela dépendent la taille de la boîte (l'emplacement mémoire) et le **type de codage** utilisé.

Le type d'une variable détermine ainsi l'ensemble des valeurs qu'elle peut prendre. Les types offerts par la plus part des langages sont :

- **Type numérique (entier ou réel)**

Type Numérique	Octet	Plage
Byte	1	$[-128; 128]$ ou $[0; 255]$
Entier simple (ou court)	2	$[-32\,768; 32\,767]$
Entier double (ou long)	4	$[-2^{31}; 2^{31} - 1]$
Réel simple précision	4	$[-3,4 \times 10^{38}; -1,4 \times 10^{-45}]$ pour les négatifs $[1,4 \times 10^{-45}; 3,4 \times 10^{-45}]$ pour les positifs
Réel double précision	8	$[-1,79 \times 10^{308}; -4,94 \times 10^{-324}]$ pour les négatifs $[4,94 \times 10^{-324}; 1,79 \times 10^{308}]$ pour les positifs

La syntaxe d'une déclaration de variable numérique en pseudo-langage aura la forme :

```
|| Variable g : entier
|| Variables PrixHT, TauxTVA, PrixTTC : réel
```

- **Type logique ou booléen** : on y stocke uniquement les valeurs logiques VRAI ou FAUX. On peut représenter ces notions abstraites de VRAI et de FAUX par tout ce qu'on veut : de l'anglais (TRUE et FALSE) ou des nombres (0 et 1). Le type booléen est très économique en termes de place mémoire occupée, un seul bit suffit. La syntaxe d'une déclaration de variable logique en pseudo-langage aura la forme :

```
|| Variable OK : booléen
```

- **Type caractère** : Dans une variable de ce type, on stocke des caractères, qu'il s'agisse de lettres, de signes de ponctuation, d'espaces, ou même de chiffres. La syntaxe d'une déclaration de variable caractère aura la forme :

```
|| Variables 'A', 'b', '1', '?' : caractères
```

- **Type chaîne de caractères** : Un groupe de caractères est appelé **chaîne de caractères**. En pseudo-code, une chaîne de caractères est toujours notée entre guillemets " ", car, il peut y avoir une confusion entre des nombres et des suites de chiffres. Par exemple, 423 peut représenter :

- le nombre 423 (quatre cent vingt-trois),
- ou la suite de caractères 4, 2, et 3 notée : "423"

La syntaxe d'une déclaration de variable de type chaîne de caractères en pseudo-langage aura la forme :

```
|| Variables " ", " Nom, Prénom", "code postale : 1000" : chaîne de caractères
```

1.4.2 Les erreurs à éviter

- Une variable n'est déclarée qu'une seule fois dans l'algorithme.
- Une variable est déclarée au début de l'algorithme et non dans la partie réservée aux instructions de traitements.
- Avant de pouvoir utiliser une variable, il faut l'avoir déclarer dans le bloc des variables.
- Avant de pouvoir utiliser la valeur de la variable, une valeur doit lui être attribuée.

1.5 L'entrée/sortie

En pratique, l'ordinateur (concret ou virtuel) communique au moins par deux périphériques : un pour entrer les données et un autre pour sortir les résultats. Ceux-ci peuvent devenir les données d'entrée d'un autre algorithme et ainsi de suite, mais pour un nombre fini de fois.

1.5.1 Lecture

La **lecture** permet d'entrer des données à partir du clavier. En pseudo-code, on note :

```
|| Lire(var)
```

La machine met la valeur entrée au clavier dans la zone mémoire nommée var. Le programme s'arrête lorsqu'il rencontre une instruction *Lire* et ne se poursuit qu'après la saisie de l'entrée attendue par le clavier et de la touche Entrée (cette touche signale la fin de l'entrée)

Conseil : Avant de lire une variable, il est fortement conseillé d'écrire des messages à l'écran, afin de prévenir l'utilisateur de ce qu'il doit frapper

1.5.2 Écriture

L'**écriture** permet d'afficher des résultats à l'écran (ou de les écrire dans un fichier). En pseudo-code, on note :

```
|| Écrire(liste d'expression)
```

la machine affiche les valeurs des expressions décrite dans la liste. Ces instructions peuvent être des variables ayant des valeurs, des nombres ou des commentaires sous forme de chaînes de caractères.

Exemples :

```
|| Écrire(a, b+2, "Message")
```

1.6 Les constantes

Une constante est une variable dont la valeur ne change pas au cours de l'exécution du programme, elle peut être un nombre, un caractère, ou une chaîne de caractères. En pseudo-code, *Constante identificateur=valeur : type,...* (par convention, les noms de constantes sont en majuscules)

Exemple : pour calculer la surface des cercles, la valeur de π est une constante mais le rayon est une variable.

```
|| Constante PI=3.14 : réel , MAXI=32 : entier
```

Une constante doit toujours recevoir une valeur dès sa déclaration.

1.7 L'affectation, l'expression et les opérateurs

1.7.1 Affectation

L'**affectation** est l'action élémentaire dont l'effet est de donner une valeur à une variable (ranger une valeur à une place). L'affectation est réalisée au moyen de l'opérateur $\leftarrow$ (ou $=$ en C et $:=$ en Pascal). Elle signifie "prendre la valeur se trouvant du côté droit (souvent appelée *rvalue*) et la copier du côté gauche (souvent appelée *lvalue*)". Une *rvalue* représente toute constante, variable ou expression capable de produire une valeur, mais une *lvalue* doit être une variable distincte et nommée (autrement dit, il existe un emplacement physique pour ranger le résultat). Par exemple, on peut affecter une valeur constante à une variable ($A \leftarrow 4$), mais on ne peut pas affecter quoi que ce soit à une valeur constante - elle ne peut pas être une *lvalue* (on ne peut pas écrire $4 \leftarrow A$)

Exemple :

```
|| X  $\leftarrow$  3
```

Signifie mettre la valeur 3 dans la case identifiée par X. À l'exécution de cette instruction, la valeur 3 est rangée en X (nom de la variable).

La valeur correspond au contenu : 3

La variable correspond au contenant : X

L'affectation est différente d'une équation mathématique :

- Les opérations $x \leftarrow x + 1$ et $x \leftarrow x - 1$ ont un sens en programmation et se nomment respectivement *incrément* et *décrément*.
- $A + 1 \leftarrow 3$ n'est pas possible en langages de programmation et n'est pas équivalente à $A \leftarrow 2$

Certains langages donnent des valeurs par défaut aux variables déclarées. Pour éviter tout problème il est préférable d'*initialiser les variables* déclarées.

1.7.2 Expression

Une **expression** peut être une valeur, une variable ou une opération constituée de variables reliées par des **opérateurs**

Exemple :

|| 1, b, a*2, a+3*b-c, ...

L'évaluation de l'expression fournit une valeur unique qui est le résultat de l'opération

1.7.3 Opérateurs

Un **opérateur** est un signe qui relie deux valeurs, pour produire un résultat. Les opérateurs dépendent du type de l'opération, ils peuvent être :

- des **opérateurs arithmétiques** : +, -, *, %(modulo), ^ (puissance)
- des **opérateurs logiques** : NON(!), OU(|), ET(&&)
- des **opérateurs relationnels** : =, <, >, <=, >=
- des **opérateurs alphanumérique** : &(concaténation)

1.7.4 Priorité des opérateurs

Pour les opérateurs arithmétiques donnés ci-dessus, l'ordre de priorité est le suivant (du plus prioritaire au moins prioritaire) :

- () : les parenthèses ↓
- ^ : (élévation à la puissance)
- *, / (multiplication, division)
- +, - (addition, soustraction)

Par exemple : $9 + 3 * 4$ vaut 21

En cas de besoin, on utilise les parenthèses pour indiquer les opérations à effectuer en priorité. Par exemple : $(9 + 3) * 4$ vaut 48

À priorité égale, l'évaluation de l'expression se fait de gauche à droite

1.7.5 Opérateurs booléens

- Associativité des opérateurs ET et OU.

$$a \text{ ET } (b \text{ ET } c) = (a \text{ ET } b) \text{ ET } c$$

- Commutativité des opérateurs ET

$$a \text{ ET } b = b \text{ ET } a$$

$$a \text{ OU } b = b \text{ OU } a$$

- Distributivité des opérateurs ET et OU.

$$a \text{ OU } (b \text{ ET } c) = (a \text{ OU } b) \text{ ET } (a \text{ OU } c)$$

$$a \text{ ET } (b \text{ OU } c) = (a \text{ ET } b) \text{ OU } (a \text{ ET } c)$$

- Involution (homographie réciproque) : $\text{NON NON } a = a$
- Loi de Morgan :

$$\text{NON}(a \text{ OU } b) = \text{NON } a \text{ ET } \text{NON } b$$

$$\text{NON}(a \text{ ET } b) = \text{NON } a \text{ OU } \text{NON } b$$

Remarque :

1. On ne peut pas additionner un entier et un caractère
2. Toutefois dans certains langages on peut utiliser un opérateur avec deux opérandes de types différents, c'est par exemple le cas avec les types arithmétiques ($4 + 5.5$)
3. La signification d'un opérateur peut changer en fonction du type des opérandes
 - l'opérateur $+$ avec des entiers effectue l'addition, $3 + 6$ vaut 9
 - avec des chaînes de caractères il effectue la **concaténation** "bonjour" + " tout le monde" vaut "bonjour tout le monde"
4. Pour le langage C, si x et y sont entiers, x/y est une division entière alors que si l'un des deux ne l'est pas la division est réelle.
5. $x+y/z$: est une expression arithmétique dont le type dépend des types de x , y et z .
6. $(x>y) \parallel !(x=y+1)$: est une expression booléenne ($\parallel$ dénote l'opérateur logique ou et $!$ dénote la négation)
7. Avant d'utiliser une variable dans une expression, il est nécessaire qu'une valeur lui ait été affectée.

1.8 Les instructions

Une **instruction** est, en quelque sorte, un ordre à exécuter qui peut se présenter sous des formes diverses.

Exemples :

```
|| Écrire("Le nombre P est premier")
```

1.8.1 Instructions conditionnelles

1.8.1.1 Conditions simples

Une **condition** est une expression écrite entre parenthèse à valeur booléenne. Les instructions conditionnelles servent à n'exécuter une instruction ou une séquence d'instructions que si une condition est vérifiée.

En pseudo-code :

```
|| Si condition alors
||     instruction ou suite d'instructions1
|| Sinon
||     instruction ou suite d'instructions2
|| FinSi
```

Une **condition** est ainsi une comparaison. C'est-à-dire qu'elle est composée de trois éléments :

- une valeur
- un opérateur de comparaison
- une autre valeur

Les **valeurs** peuvent être a priori de n'importe quel type (numériques, caractères...)

Les **opérateurs de comparaison** sont : $=$ $\neq$ $<$ $>$ $=<$ $>=$

L'ensemble constitue donc si l'on veut une affirmation, qui a un moment donné est VRAI ou FAUX.

Exemples :

Algorithme : ValeurAbsolu1
Variable x : réel

Début

Écrire("Entrez un réel : ")

 Lire(x)

 Si $x < 0$ **alors**

 Écrire ("la valeur absolue de ", x , "est :", $-x$)

 Sinon

 Écrire ("la valeur absolue de ", x , "est :", x)

 FinSi
Fin
Algorithme : ValeurAbsolu2
Variable x, y : réel

Début

Écrire("Entrez un réel : ")

 $y \leftarrow x$

 Lire(x)

 Si $x < 0$ **alors**

 $y \leftarrow -x$

 FinSi

 Écrire ("la valeur absolue de ", x , "est :", y)

Fin

A noter que ces opérateurs de comparaison s'emploient tout à fait avec des caractères. Ceux-ci sont codés par la machine dans l'ordre alphabétique, les majuscules étant systématiquement placées avant les minuscules. Ainsi on a :

```
"t" < "w"           VRAI
"Maman" > "Papa"     FAUX
"maman" > "Papa"     VRAI
```

1.8.1.2 Conditions composées

Certains problèmes exigent parfois de formuler des conditions qui ne peuvent pas être exprimées sous la forme simple exposée ci-dessus. Prenons le cas " n est compris entre 5 et 8". En fait cette phrase cache non une, mais deux conditions. Car elle revient à dire que " n est supérieur à 5 et n est inférieur à 8". Il y a donc bien là deux conditions, reliées par ce qu'on appelle un opérateur logique, le mot ET

Comme on l'a évoqué plus haut, l'informatique met à notre disposition trois opérateurs logiques : ET, OU, et NON. Le ET a le même sens en informatique que dans le langage courant. Il faut se méfier un peu plus du OU. Pour que $c1$ OU $c2$ soit VRAI, il suffit que $c1$ soit VRAI ou que $c2$ soit VRAI. Le point important est que si $c1$ est VRAI et $c2$ est VRAI, alors $c1$ OU $c2$ est VRAI. Le OU informatique ne veut donc pas dire "ou bien"

Exemples :

```
 $x$  compris entre 2 et 6 :     $(x \geq 2)$  ET  $(x \leq 6)$ 
 $n$  divisible par 3 ou par 2 :  $(n \% 3 = 0)$  OU  $(n \% 2 = 0)$ 
```

L'évaluation d'une condition composée se fait selon des règles présentées généralement dans ce qu'on appelle tables de vérité.

$c1$	$c2$	$c1$ ET $c2$	$c1$ OU $c2$
VRAI	VRAI	VRAI	VRAI
VRAI	FAUX	FAUX	VRAI
FAUX	VRAI	FAUX	VRAI
FAUX	FAUX	FAUX	FAUX

1.8.1.3 Tests imbriqués

Les tests peuvent avoir un degré quelconque d'imbrications.

En pseudo-code :

```

Si condition1 alors
  Si condition2 alors
    instructionsA
  Sinon
    instructionsB
  FinSi
Sinon
  Si condition3 alors
    instructionsC
  FinSi
FinSi

```

Exemple1 :

```

Variable n : entier
Début
  Écrire("Entrez un nombre :")
  Lire(n)
  Si n < 0 alors
    Écrire("Ce nombre est négatif")
  Sinon
    Si n = 0 alors
      Écrire("Ce nombre est nul")
    Sinon
      Écrire("Ce nombre est positif")
    FinSi
  FinSi
Fin

```

Exemple2 :

```

Variable n : entier
Début
  Écrire("Entrez un nombre :")
  Lire(n)
  Si n < 0 alors
    Écrire("Ce nombre est négatif")
  FinSi
  Si n = 0 alors
    Écrire("Ce nombre est nul")
  FinSi
  Si n > 0 alors
    Écrire("Ce nombre est positif")
  FinSi
Fin

```

Remarque : dans l'exemple 2 on fait trois tests systématiquement alors que dans l'exemple 1, si le nombre est négatif on ne fait qu'un seul test

Conseil : utiliser les tests imbriqués pour limiter le nombre de tests et placer d'abord les conditions les plus probables

1.8.1.4 Instruction Cas

Lorsque l'on doit comparer une *même* variable avec plusieurs valeurs, comme par exemple :

En pseudo-code :

```

Cas où v vaut
  v1 : action1
  v2 : action2
  ...
  vn : actionn
  autre : action autre
FinCas

```

$v_1, \dots, v_n$ sont des **constantes** de type **scalaire** (entier, naturel, énuméré, ou caractère)

action i est exécutée si $v = v_i$ (on quitte ensuite l'instruction cas)

action autre est exécutée si quelque soit i , $v \neq v_i$

1.8.2 Instructions itératives : les boucles

La notion d'itération (boucle) est une des notions fondamentales de l'algorithmique. On l'utilise souvent quand on doit exercer plusieurs fois le même traitement sur un même objet, ou plusieurs objets

de même nature. Mais son réel intérêt réside dans le fait que l'on peut modifier, à chaque répétition, les objets sur lesquels s'exerce l'action répétée.

Les boucles permettent un contrôle de saisie (pour vérifier que les données entrées au clavier correspondent bien à celles attendues par l'algorithme) afin d'éviter qu'un programme puisse se planter par une erreur d'exécution.

Il existe trois façons d'exprimer algorithmiquement l'itération :

- **TantQue**
- **Répéter ... jusqu'à ...**
- **Pour ... jusqu'à ...**

1.8.2.1 Boucle TantQue

En pseudo-code :

```

TantQue condition
...
instructions
...
FinTantQue

```

Le principe est simple : le programme arrive sur la ligne du **TantQue**. Il examine alors la valeur de la condition. Si cette valeur est VRAI, le programme exécute les instructions qui suivent, jusqu'à ce qu'il rencontre la ligne **FinTantQue**. Il retourne ensuite sur la ligne du **TantQue**, procède au même examen, et ainsi de suite. On ne s'arrête que lorsque la condition prend la valeur FAUX.

On constate que la condition (dite condition de contrôle de la boucle) est évaluée avant chaque itération et si la condition est fausse, on sort de la boucle et on exécute l'instruction qui est après **FinTantQue**. Il est possible que les instructions à répéter ne soient jamais exécutées.

Exemple :

```

Variable C : caractère
Début
    Écrire("Entrez une lettre majuscule :")
    Lire(C)
    TantQue (C < 'A' OU C > 'Z')
        Écrire("Saisie erronée. Recommencez")
        Lire(C)
    FinTantQue
    Écrire("Saisie valable")
Fin

```

Remarque :

1. Le nombre d'itérations dans une boucle **TantQue** n'est pas connu au moment d'entrée dans la boucle. Il dépend de l'évolution de la valeur de la condition
 2. Une des instructions du corps de la boucle doit absolument changer la valeur de la condition de vrai à faux (après un certain nombre d'itérations), sinon le programme va tourner indéfiniment.
- Attention aux boucles infinies**

Exemple de boucle infinie :

```

i ← 1
TantQue i > 0
    i ← i + 1
FinTantQue

```

Correction :

```

i ← 1
TantQue i < 100
    i ← i + 1
FinTantQue

```

1.8.2.2 Boucle Répéter ... jusqu'à ...

En pseudo-code :

```

|| Répéter condition
||   ...
||   instructions
||   ...
|| jusqu'à

```

Le principe est simple : toutes les instructions écrites entre **Répéter** et **jusqu'à** sont exécutées au moins une fois et leur exécution est répétée jusqu'à ce que la condition placée derrière **jusqu'à** soit satisfaite.

Exemple :

```

|| Variables som,i : entier
|| Début
||   som ← 0
||   i ← 0
||   Répéter
||     i ← i + 1
||     som ← som + i
||   jusqu'à (som > 100)
||   Écrire("La valeur cherchée est N= ", i)
|| Fin

```

1.8.2.3 Différences entre les boucles TantQue et Répéter ... jusqu'à

- La séquence d'instructions est exécutée au moins une fois dans la boucle **Répéter .. jusqu'à** , alors qu'elle peut ne pas être exécutée dans le cas du **TantQue** .
- la séquence d'instructions est exécutée si la condition est vraie pour **TantQue** et si la condition est fausse pour **Répéter ... jusqu'à**.
- Dans les deux cas, la séquence d'instructions doit nécessairement faire évoluer la condition, faute de quoi on obtient une boucle infinie

1.8.2.4 Boucle Pour ... jusqu'à ...

En pseudo-code :

```

|| Pour i allant de début jusqu'à fin
||   ...
||   instructions
||   ...
|| FinPour

```

Le principe est simple :

- on initialise i(appelé compteur) par début
- on teste si on n'a pas dépassé fin : si oui on sort de la boucle sinon
- on exécute les instructions
- on incrémente i ($i \leftarrow i + 1$)
- on teste si on n'a pas dépassé fin : si oui on sort de la boucle sinon
- etc.

Exemple :

```

|| Variables i : entier
|| Pour i allant de 1 à 100 faire
||   Écrire("Bonjour à tous")
|| FinPour

```

Remarque :

1. Le nombre d'itérations dans une boucle **Pour** est connu avant le début de la boucle
2. Le compteur, i , est une variable de type entier (ou caractère). Il doit être déclarée
3. Début et fin peuvent être des valeurs, des variables définies avant le début de la boucle ou des expressions de même type que i . La valeur début est affectée à la variable i .
4. Il faut éviter de modifier la valeur du compteur i (et de fin) à l'intérieur de la boucle. En effet, une telle action :
 - perturbe le nombre d'itérations prévu par la boucle **Pour**
 - rend difficile la lecture de l'algorithme
 - présente le risque d'aboutir à une boucle infinie

1.8.2.5 Lien entre **Pour** et **TantQue**

La boucle **Pour** est un cas particulier de **TantQue** (cas où le nombre d'itérations est connu et fixé) . Tout ce qu'on peut écrire avec **Pour** peut être remplacé avec **TantQue** (la réciproque est fausse)
En pseudo-code : on suppose que fin > début

Boucle **Pour** :

```

Pour  $i$  allant de début à fin faire
    instructions
FinPour
  
```

Boucle **TantQue** :

```

 $i \leftarrow$  début
TantQue  $i \leq$  fin
    instructions
     $i \leftarrow i + 1$ 
FinTantQue
  
```

Exemple1: Calcul de x^n

```

Variable  $x, \text{puiss}$  : réel
            $n, i$  : entier
Début
    Écrire("Entrez respectivement les valeurs de x et n")
    Lire( $x, n$ )
     $\text{puiss} \leftarrow 1$ 
     $i \leftarrow 1$ 
    TantQue (  $i \leq n$  )
         $\text{puiss} \leftarrow \text{puiss} * x$ 
         $i \leftarrow i + 1$ 
    FinTantQue
    Écrire(" $x$ , " à la puissance ",  $n$ , " est égal à ",  $\text{puiss}$ )
Fin
  
```

Exemple2: Calcul de x^n :

```

Variable  $x, \text{puiss}$  : réel
            $n, i$  : entier
Début
    Écrire("Entrez respectivement les valeurs de x et n")
    Lire( $x, n$ )
     $\text{puiss} \leftarrow 1$ 
    Pour  $i$  allant de 1 à  $n$  faire
         $\text{puiss} \leftarrow \text{puiss} * x$ 
    FinPour
    Écrire(" $x$ , " à la puissance ",  $n$ , " est égal à ",  $\text{puiss}$ )
Fin
  
```

1.8.2.6 Boucles imbriquées

De même qu'une structure **SI ... ALORS** peut contenir d'autres structures **SI ... ALORS** , une boucle peut contenir d'autres boucles.

Exemple :

```

Pour i allant de 1 à 5
  Pour j allant de 1 à i
    Écrire("O")
  FinPour
  Écrire("K")
FinPour

```

Exécution :

```

OK
OOK
OOOK
OOOOK
OOOOOK

```

1.8.2.7 Choix d'un type de boucle

- Si on peut déterminer le nombre d'itérations avant l'exécution de la boucle, il est plus naturel d'utiliser la boucle **Pour**
- S'il n'est pas possible de connaître le nombre d'itérations avant l'exécution de la boucle, on fera appel à l'une des boucles **TantQue** ou **Répéter ... jusqu'à**.
- Pour le choix entre **TantQue** et **Répéter ... jusqu'à** :
 - Si on doit tester la condition de contrôle avant de commencer les instructions de la boucle, on utilisera **TantQue**
 - Si la valeur de la condition de contrôle dépend d'une première exécution des instructions de la boucle, on utilisera **Répéter ... jusqu'à**

1.9 Procédures et fonctions

Une **fonction** non primitive est définie par un algorithme (ou un programme) qui associe à une donnée d'entrée une valeur de sortie unique

Une **procédure** est un algorithme (ou un programme) plus général qu'une fonction. Elle assemble une ou plusieurs instructions et, contrairement à une fonction, ne donne pas nécessairement une valeur de sortie (elle peut, par exemple, renvoyer 'a une adresse, arrêter le programme, demander des données, ...).

Fonctions et procédures ont pour vocation d'être appelées, sans changement, par d'autres algorithmes.

La **structure d'une fonction** est la suivante :

```

fonction nomDelaFonction (liste des paramètres) : typeRetourne
Début
  bloc d'instructions
Fin

```

La **structure d'une procédure** est la suivante :

```

fonction nomDelaProcédure (liste des paramètres) : vide
Début
  bloc d'instructions
Fin

```

1.9.1 Signature d'une fonction

La **signature** d'une fonction décrit les éléments permettant de l'appeler correctement :

- le nom de la fonction
- le type (et l'ordre) des paramètres
- le type de la valeur retournée

Un programmeur qui souhaite utiliser une fonction n'a pas besoin de connaître le corps de la fonction, la signature celle-ci suffit. Deux fonctions ayant des signatures différentes sont différentes.

1.9.1.1 Erreurs à éviter

Erreurs à éviter dans l'utilisation d'une fonction :

- Oublier les parenthèses
- Ne pas respecter le type retour
- Ne pas respecter le type des paramètres
- Ne pas réécrire une fonction qui existe déjà
- Croire que la fonction peut modifier la variable du programme appelant

Erreurs à éviter dans l'écriture d'une fonction :

- Donner le même nom à un paramètre et à une variable
- Placer plusieurs retours consécutifs dans la fonction
- Vouloir retourner plusieurs valeurs
- Oublier de retourner la valeur ou retourner la valeur du mauvais type
- Vouloir continuer un traitement après l'instruction retourne
- Pensez qu'en modifiant la valeur d'un paramètre, celui-ci sera modifié dans le programme appelant.

Chapitre 2

Les structures de données

2.1 Principe de base

Dans ce chapitre, nous étudions les méthodes élémentaires d'organisation des données au moyen de structures simples et faciles à traiter du double point de vue algorithmique et informatique. On est ainsi amené à poser le problème de la représentation des ensembles finis (structurés ou pas) et de construire des opérations de base sur ces ensembles. Les solutions adoptées devront tenir compte du fait que le nombre d'éléments d'un ensemble (de données) peut varier au cours de l'exécution d'un algorithme ou d'un programme. Nous considérerons les opérations *primitives* suivantes :

- **Tester** si un ensemble est vide ;
- **Ajouter** un élément à un ensemble ;
- **Rechercher** si un élément appartient à un ensemble ;
- **Supprimer** un élément d'un ensemble.

La manipulation des ensembles par une machine doit satisfaire à deux contraintes naturelles mais souvent contradictoires, la première est d'occuper un minimum de place en mémoire et la seconde, de réaliser les opérations primitives à l'aide d'un nombre restreint d'instructions machines. Notons bien que les opérations Tester et Ajouter ne dépendent pas de la taille de l'ensemble. Par contre, les opérations Rechercher et Supprimer ne peuvent pas être exécutées en un nombre d'instructions élémentaires indépendant du nombre d'éléments de l'ensemble, à moins d'en limiter leurs effets. Pour cette raison, on introduit des structures de données où les opérations de recherche et de suppression ne portent que sur le dernier ou le premier élément (structures en pile ou en file).

En fait, la manière de représenter les données en entrée et sortie est une étape importante dans la conception d'un algorithme, afin de réaliser une implémentation efficace : optimiser l'espace mémoire et les temps calculs. Il s'avère que dans de nombreuses applications, le choix d'une structure pour les données à traiter est souvent la principale difficulté à résoudre.

Les structures élémentaires que nous allons étudier sont les tables (arrays en anglais), les listes chaînées (*linked lists*), les piles (*stacks*) et les files d'attente (*queues*). A cela, il faut y associer le classement par type, qui a pour but de regrouper les données suivant la nature des valeurs ou des objets qu'elles représentent, ou encore suivant les opérations qui peuvent être effectuées sur elles, y compris leur mode de stockage. A côté des types primitifs standard tels que ENTIER (nombres entiers), REEL (nombres réels), BOOLEEN (valeurs vrai et faux) et CHAR (caractères alpha-numériques), il existe un grand nombre de types dérivés tels que les tables, les fonctions, les pointeurs, les suites de caractères (STRING)... Finalement, il est toujours possible de créer un type adapté à un problème donné, notamment par simple énumération de toutes les valeurs possibles du type en question. Par exemple, le type FORME peut être défini par l'ensemble $F = \{\text{triangle, carré, rectangle, losange, cercle, ellipse, ovale}\}$; évidemment, on veut assez de souplesse dans une telle définition pour pouvoir ajouter ou supprimer un forme.

2.2 Structures de données statiques : Tables ou Tableaux

2.2.1 Tableaux à une dimension

Le type tableau permet de stocker des valeurs de même type grâce à une seule variable.

Le type tableau : Un tableau structure un ensemble de valeurs de même type accessible par leur position.

Dimension, type et un indice d'un tableau : Le nombre maximal d'éléments du tableau, qui est précisé à la définition, s'appelle sa dimension. Le type de ses éléments s'appelle le type du tableau. Pour accéder aux éléments du tableau, un indice indique le rang de l'élément.

Un tableau est défini en deux temps. Le tableau nommé `tab` est déclaré de la manière suivante, ainsi que le type de base de ses éléments. En pseudo-code :

```
|| Variable tab :Tableau[dimension] de Type ;
```

- `tab` : identificateur
- `dimension` : représente le nombre de rangés dans le tableau.
une dimension : `[1..nbr_Max]`
deux dimensions : `[1..nbr_Max, 1..nb_Max]`
- `type` : entiers, réels, caractères, booléens,...

Exemple :

```
|| Variable notes : tableau[30] de réels
```

Pour accéder au 5ème élément , on écrit : `notes[4]`

Remarque :

1. Selon les langages, le premier indice du tableau est soit 0, soit 1. Le plus souvent c'est 0 (c'est ce qu'on va utiliser en pseudo-code). Dans ce cas, `tab[i]` désigne l'élément `i+1` du tableau `notes`
2. Il est possible de déclarer un tableau sans préciser au départ sa dimension. Cette précision est faite ultérieurement.
 - Par exemple, quand on déclare un tableau comme paramètre d'une procédure, on peut ne préciser sa dimension qu'au moment de l'appel
 - En tous cas, un tableau est inutilisable tant qu'on n'a pas précisé le nombre de ses éléments
3. Un grand avantage des tableaux est qu'on peut traiter les données qui y sont stockées de façon simple en utilisant des boucles
4. Les éléments d'un tableau s'utilisent comme des variables

2.2.1.1 Accès et modification

Les instructions de lecture, écriture et affectation s'appliquent aux tableaux comme aux variables.

Exemple1 :

```
|| x ← notes[0]   La variable x prend la valeur du premier élément du tableau
|| notes[6] ← 43  Cette instruction a modifiée le contenu du 7ème élément du tableau
```

Exemple2 : Le programme suivant comporte la déclaration d'un tableau de 20 réels (les notes d'une classe), on commence par effectuer la saisie des notes, et en suite on calcul la moyenne des 20 notes et on affiche la moyenne :

```
|| Constante Max = 200 : entier
|| Variables notes : tableau[Max] d'entiers
||           d,i,somme,n : entier
||           moyenne : réel
```

Début

```
Écrire("Entrez le nombre de notes(<200) : ")
```

```
Lire(n)
```

```
/* Saisir les notes */
```

```
Pour i allant de 0 à n-1 faire
```

```
    Écrire("Entrez une note : ")
```

```
    Lire(notes[i])
```

```
FinPour
```

```
somme ← 0
```

```
/* Effectuer la somme des notes */
```

```
Pour i allant de 0 à n-1 faire
```

```
    somme ← somme + notes[i]
```

```
FinPour
```

```
moyenne = somme/n
```

```
/* Afficher la moyenne */
```

```
Écrire("La moyenne des notes est : ", moyenne)
```

```
Fin
```

2.2.1.2 Initialisation

Le bloc d'instructions suivant initialise un à un tous les éléments d'un tableau de n éléments :

Algorithme : InitTableau

```
n, i : entier
```

Début

```
Écrire("Entrez la valeur de n")
```

```
Lire(n)
```

```
Pour i allant de 0 à n-1 faire
```

```
    tab[i] ← 0
```

```
FinPour
```

```
Fin
```

2.2.2 Tableaux à deux dimensions

Les langages de programmation permettent de déclarer des tableaux dans lesquels les valeurs sont repérées par deux indices. Ceci est utile par exemple pour représenter des matrices

En pseudo code, un tableau à deux dimensions se déclare ainsi :

```
|| Variable tab :Tableau[dimension1][dimension2] de Type ;
```

Exemple : une matrice A de 3 lignes et 4 colonnes dont les éléments sont réels

```
|| Variable A : tableau[3][4] de réels
```

A[i] [j] permet d'accéder à l'élément de la matrice qui se trouve à l'intersection de la ligne i et de la colonne j

La matrice A dans la déclaration suivante :

```
|| Variable A : tableau[3][7] de réels
```

peut être explicitée comme suit : les éléments sont rangés dans un tableau à deux entrées

	0	1	2	3	4	5	6
0	12	28	44	2	76	77	32
1	23	36	51	11	38	54	25
2	43	21	55	67	83	41	69

Ce tableau a 3 lignes et 7 colonnes. Les éléments du tableau sont repérés par leur numéro de ligne et leur numéro de colonne désignés en bleu. Par exemple A[1][4] vaut 38

2.2.2.1 Lecture d'une matrice

Le bloc d'instructions suivant saisi un à un tous les éléments d'une matrice de $N \times N$ éléments :

```

Constante N = 100 : entier
Variables i, j, n, m : entier
            A : tableau[N][N] de réels
Début
    Écrire("Entrez le nombre de lignes(<100) et le nombre de colonnes(<100) : ")
    Lire(n,m)
    Pour i allant de 0 à n-1 faire
        Écrire("Saisie de la ligne ",i+1)
        Pour j allant de 0 à m-1 faire
            Écrire("Entrez l'élément de la ligne ", i+1, " et de la colonne ", j+1)
            Lire(A[i][j])
        FinPour
    FinPour
Fin

```

2.2.2.2 Affichage d'une matrice

Le bloc d'instructions suivant affiche un à un tous les éléments d'une matrice de $N \times N$ éléments :

```

Constante N = 100 : entier
Variables i, j, n, m : entier
            A : tableau[N][N] de réels
Début
    Pour i allant de 0 à n-1 faire
        Pour j allant de 0 à m-1 faire
            Écrire("A[", i, "][", j, "] = ", A[i][j])
        FinPour
    FinPour
Fin

```

2.2.2.3 Initialisation d'une matrice

Pour initialiser une matrice on peut utiliser par exemple les instructions suivantes :

```

T1 [3] [3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}
T2 [3] [3] = {1, 2, 3, 4, 5, 6, 7, 8, 9}

```

2.2.2.4 Quelques opérations sur les matrices

Exemple 1 : Somme de deux matrices

```

Constante N = 100 : entier
Variables i, j, n : entier
            A : tableau[N][N], B : tableau[N][N], C : tableau[N][N] de réels
Début
    Écrire("Entrez la taille des matrices(<100) : ")
    Lire(n)
    Pour i allant de 0 à n-1 faire
        Pour j allant de 0 à n-1 faire
            C[i][j] ← A[i][j] + B[i][j]
        FinPour
    FinPour
Fin

```

Exemple 2 : Produit de deux matrices

```

Constante N = 20 : entier
Variables i, j, k, n, S : entier
            A : tableau[N][N], B : tableau[N][N], C : tableau[N][N] d' entiers

```

```

Début
  Écrire("Entrer la taille des matrices(<20) : ")
  Lire(n)
  Pour i allant de 1 à n faire
    Écrire("Saisir les éléments de la ligne ", i, " ligne de la matrice A : ")
    Pour j allant de 1 à n faire
      Lire(A[i][j])
    FinPour
  FinPour
  Pour i allant de 1 à n faire
    Écrire("Saisir les éléments de la ligne ", i, " ligne de la matrice B : ")
    Pour j allant de 1 à n faire
      Lire(B[i][j])
    FinPour
  FinPour
  Pour i allant de 0 à n-1 faire
    Pour j allant de 0 à n-1 faire
      S ← 0
      Pour k allant de 0 à n-1 faire
        S ← S + A [i] [k] × B [k] [j]
      FinPour
      C [i] [j] ← S
    FinPour
  FinPour
  Pour i allant de 0 à n-1 faire
    Pour j allant de 0 à n-1 faire
      Écrire(C[i][j], " ")
    FinPour
  FinPour
Fin

```

2.3 Enregistrement

Les langages de programmation offrent, en plus des types de base (entier, réel, booléen...), d'autres types de données appelés **enregistrements**.

Un enregistrement est un regroupement de données qui doivent être considérés ensemble

Exemple : les fiches d'étudiants. Chaque fiche est caractérisée par : un nom et prénom, numéro d'inscription, ensemble de notes...

En pseudo-code :

```

enregistrement FicheEtudiant
début
  nom,prenom : chaîne de caractères
  numero : entier
  notes : tableau[10] de réels
fin

```

Un enregistrement est un type comme les autres types.

Ainsi la déclaration suivante :

|| f, g : FicheEtudiant définit deux variables f et g enregistrements de type FicheEtudiant

L'enregistrement FicheEtudiant contient plusieurs parties (champs), on y accède par leur nom précédé d'un point "." :

- **f.nom** désigne le champ (de type chaîne) nom de la fiche f
- **f.notes[i]** désigne le champ (de type réel) notes[i] de la fiche f

Pour définir les champs d'un enregistrement, on écrit :

```

f : FicheEtudiant
f.nom ← "XXXXXX"
f.prenom ← "YYYYYY"
f.numero ← 1256
f.notes[2] ← 12,5

```

Les affectations entre enregistrement se font champ par champ

2.3.1 Utilisation des enregistrements

```

Procédure affiche(FicheEtudiant v)
Début
    Écrire("No :", v.numero, "-", v.prenom)
    Pour i allant de 0 à 10 faire
        Écrire(v.notes[i], " ")
    FinPour
FinProcédure

```

Exemple : Addition de deux nombres complexes

```

enregistrement Complexe
    début
        Re : réel
        Im : réel
    fin

Fonction add( z1, z2 :Complexe par valeur) : Complexe
Début
    Variable z : Complexe
    z.Re = z1.Re + z2.Re
    z.Im = z1.Im + z2.Im
    retourne(z)
FinFonction

```

Chapitre 3

Historique et présentation du langage

Ce chapitre essaye de placer le langage C dans son contexte historique et technique, de manière à approcher l'état d'esprit du langage qui nous permet de deviner les règles sous-jacentes au langage sans pour cela avoir à les mémoriser.

Les codes sources ont été testés avec l'outil Code : :Blocks 5.11 en mode « console ».

3.1 Historique du langage C

Le langage C a été créé par Dennis Ritchie aux Bell Telephone Laboratories en 1972. Il a été conçu dans un dessein bien précis : développer le système d'exploitation UNIX, déjà utilisé sur de nombreux ordinateurs. Dès l'origine, il devait donc permettre aux programmeurs de travailler de manière productive et efficace.

En raison de sa puissance et de sa souplesse, l'utilisation du C s'est rapidement répandue au-delà des laboratoires Bell. Les programmeurs ont commencé à l'utiliser pour écrire toutes sortes de programmes. Rapidement, des organisations diverses ont utilisé leurs propres versions du langage C, et de subtiles différences d'implémentation sont devenues un véritable casse-tête pour les programmeurs. En réponse à ce problème, l'American National Standards Institute (ANSI) a formé un comité en 1983 pour établir une définition standard du C, qui est devenu le C standard ANSI. À quelques exceptions près, les compilateurs C d'aujourd'hui adhèrent à ce standard.

Le nom du langage C vient de son prédécesseur qui était appelé B. Le langage B a été développé par Ken Thompson qui travaillait aussi aux laboratoires Bell.

Le langage C a peu évolué. Depuis les années 1970 sont sorties les normes C89, largement supportée par les compilateurs, et C99, plus récente. Mais, depuis dix ans, le monde a changé. D'autres normes sont sorties. D'autres besoins sont apparus. Des outils et des bibliothèques de fonctions ont été écrits.

Les normes C89 et C99 sont assez proches de la norme ANSI d'origine. Ces normes cadrent le langage C. Mais il fallait également normer les systèmes d'exploitation. En effet, cela peut sembler agréable d'entendre dire que le langage C est très portable et existe sur la plupart des plates-formes. Mais à quoi bon cette portabilité si vos programmes fonctionnent de façon différente sur chacune d'elles. Pour cela, une autre norme a été créée pour les systèmes d'exploitation. Il s'agit de la norme POSIX. La plupart des systèmes d'exploitation grand public respectent cette norme. C'est le cas entre autres de Windows NT et Vista (à condition d'activer certaines fonctionnalités optionnelles), de GNU/Linux et de Mac OS X. Il existe également d'autres normes comme BSD (BSD4.3 ou BSD4.4, par exemple), SVr4 (System V Release 4). Ces normes sont également répandues et vous pouvez vous y fier.

En résumé, les dates marquantes de l'histoire du langage C sont les suivantes :

- 1970 diffusion de la famille PDP 11
- 1971 début du travail sur le langage C, car le PDP 11 peut manipuler un octet alors que son mot mémoire est de 2 octets, il est nécessaire pour utiliser les fonctionnalités du PDP11 introduire un type de donnée **char** et un type **int**. Ces notions de type n'étant pas prises en compte par le langage B, elles le seront dans son successeur le C.

- 1972 la première version de C est écrite en assembleur par Brian W. Kernighan et Dennis M. Ritchie.
- 1973 Alan Snyder écrit un compilateur C portable (thèse MIT).
- 1975 Steve C. Johnson écrit et présente le PCC (Portable C Compiler). C'était à l'époque le compilateur le plus répandu. Le PCC a longtemps assuré sa propre norme puisqu'il était plus simple de porter le PCC que de réécrire un compilateur C (25
- 1987 début de la normalisation du C par l'IEEE avec constitution d'un comité appelé X3 J-11.
- 1989 sortie du premier document normalisé appelé norme ANSI X3-159.
- 1990 réalisation du document final normalisé auprès de l'ISO : ISO/IEC 9899 ;
- 1999 première révision de la norme ISO/IEC 9899

3.2 Présentation du langage

Le langage C est un langage de bas niveau dans le sens où il permet l'accès à des données que manipulent les ordinateurs (bits, octets, adresses) et qui ne sont pas souvent disponibles à partir de langages évolués tels que Fortran, Pascal ou ADA.

Le langage C a été conçu pour l'écriture de systèmes d'exploitation et du logiciel de base. Plus de 90 du noyau du système UNIX est écrit en langage C. Le compilateur C lui-même est écrit en grande partie en langage C ou à partir d'outils générant du langage C. Il en est de même pour les autres outils de la chaîne de compilation (assembleur, éditeur de liens, pré-processeur). De plus, tous les utilitaires du système sont écrits en C (shell, outils).

Il est cependant suffisamment général pour permettre de développer des applications variées de type scientifique ou encore pour l'accès aux bases de données. Par le biais des bases de données, il est utilisé dans les applications de gestion. De nombreux logiciels du domaine des ordinateurs personnels, tels que Microsoft Word ou Microsoft Excel, sont eux-aussi écrits à partir de langage C ou de son successeur orienté objet : C++.

Bien que pouvant être considéré de bas niveau, le langage C supporte les structures de base nécessaires à la conception des applications structurées. Cette caractéristique le range dans la catégorie des langages de haut niveau. Il est aussi un des premiers langages offrant des possibilités de programmation modulaire, c'est-à-dire qu'un programme de langage C peut être constitué de plusieurs modules. Chaque module est un fichier source que l'on peut compiler de manière autonome pour obtenir un fichier objet. L'ensemble des fichiers objets participants à un programme doivent être associés pour constituer un fichier exécutable.

Lorsque nous parlons du langage C, dans cette première partie, nous faisons référence à ce que sait faire le compilateur lui-même. Comme nous le verrons plus loin dans ce cours, plusieurs outils interviennent dans la transformation d'un ensemble de fichiers sources, constituant un programme, en un fichier binaire exécutable, qui est le résultat de ce que l'on appelle communément la compilation. Il serait plus juste de dire les compilations suivies par les traductions d'assembleur en objet, suivies de la réunion des fichiers objets.

3.3 Pourquoi utiliser le langage C ?

Il existe de nombreux langages de programmation de haut niveau comme le C, le Pascal, ou le Basic. Ils sont tous excellents et conviennent pour la plupart des tâches de programmation. Toutefois, les professionnels placent le langage C en tête de liste pour plusieurs raisons :

- Il est souple et puissant. Ce que vous pourrez accomplir avec ce langage n'est limité que par votre imagination. Vous n'aurez aucune contrainte. Le langage C est utilisé pour des projets aussi variés que des systèmes d'exploitation, des traitements de textes, des graphiques, des tableurs ou même des compilateurs pour d'autres langages.

- Lorsqu'une nouvelle architecture (nouveau processeur, nouveau système d'exploitation...) apparaît, le premier langage disponible est généralement le C car contrairement à d'autres, il est facile à porter. De plus, un compilateur C est souvent disponible sur les ordinateurs (à l'exception de Windows malheureusement), ce qui n'est pas le cas pour les autres langages.
- Avec la norme ANSI, le C est devenu un langage portable. Cela signifie qu'un programme C écrit pour un type d'ordinateur (un PC IBM, par exemple) peut être compilé pour tourner sur un autre système (comme un DEC VAX) avec très peu ou aucune modification.
- Le langage C contient peu de mots. Une poignée d'expressions appelées mots clés servent de bases pour l'élaboration des fonctions. On pourrait penser, à tort, qu'un langage possédant plus de mots clés (quelquefois appelés mots réservés) pourrait être plus puissant. Lorsque vous programmerez avec ce langage, vous vous apercevrez que vous pouvez réaliser n'importe quelle tâche.
- Le langage C est modulaire. Son code peut (et devrait) être écrit sous forme de sous programmes appelés fonctions. Ces fonctions peuvent être réutilisées pour d'autres applications ou programmes. Si vous passez des informations à ces fonctions, vous obtenez du code réutilisable.

Comme vous pouvez le constater, le choix du C en tant que premier langage de programmation est excellent. Vous avez certainement entendu parler de C++. Ce langage s'appuie sur une technique de programmation appelée programmation orientée objet.

Un autre langage, également basé sur C, a été l'objet d'une attention toute particulière. Il s'agit de Java. Si vous décidez de vous orienter vers la programmation Java, vous découvrirez rapidement qu'il existe de nombreuses similitudes entre ces deux langages.

Chapitre 4

Comment démarrer ?

Vous apprendrez dans ce chapitre les étapes du cycle de développement d'un programme.

4.1 Notion des langages informatiques

Un langage informatique est un outil permettant de donner des ordres (instructions) à la machine : à chaque instruction correspond une action du processeur.

L'intérêt des langages informatiques est écrire des programmes (suite consécutive d'instructions) destinés à effectuer une tâche donnée. Exemple : un programme de gestion de comptes bancaires

Ces langages informatiques doivent être compréhensibles par la machine.

Parmi les langages informatiques

4.2 Avant de programmer

Vous ne pouvez résoudre que les problèmes que vous aurez identifiés. Il sera alors possible de bâtir un plan pour les corriger. Lorsque vous aurez appliqué ce plan, vous devrez tester les résultats pour savoir si les problèmes ont bien été résolus. Cette logique s'applique à de nombreux domaines, la programmation en fait partie.

Voici les étapes à suivre pour créer un programme en langage C (ou dans n'importe quel autre langage) :

1. Définir les objectifs du programme.
2. Choisir les méthodes que vous voulez utiliser pour écrire ce programme
3. Créer le programme.
4. Enfin, l'exécuter et observer les résultats.

Par exemple, imaginez que quelqu'un vous demande d'écrire un programme pour calculer l'aire d'un cercle. L'étape 1 est réalisée puisque vous connaissez votre objectif : trouver la valeur de cette aire. L'étape 2 consiste à déterminer quelles sont les données à connaître pour le calcul. Si l'utilisateur du programme donne le rayon du cercle, la formule $3,14E38 \cdot r^2$ vous donnera la réponse. Vous pouvez maintenant passer aux étapes 3 et 4 qui constituent le développement du programme.

4.3 Cycle de développement du programme

La première étape du développement d'un programme est la création du code source avec un éditeur. La deuxième étape consiste à compiler ce code pour obtenir un fichier objet. Dans la troisième, vous transformez le code compilé en fichier exécutable. Le lancement du programme dans la quatrième étape permet d'en vérifier les résultats.

Ce cycle peut-être schématisé comme suit :

1. Utilisez un éditeur pour créer le code source. Par convention, ce fichier doit avoir l'extension `.c` (par exemple, `monprog.c`, `database.c`, etc.).
2. Compilez votre programme. Si le compilateur ne rencontre pas d'erreur dans votre code source, vous obtenez un fichier objet du même nom que votre fichier source avec une extension `.o` (par exemple, `monprog.c` est compilé en `monprog.o`). Si le code source contient des erreurs, le compilateur échoue et vous les affiche pour correction.
3. Exécutez la liaison. Si aucune erreur n'apparaît, vous obtenez un programme exécutable dans un fichier du même nom que le fichier objet (avec une extension `.exe` sur Windows par exemple, `monprog.o` devient `monprog.exe`).
4. Exécutez votre programme. Contrôlez les résultats obtenus et recommencez à l'étape 1 si des modifications sont nécessaires dans le fichier source.

Les étapes de développement du programme sont représentées dans la Figure 4.1.

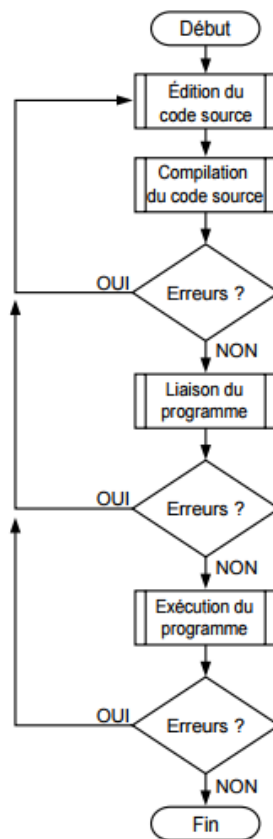


FIGURE 4.1 – Les étapes de développement d'un programme C.

Chapitre 5

Éléments de langage C

Un programme C est constitué de modules distincts, nommé Fonction , qui peuvent s'appeler mutuellement. Chaque fonction effectue une tâche donnée. Des fonctions prêtes à l'emploi sont fournies par la bibliothèque standard, à charge le programmeur d'écrire les autres fonctions dont il a besoin. Un nom particulier de fonction : main, désigne la première fonction à être appelé au démarrage du programme. Tous les autres programmes sont des sous-programmes de celles-ci.

5.1 Structure d'un programme C

5.1.1 Exemple de programme

```
/* multiplier. c : Calcul du produit de deux nombres. */
# include <stdio .h>

/* La fonction renvoie le produit de ses deux arguments */
int produit( int x, int y){
    return ( x * y);
}

int main (){
    int a,b, c;

    /* Lecture du premier nombre */
    printf(" Entrez un nombre entre 1 et 100 : "); scanf("% d", & a);
    /* Lecture du deuxième nombre */
    printf(" Entrez un autre nombre entre 1 et 100 : "); scanf("% d", &
        b);
    /* Calcul du produit et affichage du résultat */
    c = produit(a, b);
    printf (" \ n% d fois % d = % d", a, b, c);

    return 0;
}
```

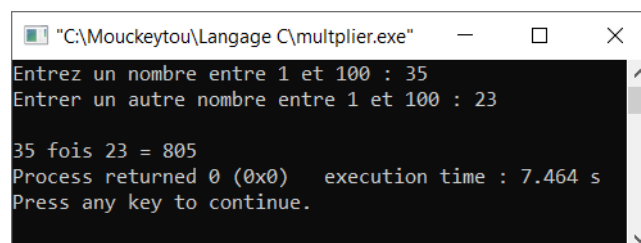


FIGURE 5.1 – Exécution de multiplier.exe

5.1.2 Structure du programme

Nous allons examiner le programme précédent ligne par ligne pour en isoler les différents composants.

5.1.2.1 La fonction `main()`

La fonction `main()` est le seul bloc obligatoire d'un programme C. Sa forme la plus simple consiste à saisir son nom, `main`, suivi de parenthèses `()` vides et d'une paire d'accolades. Celles-ci renferment la partie principale du programme. L'exécution du programme débute à la première instruction de `main()` et se termine avec la dernière instruction de cette fonction.

5.1.2.2 Appel d'un fichier `#include`

L'instruction d'appel `#include`, indique au compilateur C qu'il doit inclure le contenu d'un fichier dans le programme pendant la compilation. Ce fichier inclus (aussi appelé fichier en-tête) contient des informations destinées à votre programme ou au compilateur. Plusieurs de ces fichiers ont été livrés avec votre compilateur, vous ne devez pas en modifier les informations. Ils ont tous une extension `.h` (par exemple, `stdio.h`).

Dans notre exemple, l'instruction d'appel `#include` signifie "ajouter le contenu du fichier `stdio.h`".

5.1.2.3 Définition de variable

Une *variable* est un nom donné à une zone mémoire. En effet, votre programme a besoin de mémoire pour stocker ses données en cours d'exécution. En C, une variable doit être définie avant d'être utilisée. La définition de variable indique son nom au compilateur et le type de données que l'on pourra y stocker. Dans notre exemple, `int a, b, c;`, définit trois variables appelées `a`, `b`, et `c` qui contiendront chacune une valeur entière. Les variables et constantes numériques sont traitées respectivement au Chapitres 4 et 5 et de manière plus approfondie au Chapitre 15.

5.1.2.4 Déclaration de fonction

La *déclaration de fonction* indique au compilateur C le nom et les arguments d'une fonction qui sont utilisés dans le programme. Cette déclaration doit apparaître avant l'utilisation de la fonction et ne doit pas être confondue avec la définition de fonction qui contient les instructions propres à cette fonction. Cette déclaration est facultative si la fonction peut être définie avant tout appel à elle.

5.1.2.5 Instructions

Les instructions constituent le travail réalisé par le programme. Elles affichent les informations sur l'écran, lisent les données saisies au clavier, effectuent les opérations mathématiques, appellent les fonctions, lisent les fichiers et accomplissent tous types d'opérations nécessaires à un programme. Chaque instruction occupe généralement une ligne et se termine par un point-virgule. Les directives au préprocesseur font toutefois exception à cette règle.

L'instruction `return` termine l'exécution d'une fonction et retourne le contrôle à la fonction appelante. L'exécution reprend dans la fonction d'appel au point immédiatement après l'appel. Une instruction `return` peut également retourner une valeur à la fonction appelante.

5.1.2.6 Définition de fonction

Une *Fonction* est une portion de code indépendante qui a été écrite pour effectuer une certaine tâche. On appelle cette fonction dans un programme en introduisant son nom dans une instruction.

La fonction `produit()` est une *Fonction utilisateur*. Comme son nom l'indique, une fonction utilisateur est écrite par le programmeur pendant le développement de son programme.

En réalité, vous n'avez pas besoin de créer une fonction pour une tâche aussi simple que la multiplication de deux nombres. Nous l'avons fait pour vous donner un exemple.

Le langage C possède de multiples fonctions de bibliothèques qui sont fournies avec le compilateur. Ces fonctions réalisent la plupart des tâches de base (comme les entrées/ sorties de l'écran, du clavier, et du disque) dont votre programme a besoin. Dans notre exemple, `printf()` et `scanf()` sont des fonctions de bibliothèque.

`printf()` est une fonction de bibliothèque qui envoie des informations à l'écran. Elle peut afficher un message texte simple ou un message accompagné de variables issues du programme.

`scanf()` est une autre fonction de bibliothèque. Elle lit les données entrées au clavier et les attribue à des variables du programme.

5.1.2.7 Commentaires du programme

La partie de code du programme qui commence par `/*` et qui se termine par `*/` est un commentaire. Le compilateur l'ignore. Vous pouvez le placer n'importe où, il n'a aucune influence sur le déroulement du programme.

Vous ne devez pas imbriquer des commentaires, cela provoque une erreur avec beaucoup de compilateurs

5.1.2.8 Accolades

Les accolades (`{ }`) permettent d'encapsuler les lignes de programmes qui constituent chaque fonction C. On appelle *bloc* l'ensemble des instructions qui se trouvent entre ces accolades

5.1.3 Remarques

Un ordinateur est précis et rapide, mais il ne fait qu'exécuter des ordres. Il est parfaitement incapable de corriger la moindre erreur.

Cela est valable pour votre code source C. Le compilateur échouera à la moindre faute de frappe. Heureusement, même s'il ne peut pas corriger vos erreurs, il sait les reconnaître pour vous les indiquer.

5.1.4 Pour aller plus loin

Les instructions qui constituent les fonctions, les déclarations et les directives au préprocesseur forment le code source d'un programme C. Pour les petits programmes, le code source est écrit dans un seul fichier source. Les programmes C de grande taille font appel à plusieurs fichiers source qui peuvent être modifiés et compilés séparément. Les informations requises dans plusieurs fichiers source, comme les déclarations, sont placées dans les fichiers d'en-tête. Ceux-ci peuvent être inclus dans chaque fichier source via la directive `#include`. Le nom des fichiers source se termine par `.c`, celui des fichiers d'en-tête par `.h`. Une fonction ne peut-être définie à l'intérieur d'une autre fonction. Les directives au préprocesseur sont des commandes que le préprocesseur doit exécuter avant que le programme lui-même soit compilé, et chacune d'entre elles commence par un caractère dièse `#`.

5.2 Jeux de caractères

Deux jeux de caractères sont définis dans le cadre du C ANSI. Le premier, appelé jeux de caractères source, correspond à l'ensemble des caractères que l'on peut utiliser dans un fichier source. le second, le jeu de caractères d'exécution, est formé de tous les caractères interprétés lors de l'exécution du programme, par exemple les caractères d'une constante chaîne. Chacun de ces ensembles contient un jeu de caractères de base qui comprend :

- Les 52 lettres(minuscules et majuscules) de l'alphabet latin :
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
- Les dix chiffres de base du système décimal : 0 1 2 3 4 5 6 7 8 9

- Les 26 caractères graphiques suivants : ! " #
- Les 5 caractères d'espacement : espace, tabulation horizontale, tabulation verticale, retour-chariot, saut de page

Le jeu de caractères d'exécution de base comprend en plus :

- Le caractère *NUL* noté \0 qui termine les chaînes de caractères ;
- Les caractères de contrôle, représentés par les séquences d'échappement standard du tableau 5.1, qui permettent de gérer les périphériques de sortie (terminaux, imprimantes, etc.).

Séquence d'échappement	Effet sur le périphérique d'affichage
\a	Signal sonore (beep)
\b	Retour en arrière
\f	Saut de page
\n	Fin de ligne
\r	Retour chariot
\t	Tabulation horizontale
\v	Tabulation verticale
\'	Le caractère '
\"	Le caractère "
\?	Le caractère ?
\\	Le caractère \
\o \oo \ooo (o=chiffre octal)	Le caractère correspondant à ce code octal
\xh.. h.. chaîne de chiffres hexadécimaux	Le caractère correspondant à ce code hexadecimal

TABLE 5.1 – Les séquences d'échappement standard

Tout autre caractère – mais cela dépend de votre compilateur, peut être utilisé dans les commentaires, les chaînes et les constantes caractères (le caractère dollar ou les caractères accentués, par exemple. Notez que l'utilisation de tels caractères peut nuire à la portabilité de vos programmes.

L'encodage Unicode, qui permet de passer du code ASCII standard (norme informatique de codage de caractères apparue dans les années 1960) à un codage ou près de 35000 caractères provenant de 24 pays peuvent être représentés, est maintenant très souvent utilisé.

5.3 Identificateurs

Le terme *identificateur* désigne le nom des variables, des fonctions, des macro-instructions, des types, etc. Les identificateurs obéissent aux règles suivantes :

- Un identificateur est formé d'une suite de lettres (A à Z, a à z) ; de chiffres (0 à 9) ou de caractères de soulignement (_)
- Le premier caractère d'un identificateur ne peut pas être un chiffre. Le caractère () est aussi autorisé, mais il n'est pas recommandé ;
- Les identificateurs sont sensibles à la case ou plus simplement les lettres majuscules sont différentes des minuscules. Par exemple, `compte` et `COMPTE` ne représentent pas le même identificateur ;
- Il n'y a pas de limite à la longueur d'un identificateur. Toutefois, seuls les 31 premiers caractères sont en général significatifs.

Les mots-clés sont réservés et ne doivent pas être utilisés comme identificateurs. Voici une liste non exhaustive de mots-clés :

auto	enum	restrict	unsigned	break	extern	return	void	case
float	short	char	volatile	char	for	continue	signed	while
const	goto	sizeof	_Bool	union	if	static	int	default
inline	struct	do	switch	double	long	typedef	else	

Les *noms externes*, c'est-à-dire les identificateurs de fonctions et de variables externes liées peuvent faire l'objet d'autres restrictions qui dépendent de l'éditeur de liens.

Voici quelques exemples d'identificateurs :

- valides : a, DM, dm, FLOAT, __var1, HautDeFenetre ;
- non valides : do, 586cpu, z

5.4 Catégories et portée des identificateurs

Tout identificateur appartient à l'une des quatre catégories suivantes :

- Les noms d'étiquettes ;
- Les *marqueurs* de structure, d'union et d'énumération. Ce sont des identificateurs placés à la suite des mots-clés **struct** , **union** et **enum** .

struct étiquette identificateur ;

- Les noms des membres d'une *structure* ou *union* possède un espace de noms distinct pour ses membres ;
- Tous les autres identificateurs, appelés *identificateurs ordinaires*

Un nom d'étiquette peut aussi, par exemple, être utilisé comme nom de fonction. Un tel réemploi est fréquent dans les structures ou la même chaîne peut désigner un type *structure*, l'un de ses membres est une variable.

```
|| struct person { char * person ; /*....*/ } personne ;
```

Chaque identificateur dans le code source possède une certaine portée. Cette portée correspond à la portion de programme dans laquelle l'identificateur peut être employé. Les quatre portées possibles sont :

- *Prototype d'une fonction* Les identificateurs contenus dans la liste des déclarations de paramètres du prototype d'une fonction (pas dans sa définition) ont une portée confinée à ce prototype ;
- *Fonction*
Seuls les noms d'étiquettes ont comme portée le corps d'une fonction. Leur emploi est limité au bloc de la fonction dans lequel l'étiquette est définie. Les noms d'étiquettes doivent être uniques à l'intérieur de la fonction. L'instruction goto provoque un saut vers l'instruction qui suit une étiquette à l'intérieur de la fonction courante ;
- *Bloc*
les identificateurs, autres que les noms d'étiquettes, déclarés à l'intérieur d'un bloc ont une portée confinée à ce bloc. Les paramètres de la définition d'une fonction ont aussi une portée limitée au bloc. À l'intérieur d'un bloc, la portée d'un identificateur commence avec sa définition et se termine à l'accolade fermante (}) du bloc ;
- *Fichier*
Les identificateurs déclarés à l'extérieur de tout bloc ou liste de paramètre ont comme portée le fichier lui-même. Leur portée commence à leur définition et se prolonge jusqu'à la fin du fichier source.

Chapitre 6

Types élémentaires

Le type d'une variable détermine l'espace qu'elle occupe en mémoire et la façon dont sera interprétée la séquence de bits stockée. De même, le type d'une fonction détermine la façon dont sera interprétée sa valeur de retour.

Les types peuvent être soit prédéfinis, soit dérivés. Les prédéfinis du C sont les types de base et le type `void`. Les types de base sont constitués :

- des types entiers ;
- des types flottants ;
- des types caractères ;
- et des types chaîne de caractères.

6.1 Types entiers

Il existe en C plusieurs types de variable entière. Leurs différences s'expliquent par le fait que des entiers selon leur taille ont des besoins de mémoire différents et que les opérations mathématiques ne s'effectuent pas de la même façon selon le type de variables. Les petits entiers (par exemple 1, 199 et 8) demandent peu d'espace mémoire et les opérations mathématiques sont réalisées très rapidement par votre ordinateur. Les grands nombres et les valeurs en virgule flottante (123 000 000 ou 0,000000871256, par exemple), au contraire, nécessitent plus d'espace mémoire et les opérations mathématiques sont plus longues.

On dénombre 5 types d'entier signés :

- `signed char`,
- `short int` (ou `short`),
- `int`,
- `long int`(ou `long`), et `long long int`(C99) (ou `long long`(C99))

À chacun de ces types correspond un type entier non signé occupant la même place mémoire. Le type non signé est désigné par le préfixe `unsigned` devant le spécificateur de type, `unsigned int`, par exemple.

D'un point de vue formel, les types `char`, `signed char` et `unsigned char` sont différents. Toutefois, selon le paramétrage du compilateur, `char` est équivalent à `signed` ou à `unsigned char`. le préfixe `signed` n'a aucun sens pour les types `short`, `int`, `long` et `long long` puisque ils sont toujours signés. Ainsi, `short` et `signed short` désignent le même type.

Les valeurs extrêmes de ces entiers, pour une implémentation donnée, sont définies dans le fichier en-tête `limits.h`

Le type `_Bool` a été ajouté dans ANSI C99 pour représenter les valeurs booléennes. la valeur booléenne `true` est représentée par `1`, et `false` par `0`. Si le fichier en-tête `stdbool.h` a été inclus, `bool`, peut être utilisé pour `_Bool` et les macros `true` et `false` respectivement pour `1` et `0`. Vous trouverez dans le

tableau 6.1 les types entiers standard et leurs intervalles de valeurs usuels.

Type	Taille en mémoire	Intervalle de valeur (décimal)
<code>_Bool</code>	1 octet	0 et 1
<code>char</code>	1 octet	−128 à 127 ou 0 à 255
<code>unsigned char</code>	1 octet	0 à 255
<code>signed char</code>	1 octet	−128 à 127
<code>int</code>	2 ou 4 octets	−32 768 à 32 767 ou −2 147 483 648 à 2 147 483 647
<code>unsigned int</code>	2 ou 4 octets	0 à 65 535 ou 0 à 4 294 967 295
<code>short</code>	2 octets	−32 768 à 32 767
<code>unsigned short</code>	2 octets	0 à 65 535
<code>long</code>	4 octets	−2 147 483 648 à 2 147 483 647
<code>unsigned long</code>	4 octets	0 à 4 294 967 295
<code>long long(C99)</code>	8 octets	−9 223 372 036 854 775 808 à 9 223 372 036 854 775 807
<code>unsigned long long(C99)</code>	8 octets	0 à 18 446 744 073 709 551 615

TABLE 6.1 – Les types entiers standard

Les types entiers à taille définie ont été introduits dans ANSI C99 par l'intermédiaire du fichier d'en-tête `stdint.h`. La taille N d'un entier correspond au nombre de bits utilisés pour représenter les valeurs dans ce type, bit de signe y compris. (En général $N = 8, 16, 32$ ou 64)

Remarque : En langage C, le type `char` est un cas particulier du type entier

6.1.1 Déclarations de variables

Avant d'utiliser une variable dans un programme C, il faut la déclarer. Cette déclaration indiquera au compilateur le nom et le type de la variable et elle pourra l'initialiser à une certaine valeur. Si votre programme utilise une variable qui n'a pas été déclarée, le compilateur génère un message d'erreur. Une déclaration de variable a la forme suivante :

```
typename varname;
```

Vous pouvez déclarer plusieurs variables du même type sur une seule ligne en les séparant par des virgules.

```
|| int count, number, start;    /* trois variables entières */
```

6.1.2 Mot-clé typedef

Le mot-clé `typedef` permet de créer un synonyme pour un type de donnée existant. Par exemple :

```
|| typedef int entier ;
```

créé le synonyme `entier` pour `int`. Vous pourrez ainsi utiliser `entier` pour définir des variables de type `int`, comme dans l'exemple suivant :

```
|| entier compte ;
```

`typedef` ne crée pas un nouveau type de donnée, il permet seulement d'utiliser un nom différent pour un type de donnée déjà définie. L'usage le plus fréquent de `typedef` concerne les données agrégées qui sont expliquées au Chapitre 15.

6.1.3 Initialisation des variables entières

La déclaration de variable permet au compilateur de réserver l'espace mémoire destiné à cette variable. La donnée qui sera stockée dans cet emplacement, la valeur de la variable, n'est pas encore définie. Avant d'être utilisée, la variable déclarée doit être initialisée. Cela peut se faire en utilisant une instruction d'initialisation comme dans notre exemple :

```
|| int count;    /* Réserve de la mémoire pour count */
|| count = 0;    /* Stocke 0 dans count */
```

En programmation, ce signe = n'a pas le même sens qu'en algèbre. Par exemple, l'instruction "**x** = 2" signifie "donner la valeur 12 à la variable appelée x".

Vous pouvez initialiser une variable au moment de sa déclaration. Il suffit de faire suivre le nom de variable, dans l'instruction de déclaration, du signe égal suivi de la valeur initiale :

```
|| int count = 0;
|| double percent = 0.01 , taxrate = 28.5;
```

Attention, il ne faut pas initialiser la variable avec une valeur qui ne correspond pas au type déclaré. Voici un exemple d'initialisation incorrecte :

```
|| unsigned int valeur = -2500;
```

Le compilateur ne détecte pas ce genre d'erreur, et le programme pourrait vous donner des résultats surprenants.

6.2 Types flottants ou réels

Trois types sont définis pour représenter les nombres réels : **float**, **double**, **long double**. En C, ces trois types sont appelés *types réels flottants*.

La taille en mémoire et la représentation interne de ces types n'est pas spécifiée en C standard et peut varier selon le compilateur. Cependant, la plupart des compilateurs suivent la norme IEE 754-1985 pour l'arithmétique binaire en virgule flottante. le tableau 3 suit cette norme. * Valeur approximative :

Type	Taille en mémoire	Intervalle de valeur (décimal)
float	4 octets	1,2 E – 38 à 3,4 E38*
double	8 octets	2,3 E – 308 à 1,7 E308**
long double	10 octets	3,4 E – 4932 à 1,1 E4932***

TABLE 6.2 – Les séquences d'échappement standard

précision = 6 chiffres

** Valeur approximative : précision = 15 chiffres

*** Valeur approximative : précision = 19 chiffres

Valeur approximative signifie la plus haute et la plus petite valeur qu'une variable donnée puisse recevoir. Précision indique la précision avec laquelle la variable est stockée (par exemple, pour évaluer $\frac{1}{3}$, la réponse est 0,333 avec des 3 à l'infini ; une variable avec une précision de 7 s'écrit avec sept chiffres 3 après la virgule)

La représentation d'un nombre en virgule flottante consiste toujours en un *signe* s, une *mantisse* m, et un *exposant* exp puissance de 2.

$$s = s \cdot m \cdot 2^{\text{exp}} \quad \text{avec} \quad 1, 0 \leq m \leq 2 \quad \text{ou} \quad m = 0$$

C'est le nombre de bits utilisés pour stocker la mantisse qui détermine la précision d'un type flottant. L'intervalle de valeur est déterminé par le nombre de bits utilisés pour l'exposant.

6.3 Types caractères

Le type `char` (provenant de l'anglais *character*) permet de stocker la valeur ASCII d'un caractère, c'est-à-dire un nombre entier !

Par défaut les nombres sont signés, cela signifie qu'ils comportent un signe. Pour stocker l'information concernant le signe (en binaire), les ordinateurs utilisent le complément à deux. Une donnée de type `char` est donc signée, cela ne signifie bien sûr pas que la lettre possède un signe mais tout simplement que dans la mémoire la valeur codant le caractère peut être négative...

Le type `char` permet de stocker des nombres compris entre -128 et 127. Si ce type `char` permet de stocker des nombres, il faut savoir qu'en C on l'utilise rarement pour ça. En général, même si le nombre est petit, on le stocke dans un `int`.

- Si une variable `char` est utilisée à un endroit du programme où un caractère est attendu, elle sera interprétée comme un caractère.
- Si une variable `char` est utilisée à un endroit du programme où un nombre est attendu, elle sera interprétée comme un nombre

Le site Asciitable.com est célèbre pour proposer cette table mais ce n'est pas le seul, on peut aussi la retrouver sur Wikipédia et bien d'autres sites encore.

Il n'existe pas de type de données pour les chaînes de caractères (suite de caractères) en langage C. Pour créer une chaîne de caractères on utilisera donc des tableaux contenant dans chacune de ses cases un caractère...

Comme toutes les autres variables, une variable `char` doit être déclarée, et elle peut être initialisée dans la même instruction :

```
char a, b, c;    /* Déclaration de trois variables char non */
                 /* initialisées */
char code='x';   /* Déclaration d'une variable char appelée code */
                 /* dans laquelle on stocke le caractère x */
code='!';        /* On stocke le caractère ! dans la variable code */
```

Chapitre 7

Constantes

Une constante est un emplacement mémoire utilisé par votre programme. À l'inverse d'une variable, la valeur stockée dans une constante ne peut changer pendant l'exécution du programme. Le langage C possède deux types de constantes qui ont chacune un usage spécifique

7.1 Constantes entières

Les constantes entières sont représentées par des nombres exprimés en notation décimale, octale ou hexadécimale :

- une *constante décimale* (base 10) commence par un chiffre qui n'est pas zéro. Les constantes décimales s'expriment à l'aide des chiffres 0 à 9 accompagnés d'un signe moins ou plus ;
- une *constante octale* (base 8) commence par un zéro. Une constante en octal peut contenir les chiffres 0 à 7 accompagnés du signe moins ou plus.
- une *constante hexadécimale* (base 16) commence par un zéro suivi d'un x(0x ou 0X. Les constantes hexadécimales s'expriment à l'aide des chiffres 0 à 9, des lettres A à F, et du signe moins ou plus. Les chiffres hexadécimaux A, B, D, E et F ne sont pas sensibles à la case.

Ces conventions d'écriture des constantes ne respectent pas l'écriture mathématique, puisque 010 devant être interprété en octal, n'est pas égal à 10.

7.2 Constantes flottantes

Une constante flottante est représentée par une suite de chiffres décimaux avec un point, ou par notation avec mantisse et exposant. La mantisse est composée d'une partie entière suivie du signe . (point) suivi de la partie fractionnaire. La partie entière et la partie fractionnaire sont exprimées en décimal et l'une ou l'autre peuvent être omises.

L'exposant est introduit par la lettre e sous la forme minuscule ou majuscule. L'exposant est un nombre décimal éventuellement signé. La lettre P ou p est utilisée pour représenter une constante flottante avec un exposant puissance de 2 (ANSI C99) ; par exemple :

```
|| 2.7 P+6 /* Le nombre 2.7*2^6 */
```

Afin de distinguer une constante flottante d'une constante entière, le signe . ou la notation avec exposant sont nécessaires. Sauf indication contraire, une constante flottante est du type `double`. Le suffixe `L` ou `l` permet de lui affecter le type `long double`. Une constante suffixée par `f` ou `F` a le type `float`.

Si la valeur résultante ne correspond pas au type, la valeur est arrondie vers une valeur supérieure ou inférieure (le choix dépend de l'implémentation).

Remarque :

Notation C	2.	.3	2.3	2e4	2.e4	.3e4	2.3e4	2.3e-4
Notation mathématique	2	0.3	2,3	$2 \cdot 10^4$	$2 \cdot 10^4$	$0,3 \cdot 10^4$	$2,3 \cdot 10^4$	$2,3 \cdot 10^{-4}$

7.3 Constantes caractères

Une constante caractère s'écrit entourée du signe `'`. La règle générale consiste à écrire le caractère entouré du signe `'` ; par exemple, la constante caractère correspondant au caractère `g` s'écrit `'g'`.

La valeur d'une constante caractère contenant plus d'un caractère dépend de l'implémentation. Dans un souci de portabilité, les constantes caractères à plus d'un caractère devraient être évitées.

Les séquences d'échappement comme `'\n'` peuvent être employées dans les constantes caractères. Les caractères `'` et `\` sont représentés de cette façon. Les cas particuliers sont traités par une séquence d'échappement introduite par le caractère `\`.

- Caractères ne disposant pas de représentation imprimable.
 1. On peut les désigner par la notation `'\nb'` où `nb` est le code octal du caractère.

Exemple :

Constante caractère	<code>'\0'</code>	<code>'\12'</code>	<code>'\15'</code>	<code>'\33'</code>
Sémantique	nul	nouvelle ligne	retour chariot	échappement

2. On peut les désigner par la notation `'\xnb'` où `xnb` est le code en hexadécimal du caractère.

Exemple :

Constante caractère	<code>'\x0A'</code>	<code>'\x0D'</code>	<code>'\x1B'</code>
Sémantique	nouvelle ligne	retourner	échappement

3. Certains d'entre eux, utilisés très fréquemment, disposent d'une notation particulière. Il s'agit des caractères suivants :

Exemple :

Constante caractère	<code>'\n'</code>	<code>'\t'</code>	<code>'\v'</code>
Sémantique	nouvelle ligne	tabulation horizontale	tabulation verticale

Constante caractère	<code>'\b'</code>	<code>'\r'</code>	<code>'\f'</code>	<code>'\a'</code>
Sémantique	retour en arrière	retour chariot	saut de page	signal sonore

- Caractères disposant d'une représentation imprimable mais devant être désignés par une séquence d'échappement.

Constante caractère	<code>'\"'</code>	<code>'\\'</code>
Sémantique	<code>"</code>	<code>\</code>

- Caractères disposant d'une représentation imprimable et pouvant être désignés soit par une séquence d'échappement soit par eux-mêmes.

Constante caractère	<code>'\"'</code> ou <code>'\"'</code>	<code>'\?'</code> ou <code>'?'</code>
Sémantique	<code>"</code>	<code>?</code>

7.4 Constantes littérales et Constantes symboliques

7.4.1 Constantes littérales

Une *constante littérale* est une valeur qui est introduite directement dans le code source.

Voici deux exemples :

```
|| int count = 20;  
|| float tax_rate = 0.28;
```

20 et 0.28 sont des constantes littérales. Ces deux instructions stockent ces valeurs dans les variables `count` et `tax_rate`.

7.5 Constantes symboliques

Une constante symbolique est une constante représentée par un nom (symbole) dans votre programme. Comme la constante littérale, cette constante symbolique ne peut changer. Vous utilisez son nom dans le programme chaque fois que vous avez besoin de sa valeur. Cette valeur doit être initialisée une fois au moment de la définition de la variable.

Ces constantes ont deux avantages sur les constantes littérales :

- Rend le code plus facile à lire et à comprendre.

Exemple :

```
|| circonference = 3.14 * 159* (2 * rayon);  
|| aire = 3.14 * (rayon) * (rayon);
```

La première instruction signifie "multiplier par 2 la valeur stockée dans la variable `rayon`, puis multiplier le résultat par 3,14159, enfin, stocker le résultat dans la variable `circonférence`".

Si vous définissez une constante symbolique de nom `PI` et de valeur 3,14, vous pourriez écrire :

```
|| circonference = PI * 159* (2 * rayon);  
|| aire = PI * (rayon) * (rayon);
```

- Le second avantage des constantes symboliques apparaît quand vous avez besoin de changer cette constante. Si vous décidez, dans l'exemple précédent, d'utiliser une valeur de plus précise (3,14159 plutôt que 3,14), vous ne devez changer cette valeur qu'une fois, au niveau de la définition. Avec une constante littérale, vous devez changer chaque occurrence du code source.

L'instruction suivante crée une constante appelée `CONSTNAME` avec la valeur `literal` :

```
|| #define CONSTNAME literal
```

`literal` représente une constante littérale. Par convention, le nom des constantes symboliques s'écrit en lettres majuscules. Cela permet de les distinguer des noms de variables qui sont par convention en lettres minuscules. Dans l'exemple précédent, la commande `#define` aurait été :

```
|| # define PI 3.1415
```

7.5.1 Fonctionnement de `#define`

L'instruction `#define` ne se termine pas par un point-virgule (;). La commande `#define` peut se trouver n'importe où dans le code source, mais son effet est limité à la partie de code qui la suit.

Le rôle de `#define` est d'indiquer au compilateur la directive : "dans le code source, remplacer `CONSTNAME` par `literal`". Bien sur, `#define` ne remplace pas les occurrences qui pourraient se trouver à l'intérieur d'un mot plus long, entre guillemets, ou dans un commentaire du programme. Dans le code suivant, les valeurs de π des deuxième et troisième lignes resteraient identiques :

```
|| #define PI 3.14159  /* vous avez défini la constante PI. */  
|| #define PIPETTE 100
```

7.5.2 Définition des constantes avec le mot-clé const

La seconde méthode pour définir une constante symbolique est d'utiliser le mot-clé `const` qui peut modifier n'importe quelle déclaration de variable. Une variable définie avec ce mot-clé ne peut être modifiée pendant l'exécution du programme. Voici quelques exemples :

```
|| const int count = 100;  
|| const float pi = 3.14159 ;  
|| const long debt = 12000000 , float tax_rate = 0.21;
```

`const` s'applique sur toutes les variables de la ligne de déclaration `debt` et `tax_rate` sont des constantes symboliques. Si votre programme essaie de modifier une variable `const`, le compilateur génère un message d'erreur comme dans l'exemple suivant :

```
|| const int count = 100;  
|| count = 200;  /* Pas de compilation ! On ne peut pas changer */  
||               /* la valeur d'une constante. */
```

Chapitre 8

Expressions et opérateurs

Les programmes C sont constitués d'instructions qui contiennent, pour la plupart, des expressions et des opérateurs

8.1 Expressions

Une expression est une combinaison d'opérateurs et d'opérandes. Dans le cas le plus simple, une expression peut-être constituée simplement d'une constante, d'une variable ou de l'appel à une fonction. Les expressions peuvent aussi faire fonction d'opérande et être associées entre elles par des opérateurs pour créer des expressions plus complexes.

Chaque expression a un type et, si ce type n'est pas `void`, une valeur. Voici quelques exemples d'expressions :

```
printf(" Un exemple !\n n")           // Type : int
1.0 + sin (x)                         // Type : double
srand((unsigned) time(NULL))          // Type : void
(int *) malloc(count*sizeof(int))     // Type : int *
```

8.2 Opérateurs

Un *opérateur* est un symbole qui décrit une opération ou une action à effectuer sur une ou plusieurs *opérandes*. En langage C, les opérandes sont toujours des expressions. Les opérateurs sont divisés en 5 catégories :

- les opérateurs arithmétiques ;
- l'opérateur d'affectation ;
- les opérateurs de comparaison ;
- les opérateurs logiques ;
- les opérateurs bit à bit ;
- les opérateurs d'accès à la mémoire ;
- les autres opérateur.

Dans les expressions comprenant plus d'un opérateur, les règles de priorité des opérateurs déterminent l'ordre d'évaluation. Par exemple, les opérateurs arithmétiques `*`, et `-`. En d'autres termes, dans les expressions arithmétiques, on applique les règles usuelle pour l'ordre des opérations.

Par exemple :

```
|| 4 + 6 * 512 /* est équivalent à 4 + (6*512) */
```

Si l'on désire un autre ordre d'évaluation, on utilisera les parenthèses :

```
|| (4 + 6) * 512
```

Vous trouverez au tableau 8.1 les règles de priorité des opérateurs.

Si deux opérateurs ont la même priorité, c'est l'ordre d'évaluation indiqué dans la colonne « ordre d'évaluation » du tableau 8.1 qui prévaut. Par exemple :

`|| 2 * 5 / 3 // est équivalent à (2*5)/3`

Priorité	Opérateur	Ordre d'évaluation
1	() [] -> .	gauche à droite
2	! ~ ++ -- + - (type) * & sizeof	droite à gauche
3	* / %	gauche à droite
4	+ - &	gauche à droite
5	« »	gauche à droite
6	< <= > >=	gauche à droite
7	== !=	gauche à droite
8	&	gauche à droite
9	^	gauche à droite
10		gauche à droite
11	&&	gauche à droite
12		gauche à droite
13	?:	gauche à droite
14	+= -= *= /= %= gauche à droite &= ≐ = <<= >>=	
15	,	gauche à droite

TABLE 8.1 – Les règles de priorité des opérateurs

Les opérateurs peuvent être *unaires* ou *binaires*. Un opérateur unaire a un seul opérande tandis qu'un opérateur binaire en a deux. Cette distinction est importante pour deux raisons :

- Tous les opérateurs unaires ont la même priorité ;
- Les quatre caractères -, +, * et & représentent un opérateur unaire ou binaire selon le nombre de leurs opérandes.

C possède en outre l'opérateur conditionnel ?: qui est un opérateur *ternaire*.

Chaque opérateur est brièvement décrit dans la suite de ce chapitre

8.2.1 Opérateurs arithmétiques

Les opérateurs arithmétiques de C réalisent des opérations mathématiques comme l'addition ou la soustraction. Il en existe deux unaires et cinq binaires.

8.2.1.1 Opérateurs arithmétiques unaires

Ces opérateurs ne peuvent être utilisés qu'avec des variables. L'opération réalisée est d'ajouter ou de soustraire 1 de l'opérande.

L'opérateur unaire peut être placé avant (mode *préfix*) ou après (mode *postfix*) l'opérande :

- En mode préfix, l'incrémentation et la décrémentation sont effectuées avant l'utilisation de l'opérande.

Opérateur	Symbole	Opération	Exemples
Incrémentation	++	Augmente de 1 la valeur de l'opérande	++x, x++
Décrémentation	-	Décrémente de 1 la valeur de l'opérande	- -x, x-
Signe plus	+	La valeur de x	+x
Signe opposé	-	L'opposé de x	-x

TABLE 8.2 – Les opérateurs mathématiques unaires du langage C

- En mode postfix les opérateurs d'incrémentation et de décrémentation modifient l'opérande après son utilisation.

Cette explication sera plus claire avec exemple :

```
x = 10;    // x prend la valeur 10
y = x ++; // y prend la valeur 11
```

8.2.1.2 Opérateurs arithmétiques binaires

Opérateur	Symbole	Opération	Exemples
Addition	+	Additionne les deux opérandes	x+y
Soustraction	-	Soustrait la valeur du 2e opérande à la valeur du 1 ^{er}	x-y
Multiplication	*	Multiplie les deux opérandes	x*y
Division	/	Divise le premier opérande par le second	x/y
Modulo	%	Donne le reste de la division du 1 ^{er} opérande par le 2 ^e	x%y

TABLE 8.3 – Les opérateurs mathématiques binaires du langage C

Les opérandes des opérateurs arithmétiques peuvent être de n'importe quel type, à l'exception de ceux de l'opérateur % qui doivent être de type entier.

Les conversions arithmétiques usuelles sont effectuées sur les opérandes. Par exemple, 3.0/2 est équivalent à 3.0/2.0. Après une telle conversion, le résultat sera du type des opérandes.

Notez bien que le résultat de la division de deux opérandes de type entier est aussi un entier !

```
6/4;      // Résultat : 1
6%4;      // Résultat : 2
6.0/4.0   // Résultat : 1.5
```

On peut aussi employer les opérateurs ++, -, +(addition) et - (soustraction) avec les *pointeurs*. pour plus d'information sur les *pointeurs* et leur arithmétique, voyez le chapitre 13.

8.2.1.3 Opérateurs d'affectation

L'opérateur d'affectation est le signe égale (=). Dans un programme C, l'instruction :

```
x = y;    //Ne signifie pas que x égale y
```

L'opérateur d'affectation indique à l'ordinateur "d'affecter la valeur de y à x". Cette instruction doit être composée d'une expression à droite du signe égale, et d'un nom de variable à gauche de ce signe :

`variable = expression;`

Les affectations sont effectuées par les opérateurs d'affectation *simple* ou les opérateurs d'affectation *composée*, comme le montre le tableau 8.4 L'opérande gauche d'une affectation doit être une *lvalue*,

Opérateur	Signification	Exemple	Résultat
=	Affectation simple	<code>x = y</code>	Affecté la valeur de y à x
<code>op=</code>	Affectation composée	<code>x +=y</code>	<code>x op= y</code> est équivalent à <code>x = x op (y)</code> (ou <code>op</code> est un opérateur arithmétique binaire ou un opérateur logique bit à bit binaire)

TABLE 8.4 – Les opérateurs d'affectation

c'est-à-dire une expression désignant un objet. Une nouvelle valeur est affectée à cet objet.

Les noms de variables sont les exemples les plus simples de *lvalues*. Dans le cas d'une variable *ptr* de type *pointeur*, `ptr` et `*ptr` sont tous deux des *lvalues*.

En revanche, les constantes et les expressions comme `x + 1` ne sont pas des *lvalues*.

Les opérandes suivants sont autorisés dans une affectation simple :

- Deux opérandes de *type arithmétique* ;
- Deux opérandes avec le même type *union* ou *structure* ;
- Deux *pointeurs* vers des objets du même *type*, à moins que l'opérande droite soit la constante `NULL`.

Si un opérande est un *pointeur* vers un objet, alors l'autre opérande peut être un pointeur vers le type `void` (c'est-à-dire `void *`)

Si les deux opérandes ont des types différents, la valeur de l'opérande droit est convertie dans le type de celui de gauche.

Une *expression d'affectation* a le type et la valeur de l'opérande gauche après affectation. L'ordre des affectations est de droite à gauche.

```
|| a = b = 100; // est équivalent à a=( b=100)
|| // la valeur de 100 est affectée à b et a
```

Une *affectation composée* est de la forme : `x op= y`, où `op` est un opérateur arithmétique binaire ou opérateur logique bit à bit binaire. La valeur de `x op y` est affectée à `x`. Par exemple :

```
|| a *= b + 1; // est équivalent à a=a*( b+1)
```

Dans

une affectation composée `x op= y`, l'expression `x` n'est évaluée qu'une seule fois. C'est la seule différence entre `x op= y` et `x = x op y`

8.2.2 Opérateurs de comparaison

Toute *comparaison* est une expression de type `int` qui retourne la valeur 1 ou différente de 0 pour «vrai» ou 0 pour «faux». Les comparaisons sont effectuées grâce aux opérateurs de comparaison du tableau 8.5.

Un pointeur peut être comparé à `NULL` ou à un pointeur vers `void`. Par exemple :

```
|| int cmp, *p1, *p2 ;
|| . . .
|| cmp = p1 < p2 ; // si p1 est inférieur à p2 alors cmp =1; sinon cmp=0
```

Opérateur	Signification	Exemple	Résultat : 1 (vrai) ou 0 (faux)
<	Inférieur à	<code>x < y</code>	1 si x est strictement inférieur à y
<=	Inférieur ou égal à	<code>x <= y</code>	1 si x est inférieur ou égal à y
>	Supérieur à	<code>x > y</code>	1 si x est strictement supérieur à y
>=	Supérieur ou égal à	<code>x >= y</code>	1 si x est strictement supérieur ou égal à y
==	Égal à	<code>x == y</code>	1 si x est égal à y
!=	Différent de	<code>x != y</code>	1 si x est différent de y
			Dans tous les autres cas, l'expression retourne 0

TABLE 8.5 – Les opérateurs de comparaison

Expression	Signification	Valeur
<code>5 == 1</code>	La valeur 5 est-elle égale à 5	faux
<code>5 > 1</code>	5 est-elle plus grande que 1 ?	vrai
<code>5 != 1</code>	La valeur 5 est-elle différente de 1 ?	vrai
<code>(5 + 10) == (3 * 5)</code>	L'expression (5 + 10) est-elle égale à (3 * 5) ?	vrai

TABLE 8.6 – Exemples d'utilisations des opérateurs de comparaison

8.2.3 Opérateurs logiques

Les opérateurs logiques de C permettent de vérifier plusieurs comparaisons dans une même question. Par exemple "s'il est 7 heures du matin, un jour de semaine, et que je ne suis pas en vacances, faire sonner le réveil".

Opérateur	Signification	Exemple	Résultat : 1 (vrai) ou 0 (faux)
&&	ET logique	<code>x && y</code>	1 si x et y non tous les 2 nuls
	OU logique à	<code>x y</code>	1 si x ou y (ou les 2) non nuls
!	NON logique	<code>!x</code>	1 si x=0. Dans les autres cas, l'expression retourne 0

TABLE 8.7 – Les opérateurs logiques

Expression	Valeur
<code>exp1 && exp2</code>	Vraie si exp1 et exp2 vraies. Sinon faux
<code>exp1 exp2</code>	Vraie si exp1 vraie ou exp2 vraie. Faux si les deux expressions sont fausses
<code>!exp1</code>	Faux si exp1 est vraie. Vraie si exp1 est fausse.

TABLE 8.8 – Utilisation des opérateurs logiques

Les opérateurs peuvent avoir des opérandes de n'importe quel type scalaire (c'est-à-dire arithmétique ou pointeur). Toute valeur différente de 0 est considérée comme «vraie» ; 0 est «faux».

Comme les expressions de comparaison, les expressions logiques retournent des valeurs «vraie» ou «faux», c'est-à-dire les valeurs `int` 0 ou 1. Les opérateurs `&&` et `||` évaluent d'abord l'opérande gauche. Si cette évaluation suffit à connaître le résultat de l'opération (par exemple si l'opérande gauche de

Expression	Valeur
(5 == 5) && (6 != 2)	Vraie, car les deux opérandes sont vrais
(5 > 1) (6 < 1)	Vraie, car un opérande est vrai
(2 == 1) && (5 == 5)	Faux, car un opérande est faux
!(5 == 4)	Vrai, car l'opérande est faux

TABLE 8.9 – Exemples d'utilisation des opérateurs logiques

&& vaut 0 ou l'opérande gauche de || n'est pas nul), alors l'opérande gauche n'est pas évalué. Par exemple :

```
|| i < max && scanf("%d", &x) == 1
```

Dans cette expression logique, la fonction `scanf` n'est appelée que si `i` est inférieur à `max`.

8.2.4 Opérateurs bit à bit

Le langage C comporte six opérateurs bit à bit, qui sont décrits dans le tableau 8.10. Ils requièrent tous des opérandes entiers.

Op.	Signification	Exemple	Résultat : 1 (vrai) ou 0 (faux)
&	ET bit à bit	<code>x & y</code>	1, si 1 à la fois dans <code>x</code> et <code>y</code>
	OU bit à bit	<code>x y</code>	1, si 1 dans <code>x</code> ou <code>y</code> , ou les deux
^	OU exclusif bit à bit	<code>x ^ y</code>	1, si 1 dans <code>x</code> ou <code>y</code> , mais pas les deux
~ x	NON bit à bit	<code>~ x</code>	1, si 0 dans <code>x</code>
«	Décalage à gauche	<code>x « y</code>	Chaque bit dans <code>x</code> est décalé de <code>y</code> positions vers la g.
»	Décalage à droite	<code>x » y</code>	Chaque bit dans <code>x</code> est décalé de <code>y</code> positions vers la dr.

TABLE 8.10 – Les opérateurs bit à bit

Les opérateurs logiques bit à bit &(ET), |(OU),^(OU exclusif) et ~(NON), interprètent leurs opérandes bit par bit : un bit positionné à 1 est considéré comme «vrai», un bit positionné à 0 sera «faux». Ainsi, dans le résultat de `z = x & y`, un bit est positionné à 1 si et seulement si le bit correspondant est positionné à la fois dans `x` et `y`. Les conversions arithmétiques usuelles sont effectuées sur les opérandes.

Les opérateurs de décalage « et » déplacent la séquence de bits de l'opérande gauche d'un nombre de positions indiqué par l'opérande droite.

```
int x = 0xF, resultat;
resultat = x << 4; // retourne 0xF0
resultat = x >> 2; // retourne 0x3
```

8.2.5 Opérateurs d'accès à la mémoire

Les opérateurs du tableau 8.11 sont utilisés pour accéder aux objets en mémoire. Les termes employés ici (*pointeur*, *tableau*, *structure*, etc.) seront expliqués dans les chapitres 12 à 16.

L'opérande de l'opérateur d'adresse & doit être une expression qui désigne une fonction ou un objet. L'opérateur d'adresse & retourne l'adresse mémoire de son opérande. Une expression de la forme &x

Opérateur	Signification	Exemple	Résultat : 1 (vrai) ou 0 (faux)
<code>&</code>	Adresse de	<code>&x</code>	Un pointeur constant vers x
<code>*</code>	Indirection ou égal à	<code>*p</code>	La valeur (ou la fonction) pointée par p
<code>[]</code>	Élément d'un tableau	<code>x[i]</code>	<code>*(x+i)</code> , l'élément d'index i dans le tableau x
<code>.</code>	Membre d'une structure ou d'une union	<code>s.x</code>	Le membre de nom x dans la structure ou dans l'union s
<code>-></code>	Membre d'une structure ou d'une union	<code>p->.x</code>	Le membre de nom x dans la structure ou dans l'union pointé par p

TABLE 8.11 – Les opérateurs d'accès à la mémoire

est un *pointeur* vers x. L'opérande de `&` ne doit pas être un champ de bits, ni une variable déclarée avec le spécificateur de classe de stockage `register`.

L'opérateur d'indirection `*` permet d'accéder à un objet ou une fonction au moyen d'un *pointeur*. Si `ptr` est un *pointeur* vers un objet ou une fonction, alors `*ptr` désigne la valeur ou la fonction pointée par `ptr`. Par exemple :

```
float a[10], *pa; // Un tableau et un pointeur
pa = a; // pa pointe sur a[0]
```

En effet, puisque `pa` pointe sur `a[0]`, `pa[3]` est équivalent à `a[3]` ou `*(a+3)`.

Les opérateurs `.` et `->` désignent un membre de *structure* ou d'*union*. L'opérande gauche de l'opérateur point doit être du type *union* ou *structure*. Celui de l'opérateur flèche est toujours un pointeur vers une *structure* ou *union*. Dans les deux cas, l'opérande droit désignera un membre du type *structure* ou *union*. Le résultat a le type et la valeur du membre considéré.

Si `p` est un *pointeur* vers une *structure* ou *union* et `x` le nom de l'un des membres, alors `p->x` équivaut à `(*p).x`, et retourne le membre `x` de la *structure* ou de l'*union* sur laquelle pointe `p`.

Les opérateurs `.` et `->`, tout comme `[ ]`, ont la plus haute priorité. Ainsi l'expression `++p->` est équivalent à `++(p->x)`.

8.2.6 Autres opérateurs

Les opérateurs du tableau 8.12 n'appartiennent à aucune des catégories rencontrées jusqu'à présent.

Op.	Signification	Exemple	Résultat : 1 (vrai) ou 0 (faux)
<code>()</code>	Appel de fonction	<code>pow(x, y)</code>	Exécute la fonction avec les arguments x et y
<code>(type)</code>	Conversion	<code>(long)x</code>	La valeur de x avec le type spécifié
<code>sizeof</code>	La taille en octets	<code>sizeof(x)</code>	Le nombre d'octets occupés en mémoire par x
<code>?:</code>	Évaluation conditionnelle	<code>x?:y:z</code>	Si est différent de 0, alors y, sinon z
<code>,</code>	Opérateur séquentiel	<code>x,y</code>	Évalue d'abord x, puis y

TABLE 8.12 – Les autres opérateurs

Un *appel de fonction* est constitué d'un pointeur vers une fonction (le nom d'une fonction par exemple) suivi par une liste d'arguments entre parenthèses `( )`, et qui peut être vide.

L'opérateur de conversion (`cast`) ne peut être utilisé qu'avec des opérandes de type *scalaire* ! L'expression `(type) x` retourne la valeur de l'opérande `x` convertie dans le type spécifié entre parenthèses.

L'opérande de l'opérateur `sizeof` est soit un nom de type entre parenthèses, soit une expression qui n'est pas de type fonction. L'opérateur `sizeof` retourne le nombre d'octets nécessaires au stockage d'un objet du type spécifié ou du type de l'expression entre parenthèses

On peut écrire une expression conditionnelle grâce à l'opérateur *conditionnel* `?:` . Dans l'expression `x?:y:z`, l'opérande gauche `x` est évalué en premier. Si le résultat est différent de 0 (c'est-à-dire `x` «vrai»), le deuxième opérande `y` est évalué, et l'expression retourne la valeur de `y`. Si `x` est égal à 0 («faux»), le troisième opérande `z` est évalué et l'expression retourne la valeur de `z`.

Le premier opérande peut être de n'importe quel type scalaire. Si les deux autres opérandes ne sont pas du même type, ils subissent une conversion de type. Leur type commun sera alors le type du résultat. Les types suivants sont autorisés pour les deuxième et troisième opérandes :

- Deux opérandes de type arithmétiques ;
- Deux opérandes du même type *structure* ou *union*, ou bien le type `void` ;
- Deux pointeurs vers des objets du même type, à moins que l'un d'eux pointe vers la constante `NULL` . Si l'un des opérandes est un pointeur vers un objet, l'autre peut pointer vers `void`.

Chapitre 9

Conversions de type

Tous les objets C ont un type défini. Une conversion de type retourne la valeur d'une expression dans un autre type. Cette conversion ne peut être effectuée que sur des variables numériques. Une variable numérique peut être un `int` ou un `float`, un pointeur peut pointer vers un `double` ou un `char`, et ainsi de suite. Dans la mesure où elle peut être représentée dans le nouveau type, une conversion conserve toujours la valeur d'origine. Par exemple, les nombres en virgule flottante sont arrondis dans une conversion de `double` en `float`. Dans les programmes, on a souvent besoin de mélanger différents types dans une même expression. Qu'arrive-t-il alors ? Parfois, C se charge automatiquement des conversions nécessaires : on parle de conversion implicite ; à d'autres moments, c'est à vous d'effectuer ces conversions pour éviter d'obtenir des résultats erronés : on parle de conversion explicite. Il est d'usage pour un bon programmeur de toujours utiliser l'opérateur de conversion lorsqu'une conversion de type est nécessaire. Cela évite les alertes du compilateur en rendant toutes les conversions explicites.

9.1 Conversions automatiques de types

Comme leur nom l'indique, ce sont des conversions effectuées automatiquement par le compilateur C, sans que vous ayez à intervenir.

9.1.1 Promotion de type dans une expression

Lorsqu'une expression C est évaluée, la valeur qui en résulte est d'un type particulier. Si tous les constituants de l'expression sont du même type, le résultat est lui-même de ce type. Par exemple, si `x` et `y` sont des `int`, le résultat de l'évaluation de l'expression suivante est aussi du type `int` :

```
|| int x, y; // x et y deux entiers  
|| x + y;
```

Que va-t-il se passer si l'une des variables n'est pas du même type ? Dans ce cas, le compilateur va "s'arranger" pour éviter une perte d'informations en allant du plus "pauvre" vers le plus "riche" dans la liste suivante : `char`, `int`, `long`, `float` et `double`. Donc, si, dans l'expression, `y` était un `char`, le résultat serait du type `int`. Si on associe un `long` et un `float` dans une expression, le résultat sera de type `float`.

À l'intérieur d'une expression, les opérandes individuels sont *promus*, si c'est nécessaire, pour correspondre aux opérandes de l'expression auxquels ils sont associés. Voici les règles suivies, dans cet ordre, par le mécanisme de promotion des opérandes :

- Si l'un des opérandes est un `double`, l'autre opérande est promu au type `double`.
- Si l'un des opérandes est un `float`, l'autre opérande est promu au type `float`.
- Si l'un des opérandes est un `long`, l'autre opérande est promu au type `long`.

Si, par exemple, `x` est un `int` et `y` un `float`, l'évaluation de l'expression `x/y` entraînera la promotion de `x` en `float` avant que ne soit effectuée la division. Cela ne signifie pas que le type de `x` a été changé

mais, tout simplement, qu'une copie de `x` a été convertie en `float` avant d'effectuer la division. Le résultat de celle-ci est naturellement de type `float`. De la même façon, si `x` est un `double` et `y` un `float`, `y` sera promu en `double`.

9.1.2 Conversion par affectation

Les promotions interviennent de chaque côté du signe (`=`). Une fois évaluée, l'expression de droite est toujours promue au type de la *L Value* de gauche. Notez que cette opération peut aboutir à une "dégradation" du résultat, c'est-à-dire à sa conversion dans un type plus pauvre. Lorsqu'un `float` est dégradé en `int`, il subit généralement des pertes alors que, dans l'autre sens, ce n'est pas souvent le cas. Par exemple :

```
float f=1.23;
int i;
i = f; // i a la valeur 1 et f vaut toujours 1,23

float f; int i = 3; f = i;
printf("% f\\ n", f); // 3 .000000 qui s'affichera
```

On aurait eu d'autres surprises si `i` avait été de type `long`, comme le montre l'exemple suivant :

```
float f;
long i = 987654321 ; f = i;
printf("% f\\ n", f); // 987654336.000000 qui serait affiché
```

Là, l'erreur provient du fait que le nombre de chiffres significatifs d'un `float` est plus petit que celui d'un `long`.

9.2 Conversions explicites : l'opérateur de cast

Nous disposons avec l'opérateur de cast un moyen d'effectuer une conversion explicite d'un type dans un autre. L'opérateur de cast s'écrit en plaçant le type du résultat à obtenir entre parenthèses, devant la variable ou l'expression (entre parenthèses, elle aussi, dans ce cas) à forcer.

9.2.1 Conversion par «cast» dans les expressions arithmétiques

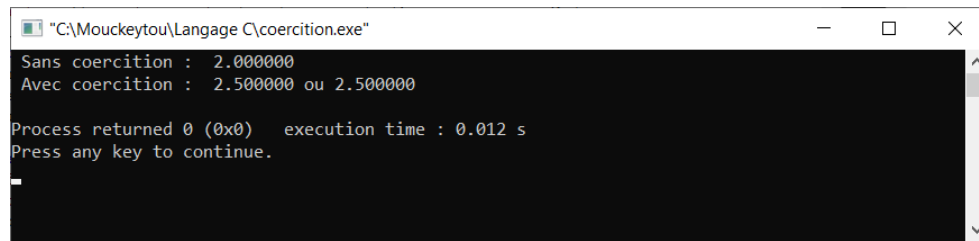
La conversion par «cast» oblige le compilateur à effectuer une conversion de type, même (et surtout) s'il n'était pas décidé à le faire implicitement. Par exemple, si `i` est de type `int`, écrire :

```
int i;
( float ) i;
```

transforme la copie interne de `i` en `float` (toujours sans changer ce qui se trouve dans la variable `i`).

À quel moment doit-on faire usage de la conversion par «cast»? Souvent pour éviter toute perte de précision dans une division, comme on peut le voir dans l'exemple suivant :

```
# include <stdio.h>
#include <stdlib.h>
int main ()
{
    int i1 = 100, i2 = 40;
    float f1, f2, f3 ;
    f1 = i1 / i2 ;
    f2 = (float)i1 / i2 ;
    f3 = (float)i1 / (float)i2 ;
    printf("Sans coercion : %f Avec coercion : %f\\n ou %f\\n",f1, f2, f3);
    exit( EXIT_SUCCESS );
}
```



```
"C:\Mouckeytou\Langage C\coercition.exe"
Sans coercion : 2.000000
Avec coercion : 2.500000 ou 2.500000

Process returned 0 (0x0)   execution time : 0.012 s
Press any key to continue.
_
```

FIGURE 9.1 – Les étapes de développement d'un programme C.

9.2.2 Conversion par «cast» appliquée aux pointeurs

Un pointeur vers void peut-être converti en n'importe quel autre pointeur vers un objet. Réciproquement, un pointeur vers un objet peut-être converti en un pointeur vers void. L'adresse qu'il indique - sa valeur - reste inchangée.

Chapitre 10

Structures de contrôle

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

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

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

10.1 Instructions

Une *instruction* précise une action à effectuer; par exemple, une opération arithmétique ou un appel de fonction. De nombreuses instructions permettent, au moyen de *boucles* et de *branchements*, de contrôler le déroulement d'un programme. Les instructions sont traitées séquentiellement.

Toute instruction qui n'est pas un bloc doit se terminer par un point-virgule.

10.2 Blocs et expressions

Un *bloc*, ou instruction composée, regroupe un ensemble d'instruction en une seule. Un bloc peut contenir des déclarations.

La syntaxe d'un bloc est la suivante :

[Liste de déclarations][liste d'instructions]

Voici un exemple :

```
{
    int i=0;           // Déclaration et initialisation de la variable i
    static long a;     // Déclaration de la variable a
    extern long max ;  // Déclaration de la variable max

    ++a;               // Instruction
    if ( a >= max )
    { . . . }          // Un bloc imbriqué
}
```

une *instruction-expression* est une expression suivie d'un point-virgule. Sa syntaxe est la suivante :

[expression] ;

Voici un exemple d'instruction-expression :

```
|| y = x; // Affectation
```

Une expression-une affectation ou un appel de fonction, par exemple - est évaluée pour ses effets de bords. Son type et sa valeur sont négligés.

Une instruction constituée d'un point-virgule seul est appelée *instruction vide*, elle n'effectue aucune opération.

10.3 Instructions de contrôle

Les instruction suivantes permettent de contrôler le déroulement d'un programme :

- les instructions de branchement ou conditionnelles : `if ... else`, `if ... else if ... else if`, `switch`;
- les boucles;
- les instruction de saut : `goto`, `continue`, `break`, `return` .

10.3.1 Instruction if

Les opérateurs de comparaison sont principalement utilisés dans les instructions `if` et `while` pour le contrôle de l'exécution du programme. Une *structure de contrôle* modifie l'ordre d'exécution des instructions. Une instruction de contrôle peut provoquer l'exécution de certaines instructions du programme plusieurs fois, ou pas d'exécution du tout selon les circonstances. L'instruction `if` en fait partie, ainsi que `do` et `while`.

L'instruction `if` évalue une expression, et oriente l'exécution du programme en fonction du résultat de cette évaluation. La syntaxe est la suivante :

```
|| if ( expression )  
||     instruction ;
```

Si le résultat de l'évaluation est vrai, l'instruction est exécutée. Dans le cas contraire, l'exécution du programme se poursuit avec l'instruction qui suit l'instruction `if`. En d'autres termes, "Exécute l'instruction suivante seulement si telle condition est vraie". Voici un exemple :

```
|| if ( x > y )  
||     y = x;
```

y prendra la valeur de x seulement si x est plus grand que y. Si x est plus petit, l'instruction n'est pas exécutée.

10.3.2 Instruction if ... else

L'instruction `if` génère un *saut conditionnel*. Sa syntaxe est la suivante :

```
|| if ( expression )  
||     instruction 1 ;  
|| else  
||     instruction 2 ;
```

Si expression est évaluée comme étant vraie, *instruction1* est exécutée, sinon c'est *instruction2* qui est exécutée. Ces deux instructions peuvent être remplacées par des blocs.

L'instruction `if` peut faire partie d'une autre instruction `if` : on parle d'instruction *imbriquée*. Voici quelques syntaxes possibles.

```

if ( expression )
    instruction 1 ;
instruction_suivante ;

```

L'instruction `if` est ici dans sa forme la plus simple. Si *expression* est vraie, *instruction1* est exécutée. Si *expression* est fausse, *instruction1* est ignorée

```

if( expression )
    instruction1 ;
else
    instruction2 ;
instruction_suivante ;

```

C'est la forme la plus courante. Si *expression* est vraie, *instruction1* est exécutée, sinon c'est *instruction2* qui est exécutée.

```

if( expression1 )
    instruction1 ;
else if( expression2 )
    instruction2 ;
else
    instruction3 ;
instruction_suivante ;

```

Si *expression1* est vraie, *instruction1* est exécutée. Dans le cas contraire, *expression2* est évaluée. Si cette dernière est vraie, *instruction2* est exécutée. Si les deux expressions sont fausses, c'est *instruction3* qui est exécutée.

10.3.3 Instruction switch

L'instruction `switch` est l'instruction de contrôle la plus souple du langage C. Elle permet à votre programme d'exécuter différentes instructions en fonction d'une expression qui pourra avoir plus de deux valeurs. Les instructions de contrôle précédentes, comme `if`, ne pouvaient évaluer que deux valeurs d'une expression : vrai ou faux. Cela obligerait, dans le cas d'un test de plus de deux valeurs, à utiliser des instructions imbriquées. L'instruction `switch` résout ce problème de façon plus simple.

Dans une instruction `switch`, la valeur de l'expression de contrôle est comparée aux constantes associées aux étiquettes `case`. Si le résultat de l'évaluation de l'expression correspond à l'une de ces constantes, l'exécution du programme se poursuit à partir de l'étiquette correspondante. Si aucune constante ne correspond, le programme exécute l'instruction suivant l'étiquette `default`. Si celle-ci est absente, l'exécution se poursuit à l'instruction suivant la fin de l'instruction `switch`.

La syntaxe de `switch` est la suivante :

```

switch( expression )
{
    case modele_1 : instruction (s);
    case modele_2 : instruction (s);
    ...
    case modele_n : instruction (s);
    default : instruction (s);
}

```

expression représente une expression qui peut être évaluée avec une valeur entière de `type long`, `int` ou `char`. L'instruction `switch` évalue cette expression, compare la valeur obtenue avec les modèles fournis dans chaque instruction `case`, puis :

- Si l'expression correspond à un des modèles énoncés, l'instruction qui suit l'instruction `case` correspondante est exécutée.

- Si l'expression ne correspond à aucun modèle, c'est l'instruction située après l'instruction **default** qui est exécutée.
- Si l'expression ne correspond à aucun modèle, et que l'instruction **switch** ne contient pas l'instruction **default**, l'exécution se poursuit avec l'instruction qui suit l'accolade de fin de **switch**.

Le code ci-dessus présente un programme simple, qui affiche un message en fonction du choix de l'utilisateur.

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

int main()
{
    int reponse ;
    puts(" Entrez un nombre entre 1 et 5 , ou 0 pour sortir : ");
    scanf("%d", &reponse);
    switch( reponse ){
        case 1: puts(" Vous avez tape 1.");
        case 2: puts(" Vous avez tape 2.");
        case 3: puts(" Vous avez tape 3.");
        case 4: puts(" Vous avez tape 4.");
        case 5: puts(" Vous avez tape 5.");
        default : puts(" choix incorrect , essayez de nouveau .");
    }
    exit( EXIT_SUCCESS );
}
```

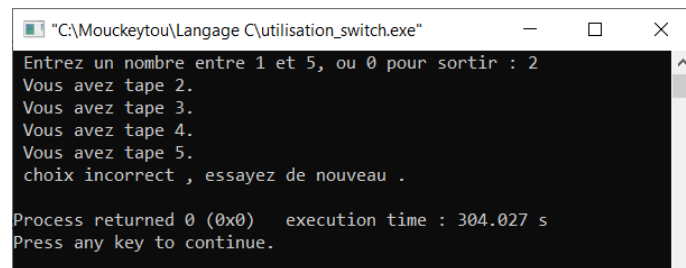


FIGURE 10.1 – Exécution d'utilisation_switch.exe

Ce résultat n'est pas satisfaisant. L'instruction **switch** a trouvé le modèle, mais elle a exécuté toutes les instructions suivantes (pas seulement celle qui est associée au modèle). Pour n'exécuter que les instructions associées à ce modèle, vous devez inclure une instruction **break**. Dans le code suivant suivant, le programme précédent est revu et corrigé en ce sens.

```
# include <stdio.h>
# include <stdlib.h >
int main()
{
    int reponse ;
    puts(" Entrez un nombre entre 1 et 5, ou 0 pour sortir : ");
    scanf ("%d",&reponse );
    switch( reponse ){
        case 1: puts(" Vous avez tape 1. "); break;
        case 2: puts(" Vous avez tape 2. "); break;
        case 3: puts(" Vous avez tape 3. "); break;
        case 4: puts(" Vous avez tape 4. "); break;
        case 5: puts(" Vous avez tape 5. "); break;
        default: puts(" Choix incorrect , essayez de nouveau .");
    }
    exit( EXIT_SUCCESS );
}
```

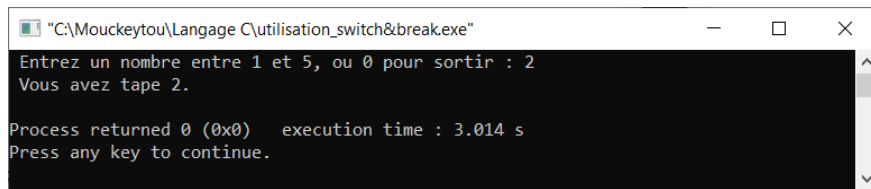


FIGURE 10.2 – Exécution d'utilisation_switch&break.exe

Cette version du programme utilise une nouvelle fonction de bibliothèque : `exit()`. La fonction `exit()` interrompt l'exécution du programme et redonne le contrôle au système d'exploitation. Cette fonction possède un argument unique de type `int`, qui lui permet de transmettre au système d'exploitation une valeur indiquant le succès ou l'échec de l'exécution du programme. Cette fonction a la syntaxe suivante :

```
|| exit(status);
```

Si la valeur de `status` est 0, cela indique une fin normale du programme. Une valeur de 1 indique, au contraire, qu'une erreur s'est produite en cours d'exécution.

Le fichier en-tête `stdlib.h` doit être inclus pour que le programme puisse utiliser la fonction `exit()`. Ce fichier définit les deux constantes symboliques qui sont les arguments de cette fonction :

```
|| # define EXIT_SUCCESS 0  
|| # define EXIT_FAILURE 1
```

10.3.4 Boucle for

La boucle `for` permet d'exécuter un certain nombre de fois un bloc d'une ou plusieurs instructions. La boucle `for` a la structure suivante :

```
|| for(initial; condition; incrément)  
|| instruction(s)
```

initial, *condition* et *incrément* sont des expressions; *instruction(s)* représente une instruction simple ou composée. La boucle `for` fonctionne de la façon suivante :

- L'expression *initial* est évaluée. Il s'agit en général d'une instruction d'affectation qui initialise une variable.
- L'expression *condition* est évaluée. *condition* est souvent une expression de comparaison. Quand *condition* est fausse (valeur 0), la boucle `for` se termine et l'exécution se poursuit avec la première instruction qui suit *instruction(s)*. Dans le cas contraire, les instructions de *instruction(s)* sont exécutées.
- L'expression *incrément* est calculée, en règle générale, incrémente la valeur de la variable initialisée dans l'expression *initial*.
- *instruction(s)* représente les instructions exécutées aussi longtemps que la condition reste vraie.

L'expression *initial* est exécutée la première, suivie de la condition. Si celle-ci est vraie, les instructions sont exécutées et l'expression *incrément* est calculée. La condition est alors contrôlée de nouveau et l'exécution se poursuit jusqu'à ce que la condition devienne fausse. Voici quelques exemples d'utilisation de la boucle `for`.

```
|| int x;  
|| for(x =0; x <10; x ++)  
|| printf("\nLa valeur de x est %d", x);
```

Ce code affiche la valeur de `x` en comptant de 0 à 9

```

int nbr = 0;
for( ; nbr != 99;)
    scanf("%d", &nbr);

```

Ce code demande une valeur à l'utilisateur jusqu'à l'obtention de 99

```

int valeur[10];
int ctr, nbr =0;
for(ctr = 0; ctr < 10 && nbr != 99; ctr++){
    puts(" Entrez un nombre ou 99 pour sortir");
    scanf("% d", &nbr);
    valeur[ctr] = nbr;
}

```

Ce code permet à l'utilisateur d'entrer 10 valeurs entières puis stocke ces valeurs dans un tableau. La boucle est interrompue si l'utilisateur tape 99.

On peut exécuter une instruction `for` à l'intérieur d'une autre instruction `for`. En imbriquant des instructions `for`, on peut réaliser une programmation très complexe. Voici un exemple d'utilisation des boucles `for` imbriquées.

```

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

void boite( int ligne , int colonne );
int main()
{
    boite (8 , 35);
    exit( EXIT_SUCCESS );
}

void boite( int ligne, int colonne ){
    int col;
    for( ; ligne >0; ligne --){
        for ( col=colonne ; col >0; col --)
            printf("X");
        printf("\n");
    }
}

```

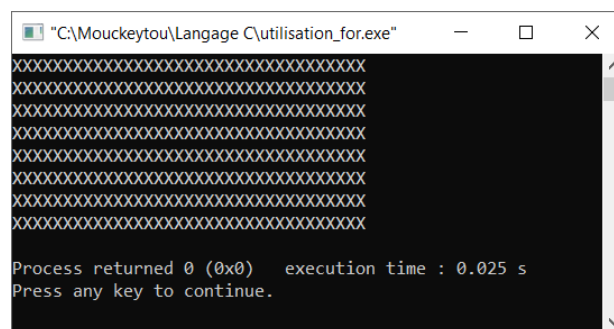


FIGURE 10.3 – Exécution d'utilisation_for.exe

10.3.5 Boucle while

La boucle `while` exécute un bloc d'instructions tant qu'une condition reste vraie. Sa structure est la suivante :

```

while( condition )
    instruction(s)

```

La *condition* est une expression C et *textslinstruction(s)* représente une instruction simple ou composée. La boucle **while** fonctionne de la façon suivante :

- L'expression *condition* est évaluée. Il s'agit en général d'une expression de comparaison. Quand cette condition est fausse (valeur 0), la boucle **while** se termine et l'exécution se poursuit avec l'instruction qui suit *textslinstruction(s)*. Sinon, les *textslinstruction(s)* sont exécutées.
- *textslinstruction(s)* représentent des instructions qui sont exécutées tant que *condition* reste vraie.

La boucle **while** permet de répéter l'exécution d'une instruction, ou d'un bloc d'instructions, tant qu'une *condition* reste vraie (valeur différente de 0). Si la *condition* est fausse à la première exécution de boucle **while**, les *textslinstruction(s)* ne sont jamais exécutées. Voici quelques exemples d'utilisation de la boucle **while**.

```
int x = 0;
while(x < 10){
    printf("La valeur de x est %d\n", x);
    x++;
}
```

Ce code est la version avec la boucle **while** du premier code de la section 8.3.4.

```
int nbr = 0;
(nbr <= 99){
    scanf("%d",&nbr);
}
```

Ce code est la version avec la boucle **while** du deuxième code de la section 8.3.4.

```
int ctr = 0;
int nbr;
while (ctr<10 && nbr!=99) {
    puts("Entrez un nombre ou 99 pour sortir");
    scanf("%d", &nbr);
    valeur[ctr] = nbr;
    ctr ++;
}
```

Ce code est la version avec la boucle **while** du troisième code de la section 8.3.4.

On peut imbriquer des instructions **while** comme avec **for** ou **if**. Voici un exemple d'utilisation des boucles **while** imbriquées.

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

int tableau [5];
int main(){
    int ctr = 0, nbr = 0;
    printf("Ce programme vous demande d'entrer 5 nombres,\n\n");
    printf("chacun compris entre 1 et 10\n");
    while (ctr <5){
        nbr = 0; while (nbr <1 || nbr >10) {
            printf("\nEntrez le nombre numero %d sur 5 : ", ctr+1);
            scanf("%d", &nbr);
        }
        tableau[ctr]=nbr;
        ctr ++;
    }
    printf("\n");
    for(ctr=0; ctr <5; ctr++)
        printf("La valeur %d est %d\n", ctr + 1, tableau[ctr]);
    exit(EXIT_SUCCESS);
}
```

```

"C:\Mouckeytou\Langage C\utilisation_while.exe"
Ce programme vous demande d'entrer 5 nombres, chacun compris entre 1 et 10

Entrez le nombre numero 1 sur 5 : 5
Entrez le nombre numero 2 sur 5 : 9
Entrez le nombre numero 3 sur 5 : 8
Entrez le nombre numero 4 sur 5 : 7
Entrez le nombre numero 5 sur 5 : 3

La valeur 1 est 5
La valeur 2 est 9
La valeur 3 est 8
La valeur 4 est 7
La valeur 5 est 3

Process returned 0 (0x0)   execution time : 8.324 s
Press any key to continue.

```

FIGURE 10.4 – Exécution d'utilisation_while.exe

10.3.6 Boucle do-while

La troisième boucle est la boucle `do while` qui exécute un bloc d'instructions tant qu'une condition reste vraie. La différence entre la boucle `do while` et les deux boucles précédentes est que le test de la condition s'effectue à la fin de la boucle. Pour les deux autres, le test se fait au début. La boucle `do while` a la structure suivante :

```

do
    instructions
while(condition);

```

condition est une expression du langage C, et *instruction(s)* représente une instruction simple ou composée. Quand une boucle `do while` est rencontrée pendant l'exécution du programme, voici ce qui se passe :

- Les instructions de *instruction(s)* sont exécutées.
- *condition* est testée. Si elle est vraie, l'exécution reprend à l'étape précédente. Si elle est fausse, la boucle se termine.

Le test de condition étant réalisé à la fin, les instructions qui font partie de la boucle `do while` sont toujours exécutées au moins une fois. Au contraire, dans les boucles `for` et `while`, si le test est faux dès le début, les instructions ne sont jamais exécutées. Quelques exemples d'utilisation de la boucle `do while`.

```

int x = 10;
do{
    printf("La valeur de x est %d\n", x);
}while (x != 10);

```

Ce code affiche 10 même si la condition fausse.

```

int nbr;
do{
    scanf("%d", &nbr);
}while(nbr <= 99);

```

Ce code est la version avec la boucle `do while` du deuxième code de la section 8.3.4.

```

int ctr = 0;
int nbr;
do{ puts("Entrez un nombre, ou 99 pour sortir");
    scanf("%d", &nbr);
    ctr ++;
}while(ctr < 10 && nbr != 99);

```

Ce code est la version avec la boucle `do while` du troisième code de la section 8.3.4.

10.3.7 Instruction goto

Cette instruction fait partie des instructions du langage C qui exécutent un saut inconditionnel ou un branchement. Rencontrée au cours de l'exécution d'un programme, une instruction `goto` provoque le transfert ou le branchement immédiat vers l'emplacement désigné. Ce branchement est dit inconditionnel, car il ne dépend d'aucune condition.

La syntaxe de `goto` est la suivante :

```
|| goto nom_etiquette;
```

Une *étiquette* est constituée d'un nom suivi d'un deux-points(:), qui précède une instruction. Voici un exemple d'utilisation de `goto`

```
|| for( ... )                //Saut hors
||     for( ... )
||         if(erreur)        //des boucles imbriquées
||             goto gestion_erreur;
||     ...
|| gestion_erreur:           //Ici on gère les erreurs
||     ...
||     ...
```

L'instruction `goto` et l'étiquette doivent appartenir à la même fonction, c'est la seule restriction. Néanmoins, l'instruction `goto` ne doit en aucun cas faire sauter de l'extérieur vers l'intérieur d'un bloc.

10.3.8 Instruction continue

L'instruction `continue` ne peut être utilisée qu'à l'intérieur du corps d'une boucle. Elle permet de sauter le reste du corps. Ainsi, dans une boucle `while` ou `do ... while`, elle passe directement au prochain test de l'expression de contrôle. Dans une boucle `for` elle saute à l'évaluation de l'expression de mise à jour.

La syntaxe de `continue` est la suivante :

```
|| continue;
```

Voici un exemple d'utilisation de l'instruction `continue` :

```
|| for(i=-10; i<10; ++i)
|| {
||     ...
||     if(i == 0)
||         continue; //ne pas tenir compte de 0
||     ...
||     ...
|| }
```

10.3.9 Instruction break

L'instruction `break` saute immédiatement à l'instruction suit la fin d'une boucle ou d'une instruction `switch`. Cela permet de terminer l'exécution d'une boucle à partir de n'importe où dans le corps de celle-ci.

La syntaxe de `break` est la suivante :

```
|| break;
```

Voici un exemple d'utilisation de l'instruction `break` :

```
|| while (1){  
||     ....  
||     if(command == ESC)  
||         continue;    // sortir de la boucle  
||     ...  
|| }
```

10.3.10 Instruction return

L'instruction **return** met un terme à l'exécution de la fonction courante et redonne le contrôle à la fonction appelante. La valeur de l'expression contenue dans l'instruction **return** est retournée à la fonction appelante : c'est la *valeur de retour* de la fonction.

La syntaxe de **return** est la suivante :

```
|| return expression;
```

Voici un exemple d'utilisation de l'instruction **return** :

```
|| int max(int a, int b) //Le maximum de a et b  
|| {  
||     return (a>b ? a : b);  
|| }
```

Une fonction peut contenir n'importe quel nombre d'instructions **return**. La valeur de l'expression **return** est convertie, si nécessaire, dans le type de la fonction. Dans l'expression **return**, l'expression entre parenthèses peut être omise. Cela n'a de sens que dans les fonctions de type **void**, mais dans ce cas c'est l'instruction **return** elle-même qui peut être omise. La fonction redonne alors le contrôle à la fonction appelante à la fin du bloc de fonction.

Chapitre 11

Principes de base des entrées/sorties

Entrée-sortie se traduit par input-output en anglais. C'est pourquoi la bibliothèque qui gère les entrées/sorties s'appelle `stdio.h` comme STanDard Input Output.

La plupart des programmes que vous allez écrire auront besoin d'afficher des informations à l'écran, ou de lire des données entrées au clavier. Dans ce chapitre, vous allez apprendre à :

- Afficher des informations à l'écran avec les fonctions de bibliothèque **`printf()`** et **`puts()`**
- Mettre en forme les messages envoyés vers l'écran
- Lire les données entrées au clavier avec la fonction de bibliothèque **`scanf()`**

11.1 Flots du C

Toutes les entrées/sorties du langage C se font avec les flots, quelle que soit leur origine ou destination. Vous constaterez que cette méthode standard de manipulation des entrées/sorties présente des avantages pour le programmeur. Il est bien sûr essentiel, de bien comprendre ce que représentent les flots et leur fonctionnement.

11.1.1 Définition des entrées/sorties

Vous savez maintenant qu'un programme en cours d'exécution range ses données dans la mémoire RAM. Ces données représentent les variables, les structures et les tableaux qui sont déclarés dans le programme. Voici l'origine de ces données et ce que le programme peut en faire :

- Les données sont issues d'un emplacement externe au programme. Les données qui sont transférées en mémoire RAM pour que le programme puisse y accéder, sont appelées les *entrées*. Les entrées proviennent le plus souvent du clavier ou de fichiers sur disque.
- Les données peuvent aussi être envoyées quelque part en dehors du programme : on les appelle alors les *sorties*. Les destinations les plus fréquentes sont l'écran, la sortie d'erreur et les fichiers disque.

Les sources d'entrées et les destinations de sorties sont appelées *unités*.

Le langage C effectue ses opérations d'entrées/sorties par l'intermédiaire des flots.

11.1.2 Définition d'un flot

Un *flot* est une séquence de caractères ou, plus exactement, une séquence d'octets de données. Une séquence d'octets qui arrive dans le programme est un flot d'entrées. Une séquence d'octets qui sort du programme est un flot de sorties. Le principal avantage des flots est que la programmation des entrées/sorties est indépendante des unités. Les programmeurs n'ont pas à créer de fonctions particulières d'entrées/sorties pour chaque unité. Le programme "voit" ses entrées/sorties comme un flot continu d'octets, quelle que soit la source ou la destination de ces données.

11.1.3 Flots de texte versus flots binaires

Il existe deux modes pour les flots de C : le mode texte ou le mode binaire. Un flot texte est constitué exclusivement de caractères, comme par exemple un message que l'on envoie sur l'écran. Le flot texte est divisé en lignes pouvant contenir 255 caractères chacune, qui se terminent par le caractère de retour à la ligne. Certains caractères de ce flot ont une signification particulière. Ce chapitre est consacré aux flots texte.

Un flot binaire peut contenir toute sorte de données, y compris les données texte. Les octets d'un flot binaire ne sont pas interprétés. Ils sont lus et écrits tels quels.

Certains systèmes d'exploitation dont Linux ne font pas la différence entre flot texte et flot binaire : tout est considéré comme flot binaire.

11.1.4 flots prédéfinis

La norme ANSI contient trois flots prédéfinis appelés *fichiers entrées/sorties standard*. Ces flots sont ouverts automatiquement au début de l'exécution d'un programme C, et fermés à la fin du programme. Le Tableau 11 contient la liste des flots standard avec les unités qui leur sont normalement connectées. Ces flots standard appartiennent au mode texte.

Nom	Flot	Unité
<code>stdin</code>	Entrée standard	Clavier
<code>stdout</code>	Sortie standard	Écran

TABLE 11.1 – Flots standard de C

Chaque fois que vous avez utilisé les fonctions `printf()` ou `puts()` pour afficher un texte à l'écran, vous avez utilisé le flot `stdout`. De la même façon, lorsque vous lisez les données entrées au clavier avec `scanf()`, vous utilisez le flot `stdin`.

11.2 Les fonctions d'entrées/sorties

La bibliothèque standard du C possède toute une variété de fonctions destinées aux entrées/sorties. Ces fonctions sont divisées en deux catégories : la première utilise toujours des flots standards, la seconde demande au programmeur de spécifier le flot. Le Tableau 11.2 donne une liste partielle de ces fonctions.

L'utilisation de toutes ces fonctions requiert le fichier en-tête `stdlib.h`. Celui-ci devra aussi être inclus pour la fonction `perror()`, et le fichier `stdargs.h` est nécessaire pour les fonctions `vprintf()` et `vfprintf()`.

11.3 Fonction printf()

Nous avons vu des exemples d'utilisation de `printf()`, mais vous n'en connaissez pas encore le mode de fonctionnement. La fonction `printf()`, qui fait partie de la bibliothèque standard du C, `stdio.h`, est la plus polyvalente pour afficher des informations à l'écran.

Afficher un texte sur l'écran est très simple : il suffit d'appeler la fonction `printf()` et de lui passer le message entre guillemets (" "). Voici un exemple :

```
|| printf("une erreur s'est produite !");  
|| /* Affiche le message une erreur s'est produite ! */
```

Utilise un flot standard	Exige un nom de flot	Action
printf()	fprintf()	Sortie formatée
vprintf()	vfprintf()	Sortie formatée avec liste d'arguments
printf()	fprintf()	Sortie formatée
puts()	fputs()	Sortie chaîne
putchar()	putc()fputc()	Sortie caractère
scanf()	fscanf()	Entrée formatée
gets() (à proscrire)	fgets()	Entrée chaîne
getchar()	getc(), fgetc()	Entrée caractère
perror	-	Sortie chaîne pour stderr

TABLE 11.2 – Les fonctions entrées/sorties de la bibliothèque standard

Pour introduire la valeur d'une variable dans votre message, transmettre deux arguments à `printf()`. Le premier est la *chaîne format* qui doit se trouver entre guillemets, le second, le nom de la variable qui contient la valeur à afficher. Par exemple :

```
|| printf("La valeur de x est %d\n", x);
|| /* Affiche le message La valeur de x est 12 */
```

11.3.1 Chaînes format de print()

Comme son nom l'indique, la chaîne format de `printf()` indique les spécifications de format pour la chaîne à émettre. Voici les trois éléments que l'on peut introduire dans une chaîne format :

- Un texte simple qui sera affiché tel quel. "La valeur de x est" représente la partie texte de l'exemple précédent.
- Un ordre de contrôle qui commence par un antislash (\) suivi d'un caractère. Dans notre exemple, \n est l'ordre de contrôle. C'est le caractère de retour à la ligne ; il signifie littéralement "se placer au début de la prochaine ligne".
- Une spécification de conversion qui est représentée par le signe de pourcentage (Celle de notre exemple est %d. Elle indique à la fonction `printf()` comment interpréter les variables que l'on veut afficher. %d signifie pour `printf()` que la variable x est un entier décimal signé.

11.3.2 Spécifications de conversion de printf()

La chaîne format contient des spécifications de conversion qui doivent correspondre, en nombre et en type, aux arguments. Cela signifie que si vous voulez afficher une variable de type *entier décimal signé* (types `int` et `long`), vous devez inclure dans la chaîne %d. Pour un *entier décimal non signé* (types `unsigned int` et `unsigned long`), vous devez inclure %u. Enfin, pour une variable à virgule flottante (types `float` ou `double`), il faut utiliser %f.

L'instruction `printf()` n'est pas limitée quant au nombre de variables qu'elle peut afficher. La chaîne format doit cependant contenir une spécification de conversion pour chacune de ces variables

11.3.3 Syntaxe de la fonction printf()

```
|| #include <stdio.h>
|| printf(chaîne -format [, arguments ,...]);
```

La fonction `printf()` peut recevoir des arguments. Ceux-ci doivent correspondre, en nombre et en

Conversion	Signification	Types convertis
%c	Un seul caractère	char
%d	Entier décimal signé	int, short
%ld	Entier décimal signé long	long
%f	Nombre à virgule flottante	float, double
%s	Chaîne de caractères	tableaux char
%u	Entier décimal non signé	unsigned int, unsigned short
%lu	Entier décimal non signé long	unsigned long

TABLE 11.3 – Spécifications de conversion les plus fréquentes

type, aux spécifications de conversion contenues dans la chaîne format `printf()` envoie les informations mises en forme vers la sortie standard (l'écran)

11.4 Fonction puts()

La fonction `puts()` permet aussi d'afficher du texte à l'écran, mais pas de variable. Elle reçoit une chaîne de caractères en argument et l'envoie vers la sortie standard (écran), en ajoutant automatiquement un retour à la ligne à la fin du message. L'instruction :

```
|| puts("Hello, world.");
```

aura le même résultat que l'instruction :

```
|| printf("Hello, world.\n");
```

Vous pouvez inclure des ordres de contrôle dans une chaîne passée à la fonction `puts()`, ils ont la même signification que pour la fonction `printf()`.

Si votre programme utilise `puts()`, vous devez inclure le fichier en-tête `stdio.h`. Voici un exemple d'appels de cette fonction :

```
|| puts("Première ligne du message. \nSeconde ligne du message.");
|| puts("Voici la troisième ligne.");
|| puts("Si nous avons utilisé printf (), ces 4 lignes auraient été sur 2
||     lignes !");
```

11.5 Lecture de données numériques avec scanf()

L'immense majorité des programmes a besoin d'afficher des données à l'écran et, de la même façon, de récupérer des données à partir du clavier. La façon la plus souple de le faire est d'utiliser la fonction `scanf()`.

La fonction `scanf()` lit les données entrées au clavier en fonction du format spécifié, et les attribue à une ou plusieurs variables du programme. Elle ne reçoit aucun caractère tant que l'utilisateur n'a pas appuyé sur la touche **Entrée**. La ligne entière de caractères est alors envoyée dans `stdin` pour être traitée par `scanf()`. L'exécution de `scanf()` se termine lorsqu'elle a traité le nombre de champs d'entrées correspondant aux conversions de sa chaîne format. Si des caractères subsistent dans `stdin` après la fin de l'exécution de `scanf()`, ils risquent de provoquer des erreurs. Cette fonction utilise, comme `printf()`, une chaîne format qui décrit le format des données qui seront lues. La chaîne format contient les mêmes spécifications de conversion que pour la fonction `printf()`. Par exemple :

```
|| scanf("%d %f", &x, &taux);
```

La chaîne reçue en argument par `scanf()` peut contenir un nombre illimité de variables.

Le symbole `&` est l'opérateur d'adresse du langage C. Ce concept est expliqué en détail dans le Chapitre 13.

Retenez simplement que vous devez le placer devant le nom de chaque variable. L'utilisation de `scanf()` implique celle du fichier en-tête `stdio.h`. voici un exemple de syntaxe de la fonction `scanf()` :

```
#include <stdio.h>
scanf(chaîne format[ , arguments ,...]);
```

La fonction `scanf()` lit des données entrées au clavier et utilise les spécifications de conversion de la chaîne format pour les attribuer aux différents arguments. Ces arguments représentent les adresses des variables plutôt que les variables elles-mêmes. L'adresse d'une variable numérique est représentée par le nom de la variable précédé du signe (`&`). Voici un exemple d'appels de cette fonction :

```
#include <stdio.h>
#include <stdlib.h>
int main(){
    float y;
    int x;
    puts("Entrez un nombre entier puis un nombre a virgule flottante :");
    scanf("%d\n %f",&x,&y);
    printf("Vous avez tape %d et %f ",x,y);
    exit(EXIT_SUCCESS);
}
```

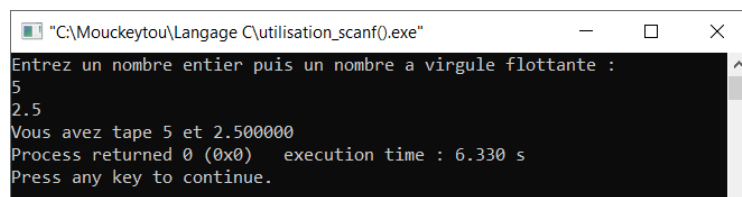


FIGURE 11.1 – Exécution de utilisation_scanf().exe

11.6 Lecture et écriture de caractères

11.6.1 Fonction getchar()

La fonction `getchar()` permet d'obtenir le caractère suivant du flot `stdin`. Elle utilise la mémoire tampon et recopie les caractères lus sur l'écran. Voici la syntaxe de sa déclaration

```
int getchar(void);
```

Voici un exemple :

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 80
int main(){
    char ch, buffer[MAX +1];
    int x = 0;
    while ((ch = getchar ()) != '\n' && x < MAX)
        buffer[x++] = ch;
    buffer[x] = '\0';
    printf("Le texte ou les caracteres choisis sont : %s\n", buffer);
    exit(EXIT_SUCCESS);
}
```

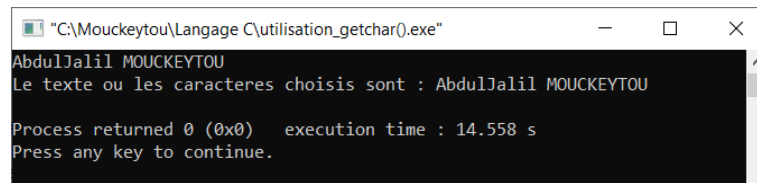


FIGURE 11.2 – Exécution de utilisation_getchar().exe

11.6.2 Fonction putchar()

La fonction `putchar()` écrit le caractère passé en paramètre sur le flux standard de sortie, en principe l'écran. Elle est équivalente à l'instruction `printf("%c",c)`, mais plus rapide. Elle se trouve dans le fichier en-tête `stdio.h`. Voici la syntaxe de sa déclaration :

```
|| int putchar(int c);
```

Bien que l'argument indiqué dans cette déclaration soit de type `int`, vous transmettez à la fonction une variable de type `char`. Vous pouvez aussi lui transmettre une variable de type `int`, du moment que sa valeur appartient à l'intervalle autorisé (0 à 255). La fonction renvoie le caractère qui vient d'être écrit ou EOF si elle a rencontré une erreur. Le programme suivant affiche, en lettre majuscules, l'alphabet française.

```
|| #include <stdio.h>
|| #include <stdlib.h>
||
|| int main()
|| {
||     char ch;
||     for(ch = 'A'; ch <= 'Z'; ch++)
||         putchar(ch);
||     putchar('\n');
||     exit(EXIT_SUCCESS);
|| }
```

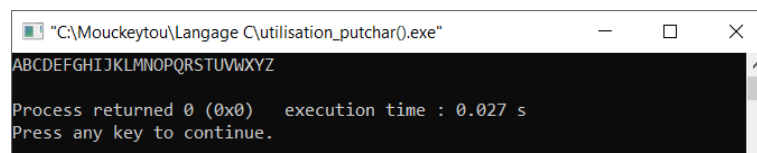


FIGURE 11.3 – Exécution de utilisation_putchar().exe

11.6.3 Fonctions putc() et fputc()

Ces deux fonctions sont analogues, elles envoient un caractère vers le flot indiqué. `putc()` est l'implémentation macro de `fputc()`. La déclaration de `fputc()` a la forme suivante :

```
|| int fputc(int c, FILE* fp);
```

Le flot de sorties est transmis à la fonction par l'intermédiaire de l'argument `FILE* fp`. Si le flot indiqué est `stdout`, `fputc()` se comportera exactement comme `putchar()`. Les deux instructions suivantes sont donc équivalentes :

```
|| putchar('x');
|| fputc('x', stdout);
```

11.7 Ouverture/fermeture de fichier

Pour l'instant, les opérations de lecture et d'écriture que nous avons faites concernaient essentiellement l'entrée et la sortie standard `stdin` et `stdout`. Il s'agit de flux (stream en anglais) dont le type est `FILE` *.

Il est possible d'associer un fichier physique (c'est à dire `C : \Users\Admin\Desktop\unfichier.txt` par exemple) à un flux. C'est ce que l'on appelle l'ouverture de fichier. L'opération inverse consiste à fermer le fichier.

11.7.1 Ouverture

Elle se fait par la fonction `fopen()` dont le prototype est :

```
|| FILE *fopen(const char *chemin , const char *mode);
```

qui prend deux arguments :

- *chemin* : le nom du fichier physique
- *mode* : le mode d'ouverture

Le mode d'ouverture est une chaîne de caractères dont voici quelques significations : "r" lecture seule "w" écriture seule, le fichier est écrasé s'il existait "r+" lecture/écriture "a" ajout de données (positionnement initial à la fin du fichier)

En plus de cela, on peut ajouter le caractère `b` pour indiquer qu'il s'agit de données binaires qui vont être écrites (norme ANSI). Sous un système POSIX comme linux, il sera ignoré. `fopen()` renvoie un descripteur de fichier ou flux, qui sera utilisable par beaucoup de fonctions de la bibliothèque `stdio.h`. Si l'ouverture a échoué, la fonction renvoie `NULL` le pointeur nul.

11.7.2 Fermeture

Lorsque l'on a terminé de lire, ou modifier un fichier, il faut impérativement le fermer car sous certains systèmes, il n'est pas automatiquement fermé à la fin de l'exécution du programme. Cela permet de finir d'écrire des données si elles avaient uniquement été stockées dans une mémoire tampon.

```
|| int fclose( FILE *stream );
```

La fonction renvoie un entier qui vaut 0 si tout s'est bien passé. Sinon, EOF est retourné et la variable `errno` est positionnée.

11.8 Lecture/écriture de fichier

Une fois qu'un fichier est ouvert, on peut le lire, écrire à l'intérieur (tout dépend de la manière dont on l'a ouvert évidemment). Il existe fondamentalement deux façons d'accéder à un fichier :

- Accès textuel : dans ce cas là, le fichier ne contient que des caractères ASCII et est affichable directement
- Accès binaire : dans ce cas là, le fichier contient des informations binaires qui sont la sauvegarde directe d'informations contenues dans les variables

Alors que l'on ne peut accéder aux fichiers textuels que de manière séquentielle, on peut accéder à un fichier binaire de manière directe en se positionnant à un endroit précis. Les fichiers binaires ont une structure qui doit être définie à l'avance, les fichiers textuels n'ont pas vraiment de structure par défaut.

11.8.1 Accès textuel

Toutes les fonctions de lecture/écriture sur l'entrée et la sortie standard ont leur équivalent pour ce qui est des fichiers quelconques. Il suffit en général de mettre un f devant. Voici quelques exemples :

fprintf - Il s'agit de la transposition de **printf** :

```
|| int fprintf( FILE *stream , const char *format , ... );
```

fscanf - Il s'agit de la transposition de **fscanf** :

```
|| int fscanf( FILE *stream , const char* format , ... );
```

fgets Il s'agit de la transposition de **gets** :

```
|| char* fgets( char* s , int size , FILE *stream );
```

fputs - Il s'agit de la transposition de **puts** :

```
|| int fputs( const char* s , FILE *stream );
```

11.8.2 Accès binaire

Il existe deux fonctions (**fwrite** et **fread**) pour écrire et lire des données binaires. On fournit à ces fonctions quatre arguments :

- Un pointeur qui va être le point mémoire qui va être lu ou écrit
- La taille d'un élément de base à copier (en octets)
- Le nombre éléments de base à copier
- Le flux concerné

Cela donne les deux prototypes suivants :

```
|| size_t fread(void* ptr , size_t size , size_t nmemb , FILE *stream);  
|| size_t fwrite(const void* ptr , size_t size , size_t nmemb , FILE *stream);
```

Chapitre 12

Tableaux

Les tableaux représentent un type de stockage de données souvent utilisé en langage C. Dans ce chapitre, vous allez étudier :

- La définition d'un tableau
- Les tableaux à une ou plusieurs dimensions
- La déclaration et l'initialisation des tableaux

12.1 Définition

Un tableau représente un ensemble d'emplacements mémoire continue qui portent le même nom et contiennent le même type de données. Chacun de ces emplacements est un élément du tableau. Pour démontrer l'utilité des tableaux, le mieux est de prendre un exemple. Supposons que vous écriviez un programme pour le calcul de vos frais professionnels. Vous gardez une trace de vos frais professionnels en classant vos reçus mois par mois. Ce programme pourrait déclarer douze variables différentes, correspondant aux douze mois de l'année. Un bon programmeur utilisera plutôt un tableau de douze éléments, où le total de chaque mois est stocké dans l'élément correspondant.

La Figure 10.1 vous montre la différence entre l'utilisation de variables individuelles et un tableau.

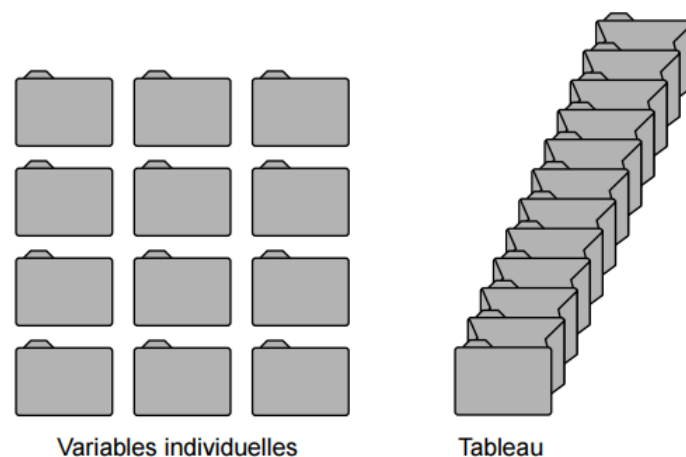


FIGURE 12.1 – Variables en dossiers individuels vs tableau en dossier ayant de multiples compartiments.

12.1.1 Tableaux à une dimension

Un *tableau à une dimension* ne possède qu'un seul index. Un *index* est le nombre entre crochets qui suit le nom du tableau. Il indique le nombre d'éléments du tableau. Dans le cas du programme de calcul de vos frais professionnels, par exemple, vous pourriez déclarer un tableau de type `float` :

```
|| float depenses [12];
```

Un élément de tableau peut contenir n'importe quel type de donnée du langage C. Les éléments sont toujours numérotés en commençant à 0.

Pour chaque déclaration de tableau, le compilateur réserve un bloc de mémoire d'une taille suffisante pour contenir la totalité des éléments. Ceux-ci seront stockés séquentielle-ment comme le montre la Figure 10.2 Un élément de tableau s'utilise comme une variable isolée de même type.

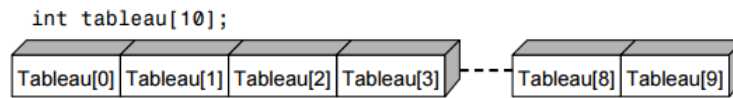


FIGURE 12.2 – Éléments de tableau sont stockés en mémoire de façon séquentielle.

Il est référencé au moyen du nom de tableau suivi de son index entre crochets. L'instruction suivante, par exemple, stocke la valeur 89,95 dans le second élément de notre tableau `depenses` :

```
|| depenses [1] = 89.95;
```

L'index du tableau peut être une constante comme dans notre exemple, mais aussi une expression, une variable entière, ou même un autre élément de tableau. Voici quelques exemples :

```
|| float depenses [100];
|| int a [10];
||
|| depenses [i]=100;          /* i est une variable entière */
|| depenses [2 + 3] = 100;    /* équivalent de depenses [5] */
|| depenses [a[2]] = 100;     /* a[] est un tableau contenant des entiers */
```

N'oubliez pas que, dans un tableau de n éléments, l'index est compris entre 0 et $n-1$. L'emploi de la valeur d'index n ne sera pas décelé par le compilateur, mais cela provoquera des erreurs dans les résultats du programme.

12.1.2 Tableaux à plusieurs dimensions

Un tableau à plusieurs dimensions possède plusieurs index. Un tableau à deux dimensions en a deux, un tableau à trois dimensions en a trois, etc. En langage C, il n'y a pas de limite au nombre de dimensions qu'un tableau peut avoir.

Quel que soit le nombre de ses dimensions, un tableau est stocké en mémoire de façon séquentielle.

12.2 Nom et déclaration des tableaux

Les règles concernant l'attribution d'un nom à un tableau sont les mêmes que celles qui régissent les noms de variables (voir Chapitre 6). Un nom de tableau doit être unique : il ne doit pas avoir été attribué précédemment à un autre tableau, une variable ou une constante, etc. Une déclaration de tableau a la même forme qu'une déclaration de variable, mis à part le fait que le nombre d'éléments du tableau doit apparaître entre crochets immédiatement après son nom.

Dans la déclaration, le nombre d'éléments du tableau peut être une constante littérale ou une constante symbolique créée avec l'ordre `#define`. Exemple :

```
|| #define MOIS 12
|| int tableau[MOIS];
```

12.2.1 Initialisation de tableaux à une dimensions

Vous pouvez initialiser l'intégralité ou seulement une partie d'un tableau, au moment de sa déclaration. Il faut prolonger la déclaration du tableau d'un signe égal suivi d'une liste entre accolades de valeurs séparées par des virgules. Ces valeurs sont attribuées dans l'ordre aux éléments du tableau en commençant à l'élément 0. Par exemple, l'instruction suivante attribue la valeur 100 à `tableau[0]`, 200 à `tableau[1]`, 300 à `tableau[2]` et 400 à `tableau[3]` :

```
|| float dépenses [100];  
|| int tableau [4] = { 100, 200, 300, 400 };
```

Si vous n'indiquez pas la taille du tableau, le compilateur va créer un tableau avec autant d'éléments que de valeurs initiales. L'instruction suivante est donc parfaitement équivalente à la précédente :

```
|| float dépenses [100];  
|| int tableau [] = { 100, 200, 300, 400 };
```

Si les éléments d'un tableau ne sont pas initialisés en début de programme, vous ne connaissez pas les valeurs qui y sont stockées quand le programme s'exécute. Si vous initialisez plus d'éléments que le tableau n'en contient, le compilateur envoie un message d'erreur.

12.2.2 Initialisation de tableaux à plusieurs dimensions

Les tableaux à plusieurs dimensions peuvent aussi être initialisés. Les valeurs sont attribuées séquentiellement en incrémentant d'abord le dernier index :

```
|| int tableau [4] [3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
```

Cette instruction affecte les valeurs dans l'ordre suivant :

<code>tableau[0][0] = 1</code>	<code>tableau[0][1] = 2</code>	<code>tableau[0][2] = 3</code>
<code>tableau[1][0] = 4</code>	<code>tableau[1][1] = 5</code>	<code>tableau[1][2] = 6</code>
<code>tableau[2][0] = 7</code>	<code>tableau[2][1] = 8</code>	<code>tableau[2][2] = 9</code>
<code>tableau[3][0] = 10</code>	<code>tableau[3][1] = 11</code>	<code>tableau[3][2] = 12</code>

Quand vous initialisez un tableau à plusieurs dimensions, vous pouvez rendre le code source plus clair en groupant les valeurs entre des accolades supplémentaires, et en les répartissant sur plusieurs lignes. Notre exemple précédent pourrait être réécrit de cette façon :

```
|| int tableau [4] [3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9} , {10, 11, 12}};
```

Il ne faut pas oublier la virgule qui doit séparer les valeurs, même si elles sont déjà séparées par des accolades

12.2.3 Taille maximale

La mémoire occupée par un tableau dépend du nombre et de la taille des éléments qu'il contient. La taille d'un élément dépend de la taille, sur votre ordinateur, du type de donnée qu'il contient. L'espace mémoire peut être calculé à l'intérieur d'un programme au moyen de l'opérateur `sizeof()`. C'est un opérateur unaire, et non une fonction.

Chapitre 13

Pointeurs

Les pointeurs jouent un rôle très important dans le langage C. Ils permettent de manipuler les données dans vos programmes. Vous apprendrez dans ce chapitre :

- La définition d'un pointeur
- L'utilisation des pointeurs
- La déclaration et l'initialisation des pointeurs
- L'utilisation des pointeurs avec des variables simples et des tableaux
- Le passage des tableaux à une fonction avec les pointeurs

L'utilisation de pointeurs offre de nombreux avantages qui peuvent être partagés en deux catégories : celle des tâches qui sont exécutées de manière plus simple avec des pointeurs, et celle des tâches qui ne pourraient pas être réalisées sans pointeurs. Pour devenir un bon programmeur en langage C, il est impératif de bien comprendre les principes de fonctionnement des pointeurs.

13.1 Définition

Pour comprendre ce que sont les pointeurs, vous devez avoir une idée de la façon dont votre ordinateur stocke les informations en mémoire.

13.1.1 Mémoire de votre ordinateur

La mémoire vive de votre PC est constituée de millions d'emplacements mémoire rangés de façon séquentielle. Chaque emplacement a une adresse unique, comprise entre 0 et une valeur maximale qui dépend de la quantité de mémoire installée sur votre machine.

Quand votre ordinateur fonctionne, une partie de sa mémoire vive est occupée par le système d'exploitation. Si vous lancez un programme, le code (les instructions en langage machine) et les données qu'il utilise occuperont en partie cette mémoire. Nous allons étudier de quelle façon votre programme occupera cette mémoire.

Quand un programme déclare une variable, le compilateur réserve un emplacement mémoire avec une adresse unique pour stocker cette variable. Il associe l'adresse au nom de la variable. Quand le programme utilise le nom de la variable, il accède automatiquement à l'emplacement mémoire correspondant. Ce mécanisme est transparent pour l'utilisateur du programme, qui utilise le nom de la variable sans se soucier de l'endroit où l'ordinateur l'a stockée. Cela est illustré par le schéma de la Figure 13.1.

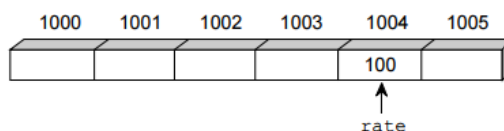


FIGURE 13.1 – Une variable de programme a une adresse de mémoire spécifique

Une variable appelée *rate* est déclarée et initialisée à 100. Le compilateur a réservé un emplacement mémoire à l'adresse 1004, qu'il associe donc au nom de la variable.

13.1.2 Création d'un pointeur

Vous pouvez remarquer que l'adresse de la variable *rate* est un nombre, et qu'elle peut donc être utilisée comme n'importe quel autre nombre en langage C. Si vous connaissez l'adresse d'une variable, vous pouvez créer une autre variable pour y stocker l'adresse de la première. Dans notre exemple, la première étape consiste à déclarer la variable dans laquelle on stockera l'adresse de *rate*. Nous allons l'appeler *p_rate*. Le schéma suivant montre qu'un emplacement a bien été réservé pour *p_rate*, mais la valeur qui y est stockée est inconnue.

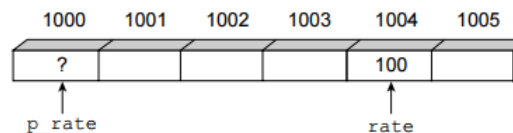


FIGURE 13.2 – Un emplacement a été alloué à la variable *p_rate*

L'étape suivante consiste à stocker l'adresse de *rate* dans la variable *p_rate*. Celle-ci représente maintenant l'emplacement mémoire de la variable *rate* : en langage C, *p_rate* pointe sur *rate*, ou *p_rate* est un pointeur vers *rate*.

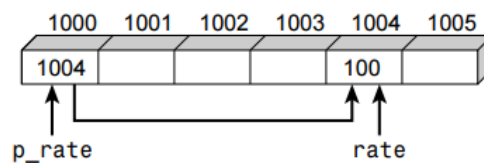


FIGURE 13.3 – *p_rate* contient l'adresse de la variable *rate*, *p_rate* est un pointeur vers *rate*

En résumé, un pointeur est une variable qui contient l'adresse d'une autre variable. Étudions maintenant l'utilisation de ces pointeurs dans un programme C.

13.2 Pointeurs et variables simples

L'exemple précédent montrait une variable pointant sur une autre variable simple (ce n'est pas un tableau). Voyons maintenant comment créer et utiliser ce type de pointeur.

13.3 Déclaration

Un pointeur est une variable numérique qui doit être déclarée, comme toutes les variables, avant d'être utilisée. Le nom des pointeurs suit les mêmes règles que celui des autres variables et doit être unique.

La déclaration d'un pointeur a la forme suivante :

```
|| nomtype *ptr;
```

nomtype représente le type de la variable pointée. L'astérisque (*) est l'opérateur indirect, il indique que *ptr* est un pointeur vers une variable de type **nomtype**, et non une variable de type **nomtype**. Une déclaration peut contenir des pointeurs et des variables. Voici quelques exemples :

```
|| char *ch1, *ch2;           //ch1 et ch2 pointent sur une variable de type char
|| float *valeur, pourcent; // valeur est un pointeur vers une variable de
||                               //type float et pourcent est une variable de type float
```

13.3.1 Initialisation

Déclarer un pointeur n'est pas suffisant ; si vous ne le faites pas pointer sur une variable, il est inutile. Pour les mêmes raisons qu'avec des variables, travailler avec un pointeur qui n'a pas été initialisé peut se révéler désastreux. Un pointeur doit contenir l'adresse d'une variable, et c'est le programme qui doit s'en charger en utilisant l'opérateur d'adresse (&). Quand il est placé avant le nom de la variable, l'opérateur d'adresse renvoie l'adresse de cette variable. L'initialisation d'un pointeur est donc une instruction de la forme :

```
|| pointeur = &variable;
```

13.4 Pointeurs, mode d'emploi

Maintenant que vous savez déclarer et initialiser un pointeur, vous devez apprendre à l'utiliser. L'opérateur indirect (*) entre de nouveau en jeu. Quand cet opérateur précède le nom d'un pointeur, il fait référence à la variable qui est pointée.

Reprenons notre pointeur `p_rate` initialisé pour pointer vers la variable `rate`. `*p_rate` représente la variable `rate`. Pour afficher la valeur de la variable, vous pouvez écrire :

```
|| printf("%d", rate);          ou bien          printf("%d", *p_rate);
```

En langage C, ces deux instructions sont équivalentes. Si vous accédez au contenu de la variable en utilisant son nom, vous effectuez un *accès direct*. Si vous accédez au contenu d'une variable par l'intermédiaire de son pointeur, vous effectuez un *accès direct* ou une *indirection*.

La Figure 13.4 montre que le nom d'un pointeur précédé d'un opérateur d'indirection se réfère à la valeur pointée.

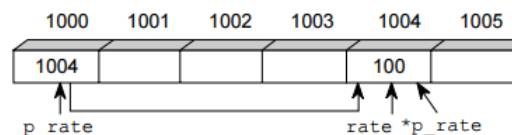


FIGURE 13.4 – L'opérateur d'indirection associé à un pointeur

Les pointeurs sont très importants en langage C, il est essentiel de bien comprendre leur fonctionnement. Si votre pointeur s'appelle `ptr` et qu'il a été initialisé pour pointer sur la variable `var` alors :

- `*ptr` et `var` représentent le contenu de `var`.
- `ptr` et `&var` représentent l'adresse de `var`.

Voici un exemple :

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

int main()
{
    int var = 1;
    int *ptr;

    ptr = &var;
    printf("Accès direct, var = %d\n", var);
    printf("Accès indirect, var = %d\n\n", *ptr);
    printf("L'adresse de var = %d\n", &var);
    printf("L'adresse de var = %d\n", ptr);

    exit(EXIT_SUCCESS);
}
```

```

"C:\Mouckeytou\Langage C\utilisation_pointeur-indirection.exe"
Acces direct , var = 1
Acces indirect , var = 1

L'adresse de var = 6422036
L'adresse de var = 6422036

Process returned 0 (0x0)   execution time : 0.029 s
Press any key to continue.

```

FIGURE 13.5 – Exécution de utilisation_pointeur-indirection.exe

13.5 Pointeurs et types de variables

Les différents types de variables du langage C n'occupent pas tous la même mémoire. Sur la plupart des systèmes, une variable `int` prend quatre octets, une variable `double` en prend huit, etc. Chaque octet en mémoire possède sa propre adresse; une variable qui occupe plusieurs octets occupe donc plusieurs adresses.

L'adresse d'une donnée est en fait l'adresse du premier octet occupé. Voici un exemple qui déclare et initialise trois variables :

```

int vint = 12252;
char vchar = 90;
double vdoube = 1200.156004;

```

Ces variables sont stockées en mémoire comme le montre la Figure 13.6

Voici la déclaration et l'initialisation des pointeurs vers ces trois variables :

```

int *p_vint;
char *p_vchar;
double *p_vdouble;
//des instructions peuvent être insérées ici
p_vint = &vint;
p_vchar = &vchar;
p_vdouble = &vdoube;

```

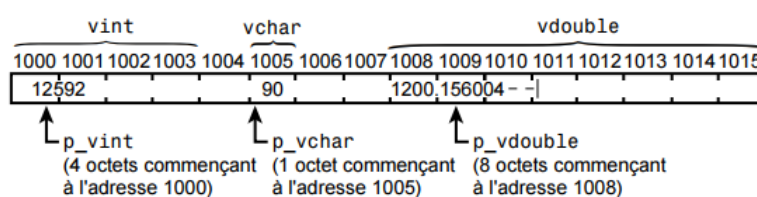


FIGURE 13.6 – Des variables de différents types n'occupent pas tous la même quantité de mémoire

Chaque pointeur contient l'adresse du premier octet de la variable pointée. Ainsi, `p_vint` est égal à 1000, `p_vchar` a la valeur 1004 et `p_vdouble` est égal à 1005. Le compilateur sait qu'un pointeur vers une variable de type `int` pointe sur le premier des quatre octets, qu'un pointeur de variable de type `double` pointe sur le premier des huit octets, etc.

13.6 Pointeurs et tableaux

Les pointeurs sont très utiles pour travailler avec les variables, mais ils le sont encore plus quand on les utilise avec les tableaux. En fait, les index de tableaux dont on a parlé au Chapitre 10 ne sont rien d'autre que des pointeurs.

13.6.1 Noms de tableau et pointeurs

Un nom de tableau sans les crochets s'utilise la plupart du temps comme un pointeur vers le premier élément du tableau. Si vous avez déclaré le tableau `data[]`, `data` aura la valeur de l'adresse de `data[0]` et sera donc équivalent à l'expression `&data[0]`.

Le nom du tableau n'est pas un pointeur même s'il est souvent utilisé comme tel.

Vous pouvez quand même déclarer un pointeur variable et l'initialiser pour pointer sur le premier élément du tableau :

```
int tableau[100], *p_tableau;
// instructions
p_tableau = tableau;
```

`p_tableau` étant un pointeur variable, il peut être modifié pour pointer ailleurs. Contrairement à `tableau`, `p_tableau` ne pointe pas obligatoirement sur le premier élément de `tableau[]`. Il pourrait, par exemple, pointer vers d'autres éléments du tableau. Mais avant cela, il faut savoir comment sont stockés en mémoire les éléments d'un tableau

13.6.2 Stockage des éléments d'un tableau

Les éléments d'un tableau sont stockés dans des emplacements mémoire séquentiels, le premier élément ayant l'adresse la plus basse. L'adresse d'un autre élément, par rapport au premier, dépend de la taille des valeurs stockées dans le tableau et de l'index de cet élément.

Prenons comme exemple un tableau de type `int`. Une variable `int` occupe quatre octets en mémoire. Chaque élément du tableau sera donc situé quatre octets plus loin que le précédent, et son adresse sera égale à celle de l'élément précédent plus quatre. Avec des variables de type `double` (8 octets), chaque élément serait stocké dans huit octets adjacents; la différence entre les adresses de deux éléments voisins serait alors de huit.

La Figure 13.7 illustre les relations existant entre le stockage du tableau et les adresses, pour un tableau de type `int` avec six éléments, et pour un tableau de type `double` avec trois éléments.

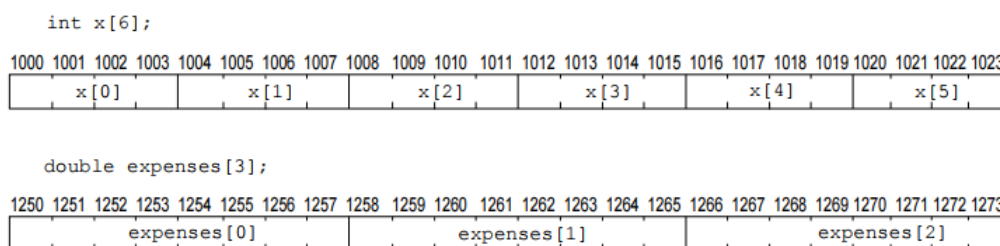


FIGURE 13.7 – Stockage de tableaux contenant différents types de données

En étudiant la Figure 13.7, vous comprendrez pourquoi les expressions suivantes sont vraies :

```
1: x == 1000
2: &x[0] == 1000
3: &x[1] == 1004
4: expenses == 1250
5: &expenses[0] == 1250
6: &expenses[1] == 1258
```

Vous pouvez constater, à partir de ces exemples, qu'un pointeur devra être incrémenté de quatre pour accéder à des éléments successifs d'un tableau de type `int`, et de huit dans le cas d'un tableau de type `double`.

Pour accéder à des éléments successifs d'un tableau contenant un type de donnée particulier, le pointeur

devra être incrémenté de la valeur `sizeof` (typedonnée). L'opérateur `sizeof` renvoie la taille en octets du type de donnée reçu en argument. Mieux, s'il s'agit bien d'un pointeur et non d'un tableau, incrémentez-le de la valeur de `sizeof(ptr)` où `ptr` est le nom de votre pointeur.

13.6.3 Pointeur arithmétique

Nous venons de voir que le pointeur du premier élément d'un tableau doit être incrémenté d'un nombre d'octets égal à la taille des données du tableau pour pointer sur l'élément suivant. Pour pointer sur un élément quelconque en utilisant une notation de type pointeur, on utilise le *pointeur arithmétique*.

13.6.3.1 Incrémenter les pointeurs

Incrémenter un pointeur consiste à en augmenter la valeur. Si vous incrémentez un pointeur de 1, le pointeur arithmétique va augmenter sa valeur pour qu'il accède à l'élément de tableau suivant

Par exemple, si `ptr_int` pointe sur un élément de tableau de type `int`, l'instruction suivante :

```
|| ptr_int ++;
```

incrémente la valeur de ce pointeur de 4 pour qu'il pointe sur l'élément `int` suivant.

De la même façon, si vous augmentez la valeur du pointeur de `n`, C va incrémenter ce pointeur pour qu'il pointe sur le *n*-ième élément suivant :

```
|| ptr_int += 2;
```

Cette instruction va augmenter de 8 la valeur du pointeur, pour qu'il pointe 2 éléments plus loin.

13.6.3.2 Décrémenter les pointeurs

La décrémentation des pointeurs suit le même principe que l'incrémentation. Si vous utilisez les opérateurs `--` ou `(-)` pour décrémenter un pointeur, le pointeur arithmétique va diminuer sa valeur automatiquement en fonction de la taille des données pointées.

13.6.3.3 Autre utilisation des pointeurs

La dernière opération que l'on peut effectuer avec des pointeurs est la *différence* entre deux pointeurs. Si vous avez deux pointeurs sur un même tableau, le résultat de leur soustraction correspond au nombre d'éléments les séparant. Exemple :

```
|| ptr1 - ptr2
```

Cette instruction donne le nombre d'éléments qui séparent les deux éléments pointés par `ptr1` et `ptr2`. Les opérateurs de comparaison `==`, `!=`, `>`, `<`, `>=` et `<=` peuvent aussi être utilisés. Les premiers éléments d'un tableau ont toujours une adresse plus basse que les derniers. Ainsi, si `ptr1` et `ptr2` sont deux pointeurs d'un même tableau, la comparaison :

```
|| ptr1 < ptr2
```

est vraie si `ptr1` pointe sur un élément d'index plus petit que `ptr2`.

Beaucoup d'opérations arithmétiques sont réservées aux variables simples, car elles n'auraient aucun sens avec les pointeurs. Par exemple, si `ptr` est un pointeur, l'instruction :

```
|| ptr *= 2;
```

générera un message d'erreur. La liste des six opérations possibles avec les pointeurs est la suivante :

Opérateur	Description
Affectation	Vous pouvez attribuer une valeur à un pointeur, valeur qui doit correspondre à une adresse obtenue avec l'opérateur d'adresse (&), ou à partir d'un pointeur constant.
Indirection	L'opérateur indirect (*) donne la valeur stockée à l'emplacement pointé
Adresse de	Vous pouvez utiliser l'opérateur d'adresse pour trouver l'adresse d'un pointeur et obtenir un pointeur vers un pointeur.
Incrément	On peut ajouter un nombre entier à la valeur d'un pointeur pour pointer sur un emplacement mémoire différent.
Décrément	On peut soustraire un entier à la valeur d'un pointeur pour pointer sur un emplacement mémoire différent.
Différence	Vous pouvez soustraire un entier de la valeur d'un pointeur pour pointer sur un emplacement mémoire différent.
Comparaison	Ces opérateurs ne sont valides que pour deux pointeurs d'un même tableau

13.7 Précautions d'emploi

Quand vous utilisez des pointeurs dans un programme, une grosse erreur est à éviter : utiliser un pointeur non initialisé à gauche d'une instruction d'affectation. Par exemple, dans la déclaration suivante :

```
|| int *ptr;
```

Le pointeur n'est pas initialisé. Cela signifie qu'il ne pointe pas sur un emplacement connu. Si vous écrivez :

```
|| *ptr = 12;
```

La valeur 12 va être stockée à l'adresse (inconnue) pointée par *ptr*. Cette adresse peut se situer n'importe où en mémoire. La valeur 12 risque d'écraser une donnée importante, et le résultat peut aller de simples erreurs dans un programme à l'arrêt complet du système.

À ne pas faire : Effectuer des opérations mathématiques comme des divisions, des multiplications ou des modulus avec des pointeurs. Les seules opérations possibles sont l'incrémentation ou le calcul de la différence entre deux pointeurs d'un même tableau.

N'oubliez pas que l'ajout ou la soustraction d'un entier à un pointeur change la valeur de ce pointeur en fonction de la taille des données sur lesquelles il pointe.

Incrémenter ou décrémenter une variable de tableau. Initialisez un pointeur avec l'adresse de début du tableau et incrémentez-le.

13.8 Pointeurs et index de tableaux

Un nom de tableau sans les crochets est un pointeur vers le premier élément du tableau. Par conséquent, vous pouvez accéder au premier élément de ce tableau avec l'opérateur indirect (*). Si *tab* est un tableau, l'expression **tab* représente le premier élément de ce tableau, **(tab+1)* est le deuxième élément, etc. En généralisant, nous obtenons les relations suivantes :

```
|| *(tab) == tab [0]           || //saut au (n+1)-ième élément
|| *(tab +1) == tab [1]       || *(tab+n) == tab [n]
```

Ces expressions vous donnent les équivalences entre la notation de l'index et celle utilisant les pointeurs. Le compilateur C ne fait aucune différence entre ces deux méthodes d'accès aux données d'un tableau.

Chapitre 14

Chaînes de caractères

Une chaîne de caractères est une séquence de caractères qui permet de manipuler des textes. Vous apprendrez dans ce chapitre :

- Créer des tableaux de type `char` pour stocker des chaînes de caractères
- Initialiser les chaînes de caractères
- Utiliser les pointeurs avec les chaînes de caractères
- Lire et imprimer des chaînes de caractères

Il n'existe pas de type de donnée pour les chaînes de caractères. Un nom ou une adresse sont des exemples de chaînes de caractères. Le langage C manipule ce genre d'informations à l'aide des tableaux de caractères.

14.1 Tableaux de caractères

Pour stocker une chaîne de six caractères, il faut déclarer un tableau de type `char` avec sept éléments :

```
|| char chaine[7];
```

Une chaîne est une séquence de caractères qui se termine par le caractère nul `\0`. C'est le septième élément. Bien qu'il soit représenté par deux caractères (antislash et zéro), le caractère nul est interprété comme un seul caractère et sa valeur ASCII est 0.

Si un programme C stocke la chaîne Alabama, il stocke les sept caractères A, l, a, b, a, m et a, suivis du caractère nul. Il faut donc un tableau de huit éléments.

La taille d'une variable de type `char` est de un octet. Le nombre d'octets d'un tableau de caractères sera donc égal au nombre d'éléments.

14.2 Initialiser les tableaux de caractères

Les tableaux de caractères peuvent être initialisés dans l'instruction de déclaration de cette façon :

```
|| char chaine[10] = { 'A', 'l', 'a', 'b', 'a', 'm', 'a', '\0' };
```

Vous pouvez aussi utiliser la chaîne littérale, séquence de caractères entre guillemets, qui est plus facile à écrire et à lire :

```
|| char chaine[10] = "Alabama";
```

Dans ce cas, le compilateur ajoute automatiquement le caractère nul à la fin de la chaîne. De même, si la taille du tableau n'a pas été indiquée, le compilateur la calculera. La ligne suivante, par exemple, crée et initialise un tableau de huit éléments :

```
|| char chaine[] = "Alabama"
```

Le caractère nul permet aux fonctions qui manipulent des chaînes de caractères de connaître la longueur de la chaîne. Si vous l'avez oublié, la fonction n'a aucun autre moyen de déterminer la fin de la chaîne ; elle va continuer à traiter les données en mémoire tant qu'elle ne rencontre pas de caractère nul.

14.3 Chaînes et pointeurs

Une chaîne de caractères est stockée dans un tableau de type `char` , et la fin de cette chaîne est représentée par le caractère nul. Pour définir une chaîne, il suffit donc de pointer au début de cette chaîne. L'utilisation du nom du tableau dans lequel elle est stockée, non suivi de crochets, est la méthode standard d'accès à une chaîne de caractères. La bibliothèque standard de C contient de nombreuses fonctions qui manipulent des chaînes de caractères.

Pour passer une chaîne en argument, la fonction s'attend à recevoir le nom du tableau dans lequel elle est stockée. C'est la méthode à utiliser avec les fonctions de la bibliothèque, en particulier avec `printf()` et `puts()` que nous avons étudiées.

14.4 Chaînes sans tableaux

Nous venons de voir que le nom du tableau qui contient la chaîne est un pointeur sur le début de la chaîne, et que le caractère nul représente la fin de cette chaîne. Le rôle du tableau ne consiste qu'à fournir de la place mémoire pour stocker la chaîne de caractères.

Pour se passer du tableau, il faut pouvoir s'allouer de la place mémoire, définir un pointeur en début de chaîne et placer le caractère nul à la fin. Il existe deux méthodes : dans la première, pour une chaîne littérale dont la taille est définie au moment de la compilation du programme, la mémoire est allouée une bonne fois pour toutes au lancement du programme. La seconde consiste à utiliser la fonction `malloc()` qui alloue la mémoire au moment de l'exécution du programme ; le procédé s'appelle allocation dynamique.

14.4.1 Allouer la mémoire nécessaire à la compilation

Le début de la chaîne est représenté par un pointeur vers une variable `char`. Par exemple :

```
|| char *message;
```

Cette instruction déclare le pointeur `message` vers une variable de type `char`, mais le pointeur ne pointe encore sur rien. Si vous écrivez :

```
|| char *message = "Le fantôme \du grand César !";
```

vous obtenez le stockage de la chaîne de caractères `Le fantôme du grand César !` (avec un caractère nul à la fin) quelque part en mémoire, et le pointeur `message` est initialisé pour pointer sur le premier caractère. Pour accéder à la chaîne, on utilisera le pointeur.

Voici une instruction équivalente à la précédente :

```
|| char message[] = "Le fantôme \du grand César !";
```

Cette méthode d'allocation de mémoire est parfaite quand vous connaissez vos besoins en écrivant le programme. Dans le cas contraire, il faudra utiliser la fonction `malloc()` pour allouer la mémoire de façon dynamique.

14.4.2 Fonction `malloc()`

La fonction `malloc()` est une des fonctions de C qui permettent de réserver de l'espace mémoire. On lui transmet en argument le nombre d'octets nécessaires, elle se charge de trouver et de réserver un

bloc de mémoire libre, puis renvoie l'adresse du premier octet de ce bloc au programme appelant. La donnée renvoyée par la fonction `malloc()` est un pointeur de type `void`. Un pointeur de ce type sera compatible avec tous les types de données.

14.4.2.1 Syntaxe de la fonction `malloc()`

```
|| #include <stdlib.h>  
|| void *malloc(size_t taille);
```

La fonction `malloc()` réserve un bloc de mémoire du nombre d'octets indiqué dans *taille*. Cette méthode d'allocation de mémoire permet d'optimiser la gestion de la mémoire de votre ordinateur. L'appel de cette fonction ne peut se faire que si vous avez inclus le fichier en-tête `stdlib.h`. La valeur renvoyée par `malloc()` est un pointeur vers le bloc de mémoire qui a été réservé. Si la recherche de la mémoire a échoué, la valeur renvoyée est nulle. Il est donc nécessaire de contrôler cette valeur même si la taille de la mémoire demandée est petite.

14.4.2.2 Utilisation de `malloc()`

La fonction `malloc()` peut servir à réserver de la mémoire pour un seul caractère. On déclare un pointeur vers une variable de type `char` :

```
|| char *ptr;
```

On appelle la fonction en lui passant la taille du bloc mémoire désiré. Une variable de type `char` a une taille d'un octet et la valeur renvoyée sera attribuée au pointeur :

```
|| ptr = malloc(1);
```

Cet octet réservé n'a pas de nom, `ptr` est donc le seul moyen d'y accéder. Pour y stocker le caractère `x`, vous devez écrire :

```
|| *ptr = 'x';
```

Pour réserver la mémoire nécessaire à une chaîne de caractères, la procédure est identique, mais il faut déterminer la taille maximale de cette chaîne. Notre exemple consiste à réserver de la mémoire pour une chaîne de 99 caractères, plus un pour le caractère nul. On commence en déclarant un pointeur vers une variable `char`, puis on appelle la fonction :

```
|| char *ptr;  
|| ptr = malloc(100* sizeof (*ptr));
```

`ptr` pointe maintenant sur un bloc de 100 octets qui peut recevoir une chaîne de caractères.

Avec la fonction `malloc()`, la mémoire n'est réservée qu'au moment où on en a besoin. La taille de la mémoire disponible dépend de la taille de la mémoire installée sur votre ordinateur et des besoins des autres programmes en cours d'exécution. Si la mémoire libre n'est pas suffisante, la fonction `malloc()` renvoie la valeur 0. Votre programme doit tester cette valeur pour vérifier que la mémoire demandée a bien été attribuée. Vous pouvez tester la valeur renvoyée par `malloc()` en la comparant à la constante symbolique `NULL` qui est définie avec `stdlib.h`.

14.4.3 Fonction `calloc`

La fonction `calloc()` alloue aussi de la mémoire mais, au lieu d'allouer un groupe d'octets comme le fait `malloc()`, `calloc()` alloue un groupe d'*objets*. La zone de mémoire allouée est remise à zéro et la fonction retourne un pointeur vers le premier octet de la zone. En somme, `calloc()` initialise la mémoire allouée, ce que ne fait pas `malloc()`. Si l'allocation ne peut être satisfaite, `calloc()` se contente de renvoyer `NULL`. Voici son prototype :

```
|| void *calloc(size_t num , size_t size);
```

On alloue (ou, plus exactement, on tente d'allouer) *num* blocs de *size* octets chacun.

14.4.4 Fonction `realloc()`

Cette fonction modifie la taille d'un bloc mémoire précédemment alloué par un appel à `malloc()` ou `calloc()`. Voici son prototype :

```
|| void *realloc(void *ptr , size_t size);
```

Le pointeur *ptr* pointe sur le bloc de mémoire original. L'argument *size* indique non pas le supplément de mémoire qu'on veut obtenir, mais la taille totale qu'on veut donner au bloc (inférieure ou supérieure à celle du bloc original). Plusieurs cas sont possibles :

- S'il y a assez de place pour satisfaire la requête, un nouveau bloc est alloué et *ptr* contient son adresse. Le contenu de la zone de mémoire supplémentaire est indéterminé, l'ancien contenu étant inchangé.
- S'il n'y a pas assez de place, la fonction renvoie `NULL` et le contenu de l'ancien bloc n'est pas altéré.
- Si *ptr* contient `NULL`, `realloc()` se comporte comme `malloc()`.
- Si *size* vaut zéro et que *ptr* ne vaut pas `NULL`, la zone précédemment allouée et pointée par *ptr* est libérée et la fonction renvoie `NULL`.

14.4.5 Fonction `free()`

Le pool de mémoire dans lequel on effectue des prélèvements pour satisfaire les appels à `malloc()`, `calloc()` ou `realloc()` n'est pas infini ; il convient de restituer ce qui a été acquis dans un programme avant de passer la main au système d'exploitation. La libération d'un bloc de mémoire s'effectue en appelant la fonction `free()` dont le prototype est :

```
|| void free(void *ptr);
```

Attention, cette fonction ne renvoie rien. La mémoire pointée par *ptr* est restituée au pool de mémoire. Si *ptr* vaut `NULL`, la fonction ne fait rien du tout.

14.5 Affichage de chaînes et de caractères

Un programme qui manipule des chaînes de caractères a souvent besoin de les afficher à l'écran. Les deux fonctions utilisées sont `puts()` et `printf()`, deux fonctions étudiées au Chapitre 11.

14.6 Lecture des chaînes de caractères

Les deux fonctions de bibliothèque `fgets()` et `scanf()` complètent les deux précédentes en permettant la lecture de chaînes de caractères entrées au clavier. Avant de pouvoir lire une chaîne, un programme doit prévoir un endroit pour la stocker. Il peut suivre pour cela une des deux méthodes déjà étudiées : déclarer un tableau ou appeler la fonction `malloc()`.

14.6.1 Fonction `fgets()`

La fonction `fgets()` lit une chaîne de caractères entrée au clavier par l'utilisateur. Quand cette fonction est appelée, elle lit tout ce qui est tapé sur le clavier, jusqu'à ce que l'utilisateur enfonce la touche de retour à la ligne (touche Entrée). La fonction prend en compte le retour à la ligne, ajoute un caractère nul en fin de chaîne et renvoie la chaîne (tronquée si nécessaire à la taille indiquée en argument) au programme appelant. Cette chaîne est stockée à l'adresse indiquée par le pointeur de type `char` qui a

été passé à `fgets()`. Le prototype de `fgets()` se trouve dans `stdio.h` :

```
|| char *fgets(char *str , int n, FILE *fp);
```

L'argument `str` est un pointeur vers une mémoire tampon dans lequel sera placée la ligne de caractères lue; `n` est le nombre maximum de caractères à lire et `fp` est le pointeur de type `FILE` renvoyé par `fopen()` lors de l'ouverture du fichier. La fonction `fgets()` renvoie un pointeur vers la chaîne qu'elle vient de lire. En cas de problème, la valeur renvoyée est nulle. Avec `stdin`, l'exécution de `fgets()` se poursuivra jusqu'à ce que vous tapiez sur la touche Entrée, mais seuls `n-1` caractères seront stockés dans la chaîne. Le programme ci-dessus présente le fonctionnement de la fonction `fgets()`

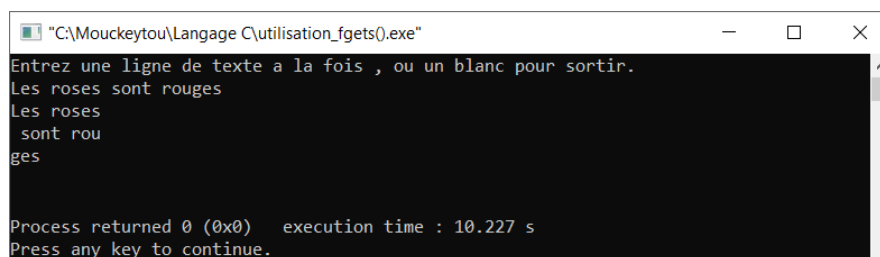


FIGURE 14.1 – Exécution d'`utilisation_fgets().exe`.

14.6.2 Fonctions `scanf()` et `fscanf()`

Pour lire à partir du clavier la valeur 12,86, par exemple, et l'attribuer à une variable de type `float`, vous devrez utiliser les fonctions `scanf()` et `fscanf()`. Ces deux fonctions sont analogues. `scanf()` utilise toujours `stdin`, alors que `fscanf()` utilise le flot d'entrées spécifié par l'utilisateur. `fscanf()` est utilisée pour les entrées fichier. Le prototype de `fscanf()` se trouve dans le fichier d'en-tête `stdio.h` et se présente ainsi :

```
|| int fscanf(FILE *fp , char *fmt ,...);
```

Le premier argument est un pointeur vers un objet de type `FILE`. Pour lire des informations à partir d'un fichier sur disque particulier, il faut passer à la fonction le pointeur récupéré au moment de l'ouverture de ce fichier par `fopen()`.

Le deuxième argument est une chaîne de caractères contenant le format à utiliser. Vous avez déjà rencontré ce type de chaîne de caractères, lorsque nous avons étudié `scanf()`.

Le dernier argument, `...`, signifie qu'on trouve ensuite un nombre variable d'arguments. Ces derniers sont les adresses des variables dans lesquelles `fscanf()` placera les valeurs lues sur le fichier disque.

14.7 Manipulation de chaîne de caractères

Le texte, sous forme de chaînes de caractères, constitue une partie importante de beaucoup de programmes. Jusqu'ici, vous avez appris à faire des entrées-sorties de texte, et vous savez comment les chaînes de caractères sont conservées en mémoire. Pour leur manipulation, C dispose d'une grande variété de fonctions concentrées majoritairement dans les bibliothèques `string.h`, `ctype.h` et `stdlib.h`.

Dans la bibliothèque `string.h`, vous trouverez les fonctions suivantes :

```
|| size_t strlen(char *str);
```

Calcule la longueur d'une chaîne de caractères

```
|| strcpy(char *destination, char *source);
```

Copie la totalité de la chaîne source dans la chaîne destinataire

```
|| strncpy(char *destination, char *source, size_t n);
```

Copie, au plus, les n premiers caractères de la chaîne source dans la chaîne destinataire

```
|| char *strdup(char *source);
```

Identique à strcpy() avec allocation automatique de mémoire pour la chaîne destinataire

```
|| void *memcpy(char *destination, char *source, size_t n);
```

Copie exactement les n premiers caractères de la chaîne source dans la chaîne destinataire

```
|| char *strcat(char *str1, char *str2);
```

Concatène les chaînes *str1* et *str2*

```
|| char *strncat(char *str1, char *str2, size_t n);
```

Ajoute les n premiers caractères de la chaîne *str1* à la chaîne *str2*

```
|| int strcmp(char *str1, *str2);
```

Effectue la comparaison de deux chaînes de caractères par caractères

```
|| int strncmp(char str1, str2, size_t n);
```

Compare les n premiers caractères de la chaîne *str1* à la chaîne *str2*

```
|| char *strchr(char str, int ch);
```

Recherche la première occurrence du caractère *ch* dans la chaîne de caractères *str*.

```
|| char *strrchr(char *str, int ch);
```

Recherche la dernière occurrence du caractère *ch* dans la chaîne de caractères *str*.

```
|| char *strcspn(char *str1, char *str2);
```

Recherche la première occurrence dans *str1*, de l'un des caractères de *str2*

```
|| size_t strspn(char *str1, char *str2);
```

Renvoie la position du premier caractère dans *str1* n'ayant pas d'équivalent dans *str2*

```
|| char *strpbrk(char *str1, char *str2);
```

Renvoie un pointeur vers le premier caractère de *str1* qui correspond à un caractère de *str2*

```
|| char *strstr(char *str1, char *str2);
```

Retourne un pointeur vers la première occurrence de *str2* dans *str1*

Dans la bibliothèque `ctype.h`, vous trouverez les fonctions suivantes :

```
|| char *tolower(char c);
```

Convertit le caractère fourni en argument de majuscules en minuscules

```
|| char *toupper(char c);
```

Convertit le caractère fourni en argument de minuscules en majuscules

```
|| int isalnum(int c);
```

Renvoie vrai si *c* est une lettre ou un chiffre

```
|| int isalpha(int c);
```

Renvoie vrai si *c* est une lettre

```
|| int isascii (int c);
```

Renvoie vrai si *c* est un caractère ASCII standard (compris entre 0 et 127)

```
|| int iscntrl (int c);
```

Renvoie vrai si *c* est un caractère de contrôle

```
|| int isdigit (int c);
```

Renvoie vrai si *c* est un chiffre

```
|| int isgraph (int c);
```

Renvoie vrai si *c* est un caractère imprimable (autre qu'un espace)

```
|| int islower (int c);
```

Renvoie vrai si *c* est une lettre minuscule (bas de casse)

```
|| int isprint (int c);
```

Renvoie vrai si *c* est un caractère imprimable (espace compris)

```
|| int ispunct (int c);
```

Renvoie vrai si *c* est un caractère de ponctuation

```
|| int isspace (int c);
```

Renvoie vrai si *c* est un caractère d'espacement

```
|| int isupper (int c);
```

Renvoie vrai si *c* est une lettre majuscule (capitale)

```
|| int isxdigit (int c);
```

Renvoie vrai si *c* est un chiffre hexadécimal (0 à 9 et A à F)

Dans la bibliothèque `stdlib.h`, vous trouverez les fonctions suivantes :

```
|| long int strtol(const char *ptr, char **endptr, int base);
```

Convertit une chaîne de caractères en une valeur de type *long int*

```
|| long long int strtoll(const char *ptr, char **endptr, int base);
```

Convertit une chaîne de caractères en une valeur de type *long long int*

```
|| unsigned long int strtoul(const char *ptr, char **endptr, int base);
```

Convertit une chaîne de caractères en une valeur de type *unsigned long int*

```
|| unsigned long long int strtoull(const char *ptr, char **endptr, int base);
```

Convertit une chaîne de caractères en une valeur de type *unsigned long long int*

```
|| double strtod(char *ptr, char ** endptr);
```

Convertit une chaîne de caractères en une valeur de type *double*

```
|| float strttof(const char *ptr, char ** endptr);
```

Convertit une chaîne de caractères en une valeur de type *float*

```
|| int sprintf(char *str, const char *format, ...);
```

Convertit des nombres (et des chaînes) en une chaîne de caractères

Chapitre 15

Types évolués

Il est aussi possible de définir de nouveaux types comme les *types énumérés* et les types dérivés. Ceux-ci comprennent les pointeurs, les tableaux, les structures et les unions.

Les types de base et les types énumérés sont aussi appelés, de manière générique, *types arithmétiques*. Les types arithmétiques et les pointeurs forment l'ensemble des *types scalaires*. Les tableaux et structures sont appelés les *types agrégés*.

15.1 Définition d'un nouveau type

15.1.1 Commande typedef

Il est possible de définir des raccourcis vers de nouveaux types avec la commande `typedef`. La syntaxe :

```
|| typedef <un type existant> <nouveau type>;
```

Par exemple le type booléen pourrait être défini ainsi :

```
|| typedef int booleen;
```

En réalité, la bibliothèque `stdbool.h` le définit grâce à une commande du préprocesseur `#define` car dans ce cas là, cela ne pose aucun problème. Ainsi, on pourrait déclarer une variable booléenne grâce à :

```
|| booleen variable_booleenne;
```

Ou même un tableau de booléens :

```
|| booleen tableau_de_booleens [20];
```

Voici d'autres exemples :

```
|| typedef int Entier;           // Entier voudra dire int
|| typedef char *Adresse;       // type Adresse = pointeur vers char
|| typedef int Hauteur;         // Hauteur est complètement équivalent à int
```

15.1.2 Types qui représentent des tableaux

On peut utiliser `typedef` pour faire des raccourcis vers des types qui représentent des tableaux. Attention à la syntaxe :

```
|| typedef char Chaine20 [21];
```

Dans ce cas là, *Chaine20* est un nouveau type qui représente un tableau de 21 caractères. On peut

donc déclarer des variables comme ceci :

```
|| Chaine20 nom , prenom;
```

15.1.3 typedef vs. #define

Tout d'abord, attention, la syntaxe est inversée :

```
|| #define Hauteur int           ⇔           || typedef int Hauteur;
```

Ne pas oublier le ; à la fin d'une **typedef**. Ne surtout pas mettre de ; à la fin d'un **#define**, cela provoque des erreurs très dures à détecter.

Les principales différences :

- **#define** est une directive du préprocesseur, cela signifie que lors de la compilation, toutes les occurrences du nouveau type auront déjà été remplacée par le type équivalent.
- **typedef** est une instruction du C
- **#define** ne permet pas de définir des types de tableaux
- **#define** est normalement plus rapide
- **typedef** est surtout utilisé pour définir des types qui sont des raccourcis vers des structures (voir la suite du cours).

15.2 Structures

La réalisation de nombreuses tâches de programmation se trouve simplifiée par les structures. Une structure est une méthode de sauvegarde des données choisie par le programmeur ; elle répond donc exactement aux besoins du programme. Vous allez étudier :

- ce que représente une structure simple ou complexe ;
- la définition et la déclaration des structures ;
- le mode d'accès aux données des structures ;
- la création des structures pour stocker des tableaux et des tableaux des structures ;
- la déclaration de pointeurs dans les structures et de pointeurs vers de structures ;
- le passage de structures en arguments de fonctions ;
- l'utilisation de définitions types avec les structures.

15.2.1 Structures simples

Une structure contient une ou plusieurs variables groupées sous le même nom pour être traitées comme une seule entité. Contrairement aux variables stockées dans un tableau, les variables d'une structure peuvent être de types différents. Une structure peut contenir tous les types de données C, y compris les tableaux et les autres structures. Chaque variable d'une structure est appelée *membre* de cette structure.

15.2.1.1 Définition et déclaration des structures

Le code d'un programme graphique doit travailler avec les coordonnées des points de l'écran. Vous pouvez définir une structure **coord** contenant les valeurs **x** et **y** d'un écran de cette façon :

```
|| struct coord{  
||     int x;  
||     int y;  
|| };
```

Le mot clé **struct** identifie le début de la structure et informe le compilateur que *coord* est le type de structure. Les accolades renferment la liste des variables membres de la structure. Chaque membre doit être défini par un type de variable et un nom.

Notre exemple définit une structure *coord* qui contient deux variables entières *x* et *y*, mais ne déclare pas de structure. Il y a deux façons de déclarer les structures :

Déclaration 1	Déclaration 2
<pre>struct coord{ int x; int y; }premier ,second;</pre>	<pre>struct coord{ int x; int y; }; struct coord premier , second;</pre>

Ces instructions définissent la structure de type *coord* et déclarent deux structures appelées *premier* et *second* appartenant au type *coord*. *premier* contient deux membres entiers appelés *x* et *y*, tout comme *second*.

15.2.1.2 Accès aux membres d'une structure

Pour faire référence à un membre particulier, on sépare le nom de la structure concernée de celui du membre visé, avec l'opérateur (*.*). Pour que la structure *premier* représente le point de l'écran de coordonnées *x* = 50, *y* = 100, on utilise la syntaxe suivante :

```
premier.x = 50;
premier.y = 100;
```

Le principal avantage des structures sur les variables est la possibilité de copier des informations entre structures de même type par une seule instruction simple. Dans le cas de notre exemple précédent, l'instruction :

```
premier = second;                    ⇔                    premier.x = second.x;
premier.y = second.y;
```

Syntaxe du mot clé struct

```
struct nom_modele{
    membre(s);
    /* instructions */
}occurrence;
```

Le mot clé **struct** permet de définir la structure. Une structure contient une ou plusieurs variables (membre(s)) groupées sous un même nom pour être traitées comme une seule entité. Ces variables peuvent être de n'importe quel type : tableaux, pointeurs ou autres structures.

15.2.2 Structures plus complexes

Après avoir introduit les structures simples, nous pouvons étudier des types de structures plus intéressants. Ce sont les structures dont les membres sont d'autres structures ou des tableaux.

15.2.2.1 Structures contenant des structures

Les membres d'une structure peuvent être d'autres structures. Reprenons l'exemple précédent. Supposons que votre programme ait besoin de traiter des rectangles. Un rectangle peut être représenté par les coordonnées de deux coins diagonalement opposés. Vous pouvez définir une structure de type rectangle avec les instructions suivantes :

```
struct rectangle{
    struct coord hautgauche;
    struct coord basdroite;
};
```

La structure de type *rectangle* contient deux structures appartenant au type *coord* : *hautgauche* et *basdroite*.

Ces instructions ne constituent que le modèle de la structure, pour la déclarer, vous devez inclure une instruction du type :

```
|| struct rectangle maboite;
```

Vous auriez pu combiner définition et déclaration de cette façon :

```
|| struct rectangle{
||     struct coord hautgauche;
||     struct coord basdroite;
|| }maboite;
```

Pour accéder aux membres de type *int* de deux structures imbriquées, il faut utiliser deux fois l'opérateur (*.*) : *maboite.hautgauche.x*

La Figure 15.1 vous explique les relations existant entre les différents membres et variables de cet exemple.

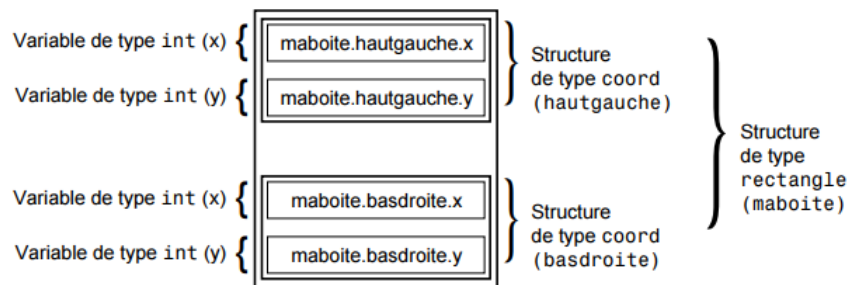


FIGURE 15.1 – Relations entre une structure, des structures imbriquées et les membres de celles-ci.

15.2.2.2 Tableaux membres de structures

Vous pouvez définir une structure constituée d'un ou de plusieurs tableaux. Les tableaux peuvent contenir tous les types de données C. Les instructions suivantes, par exemple, définissent un modèle de structure *data* qui contient un tableau d'entiers de 4 éléments appelé *x*, et un tableau de caractères de 10 éléments appelé *y* :

```
|| struct data{
||     int x[4];
||     char y[10];
|| };
```

Vous pouvez ensuite déclarer une structure record appartenant au type *data* avec l'instruction :

```
|| struct data record;
```

La Figure 15.2 représente cette structure de tableaux. Pour accéder aux éléments de tableaux qui sont

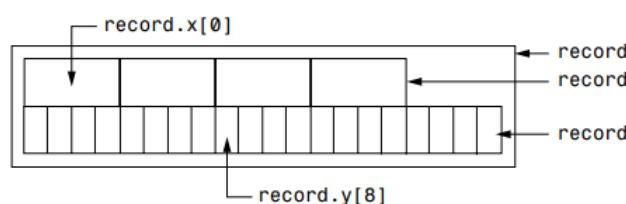


FIGURE 15.2 – Organisation d'une structure contenant des tableaux.

membres d'une structure, on associe le nom du membre et l'index du tableau :

```
|| record.x[2] = 100;
|| record.y[1] = 'x';
```

15.2.3 Tableaux de structures

Le langage C est un langage très puissant et sa faculté de créer des tableaux de structures en est un bon témoignage. Voici pourquoi les tableaux de structures représentent un formidable outil de programmation.

En général, le programme a besoin de plusieurs modèles de données. Par exemple, dans un programme qui remplit la fonction de répertoire téléphonique, vous pouvez définir une structure contenant le nom et le numéro de téléphone de chaque personne :

```
|| struct entry{
||     char fnom [10];
||     char pnom [12];
|| };
```

Une liste telle que celle-ci doit pouvoir recevoir de nombreuses occurrences. La déclaration d'une structure de ce type ne sera donc pas très utile. Pour obtenir un vrai répertoire il faut créer un tableau de structures de type *entry* :

```
|| struct entry list [1000];
```

Cette instruction déclare un tableau *list* de 1000 éléments. Chacun de ces éléments est une structure de type *entry* qui sera identifiée par un index, comme tout élément de tableau.

Quand votre tableau de structures est déclaré, vous avez de nombreuses possibilités de manipulation des données. Par exemple, pour copier un élément de tableau dans un autre, vous pouvez écrire :

```
|| list [1] = list [5];
```

ou pour copier un membre d'un élément du tableau dans un autre :

```
|| strcpy(list [1].phone, list [5].phone);
```

15.2.4 Initialisation des structures

Comme tout autre type de variable C, les structures peuvent être initialisées quand elles sont déclarées. La procédure à suivre est la même que pour les tableaux. La déclaration est suivie d'un signe égal puis, entre accolades, d'une liste de valeurs d'initialisation séparées par des virgules ou pour copier un membre d'un élément du tableau dans un autre :

```
|| 1: struct vente{
|| 2:     char client [20];
|| 3:     char article [20];
|| 4:     float montant;
|| 5: }mesventes = { "Acme Industries",
|| 6:               "ciseaux gauchers",
|| 7:               1000.00
|| 8: };
```

L'exécution de ces instructions donne les résultats suivants :

- Définition d'un type de structure appelé *vente* (lignes 1 à 5).
- Déclaration d'une structure appelée *mesventes* appartenant au type *vente* (ligne 5).
- Initialisation du membre *mesventes.client* avec la chaîne "Acme Industries" (ligne 5).
- Initialisation du membre *mesventes.article* avec la chaîne "ciseaux gauchers" (ligne 6).
- Initialisation du membre *mesventes.montant* avec la valeur 1000.00 (ligne 7).

15.2.5 Structures et pointeurs

Les pointeurs étant un concept très important en langage C, il n'est pas surprenant de les retrouver avec les structures. Un membre de structure peut être un pointeur, et vous pouvez déclarer un pointeur vers une structure.

15.2.5.1 Pointeurs membres d'une structure

Un pointeur qui est un membre d'une structure se déclare de la même façon qu'un pointeur qui ne l'est pas, en utilisant l'opérateur indirect (*):

```
|| struct data{  
||     int *valeur;  
||     int *taux;  
|| }premier;
```

Cette déclaration doit être suivie de l'initialisation de ces pointeurs avec les adresses des variables pointées. Si nous supposons que *cout* et *interet* sont des variables de type `int`, l'initialisation des pointeurs suit la syntaxe suivante :

```
|| premier.valeur = &cout;  
|| premier.taux = &interet;
```

Vous pouvez maintenant utiliser l'opérateur indirect (*). L'expression `*premier.valeur` a la valeur de la variable *cout* et l'expression `*premier.taux` a la valeur de la variable *interet*. Autre exemple :

```
|| struct msg{  
||     char *p1;  
||     char *p2;  
|| }myptrs;  
  
|| myptrs.p1 = "Le langage C";  
|| myptrs.p2 = "par Willam MOUCKEYTOU";
```

15.2.5.2 Pointeurs vers des structures

Un programme C peut déclarer et utiliser des pointeurs vers des structures exactement comme il peut déclarer des pointeurs vers tout autre type de donnée. Ces pointeurs sont souvent utilisés pour passer une structure comme argument à une fonction.

Voici comment un programme peut créer et utiliser des pointeurs vers des structures. La première étape est la définition de la structure :

```
|| struct part{  
||     int nombre;  
||     int *taux;  
|| }premier;
```

La seconde est la déclaration d'un pointeur vers une variable de type *part*.

```
|| struct part *p_part;
```

Le pointeur ne peut être initialisé, car il n'existe pas encore de structure appartenant au type *part*. La valeur d'un pointeur étant une adresse en mémoire, il faut déclarer une structure de type *part* avant de pouvoir pointer dessus :

```
|| struct part gizmo;
```

Cette instruction permet d'initialiser le pointeur :

```
|| p_part = &gizmo;
```

La valeur du pointeur `p_part` représente l'adresse de la structure `gizmo`, tandis que `p_part` fait directement référence à `gizmo`.

Il existe une autre technique pour accéder aux membres d'une structure avec le pointeur vers cette structure. Cette technique utilise l'opérateur d'indirection, représenté par le signe moins suivi du signe "supérieur à" (`->`). Cet opérateur est placé entre le nom du pointeur et le nom du membre. Pour accéder au membre `degizmo` avec le pointeur `p_part`, utilisez la syntaxe suivante :

15.2.5.3 Pointeurs et tableaux de structures

Les tableaux de structures et les pointeurs de structures sont des outils de programmation très puissants. On peut les combiner en utilisant les pointeurs pour accéder aux structures qui sont des éléments de tableaux.

Reprenons, pour notre démonstration, cette définition de structure :

```
|| struct part{
|     int nombre;
|     char nom[10];
| };
```

La structure étant définie, nous pouvons déclarer un tableau appartenant au type `part` :

```
|| struct part data[100];
```

Nous pouvons ensuite déclarer un pointeur vers une structure de type `part`, et l'initialiser pour pointer sur la première structure du tableau `data` :

```
|| struct part *p_part;
|| p_part = &data [0];
```

Nous aurions pu écrire la deuxième instruction de cette façon :

```
|| p_part = data;
```

Nous obtenons un tableau de structures de type `part` et un pointeur vers le premier élément du tableau. Un programme pourra afficher le contenu de ce premier élément avec l'instruction :

```
|| printf("%d %s", p_part -> nombre , p_part -> nom);
```

Pour afficher tous les éléments du tableau, il faudrait utiliser une boucle `for` et les afficher un par un à chaque exécution de la boucle.

Les éléments de tableau étant stockés en mémoire de façon séquentielle, les pointeurs arithmétiques sont particulièrement appropriés. La Figure 15.3 nous montre un tableau nommé `x[]` qui contient des éléments ayant une taille de quatre octets (une structure contenant deux membres de type `int` par exemple). Le pointeur `ptr` a été initialisé pour pointer sur `x[0]`. Chaque fois que l'opérateur (`++`) sera utilisé, `ptr` pointera sur l'élément suivant.

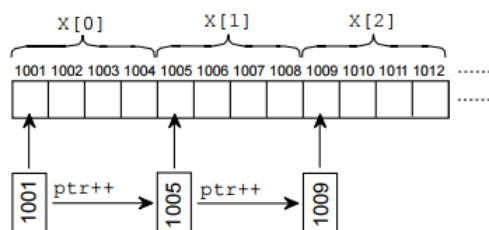


FIGURE 15.3 – Opérateur `++` positionne le pointeur sur l'élément suivant.

Cela signifie qu'un programme peut accéder aux éléments d'un tableau de structures en incrémentant un pointeur. Voici un exemple :

```
for(count = 0; count < MAX; count++){
    printf("A l'adresse %d: %d %s\n", p_part, p_part->nombre, p_part->nom);
    p_part++;
}
```

Cette notation est plus facile et plus concise que celle qui utilise l'index.

15.2.5.4 Passage de structures comme arguments de fonctions

Le programme suivant vous montre comment passer une structure à une fonction.

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

/* Déclaration et définition d'une structure stockant les données.*/
struct data{
    float montant;
    char fnom [30];
    char lnom [30];
}rec;

/* Prototype de la fonction. Cette fonction ne renvoie pas de */
/* valeur , et son argument est une structure de type data. */
void print_rec(struct data x);

int main()
{
    /* Lecture des données au clavier. */
    printf("Entrez les nom et prénom du donateur ,\n");
    printf("séparés par un blanc : ");
    scanf("%30s %30s", rec.fnom, rec.lnom);
    printf("\nEntrez le montant du don : ");
    scanf("%f", &rec.montant);
    /* Appel de la fonction. */
    print_rec(rec);
    exit(EXIT_SUCCESS);
}

void print_rec(struct data x){
    printf("\nLe donateur %s %s a donné %.2f FCFA\n", x.fnom, x.lnom,
        x.montant);
}
```

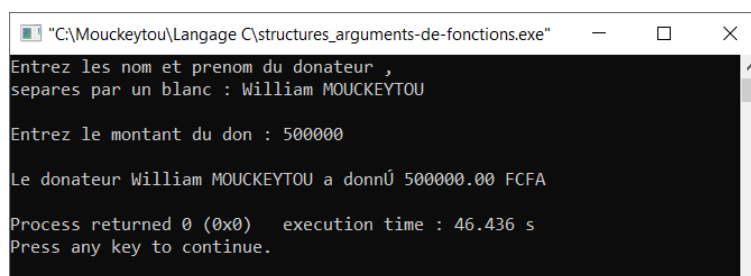


FIGURE 15.4 – Exécution de structures_arguments-de-fonctions.exe

15.3 Unions

Unions et *structures* sont similaires. Une **union** est déclarée et utilisée comme une structure, mais on ne peut travailler qu'avec un seul de ses membres à la fois. La raison en est simple, tous les membres d'une union sont stockés un par un dans le même emplacement mémoire

15.3.1 Définition, déclaration et initialisation des unions

Les unions sont définies et déclarées de la même façon que les structures, avec un mot clé différent. Sa syntaxe :

```
|| union{
|     type1 nom_membre1;
|     type2 nom_membre2;
|     ...
|     typen nom_membren;
| }variable;
```

Le mot clé **union** annonce la déclaration d'une union. Une union regroupe une ou plusieurs variables (membre(s)) sous un nom identique. Tous les membres de cette union occuperont le même emplacement mémoire.

L'instruction suivante définit une **union** simple d'une variable **int**, d'une variable **float** et d'une variable **char** :

```
|| union symbole{
|     int i;
|     float f;
|     char c;
| };
```

Ce modèle d'union, appelé *symbole*, va permettre de créer des unions qui pourront contenir soit un entier *i*, soit un réel *f*, soit un caractère *c*

L'**union** peut être initialisée par son instruction de déclaration. Un seul membre pouvant être utilisé à la fois, pour éviter les erreurs, seul le premier peut être initialisé. Voici un exemple de déclaration et d'initialisation d'une union de type *symbole* :

```
|| union symbole variable_generic = {'@'};
```

15.3.2 Accéder aux membres d'une union

On utilise les membres d'une union de la même façon que les membres de structure : avec l'opérateur (**.**). Il y a cependant une différence importante pour accéder aux membres d'une union. En effet, celle-ci sauvegarde ses membres à la suite les uns des autres, il est donc important d'y accéder un par un.

15.4 Structures et typedef

Vous pouvez utiliser le mot clé **typedef** pour créer le synonyme d'une structure ou d'une union. Les instructions suivantes, par exemple, définissent *Coord* comme synonyme de la structure désignée :

```
|| typedef struct coord{
|     int x;
|     int y;
| }Coord;
```

Vous pourrez ensuite déclarer des structures de ce type en utilisant *coord* :

```
|| struct coord hautgauche ,basdroit;      ou      || Coord hautgauche ,basdroit;
```

En pratique, l'une et l'autre peuvent être utilisés indifféremment. **typedef** donne un code un peu plus concis, mais l'utilisation du mot clé **struct** ne laisse aucun doute sur le type de variable qui a été déclaré.

Chapitre 16

Liste chaînées

Une liste chaînée est une méthode d'enregistrement des données que l'on peut facilement implémenter avec le langage C. Il existe plusieurs sortes de listes chaînées : les listes chaînées simples, les listes chaînées doubles, les arbres binaires... Chacun de ces types convient plus particulièrement à certaines tâches. Le point commun est que les liens entre les articles sont explicites et définis par des informations placées dans les articles eux-mêmes. C'est là une différence essentielle par rapport aux tableaux, dans lesquels les liens entre les articles sont implicites, et résultent de l'agencement général du tableau. Dans ce chapitre, nous allons expliquer les listes chaînées simples (que l'on appellera listes chaînées).

16.1 Principes de base des listes chaînées

Chaque élément de donnée d'une liste chaînée appartient à une structure (voir Chapitre 15). Celle-ci contient les éléments de données requis pour enregistrer les données spécifiques d'un programme. Cette structure contient un élément de donnée supplémentaire : un pointeur qui fournit les liens de la liste chaînée. Voici un exemple simple :

```
struct person{  
    char name [20];  
    struct person *next;  
};
```

Ces lignes de code définissent la structure *person*. Une structure de type *person* contient un ensemble de données, et qu'elle peut aussi pointer vers une autre structure *person*. La Figure 16.1 présente le chaînage des structures dans une liste.

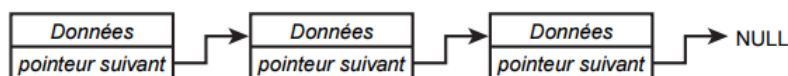


FIGURE 16.1 – Établissement du chaînage entre les éléments d'une liste chaînée

Remarquez que dans la Figure 16.1, chaque structure *person* pointe sur la structure *person* suivante. La dernière ne pointe sur rien. Pour concrétiser ce fait, on donne à son pointeur la valeur **NULL**.

Les structures constituant une liste chaînée sont appelées liens, noeuds, ou éléments d'une liste chaînée.

Vous accédez au premier élément d'une liste chaînée par l'intermédiaire d'un *pointeur de tête*, et cet élément contient un pointeur vers le deuxième. Ce deuxième élément contient un pointeur vers le troisième, et ainsi de suite jusqu'à ce que le pointeur rencontré ait la valeur **NULL**. Si la liste est vide (sans aucun lien), c'est le pointeur de tête qui a la valeur **NULL**.

Le pointeur de tête est un pointeur vers le premier élément d'une liste. On le désigne parfois sous le nom de "pointeur vers le sommet de la liste". La Figure 16.2 présente ce pointeur de tête lors de l'initialisation de la liste, puis après l'ajout du premier élément.

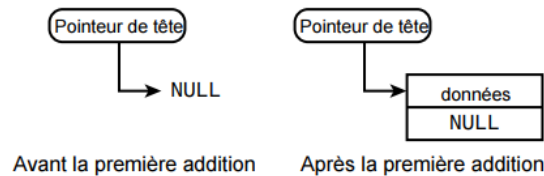


FIGURE 16.2 – Le pointeur de tête d’une liste chaînée

16.2 Utilisation de listes chaînées

Une liste chaînée, c’est un peu comme un fichier disque : des maillons peuvent être ajoutés, modifiés ou supprimés, mais il est indispensable d’ajuster les pointeurs de liaison (de chaînage) lorsqu’on exécute ce type d’opération. Nous allons étudier dans ce chapitre quelques opérations indispensables à la manipulation des listes chaînées en utilisant la structure *person* présentée plus haut.

16.2.1 Préliminaire

Pour commencer une liste chaînée, vous devez définir une structure de données et le pointeur de tête. Ce pointeur doit être initialisé avec la valeur nulle puisque la liste est vide au moment de sa création. Vous aurez besoin d’un pointeur supplémentaire vers le type de structure de la liste pour pouvoir ajouter des enregistrements (vous verrez plus loin que des pointeurs supplémentaires pourront être nécessaires). Voici comment procéder :

```
struct person{
    char name [20];
    struct person *next;
};
struct person *new;
struct person *head;
head = NULL;
```

16.2.2 Ajout d’un maillon dans une liste

Si le pointeur de tête a la valeur NULL, la liste est vide et le nouveau maillon sera l’unique élément de la liste. Si ce pointeur a une valeur différente, la liste contient déjà un ou plusieurs éléments. Dans tous les cas, la procédure pour ajouter un maillon est identique :

- Créez une structure en utilisant `malloc()` pour allouer la mémoire nécessaire.
- Définissez le pointeur du nouvel élément avec la valeur du pointeur de tête. Cette valeur sera nulle si la liste est vide ou égale à l’adresse du premier élément en cours.
- Modifiez la valeur du pointeur de tête avec l’adresse du nouvel élément.

Voici le code correspondant :

```
new = malloc(sizeof(struct person)); // Ou new = calloc(1, struct person);
new ->next = head;
head = new
```

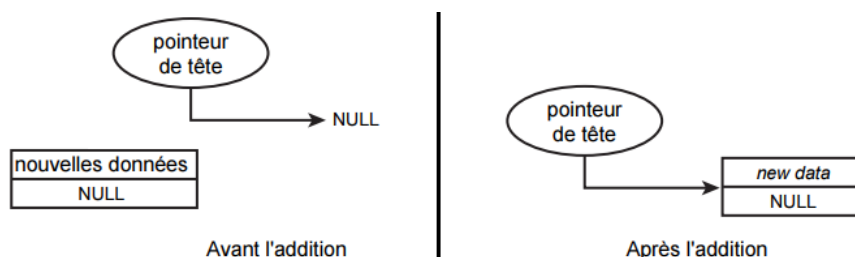


FIGURE 16.3 – Ajout d’un élément dans une liste vide

La Figure 16.3 présente l'ajout d'un maillon à une liste vide et la Figure 16.4 l'ajout du premier maillon à une liste existante.

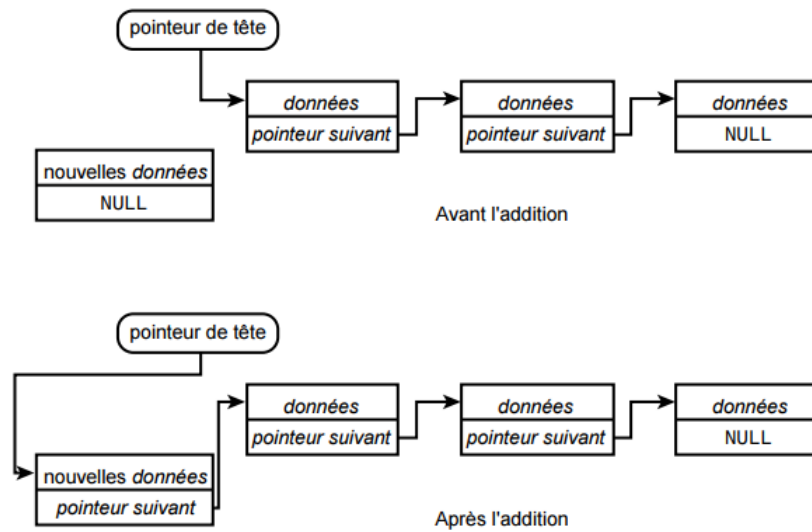


FIGURE 16.4 – Ajout dans une liste d'un nouveau premier élément.

16.2.3 Ajout d'un maillon en queue de liste

À partir du pointeur de tête, vous devez parcourir la liste pour retrouver le dernier élément. Suivez ensuite ces étapes :

1. Créez une structure en utilisant `malloc()` pour allouer la mémoire nécessaire.
2. Redéfinissez le pointeur du dernier élément pour qu'il pointe vers le nouvel élément (dont l'adresse a été renvoyée par `malloc()`).
3. Définissez le pointeur du nouvel élément avec la valeur `NULL` qui indique la fin de la liste

Voici le code correspondant :

```

person *current;
...
current = head;
while(current->next != NULL)
    current = current->next;
new = malloc(sizeof(struct person));
current->next = new;
new->next = NULL;

```

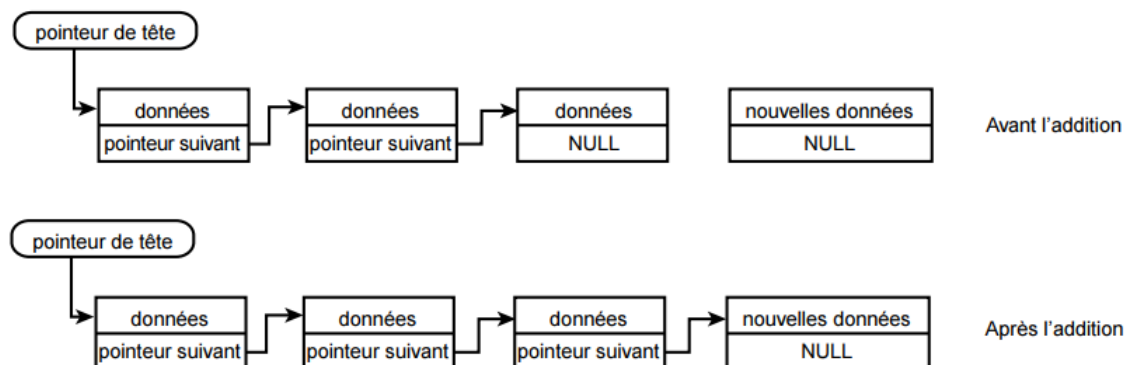


FIGURE 16.5 – Ajout d'un élément en queue de liste

16.2.4 Ajout d'un maillon au milieu

Lorsque l'on travaille avec une liste chaînée, on doit la plupart du temps ajouter des maillons quelque part entre le premier et le dernier. L'emplacement d'insertion exact varie en fonction de la gestion de la liste : elle peut être triée, par exemple, sur un ou plusieurs éléments de données. Vous devez donc d'abord vous placer au bon endroit de la liste avant d'effectuer l'ajout en suivant les étapes ci-après :

1. Localisez l'élément de la liste après lequel le nouveau maillon devra être inséré. Cet élément sera nommé élément de référence.
2. Créez une structure en utilisant `malloc()` pour allouer la mémoire nécessaire.
3. Modifiez le pointeur de l'élément de référence pour qu'il pointe vers le nouvel élément (dont l'adresse a été renvoyée par `malloc()`).
4. Définissez le pointeur du nouvel élément avec l'ancienne valeur de celui de l'élément de référence.

Voici le code correspondant :

```
person *marker;
/* Insérez ici le code nécessaire pour faire pointer l'élément de référence
   (marker) sur l'emplacement requis de la liste.*/
...
new = malloc(sizeof(struct person));
new->next = marker->next;
marker->next = new;
```

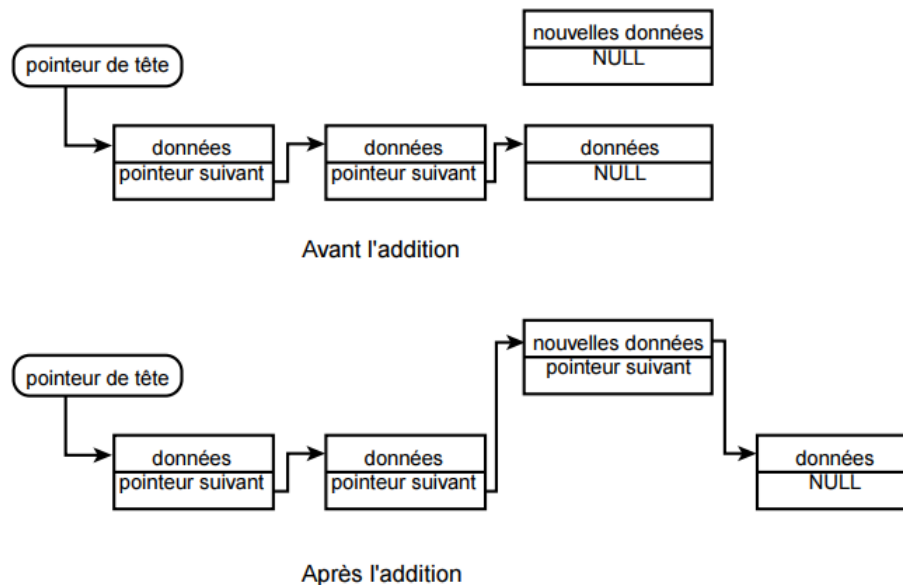


FIGURE 16.6 – Ajout d'un élément au milieu d'une liste chaînée.

16.2.5 Suppression d'un maillon d'une liste

La suppression d'un maillon se résume à un simple ajustement des pointeurs. Le processus varie en fonction de l'emplacement de l'élément :

- Pour supprimer le premier élément, il faut faire pointer le pointeur de tête vers le deuxième élément.
- Pour supprimer le dernier élément, il faut donner au pointeur de l'avant dernier élément la valeur NULL.
- Pour supprimer un maillon intermédiaire, il faut modifier le pointeur de l'élément qui le précède pour le faire pointer vers l'élément qui suit l'élément à supprimer.

Les lignes de code suivantes suppriment le premier élément de la liste chaînée :

```
|| head = head ->next;
```

Les lignes de code suivantes suppriment le dernier élément de la liste chaînée :

```
|| person *current1, *current2;
|| current1 = head;
|| current2= current1->next;
|| while (current2->next != NULL){
||     current1 = current2;
||     current2= current1->next;
|| }
|| current1 ->next = null;
|| if(head == current1)
||     head = null;
```

Enfin, les lignes de code suivantes suppriment un élément intermédiaire de la liste chaînée :

```
|| person *current1 , *current2;
|| /* Insérer ici le code nécessaire pour que current1 pointe vers l'élément
|| qui précède le maillon à supprimer. */
|| current2 = current1->next;
|| current1->next = current2->next;
```

Après l'exécution de ces lignes, l'élément supprimé est toujours en mémoire, mais il ne fait plus partie de la liste puisqu'il n'est plus "pointé". Dans vos programmes, n'oubliez pas de libérer la mémoire occupée par cet élément avec la fonction `free()` .

16.3 Un exemple de liste chaînée

Le programme suivant illustre les opérations de base du traitement d'une liste chaînée. Il n'a pas d'autre objectif que celui de vous présenter le code correspondant à ces opérations puisqu'il ne reçoit aucune donnée de la part de l'utilisateur et qu'il ne réalise aucune tâche particulière.

```
|| #include <stdio.h>
|| #include <stdlib.h>
||
|| //définition de la structure de donnée CelluleReel
|| typedef struct celluleReel{
||     float valeur;           //donnée valeur de la cellule
||     struct celluleReel *suivant; //donnée pointeur sur une cellule
|| }CelluleReel;
|| //définition de la Liste des cellules
|| typedef CelluleReel *Liste;
||
|| //Construit une Liste constituée de val et la liste L
|| Liste ConsListe(float val ,Liste L){
||     Liste L1=NULL;
||     L1 = malloc(sizeof(CelluleReel));
||     L1->valeur = val;
||     L1->suivant = L;
||     return L1;
|| }
||
|| //Ajoute la donnée val en tete de la liste L
|| Liste ajouterTete(float val ,Liste L){
||     L = ConsListe(val ,L);
||     return L;
|| }
```

```

//Ajoute en queue de la liste L, la donnée val
Liste ajouterQueue(float val ,Liste L){
    Liste R1=L, R2;
    R2 = ConsListe(val, NULL);
    while(R1->suivant != NULL)
        R1 = R1->suivant;
    R1->suivant = R2;
    R2 = NULL;
    return L;
}

//Ajoute un réel dans la liste triée L
Liste insererVal(float val ,Liste L){
    Liste R1,R2;
    if(L == NULL)
        L = ajouterTete(val ,L);
    else if(L->suivant == NULL)
        L = ajouterQueue(val ,L);
    else
    {
        R1 = L;
        R2 = R1 ->suivant;
        while(R2 ->valeur > val)
        {
            R1 = R2;
            R2 = R2 ->suivant;
        }
        R2 = ajouterTete(val ,R2);
        R1->suivant = R2;
    }
    return L;
}

//Affiche la liste L
void AfficherListe(Liste L){
    Liste R = L;
    printf("(");
    while(R != NULL)
    {
        if(R->suivant == NULL)
            printf("%.2f)\n",R->valeur);
        else
            printf("%.2f, ",R->valeur);
        R = R->suivant;
    }
}

int main(void){
    Liste L1=NULL ,L2=NULL ,L3=NULL;
    printf("Test des operations de base dans une liste chainee\n");
    printf("Ajout en tete de la valeur 12 dans la liste vide: ");
    L1 = ajouterTete (12,L1);
    AfficherListe(L1);
    L2 = ajouterTete (14,L1);
    printf("\nAjout en queue de la valeur 10 dans une liste non vide: ");
    L3 = ajouterQueue (10,L2);
    AfficherListe(L3);
    printf("\nInsere la valeur 11 dans une liste trieée precedente : ");
    L3 = insererVal (11,L3);
    AfficherListe(L3);
    exit(EXIT_SUCCESS);
}

```

Chapitre 17

Portée des variables

Dans ce chapitre, vous allez apprendre :

- La portée d'une variable et les raisons de son importance
- Les variables externes et les raisons pour lesquelles il vous faudra les éviter
- Les entrées/sorties des variables locales
- La différence entre variables statiques et variables automatiques

17.1 Définition de la portée et importance de cette notion

La notion de portée de la variable fait référence aux zones du programme dans lesquelles cette variable est connue, c'est-à-dire aux parties du programme où cette variable est *visible* ou *accessible*. Le terme variable utilisé tout au long de ce chapitre englobe tous les types de variables du langage C : variables simples, tableaux, structures, pointeurs, etc. Il représente aussi les constantes symboliques définies avec le mot clé `const`.

Le "temps de vie" de la variable, c'est-à-dire le temps pendant lequel la donnée est conservée en mémoire, dépend aussi de la portée de la variable.

L'importance de la notion de portée est liée au principe de la programmation modulaire. Cette méthode de programmation consiste à diviser le programme en fonctions *indépendantes*, chacune de ces fonctions réalisant une tâche spécifique. Le mot clé de cette définition est indépendance. Pour que cette indépendance soit réelle, la fonction doit posséder ses propres variables, sans possibilité d'interférences avec le reste du programme. La meilleure façon d'obtenir d'une fonction un résultat fiable est d'en isoler les données. Toutefois, dans certains cas, une isolation complète des données n'est pas souhaitable. Vous apprendrez vite, en tant que programmeur, à jouer sur la portée des variables pour contrôler le niveau "d'isolement" de vos données.

17.2 Variables externes

Une variable *externe* est définie en dehors de toute fonction, y compris de la fonction principale `main()`. Les variables externes sont aussi appelées variables *globales*. Si vous n'initialisez pas une telle variable lors de sa définition, le compilateur le fait de façon implicite avec la valeur 0.

17.2.1 Portée des variables externes

La portée d'une variable externe s'étend au programme tout entier. Elle peut donc être utilisée par la fonction `main()` ou toute autre fonction du programme. Il existe cependant une restriction. Cette définition n'est vraie que si le code source de votre programme est sauvegardé dans un fichier unique.

17.2.2 Quand utiliser les variables externes

Les variables externes sont rarement utilisées, car elles vont à l'encontre des principes d'indépendance de la programmation modulaire entre les différents blocs du programme. Chaque bloc ou fonction doit contenir le code et les données nécessaires à l'exécution de la tâche qui lui est confiée.

Vous pouvez utiliser une variable externe quand la plupart des fonctions du programme ont besoin d'accéder à une même donnée. Les constantes symboliques définies avec le mot clé `const` sont souvent utilisées de cette façon. Si la donnée ne doit être connue que d'un petit nombre de fonctions, il est préférable de la passer en argument.

17.2.3 Mot clé `extern`

Quand une fonction utilise une variable externe, il est bon de déclarer cette variable dans la fonction avec le mot clé `extern` :

```
|| extern type nom;
```

17.3 Variables locales

Une variable *locale* est une variable définie dans une fonction, et sa portée se limite à la fonction dans laquelle elle est définie. Contrairement aux variables globales, celles-ci ne sont pas initialisées par le compilateur lorsque vous omettez de le faire. Une variable locale qui n'a pas été initialisée contient une valeur indéterminée.

17.3.1 Variables statiques et automatiques

Par défaut, les variables locales sont automatiques. Cela signifie qu'elles sont créées et détruites avec l'appel et la fin de la fonction. En d'autres termes, la valeur de cette variable n'est pas conservée entre deux appels de la fonction dans laquelle elle est définie. Si la dernière valeur de la variable locale doit être connue à l'appel de la fonction, la variable doit être définie avec le mot clé `static`. Voici un exemple de définition d'une variable statique :

```
|| int fonction_etrange(void){  
||     static int val = 0;  
||     val ++;  
||     return val;  
|| }
```

17.3.2 Portée des paramètres d'une fonction

Les variables de la liste des paramètres d'une fonction ont une portée locale. Les variables paramètres étant toujours initialisées avec la valeur passée en argument par le programme appelant, les termes *statique* ou *automatique* n'ont pas de sens en ce qui les concerne.

17.3.3 Variables statiques externes

On peut donner le type statique à une variable externe en ajoutant le mot clé `static` dans sa définition :

```
|| static float taux;  
|| int main()  
|| {  
||     /* instructions ... */  
|| }
```

La différence entre une variable externe et une variable externe statique concerne la portée de ces variables. Une variable externe statique est visible par toutes les fonctions du fichier dans lequel elle se

trouve. Une variable externe simple est visible par toutes les fonctions du fichier et peut être utilisée par des fonctions dans d'autres fichiers.

17.3.4 Variables de type `register`

Le mot clé `register` est utilisé pour demander au compilateur de stocker une variable locale automatique dans un registre plutôt que dans un emplacement de la mémoire standard.

Le processeur central (CPU) de votre ordinateur contient quelques emplacements mémoire appelés registres. Ces registres sont utilisés pour des opérations comme l'addition ou la division. La CPU prend les données en mémoire, les copie dans les registres, réalise l'opération demandée, puis les replace en mémoire. Si une variable est enregistrée directement dans un registre, les opérations qui la concernent seront effectuées plus rapidement.

Le mot clé `register` ne devrait plus être utilisé pour les variables de programmes destinés à être utilisés sur les machines puissantes d'aujourd'hui. En effet, les processeurs sont devenus complexes et il vaut mieux faire confiance aux compilateurs qui savent mieux optimiser que la plupart des programmeurs.

17.4 Variables locales et la fonction `main()`

La fonction `main()` est appelée quand le programme commence son exécution, et rend le contrôle au système d'exploitation quand le programme est terminé.

Cela signifie qu'une variable locale définie dans `main()` est créée quand le programme commence, et qu'elle cesse d'exister quand le programme se termine. Pour cette fonction, la notion de variable locale statique n'a donc aucun sens. Une variable ne peut subsister entre deux exécutions du programme. Il n'y a donc aucune différence pour la fonction `main()` entre une variable locale statique et une variable locale automatique.

Chapitre 18

Retour sur les fonctions

Tout programme C contient au moins la fonction `main()`, qui est la première à être exécutée quand le programme démarre, toutes les autres fonctions n'étant que de sous-programmes.

La définition d'une fonction consiste en une liste d'instructions qui seront exécutées lors de son appel. Une fonction doit être déclarée avant de pouvoir être appelée dans une unité de traduction donnée. La définition d'une fonction fait aussi office de déclaration. Cette déclaration fournit au compilateur le type retour de la fonction. Par exemple :

```
|| extern double puiss();
```

puiss() est déclarée comme une fonction qui retourne une valeur de type `double`. Les noms de fonctions étant par défaut des noms externes, le spécificateur de classe de mémorisation `extern` peut aussi être omis. Comme vous avez dû vous en apercevoir, les fonctions constituent l'un des éléments les plus importants de la puissance et de la souplesse du langage C. Dans ce chapitre, nous allons voir les points suivants :

- Prototype de fonction
- Définition de fonctions
- Appels de fonction

18.1 Prototypes de fonctions

Comme vu dans les chapitres précédents, un prototype de fonction est une déclaration qui indique à la fois le type des paramètres et de valeur de retour d'une fonction. Par exemple :

```
|| double puiss(double, double); //prototype de puiss()
```

Ce prototype informe le compilateur que la fonction *puiss* accepte deux arguments de type `double`, et retourne un résultat de type `double`. Le type de chaque argument peut-être suivi d'un nom. Celui-ci n'aura toutefois de plus de valeur d'un commentaire, sa portée étant uniquement confinée au prototype de la fonction. Par exemple :

```
|| double puiss(double base , double exposant);
```

Les fonctions qui ne retournent aucune valeur sont déclarées de type `void`. Par exemple :

```
|| //fonc1 accepte une chaine comme argument et n'a pas de valeur de retour.  
|| void fonc1(char *str);
```

Pour déclarer une fonction qui n'a pas d'argument, on utilise le spécificateur de type `void` dans la liste des arguments :

```
|| //fonc2 ne prend aucun argument et retourne une valeur de type int  
|| int fonc2(void);
```

Les déclarations de fonctions devraient toujours se faire sous la forme de prototypes. Toutes les fonctions C standard sont définies dans plusieurs d'en-tête. Par exemple, `math.h` contient les prototypes de toutes les fonctions mathématiques – `sin`, `cos`, `pow`, etc.

18.2 Définitions de fonctions

La forme la plus générale d'une fonction est :

```
/* déclaration de fonction */
[classe_mémorisation] [type] nom([ liste_arguments ])
{
    /* déclaration , instructions */
}
```

L'un des spécificateurs de classe de mémorisation `extern` ou `static`. La classe de mémorisation par défaut des fonctions étant `extern`, la plupart des définitions de fonctions ne comportent pas de spécificateur de classe de mémorisation.

type : Le type de la valeur de retour de la fonction. Cela peut être `void` ou tout autre à l'exception d'un tableau.

nom : Le nom de la fonction

liste_arguments : Les déclarations aux arguments de la fonction. Si elle n'en possède pas, la liste sera vide. Voici un exemple de définition de fonction :

```
long somme(int tab[], int long) //Effectue la somme des
{
    int i;
    long resultat = 0;
    for(i=0; i<long; ++i)
        resultat += (long)tab[i];
    return resultat;
}
```

par défaut, les noms de fonctions sont des noms externes ; les fonctions d'un programme peuvent ainsi être réparties parmi plusieurs fichiers source et apparaître n'importe où dans ces fichiers.

Toutefois, les fonctions de la classe `static` ne peuvent être appelées qu'à partir de l'unité de traduction dans laquelle elles sont définies. Mais il n'est pas possible de définir des fonctions qui ont le bloc comme portée – en d'autres termes, on ne peut pas définir une fonction à l'intérieur d'une autre fonction.

Les instructions du corps de la fonction précisent ce que doit faire la fonction. Lorsque, au cours de l'exécution, le programme rencontre une instruction `return` ou la fin du corps de la fonction, le contrôle est rendu à la fonction appelante.

On appelle fonction récursive, une fonction qui s'appelle elle-même, que ce soit directement ou indirectement. Le langage C permet la définition de fonctions récursives, les variables de classe de mémorisation automatique sont recréées (généralement dans la pile) à chaque appel de la fonction.

En ANSI C99, les fonctions peuvent être définies comme `inline` (en ligne). Le spécificateur de fonction `inline` demande au compilateur d'optimiser la vitesse de l'appel de fonction, en plaçant généralement le code machine de la fonction dans la procédure d'appel. Le mot-clé `inline` est placé devant la définition de la fonction :

```
inline int max(int x, int y)
{
    return (x >= y ? x : y);
}
```

Dans le cas où une fonction en ligne contient trop d'instructions, le compilateur ignorera le spécificateur `inline` et générera un appel standard de fonction.

Une fonction *en ligne* doit être définie et appelée dans la même unité de traduction. En d'autres termes, le corps de la fonction doit être « visible » lorsque « l'appel en ligne » est compilé. C'est plutôt une bonne idée que de définir – contrairement aux fonctions ordinaires – les fonctions en ligne dans un fichier d'en-tête.

18.3 Appels de fonction

Un appel de fonction est une expression dont la valeur et le type sont ceux de la valeur de retour d'une fonction.

Lors de l'appel d'une fonction, le nombre et le type des arguments doivent coïncider avec ceux de la définition de la fonction. Toute expression, y compris les constantes et les expressions arithmétiques, peut-être donnée comme argument dans un appel de fonction. Lorsque la fonction est appelée, la *valeur* de l'argument est *copiée* dans l'argument correspondant de la fonction ! Par exemple :

```
//Déclaration
double x = 0.5, y, puiss ();
//L'appel de puiss retourne la valeur double (1.0+x) ^2.5
y = puiss (1.0 + x, 2.5);
```

18.3.1 Passage de pointeurs à une fonction

La méthode par défaut pour passer un argument à une fonction est un passage par valeur, ce qui signifie que la fonction reçoit une copie de la valeur de l'argument. Cette méthode se réalise en trois étapes :

1. L'expression à passer en argument est évaluée (ou prise telle quelle s'il s'agit d'une constante ou d'une variable).
2. Le résultat est recopié sur la pile (stack) qui est une zone de stockage temporaire en mémoire
3. La fonction récupère la valeur de l'argument sur la pile.

Lorsqu'une variable est passée à une fonction par valeur, la fonction a accès à la valeur de la variable, mais pas à la variable elle-même. Il en résulte que la fonction ne peut pas modifier la valeur originale de cette variable. C'est la principale raison d'utiliser un passage d'argument par valeur. Les données situées à l'extérieur d'une fonction sont protégées de toute altération accidentelle.

Si l'on désire qu'une fonction modifie la valeur d'une valeur, la fonction appelante doit passer en argument à la fonction, l'*adresse* de cette variable : la variable doit être passée à la fonction par *référence*. Par exemple :

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

void par_valeur(int a, int b, int c);
void par_adresse(int *a, int *b, int *c);

int main()
{
    int x = 2, y = 4, z = 6;
    printf("\nAvant d'appeler par_valeur() , x=%d, y=%d, z=%d.", x, y, z);
    par_valeur(x, y, z);
    printf("\nAprès avoir appele par_valeur() , x=%d, y=%d, z=%d.", x, y, z);
    par_adresse (&x, &y, &z);
    printf("\nAprès avoir appele par_adresse() , x=%d, y=%d, z=%d.\n", x, y, z);
    exit(EXIT_SUCCESS);
}
```

```

void par_valeur(int a, int b, int c)
{
    a = 0;
    b = 0;
    c = 0;
}

void par_adresse(int *a, int *b, int *c)
{
    *a = 0;
    *b = 0;
    *c = 0;
}

```

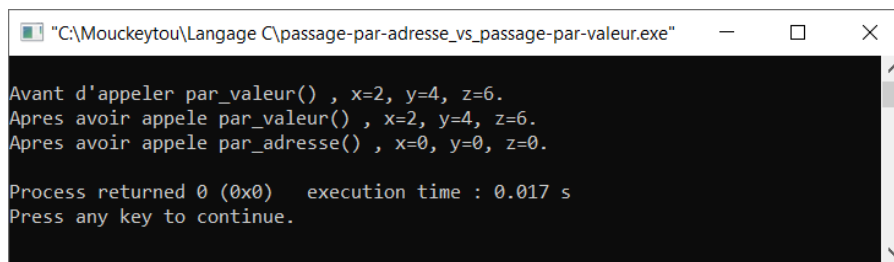


FIGURE 18.1 – Exécution de passage-par-adresse_vs_passage-par-valeur.exe

Passer un argument par adresse est à la fois un avantage et un inconvénient.

Si le programme a besoin de modifier la valeur de la variable, alors, c'est un avantage. Si ce n'est pas le cas, cela peut dégénérer en inconvénient.

Il est possible de mélanger les deux types de passage d'arguments dans un même appel de fonction. Il faudra bien faire attention, en appelant plus tard la fonction, à ne pas les mélanger.

Conseils : Passer des variables par valeur pour ne pas risquer de les voir modifiées par la fonction appelée.

18.3.2 Fonctions renvoyant un pointeur

Vous avez déjà rencontré, dans la bibliothèque standard, des fonctions renvoyant un pointeur au programme qui les a appelées. Rien ne vous empêche d'écrire de telles fonctions. Comme vous pouvez vous y attendre, l'opérateur d'indirection (`*`) est utilisé à la fois dans la déclaration de la fonction et dans sa définition. Voici quelle en est la forme générale :

```

|| type *fonc(liste d'arguments);

```

Cette instruction déclare une fonction `fonc()` qui renvoie un pointeur vers un type. En voici deux exemples concrets :

```

|| double *fonc1(liste d'arguments);
|| struct adresse *fonc2(liste d'arguments);

```

La première ligne déclare une fonction qui renvoie un pointeur vers un type `double`. La seconde ligne déclare une fonction qui renvoie un pointeur vers un type `adresse` (supposé défini par l'utilisateur).

Ne confondez pas *une fonction qui renvoie un pointeur avec un pointeur vers une fonction* . Si vous mettez une paire de parenthèses supplémentaire dans la déclaration, c'est la dernière forme que vous déclarez, comme on le voit dans ces deux exemples :

```

|| double (*fonc)(...); //pointeur vers une fonction qui renvoie un double
|| double *fonc (...); // fonction qui renvoie un pointeur vers un double

```

Maintenant que nous savons déclarer correctement notre fonction, une question se pose. Comment allons- nous l'utiliser ? Il n'y a rien de spécial à préciser au sujet de ces fonctions : on les utilise comme n'importe quelle autre fonction, en assignant leur valeur de retour à une variable du type approprié (donc un pointeur). Comme, en C, un appel de fonction est une expression, vous pouvez vous en servir à n'importe quel endroit de votre programme.

Bibliographie

- [1] Peter Aitken et Bradley L. Jones, *Le langage C*. Pearson Education France, Paris, Nouvelle édition française revue, corrigée et complétée par Yves Mettier, 2009.
- [2] Peter Prinz et Ulla Kirch-Prinz, *C – précis & concis*. Editions O'reilly, Paris, 2003.