Université Hassan 1er

Faculté des Sciences et Techniques de Settat

COURS DE PROGRAMMATION EN «C» ET STRUCTURES DE DONNÉES

Professeur Laachfoubi Nabil

Département des Mathématiques et Informatique

Sommaire

A. Introduction

A.1. Présentation du langage C

Le langage «C» est un langage de programmation qui a été développé - en 1972 dans les laboratoires informatiques de la multinationale «Bell» - par les chercheurs Dennis Ritchie et Ken Thompson, dans le but d'écrire le système d'exploitation Unix. L'objectif principal de ce langage était d'être portatif1. Depuis, plusieurs évolutions et standards sont apparus, ce qui a mis cette portabilité à rude épreuve. On distingue trois versions majeures :

- La version Kernighan & Ritchie ou encore C K&R, stabilisée en 1978 (version K&R),
- La version C89 ou encore C ANSI, stabilisée 1989 (version standard ANSI),
- La version C99, stabilisée en 1999 (version standard ISO10),

Le «C» est un langage presque de bas niveau car à la compilation, il utilise directement en langage assembleur, ce qui lui confère une puissance et une rapidité d'exécution beaucoup plus élevées que la plupart des langages récents.

A.2. Quelques définitions importantes et culture générale

Dans ce qui suit, nous allons donner un certain nombre de définitions qui constituent la base des termes utilisés dans la programmation en informatique. Certains termes seront détaillés plus amplement en cours, tandis que d'autres seront repris et mieux illustrés à travers des exemples sur le présent support.

Mot clé :

Chaque langage de programmation possède un certain nombre de mots clés lui permettant de réaliser des actions particulières.

Instruction :

C'est une action informatique compréhensible par l'ordinateur et éventuellement par le programmeur.

Jeu d'instructions :

C'est ensemble de mots clés spécifiques à un langage de programmation.

Bloc d'instructions :

En langage C, il s'agit d'un ensemble d'instructions délimitées par des accolades ouvrantes et fermantes.

Imbrication des instructions :

Fait référence à l'utilisation d'un ou de plusieurs blocs d'instructions à l'intérieur d'autres blocs d'instructions.

Condition :

Correspond à une expression ou un ensemble d'expressions dont la valeur de l'évaluation ne peut être que l'une des valeurs vrai ou faux. Il s'agit en fait de répondre à une question ou à un ensemble de questions par le terme vrai ou bien faux.

Boucle :

Il s'agit d'une instruction du langage de programmation permettant de répéter un bloc d'instructions autant de fois que nécessaire, en fonction d'un certain nombre de conditions. Tant que ces conditions sont vraies, le

[1] Un langage est dit portatif ou portable, s'il peut être compilé sur des machines et des systèmes différents sans être adapté.

traitement va être ré-exécuté. Si les conditions ne sont plus vérifiées, le traitement répétitif s'arrête pour donner la main à la suite du programme. Il existe plusieurs manière de formuler des boucles comme les instructions «for», «while», ou encore «do while».

Itération d'une boucle :

Terme utilisé pour faire référence à un cycle complet d'une boucle. Le cycle comprend la phase de vérification des conditions, et la phase d'exécution du bloc d'instruction (sujet à répétition).

Boucle infinie :

Se dit sur une instruction de boucle dont les conditions seront toujours évaluées à vraie, ce qui va provoquer un nombre interminable de répétitions. En général, le système d'exploitation va déceler ce comportement anormal, et va stopper l'exécution du programme à l'origine de cette anomalie.

Algorithme :

C'est un ensemble d'instructions - compréhensibles par l'être humain, et non compréhensibles par la machine - exécutées dans un ordre précis pour résoudre un problème donné.

Algorithme de tri :

Algorithme permettant d'ordonner une liste de valeurs.

Complexité d'un algorithme :

C'est le nombre d'opérations élémentaires nécessaires pour résoudre un problème donné.

Algorithmique :

C'est l'étude des algorithmes.

Code binaire :

C'est un ensemble d'informations exprimées en langage machine, et pouvant être interprétées directement par le processeur.

Programme :

C'est un ensemble d'instructions écrites dans un premier temps dans un fichier ou un ensemble de fichiers, avec un langage de programmation compréhensible à la fois par le programmeur et par l'ordinateur. Ces instructions sont traduites ensuite par ce dernier code binaire de manière à être exécutées par le processeur.

Code source d'un programme :

C'est un ensemble d'instructions exprimées avec un langage de programmation compréhensible à la fois par le programmeur et par l'ordinateur, mais ne pouvant pas être exécutées directement par le processeur.

Portabilité d'un code source :

Elle est associée au fait que le code soit compilable sur des machines et sur des systèmes d'exploitation différents sans subir de modifications permettant de l'adapter.

Lisibilité d'un code source :

Il s'agit d'une caractéristique que chaque code source doit posséder afin d'être facilement compris par un programmeur qui ne l'a pas forcément écrit. Ecrire un code lisible suppose la mise en place et le respect d'un certain nombre de règles de bonnes pratiques.

Indentation du code source :

Il s'agit d'un décalage qu'un programmeur insère en début de ligne pour aligner horizontalement certaines instructions afin de mettre en évidence leur appartenance à un block ou à une hiérarchie donnée.

Module :

En langage «C», il s'agit d'un couple de fichiers portant le même nom mais avec des extensions différentes. Le premier fichier a une extension «.c» et contient le code source de toutes les procédures et fonctions ayant un point commun et traitant d'un problème particulier. Le second fichier a une extension «.h» et regroupe uniquement les prototypes des fonctions, procédures, variables et constantes qu'on souhaite rendre visible pour les autres fichiers afin qu'ils puissent les réutiliser à volonté au lieu de les reprogrammer.

Projet :

En langage «C», il s'agit d'un ensemble de fichiers source et/ou de modules qui sont regroupés au sein d'une même entité pour qu'ils soient compilés ensembles, afin de donner lieu à un seul programme sous forme binaire directement exécutable par le processeur.

Opérateurs :

Ce sont des symboles utilisés en programmation pour effectuer des opérations de type arithmétiques, logiques, de comparaison ou d'incrémentation/décrémentation.

Opérandes :

Ce sont les paramètres des opérateurs.

Arité d'un opérateur :

Correspond au nombre de paramètres d'un opérateur. L'arité peut être de type unaire (un seul paramètre), de type binaire (deux paramètres), de type ternaire (trois paramètres), etc.

Priorité des opérateurs :

Correspond à l'ordre dans lequel les opérateurs vont être évalués lorsque ces derniers figurent dans une même expression.

Opérateurs arithmétiques :

Ce sont des symboles permettant d'effectuer des opérations arithmétiques comme (*, +, -, /)

Opérateurs logiques :

Ce sont des symboles permettant d'effectuer des opérations logiques comme (&&, ||, !). Ces opérateurs permettent de combiner plusieurs tests logiques pour s'assurer qu'un ensemble de conditions sont vérifiées ou non.

Opérateurs de comparaison :

Ce sont des symboles permettant d'effectuer des opérations de comparaison comme (>, >=, <, <=, ==)

Opérateurs d'incrémentation / décrémentation :

Ce sont des symboles permettant de rajouter une unité ou de réduire une unité à une variable donnée comme (++,--), (exemple : int a = 10 ; int b = ++a ;). La première instruction va attribuer à la variable «a» la valeur «10». La seconde instruction va dans un premier temps augmenter la variable «a» de «1» avant d'attribuer la nouvelle valeur de «a» c.à.d. «11» à la variable «b».

Caractères imprimables :

Se dit sur les caractères que l'ordinateur peut afficher à l'écran ou à travers une imprimante sur du papier.

Caractères non-imprimables :

Se dit sur les caractères que l'ordinateur ne peut pas afficher à l'écran ou à travers une imprimante sur du papier. Ils ne représentent pas des caractères visibles mais plutôt des commandes comme l'escape ou encore les séquences de de contrôle comme Ctrl+C, Ctrl+V, etc.

Code ASCII d'un caractère :

Se dit sur le code - en base 10 ou en base 16 - correspondant à la conversion du signal électrique envoyé par une touche clavier au système d'exploitation. En effet, chaque caractère - imprimable ou pas - utilisé par l'ordinateur possède un numéro permettant de l'identifier. D'ailleurs, même si on a l'impression que l'ordinateur reconnait les caractères directement, en réalité l'ordinateur ne travaille qu'avec leur code.

Chaîne de caractères :

Suite continue de caractères formant du texte.

Caractère de fin de chaîne :

Caractère non-imprimable noté '\0' et utilisé en langage C pour marquer la fin d'une chaine de caractères.

Identificateur ou identifiant :

Il s'agit d'un nom que le programmeur attribue à un fichier, à une fonction, à une procédure, à une variable, à une constante, ou de façon générale, à tout élément d'un programme pour pouvoir l'identifier de façon unique dans le contexte où il est défini.

Constante :

Se dit sur un identificateur associé à une zone mémoire allouée par le système d'exploitation pour un type de donnée spécifique, et dont la valeur ne peut être modifiée durant l'exécution du programme.

Variable :

Se dit sur un identificateur associé à une zone mémoire allouée par le système d'exploitation pour un type de donnée spécifique, et dont la valeur peur être modifiée durant l'exécution du programme.

Valeur ou constante littérale :

Il s'agit d'une valeur formulée explicitement dans le code source d'un programme. Elle n'est pas associée à un identificateur comme c'est le cas pour les variables ou les constantes.

Déclaration d'une variable :

Consiste à spécifier la nature de l'information qu'on est censé associer à un identificateur afin que le système d'exploitation puisse lui réserver l'espace mémoire nécessaire. Cette action permet également au système d'exploitation de définir la manière avec laquelle il doit coder ou décoder l'information associée à cet identificateur lors des opérations d'écriture ou de lecture de cette zone mémoire.

Initialisation d'une variable ou d'une constante :

Se dit sur la toute première opération d'affectation d'une valeur à la variable ou à la constante. Notons que dans le cas des constantes les opérations de déclaration et d'initialisation doivent se faire dans une même instruction (exemple : const int a = 10 ;). En ce qui concerne les variables, ces deux opérations peuvent éventuellement être différées (exemple int a ; a = 10 ;).

Portée d'une variable :

Correspond à la portion du code source dans laquelle la variable est visible et accessible.

Fonction :

C'est un bloc d'instructions identifié par un nom unique (identificateur), par des paramètres en entrées (non obligatoires), entourés de deux parenthèses, une ouvrante et l'autre fermante, et par une valeur de sortie typée.

Procédure :

C'est un bloc d'instructions identifié par un nom unique, par des paramètres en entrées (non obligatoires), entourés de deux parenthèses, une ouvrante et l'autre fermante, mais ne retournant aucune valeur.

Corps de la fonction ou de la procédure :

Représente le bloc d'instructions - entouré par des accolades - que doit réaliser la fonction ou la procédure.

Prototype de la fonction ou de la procédure :

Représente la suite d'instructions permettant de déclarer la fonction ou la procédure, à savoir, le type de retour (**void** pour les procédures et autre type que **void** pour les fonctions), l'identificateur, et éventuellement la liste des paramètres entourée par les parenthèses ouvrante et fermante.

Fonction principale ou fonction main :

Représente le point d'entrée d'un programme. Elle est obligatoire, et ne doit être déclarée qu'une seule fois dans un programme, même ce dernier est constitué de plusieurs fichiers.

Paramètre formel d'une procédure ou d'une fonction :

Fait référence à la variable déclarée dans le prototype de la procédure ou de la fonction.

Paramètre effectif d'une procédure ou d'une fonction :

Fait référence à la valeur ou à l'expression passée à la procédure ou à la fonction lors d'un appel.

Fonctions ou procédures variadiques :

Se dit sur les fonctions ou procédures d'arité indéfinie. En d'autres termes, il s'agit de fonctions ou de procédures dont le nombre de paramètres n'est pas fixe.

Fonctions ou procédures récursives :

Ce terme est associé aux fonctions ou aux procédures qui s'appellent elles-mêmes. A titre d'exemple, citons la fonction factorielle : Fact(n) = n! = nx(n-1)x(n-2)x ….x3x2x1 = nx(n-1)! = nxFact(n-1).

Récursivité terminale / non-terminale :

La récursivité terminale est dite sur les appels récursifs qui s'arrêtent à un moment donné suite à une condition qui n'est plus vérifiée. Elle est appelée non-terminale dans le cas contraire. Lorsque la récursivité dépasse un certain nombre d'appels le programme s'arrêter suite à un débordement de capacité.

Bibliothèque :

Correspond à un ensemble de fonctions et/ou de procédures traitant d'un thème particulier et pouvant être réutilisées pour créer d'autres programmes. Il existe deux types de bibliothèques. Celles livrées en standard avec le langage de programmation. Puis, celles développées par le programmeur lui-même ou par un autre programmeur.

Compilation :

Elle consiste à convertir le code source d'un programme en un code binaire exécutable, compréhensible par le processeur.

Erreur de compilation :

Elle est associée au fait de détecter une anomalie dans le code source d'un programme.

Erreur d'exécution :

Elle est associée au fait de détecter une anomalie lors de l'exécution d'un programme.

Adresse mémoire :

La mémoire vive d'un ordinateur peut être assimilée à une matrice de 8 colonnes (8 bits) et «N» lignes (N octets). «N» correspond donc à la taille de la barrette mémoire exprimée en octet (byte). Les lignes sont numérotées de 0 à N-1. Cette numérotation est à l'origine de l'adressage mémoire. En résumé, l'adresse mémoire d'une variable n'est ni plus, ni moins que le numéro associé au premier octet de cette zone mémoire.

Adresse d'une variable :

Correspond à l'adresse du premier octet de la zone mémoire allouée par le système d'exploitation pour cette variable.

Pointeur :

Se dit sur une variable qui contient l'adresse mémoire d'une autre variable.

Taille mémoire d'un type de données :

Représente le nombre d'octets nécessaires pour stocker les valeurs associées à un type de données.

Type de données simple :

Il s'agit d'un type de données ne contenant qu'une information atomique (entier, réel, caractère, …).

Type de données multidimensionnel :

Il s'agit d'un type de données pouvant contenir plusieurs informations atomiques de même nature (liste d'entiers, liste de réels, liste de caractères, …). Les valeurs de ce type peuvent être rangées dans un vecteur (tableau monodimensionnel), dans une matrice (tableau bidimensionnel), dans un cube (tableau tridimensionnel), et ainsi de suite.

Type de données complexe ou structure de données :

Il s'agit d'un type de données composé de plusieurs informations pouvant être de même nature ou de natures différentes (types simples, multidimensionnelles, ou même complexes). On parle alors de structure de données, ou de type de données personnalisé.

Domaine de définition d'un type de données simple :

Etant donné que le système d'exploitation alloue automatiquement pour chaque type de données simple un nombre d'octets limités, le nombre de valeurs possibles pouvant être stockées dans cette espace mémoire est également limité. La plus petite et la plus grande valeur représentent respectivement les limites inférieures et supérieures du domaine de définition associé à ce type.

Dépassement de capacité :

Ce phénomène se produit lorsqu'un programme essaie de stocker dans l'espace mémoire réservé à un type de données, une valeur qui se situe en dehors de son domaine de définition.

Conversion de type de données :

Consiste à changer le type de données d'une valeur ou d'une expression.

Conversion implicite de type de données :

C'est une opération de conversion qui s'effectue automatiquement par le compilateur sans aucune spécification particulière du programmeur (exemple : float a = 5). Bien que la valeur «5» soit une valeur entière, le compilateur va la stocker sous forme réelle dans la zone mémoire associée à la variable «a».

Conversion explicite de type de données :

C'est une opération de conversion qui s'effectue suite à une demande formulée par le programmeur grâce à une instruction de transtypage (exemple : int a = (int) 7.82). Dans la variable «a» ne sera stocké sous forme binaire que la partie entière de la valeur «7.82» c.à.d. la valeur «7».

Allocation mémoire :

Consiste à réserver de l'espace mémoire - à travers le système d'exploitation - pour répondre à un besoin particulier d'un programme.

Allocation mémoire implicite :

Se dit lorsque le système d'exploitation alloue de la mémoire automatiquement. Cela arrive lorsque dans un programme on déclare des constantes ou des variables atomiques ou scalaires.

Allocation mémoire explicite :

Se dit lorsque le système d'exploitation alloue de la mémoire suite à une demande explicite du programme. Cela arrive lorsque dans un programme on déclare par exemple des tableaux, des matrices, ou lorsqu'on fait appel à des fonctions d'allocation mémoire comme malloc, realloc, calloc.

Mémoire statique :

Se dit sur une zone mémoire qui a été allouée par le système d'exploitation pour répondre aux besoins d'un programme. Le contenu de cette zone mémoire est accessible en mode lecture et écriture. Cependant, la taille de cette zone mémoire doit être connue avant même de compiler le programme, et ne peut être réajustée (taille fixe) durant l'exécution de ce dernier.

Mémoire dynamique :

Se dit sur une zone mémoire qui a été allouée par le système d'exploitation pour répondre aux besoins d'un programme. Le contenu de cette zone mémoire est accessible en mode lecture et écriture. La taille de cette zone mémoire peut être réajustée en fonction des besoins du programme.

Désallocation mémoire :

Consiste à restituer au système d'exploitation un espace mémoire qui a été alloué par ce dernier afin de répondre à un besoin spécifique d'un programme. Cette libération de mémoire peut se faire automatiquement de façon implicite lorsque l'allocation a été effectuée de façon statique ou implicite, ou alors de façon explicite - avec la fonction free - lorsque l'allocation mémoire a été effectuée de façon explicite avec les fonctions d'allocation dynamique comme «malloc», «realloc» ou encore «calloc».

Mémoire contigüe :

Fait référence à un espace mémoire continue. En d'autres termes, il s'agit d'un espace mémoire formé par des octets adjacents les uns aux autres et dont les adresses se succèdent de façon continue.

Tableau :

C'est une variable dont l'espace mémoire associé est alloué de façon contigüe. Cet espace permet de stocker plusieurs valeurs de même type de données. Le contenu de cette variable n'est ni plus, ni moins que l'adresse mémoire du premier octet de cet espace.

Adresse d'un tableau :

Est l'adresse mémoire du premier octet, de la première valeur du tableau.

Taille d'un tableau :

Représente le nombre d'éléments maximum que ce dernier peut stocker. On parle également du nombre de cases mémoire du tableau (sachant qu'une case mémoire peut porter sur plusieurs octets en fonction du type de données considéré).

Indice d'un élément du tableau :

Il ne s'agit pas d'une adresse mémoire, mais d'un numéro associé à la case mémoire où se trouve l'élément en question. Pour mieux comprendre, considérons un tableau de cinq entiers (un entier est codé sur quatre octets sur un système 32 bits), et que l'adresse mémoire du premier octet de cet espace est l'adresse alpha. Ce tableau contient donc cinq cases faisant chacune quatre octets. Ces cases sont numérotées de zéro à quatre (0 à 5-1 → 4). Les valeurs entières de 0 à 4 sont donc appelées «indices du tableau».

Débordement dans un tableau :

Ce phénomène se produit lorsque le programme tente d'accéder à un indice situé en dehors du tableau.

B. Les types de données

En informatique, toutes les informations que l'ordinateur manipule seront un moment ou un autre stockées au niveau de la mémoire volatile «RAM». La taille mémoire nécessaire ainsi que la manière avec laquelle ces informations seront stockées dépendent en réalité de la nature de ces informations (des entiers, des réels, du texte, ...).

B.1. Unités de mesure de la mémoire

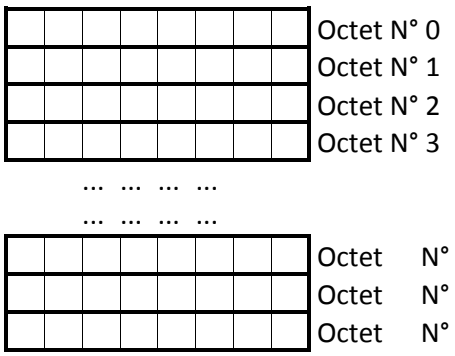
Bien que la plus petite unité au niveau de la mémoire soit le bit, la plus petite unité exploitable par un ordinateur reste l'octet (8 bits). Quel que soit la nature de l'information que l'ordinateur doit stocker, la taille de la mémoire qui va être allouée sera un multiple d'un octet. En d'autres termes, l'ordinateur ne peut allouer que des multiples de 8 bits (1 octet, 2 octets, 3 octets, etc.).

Unité	Symbole	Taille en octet
Kilo-octet	Ko	2^{10} octet
Méga-octet	Mo	2^{20} octet
Giga-octet	Go	2^{30} octet
Terra-octet	To	2^{40} octet
Péta-octet	Po	2^{50} octet

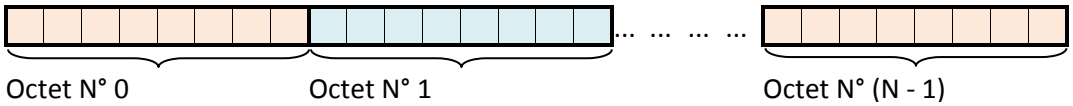
Unité	Symbole	Conversion
Kilo-octet	Ko	2^{10} Oct = 1024 Octet
Méga-octet	Mo	2^{10} Ko = 1024 Ko
Giga-octet	Go	2^{10} Mo = 1024 Mo
Terra-octet	To	2^{10} Go = 1024 Go
Péta-octet	Po	2^{10} To = 1024 To

B.2. Structure de la mémoire d'un ordinateur

La mémoire «RAM» d'un ordinateur se présente sous forme d'une matrice à huit colonnes par «N» Lignes. Les huit colonnes représentent les huit bits d'un octet, chaque ligne de cette matrice représente donc un octet. Le «N», représente le nombre d'octets de cette mémoire, autrement dit, il représente la taille mémoire de la «RAM» exprimée en octet. Ces ligne sont numérotées de zéro à «N-1». Ces numéros sont appelés adresses mémoires. Chaque octet possède donc sa propre adresse mémoire. Cet adressage représente la clé de voute pour la manipulation de cette mémoire.
Une autre représentation, consiste à considérer la mémoire «RAM» comme étant un vecteur de bits où chaque groupement de huit bits forme un octet, et à chaque octet on associe une adresse.



Présentation de la mémoire sous forme d'une matrice de «N» octets



Présentation de la mémoire sous forme d'un vecteur de «N» octets

A titre d'exemple : si une barrette mémoire a une taille de 4 Go c.à.d. $2^2 \times 2^{30} = 2^{32}$ octets, les adresses mémoires des octets sont les valeurs entières appartenant à l'intervalle semi ouvert $[0 , N = 2^{32}[$.

B.3. Codage binaire de l'information

Toutes les informations manipulables par l'ordinateur sont dans un premier temps, codées en base deux avant d'être stockées en mémoire. La manière avec laquelle ce codage va s'opérer dépend en fait de la nature de cette information (caractère, entier, réel, etc.).

B.3.a) Codage d'un nombre entier

Prenons par exemple la déclaration suivante en langage C «int a = 36». Avec un système d'exploitation 32 bits, l'ordinateur va allouer de façon implicite quatre octets pour la variable «a», dans lesquels il va stocker la conversion en binaire de la valeur décimale $(36)_{10}$.

Voici donc les étapes de cette conversion :

1. On effectue une division entière de «36» par «2»,
2. On effectue successivement des divisions entières du quotient par «2» jusqu'à obtention d'un quotient égale à zéro.
3. Le regroupement des restes des divisions entières dans le sens inverse des opérations forme le codage binaire de la valeur «36».

36	2	18	2	9	2	4	2	2	2	1	2
0	18	0	9	1	4	0	2	0	1	1	0

$(36)_{10} = (100100)_2 = (00000000000000000000000000\underline{100100})_2$

Notons que pour coder la valeur «36» on a besoin uniquement de six bits, mais en réalité le résultat sera complété par «26» bits auxquels le système va attribuer la valeur zéro, pour avoir un codage de la valeur entière sur 32 bits (4 octets).

Dans le cas général, en base «b», un nombre entier positif «X» codé sur «n» bits est représenté par une suite de chiffres $a_{n-1} \dots a_1\ a_0$ et donné par la formule suivante :

$$X = (a_{n-1} \dots a_1\ a_0)_b = \sum_{k=0}^{n-1} a_k * b^k$$

a_0 est le chiffre de poids faible, et a_n le chiffre de poids fort.

Lorsqu'on a affaire à des nombre entiers qui peuvent être positifs en négatifs, le bit «a_n» de poids fort est réservé pour signaler le signe de la valeur «X». Ce bit est assigné à zéro dans le cas d'un nombre positif, et à un dans le cas d'un nombre négatif. Par conséquent, il ne reste plus que «n-1» bits pour coder la valeur associée à «X».

A tire d'exemple, si on souhaite coder un entière positif sur un octet (8 bits), on pourra code 2^8 = 256 valeurs de 0 à 255. En effet, un bit permet de coder deux valeurs zéro ou un (2^1), deux bits permettent de coder 4 valeurs (2^2) , trois bits permettent de coder 8 valeurs (2^3), etc..

Nbr. de bits	Nbr. de valeurs	Liste des valeurs possibles en base 2	Domaine en base 10
1	$2 = 2^1$	{0,1}	[0 1] = [0 2^1-1]
2	$4 = 2^2$	{00,01,10,11)}	[0 3] = [0 2^2-1]
3	$8 = 2^3$	{000,001,010,011,100,101,110,111}	[0 7] = [0 2^3-1]
...	...	...	...

Si maintenant on est amené à coder un nombre entier pouvant être positif ou négatif sur un octet (8 bits), le bit de poids le plus fort sera réservé pour coder le signe «+» ou «-», et les sept bits restants serviront pour coder les valeurs possible c.à.d. 2^7 = 128 valeurs de -128 à -1 puis de 0 à 127. Ce qui nous donne [-2^7 2^7-1]

B.3.b) Codage d'un nombre réel

(1) Représentation arithmétique d'un nombre réel (norme IEEE)

Pour des raisons liées à la précision et à la normalisation on ne rentrera pas en détail dans le traitement des cas particuliers.

Les nombres réels en informatique sont des ombres qui comportent des chiffres après la virgule, avec une partie réelle et une partie décimale. A titre d'exemple la valeur 57,346 peut d'écrire de la manière suivante :

$$(36{,}375)_{10} = (36)_{10} + (0{,}375)_{10}$$

Dans le système décimal (base 10), tout nombre réel «d» peut s'écrire de la manière suivante :

$$d = d_n\ 10^n + d_{n-1}10^{n-1} + \dots + d_0\ 10^0 + d_{-1}\ 10^{-1} + \dots + d_{-p}\ 10^{-p} = \sum_{i=n}^{-p} d_i * 10^i$$

$$\text{avec } \forall\ i,\ d_i \in \{0,1,2,3,4,5,6,7,8,9\}$$

Exemple : $(36{,}375)_{10} = 3*10^1 + 6*10^0 + 3*10^{-1} + 7*10^{-2} + 5*10^{-3}$

En base 2, tout nombre réel «X» peut s'écrire de la manière suivante :

$$X = b_n\ 2^n + b_{n-1}2^{n-1} + \dots + b_0\ 2^0 + b_{-1}\ 2^{-1} + \dots + b_{-p}\ 2^{-p} = \sum_{i=n}^{-p} b_i * 2^i$$

$$= (b_n\ b_{n-1} \dots b_0\ , b_{-1} \dots b_{-p})_2 \qquad \text{avec } \forall\ i,\ b_i \in \{0,1\}$$

Exemple : $(36{,}375)_{10} = (36)_{10} + (0{,}375)_{10} = (100100)_2 + (011)_2 = (100100{,}011)_2$

Pour la valeur «36» on a déjà vu comment effectuer sa conversion en base deux (voir section B.3.b). Il nous reste à montrer comment obtenir la représentation binaire de la valeur «0,375».

Algorithme de conversion en base 2 de la partie décimale «0,375» :

0,375 * 2 = 0,75
0,75 * 2 = 1,5
0,5 * 2 = 1,0
0,0 * 2 = 0,0 ↓

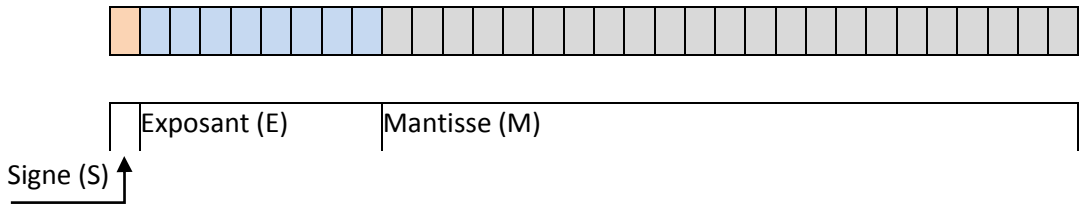
Ce qui nous permet d'écrire : $(0{,}375)_{10} = (011)_2$

Notons que cet algorithme n'est pas forcément terminal. En effet, pour certaines valeurs la multiplication par deux ne donnera jamais la valeur zéro :

0,3 * 2 = 0,6
0,6 * 2 = 1,2
0,2 * 2 = 0,4
0,4 * 2 = 0,8
0,8 * 2 = 1,6
0,6 * 2 = 1,2
0,2 * 2 = 0,4
0,4 * 2 = 0,8
0,8 * 2 = 1,6
0,6 * 2 = 1,2 ↓
...

(2) Représentation mémoire d'un nombre réel (norme IEEE)

Tout nombre réel peut s'écrit en mémoire en respectant le schéma suivant :



Les valeurs à stocker dans cette mémoire doivent être normalisée selon le modèle ci-dessous :

$$X = (-1)^S (M)_2 (2^E)_{10}$$

S : Représente la valeur du bit le plus fort, elle permet de coder le signe de «X». Ce bit vaut zéro pour les valeurs positives, et un pour les valeurs négatives.

M : Représente la mantisse, elle permet de coder la valeur de «X» sous la forme normalisée «$1,b_0b_{1...}b_{k-1}$». Etant donné que toutes les valeurs seront représentées avec un 1 à gauche de la virgule, ce terme ne sera pas stocké en mémoire. Néanmoins, il sera considéré dans les calculs. Cette technique nous fait gagné un bit supplémentaire pour la représentation de «X». On dit alors que «X» est de précision «k».

E : Représente la valeur de l'exposant exprimée en base dix. Elle permet de déterminer la position de la virgule dans la représentation de la mantisse. Cette valeur doit être dans un premier temps, reformulée pour pouvoir prendre en considération des exposants - éventuellement - négatifs, avant d'être exprimée

en base deux et stockée en mémoire.

En effet, l'exposant peut être négatif, la solution qui a été retenue dans la norme «IEEE» consiste à le représenter de la manière suivante :

$E_{max} = 2^{w-1} - 1$ «w» : représente le nombre de bits utilisés pour coder l'exposant

$E_{min} = 1 - E_{max}$

$E_{codé} = (E + E_{max})_2$

La représentation de «X» en mémoire va donc prendre la forme suivante :

$$(X)_2 = (S)_2\ (E_{codé})_2\ (M)_2$$

Avec $S \in \{0,1\}$, et le terme 1 à gauche de la virgule située dans la mantisse sera ignoré.

- Configuration d'un nombre flottant en simple précision stocké sur 4 octets c.à.d. 32 bits : 1 bit sera utilisé pour le signe «+» ou «-», 8 bits pour l'exposant et 23 bits pour la mantisse.
- Configuration d'un nombre flottant en double précision, la représentation est identique à celle de la simple précision, à la différence que les champs réservés sont plus larges. En effet, la mantisse sera codée sur 52 bits alors que l'exposant - quant à lui - sera codé sur 11 bits.

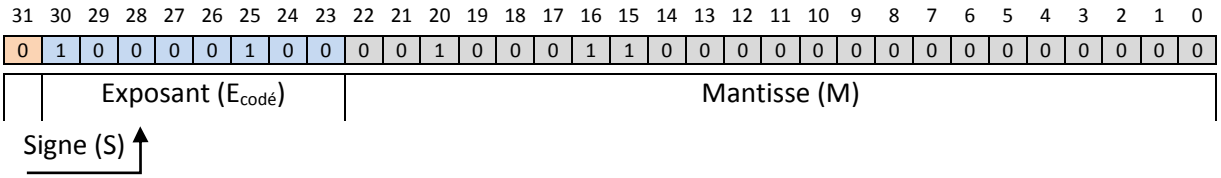
Exemple avec une valeur réelle définie avec une simple précision et codée sur 32 bits, dont 8 bits pour coder l'exposant et 1 bit pour coder le signe :

$$X = (36{,}375)_{10} = (36)_{10} + (0{,}375)_{10} = (100100)_2 + (011)_2 = (100100{,}011)_2 = +(1{,}00100011)_2 * (2^5)_{10}$$

S = 0 car la valeur «X» est positive.

M = 1,00100011 (notons que la valeur 1 à gauche de la virgule ne sera pas représentée en mémoire)

E = 5 $\Longrightarrow$ $E_{codé} = E + E_{max} = (5 + 2^{8-1} - 1)_{10} = (5 + 127)_{10} = (132)_{10} = (10000100)_2$



B.4. Les types de données primitifs en C, leurs tailles et leurs domaines de valeurs

En langage «C», il existe plusieurs types de données de base prédéfinis pour représenter des valeurs entières, des valeurs réelles, des caractères ou encore des adresses mémoires. Les tailles qu'occupent ces types de données, et par conséquent leurs domaines de valeurs, dépendent des systèmes de chiffrement de votre système d'exploitation et de votre compilateur. Pour connaitre la taille qu'occupe chaque type de données sur votre machine, il suffit de faire appel à l'opérateur «**sizeof**». Ayant l'apparence d'une fonction, cet opérateur prend en paramètre le type de données pour lequel vous souhaitez connaître la taille, et vous renvoi la taille mémoire - exprimée en octet - que va attribuer votre système d'exploitation à ce type. A titre d'exemple, le type entier sera codé sur 16 bits, respectivement 32 bits, selon si le système de chiffrement de votre système d'exploitation est 16 bits, respectivement 32 bits.

Type de donnée	Signification	Taille	Plage de valeurs
char	Caractère	8 bits	-128 à 127 → $[-2^7 : +(2^7 - 1)]$
unsigned char	Caractère non signé	8 bits	0 à 255 → $[0 : +(2^8 - 1)]$
short int	Entier court	16 bits	-32 768 à 32 767 → $[-2^{15} : +(2^{15} - 1)]$
unsigned short int	Entier court non signé	16 bits	0 à 65 535 → $[0 : +(2^{16} - 1)]$
int	Entier sur processeur 16 bits	16 bits	-32 768 à 32 767 → $[-2^{15} : +(2^{15} - 1)]$
	Entier sur processeur 32 bits	32 bits	-2 147 483 648 à 2 147 483 647 → $[-2^{31} : +(2^{31} - 1)]$
unsigned int	Entier positif sur proc. 16 bits	16 bits	0 à 65 535 → $[0 : +(2^{16} - 1)]$
	Entier positif sur proc. 32 bits	32 bits	0 à 4 294 967 295 → $[0 : +(2^{32} - 1)]$
long int	Entier long	32 bits	-2 147 483 648 à 2 147 483 647 → $[-2^{31} : +(2^{31} - 1)]$
unsigned long int	Entier long non signé	32 bits	0 à 4 294 967 295 → $[0 : +(2^{32} - 1)]$
float	Flottant (réel simple précision)	32 bits	$3.4*10^{-38}$ à $3.4*10^{38}$
double	Flottant (réel double précision)	64 bits	$1.7*10^{-308}$ à $1.7*10^{308}$

Notons que pour les types de données «float», «double» et «long double», seules leurs valeurs positives ont été présentées sur le tableau ci-dessus. Pour avoir une idée plus précise sur leurs domaines de valeurs, voici à titre d'exemple, les codages respectifs du type «float» et «double» :

Les nombres de type «float» sont codés sur 32 bits (4 octets) dont :

- 23 bits pour la mantisse
- 8 bits pour l'exposant
- 1 bit pour le signe
- Plage de valeurs : $[-3.4028234710^{38} : -1.4023984610^{-45}] \cup \{0\} \cup [1.4023984610^{-45} : 3.4028234710^{38}]$

Les nombres de type «double» sont codés sur 64 bits (8 octets) dont :

- 52 bits pour la mantisse
- 11 bits pour l'exposant
- 1 bit pour le signe
- Plage de valeurs : $[-1.79769313486231570x10^{308} : -4.9406564584124654410^{-324}] \cup \{0\} \cup [4.94065645841246544x10^{-324} : 1.79769313486231570x10^{308}]$

Les nombres de type «long double» sont codés sur 8, 12 ou 16 octets selon l'architecture de la machine et celle du compilateur (32 ou 64 bits). Pour illustrer ce comportement, nous allons compiler le code source suivant avec trois compilateurs différents :

Exemple :

```
void main()
{
   printf("\n Taille du type char : %d octet(s)", sizeof(char));
   printf("\n Taille du type unsigned char : %d octet(s)", sizeof(unsigned char));
   printf("\n Taille du type short int : %d octet(s)", sizeof(short int));
   printf("\n Taille du type int : %d octet(s)", sizeof(int));
   printf("\n Taille du type long int : %d octet(s)", sizeof(long int));
   printf("\n Taille du type float : %d octet(s)", sizeof(float));
   printf("\n Taille du type double : %d octet(s)", sizeof(double));
   printf("\n Taille du type long double : %d octet(s)", sizeof(long double));
   printf("\n_____");
   printf("\n Taille du type char * : %d octet(s)", sizeof(char *));
   printf("\n Taille du type unsigned char * : %d octet(s)", sizeof(unsigned char *));
   printf("\n Taille du type short int * : %d octet(s)", sizeof(short int *));
   printf("\n Taille du type int * : %d octet(s)", sizeof(int *));
   printf("\n Taille du type long int * : %d octet(s)", sizeof(long int *));
   printf("\n Taille du type float * : %d octet(s)", sizeof(float *));
```

```
   printf("\n Taille du type double * : %d octet(s)", sizeof(double *));
   printf("\n Taille du type long double * : %d octet(s)", sizeof(long double *));
}
```

Résultat sur une machine 64 bits, munie de «Windows 10» 64 bits, et compilé avec DevCpp 64 bits :

```
Taille du type char : 1 octet(s)
Taille du type unsigned char : 1 octet(s)
Taille du type short int : 2 octet(s)
Taille du type int : 4 octet(s)
Taille du type long int : 4 octet(s)
Taille du type float : 4 octet(s)
Taille du type double : 8 octet(s)
Taille du type long double : 16 octet(s)

--------------------------------------------------
Taille du type char * : 8 octet(s)
Taille du type unsigned char * : 8 octet(s)
Taille du type short int * : 8 octet(s)
Taille du type int * : 8 octet(s)
Taille du type long int * : 8 octet(s)
Taille du type float * : 8 octet(s)
Taille du type double * : 8 octet(s)
Taille du type long double * : 8 octet(s)
```

Résultat sur une machine 64 bits, munie de «Windows 10» 64 bits, et compilé avec «CodeBlocks» 32 bits :

```
Taille du type char : 1 octet(s)
Taille du type unsigned char : 1 octet(s)
Taille du type short int : 2 octet(s)
Taille du type int : 4 octet(s)
Taille du type long int : 4 octet(s)
Taille du type float : 4 octet(s)
Taille du type double : 8 octet(s)
Taille du type long double : 12 octet(s)

--------------------------------------------------
Taille du type char * : 4 octet(s)
Taille du type unsigned char * : 4 octet(s)
Taille du type short int * : 4 octet(s)
Taille du type int * : 4 octet(s)
Taille du type long int * : 4 octet(s)
Taille du type float * : 4 octet(s)
Taille du type double * : 4 octet(s)
Taille du type long double * : 4 octet(s)
```

Résultat sur une machine 64 bits, munie de «Windows 10» 64 bits, et compilé avec «Visual Studio» 32 bits :

```
Taille du type char : 1 octet(s)
Taille du type unsigned char : 1 octet(s)
Taille du type short int : 2 octet(s)
Taille du type int : 4 octet(s)
Taille du type long int : 4 octet(s)
Taille du type float : 4 octet(s)
Taille du type double : 8 octet(s)
Taille du type long double : 8 octet(s)

--------------------------------------------------
Taille du type char * : 4 octet(s)
Taille du type unsigned char * : 4 octet(s)
Taille du type short int * : 4 octet(s)
Taille du type int * : 4 octet(s)
Taille du type long int * : 4 octet(s)
Taille du type float * : 4 octet(s)
Taille du type double * : 4 octet(s)
Taille du type long double * : 4 octet(s)
```

Notons que le même programme source - compilé sur une même machine 64 bits, munie du système d'exploitation «Windows 10» 64 bits, mais compilé avec des compilateurs différents - a donné lieu à des

plages de valeurs différentes mais pour des types identiques. C'est le cas notamment pour les pointeurs et le type «long double».

En résumé, les types de données sont tous codés en binaire au niveau de la mémoire. L'espace mémoire sera allouée automatiquement pour les types primitifs, mais les tailles des zones mémoires, et par conséquent leurs domaines de valeurs, dépendent à la fois du compilateur et de l'architecture de la machine hôte. L'utilisation de l'opérateur «**sizeof**» - pour connaitre la taille de chaque type de données sur une configuration matérielle et/ou logicielle donnée - devient une obligation dans le cas d'une allocation dynamique de la mémoire.

B.5. Les formats d'affichage et de saisie pour les types de données primitifs

Avant d'entamer ce volet, il faut préciser que les données qui doivent être saisies au clavier ou afficher à l'écran - par un programme donné -, transitent par des fichiers systèmes nommés respectivement «stdin» et «stdout». Le programme va devoir donc soit récupérer ces données à partir du fichier «stdin» (fichier d'entrée standard : clavier), ou alors les envoyer au fichier «stdout» (fichier de sortie standard : écran). Le problème qui se pose est que lorsqu'on souhaite par exemple saisir ces données à partir du clavier, ces dernières sont considérées comme un flux d'informations non typées. De la même manière, mais dans le sens inverse, un programme manipule des données qui sont stockées sous forme binaire dans la mémoire. Comment faut-il faire pour les afficher de façon compréhensible à l'écran ?.

Dans la bibliothèque «stdio.h», il existe une multitude de fonctions prédéfinies, permettant au programmeur d'établir une interface de communication avec l'environnement extérieur (clavier, écran, disque dur, ...). Parmi ces fonctions, on trouve les fonctions «**scanf**» et «**printf**» qui permettent respectivement de saisir des données au clavier, et d'afficher des informations à l'écran. Avec ces fonctions, on parle respectivement d'entrée et de sortie de données formatées. En réalité, il s'agit dans les deux cas de fonctions permettant d'effectuer des conversions des données traitées.

Nous allons présenter dans ce qui suit les prototypes de ces fonctions ainsi que quelques formats de conversion usuels :

int **scanf**(const char* chaine_formatée [[, argument_typé]] ...) ; //Les «...» signifient que cette fonction est variadique

int **printf**(const char* chaine_formatée [[, argument_typé]] ...) ; //Les «...» signifient que cette fonction est variadique

Notons que ces deux fonctions sont des fonctions variadiques, ce qui signifie que le nombre d'arguments qu'on peut leur transmettre n'est pas fixé (arité non déterminée).

chaine_formatée : est une chaîne de caractères qui sert à mettre en forme les arguments selon leur type de données grâce au caractère de positionnement «%» suivi immédiatement du format de conversion. Le nombre d'éléments formatés à l'intérieur de cette chaine doit correspondre au nombre d'arguments typés.

argument_typé : correspond à la valeur qui doit être associée au type de conversion utilisé dans la chaine formatée. Etant donné que le nombre d'arguments n'est pas limité, non seulement le nombre d'éléments à formater à l'intérieur de la chaine doit correspondre au nombre d'arguments, mais les formats de conversions utilisés doivent correspondre - dans le même ordre - aux types d'arguments fournis.

Prototype du format de conversion. Les termes entre les crochets ouvrants et fermants sont optionnels :

%[indicateur][largeur][.précision][taille]type

[taille]	signification
h	L'argument doit être traité comme un entier court (short)
l	L'argument doit être traité comme un entier long pour les types entiers (long) L'argument doit être traité comme un réel avec une double précision (double)
L	L'argument doit être traité comme un réel long avec une double précision (long double)

Voici quelques formats usuels de conversion :

N.B : Dans la notation «[-]mmm.nnnnnn», le nombre de «n» dépend de la précision.

format	nature de la conversion pour afficher ou saisir des entiers
%d ou %i	Un nombre entier (conversion en int) en base dix
%u	Un nombre entier supérieur ou égale à zéro (conversion en **unsigned int**) en base dix
%x	Un nombre entier (int) exprimé en hexadécimal avec des lettres affichées en minuscule
%X	Un nombre entier (int) exprimé en hexadécimal avec des lettres affichées en majuscule
%hd ou %hi	Un nombre entier (conversion en short int) en base dix
%hu	Un nombre entier supérieur ou égale à zéro (conversion en unsigned short int) en base dix
%hx	Un nombre entier (short int) exprimé en hexadécimal avec des lettres affichées en minuscule
%hX	Un nombre entier (short int) exprimé en hexadécimal avec des lettres affichées en majuscule
%ld ou %li	Un nombre entier (conversion en long int) en base dix
%lu	Un nombre entier supérieur ou égale à zéro (conversion en unsigned long int) en base dix
%lx	Un nombre entier (long int) exprimé en hexadécimal avec des lettres affichées en minuscule
%lX	Un nombre entier (long int) exprimé en hexadécimal avec des lettres affichées en majuscule

format	nature de la conversion pour afficher ou saisir des réels
%f	Un nombre réel (float) sous la forme [-]mmm.nnnnnn en base dix
%e	Un nombre réel (float) sous la forme [-]m.nnnnnne$^{[+\|-]xx}$ en base dix
%E	Un nombre réel (float) sous la forme[-]m.nnnnnnE$^{[+\|-]xx}$ en base dix
%lf	Un nombre réel (double) sous la forme [-]mmm. nnnnnn en base dix
%le	Un nombre réel (double) sous la forme [-]m. nnnnnne$^{[+\|-]xx}$ en base dix
%lE	Un nombre réel (double) sous la forme[-]m.nnnnnnE$^{[+\|-]xx}$ en base dix
%Lf	Un nombre réel (long double) sous la forme [-]mmm. nnnnnn en base dix

format	nature de la conversion pour afficher ou saisir des caractères ou chaine de caractères
%c	Un caractère (conversion en unsigned char)
%s	Une chaîne de caractères (conversion en char*)

[précision]	signification
.nombre	Utilisé conjointement avec les formats : s Représente le nombre maximum de caractères à afficher
	Utilisé conjointement avec les formats : d, i, u, x ou X Représente le nombre minimum de chiffres décimaux à afficher Ajoute des espaces si la valeur à afficher est plus courte que le nombre de chiffres demandés
	Utilisé conjointement avec les formats : f, e, ou E Représente le nombre de chiffres à afficher après la virgule

[largeur]	signification
nombre	Représente le nombre minimum de caractères à afficher Dans le cas d'une valeur courte par rapport à l'affichage demandé, des espaces sont ajoutés
0nombre	Représente le nombre minimum de caractères à afficher Dans le cas d'une valeur courte par rapport à l'affichage demandé, des zéros sont ajoutés
*nombre	La largeur n'est pas spécifiée dans la chaine formatée, mais par un entier précédent l'argument à afficher

[indicateur]	signification
-	Alignement à gauche pour la largeur donnée
+	Affichage forcé du signe de la valeur numérique
espace	Insertion d'un caractère d'espacement si la valeur numérique est positive

caractères spéciaux	signification
\0	caractère **NULL** ayant la valeur 0 : marqueur de fin d'une chaîne de caractères
\a	*beep* système : alerte sonore
\b	*backspace* : déplacement du curseur d'un caractère en arrière
\f	*form feed* : saut de page
\n	*new line* : saut de ligne
\r	*carriage return* : retour chariot
\t	tabulation horizontale
\v	tabulation verticale
\\	affichage du caractère \
\'	affichage du caractère '
\"	affichage du caractère "
%%	affichage du caractère %

Exemples d'utilisation des formats de conversion :

```
void main()
{
   printf("\n Taille du type char : %d octet(s)", sizeof(char));
   printf("\n Taille du type unsigned char : %d octet(s)", sizeof(unsigned char));
   printf("\n Taille du type short int : %d octet(s)", sizeof(short int));
   printf("\n Taille du type int : %d octet(s)", sizeof(int));
   printf("\n Taille du type long int : %d octet(s)", sizeof(long int));
   printf("\n Taille du type float : %d octet(s)", sizeof(float));
   printf("\n Taille du type double : %d octet(s)", sizeof(double));
   printf("\n Taille du type long double : %d octet(s)", sizeof(long double));
   printf("\n_____");
   printf("\n Taille du type char * : %d octet(s)", sizeof(char *));
   printf("\n Taille du type unsigned char * : %d octet(s)", sizeof(unsigned char *));
   printf("\n Taille du type short int * : %d octet(s)", sizeof(short int *));
   printf("\n Taille du type int * : %d octet(s)", sizeof(int *));
```

```
   printf("\n Taille du type long int * : %d octet(s)", sizeof(long int *));
   printf("\n Taille du type float * : %d octet(s)", sizeof(float *));
   printf("\n Taille du type double * : %d octet(s)", sizeof(double *));
   printf("\n Taille du type long double * : %d octet(s)", sizeof(long double *));
}
```

Résultat sur une machine 64 bits, munie de «Windows 10» 64 bits, et compilé avec «Visual Studio» 32 bits :

```
Taille du type char : 1 octet(s)
Taille du type unsigned char : 1 octet(s)
Taille du type short int : 2 octet(s)
Taille du type int : 4 octet(s)
Taille du type long int : 4 octet(s)
Taille du type float : 4 octet(s)
Taille du type double : 8 octet(s)
Taille du type long double : 8 octet(s)
_____________________________________________
Taille du type char * : 4 octet(s)
Taille du type unsigned char * : 4 octet(s)
Taille du type short int * : 4 octet(s)
Taille du type int * : 4 octet(s)
Taille du type long int * : 4 octet(s)
Taille du type float * : 4 octet(s)
Taille du type double * : 4 octet(s)
Taille du type long double * : 4 octet(s)
```

C. Les constantes et les variables

En programmation, on est souvent amenés à manipuler des données de natures différentes. En effet, certaines données se distingue par leur aspect invariant, tandis que d'autres peuvent évoluer au cours des traitements. Dans le premier cas, on parle d'informations constantes. Dans le second cas, on parle d'informations variables. Dans les deux cas de figures, on doit leur associer un identificateur et de la mémoire. Selon l'aspect de l'information à manipuler (constant ou variant), la zone mémoire associée à ces identificateurs peut être accessible en lecture mais protégée contre la réécriture (cas des constantes après leur initialisation), ou alors accessible en lecture/écriture (cas des variables). Par ailleurs, la taille mémoire ainsi que la manière avec laquelle ces données seront lues et écrits dans leur emplacement mémoire vont dépendre du type de données associé à ces informations (caractère, chaine de caractères, entier, réel, ...).

Pour chaque type de données, on doit donc associer identificateurs dont le contenu de la mémoire sera soit invariant, on parle alors de constantes symboliques, soit variant, on parle dans ce cas de variables.

Il existe aussi des constantes dites littérales car elles ne sont associées a aucun identificateur. En effet, elle sont définie dans le code source de façon explicite (123, 3.14 , 'A', "Bonjour tout le monde"). Notons que les identificateurs peuvent être des mots alphanumériques pouvant contenir le caractère de liaison "_" (underscore : tiret du 8). Toutefois, ils ne doivent ni commencer par un chiffre, ni correspondre à des mots clés du langage. Les variables ou les constantes associées à ces identificateurs doivent être initialisées, sinon elles vont contenir les valeurs laissées par les autres programmes.

```
#include <stdio.h>
#define CPI 3.14//macro précompilée, associée à une constante symbolique dont la valeur va être
                //traduite implicitement en type double
void main()
{
   char cc = 'B' ;                          //variable symbolique de type caractère
   unsigned int uic = 66 ;                    //variable symbolique de type entier non signé
   int ia = 123 ;                             //variable symbolique de type entier
   float fb = 123.456 ;                     //variable symbolique de type réel simple précision
```

```
   const cia = 321 ;                        //constante symbolique de type entier
   const float cfpi = 3.14 ;                  //constante symbolique de type réel
   const double cdpi = 3.14159265358979 ;   //constante symbolique de type double

   printf("\n--------LES VARIABLES SYMBOLIQUES---------") ;
   printf("\nL01==>%c ==> %d", cc, cc) ;
   printf("\nL02==>%c ==> %u", uic, uic) ;
   printf("\nL03==>%d", ia) ;
   printf("\nL04==>%f", fb) ;
   printf("\n--------LES CONSTANTES SYMBOLIQUES--------") ;
   printf("\nL05==>%d", cia) ;
   printf("\nL06==>%f", cfpi) ;
   printf("\nL07==>%f", CPI) ;
   printf("\nL08==>%f", cdpi) ;
   printf("\nL09==>%lf", cdpi) ;
   printf("\nL10==>%Lf", cdpi) ;
   printf("\nL11==>%.10f", cdpi) ;
   printf("\nL12==>%.15lf", cdpi) ;
   printf("\nL13==>%.20Lf", cdpi) ;
   printf("\n--------LES CONSTANTES LITTERALES--------") ;
   printf("\nL14==>%d", 789) ;
   printf("\nL15==>%f", 789.123) ;
   printf("\nL16==>char : %d ou %d ===> %c %c", 65, 'A', 65, 'A') ;
   printf("\nL17==>char* : %s", "Bonjour tout le monde") ;
   getchar();
}
```

Résultat :

```
--------LES VARIABLES SYMBOLIQUES---------
L01==>B ==> 66
L02==>B ==> 66
L03==>123
L04==>123.456001
--------LES CONSTANTES SYMBOLIQUES--------
L05==>321
L06==>3.140000
L07==>3.140000
L08==>3.141593
L09==>3.141593
L10==>3.1415926536
L11==>3.141592653589790
L12==>3.14159265358979000000
--------LES CONSTANTES LITTERALES--------
L13==>789
L14==>789.123000
L16==>char : 65 ou 65 ===> A A
L17==>char* : Bonjour tout le monde
```

Notons également qu'on peut déclarer plusieurs variables de même types et les initialiser dans la même instruction (une instruction en «C» se termine par un point-virgule) :

Exemple :

int a , b = 10 , c = 2*b , d = 100 , e = c*d , e , f ;

Les variables a, e et f ont des valeurs mais elles ne sont pas connues. Ces valeurs ont été laissées par les autres programmes dans les zones mémoires qui ont été allouées pour ces variables.

Les variables b, c, d et e vont être initialisées. Elles vont contenir respectivement les valeurs suivantes : 10, 20, 100, 2000.

Pour les constantes, la déclaration et l'initialisation doivent s'effectuer obligatoirement dans la même instruction, car c'est le seul moment où l'on peut leur attribuer des valeurs, après cela la zone mémoire qui leur est attribuée va être protégée contre l'écriture, et on ne pourra plus leur affecter des valeurs.

Exemple :

```
const float pi = 3.14  ;          //instruction valable
const float pi ; pi = 3.14  ;     //instruction non valable
```

D. La portée des variables

La portée d'une variable correspond à la portion du code source dans laquelle elle est visible (accessible). Il existe plusieurs niveaux de visibilité pour les variables, ce qui rend possible l'association d'un même identificateur avec plusieurs variables différentes à condition - bien évidemment - que cela se passe sur des niveaux différents. En effet, le fait de déclarer plusieurs variables avec le même identificateur sur un même niveau provoque des erreurs durant la phase de compilation. Notons par ailleurs que le langage «C» est sensible à la casse, ce qui signifie que l'instruction «int a = 10 , A = 20 ;» est correcte. Elle permet de déclarer deux variables distinctes de valeurs respectives 10 et 20.

La visibilité d'une variable est généralement limitée aux accolades ouvrante et fermante qui englobent cette variable. Toutefois, il existe des variables qui ne sont pas limitées par des accolades. Elles sont définies en dehors de toutes fonctions et procédures. Ces variables sont appelées variables globales, leur visibilité porte sur la totalité du fichier où elles sont définies, ainsi que sur les fichiers d'un même projet.

```
#include <stdio.h>

int A = 100 ; // Déclaration d'une variable globale

void procedure_1()
{
   int a = 10 ;
   printf("\nL01==> a vaut : %d\t\tA vaut : %d", a, A) ; //a(10) variable local, A(100) variable globale
}

void procedure_2()
{
   int a = 20, A = 30 ;
   printf("\nL02==> a vaut : %d\t\tA vaut : %d", a, A) ; // a(20) et A(30) variables locales
}

void procedure_3(int a) // déclaration de la procédure avec comme paramètre l'entier "a"
{
   //int a = 40 ;  // cette déclaration possède la même portée que le paramètre "a" de la procédure,
                   // elle va donc causer une erreur de compilation, il faut la laisser en commentaire.
   printf("\nL03==> a vaut : %d\t\tA vaut : %d", a, A) ; // a paramètre, A(100) variable globale
   int A = 30 ;    // Déclaration d'une nouvelle variable locale A(30)
   printf("\nL04==> a vaut : %d\t\tA vaut : %d", a, A) ; // a paramètre, A(30) variable locale.
   {
      int A = 50 ; // Déclaration d'une nouvelle variable locale A dont la portée est limitée aux accolades
      printf("\nL05==> a vaut : %d\t\tA vaut : %d", a, A) ; // a paramètre, A(50) variable locale
      int a = 60 ; // Déclaration d'une nouvelle variable locale a dont la portée est limitée aux accolades
      printf("\nL06==> a vaut : %d\t\tA vaut : %d", a, A) ; // a(60) variable locale, A(50) variable locale
      {
         printf("\nL07==> a vaut : %d\t\tA vaut : %d", a, A) ; // a(60) variable locale, A(50) variable locale
         int A = 70 ; // Déclaration d'une nouvelle variable locale A dont la portée est limitée aux accolades
         printf("\nL08==> a vaut : %d\t\tA vaut : %d", a, A) ; // a(60) variable locale, A(70) variable locale
         int a = 80 ; // Déclaration d'une nouvelle variable locale a dont la portée est limitée aux accolades
         printf("\nL09==> a vaut : %d\t\tA vaut : %d", a, A) ; // a(80) variable locale, A(70) variable locale
      }
   }
   printf("\nL10==> a vaut : %d\t\tA vaut : %d", a, A) ; // a paramètre, A(30) la variable locale.
}

void main()
```

```
{
   int a = 90 ;
   printf("\nL00==> a vaut : %d\t\tA vaut : %d", a, A) ; // a(90) variable locale, A(100) variable globale.
   procedure_1() ;
   procedure_2() ;
   procedure_3(200) ;
   printf("\nL11==> a vaut : %d\t\tA vaut : %d", a, A) ; // a(90) variable locale, A(100) variable globale.
   getchar();
}
```

Résultat :

```
L00==> a vaut : 90   A vaut : 100
L01==> a vaut : 10   A vaut : 100
L02==> a vaut : 20   A vaut : 30
L03==> a vaut : 200    A vaut : 100
L04==> a vaut : 200    A vaut : 30
L05==> a vaut : 200    A vaut : 50
L06==> a vaut : 60   A vaut : 50
L07==> a vaut : 60   A vaut : 50
L08==> a vaut : 60   A vaut : 70
L09==> a vaut : 80   A vaut : 70
L10==> a vaut : 20   A vaut : 100
```

E. Les classes de stockage usuelles pour les variables et les constantes

La classe de stockage d'une variable permet de spécifier sa durée de vie et sa portée. Les classifications mémoire les plus connues que l'on puisse faire des variables sont les classifications locales ou globales. En effet, comme on l'a déjà expliqué précédemment, les variables locales sont créées à l'intérieur d'un bloc d'instructions, alors que les variables globales sont déclarées en dehors de tout bloc d'instructions. Elles ont des portées et des durées de vie différentes. Une variable globale est visible dans tout le programme, ce qui lui permet d'avoir une durée de vie similaire (on parle de persistance), alors que la portée d'une variable locale porte - plus précisément - sur la portion du bloc démarrant à l'endroit où elle a été déclarée jusqu'a l'accolade fermante de ce même bloc. Par conséquent, sa durée de vie se limitera uniquement à cette partie. En dehors de ce bloc d'instructions, elle sera détruite, et ses ressources seront restituées au système. Cependant, il est tout à fait possible de faire en sorte que les variables locales puissent survivre à la sortie de leur bloc d'instructions. Il suffit pour cela de les associer à la classe de stockage «static». Etudions quelques classes usuelles de stockage :

const :

Est la classe qui permet de rendre l'espace mémoire associé à une variable non modifiable après initialisation. Par conséquent, l'instruction de déclaration et d'initialisation doivent figurer dans la même instruction.

auto :

Est la classe de stockage par défaut, ce mot clé est donc facultatif. Les variables associées à cette classe Sont les variables locales. Elles ont pour portée le bloc d'instructions dans lequel elles sont définies, elles ne sont accessibles que dans ce bloc, et leur durée de vie est également restreinte à ce bloc.

extern :

Est la classe qui permet de signaler qu'une variable peut être définie dans un autre fichier. En d'autres termes, elle permet de déclarer qu'une variable existe probablement ailleurs que dans le fichier courant, mais sans la définir. Ce mot est utilisé dans le cadre de la compilation séparée car il permet à des variables, à des procédures ou à des fonctions d'être utilisées par d'autres fichiers ou d'autres modules d'un même projet, au lieu d'êtres réécrites à chaque besoin. Toutes les variables globales, procédures ou fonctions qui ne sont pas associées à la classe «static» sont considérées par défaut comme étant de classe «extern».

static :

Est la classe de stockage qui permet de spécifier que des variables locales dont la portée se limite au bloc d'instructions où elles sont définies, auront une durée de vie identique à celle du programme. Ces variables ne seront pas détruites lors de la sortie du bloc et garderont en mémoire la dernière valeur stockée. On parle alors de variables persistantes. Cette classe peut être également associée à des variables globales, à des procédures ou à des fonctions pour limiter la visibilité et la persistantes de ces dernières au fichier qui les englobe. L'encapsulation rendue possible par cette classe de stockage va permettre d'utiliser les mêmes noms de variables globales, les mêmes noms de procédures ou de fonctions dans d'autres modules ou fichiers d'un même projet. Ainsi, on peut déclarer par exemple deux variables globales statiques portant le même nom dans deux fichiers différents d'un même projet sans qu'elles interfèrent entre elles.

Pour déclarer qu'une variable est associée à une classe de stockage particulière, il suffit de faire précéder le type de cette variable par l'un de ces mots clés. Certains de ces mots peuvent être combinés dans une même déclaration, alors que d'autres sont incompatibles et ne peuvent faire l'objet d'un regroupement.

En résumé, le terme «const» permet de protéger le contenu d'une variable contre l'écriture, tandis que les termes «static» et «extern» sont associés aux type de données d'un projet pour distinguer les objets privés (variables, procédures, fonctions, structures, ...) qui ne doivent être reconnus que dans le fichier où ils sont définis, de ceux qui sont publics et pourront par conséquent être accessibles de n'importe quel fichier du même projet.

F. Les opérateurs et leur priorités

En «C», Il existe plusieurs types d'opérateurs qui permettent - notamment - de définir des expressions à caractère :

- Arithmétique comme l'addition «+», la soustraction «-», la division «/», ou encore le modulo «%».
- Logique comme le et logique «&&», le ou logique «||», ou encore la négation «!».
- De comparaison comme l'égalité «==», la différence «!=», le supérieur stricte «>», le supérieur ou égale «>=», l'inférieur stricte «<», ou encore l'inférieur ou égale «<=».
- D'affectation comme l'affectation simple «=», ou encore l'affectation combinée «+=, -=, *=, /=, %=»
- D'incrémentation ou de décrémentation comme le «++» et le «--».

Il existe des opérateurs unaires (admettent un seul opérande), binaires (admettent deux opérandes), et même ternaire admettent trois opérandes). Tous ces opérateurs peuvent être combinés pour former des expressions complexes des fois, le tout est de savoir comment les appliquer. Pour les opérateurs binaires et ternaires l'opérateur se trouve entre les opérandes, alors que les opérateurs unaires se positionnent généralement avant l'opérande, à l'exception des opérateurs «++» et «--» qui peuvent s'écrire de manière préfixée ou suffixée.

Lorsqu'une expression contient plusieurs opérateurs, il est important de savoir dans quel ordre ils seront évalués. Comme il est important également de savoir dans quel ordre seront traités les arguments des opérateurs ayant la même priorité.

Le tableau ci-dessous contient une liste d'opérateurs usuels classés par ordre de priorité décroissante, de l'opérateur le plus prioritaire en haut du tableau à l'opérateur le moins prioritaire en bas du tableau. La colonne «Prio» indique la priorité de l'opérateur ou des opérateurs car certains opérateurs possèdent la même priorité. La colonne «Eval» indique le sens de l'évaluation des opérateurs.

Remarque :

Sur ce tableau, certains opérateurs apparaissent plusieurs fois avec des priorités différentes car leur rôle est attaché au contexte dans lequel ils sont définis. C'est le cas notamment des parenthèses «( )», de l'étoile «*» et du et commercial «&».

Notations :

I=>E : indique une évaluation de l'opérateur de l'intérieur vers l'extérieur,

G=>D : : indique une évaluation de l'opérateur de la gauche vers la droite,

I=>E : : indique une évaluation de l'opérateur de la droite vers la gauche.

Prio.	Opérateur	Parité	Eval.	Description
1	( )		I=>E	Parenthèses limitant une expression
2	( ) [ ] . ->		G=>D	Parenthèses de fonctions ou de procédures, index de tableau, membre de structure, pointe sur un membre de structure
3	!	unaire	D=>G	La négation booléenne
4	~	unaire	D=>G	La négation binaire
5	++ --	unaire	D=>G	L'incrémentation et la décrémentation
6	-	unaire	D=>G	L'opposé
7	(type)	unaire	D=>G	Parenthèses enveloppant un type de données : opérateurs de transtypage (cast ou encore conversion explicite)
8	*	unaire	D=>G	Opérateur d'indirection (déréférencement)
9	&	unaire	D=>G	Opérateur d'adressage (référencement)
10	**sizeof**(type / variable)	unaire	D=>G	Fournit la taille mémoire en octet d'un type de données ou d'une variable
11	* / %	binaire	G=>D	Multiplication, division, modulo (reste de la division entière)
12	+ -	binaire	G=>D	Addition / Soustraction
13	>> <<	binaire	G=>D	Décalages de bits
14	> >= < <=	binaire	G=>D	Comparaisons
15	== !=	binaire	G=>D	Egalité et différence
16	&	binaire	G=>D	Le et binaire
17	^	binaire	G=>D	Le ou exclusif binaire
18	|	binaire	G=>D	Le ou inclusif binaire
19	&&	binaire	G=>D	Le et logique
20	||	binaire	G=>D	Le ou logique
21	(exp)? action1: action2	ternaire	D=>G	Renvoi l'évaluation de action1 si expression est vrai, sinon ça renvoi l'évaluation de action2
22	= += -= *= /= %= ^= &= |= >>= <<=	binaire	D=>G	Affectation simple et affectations combinées
23	,	binaire	G=>D	Séquencement

Opérateurs d'affectation	Rôle
=	Stocke une valeur à droite dans une variable à gauche
+=	Additionne deux valeurs et stocke le résultat dans la variable à gauche
-=	Soustrait deux valeurs et stocke le résultat dans la variable à gauche
*=	Multiplie deux valeurs et stocke le produit dans la variable à gauche
/=	Divise deux valeurs et stocke le résultat dans la variable à gauche
%=	Effectue une division entière et stocke le reste dans la variable à gauche

Exemple :

```
#include <stdio.h>

void main()
{
  int a = 10, b = 20, c ; // la virgule permet de définir une séquence d'instructions

  b += 5 ; // b vaut 25
  printf("\n1) b vaut : %d", b) ;
  a *= b ; //a vaut 250
  printf("\n2) a vaut : %d", a) ;
  a -= 2*b ; //a vaut 200
  printf("\n3) a vaut : %d", a) ;
  a /= b ; //a vaut 8
  printf("\n4) a vaut : %d", a) ;
  b %= a ; //b vaut 1
  printf("\n5) b vaut : %d", b) ;
  a = ++b ; //a vaut 2, b vaut 2
  printf("\n6) a vaut : %d et b vaut : %d", a, b) ;
  a *= b += 3 ; //b vaut 5, a vaut 10
  printf("\n6) a vaut : %d et b vaut : %d", a, b) ;
  c = !(a -= 10) == !!b; //b vaut 5, a vaut 0 et c vaut 1, attention !!b et différent de b
  printf("\n7) a vaut : %d, b vaut : %d et c vaut : %d", a, b, c) ;
  c = (a > b)?a:b ; //condition ternaire : si (a > b) alors retourner a, sinon retourner b ;
  printf("\n8) a vaut : %d, b vaut : %d et c vaut : %d", a, b, c) ;
  getchar() ;
}
```

Résultat :

```
1) b vaut : 25
2) a vaut : 250
3) a vaut : 200
4) a vaut : 8
5) b vaut : 1
6) a vaut : 2 et b vaut : 2
6) a vaut : 10 et b vaut : 5
7) a vaut : 0, b vaut : 5 et c vaut : 1
8) a vaut : 0, b vaut : 5 et c vaut : 5
```

G. Les tests logiques

G.1. Syntaxe générale du «if»

Les symboles «[» et «]» ne font pas parti de la syntaxe du «if». Ils sont là uniquement pour signaler que l'élément englobé par ces symboles est optionnel. Par conséquent, seule la première ligne est obligatoire, les autres lignes sont optionnelles, leur utilisation dépend des contraintes algorithmiques. Toutefois, il y a un

ordre à respecter. En effet, lorsque ces termes sont utilisés ensemble, le «if» se positionne toujours au début de l'instruction, le «else» se positionne toujours à la fin de l'instruction, alors que le «else if» se situe toujours au milieu de l'instruction. Le nombre des «else if» nécessaires dépend des besoins algorithmiques. Notons que dans le «else» il n'y a aucun test logique, ce terme permet en fait de regrouper les traitements liés à tous les cas de figures non traités dans le «if» et dans les différents «else if». En d'autres termes, le «else» représente la négation de tous les tests qui le précède dans le «if» et les «else if».

```
if (test_logique_1) {bloc_insctuctions_1 ;}
[
   [else if (test_logique_2) {bloc_insctuctions_2 ;}]
   […]
   [else if (test_logique_n) {bloc_insctuctions_n ;}]
   [else {block_insctuctions_n_+_1 ;}]
]
```

Il est nécessaire de signaler que les tests logiques peuvent être composés de plusieurs tests élémentaires liés à travers des opérateurs logiques. Dans ce cas de figure, il est important de savoir que l'évaluation des tests élémentaires liés à travers un «et» logique s'arrête dès que l'un des tests élémentaires est évalué à faux. Réciproquement, l'évaluation des tests élémentaires liés à travers un «ou» logique s'arrête dès que l'un des tests élémentaires est évalué à vrai. Par conséquent, suite à cet arrêt, les tests restants ne seront jamais évalués. Par ailleurs, lorsque respectivement, le bloc d'instructions associé à un «if», à un «else if», ou à un «else» ne contient qu'une seule instruction, l'utilisation des accolades pour englober cette instruction n'est pas obligatoire. Notons également que les structures des «if» peuvent être imbriquées en cascade les unes à l'intérieur des autres.

Les différents cas d'utilisation du «if» :

if (test_logique_1)
{
  bloc_instructions_1 ;
}
	if (test_logique_1)
	{
	  bloc_instructions_1 ;
	}
	else
	{
	  bloc_instructions_2 ;
	}
		if (test_logique_1)
		{
		  bloc_instructions_1 ;
		}
		else if (test_logiques_2)
		{
		  bloc_instructions_2 ;
		}

if (test_logique_1)
{
  bloc_instructions_1 ;
}
else if (test_logique_2)
{
  bloc_instructions_2 ;
}
else
{
  bloc_instructions_3 ;
}
	if (test_logique_1)
	{
	  bloc_instructions_1 ;
	}
	else if (test_logique_2)
	{
	  bloc_instructions_2 ;
	}
	…
	else if (test_logique_n)
	{
	  bloc_instructions_n ;
	}
		if (test_logique_1)
		{
		  bloc_instructions_1 ;
		}
		else if (test_logique_2)
		{
		  bloc_instructions_2 ;
		}
		…
		else if (test_logique_n)
		{
		  bloc_instructions_n ;
		}
		else
		{
		  bloc_instructions_n_+_1 ;
		}

G.2. Syntaxe générale du «switch»

Les symboles «[» et «]» ne font pas parti de la syntaxe du «switch». Ils sont là uniquement pour signaler que l'élément englobé par ces symboles est optionnel. Syntaxiquement, seule la première ligne est obligatoire, les autres lignes sont optionnelles, mais fonctionnellement, l'utilisation du «switch» suppose l'évaluation de l'expression puis sa comparaison les constantes associées à l'instruction «case». Dès que l'évaluation de l'expression correspond à une constante liée à un «case», tous les traitements liés aux différents «case» situés en-dessous du «case» en question seront exécutés, à moins que l'un des blocs situé en-dessous contient un «break» auquel cas, l'exécution des traitements sera stoppée provocant ainsi la sortie du «switch». Si l'évaluation de l'expression ne correspond à aucune constante, c'est le bloc d'instructions associé au «default» qui sera exécuté dans la mesure où ce dernier a été défini.

```
switch (expression)
{
   [ case constante_1 : [bloc_insctuctions_1 ;][break ;] ]
   [ … ]
   [ case constante_n : [bloc_insctuctions_n ;][break ;] ]
   [ default :  [bloc_insctuctions_n_+_1 ;] [break ;] ]
}
```

Notons, que les constantes liées aux «case» doivent être connues avant la compilation, et doivent être différentes les unes des autres.

Exemple :

```c
#include <stdio.h>

void main()
{
  int choix ;

  printf("\n***** Menu *****") ;
  printf("\n1) Option 1") ;
  printf("\n2) Option 2") ;
  printf("\n3) Option 3") ;

  printf("\n\n Entrez votre choix : ") ;
  scanf("%d", &choix) ;

  switch (choix)
  {
    case 1:
         printf("\nVous avez choisi l'option 1.") ;
         break ;
    case 2:
         printf("\nVous avez choisi l'option 2.") ;
         break ;
    case 3:
         printf("\nVous avez choisi l'option 3.") ;
         break ;
    default:
         printf("\nL'option %d est incorrecte.", choix) ;
         break ;
  }
  printf("\n") ;
  getchar() ;
}
```

Résultat :

```
***** Menu *****
1) Option 1
2) Option 2
3) Option 3

 Entrez votre choix : 2

Vous avez choisi l'option 2.
```

```
***** Menu *****
1) Option 1
2) Option 2
3) Option 3

 Entrez votre choix : 5

L'option 5 est incorrecte.
```

```
#include <stdio.h>

void main()
{
  int choix ;

  printf("\n***** Menu *****\") ;
  printf("\n1) Option 1") ;
  printf("\n2) Option 2") ;
  printf("\n3) Option 3") ;

  printf("\n\n Entrez votre choix : ") ;
  scanf("%d", &choix) ;

  switch (choix)
  {
    case 1:
       printf("\nVous avez choisi l'option 1.") ;
       break ;
    case 2:
       printf("\nVous avez choisi l'option 2.") ;
    case 3:
       printf("\nVous avez choisi l'option 3.") ;
    default:
       printf("\nL'option %d est incorrecte.", choix) ;
  }
  printf("\n") ;
  getchar() ;
}
```

Résultat :

```
***** Menu *****
1) Option 1
2) Option 2
3) Option 3

 Entrez votre choix : 1

Vous avez choisi l'option 1.
```

```
***** Menu *****
1) Option 1
2) Option 2
3) Option 3
```

```
 Entrez votre choix : 2

Vous avez choisi l'option 2.
Vous avez choisi l'option 3.
L'option 2 est incorrecte.
```

```
***** Menu *****
1) Option 1
2) Option 2
3) Option 3

 Entrez votre choix : 3

Vous avez choisi l'option 3.
L'option 3 est incorrecte.
```

```
***** Menu *****
1) Option 1
2) Option 2
3) Option 3

 Entrez votre choix : 5

L'option 5 est incorrecte.
```

G.3. Syntaxe de la condition ternaire

La condition ternaire est un outil supplémentaire permettant d'effectuer des traitements renvoyant un résultat en fonction de l'évaluation d'un test logique effectué au préalable. En réalité, il s'agit d'un opérateur utilisant trois opérandes. Le premier opérande est une expression représentant un test logique. Le second et le troisième opérande sont des expressions dont l'une uniquement va être évaluées et retournées en fonction du résultat du test logique. En effet, si le test logique est vrai, seule la première expression va être évaluée et retournée. Dans le cas contraire, ça sera la deuxième expression qui va être évaluée et retournée. Sémantiquement, la condition ternaire est similaire un test logique avec un «if» accompagné d'un «else».

```
(test_logique) ? expression_si_test_est_vrai : expression_si_test_est_faux ;
```

Exemple :

```c
#include <stdio.h>
#include <string.h> // pour pouvoir utiliser la fonction strcpy afin de copier une chaine caractères en mémoire
#include <locale.h> // pour pouvoir utiliser la fonction setlocale afin d'imprimer des caractères accentués en C

void MaxValeurs1(int val1, int val2)
{
  int val_max ;

  val_max = (val1 >= val2) ? val1 : val2 ;
  printf("\nMaxVal1(%d , %d) : est %d", val1, val2, val_max) ;
}

void MaxValeurs2(int val1, int val2)
{
  printf("\nMaxVal2(%d , %) : est %d", val1, val2, (val1 >= val2) ? val1 : val2) ;
}
```

```c
void MineurMajeur1(int age)
{
  char *msg ;

  msg = (age >= 18) ? "majeur" : "mineur" ;
  printf("\nVotre age est : %d, vous êtes alors : %s", age, msg) ;
}

void MineurMajeur2(int age)
{
  char Message[10], *str ;
  // strcpy permet de copier une chaine caractères dans la zone mémoire associée à Message,
  // il faut inclure <string.h>
  str = (age >= 18) ? strcpy(Message,"majeur") : strcpy(Message, "mineur") ;
  printf("\nVotre age est : %d, vous êtes alors : %s (%s)", age, Message, str) ;
}

void main()
{
  setlocale(LC_CTYPE, "") ; // pour pouvoir imprimer des caractères accentués en C, il faut
inclure <locale.h>
  MaxValeurs1(10, 20) ;
  MaxValeurs2(20, 10) ;
  MineurMajeur1(10) ;
  MineurMajeur2(20) ;
  printf("\n") ;
  getchar();
}
```

Résultat :

```
MaxVal1(10 , 20) : est 20
MaxVal2(20 , 10) : est 20
Votre age est : 10, vous êtes alors : mineur
Votre age est : 20, vous êtes alors : majeur --- (majeur)
```

H. Les boucles

Les boucles sont des instructions permettant de répéter un bloc d'instructions un certain nombre de fois, et ce, tant qu'une condition ou un ensemble de conditions sont vérifiées. La règle d'or lorsqu'on utilise les boucles est de s'assurer qu'à un certain moment le test logique sera évalué à faux, car dans le cas contraire, on aura affaire à une boucle infinie. En effet, si le test logique est indéfiniment vrai, les structures de boucles vont reboucler à l'infini, ce qui évidement n'est pas appréciable.

Nous allons traiter dans cette section, les structures de boucles usuelles, à savoir le «while», le «do while» et le «for».

H.1. La boucle «while»

Syntaxe :

```
while (test_logique_1)
[ ; |
   {
     [bloc_instructions_1] ;
     [structure_if  (test_logique_2)  {[bloc_instructions_2] ;  [[continue] |  [break]]  ;]
     [...]
   }
]
```

Remarques :

- Le test logique peut être un test simple ou un test composé de plusieurs expressions,
- Les crochets sont là uniquement pour signaler que certaines instructions sont optionnelles,
- Le «continue» permet de passer à l'itération suivante sans exécuter le code source pouvant être situé après cette instruction,
- Le «break» permet de sortir immédiatement de la boucle sans exécuter le code source pouvant être situé après cette instruction,
- Lorsque le corps du «while» ne contient qu'une instruction, les accolades englobant cette instruction sont facultatives.
- Le «while» commence par effectuer le test d'arrêt avant d'exécuter le bloc d'instruction,
- Lorsque le l'instruction «while» se termine après le test logique par un point-virgule, toutes les instructions entre les accolades ne seront plus liées à cette structure et ne seront donc pas sujet à répétition.

Exemple :

```c
#include <stdio.h>
#include <locale.h> // pour pouvoir utiliser la fonction setlocale afin d'imprimer des caractères accentués en C

void AfficherValeurs (int val1, int val2)
{//affiche des valeurs entre val1 et val2
  int cpt = val1 ;

  while (cpt <= val2)
  {
    printf("\n=> valeur de cpt à l'itération %d est : %d", cpt-val1+1, cpt) ;
    cpt++ ;
  }
}

void AfficherValeursCondition (int val1, int val2)
{//affiche des valeurs entre val1 et val2 jusqu'à croiser un multiple de 7 en utilisant break
  int cpt = val1 ;

  while (cpt <= val2)
  {
    if(cpt % 7 == 0)
      break ;
    printf("\n=> valeur de cpt à l'itération %d est : %d", cpt-val1+1, cpt) ;
    cpt++ ;
  }
}
```

```
void ValeursPaires_1 (int val1, int val2)
{//affiche les multiples de 2 entre val1 et val2
  int cpt = val1 ;

  while (cpt <= val2)
  {
    if(cpt % 2 == 0)
    {
      printf("\n=> %d est un nombre pair", cpt) ;
    }
    cpt++ ;
  }
}

void ValeursPaires_2 (int val1, int val2)
{//affiche les multiples de 2 entre val1 et val2 en utilisant continue
  int cpt = val1-1 ;

  while (++cpt <= val2)
  {
    if(cpt % 2)
      continue ;
    printf("\n=> %d est un nombre pair", cpt) ;
  }
}

void main()
{
  setlocale(LC_CTYPE, "") ; // pour pouvoir imprimer des caractères accentués en C, il faut
inclure <locale.h>
  AfficherValeurs (6, 10) ;
  printf("\n_____\n") ;
  AfficherValeursCondition (3, 10) ;
  printf("\n_____\n") ;
  ValeursPaires_1 (0, 10) ;
  printf("\n_____\n") ;
  ValeursPaires_2 (11, 20) ;
  printf("\n") ;
  getchar();
}
```

Résultat :

```
=> valeur de cpt à l'itération 1 est : 6
=> valeur de cpt à l'itération 2 est : 7
=> valeur de cpt à l'itération 3 est : 8
=> valeur de cpt à l'itération 4 est : 9
=> valeur de cpt à l'itération 5 est : 10
_____
=> valeur de cpt à l'itération 1 est : 3
=> valeur de cpt à l'itération 2 est : 4
=> valeur de cpt à l'itération 3 est : 5
=> valeur de cpt à l'itération 4 est : 6
_____
=> 0 est un nombre pair
=> 2 est un nombre pair
=> 4 est un nombre pair
=> 6 est un nombre pair
=> 8 est un nombre pair
=> 10 est un nombre pair
_____
=> 12 est un nombre pair
=> 14 est un nombre pair
=> 16 est un nombre pair
=> 18 est un nombre pair
=> 20 est un nombre pair
```

H.2. La boucle «do while»

Syntaxe :

```
do
{
   [bloc_instructions_1] ;
   [structure_if (test_logique_2) {[bloc_instructions_2] ; [[continue] | [break]] ;] […] }
   […]
} while (test_logique_1) ;
```

Remarques :

- Le test logique peut être un test simple ou un test composé de plusieurs expressions,
- Les crochets sont là uniquement pour signaler que certaines instructions sont optionnelles,
- Le «continue» permet de passer à l'itération suivante sans exécuter le code source pouvant être situé après cette instruction,
- Le «break» permet de sortir immédiatement de la boucle sans exécuter le code source pouvant être situé après cette instruction,
- Lorsque le corps du «do while» ne contient qu'une instruction, les accolades englobant cette instruction sont facultatifs.
- Le «do while» commence par exécuter le bloc d'instruction et effectue le test d'arrêt à la fin,
- Il se peut que l'instruction «do while» ne contienne aucune instruction entre les accolades.

Exemple :

```c
#include <stdio.h>
#include <locale.h> // pour pouvoir utiliser la fonction setlocale afin d'imprimer des caractères accentués en C

void dw_AfficherValeurs (int val1, int val2)
{//affiche des valeurs entre val1 et val2
  int cpt = val1 ;
  do
  {
    printf("\n=> valeur de cpt à l'itération %d est : %d", cpt-val1+1, cpt) ;
    cpt++ ;
  } while (cpt <= val2) ;
}

void dw_AfficherValeursCondition (int val1, int val2)
{//affiche des valeurs entre val1 et val2 jusqu'à croiser un multiple de 7 en utilisant break
  int cpt = val1 ;
  do
  {
    if(cpt % 7 == 0)
      break ;
    printf("\n=> valeur de cpt à l'itération %d est : %d", cpt-val1+1, cpt) ;
    cpt++ ;
  } while (cpt <= val2) ;
}

void dw_ValeursPaires_1 (int val1, int val2)
{//affiche les multiples de 2 entre val1 et val2
  int cpt = val1 ;
  do
  {
    if(cpt % 2 == 0)
    {
```

```
      printf("\n=> %d est un nombre pair", cpt) ;
    }
  } while (++cpt <= val2) ;
}
void dw_ValeursPaires_2 (int val1, int val2)
{//affiche les multiples de 2 entre val1 et val2 en utilisant continue
  int cpt = val1-1 ;

  do
  {
    if(cpt % 2)
      continue ;
    printf("\n=> %d est un nombre pair", cpt) ;
  } while (++cpt <= val2) ;
}

void main()
{
  setlocale(LC_CTYPE, "") ; // pour pouvoir imprimer des caractères accentués en C, il faut
inclure <locale.h>
  dw_AfficherValeurs (6, 10) ;
  printf("\n_____________________") ;
  dw_AfficherValeursCondition (3, 10) ;
  printf("\n_____________________") ;
  dw_ValeursPaires_1 (0, 10) ;
  printf("\n_____________________") ;
  dw_ValeursPaires_2 (11, 20) ;
  printf("\n") ;
  getchar();
}
```

Résultat :

```
=> valeur de cpt à l'itération 1 est : 6
=> valeur de cpt à l'itération 2 est : 7
=> valeur de cpt à l'itération 3 est : 8
=> valeur de cpt à l'itération 4 est : 9
=> valeur de cpt à l'itération 5 est : 10
_____________________
=> valeur de cpt à l'itération 1 est : 3
=> valeur de cpt à l'itération 2 est : 4
=> valeur de cpt à l'itération 3 est : 5
=> valeur de cpt à l'itération 4 est : 6
_____________________
=> 0 est un nombre pair
=> 2 est un nombre pair
=> 4 est un nombre pair
=> 6 est un nombre pair
=> 8 est un nombre pair
=> 10 est un nombre pair
_____________________
=> 12 est un nombre pair
=> 14 est un nombre pair
=> 16 est un nombre pair
=> 18 est un nombre pair
=> 20 est un nombre pair
```

H.3. La boucle «for»

Syntaxe :

```
for ([instructions_initialisation] ; [test_logique_1] ; [instructions_incrém_décrém])
[ ; |
   {
      [bloc_instructions_1 ;]
      [structure_if (test_logique_2) {[bloc_instructions_2 ;]  [[continue] | [break]] ;] […] }
      […]
   }
]
```

Remarques :

- Le «for» possède trois paramètres facultatifs,
- Le test logique peut être un test simple ou un test composé de plusieurs expressions,
- Les crochets sont là uniquement pour signaler que certaines instructions sont optionnelles,
- Le «continue» permet de passer à l'itération suivante sans exécuter le code source pouvant être situé après cette instruction,
- Le «break» permet de sortir immédiatement de la boucle sans exécuter le code source pouvant être situé après cette instruction,
- Lorsque le corps du «for» ne contient qu'une instruction, les accolades englobant cette instruction sont facultatives,
- Le «for» commence par exécuter les instructions d'initialisation, notons que cette phase ne s'effectuera qu'une seule fois. Le «for» évalue ensuite le test logique. Deux cas se présentent alors :
  - Soit le test logique va être évalué à vrai, auquel cas, le bloc d'instructions entre les accolades sera exécuté, suivit des instructions d'incrémentation et/ou de décrémentation. Dans ce cas de figure, ce processus va recommencer les traitements au stade du test logique.
  - Soit le test logique va être évalué à faux, auquel cas, les itérations du «for» vont s'arrêter, et le programme va enchainer avec le code source situé après la structure de «for» (si code source il y a).

Exemple :

```c
#include <stdio.h>
#include <locale.h> // pour pouvoir utiliser la fonction setlocale afin d'imprimer des caractères accentués en C

void for_AfficherValeurs (int val1, int val2)
{//affiche des valeurs entre val1 et val2
  int cpt ;

  for (cpt = val1 ; cpt <= val2 ; cpt++)
  {
    printf("\n=> valeur de cpt à l'itération %d est : %d", cpt-val1+1, cpt) ;
  }
}

void for_AfficherValeursCondition (int val1, int val2)
{//affiche des valeurs entre val1 et val2 jusqu'à croiser un multiple de 7 en utilisant break
  int cpt ;
  for (cpt = val1 ; cpt <= val2 ; cpt++)
  {
    if(cpt % 7 == 0)
      break ;
```

```
    printf("\n=> valeur de cpt à l'itération %d est : %d", cpt-val1+1, cpt) ;
  }
}

void for_ValeursPaires_1 (int val1, int val2)
{//affiche les multiples de 2 entre val1 et val2
  int cpt ;
  for (cpt = val1 ; cpt <= val2 ; cpt++)
  {
    if(cpt % 2 == 0)
    {
      printf("\n=> %d est un nombre pair", cpt) ;
    }
  }
}

void for_ValeursPaires_2 (int val1, int val2)
{//affiche les multiples de 2 entre val1 et val2 en utilisant continue
  int cpt ;
  for (cpt = val1 ; cpt <= val2 ; cpt++)
  {
    if(cpt % 2)
      continue ;
    printf("\n=> %d est un nombre pair", cpt) ;
  }
}

void for_ValeursPaires_3 (int val1, int val2)
{//affiche les multiples de 2 entre val1 et val2 en utilisant continue
  int cpt = val1;

  for (  ;   ;  )
  {
    if (cpt > val2)
      break ;
    if(cpt % 2)
      continue ;
    printf("\n=> %d est un nombre pair", cpt) ;
    cpt++ ;
  }
}

void main()
{
  setlocale(LC_CTYPE, "") ; // pour pouvoir imprimer des caractères accentués en C, il faut
inclure <locale.h>
  for_AfficherValeurs (6, 10) ;
  printf("\n_____") ;
  for_AfficherValeursCondition (3, 10) ;
  printf("\n_____") ;
  for_ValeursPaires_1 (0, 10) ;
  printf("\n_____") ;
  for_ValeursPaires_2 (11, 20) ;
  printf("\n_____") ;
  for_ValeursPaires_3 (11, 20) ;
  printf("\n") ;
  getchar();
}
```

Résultat :

```
=> valeur de cpt à l'itération 1 est : 6
=> valeur de cpt à l'itération 2 est : 7
=> valeur de cpt à l'itération 3 est : 8
=> valeur de cpt à l'itération 4 est : 9
=> valeur de cpt à l'itération 5 est : 10
_____________________
=> valeur de cpt à l'itération 1 est : 3
=> valeur de cpt à l'itération 2 est : 4
=> valeur de cpt à l'itération 3 est : 5
=> valeur de cpt à l'itération 4 est : 6
_____________________
=> 0 est un nombre pair
=> 2 est un nombre pair
=> 4 est un nombre pair
=> 6 est un nombre pair
=> 8 est un nombre pair
=> 10 est un nombre pair
_____________________
=> 12 est un nombre pair
=> 14 est un nombre pair
=> 16 est un nombre pair
=> 18 est un nombre pair
=> 20 est un nombre pair
_____________________
=> 12 est un nombre pair
=> 14 est un nombre pair
=> 16 est un nombre pair
=> 18 est un nombre pair
=> 20 est un nombre pair
```

I. Les fonctions et les procédures

Les fonctions sont des outils qui permettent de décomposer un programme informatique complexe en plusieurs blocs d'instructions identifiés de façon unique, et réalisant chacun un traitement particulier. Ainsi le programme peut être recomposé en faisant appel à ces fonctions.

Cette façon de décomposer le code source d'un programme permet d'écrire des blocs d'instructions (fonctions) indépendants, paramétrables et réutilisables. En effet, chaque bloc peut être implémenté et testé indépendamment du reste du programme. Si le programme contient des traitements répétitifs, il suffit de les implémenter dans le cadre d'une fonction, et de remplacer ensuite ces traitements par l'appel à cette fonction au lieu de dupliquer à chaque fois ces traitements.

Les fonctions doivent en générale renvoyer une valeur typée. Cependant, lorsqu'on n'a pas besoin de retourner de valeurs, il suffit de déclarer que son type de retour c'est le «**void**», on parle alors de procédure. On peut donc dire qu'une procédure est une fonction qui ne retourne aucune valeur.

Syntaxe :

```
type identificateur ([type  parametre1] [,type  parametre2] [,...])
{
   [bloc_instructions]
   [ return expression ;]
}
```

Exemple :

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

void ProcedureSommeValeurs(int a, int b) //Déclaration d'une procédure qui affiche la somme de deux entiers
{
  printf("\nLa somme de %d et de %d vaut %d", a, b, a + b) ;
}

void ProcedureMaxValeurs(int a, int b) //Déclaration d'une procédure qui affiche l'ordre de deux entiers
{
  if(a > b)
  {
    printf("\nLa valeur %d est supérieure à %d", a, b) ;
    return ;
  }
  printf("\nLa valeur %d est inférieure à %d", a, b) ;
}

int FonctionSommeValeurs(int a, int b) // Déclaration d'une fonction qui renvoie la somme de deux entiers
{
  return a + b ;
}

int FonctionMaxValeurs (int a, int b) // Déclaration d'une fonction qui renvoie le max de deux entiers
{
  return (a >= b)?a:b ;
}

void main()
{
  int a = 10, b = 20, c ;
  setlocale(LC_CTYPE, "") ;
  ProcedureSommeValeurs(a, b) ;
  ProcedureMaxValeurs(a, b) ;
  c = FonctionSommeValeurs(a, b) ;
  printf("\nLa somme de %d et de %d vaut %d", a, b, c) ;
  printf("\nMaxValeurs(%d , %d) : %d", a, b, FonctionMaxValeurs(a , b)) ;
  getchar();
}
```

Résultat :

```
La somme de 10 et de 20 vaut 30
La valeur 10 est inférieure à 20
La somme de 10 et de 20 vaut 30
MaxValeurs(10 , 20) : 20
```

J. Les paramètres des fonctions/procédures : passages par valeur ou par adresse ?

Lorsqu'on souhaite définir des paramètres formels pour des fonctions ou des procédures, on a la possibilité de spécifier au compilateur comment il doit se comporter avec les valeurs effectives lors de l'appel, doit-il faire une copie de ces valeurs et travailler avec ces copies, ou alors, doit-il travailler directement sur la valeur transmise à travers son adresse mémoire. Dans le premier cas on parle de paramètres passés par valeurs. Dans le second cas, on parle de paramètre transmis par adresse. En effet, dans le premier cas on ne travaille

que sur une copie de la variable transmise, alors que dans le deuxième cas, on doit travailler directement sur l'adresse mémoire de la variable transmise.

Exemple :

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

void ProcedurePassageParValeur(int a)
{
  printf("\n\t1.1) valeur du paramètre a dans ProcedurePassageParValeur : %d", a) ;
  a = 100 ;
  printf("\n\t1.2) nouvelle valeur du paramètre a dans ProcedurePassageParValeur : %d", a) ;
}

void ProcedurePassageParAdresse(int * a)
{
  printf("\n\t2.1) valeur du paramètre a dans ProcedurePassageParAdresse : %d", *a) ;
  *a = 200 ;
  printf("\n\t2.2) nouvelle valeur du paramètre a dans ProcedurePassageParAdresse : %d", *a) ;
}

void main()
{
  int a = 10 ;
  setlocale(LC_CTYPE, "") ;
  printf("\n1) valeur de la variable a dans le main : %d", a) ;
  ProcedurePassageParValeur(a) ; //le paramètre formel opère sur une copie du paramètre effectif
  printf("\n2) nouvelle valeur de la variable a dans le main : %d", a) ;
  ProcedurePassageParAdresse(&a) ; //les paramètres formel et effectif opèrent sur la même variable
  printf("\n3) nouvelle valeur de la variable a dans le main : %d", a) ;
  getchar();
}
```

Résultat :

```
1) valeur de la variable a dans le main : 10
        1.1) valeur du paramètre a dans ProcedurePassageParValeur : 10
        1.2) nouvelle valeur du paramètre a dans ProcedurePassageParValeur : 100
2) nouvelle valeur de la variable a dans le main : 10
        2.1) valeur du paramètre a dans ProcedurePassageParAdresse : 10
        2.2) nouvelle valeur du paramètre a dans ProcedurePassageParAdresse : 200
3) nouvelle valeur de la variable a dans le main : 200
```

K. Les tableaux multidimensionnels

Contrairement aux variables de type primitif - dites aussi variables simple - qui ne peuvent contenir que des valeurs atomiques, les tableaux sont des variables, qui permettent d'identifier plusieurs valeurs de même type à la fois. Il n'est donc pas nécessaire de créer plusieurs variables de même type. En effet, une seule variable permet de faire référence à une zone mémoire contigüe qui contiendra la liste des valeurs - de même type - à stocker, mais pour cela, il faut indiquer au système le nombre de valeurs à stocker et ainsi que leur type, car de cette manière, le système saura la taille totale qu'il doit réserver en mémoire (nombre_de_valeurs***sizeof**(type)) ainsi que la manière avec laquelle il doit lire et/ou écrire les valeurs dans cette zone mémoire.

On a la possibilité de déclarer des tableaux monodimensionnels (vecteurs), bidimensionnels (matrices), tridimensionnels (cubes) et de façon générale des tableaux multidimensionnels.

En langage «C», on distingue deux variantes de tableaux :

- Les tableaux statiques dont le contenu est modifiable, mais dont la taille mémoire doit être connue avant même de compiler le programme. La modification de cette taille pendant la phase d'exécution du programme est donc impossible. Par conséquent, la taille mémoire doit être définie dans le code source, soit comme une constante littérale, soit comme une macro. Le terme statique ici fait référence à l'aspect invariant de la taille de cette zone mémoire.
- Les tableaux dynamiques dont le contenu est modifiable, mais dont la taille peut être fournie à tout moment, notamment durant la phase d'exécution du programme.

Dans cette section, nous allons nous intéresser uniquement aux tableaux statiques. Les tableaux dynamiques seront traités - plus loin - dans la section relative à l'allocation dynamique de la mémoire.

Syntaxe :

```
type  identifiant [M] ;          // déclaration d'un vecteur de taille N
type  identifiant [M][N] ;       // déclaration d'une matrice de taille M lignes et N colonnes
type  identifiant [M][N][P] ;    // déclaration d'un cube de taille M lignes, N colonnes et P étages
...     ...        ... ... ... ...   // de la même manière on peut déclarer des tableaux à plusieurs dimensions
```

Notons quelques particularités concernant les tableaux en «C» :

- Les indices des tableaux commencent toujours à zéro et se terminent à l'indice taille-1. Il est donc primordial de ne pas oublier cette règle.
- On a la possibilité de déclarer des tableaux et de les initialiser dans la même instruction, dans ce cas, la première taille peut ne pas être fournie, elle sera en effet calculée implicitement par le compilateur.
- Dans le cas des tableaux statiques, les identificateurs sont assimilés par le compilateur - à la fois - à leurs adresses mémoires, et aux adresses mémoires de leurs premiers éléments. Malgré l'apparence, le contenu de l'identificateur et totalement différent de celui du premier élément. Notons que cette particularité est liée uniquement aux tableaux statiques.

Voici donc une liste non exhaustive de possibilités :

Exemple : Tableaux monodimensionnels

```c
#include <stdio.h>
#include <locale.h> // pour pouvoir utiliser la fonction setlocale afin d'imprimer des caractères accentués en C
#define NBE 5    // définition de NBE comme étant une macro qui sera précompilée.

void DeclarationsVecteursStatiques()
{
  int i, t ;
  int T[5]     = {-5, 7, -25, 100, 6} ;
  float U[NBE] = {-15.36, 17.21, 25, 33, 32.23} ;
  char TCH1[8] = "Bonjour" ;
  char TCH2[8] = {'B', 'o', 'n', 'j', 'o', 'u', 'r', '\0'} ;
  char TCH3[8] = {66, 111, 110, 'j', 111, 117, 114, 0} ;
  char TCH4[]  = {66, 111, 110, 'j', 111, 117, 114, 0} ;
  t = sizeof(TCH4)/sizeof(char) ;

  printf("\n1) Les éléments du vecteur T en utilisant les indices :\n") ;
  for(i = 0 ; i < 5 ; i++)
  {
    printf("\t%d", T[i]) ;
```

```
  }

  printf("\n2) Les éléments du vecteur U en utilisant les indices :\n") ;
  for(i = 0 ; i < NBE ; i++)
  {
    printf("\t%.2f", U[i]) ;
  }

  printf("\n3) Les éléments du vecteur TCH1 en utilisant les indices :\n") ;
  for(i = 0 ; i < 8 ; i++)
  {
    printf("\t%c", TCH1[i]) ;
  }

  printf("\n4) Les éléments du vecteur TCH2 en utilisant les indices :\n") ;
  for(i = 0 ; i < 8 ; i++)
  {
    printf("\t%c", TCH2[i]) ;
  }

  printf("\n5) Les éléments du vecteur TCH3 en utilisant les indices le déréférencement :\n")
;
  for(i = 0 ; i < 8 ; i++)
  {
    printf("\t%c", *(TCH3 + i)) ;
  }

  printf("\n6) Les éléments du vecteur TCH4 en utilisant les indices :\n") ;
  for(i = 0 ; i < t ; i++)
  {
    printf("\t%c", TCH4[i]) ;
  }

  printf("\n7) Les vecteurs TCH1, TCH2, TCH3 et TCH4 en utilisant leurs adresses mémoires :")
;
  printf("\n\t=> TCH1 : %s\n\t=> TCH2 : %s\n\t=> TCH3 : %s\n\t=> TCH4 : %s", TCH1, TCH2, TCH3,
TCH4) ;
  printf("\n8) Les vecteurs TCH1, TCH2 et TCH3 en utilisant les adresses mémoires des premiers
éléments :");
  printf("\n\t=> TCH1 : %s\n\t=> TCH2 : %s\n\t=> TCH3 : %s", &TCH1[0], &TCH2[0], &TCH3[0]) ;
  printf("\n9) Adresse TCH1 : %d --- Adresse &TCH1[0] : %d --- Contenu TCH1 : %d --- Contenu
TCH1[0] : %d\n", &TCH1, &TCH1[0], TCH1, TCH1[0]) ;
}

void main()
{
  setlocale(LC_CTYPE, "") ; // pour pouvoir imprimer des caractères accentués en C, il faut
inclure <locale.h>
  DeclarationVecteursStatiques() ;
  getchar();
}
```

Résultat :

```
1) Les éléments du vecteur T en utilisant les indices :
        -5      7       -25     100     6
2) Les éléments du vecteur U en utilisant les indices :
        -15.36  17.21   25.00   33.00   32.23
3) Les éléments du vecteur TCH1 en utilisant les indices :
        B       o       n       j       o       u       r
4) Les éléments du vecteur TCH2 en utilisant les indices :
        B       o       n       j       o       u       r
5) Les éléments du tableau TCH3 en utilisant les indices le déréférencement :
        B       o       n       j       o       u       r
```

```
6) Les éléments du tableau TCH4 en utilisant les indices :
        B       o       n       j       o       u       r
7) Les vecteurs TCH1, TCH2, TCH3 et TCH4 en utilisant leurs adresses mémoires :
        => TCH1 : Bonjour
        => TCH2 : Bonjour
        => TCH3 : Bonjour
        => TCH4 : Bonjour
8) Les vecteurs TCH1, TCH2 et TCH3 en utilisant les adresses mémoires des premiers éléments :
        => TCH1 : Bonjour
        => TCH2 : Bonjour
        => TCH3 : Bonjour
9) Adresse TCH1 : 2293200 --- Adresse &TCH1[0] : 2293200 --- Contenu TCH1 : 2293200 ---
Contenu TCH1[0] : 66
```

Exemple : Tableaux bidimensionnels

```c
#include <stdio.h>
#include <locale.h>//pour pouvoir utiliser la fonction setlocale afin d'imprimer des caractères accentués en C
#include <string.h>//pour pouvoir utiliser la fonction strlen qui renvoie la longueur d'une chaine de caractères
#define NBL 2
#define NBC 3

void DeclarationsMatricesStatiques()
{
  int i, j, t ;
  int T[2][3]  = {{-5, 7, -25} , {100, 6, -32}} ;
  float U[NBL][NBC]= {{-5.0, 7, -25.12} , {10, .6, -32.}} ;
  char TCH1[4][8]= {"Bonjour", "tout", "le", "monde"} ;
  char TCH2[][8] = {"Bonjour", "tout", "le", "monde"} ;
  t = sizeof(TCH2)/sizeof(char*) ;

  printf("\n1) Les éléments de la matrice T en utilisant les indices :") ;
  for(i = 0 ; i < 2 ; i++)
  {
    printf("\n") ;
    for(j = 0 ; j < 3 ; j++)
    {
      printf("\t%d", T[i][j]) ;
    }
  }
  printf("\n2) Les éléments de la matrice U en utilisant les indices :") ;
  for(i = 0 ; i < NBL ; i++)
  {
    printf("\n") ;
    for(j = 0 ; j < NBC; j++)
    {
      printf("\t%.2f", U[i][j]) ;
    }
  }
  printf("\n3) Les éléments de la matrice TCH1 en utilisant les indices :") ;
  for(i = 0 ; i < 4 ; i++)
  {
    printf("\n\t=> Ligne(%d) : %s", i, TCH1[i]) ;
  }
  printf("\n4) Les éléments de la matrice TCH1 en utilisant les indices :") ;
  for(i = 0 ; i < 4 ; i++)
  {
    printf("\n\t=> Ligne(%d) : ", i) ;
    for(j = 0 ; ((j < 8) && (TCH1[i][j] != '\0')) ; j++)
    {
      printf("%c", TCH1[i][j]) ;
    }
```

```
  }
  printf("\n5) Les éléments de TCH1 en utilisant les indices et l'adresse de TCH1 :") ;
  for(i = 0 ; i < 4 ; i++)
  {
    printf("\n\t=> Ligne(%d) : %s", i, TCH1 +i) ;
  }
  printf("\n6) Les éléments de TCH1 en utilisant les indices, l'adresse de TCH1 et le
déréférencement :") ;
  for(i = 0 ; i < 4 ; i++)
  {
    printf("\n\t=> Ligne(%d) : ", i);
    for(j = 0 ; ((j < 8) && (*(*(TCH1 + i) + j) != '\0')) ; j++)
    {
      printf("%c", *(*(TCH1 + i) + j)) ;
    }
  }
  printf("\n7) Les éléments de TCH1 en utilisant les indices, l'adresse de TCH1 et le
déréférencement :") ;
  for(i = 0 ; i < 4 ; i++)
  {
    printf("\n\t=> Ligne(%d) : ", i);
    for(j = 0 ; (j < strlen(*(TCH1 + i))) ; j++)
    {
      printf("%c", *(*(TCH1 + i) + j)) ;
    }
  }
  printf("\n8) Les éléments de TCH2 en utilisant les indices et l'adresse de TCH2 :") ;
  for(i = 0 ; i < t ; i++)
  {
    printf("\n\t=> Ligne(%d) : %s", i, TCH2 + i) ;
  }
}
void main()
{
  setlocale(LC_CTYPE, "") ; // pour pouvoir imprimer des caractères accentués en C, il faut
inclure <locale.h>
  DeclarationVecteursStatiques() ;
  getchar();
}
```

Résultat :

```
1) Les éléments de la matrice T en utilisant les indices :
        -5      7       -25
        100     6       -32
2) Les éléments de la matrice U en utilisant les indices :
        -5.00   7.00    -25.12
        10.00   0.60    -32.00
3) Les éléments de la matrice TCH1 en utilisant les indices :
        => Ligne(0) : Bonjour
        => Ligne(1) : tout
        => Ligne(2) : le
        => Ligne(3) : monde
4) Les éléments de la matrice TCH1 en utilisant les indices :
        => Ligne(0) : Bonjour
        => Ligne(1) : tout
        => Ligne(2) : le
        => Ligne(3) : monde
5) Les éléments de TCH1 en utilisant les indices et l'adresse de TCH1 :
        => Ligne(0) : Bonjour
        => Ligne(1) : tout
        => Ligne(2) : le
        => Ligne(3) : monde
6) Les éléments de TCH1 en utilisant les indices, l'adresse de TCH1 et le déréférencement :
        => Ligne(0) : Bonjour
        => Ligne(1) : tout
        => Ligne(2) : le
        => Ligne(3) : monde

7) Les éléments de TCH1 en utilisant les indices, l'adresse de TCH1 et le déréférencement :
        => Ligne(0) : Bonjour
        => Ligne(1) : tout
        => Ligne(2) : le
        => Ligne(3) : monde
8) Les éléments de TCH2 en utilisant les indices et l'adresse de TCH2 :
        => Ligne(0) : Bonjour
        => Ligne(1) : tout
        => Ligne(2) : le
        => Ligne(3) : monde
```

L. L'allocation dynamique de la mémoire

On a vu dans les sections précédentes que pour les types de données primitifs, l'allocation mémoire des variables se fait automatiquement. On a vu également que dans le cas des tableaux statiques, la taille de la mémoire doit être fixée avant même de compiler de code source, et que cette taille ne peut être modifiée au cours de la phase d'exécution du programme.

L'allocation dynamique est un moyen offert par le langage «C» pour allouer de la mémoire de façon explicite et éventuellement réajuster sa taille en fonction des besoins. La modification de cette taille peut intervenir même courant la phase d'exécution du programme.

Quand l'allocation mémoire s'effectue à l'intérieur d'une fonction ou d'une procédure, l'adresse mémoire de la zone allouée doit être stockée dans une variable qui dans la plupart des cas est une variable locale. Par ailleurs, quand l'appel à une fonction ou une procédure se termine, le système libère les ressources mémoires qui leurs sont attribuées. Cela sous-entend que la variable contenant l'adresse mémoire de la zone allouée sera elle aussi libérée, et par conséquent, elle existera plus en mémoire. En revanche, la zone mémoire allouée dynamiquement ne sera pas restituée et restera en mémoire tant que la libération de la mémoire

n'est pas effectuée explicitement avec la fonction «free», ou alors, tant que le programme est toujours en cours exécution.

Moralité, quand on fait appel à l'allocation dynamique avec les fonctions usuelles «malloc», «calloc» ou «realloc», il faut soit libérer la mémoire avec la fonction «free» avant de quitter la fonction ou la procédure à l'origine de cette allocation, soit s'arranger pour renvoyer l'adresse mémoire de la zone allouée à travers une fonction (avec l'instruction return), ou alors à travers un paramètre passé par adresse, de sorte à ce qu'on puisse à tout moment avoir l'accès à cette zone pour l'exploiter ou pour la libérer.

Syntaxe :

```
type *  identifiant1 = malloc(NbrEléments * sizeof(type)) ;
type *  identifiant2 = calloc(NbrEléments , sizeof(type)) ;
type *  identifiant3 = realloc(identifiant , NbrEléments * sizeof(type)) ;
type *  identifiant4 = realloc(NULL , NbrEléments * sizeof(type)) ;
free(identifiant) ;
```

- type : fait référence à un type de données primitif, à un type de donnée multidimensionnel, ou encore à une structure de données (type de données hétérogènes définit par le programmeur),
- identifiant : fait référence au nom de la variable qui va contenir l'adresse mémoire de la zone allouée,
- NbrEléments : fait référence au nombre d'éléments que l'on souhaite stocker en mémoire,
- **sizeof**(type) : est une opérateur qui renvoie la taille qu'occupera un type de donnée en mémoire.

Les fonctions «malloc», «calloc» ou «realloc» permettent d'allouer de la mémoire contigüe dynamiquement en fonction de la taille fournie (NbrEléments * **sizeof**(type)), les deux première fonctions ont le même rôle, à la différence que «malloc» n'initialise pas la zone allouée, alors que «calloc» initialise cette zone par des zéros. La fonction «realloc» permet de réajuster l'espace mémoire déjà allouer soit en l'augmentant, soit en le réduisant. En cas d'augmentation de la taille mémoire via cette fonction, il se peut que le système ne trouve pas assez d'espace contigu pour répondre à ce besoin, dans ce cas de figure le système va rechercher un autre emplacement mémoire contigu rependant à la demande formulée, puis recopie les données de l'ancien emplacement vers le nouvel emplacement avant de libérer l'ancien emplacement. Dans ce cas, cette fonction va renvoyer l'adresse mémoire du nouvel emplacement.

Lorsque les trois fonctions d'allocation mémoire ne trouvent pas assez d'espace sur la barrette mémoire pour répondre aux besoins, ces fonctions retournent la valeur «**NULL**». Si tel est le cas, cela pose un problème pour la fonction «realloc», car on risque d'écraser l'adresse mémoire - d'un espace préalablement alloué - stockée dans l'identificateur par la valeur «**NULL**», ce qui aura pour conséquence une fuite de mémoire à cause de la perte de cette adresse. La solution à ce problème consiste à ne jamais stocker directement le résultat envoyé par cette fonction dans une variable contenant déjà une adresse mémoire d'une zone allouée. Il faut en effet utiliser une autre variable qui contiendra la valeur renvoyée. Si cette valeur est différente de «**NULL**», à ce moment on a le choix entre continuer à utiliser cette variable, ou alors écraser l'ancienne adresse de la variable d'origine avec le contenu de la nouvelle variable. Dans la mesure où la valeur renvoyée par «realloc» est «**NULL**», la variable d'origine reste intacte.

Notons que dans le cas de la déclaration de l'identifiant «identifiant4», la fonction «realloc» se comporte de la même manière que la fonction «malloc». En effet, si l'adresse mémoire passée en paramètre à cette fonction

vaut «NULL», cela indique à cette fonction qu'aucune mémoire n'a été allouée, auquel cas, cette fonction va jouer le rôle de «malloc». En d'autres termes :

realloc(NULL , NbrEléments * sizeof(type)) est équivalent à malloc(NbrEléments * sizeof(type))

Exemple :

```
#include <stdio.h>//pour pouvoir utiliser les fonctions printf et scanf
#include <locale.h>//pour pouvoir utiliser la fonction setlocale afin d'imprimer des caractères accentués en C
#include <malloc.h>//pour pouvoir utiliser les fonctions malloc, calloc, realloc et free

int * AllouerEtSaisirVecteurEntiers(unsigned int taille)
{
  int * T = malloc(taille*sizeof(int)), i ;
  if(T == NULL)
  {
    printf("\n La barrette mémoire est saturée !!!") ;
  }
  for(i = 0 ; i < taille ; i++)
  {
    printf("\n Saisir T[%d] : ", i) ;
    scanf("%d", &T[i]) ;
    getchar() ;
  }
  return T ;
}

void  AfficherVecteurEntiers(unsigned int taille, int * V)
{
  int i ;
  if((V == NULL) || (taille == 0))
  {
    printf("\n Le vecteur est vide !!!") ;
    return ;
  }
  for(i = 0 ; i < taille ; i++)
  {
    printf("\n=> V[%d] : %d", i, V[i]) ;
  }
}

char * AllouerEtSaisirVecteurCaracteres(unsigned int taille)
{
  char * T = malloc((taille + 1)*sizeof(char)), i ;
  if(T == NULL)
  {
    printf("\n La barrette mémoire est saturée !!!") ;
  }
  for(i = 0 ; i < taille ; i++)
  {
    printf("\n Saisir T[%d] : ", i) ;
    scanf("%c", &T[i]) ;
    getchar() ;
  }
  T[taille] = '\0' ;
  return T ;
}

void  AfficherVecteurCaracteres(unsigned int taille, char * V)
{
  int i ;
  if((V == NULL) || (taille == 0))
```

```
  {
    printf("\n Le vecteur est vide !!!") ;
    return ;
  }
  for(i = 0 ; i < taille ; i++)
  {
    printf("\n=> V[%d] : %c", i, V[i]) ;
  }
}

void main()
{
  unsigned int Nbe ;
  int * TE ;
  char * TC;
  setlocale(LC_CTYPE, "") ;
  printf("\n Saisir le nombre d'entiers : ") ;
  scanf("%u", &Nbe) ;
  getchar() ;
  TE = AllouerEtSaisirVecteurEntiers(Nbe) ;
  printf("\n_____________________________") ;
  AfficherVecteurEntiers(Nbe, TE) ;
  printf("\n_____________________________") ;
  free(TE) ;
  printf("\n Saisir le nombre de caractères : ") ;
  scanf("%u", &Nbe) ;
  getchar() ;
  TC = AllouerEtSaisirVecteurCaracteres(Nbe) ;
  printf("\n_____________________________") ;
  AfficherVecteurCaracteres(Nbe, TC) ;
  printf("\n_____________________________") ;
  printf("\n La chaine saisie est : %s", TC) ;
  free(TC) ;
  getchar();
}
```

Résultat :

```
 Saisir le nombre d'entiers : 5
 Saisir T[0] : 10
 Saisir T[1] : -20
 Saisir T[2] : 15
 Saisir T[3] : 3
 Saisir T[4] : 6
_____________________________
 Saisir V[0] : 10
 Saisir V[1] : -20
 Saisir V[2] : 15
 Saisir V[3] : 3
 Saisir V[4] : 6
_____________________________
 Saisir le nombre de caractères : 3
 Saisir T[0] : A
 Saisir T[1] : B
 Saisir T[2] : C
_____________________________
=> V[0] : A
=> V[1] : B
=> V[2] : C
_____________________________
 La chaine saisie est : ABC
```

M. Les structures

Le langage «C» permet au programmeur de définir leurs propres types de données. A l'inverse des types primitif (char, int, float, ...), les structures permettent de créer des types de données composés de plusieurs champs hétérogènes. Ces champs sont en fait des variables qui sont associées à des types de données scalaires, multidimensionnels ou même composites (structures).

Syntaxe :

```
struct  identifiant
{
      [type  variable_1 ;]
      [ ... ; ]
      [type  variable_n ;]
} ;
```

Exemple :

```c
#include <stdio.h>
#include <locale.h>
#include <malloc.h>

struct date
{
  unsigned int j ;
  unsigned int m ;
  unsigned int a ;
};
typedef struct date Date ;

struct client
{
  unsigned int Code ;
  char * Nom ;
  char * Prenom ;
  Date DateNaissance ;
};
typedef struct client Client ;
//avec typedef les termes «struct client» seront équivalents au terme «Client» (alias)

char * SaisirChaine()
{
  unsigned int c, nbc = 0 ;
  char * str1 = NULL, * str2 = NULL ;
  while((c = getchar()) != '\n')
  {
    str2 = realloc(str1, (nbc+1)*sizeof(char)) ;
    if(str2 == NULL)
    {
      free(str1) ;
      printf("\n Barrette mémoire saturée !!!") ;
      return NULL ;
    }
    str1 = str2 ;
    str1[nbc++] = c ;
  }
  str2 = realloc(str1, (nbc+1)*sizeof(char)) ;
  if(str2 == NULL)
  {
    free(str1) ;
    printf("\n Barrette mémoire saturée !!!") ;
    return NULL ;
  }
```

```
  str1 = str2 ;
  str1[nbc] = '\0' ;
  return str1 ;
}

void InitClientV1()
{
  Client C = {10, "Benjelloun", "Adil", {10, 3, 1980}} ;
  printf("\n=> Code\t\t : %u", C.Code) ;
  printf("\n=> Nom\t\t : %s", C.Nom) ;
  printf("\n=> Prénom\t : %s", C.Prenom) ;
  printf("\n=>   DateNaissance   :   %02u/%02u/%04u",   C.DateNaissance.j,      C.DateNaissance.m,
C.DateNaissance.a ) ;
}

void InitClientV2()
{
  unsigned int i, nbc = 3 ;
  Client TC[3] ;

  printf("\n-------------- Saisie de 3 clients dans un tableau de clients --------------") ;
  for(i = 0 ; i < nbc ; i++)
  {
    printf("\n=> Saisie du client : %d", i+1) ;
    TC[i].Code = i + 1 ;
    printf("\n\t Saisir le nom : ") ;
    TC[i].Nom = SaisirChaine() ;
    printf("\n\t Saisir le prénom : ") ;
    TC[i].Prenom = SaisirChaine() ;
    printf("\n\t Saisir la date de naissance j/m/a : ") ;
    scanf("%u/%u/%u",       &(TC[i].DateNaissance.j)       ,       &(TC[i].DateNaissance.m)       ,
&(TC[i].DateNaissance.a)) ;
    getchar() ;
  }
  printf("\n-------------- Affichage des 3 clients --------------") ;
  for(i = 0 ; i < nbc ; i++)
  {
    printf("\n=> Code\t\t : %u", TC[i].Code) ;
    printf("\n=> Nom\t\t : %s", TC[i].Nom) ;
    printf("\n=> Prénom\t : %s", TC[i].Prenom) ;
    printf("\n=> DateNaissance : %02u/%02u/%4u", TC[i].DateNaissance.j,  TC[i].DateNaissance.m,
TC[i].DateNaissance.a) ;
    free(TC[i].Nom) ;
    free(TC[i].Prenom) ;
    printf("\n--------------------------------") ;
  }
}

void InitClientV2()
{
  unsigned int i, nbc = 3 ;
  Client * TC[3] ;

  printf("\n-------------- Saisie de 3 clients dans un tableau de pointeurs sur des clients -------
-------") ;
  for(i = 0 ; i < nbc ; i++)
  {
    TC[i] = malloc(sizeof(Client)) ;
    if(TC[i] == NULL)
    {
      printf("\n Barrette mémoire saturée !!!") ;
      return ;
    }
    printf("\n=> Saisie du client : %d", i+1) ;
    TC[i]->Code = i + 1 ;
    printf("\n\t Saisir le nom : ") ;
    TC[i]->Nom = SaisirChaine() ;
    printf("\n\t Saisir le prénom : ") ;
```

```
    TC[i]->Prenom = SaisirChaine() ;
    printf("\n\t Saisir la date de naissance jj/mm/aaaa : ") ;
    scanf("%u/%u/%u",   &(TC[i]->DateNaissance.j)   ,   &(TC[i]->DateNaissance.m)   ,   &(TC[i]->DateNaissance.a)) ;
    getchar() ;
  }
  printf("\n-------------- Affichage des 3 clients --------------") ;
  for(i = 0 ; i < nbc ; i++)
  {
    printf("\n=> Code\t\t : %u", TC[i]->Code) ;
    printf("\n=> Nom\t\t : %s", TC[i]->Nom) ;
    printf("\n=> Prénom\t : %s", TC[i]->Prenom) ;
    printf("\n=> DateNaissance : %02u/%02u/%4u", TC[i]->DateNaissance.j,  TC[i]->DateNaissance.m, TC[i]->DateNaissance.a) ;
    free(TC[i].Nom) ;
    free(TC[i].Prenom) ;
    free(TC[i]) ;
    printf("\n------------------------------") ;
  }
}

void main()
{
    getchar();
}
```

Résultat :

```
=> Code          : 10
=> Nom           : Benjelloun
=> Prénom        : Adil
=> DateNaissance : 10/03/1980
-------------- Saisie de 3 clients dans un tableau de clients --------------
=> Saisie du client : 1
         Saisir le nom : Nadif
         Saisir le prénom : Adil
         Saisir la date de naissance jj/mm/aaaa : 01/01/2001
=> Saisie du client : 2
         Saisir le nom : Zaki
         Saisir le prénom : Said
         Saisir la date de naissance jj/mm/aaaa : 02/02/2002
=> Saisie du client : 3
         Saisir le nom : Alami
         Saisir le prénom : Ahmed
         Saisir la date de naissance jj/mm/aaaa : 03/03/2003
-------------- Affichage des 3 clients --------------
=> Code          : 1
=> Nom           : Nadif
=> Prénom        : Adil
=> DateNaissance : 01/01/2001
------------------------------
=> Code          : 2
=> Nom           : Zaki
=> Prénom        : Said
=> DateNaissance : 02/02/2002
------------------------------
=> Code          : 3
=> Nom           : Alami
=> Prénom        : Ahmed
=> DateNaissance : 03/03/2003
-------------- Saisie de 3 clients dans un tableau de pointeurs sur des clients --------------
=> Saisie du client : 1
         Saisir le nom : Nadif
         Saisir le prénom : Adil
         Saisir la date de naissance jj/mm/aaaa : 01/01/2001
=> Saisie du client : 2
         Saisir le nom : Zaki
         Saisir le prénom : Said
```

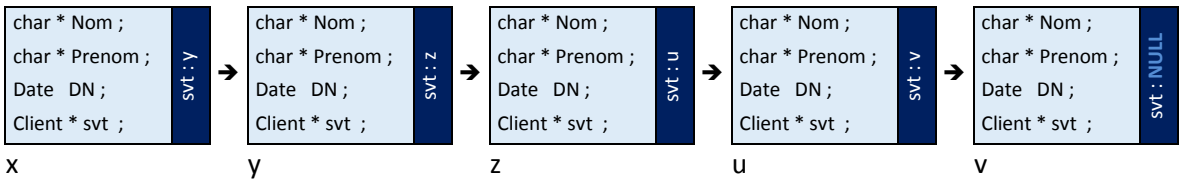
```
        Saisir la date de naissance jj/mm/aaaa : 02/02/2002
=> Saisie du client : 3
        Saisir le nom : Alami
        Saisir le prénom : Ahmed
        Saisir la date de naissance jj/mm/aaaa : 03/03/2003
-------------- Affichage des 3 clients --------------
=> Code          : 1
=> Nom           : Nadif
=> Prénom        : Adil
=> DateNaissance : 01/01/2001
------------------------------
=> Code          : 2
=> Nom           : Zaki
=> Prénom        : Said
=> DateNaissance : 02/02/2002
------------------------------
=> Code          : 3
=> Nom           : Alami
=> Prénom        : Ahmed
=> DateNaissance : 03/03/2003
------------------------------
```

N. Implémentation des structures chainées

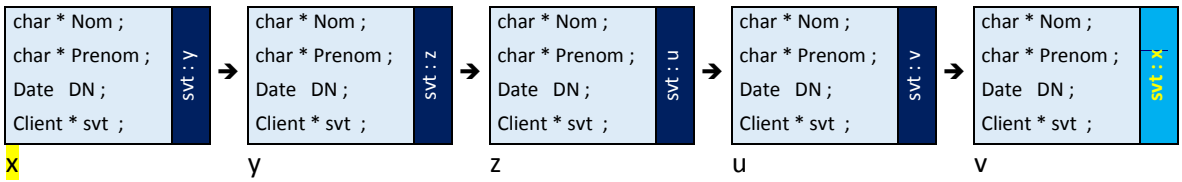
On a vu précédemment que les tableaux statiques permettent de regrouper plusieurs valeurs de même type à travers un seul identificateur qui est associé à une zone mémoire contiguë. Pour ce type de tableaux la taille mémoire doit être connues avant même de compiler le code source, ce qui ne permet de rajouter de nouveaux éléments ou d'en supprimer. La solution à ce problème consiste à utiliser les tableaux dynamiques. Cependant, l'ajout ou la suppression d'un élément au milieu d'un tableau dynamique, nécessite d'effectuer des décalages, ce qui peut présenter un inconvénient de taille pour certains algorithmes.

Les listes est un mécanisme qui consiste à lier les différents éléments - à regrouper - à travers la mémorisation des adresses mémoire pour créer ainsi un chainage.

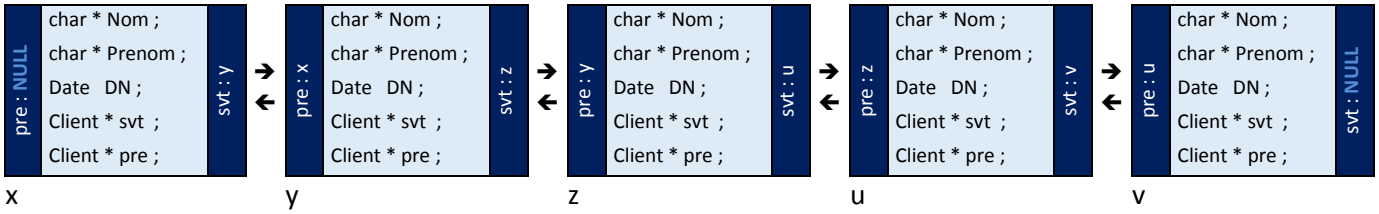
Exemple : chainage simple (x, y, z, u, v : représentent les adresses des différentes structures)



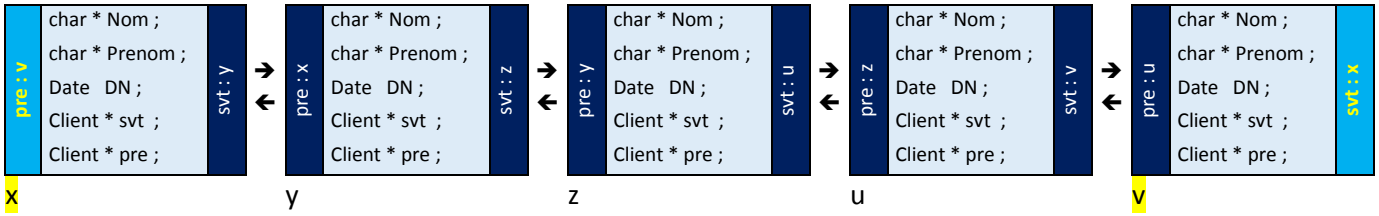
Exemple : chainage simple circulaire (x, y, z, u, v : représentent les adresses des différentes structures)



Exemple : chainage double (x, y, z, u, v : représentent les adresses des différentes structures)



Exemple : chainage double circulaire (x, y, z, u, v : représentent les adresses des différentes structures)



Notons, qu'en général, les différentes structures seront allouées dynamiquement durant la phase d'exécution du programme, puis chainées au fur et à mesure de leur création. Cela sous-entend que les zones mémoires allouées dans ce contexte ne sont pas forcément contiguës. L'ajout ou la suppression d'un élément ne nécessitera donc pas de décalage. Il suffit en effet de casser le chainage pour le reconstituer à l'endroit souhaité.

Par ailleurs, étant donné que l'allocation se fera en général de façon dynamique - avec éventuellement la possibilité d'insérer et/ou de supprimer des éléments en début, au milieu ou encore en fin de chainage -, on aura besoin de repérer l'adresse mémoire du premier élément et éventuellement celle du dernier.

Notons que dans le cas du chainage simple, le parcours de la liste ne peut se faire que du début vers la fin du chainage. La fin du chainage doit être signalée dans ce cas par le caractère «**NULL**» (svt = **NULL** pour le dernier élément). Tandis que pour le chainage double, on aura la possibilité d'effectuer ce parcours dans les deux sens, ce qui nous conduit à limiter le premier et le dernier élément par le caractère «**NULL**» (pre = **NULL** pour le premier élément, et svt = **NULL** pour le dernier élément). Dans le cas du chainage circulaire simple, le caractère «**NULL**» n'aura plus aucun rôle, puisque le dernier élément doit pointer sur le premier (svt = x). Dans le même contexte, dans le cas du chainage circulaire double, le premier élément et le dernier seront liés entre à travers la mémorisation de leurs adresses respectives (pre = v pour le premier élément, et svt = x pour le dernier élément).

La notion de chainage permet de définir plusieurs types abstraits de structures, tels que les listes, les files, les piles, les arbres ou encore les graphes. Nous verrons dans les sections suivantes comment implémenter ces structures, ainsi que les opérations usuelles qui s'y rattachent comme le parcours, la recherche, l'insertion ou la suppression.

N.1. Implémentation de la liste simplement chainée

```c
#include<stdio.h>
#include<locale.h>
#include<malloc.h>

extern char * SaisirChaine() ;

struct date
{
   unsigned int j ;
   unsigned int m ;
   unsigned int a ;
};
typedef struct date Date ; //avec typedef les termes «struct date» seront équivalents au terme «Date» (alias)

struct client
{
   unsigned int Code ;
   char * Nom ;
   char * Prenom ;
   Date DateNaissance ;
   struct client * svt ;
};
typedef struct client Client ;
//avec typedef les termes «struct client» seront équivalents au terme «Client» (alias)
```

```c
static unsigned int CC = 0 ;
static Client * DL = NULL ;
static Client * FL = NULL ;

void AfficherClient_LSC(Client * clt)
{
   if(clt != NULL)
   {
      printf("\n=> Code\t\t : %u", clt->Code) ;
      printf("\n=> Nom\t\t : %s", clt->Nom) ;
      printf("\n=> Prénom\t : %s", clt->Prenom) ;
      printf("\n=> DateNaissance : %02u/%02u/%4u", clt->DateNaissance.j,  clt->DateNaissance.m,
                                                   clt->DateNaissance.a) ;
      printf("\n------------------------------") ;
   }
   else
   {
      printf("\n Le client n'existe pas ou liste vide !!!") ;
   }
}

void SaisirClient_LSC(Client * clt)
{
   if(clt != NULL)
   {
      clt->Code = ++CC ;
      printf("\n=> Saisir le nom du client : ") ;
      clt->Nom = SaisirChaine() ;
      printf("\n=> Saisir le prénom du client : ") ;
      clt->Prenom = SaisirChaine() ;
      printf("\n\t Saisir la date de naissance jj/mm/aaaa : ") ;
      scanf("%u/%u/%u", &(clt->DateNaissance.j) , &(clt->DateNaissance.m) , &(clt->DateNaissance.a)) ;
   }
}

void AfficherClients_LSC()
{
   Client * clt = DL ;

   while(clt != NULL)
   {
      AfficherClient_LSC(clt) ;
      clt = clt->svt ;
   }
}

void AjouterClientDL_LSC()
{
   Client * clt = malloc(sizeof(Client)) ;
   if(clt == NULL)
   {
      printf("\n Barrette mémoire saturée !!!") ;
      return ;
   }
   SaisirClient_LSC(clt) ;
   clt->svt = DL ;
   DL = clt ;
   if(FL == NULL)
   {
      FL = clt ;
   }
}

void AjouterClientFL_LSC()
{
   Client * clt = malloc(sizeof(Client)) ;
   if(clt == NULL)
   {
      printf("\n Barrette mémoire saturée !!!") ;
      return ;
   }
   SaisirClient_LSC(clt) ;
   clt->svt = NULL ;
   if(FL != NULL)
   {
```

```
      FL->svt = clt ;
   }
   else
   {
      DL = clt ;
   }
   FL = clt ;
}

Client * AdresseClient_LSC(unsigned int code, Client ** pre)
{
   Client * clt = DL ;

   while((clt != NULL) && (clt->Code != code))
   {
      *pre = clt ;
      clt = clt->svt ;
   }
   return clt ;
}

void OrdonnerClientsViaCode_LSC()
{
   Client * u, * v ;
   char * nom, * prenom ;
   unsigned int code ;
   Date dn ;

   for(u = DL ; u != FL ; u = u->svt)
   {
      for(v = u->svt ; v != NULL ; v = v->svt)
      {
         if(u->Code > v->Code)
         {
            nom = u->Nom ;
            u->Nom = v->Nom ;
            v->Nom = nom ;
            prenom = u->Prenom ;
            u->Prenom = v->Prenom ;
            v->Prenom = prenom ;
            code = u->Code ;
            u->Code = v->Code ;
            v->Code = code ;
            dn = u->DateNaissance ;
            u->DateNaissance = v->DateNaissance ;
            v->DateNaissance = dn ;
         }
      }
   }
}

void SupprimerClient_LSC(unsigned int code)
{
   Client * clt, * pre ;
   clt = AdresseClient_LSC(code, &pre) ;
   if(clt == NULL)
   {
      printf("\n Le client n'existe pas dans la liste !!!") ;
      return ;
   }
   if((clt == DL) && (clt == FL))
   {
      DL = NULL ;
      FL = NULL ;
   }
   else if(clt == DL)
   {
      DL = DL->svt ;
   }
   else if(clt == FL)
   {
      pre->svt = NULL ;
      FL = pre ;
   }
   else
   {
```

```
      pre->svt = clt->svt ;
   }
   free(clt->Nom) ;
   free(clt->Prenom) ;
   free(clt) ;
}


void Menu_LSC()
{
   int choix = -1 ;
   unsigned int cc ;

   while(choix != 0)
   {
      printf("\n ------- MENU LISTE SIMPLEMENT CHAINNEE -------\n") ;
      printf("\n 0) Quitter le programme") ;
      printf("\n 1) Ajouter un client en début de liste") ;
      printf("\n 2) Ajouter un client en fin de liste") ;
      printf("\n 3) Afficher la liste des clients") ;
      printf("\n 4) Ordonner les client selon le code") ;
      printf("\n 5) Rechercher un client via son code") ;
      printf("\n 6) Supprimer un client via son code") ;
      printf("\n\t Saisir votre choix [0 6] : ") ;
      scanf("%d", &choix) ;
      getchar() ;

      switch(choix)
      {
         case 0 :
            printf("\n Sortie du programme") ;
            return ;
         case 1 :
            printf("\n Ajouter un client en début de liste") ;
            AjouterClientDL_LSC() ;
            break ;
         case 2 :
            printf("\n Ajouter un client en fin de liste") ;
            AjouterClientFL_LSC() ;
            break ;
         case 3 :
            printf("\n Afficher la liste des clients") ;
            AfficherClients_LSC() ;
            break ;
         case 4 :
            printf("\n Ordonner les client selon le code") ;
            OrdonnerClientsViaCode_LSC() ;
            break ;
         case 5 :
            printf("\n Rechercher un client via son code") ;
            printf("\n Saisir le code du client : ") ;
            scanf("%u", &cc) ;
            getchar() ;
            AfficherClient_LSC(AdresseClient_LSC(cc, NULL)) ;
            break ;
         case 6 :
            printf("\n Supprimer un client via son code") ;
            printf("\n Saisir le code du client : ") ;
            scanf("%u", &cc) ;
            getchar() ;
            SupprimerClient_LSC(cc) ;
            break ;
         default :
            printf("\n Option non gérée") ;
            break ;
      }
   }
}

void main()
{
   Menu_LSC() ;
   getchar();
}
```

N.1. Implémentation de la liste doublement chainée

```c
#include<stdio.h>
#include<locale.h>
#include<malloc.h>

extern char * SaisirChaine() ;

struct date
{
   unsigned int j ;
   unsigned int m ;
   unsigned int a ;
};
typedef struct date Date ;
//avec typedef les termes «struct date» seront équivalents au terme «Date» (alias)

struct client
{
   unsigned int Code ;
   char * Nom ;
   char * Prenom ;
   Date DateNaissance ;
   struct client * svt ;
   struct client * pre ;
};
typedef struct client Client ;
//avec typedef les termes «struct client» seront équivalents au terme «Client» (alias)

static unsigned int CC = 0 ;
static Client * DL = NULL ;
static Client * FL = NULL ;

void AfficherClient_LDC(Client * clt)
{
   if(clt != NULL)
   {
      printf("\n=> Code\t\t : %u", clt->Code) ;
      printf("\n=> Nom\t\t : %s", clt->Nom) ;
      printf("\n=> Prénom\t : %s", clt->Prenom) ;
      printf("\n=> DateNaissance : %02u/%02u/%4u", clt->DateNaissance.j,  clt->DateNaissance.m,
                                                   clt->DateNaissance.a) ;
      printf("\n-------------------------------") ;
   }
   else
   {
      printf("\n Le client n'existe pas ou liste vide !!!") ;
   }
}

void SaisirClient_LDC(Client * clt)
{
   if(clt != NULL)
   {
      clt->Code = ++CC ;
      printf("\n=> Saisir le nom du client : ") ;
      clt->Nom = SaisirChaine() ;
      printf("\n=> Saisir le prénom du client : ") ;
      clt->Prenom = SaisirChaine() ;
      printf("\n\t Saisir la date de naissance jj/mm/aaaa : ") ;
      scanf("%u/%u/%u", &(clt->DateNaissance.j) , &(clt->DateNaissance.m) , &(clt->DateNaissance.a)) ;
   }
}

void AfficherClients_LDC(char sens)
{
   Client * clt ;
   if((sens == 'D') || (sens == 'd'))
   {
      clt =  DL ;
      while(clt != NULL)
      {
         AfficherClient_LSC(clt) ;
         clt = clt->svt ;
      }
```

```
   }
   else if((sens == 'F') || (sens == 'f'))
   {
      clt =  FL ;
      while(clt != NULL)
      {
         AfficherClient_LSC(clt) ;
         clt = clt->pre ;
      }
   }
   else
   {
      printf("\n Option d'affichage : sens d'affichage non valide\n") ;
   }
}

void AjouterClientDL_LDC()
{
   Client * clt = malloc(sizeof(Client)) ;
   if(clt == NULL)
   {
      printf("\n Barrette mémoire saturée !!!") ;
      return ;
   }
   SaisirClient_LDC(clt) ;
   clt->pre = NULL ;
   clt->svt = DL ;
   if(DL != NULL)
   {
      DL->pre = clt ;
   }
   else
   {
      FL = clt ;
   }
   DL = clt ;
}

void AjouterClientFL_LDC()
{
   Client * clt = malloc(sizeof(Client)) ;
   if(clt == NULL)
   {
      printf("\n Barrette mémoire saturée !!!") ;
      return ;
   }
   SaisirClient_LDC(clt) ;
   clt->svt = NULL ;
   clt->pre = FL ;
   if(FL != NULL)
   {
      FL->svt = clt ;
   }
   else
   {
      DL = clt ;
   }
   FL = clt ;
}

Client * AdresseClient_LDC(unsigned int code)
{
   Client * clt = DL ;

   while((clt != NULL) && (clt->Code != code))
   {
      clt = clt->svt ;
   }
   return clt ;
}

void OrdonnerClientsViaCode_LDC()
{
   Client * u, * v ;
   char * nom, * prenom ;
   unsigned int code ;
```

```
   Date dn ;

   for(u = DL ; u != FL ; u = u->svt)
   {
      for(v = u->svt ; v != NULL ; v = v->svt)
      {
         if(u->Code > v->Code)
         {
            nom = u->Nom ;
            u->Nom = v->Nom ;
            v->Nom = nom ;
            prenom = u->Prenom ;
            u->Prenom = v->Prenom ;
            v->Prenom = prenom ;
            code = u->Code ;
            u->Code = v->Code ;
            v->Code = code ;
            dn = u->DateNaissance ;
            u->DateNaissance = v->DateNaissance ;
            v->DateNaissance = dn ;
         }
      }
   }
}

void SupprimerClient_LDC(unsigned int code)
{
   Client * clt ;
   clt = AdresseClient_LDC(code) ;
   if(clt == NULL)
   {
      printf("\n Le client n'existe pas dans la liste !!!") ;
      return ;
   }
   if((clt == DL) && (clt == FL))
   {
      DL = NULL ;
      FL = NULL ;
   }
   else if(clt == DL)
   {
      DL = DL->svt ;
      DL->pre = NULL ;
   }
   else if(clt == FL)
   {
      FL = FL->pre ;
      FL->svt = NULL ;
   }
   else
   {
      clt->pre->svt = clt->svt ;
      clt->svt->pre = clt->pre ;
   }
   free(clt->Nom) ;
   free(clt->Prenom) ;
   free(clt) ;
}

void Menu_LDC()
{
   int choix = -1 ;
   unsigned int cc ;
   char sens ;

   while(choix != 0)
   {
      printf("\n ------- MENU LISTE DOUBLEMENT CHAINNEE -------\n") ;
      printf("\n 0) Quitter le programme") ;
      printf("\n 1) Ajouter un client en début de liste") ;
      printf("\n 2) Ajouter un client en fin de liste") ;
      printf("\n 3) Afficher la liste des clients") ;
      printf("\n 4) Ordonner les client selon le code") ;
      printf("\n 5) Rechercher un client via son code") ;
      printf("\n 6) Supprimer un client via son code") ;
      printf("\n\t Saisir votre choix [0 6] : ") ;
```

```
      scanf("%d", &choix) ;
      getchar() ;

      switch(choix)
      {
         case 0 :
            printf("\n Sortie du programme") ;
            return ;
         case 1 :
            printf("\n Ajouter un client en début de liste") ;
            AjouterClientDL_LDC() ;
            break ;
         case 2 :
            printf("\n Ajouter un client en fin de liste") ;
            AjouterClientFL_LDC() ;
            break ;
         case 3 :
            printf("\n Afficher la liste des clients") ;
            printf("\n Saisir le sens de l'affichage (D ou F) : ") ;
            scanf("%c", &sens) ;
            getchar() ;
            AfficherClients_LDC(sens) ;
            break ;
         case 4 :
            printf("\n Ordonner les client selon le code") ;
            OrdonnerClientsViaCode_LDC() ;
            break ;
         case 5 :
            printf("\n Rechercher un client via son code") ;
            printf("\n Saisir le code du client : ") ;
            scanf("%u", &cc) ;
            getchar() ;
            AfficherClient_LDC(AdresseClient_LDC(cc)) ;
            break ;
         case 6 :
            printf("\n Supprimer un client via son code") ;
            printf("\n Saisir le code du client : ") ;
            scanf("%u", &cc) ;
            getchar() ;
            SupprimerClient_LDC(cc) ;
            break ;
         default :
            printf("\n Option non gérée") ;
            break ;
      }
   }
}

void main()
{
   Menu_LDC() ;
   getchar();
}
```

N.1. Implémentation de l'arbre binaire