INTRODUCTION À LA PROGRAMMATION – PRATIQUE DU LANGAGE C –

M. Le Gonidec

Table des matières

Chapitre 1

Introduction

1.1 Un peu d'histoire...

Les langages de programmation ont fait leur apparition aux environs de 1950, mais le souci d'automatisation des calculs date de bien avant cela. Ainsi les grecs, les chinois savaient calculer, automatiser des calculs (à la main, en utilisant par exemple un boulier).

En 820, le mathématicien Al Khawarizmi a publié à Bagdad un traité intitulé "La science de l'élimination et de la réduction" qui, importé en Europe Occidentale lors des invasions arabes a eu une grande influence sur le développement des mathématiques.

Si on veut vraiment retourner aux racines, il faut continuer en citant les cartes perforées. En 1725, Basile Bouchon inventa le principe des cartes perforés qui permettent, en 1801, aux métiers Jacquard de réaliser des motifs complexes. Si l'on considère que la manière particulière de perforer les cartes produit un effet tout aussi particulier, qu'un trou peut correspondre à un 1 en langage binaire, qu'elles ont été intégrées dans une machine considérée aujourd'hui comme l'ancêtre de l'ordinateur alors les cartes perforées sont les ancêtres des programmes informatiques.

Mais la programmation "moderne" mathématique, commence sans doute en 1840, grâce à Lady Ada Lovelace (1814-1852), qui définit le principe des itérations successives dans l'exécution d'une opération. En l'honneur d'Al Khawarizmi elle a nommé "algorithme" le processus logique d'exécution d'un programme. Elle est aussi à l'origine du **Principe des machines à calculer**. Pour elle, une machine à calculer devait comporter :

1. un dispositif permettant d'introduire les données numériques (cartes perforées, roues dentées...),
2. une mémoire pour conserver les valeurs numériques entrées,
3. une unité de commande qui indique les tâches à effectuer,
4. un "moulin" chargé d'effectuer les calculs,
5. un dispositif permettant de prendre connaissance des résultats (imprimante...).

Ces principes seront, un siècle plus tard, à la base des premiers ordinateurs. On a en quelque sorte la même décomposition :

1. l'introduction des données s'effectue grâce à un périphérique d'entrée (clavier, souris, caméra, micro...)
2. la mémoire servant à conserver les données sous différentes formes : registres (mémoire interne au processeur), mémoire cache, mémoire vive, mémoire de masse...
3. L'unité de commande (Partie Commande du processeur) indique et coordonne l'éxécution les tâches,
4. le "moulin" est l'unité arithmétique et logique (ALU) du processeur,

5. la récupérations des résultats s'effectue via un périphérique de sortie (écran, enceintes, imprimante...).

Le principal problème C'est que l'unité de commande d'un ordinateur ne comprend que le binaire ainsi que les opération logiques de base... En 1854, Boole publie un ouvrage dans lequel il démontre que tout processus logique peut être décomposé en une suite d'opérations logiques (ET, OU, NON) appliquées sur deux états (ZERO-UN, OUI-NON, VRAI-FAUX, OUVERT-FERME). En 1950, l'invention du langage assembleur par Maurice V. Wilkes de l'université de Cambridge ouvre la voie aux langages dits de haut niveau. Avant, la programmation s'effectuait directement en binaire. Grace Hopper, une américaine mobilisée comme auxiliaire dans la marine américaine, développe pour Remington Rand le tout premier compilateur, nommé **A0** . Il permet de générer un programme binaire à partir d'un "code source".

Alors que le langage **Fortran** (FORmula TRANslator, utilisé avant tout pour des applications numériques) commence à apparaitre vers 1955 notamment chez IBM, Grace Hopper s'intéresse aux langages qui utilisent des mots voire des expressions du "langage naturel". Après B0 et Flowmatic, elle participe dès 1959. à l'élaboration de ce qui deviendra le langage **Cobol** (Commun Business Oriented Langage, utilisé principalement pour la gestion). Dans le même temps (soit vers 1958) John Mc Carthy, mathématicien au MIT qui y a fondé en 1957 le département d'Intelligence Artificielle, crée le langage **Lisp** (très adapté pour l'intelligence artificelle).

Avec la fin des années 50 s'achève ce qu'on nomme aujourd'hui l'ère des ordinateurs de première génération qui utilisent principalement les cartes perforées. La seconde génération, celle qui utilise les transistors va prendre pleinement son essor.

Fortran et Cobol s'installent comme langages principaux : pendant plus 20 ans Cobol sera le langage le plus utilisé au monde. Aujourd'hui, il détient toujours le record du grand nombre de lignes écrites dans un langage de programmation. Par contre, Lisp reste cantonné à la communauté de l'intelligence artificielle. Si **Algol** (ALGOrithmic Language) devient un langage pour la publication d'algorithmes, il sera peu implémenté. Modifié en Algol W puis en Algol68, il ne parvient pas à devenir le langage standard universel qu'il aurait du être sans doute parce que trop complet pour l'époque et trop difficile à implanter pour ces années 60. Cela dit, L'Algol introduit des concepts qui seront largement utilisés dans les langages qui suivront : notions de bloc et de programmation structurée, récursivité...

Thomas Kurtz et John Kemeny créent en 1964 un langage (beaucoup plus accessible que l'Algol) au Dartmouth College pour leurs étudiants. ce langage pour "débutants" se nomme **BASIC** (Beginner's All purpose Symbolic Instruction Code) .

Depuis 1968 Niklaus WIRTH développe Algol. A force de le modifier, il finit par mettre au point un successeur d'Algol nommé le langage **PASCAL** (en l'honneur de ce cher Blaise...). Ce langage, moins puissant qu'Algol, est bien structuré, très lisible, trés "coercitif" et se trouve donc bien adapté à l'enseignement de la programmation.

Dès 1970, Ken Thompson, pensant que le système d'exploitation UNIX ne serait pas complet sans un langage de programmation de haut niveau, crée un nouveau langage, le **B** (en référence au BCPL dont il s'inspire). Deux ans plus tard, Dennis Ritchie du Bell Lab d'ATT reprend ce langage B pour mieux l'adapter au PDP/11 sur lequel UNIX vient juste d'être porté. Il fait évoluer le langage et le dote d'un vrai compilateur générant du code machine PDP/11 : c'est la naissance du langage **C**.

En 1970 aussi, le Département de la défense américaine trouve qu'il y a beaucoup trop d'assembleur dans les systèmes embarqués et aimerait utiliser un "bon" langage de programmation. En 1975, une groupe d'études ds penche sur 23 langages déjà existants. La conclusion de l'analyse sera qu'il faudrait un nouveau langage, sans doute basé à la fois sur Pascal, Algol et Pl/1. Après un appel d'offre et plusieurs sélections, c'est finalement en 1979 le projet de Jean Ichbiach, avec l'équipe d'Honeywell Bull de Marseille qui sera retenu. Ce langage sera nommé Dod-1 puis **Ada**. Ce choix par le Pentagone Américain comme l'unique langage de développement imposé à ses services à la place de la jungle des centaines de langages et dialectes aurait du en faire le premier langage de développement au monde

mais la difficulté à maitriser Ada en fera un "outsider".

Toujours au début des années 1970 (décidément !), Philippe Roussel et Alain Colmerauer dans leur Groupe d'Intelligence Artificielle de Marseille développent un langage qui fonctionne de façon totalement différente des autres langages : on y programme logiquement c'est à dire en spécifiant seulement des relations logiques : il se nomme **Prolog** et devient le langage privilégié pour l'Intelligence Artificielle.

Le début des années 80 consacre le développement de la petite informatique et de la micro-informatique : on y voit naitre les premiers PC et les premiers Apple (mais Windows n'existe pas encore). **Dbase** va s'imposer dans ces années 80 comme LE gestionnaire de "bases de données relationnelles" lorsque l'approche tableur (Multiplan, Visicalc...) n'est pas adaptée.

Pourtant, la révolution objet est en marche ; elle permet d'écrire de plus gros programmes mieux structurés, plus facilement modifiables et plus sûrs. En 1983 Bjarn Stroustrup développe une extension orientée objet au langage C qui deviendra le langage **C++** dans les mêmes laboratoires dans lesquels a vu le jour le C de Dennis Ritchie. La naissance de l'**Objective-C** arrive dans le même cadre, Brad Cox le met au point au début des années 1980. Le langage est basé sur un autre, le **Smalltalk-80**, et est destiné à être une couche supplémentaire au C pour permettre la création et la manipulation d'objets. Le code compilé Objective-C s'exécute dans un environnement d'exécution très léger (runtime) écrit en C qui ajoute peu à la taille de l'application. Le premier système d'exploitation à utiliser Objective-C fut NeXTStep, de la société NeXT, fondée par Steve Jobs... et naturellement, Objective-C sera par la suite beaucoup utilisé sur Macintosh, notamment pour les API Cocoa de Mac OS X.

Un langage complètement objet nommé **Eiffel** et mis au point par Bertrand Meyer sort en 1986. Il faudra pourtant quelques années avant que les "objets" deviennents prépondérants en programmation...

Cette même année 1986, Larry Wall, programmeur système mais aussi linguiste, décide de parfaire les outils Unix de traitement d'informations texte : le langage **Perl** vient de naitre. Il reprend des fonctionnalités du langage C et des langages de scripts. Grâce au Web et à ces scripts (programmes) parfois très courts, Perl devient un outil indispensable pour gérer les fichiers-textes notamment l'extraction d'informations et la génération de rapports. Sa souplesse autorise l'emploi de plusieurs modèles de programmation : procédurale, fonctionnelle et orientée objet.

Vers la fin des années 80, les langages de commandes et de scripts se développent pour tous les types d'ordinateurs. Parallèlement, la notion d'interface graphique pour utilisateur (GUI) commence à entrer dans les moeurs pour les "grands systèmes" ; John Osterout invente à l'Université de Californie (Berkeley) en 1988 les langages Tcl et Tk pour des "développements rapides" : Tcl est la partie "scripts de commandes" dont Tk produit l'interface. Le langage est connu depuis sous le nom de **Tcl/Tk**.

Les mathématiques ne sont pas en reste : dès la fin des années 80, Stephen Wolfram développe un langage et environnement pour programmer des mathématiques formelles et appliquées : **Mathematica** alors qu'un autre langage au début sans interface autre que la ligne de commande, commence a émerger : le langage **Maple**.

Enfin, même s'il n'est pas vraiment considéré comme un langage de programmation, le langage **HTML** (Hypertext Markup Language) est développé en 1989 par Tim Berners-Lee. Ce sera "LE" langage du Web.

En 1989, c'est également l'année où le langage C est définitivement normalisé par l'ANSI (American National Standards Institute). Ce travail aura duré 6 ans et s'achève par la définition de la norme ANSI C.

Les années 90 voient s'installer un peu partout dans le monde Windows. C'est en 1990 que Microsoft sort son produit Windows 3.0 qui est une version complétement revue des premiers Microsoft Windows. Un an plus tard, mais sans publicité, Linux 0.01 est annoncé par un étudiant, Linus Torvald à l'Université d'Helsinki. Linux va se développer très rapidement grâce à Internet et grâce à deux concepts-phare : la disponibilité du code-source des programmes et la gratuité (entre autres), suivant

en cela le projet GNU de Richard Stallman, le fameux créateur de l'éditeur emacs. Dans le même temps, les laboratoires de Sun étoffent Tcl pour en faire un langage de script universel adapté à Internet et le portent pour Windows et Macintosh. Tk va devenir une "sur-couche" de nombreux langages dont Rexx, Perl...

En 1991, Guido van Rossum crée le langage **Python** (en référence au Monty Python...). Ce langage est particulièrement répandu dans le monde scientifique, et possède de nombreuses extensions destinées aux applications numériques. Il est à la base du projet **SAGE**, un logiciel open-source de mathématiques sous licence GPL qui combine la puissance de nombreux programmes open-source en une interface commune basée sur Python. Le projet SAGE est avant tout destiné à crée une alternative viable libre et open source à Magma, Maple, Mathematica et autres Matlab.

Le développement très fort d'Internet influence fortement les concepteurs de langage. En 1995, suite à de nombreuses réunions de comité du WWW, le langage LiveScript est renommé en **Javascript** et est considéré comme une "bonne" solution pour gérer dynamiquement les pages Web. Il est aussitot incorporé dans Netscape 2. Mais il manque toujours un langage complet, objet, capable de dialoguer avec les serveurs Web et les bases de données. Dans la même année 95, **Java** est introduit comme langage de développement objet multi-OS pour combler ce manque.

La gestion des formulaires et des bases de données accessibles par le Web voit apparaitre pratiquement en même temps le langage **Php**, souvent couplé au langage de base de données **Sql** notamment dans ses implémentations **MySql** et **PosgresSql**.

Parallèlement, l'utilisation répandues des logiciels Word et Excel induit progressivement l'utilisation de "macros" pour tirer pleinement profit des possibilités de ces logiciels : le langage Basic, remanié, rendu objet avec ses fonctions liées aux documents devient le **Visual Basic**.

Le début du XXIème siècle n'a pas été très riche en nouveaux langages mais plusieurs d'entre eux ont acquis une certaine importance. En l'an 2000, Manfred vos Thun conçoit **Joy**, Walter Bright développe le langage **D**, et Microsoft dévoile **C#** avec l'implication de Anders Hejlsberg, créateur du langage **Delphi** cinq ans auparavant. Les inspirations du C# vont du C au C++ en passant par le Java dont il reprend généralement la syntaxe, mais aussi par le Pascal dont il hérite de la surcharge des opérateurs. Outre officiellement avoir été conçu pour exploiter tout le potentiel de la plateforme .Net de Microsoft, C# a également été développé pour s'affranchir de la plateforme Java de Sun, avec qui Microsoft a eu des démêlés à ce sujet.

Un an plus tard, Microsoft publie **Visual Basic.Net**. Pas un nouveau langage à proprement parler, mais plutôt une évolution, destinée à l'intégration du langage dans la plateforme .Net. La même année, Xerox PARC dévoile **AspectJ**, basé sur Java, apportant à ce langage la programmation orientée aspect. Ce type de programmation permet d'améliorer la séparation des "préoccupations" lors du développement. Ces préoccupations forment les différents aspects techniques du logiciel, habituellement fortement dépendants entre eux. La programmation orientée aspect propose des mécanismes de séparation de ces aspects, afin de modulariser le code, le décomplexifier, et le rendre plus simple à modifier.

C# fera des émules. En 2003, l'université de Wroclaw publie **Nermle** qui s'inspire tout à la fois du langage de Microsoft, de ML et de MetaHaskell. En 2004, Rodrigo de Oliveira développe **Boo** qui ajoute une syntaxe inspirée de Python compatible avec C#. La première version de **Mono**, implémentation libre du Framework .Net de Microsoft, apparaîtra la même année, conçu par Miguel de Icaza, apportant par là même le C#, devant initialement être multiplateforme, au monde du libre, et notamment Linux.

Microsoft, décidément très productif (faut bien justifier les coûts...), dévoilera **F#** en 2005. Lui aussi inspiré du C#, le langage **OCaml** (Objective Categorical Abstract Machine Language) est développé par l'INRIA et l'Ecole Normale Supérieure et Haskell, il ne sera proposé au grand public qu'avec la prochaine génération de l'environnement de développement Microsoft Visual Studio 2010. En 2006, Microsoft - encore - dévoile Windows PowerShell, qui n'est pas qu'un shell mais aussi un

langage de script destiné à l'administration de machines Windows. Même en étant tolérant, c'est une pâle copie des shell Linux. Cela dit, elle constitue une évolution de la console DOS lui apportant une grande puissance.

Le projet GNOME dévoile **Vala** en 2007, destiné à apporter aux développeurs sous Linux une syntaxe très proche du C#. Comme déja plusieurs autres langages, on ne compiler un code source Vala directement en assembleur mais d'abord en C, et c'est ce code en C qui sera compilé par la suite.

On conclura sur le **LOLCODE**, développé en 2007 par Adam Lindsay. Langage ésotérique s'il en est : sa syntaxe est celle de l'anglais SMS... nous vivons une époque moderne...

1.2 Qu'est ce qu'un langage de programmation ?

Un langage de programmation est, d'un point de vue mathématique, un langage formel, c'est-à-dire un ensemble de suites (appellés **mots**) de symboles choisis dans un ensemble donné (appelé **alphabet**) qui vérifient certaines contraintes spécifiques au langage (**syntaxe**). Dans le cas des langages de programmation, on trouve dans l'alphabet plusieurs types de symbôles :

- des mots-clé. Ex : `main`, `int`, `if`, `for` ...,
- des opérateurs. Ex : `=`, `<`,`&` ...,
- des chiffres et des lettres permettant d'identifier des variables ou est constantes,
- des caractères spéciaux : Ex : accolades, crochets, point-virgule, tiret bas ou blanc souligné (underscore)... permettant de structurer le tout...

Un **programme** est un mot du langage, c'est-à-dire une suite de symboles vérifiants les contraintes dictées par la syntaxe du langage.

Comme le but d'un langage est quand même de se faire comprendre de la machine, on a plusieurs possibilités :

1. soit on utilise un **langage de bas niveau** : onécrit un programme directement dans le langage machine (ou langage natif) compréhensible par le processeur, mais rarement intelligible puisque rares sont les personnes qui parlent le binaire courrament.
2. soit on utilise un **langage de haut niveau** : on écrit un programme dans un langage inintélligible pour le processeur, qui sera ensuite "traduit" en langage machine afin que le processeur l'éxecute. Ces langages, plus sympathiquepour l'homme, permettent en quelques sorte de décrire des taches sans se soucier des détails sur la manière dont la machine l'exécute.

Deux stragégies sont utilisées pour les langages de haut niveau. La différence réside dans la manière dont on traduit le programme, qui est à l'origine dans un ou plusieurs fichiers texte appelés **fichiers source** ou **code source** :

1. soit on écrit un programme dans un **langage interprété**. Le code source sera traduit mot à mot ou instruction par instructions, dans le langage machine propre au processeur par un programme auxiliaire : l'**interpreteur**. L'humain fait une requête, cette requête est traduite par l'interpréteur, qui détermine la stratégie d'éxecution, le processeur l'éxecute puis on passe à la requête suivante.

   Chaque éxecution du programme nécessite l'utilisation de l'interpréteur.
2. soit on écrit un programme dans un **langage compilé**. Le code source sera traduit (une bonne fois pour toute) dans le langage machine propre au processeur par un programme **compilateur**. Contrairement aux interpréteurs, les compilateurs lisent entièrement le code source avant de le traduire, détermine la stragégie d'éxecution de toutes les instructions, puis génère un fichier binaire **éxecutable** : le fichier produit n'a plus besoin d'être lancé par un autre programme, il est autonome. Cela permet de garantir la sécurité du code source mais chaque modification nécessite une recompilation.

Langage	domaine d'applications	Compilé/interprété
ADA	Programmation temps réel	compilé
BASIC	Programmation à but éducatif	interprété
C	Programmation système	compilé
C++	Programmation orientée objet	compilé
Cobol	Gestion	compilé
Fortran	Calcul	compilé
Java	Programmation internet	intermédiaire
MATLAB	Calcul mathématique	interprété
Mathematica	Calcul mathématique	interprété
LISP	Intelligence artificielle	intermédiaire
Pascal	Programmation à but éducatif	compilé
PHP	Développementde sites Web dynamiques	interprété
Prolog	Intelligence artificielle	interprété
Perl	traitement chaînes de caractères	interprété

TAB. 1.1 – Quelques exemples de langages courants

Il existe également des langages faisant en quelque sorte partie des deux catégories ci-dessus : les **langages intermédiaires**. Un programme écrit dans de tels langages doit subir une phase de compilation vers un fichier non executable qui necessite un interpréteur.

Si chaque langage de programmation a ses propres particularités, certains présentent des similitudes quant à la manière de programmer. Il y a des "styles", appelés **paradigmes** : Deux programmes repondant au même problème et écrits dans deux langages différents qui utilisent le même paradigme vont se *ressembler*, tandis qu'il seront fondamentalement différents si les deux langages utilisent des paradygmes différents. Les langages de programmation permettent souvent d'utiliser plusieurs de ces styles, dans ce cas, on parle de langage **multiparadigmes**. Ci dessous, quelques exemples de paradygmes :

- **Programmation impérative** : Chaque instruction du langage correspond à un ensemble d'instruction du langage machine. Les structures de données et les opérations sont bien sur plus complexes qu'au niveau machine, mais c'est le même paradygme qui est suivi. C'etait l'unique paradygme des premiers langages de programmation et le paradygme du langage C.
- **Programmation orientée objet** : Ce style de programmation est une réponse au besoin de concevoir des programmes complexes, dont les données et les traitement sont rassemblés au sein de structures (Les objets) qui peuvent être polymorphes.
- **Programmation fonctionnelle** : L'opération de base des langages fonctionnels n'est pas l'affectation comme pour les langages impératifs, mais l'évaluation de fonctions. Les langages fonctionnels emploient des types et des structures de données de haut niveau permettant réaliser facilement des opérations sur les listes. Un mécanisme puissant des langages fonctionnels est l'usage des fonctions d'ordre supérieur, c'est-à-dire qui peut prendre des fonctions comme argument et/ou retourner une fonction comme résultat.
- **Programmation générique** : Ce style de programmation utilise des classes d'objets et des fonctions parametrées. L'outil le plus utile de ce style de programmation est la notion de template (gabarit) pour gérer le polymorphisme. Elle permet de développer des composants paramétrés par des types de données, l'objectif principal etant la réutilisabilité du développement.
- **Programmation logique** Cette forme de programmaiton définit des applications à l'aide d'un ensemble de faits élémentaires les concernant et de règles de logique leur associant des conséquences plus ou moins directes. Ces faits et ces règles sont exploités par un démonstrateur

de théorème ou moteur d'inférence, en réaction à une question ou requête.
- **Programmation par contraintes**. Ce type de programmation, à rapprocher du précédent, consiste à poser un problème sous forme de relations logiques (contraintes) entre plusieurs variables. Un problème formulé de la sorte comporte donc un certain nombre de ces variables, chacune ayant un domaine de valeurs possibles, et un certain nombre de contraintes. Une contrainte peut porter sur une ou plusieurs variables, et restreint les valeurs que peuvent prendre simultanément ces variables. Trouver une solution à un problème de posé sous cette forme consiste à décider s'il existe ou non une affectation de toutes les variables telle que l'ensemble des contraintes du problème soient satisfaites.
- **Programmation déclarative** : En programmation déclarative, on décrit le *quoi*, c'est-à-dire le problème. Par exemple, les pages HTML sont déclaratives car elles décrivent ce que contient une page (texte, titres, paragraphes, etc.) et non comment les afficher (positionnement, couleurs, polices de caractères, etc.). Alors qu'en programmation impérative (par exemple, avec le C ou Java), on décrit le comment, c'est-à-dire la solution.

Certains langages sont (ou pas) mieux adaptés à des paradygmes. Par exemple, le C++ est à la fois un langage impératif, générique et objet ; le Python est un langage orienté objet proposant certaines fonctionnalité du paradygme fonctionnel ; et le Common Lisp permet à la fois de programme dans les styles imperatif, fonctionnel, et objet et le fait qu'il soit programmable permet également de faire de la programmation logique et de la programmation par contraintes.

Un autre trait de caractère qui peut également permettre de distinguer les langages de programmation les uns des autres, c'est le **typage des données** : un langage est dit typé s'il permet d'attribuer un type (entier, nombre flottant, chaine de caractère, fichier...) aux éléments du code source (variables, fonctions...). Le typage peut être explicite ou implicite selon que le programmeur ou le compilateur les attribue ; fort ou faible, selon la rigidité des règles de passage d'un type à un autre ; statique ou dynamique selon si l'attribution des types se fait à la compilation ou pendant l'éxecution.

Il existe bien sûr d'autres particularités permettant de catégoriser les langages de programmation,

1.3 Compilation d'un programme

Le C est un langage compilé : Le fichier source, écrit dans un fichier texte doit être traduit dans le langage machine. Le C fait partie des langages faisant appel à un **compilateur**.

Un compilateur est un programme qui "traduit" une instance d'un **langage source**, dans un autre langage, appelé le **langage cible**, en préservant la signification du texte source. Ce principe général décrit un grand nombre de programmes différents et ce que l'on entend par "signification du texte source" dépend du rôle du compilateur.

Lorsque l'on parle de compilateur, on suppose aussi en général que le langage source est, pour l'application envisagée, de plus haut niveau que le langage cible, c'est-à-dire qu'il présente un niveau d'abstraction supérieur.

En pratique, un compilateur sert le plus souvent à traduire un code source écrit dans un langage de programmation en un autre langage, habituellement un langage d'assemblage ou un langage machine.

Un compilateur fonctionne par analyse-synthèse, c'est-à-dire qu'au lieu de remplacer chaque construction du langage source par une suite équivalente de constructions du langage cible, il commence par analyser le texte source pour en construire une représentation intermédiaire appelée **code objet** à partir duquel il construit le **code executable** (instance autonome du langage machine) ou un programme équivalent au code source écrit dans le langage cible, si celui-ci n'est pas le langage machine.

Il est donc naturel de séparer – au moins conceptuellement, mais aussi en pratique – le compilateur en une partie avant (ou frontale), parfois appelée "souche", qui lit le texte source et produit la

représentation intermédiaire, et une partie arrière (ou finale), qui parcourt cette représentation pour produire le texte cible. Dans un compilateur idéal, la partie avant est indépendante du langage cible, tandis que la partie arrière est indépendante du langage source. Certains compilateurs effectuent de plus sur la forme intermédiaire des traitements substantiels, que l'on peut regrouper en une partie centrale, indépendante à la fois du langage source et de la machine cible. On peut ainsi écrire des compilateurs pour toute une gamme de langages et d'architectures en partageant la partie centrale, à laquelle on attache une partie avant par langage et une partie arrière par architecture.

Voici les 4 phases de la compilation :

1. **Le traitement par le préprocesseur** : Le fichier source est analysé par le préprocesseur qui effectue les transformations purement textuelles : le remplacement des chaines de caractères, l'inclusion d'autre fichiers sources, la suppression des commentaires.
2. **La compilation** : La compilation proprement dite traduit le code source en assembleur, c'est-à-dire une suite d'instructions qui utilise des **mnémoniques** : des mots courts correspondants un a un à des octets mais qui rendent la lecture possible. Trois phases d'analyse sont effectuées avant la traduction proprement dite : le découpage du programme en lexèmes (analyse lexicale), la vérification de la correction de la syntaxe du programme (analyse synthaxique), l'analyse des structures de données (analyse sémantique),
3. **L'assemblage** : Cette opération traduit le code assembleur en fichier binaire, compréhensibles directement par le processeur. Cette étape est en général faite direcement après la compilation, sauf si on spécifie que l'on veut uniquement le code assembleur. Le fichier obtenu est appellé **fichier objet**. Il contient également les informations nécessaires à l'étape suivante.
4. **L'édition de liens** : Un programme est souvent séparé dans plusieurs fichiers sources pour des raisons de clarté mais aussi parce que l'on fait souvent appel à des bibliothèques (library, en anglais...) de fonctions standards déjà écrites. Une fois le code source assemblé, il faut donc lier entre eux tous les fichier objets créés. L'édition de ces liens produit alors le **fichier exécutable**.

Ces différentes étapes expliquent que les compilateurs fassent toujours l'objet de recherches, particulièrement pour des questions d'optimisation.

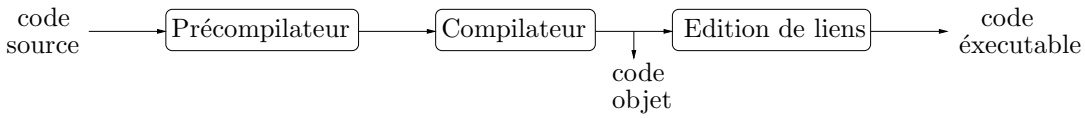


FIG. 1.1 – Schéma du processus de compilation

Voici la marche à suivre pour compiler et executer un programme C monofichier dans un environnement Unix (terminal ou console sous Linux, Cygwin sous Windows) et avec le compilateur gcc (pour GNU Compiler Collection).

On saisit le texte du programme sous un editeur de texte (emacs, gedit...) que l'on enregistre avec l'extension `".c"`. Ex : `programme.c`, `fichier.c`, `machin.c`....

Cette extension est nécessaire car gcc peut compiler des programmes écrits dans plusieurs langages (C, C++, java...). L'extension lui indique donc qu'il est en présence d'un fichier source écrit en C. Au passage, les fichiers traités par le préprocesseur ont l'extension `.i`, les fichiers assembleurs ont l'extension `.s` et les fichier objets l'extension `.o`.

On compile ensuite le programme fichier.c en lançant la commande suivante :

```
gcc [options] programme.c [-lbibliotheque]
```

Un fichier exécutable, appelé par défaut `a.out` est alors généré et l'execution du programme se fera en lancant la commande :

`/a.out`

Ce nom peut être modifié en utilisant l'option `-o`.

Les éventuelles bibliotheques précompilées nécessaires sont déclarées sous forme d'options dans la chaine `[-lbibliotheque]`. Par exemple, pour lier le programme avec la bibliothèque de fonctions mathématiques, on spécifie `-lm`. Le compilateur recherchera alors (généralement dans `/usr/lib/`) le fichier pré-compilé `libm.a` (plus généralement, si on spécifie `-lbiblio`, le compilateur recherchera alors `libbiblio.a`).

On peut également (c'est même préférable...) rajouter des options. Les deux plus usuelles sont :
- `-Wall` (W pour warning et all pour all...) qui génère une liste très complète d'avertissements sur les éventuelles incorrections du programme, qui pourrait provoquer des problèmes lors de l'execution.
- `-o` (pour out) qui spécifie le nom du fichier executable.

D'autres options existent bien sur, comme `-O` qui permet d'optimiser la taille et le temps d'execution du code executable (par défaut, le compilateur optimise son propre temps d'execution) ou `-S` qui produit le fichier assembleur.... Pour la liste complètes des options de gcc, on se réfère au manuel via la commande man gcc.

Ainsi, pour compiler un fichier source enregistré dans sous le nom `max.c`, sera compilé en lançant par exemple la commande :

```
gcc -Wall max.c -o maximum
```

puis exécuté en lançant la commande :

```
maximum
```

Après compilation, un programme C est éxecuté sous le contrôle du système d'exploitation de l'ordinateur. Il communique avec "l'extérieur" par l'intermédiaire d'unités d'entrées-sorties (un clavier, un écran, une imprimante, un disque dur, etc...). On dit qu'un programme lit des données sur une unité d'entrée et écrit des données sur une unité de sortie.

1.4 Résumé du cycle de vie d'un programme

Voici les différentes étapes de la vie d'un programme écrit en langage compilé :

1. **Conception du programme** : Analyser l'application à programmer pour mettre en évidences les types de données à manipuler et les opérations à effectuer, choisir les meilleurs algorithmes pour réalier ces opérations, décider de la manière de présenter les résultats à l'utilisateur du programme, quel langage est le mieux adapté à aux taches qui doivent être réalisées par le programme, etc ...

   C'est une étape cruciale qui peut être faite sans ordinateur sous la main...
2. **Ecriture du programme** : Si la première étape a été faite avec soin, il s'agit uniquement de traduire les résulats de la conception en un programme écrit dans le langage de programmation choisi, ici le C.
3. **Compilation** : C'est l'étape de traduction vers le langage machine. Il vérifie si le programme est lexicalement, synthaxiquement et sémantiquement correct. Si c'est le cas, il produit le code exécutable, sinon, il signale les erreurs en les localisant le plus précisément possible dans le texte du programme et vous renvoie donc à l'étape 2.
4. **Exécution** : Le code executable produit à l'issu de la compilation peut être lancé, c'est-à-dire soumis au système d'exploitation pour être executé. Tant que le texte du programme n'est pas

modifié, il n'est pas nécéssaire de recompiler avant de lancer l'executable. Il se peut que l'execution déclenche des erreurs non détectées par le compilateur.**Notez bien qu'un programme qui compile sans erreurs n'est pas forcément correct** : il se peux qu'il ne satisfasse pas le cahier des charges de l'application à programmer.... on retourne dans ce cas à l'étape 1 ou 2...

Chapitre 2

Les bases de la programmation en C

2.1 Les composants du langage C

Les “mots” du langage C peuvent être classés en 6 groupes élémentaires :
- les mot-clés,
- les constantes,
- les identificateurs,
- les chaines de caractères,
- les opérateurs,
- les signes de ponctuation,

auxquels il convient d'ajouter les commentaires au code source.

Le langage C contient 32 **mots-clés** (selon la norme ANSI C) qui ne peuvent pas être utilisés comme itentificateurs :
- les spécificateurs de stockage :
  `auto` `register` `static` `typedef`
- les spécificateurs de type :
  `char` `double` `enum` `float` `int` `long` `short` `signed` `struct` `union` `unsigned` `void`
- les qualificateurs de type :
  `const` `volatile`
- les instructions de contrôle :
  `break` `case` `continue` `default` `do` `else` `for` `goto` `if` `switch` `while`
- les autres :
  `return` `sizeof`

Le rôle des **identificateurs** est de donner un nom à une entité du programme. Un identificateur peut désigner :
- le nom d'une variable. Une variable “nomme” une ou plusieurs cases mémoires qui contiendront la dernière valeur affectée à la variable.
- le nom d'une fonction,
- le nom d'un type défini par `typedef`, `struct`, `union` ou `enum`.

Quelques règles pour le choix des identifiacteurs :
- il ne doivent pas être choisis parmi les mots clés,
- Un identificateur est une chaine de caractères choisis parmi les chiffres et les lettres NON ACCENTUÉES, en tenant compte de leur casse (Majuscule ou minuscule...). On peut aussi utiliser le tiret bas (_),

– Le premier caractère de la chaine ne doit cependant pas être un chiffre et il vaut mieux que qu'il soit le tiret bas car il est souvent employé pour les variables globales de l'environnement C,

Il se peut que le compilateur tronque les identificateurs au delà d'une certaine longueur (supérieure en général à 31 caractères). Cela laisse une marge assez large pour les choisir. Il est en général conseillé de s'imposer une nomenclature précise.

Un commentaire dans le code source débute par `/*` et se termine par `*/` :

```
/* ceci est un petit commentaire dans un code source */
```

Notez que l'on peut également mettre des commentaires courts en fin de ligne en utilisant `//` mais on préferera quand même l'environnement `/* */`. Les commentaires, bien optionnels, sont fondamentaux. Ils permettent une compréhension plus facile d'un code laissé de coté pendant des semaines et sont utiles lors de la maintenance des programme.

Une **expression** est une suite de composants élémentaires qui est synthaxiquement correcte. Par exemple :

`x=1`

ou

`(i>=1)&&(i<11)&&(i%2!=0)`

Une **instruction** est une expression suivie par un point-virgule. Le point virgule signife que l'expression va être évaluée. Plusieurs instructions peuvent être réunies entre accolades pour former une **instruction composée** ou **bloc**, synthaxiquement équivalente à une instruction. Par exemple :

```
if(x!=0)
  {
   z=y/x;
   t=y%x;
  }
```

Une instruction composée d'un spécificateur de type et d'une liste d'identifiant séparés par une virgule est une **déclaration**. Par exemple :

```
int a, b;
double c;
char message[256];
float d=1.5, x ;
```

La dernière déclaration est couplée avec une affectation (symbôle `=`).

Toute variable utilisée en C doit faire l'objet d'une déclaration avant d'être utilisée.

2.2 Anatomie d'un programme C monofichier

Voici l'ossature d'un programme monofichier écrit en C :

```
[directives au préprocesseur]
[déclaration des variables externes]
[fonctions secondaires]

type main (arguments)
  {
     declaration des variables internes
     instructions
  }
```

Tout programme C doit contenir au moins une fonction : C'est la fonction `main` (fonction principale). L'execution du programme commence par l'appel à cette fonction.

Un programme C manipule des valeurs classées par types (voir paragraphe suivant). Ici, la fonction `main` retourne un objet du type *`type`* via une instruction `return` *`resultat`* `;` où *`resultat`* doit être l'identificateur d'une variable de type *`type`*.

Un programme C est composé d'un ensemble de fonctions, qui peuvent s'appeller entre elles en se transmettant des valeurs.

Comme la fonction `main`, une fonction secondaire est décrite de la manière suivante :

```
type FonctionAuxiliaire (arguments) {
    declaration des variables internes
    instructions
    return resultat ;
}
```

La première ligne donne le **prototype** de la fonction fonction. Il spécifie :

- le nom de la fonction,
- le type des parametres qui doivent lui être fournies. La déclaration des arguments d'une fonctions s'effectue dans une syntaxe voisine des celle des variable : on mets en argument une suite d'objets de type spécifiés séparés par des virgules, ex :
  `FonctionAuxiliaire(int a, float b)`
- le type de la valeur qu'elle retourne.

Les prototype des fonctions secondaires doivent être placés avant la fonction main (on parle de déclaration de fonctions).

Le **corps de la fonction** (entre les accolades) est la suite d'intructions qui doivent être accomplies lorsque la fonction est appellée.

Les corps de fonctions peuvent quant à eux être placés indifféremment avant ou après la fonction principale.

On reviendra en détails sur les fonctions au Chapitre 3.

Les directives au préprocesseur indiquent quelles bibliothèques de fonctions prédéfinies on souhaite pouvoir utiliser dans le programme, par exemples les fonctions permettant de lire et d'écrire des données, ou des fonctions mathématiques usuelles. Lorsqu'on veut faire appel à ces fonctions, on doit le spécifier. C'est le rôle des première lignes du programme. Le plus souvent, la forme des directives au préprocesseur est sous la forme : `#include<chemin/vers/la/bibliotheque.h>`.

Par exemple, pour utiliser les fonctions qui permettent de lire et d'écrire dans les entrées/sorties standard, on utilise la directive `#include<stdio.h>` (pour StanDard In and Out), pour utiliser les fonctions mathématiques standards, on utilise la directive `#include<math.h>`, ou encore, pour utiliser des fonctions traitant de l'allocation mémoire, de conversion des chaines de caractères en type numériques ou de tirages aléatoires, on utilisera la directive `#include<stdlib.h>`.

Les directives au préprocesseur peuvent également définir des macros. Elles seront détaillées au Chapitre 3.

Ci-dessous, voici un premier exemple d'un programme (assez lourd...) qui calcule le maximum de 5 et 9.

```
1       /* Programme calculant le maximum de deux nombres entiers fixés*/

2       #include<stdio.h>

3       int Max(int x, int y)
4         {
5          if(x>y)
6            return x;
7          else
8            return y;
9         }

10      int main(void)
11        {
12           int m;
13           m=Max(5,9);
14           printf("le maximum de 5 et 9 est %d.\n",m);
15           return 0;
16        }
```

Ce programme est construit de la manière suivante :

La ligne 1 contient un commentaire qui indique ce qu'effectue le programme. Cette ligne est optionnelle, mais indispensable...

La Ligne 2 indique que le programme peut faire appel à des fonctions de la bibliothèque standard `stdio.h`, en l'occurrence, le programme contient un appel à la fonction `printf` qui permet d'afficher des données à l'écran.

Les lignes 3 à 9 constituent la définition de la fonction `Max`. La ligne 3 est appelée en-tête de la fonction ; elle indique que la fonction a pour nom `Max`, prend en argument deux entiers (de type `int`) qui sont affectés aux variables `x` et `y` et retourne un nombre entier (type `int`).

Le corps de la fonction `Max` est constitué des lignes 4 à 9. Ces lignes détaillent quoi faire si une autre fonction fait appel à la fonction `Max`. L'instruction `if`, qui commence ligne 5 et finit ligne 9, s"appuie sur la comparaison des valeurs des variables `x` et `y`. Si la valeur de x est supérieure à celle de y, alors (instruction return de la ligne 6) la valeur `x` de est retournée comme résultat à la fonction qui avait fait appel à la fonction `Max`. Sinon (instruction else de la Ligne 7), C'est la valeur de y qui sera retournée. (instruction return de la Ligne 8)

Les lignes 10 à 16 constituent la définition de la fonction`main`. Le corps de la fonction est constitué des Lignes 11 à 16. On notera que la fonction ne pred ici aucun argument (`void` pour vide...).

La ligne 12 contient da définition d'une variable `m` de type `int`. A laquelle on affectera le résultat du calcul du maximum.

L'instruction de la ligne 13 appelle la fonction `Max` pour calculer le maximum des deux nombres qui lui sont transmis en arguments. La valeur retournée par la fonction `Max` sera affectée à la variable `m` (via l'opérateur d'affectation `=`).

L'instruction de la ligne 14 fait appel à la fonction pour afficher à l'écran la chaine de caractères suivante :

`Le maximum de 5 et 9 est` ***xxx***

où ***xxx*** est la valeur de la variable `m`.

L'instruction de la ligne 15 demande que la valeur 0 soit retournée au système d'exploitation pour lui indiquer que le programme n'a rencontré aucune erreur de fonctionnement.

Une fois compilé, l'exécution du programme se déroule comme suit :

1. la fonction `main` est appelée par le système d'exploitation,
2. la fonction `Max` est appellée avec les valeur 5 et 9 (Ligne 13) qui sont affectées respectivement aux variables `x` et `y` de la fonction `Max`.
3. la condition `x>y` est testée (ligne 5) : elle est fausse, l'instruction introduite par le mot clé `else` est donc executée et la valeur 9 est retournée à la fonction `main`.
4. la valeur 9 est affectée à la varible `m` (toujours Ligne 13).
5. la chaine de caractères `Le maximum de 5 et 9 est 9` est affichée (Ligne 14).
6. l'execution de l'instruction `return 0 ;` (Ligne 15) rend la main au système d'exploitation en lui indiquant que le programme s'est bien déroulé.

2.3 Les types de données prédéfinis

Le langage C est un langage typé : toutes les variables et les constantes, ainsi que les valeurs retournées par les fonctions sont d'un type spécifié. Le **type** d'un objet spécifie la façon dont il est représenté en mémoire (quel volume, quel norme de codage...), ainsi que les fonctions qui lui sont applicables.

Dans la mémoire d'un ordinateur, les données sont stockées sous forme d'octets (séries de 8 bits). Chaque octet est caractérisé par son adresse, qui est un entier. Deux octets contigus ont des adresses qui diffèrent d'une unité. Lorsque l'on définit une variable d'un certain type, on affecte à son identificateur une zone de mémoire caractérisée par son **adresse** et dont la longueur est fixée par le type.

Les types de base en C concernent :
– les caractères,
– les nombres entiers,
– les nombres flottants (qui approximent les nombres réels).

Les mots-clés permettant de les désigner sont parmis les suivants :

```
char
int
float   double
short   long   unsigned
```

Il existe également un autre type : `void`, qui représente le vide (on rien). Il n'est pas possible de déclarer une variable de type `void` mais ce type est utile, notamment pour spécifier (de manière propre) qu'une fonction ne prend pas d'argument ou qu'elle ne revoie pas de valeur. Nous verrons cela au Chapitre 3

2.3.1 Les types caractères

Le type `char` (de l'anglais "character") permet de représenter les charactères de la machine utilisée (symboles du clavier...). Ce type est un peu particulier parce que sa taille définit l'unité de calcul pour les quantités de mémoire (et donc pour les tailles des autres types du langage).

Par définition, la taille du type `char`, notée `sizeof(char)`, vaut toujours 1 (on reviendra sur cette fonction au Chapitre 4, section 4.2). Cependant, il faut faire attention : contrairement à ce qu'on pense souvent, un `char` au sens du C ne vaut pas toujours un octet. Il occupera au minimum 8 bits certes, mais il existe des architectures, relativement spécialisées il est vrai, codant les `char` sur 9 bits ou 16

bits, voire plus. Dans une large majorité des cas, les compilateurs utilisent des char de 8 bits, à la fois par simplicité (les machines modernes fonctionnent généralement en 8, 16, 32 ou 64 bits) et pour éviter des problèmes de portabilité de code (beaucoup de codes C existants reposent sur l'hypothèse que les `char` sont codés sur 8 bits, et risque de ne pas marcher sur une autre architecture). Par souci de simplification, on utilisera le terme octet la plupart du temps.

Type de donnée	Signification	Nb Octets	Plage de valeurs acceptées
`char`	Caractère	1	-128 à 127
`unsigned char`	Caractère non signé	1	0 à 255

TAB. 2.1 – Les types caractères

Les caractères sont en fait représentés dans l'ordinateur sous forme d'entiers. Les machines utilisent le jeu de caractères (en France, typiquement la norme ISO-8859), codé sur 8 bits dont les 128 premiers charactères correspondent au code ASCII, les 128 derniers étant utilisés pour coder les caractères spécifiques au caractères propres aux différentes langues (caractères accentués, caractères cyrilliques, caractères de la langue arabe, etc... selon la norme choisie). La correspondance entre caractères imprimables et code ASCII est présentée aux Tableaux 2.7 et 2.8 en fin de chapitre. Les caractères ASCII inférieurs à 33 ne sont pas affichables, il s'agit de caractère spéciaux et de caractères de formatage (caractère non affichables à proprement parler, qui influencent la mise en page).

Comme un objet de type `char` peut être assimilable à un entier, tout caractère peut être utilisé dans une expression qui utilise des objets de type entiers. Par exemple, l'expression `a+1` pour un objet `a` de type `char` a un sens : elle désigne le caractère suivant dans le code ASCII, c'est-à-dire le caractère `b` ou l'entier `98`.

2.3.2 Les types entiers

Le mot-clé désignant une donnée de type entier est `int`. On peut faire préceder le mot-clé `int` par un attribut de précision (`short` ou `long`) et/ou d'un attribut de représentation (`signed` ou `unsigned`).

Type de donnée	Signification	Nb Octets	Plage de valeurs acceptées
`short int`	Entier court	2	-32 768 à 32 767
`unsigned short int`	Entier court non signé	2	0 à 65 535
`int`	Entier	2 4	-32 768 à 32 767 -2 147 483 648 à 2 147 483 647
`unsigned int`	Entier non signé	2 4	0 à 65 535 0 à 4 294 967 295
`long int`	Entier long	4	-2 147 483 648 à 2 147 483 647
`unsigned long int`	Entier long non signé	4	0 à 4 294 967 295

TAB. 2.2 – Les types entiers

Selon la machine sur laquelle ont travaille, la taille de la mémoire allouée à un objet de type `int` peut varier, mais elle est en général de 2 ou 4 octets (selon le processeur) et toujours supérieure à celle alouée pour un `char`.

De manière générale, pour connaitre le nombre d'octets nécessaire pour stocker un objet, on utilise le mot clé `sizeof`, qui a pour syntaxe :

`sizeof(`*`expression`*`)`

où *`expression`* est un type ou un objet. Cette fonction renvoie le nombre d'octets necessaire au stockage de *`expression`* sous la forme d'un entier.

Petite remarque à propos de la représentation binaire des entiers signés : En supposant qu'un objet de type entier soit codé sur 4 octets, c'est-à-dire 32 bits, le bit de poids fort (le premier) correspond au codage du signe (0 si positif, 1 si strictement négatif). Pour les nombres positifs, les 31 bits qui suivent correspondent à l'écriture binaire du nombre (d'où une borne supérieure en $2^{31}-1$), completée à gauche par des zéros. Pour les nombres négatifs, les 31 bits qui suivent correspondent à la valeur absolue du nombre, représentée par la technique du complément à 2 : le nombre est écrit en binaire, complété à gauche par des zéros, puis les bits sont inversés (les 1 deviennent des 0 et les 0 deviennent des 1) et enfin, on rajoute 1 au résultat.

L'attribut `unsigned` signifiant l'absence de signe, un `unsigned int` codé sur 4 octets peut donc représenter un entier entre 0 et $2^{32}-1$.

Petite remarque à propos des booléens :

Le langage C (jusqu'à la norme C99) ne fournit pas de type booléen. La valeur entière 0 prend la valeur de vérité FAUX et toutes les autres valeurs entières prennent la valeur de vérité VRAI : toute expression utilisant des opérateurs booléens (voir Paragraphe 2.5), retourne 0 si l'expression est fausse, et retourne quelque chose de non nul si l'expression est vraie.

La norme C99 a introduit le type `_Bool`, qui peut contenir les valeurs 0 et 1. Elle a aussi ajouté l'en-tête `<stdbool.h>`, qui définit le type `bool` qui est un raccourci pour `_Bool`, et les valeurs `true` et `false`. Ces nouveautés du C99 ne sont pas très utilisées, la mauvaise habitude ayant été prise d'utiliser 0 et différent de zéro pour les booléens en C.

2.3.3 Les types flottants

Les types `float`, `double` et `long double` sont utilisés pour représenter des nombres à virgule flottante, qui correspondent à des approximations de nombres réels à des degrés de précision différents.

Type de donnée	Signification	Nb Octets	Plage de valeurs acceptées
`float`	Flottant	4	$3,4 \cdot 10^{-38}$ à $3,4 \cdot 10^{38}$ env.
`double`	Flottant double	8	$1,7 \cdot 10^{-308}$ à $1,7 \cdot 10^{308}$ env.
`long double`	Flottant double long	10	$3,4 \cdot 10^{-4932}$ à $3,4 \cdot 10^{4932}$ env.

TAB. 2.3 – Les types flottants

Ces types sont en général stockés en mémoire selon la **représentation de la virgule flottante normalisée** : `s1.x---xBˆy---y`. Le symbole `s` représente le signe. La plage de chiffres `x---x` est appellée **mantisse** et la plage de chiffres `y---y` est l'**exposant** ; leurs tailles dépendent de la précision choisie. L'entier `B` représente la base (2 en général).

Un flottant est représenté ainsi : le bit de poids fort donne le signe, l'exposant est ensuite représenté en binaire puis les bits de poids faible représentent la mantisse.

Attention : Les nombres flottants peuvent pas représenter tous les réels... Certaines approximations sont faites, puisque la majorité des nombres réels ont une expansion binaire infinie...

Le codage des variables de type `float` se fait sur 4 octets. Le premier chiffre représente le signe (0 si positif, 1 si négatif). Les chiffres suivants s'obtiennent comme suit :

On écrit dans un premier temps la valeur qui l'on doit représenter en binaire :

$$2^N + a_{N-1}2^{N-1} + \dots a_1 2 + a_0 + a_{-1}2^{-1} + a_{-2}2^{-2} + \dots,$$

Les 8 chiffres suivants le code du signe représentent alors $N+127$ (en binaire, éventuellement complété par des zéros).

Les 23 chiffres restants sont donnés en utilisant la règle suivante :
– $a_{N-1} \dots a_1 a_0 a_{-1} a_{-2} \dots a_{N-23}$ si a_{N-24} est nul
– le code binaire de $\sum_{N-23}^{N-1} a_i 2^i + 1$ complété à gauche par des zéros si a_{N-24} est égal à 1.

Ainsi, lorsqu'on affecte à une variable de type `float` la valeur décimale $0,1$, dont la représentation binaire est donnée par :

$$\rho_2(0,1) = 0.0001100110110011001100110011001100 \cdots = 2^{-4} \cdot 1.10011011001100110011 \dots$$

Les 4 octets représentant cette valeur en mémoire sont :

$$\underbrace{0}_{\text{signe}} \; \underbrace{01111011}_{\text{exposant}} \; \underbrace{10011011001100110011\mathbf{0}1}_{\text{mantisse}}$$

Un arrondi est effectué (dernier chiffre de la mantisse), de sorte que ce n'est pas exactement la constante décimale $0,1$ qui est représentée en mémoire. Ces arrondis peuvent donner lieu à des écarts entre les résultats attendus et les résultats fournis par l'ordinateur lors de calculs avec des flottants.

2.3.4 Les constantes

2.3.4.1 Constantes numériques

Une constante entière peut être représentés dans 3 bases différentes : 10, 8 ou 16. voici les différentes conventions d'écriture :
– Les constantes décimales (en base 10) utilisent représentation usuelle. Ex : `0`, `256`, `-34678956734`.
– Les constantes octales (en base 8) commencent par un zéro. Ex : les représentations octales de 0 et 255 sont respectivement `00` et `0377`.
– Les constantes héxadécimales (en base 16) commencent par un zéro suivi d'un `x` et sont représentées en utilisant les lettres `a` à `f` pour les entiers de 10 à 15. Ex : les représentations octales de 13 et 249 sont respectivement `0xd` et `0xf9`.

Par défaut, une constante entière est stockée avec le format interne le plus court qui permet de la représenter. On peut cependant forcer son format en ajoutant un suffixe `U` pour indiquer que l'on désire un entier non signé et/ou en ajoutant une suffixe `L` pour indiquer qu'elle doit être de type `long`. De même on peut forcer une à stocker une constante entière dans le format flottant en lui ajoutant un point en suffixe.

Les constantes réelles sont représentées par la notation classique *`mantisse`* `e` *`exposant`*. C'est un nombre décimal éventuellement signé qui est par défaut de type `double`. On peut forcer son format en ajoutant un suffixe `F` pour indiquer que l'on désire un format `float` ou en ajoutant une suffixe `L` pour indiquer qu'elle doit être de type `long double`.

2.3.4.2 Constantes caractères et chaines de caractères

Pour désigner un caractère imprimable, il suffit de le mettre entre guillemets simples : `'A'`, `'$'`. Les seuls caractères imprimables que l'on ne peut pas représenter de cette manière sont l'antislash et le guillemet simples que l'on designe en utilisant `\\` et `\'`. Les doubles guillemets sont représentés par `\"`.

Quant aux caractères non imprimables, on utilise la convention : `\code` où `code` est le code octal ou le code héxadécimal précédé d'un `x` du caractère dans la norme ASCII (voir tableau 2.7). Par exemples `\11` indique une tabulation horizontale avec son code octal et `\x1b` indique une caractère *Escape* avec son code héxadécimal. Pour les caractères non imprimables usuels, il y a quand même des raccourcis, donnés dans le Tableau 2.4

Caractère	Correspondance
\0	caractère null
\a	cloche ou bip
\b	retour arrière (backspace)
\f	saut de page
\n	saut de ligne
\r	retour chariot
\t	tabulation horizontale
\v	tabulation verticale
\\	antislash
\'	apostrophe
\"	guillemets

TAB. 2.4 – Caractères spéciaux

Une **chaine de caractères** est une suites de caractères, imprimables ou non, entourés de guillemets (doubles). Ex :

```
"ceci est ce qu'on appelle \"une chaine de caractères\"\n"
```

On reviendra en détails sur les chaînes de caractères au Chapitre 4.

2.4 Expressions

Une **expression** est une suite de symboles formée à partir de constantes littérales, d'indentificateurs de variable et d'opérateurs (les opérateurs usuels seront détaillés au paragraphe suivant).

On parle d'**expression simple** lorqu'elle ne contient pas d'opérateur comme par exemple :

- `x`,
- `12`,

et on parle d'**expression composée** si elle en contient, comme par exemple :

- `-x`,
- `x+12`,
- `(x>4)&&(x<10)`.

Une expression est destinée à être évaluée. Elle a toujours un type et une valeur. Par exemple, si le programme est dans un état où la variable `x` de type `int` est visible et a pour valeur 8, alors :

- l'expression `x` est de type `int` et a pour valeur `8`,
- l'expression `12` est de type `int` et a pour valeur `12`,
- l'expression `-x` est de type `int` et a pour valeur `-8`,
- l'expression `x+12` est de type `int` et a pour valeur `20`,
- l'expression `(x>4)&&(x<10)` est de type `int` et a pour valeur le booléen VRAI (représenté par un entier non nul),

Une expression peut éventuellement avoir aussi une adresse mémoire (c'est le cas pour les indentificateur de variables) et son évaluation peut produire des **effets de bord** : modification de l'état de la mémoire, déclenchement d'entrées-sorties...

2.5 Les opérateurs

2.5.1 Affectation

L'affectation en C est symbolisée par le signe `=`. On l'utilise comme suit :

variable `=` *expression*

Cet opérateur évalue l'expression *expression* et affecte le résultat à la variable *variable*. Cette expression possède une valeur qui est celle du résultat de *expression* : l'expression `a=2` vaut 2.

C'est un opérateur à effet de bord puisqu'il modifie la valeur à l'emplacement en mémoire associé à la variable *variable* : une affectation substitue le résultat de l'expression de droite au contenu de la variable.

Attention : L'affectation effectue une conversion de type implicite : si *variable* et le résultat de *expression* ne sont pas du même type, alors le résultat de *expression* est convertie dans le type de *variable*.

Si la conversion est impossible, alors un message d'erreur est renvoyé.

2.5.2 Opérateur de conversion de type : transtypage

Puisque tous les types sont représentés sous forme de plages de 0 et de 1, on peut "traduire" un objet d'un certain type dans un autre type. On dit que l'on fait une opération de **transtypage** (cast en aglais), ou (par mauvais raccourci) que l'on "caste" un objet d'un type à un autre (francisation littérale du verbe anglais to cast, que l'on peut traduire par dépouiller, mouler ou bien d'autres encore...).

Le C est un langage peu typé, dans le sens où les règles de transtypage sont assez larges (d'autres langages ne le permettent pas ou le permettent, mais sous des conditions très strictes).

Il peut arriver par exemple que l'on veuille travailler sur un type de variable, puis l'utiliser sous un autre type. Imaginons que l'on travaille par exemple sur une variable en virgule flottante (type `float`), il se peut que l'on veuille "supprimer les chiffres après la virgule", c'est-à-dire convertir un `float` en `int`. Cette opération peut être réalisée de deux manières :

- soit par **conversion implicite** : une conversion implicite consiste en une modification du type de donnée effectuée automatiquement par le compilateur. Cela signifie que lorsque l'on va stocker un type de donnée dans une variable déclarée avec un autre type, le compilateur ne retournera pas d'erreur mais effectuera une conversion implicite de la donnée avant de l'affecter à la variable.

```
int x;
x = 8.324;
```

  x contiendra après affectation la valeur 8.
- soit par **conversion explicite** : une conversion explicite (appelée aussi opération de cast) consiste en une modification du type de donnée forcée. Cela signifie que l'on utilise un opérateur dit de cast pour spécifier la conversion. L'opérateur de cast est tout simplement le type de donnée, dans lequel on désire convertir une variable, entre des parenthèses précédant la variable.

```
int x;
x = (int)8.324;
```

  x contiendra après affectation la valeur 8.

  La conversion explicite est à privilégier car elle n'est pas ambigüe.

2.5.3 Opérateurs arithmétiques

Les opérateurs arithmétiques classiques sont :

- l'opérateur unaire de changement de signe : `-`,
- les quatres opérations usuelles : `+`, `-`, `*`, `/`,
- l'opérateur modulo : `%` qui donne le reste de la division entière.

Remarque : Notez que la division `/` est la même pour les flottants et les entiers. Par défaut, si les deux opérandes sont entiers, l'opérateur `/` effectue la division entière (Ex : `7/2=3`). Si au moins un des deux opérandes est un flottant, alors, c'est la division usuelle : (Ex : `7./2=3.5` ou `7/2.=3.5`).

l'opérateur modulo `%` ne s'applique quant à lui qu'à des entiers.

2.5.4 Opérateurs d'affectation composée et opérateurs d'incrémentation

Les opérateurs d'affectation composée usuels sont les suivants :

`+=` `-=` `*=` `/=`

Ce sont des "raccoucis" : L'expression *exp1* `op=` *exp2* est équivalente à *exp1* `=` *exp1* `op` *exp2* . Cependant, l'avantage d'utiliser ces notations et que *exp1* n'est évaluée qu'en seule fois lorsqu'on utilise l'opérateur composé, alors qu'elle est évaluée 2 fois sinon.

Le plus souvent lorsqu'on programme en C, on utilise des compteurs : les opérations du type `i=i+1` ou `j=j-1` apparaissent très souvent dans des boucles. On a donc mis à notre disposition l'**opérateur d'incrémentation** `++` et l'**opérateur de décrémentation** `--`.

Ils s'utilisent en suffixe ou en préfixe : les instructions `i=i+1;`, `i++;` et `++i;` incrémente `i` de 1. La différence entre ces deux notations réside dans la valeur de l'expression : si `i=2`, l'expression `i++` a pour valeur 2, la valeur de `i` avant incrémentation, et `++i` a pour valeur 3, la valeur de `i` après incrémentation.

2.5.5 Opérateurs de comparaison

Pour comparer deux expressions *exp1* et *exp2*, on dispose des opérateurs booléens de comparaison classiques :

- égalité : *exp1*`==`*exp2*,
- Inégalités strictes : *exp1*`<`*exp2* et *exp1*`>`*exp2*,
- Inégalités larges : *exp1*`<=`*exp2* et *exp1*`>=`*exp2*,
- Différence : *exp1*`!=`*exp2*.

Les deux expressions *exp1* et *exp2* sont évaluées puis comparées. Si la relation est fausse, alors cet opérateur renvoie le booléen `false` (équivalent à l'entier `0`) et renvoie le booléen `true` (équivalent à un entier non nul) si la relation est vérifiée.

Attention Ne pas confondre l'opérateur d'affectation `=` et le test d'égalité `==`, surtout car les deux peuvent être valides comme test de conditionnement.

2.5.6 Opérateurs logiques

Les opérateurs logiques sont

- la négation : `!(`*exp*`)`,
- le ET logique : *exp1* `&&` *exp2*,
- le OU logique : *exp1* `||` *exp2*.

Si la relation est fausse, alors ces opérateurs renvoient le booléen `false` (équivalent à l'entier `0`) et renvoie le booléen `true` (équivalent à un entier non nul) si la relation est vérifiée. Voici un exemple d'utilisation :

```
int i,j,k;
if((i>0)&&((j<=2)||(j>10))&&!(k==2)){
   ... }
else{
   ... }
```

Dans cet exemple, les intructions du bloc `if` seront évaluées si et seulement si la valeur de `i` est strictement positive, si la valeur de `j` est inférieure à 2 ou strictement plus grande que 10, et si la valeur de `k` est différente de 2. Si une de ces trois conditions n'est pas vérifiée, alors les instructions contenues dans le block else seront executées.
Remarque : Une expression du type *`exp1 && exp2 && exp3`* peut être ambigüe, notamment si deux des trois expressions peuvent créer des effets de bords et portent sur la même variable (les deux expressions dépend l'une de l'autre...). Dans ce cas, on préfererai utiliser un cascade de conditionnement (en utilisant des `if/else`).

2.5.7 Opérateurs bit à bit

Il existe également des opérateurs logiques permettant de modifier les entiers directement chiffre par chiffre à partir de leur écriture binaire.

Les opérateurs unaires bit à bit :

`&` : ET bit à bit, qui retourne le chiffre 1 si les deux chiffres de même poids sont à 1.
Ex : `9 & 12` (00001001 `&` 00001100) retourne `8` (00001000).

`|` : OU bit-à-bit, qui retourne le chiffre 1 si un des deux chiffres de même poids est à 1.
Ex : `9 | 12` (00001001 `|` 00001100) retourne `13` (00001101).

`^` : OU bit-à-bit exclusif, qui retourne 1 si seulement l'un des deux bits de même poids est à 1 (mais pas les deux).
Ex : `9 ^ 12` (00001001 `^` 00001100) retourne `5` (00000101).

Les opérateurs binaires bits à bit :

`~` : complément à 1 , qui retourne 1 si le bit vaut 0 et vis-versa.
Ex : `~9` (`~`00001001) retourne `-10` (11110110).

`>>` : décalage à droite (suivi d'un nombre entier), qui effectue la division entière par 2 ou une de ses puissances (décalage de l'écriture binaire) : l'écriture est décalée à droite, les places libérés à gauche sont remplies avec des 0.
Ex : `9>>1` (00001001`>>`1) retourne `4` (00000100),
Ex : `9>>2` (00001001`>>`2) retourne `2` (00000010),

`<<` : décalage à gauche) (suivi d'un nombre entier), qui effectue undécalage de l'écriture binaire à gauche, les places libérés à droite sont remplies avec des 0.
Ex : `9<<1` (00010010`<<`1) retourne `18` (00010010),
Ex : `9<<2` (00001001`<<`2) retourne `36` (01001000).

Attention : Ne confondez pas les opérateurs binaires bit à bit `&` et `|` avec les opérateurs de comparaison `&&` et `||`.

2.5.8 Opérateur adresse

La mémoire d'un ordinateur est ordonnée : chaque mot mémoire a une adresse. L'opérateur d'adresse `&` appliqué à une variable *`objet`* selon la syntaxe *`&objet`* renvoie l'adresse-mémoire de cette variable.

Cet opérateur sera étudié plus en détails au Chapitre 5. Il sera aussi utilisé pour la fonction de lecture sur l'entrée standard `scanf`.

2.5.9 Opérateur d'affectation conditionnel ternaire

exp1 `?` *exp2* `:` *exp3*
Cet opérateur renvoie *exp2* si *exp1* est vraie (si le résultat n'est pas nul) et renvoie *exp3* sinon.
Ex : l'instruction `m= a>b? a:b;` affecte à `m` le maximum de `a` et de `b`.

2.6 Les instructions de branchement

2.6.1 Branchement conditionnel : `if... else ...`

Une instruction de branchement conditionnel a généralement la forme suivante :

```
if ( test-1 ){
  instructions-1
  }
else if ( test-2 ){
  instructions-2
  }
else if ( test-3 ){
  instructions-3
  }
   ...
else{
  instructions-N
  }
```

Les accolades ne sont pas nécessaires si le bloc d'instruction *instructions-i* ne contient qu'une instruction.

Si le *test-1* est positif, alors seul le bloc *instructions-1* est effectué, sinon, on effectue le *test-2*, s'il est positif, alors seul le bloc *instructions-2* est effectué, etc...

D'autre part, les blocs `else if(){}` et le test *test-N* sont optionnels et peuvent être supprimés, le bloc `else` faisant alors office de "cas par défaut" donnant une syntaxe plus simple du type :

```
if( test ){
  instructionsIF
  }
else{
  instructionsELSE
  }
```

Le bloc `else` est optionnel.

2.6.2 Branchement multiple : `switch`

Le bloc `switch` est une instruction de choix multiple qui permet de choisir un bloc d'instructions à réaliser parmi un ensemble d'instructions possibles, en fonction de la valeur d'une expression.

```
switch( exp ){
  case val-1 :
    instructions-1
    break ;
  case val-2 :
```

```
    instructions-2
    break ;
     ...
  case val-N :
    instructions-N
    break ;
  default :
    instructions
    break ;
  }
```

Si l'évaluation de l'expression *exp* donne une valeur *v* parmi *val-1*, *val-2*, ... *val-N* alors les instructions *instructionsv* relatives à cette valeurs sont effectuées. Le cas `default :` permet d'englober toutes les cas où *exp* donne une valeur qui n'est pas parmi *val-1*, *val-2*, ... *val-N*.

L'instruction `break ;` fait ensuite sortir le programme du block switch : même si la valeur de l'expression *exp* est modifiée par les instructions *instructions-v*, les instructions relatives à cette nouvelle valeur ne seront pas effectuées. Le bloc introduit par la valeur `default` n'est pas non plus executé.

Des instructions peuvent êtres relatives à plusieurs valeurs de *exp*. Par exemple, si le bloc d'instructions *instructions* est à executer sur la valeur vaut *val-1* et *val-2*, on peut replacer :

```
  case val-1 :
    instructions
    break ;
  case val-2 :
    instructions
    break ;
```

par :

```
  case val-1 :
  case val-2 :
    instructions
    break ;
```

2.7 Les boucles

Une boucle, d'un manière générale, permet d'effectuer une suite d'instructions tant qu'une certaine condition est vérifiée, par exemple, rajouter 2 à un entier initialisé à 1, tant que le résultat ne dépasse pas 100. A son execution, le programme repassera plusieurs fois sur les mêmes instructions (d'où le nom de boucle), autant de fois que nécessaire : jusqu'à ce que la condition ne soit plus vérifiée. Trois structures sont à notre disposition pour executer une boucle :

```
  while    do ... while    for
```

2.7.1 La boucle `while`

L'instruction `while` à la forme suivante.

```
while( exp ){
  instructions
  }
```

L'expression *exp* est le **test de continuation** (qui renvoie un booléen) et le bloc d'instructions *instructions* constitue le **corps de la boucle**.

Tant que le test de continuation est vrai, le corps de la boucle est effectué. Si le test de continuation est faux au départ, le corps de la boucle n'est pas effectué.

Attention : si le corps de la boucle ne modifie pas la valeur de l'expression *exp*, il se peut que le programme effectue indéfiniment la boucle. Il faut s'assurer que le test de continuation va finir par devenir faux.

Ci-dessous, un exemple de boucle `while` qui affiche à l'écran les entiers positifs dont les carrés sont strictement inférieurs à 100.

```
int i=0;
int n=0;
while(n<100){
  printf("%d\n",i);
  i++;
  n=i*i;
}
```

2.7.2 La boucle `do ... while`

L'instruction `do ... while` prend la forme suivante :

```
do{
  instructions
} while( exp ) ;
```

Comme pour la boucle `while`, l'expression *exp* est le test de continuation (qui renvoie un booléen) et le bloc d'instructions *instructions* constitue le corps de la boucle.

Important :A la différence de la boucle `while`, l'evaluation du test de continuité se fait APRÈS l'execution des instructions.

Par exemple, si l'on veut demander à l'utilisateur d'un programme de rentrer un nombre entre 1 et 10, le bloc d'instructions :

```
int a;
do{
  printf("/n Entrez un nombre compris entre 1 et 10: ");
  scanf("%d",a);
} while((a<0)||(a>10))
```

permet de s'assurer que le nombre rentré par l'utilisateur est effectivement bien compris entre 1 et 10 avant de poursuivre son execution.

Si on reprend l'exemple précédent d'une boucle qui affiche à l'écran les entiers positifs dont les carrés sont strictement inférieurs à 100 en utilisant `do ... while`, on obtient fois-ci :

```
int i=0;
int n;
do{printf("%d\n",i);
   n=i*i;
   i++;
} while(n<100);
```

Contrairement à la même boucle en utilisant `while`, l'incrémentation de `i` se fait après avoir calculé son carré.

2.7.3 La boucle `for`

L'instruction `for` prend la forme suivante :

```
for( exp1 ; exp2 ; exp3 ){
  instructions
}
```

Une formulation équivalente plus intuitive (mais ne répondant pas au standard ANSI-C...) est la suivante :

```
exp1 ;
while(exp2 ){
  exp3 ;
  instructions
  }
```

L'instruction `for` intègre donc :
- l'initialisation (*exp1*),
- la condition de continuation (*exp2*),
- l'évolution (*exp3*) de la variable dont dépend la condition de continuation.

Par exemple, pour calculer et affiche la somme des entiers de 1 à 10, on utlise le bloc d'instructions suivant :

```
int somme,n;
somme=0;
for(n=1;n<=10;n=n+1){
  somme=somme+n;
  }
printf("Somme des entiers de 1 à 10: %d",somme);
```

On peut aussi utiliser les opérateurs d'incrémentation et affectation composée et écrire :

```
int somme,n;
somme=0;
for(n=1;n<=10;n++){
  somme+=n;
  }
printf("Somme des entiers de 1 à 10: %d",somme);
```

2.7.4 Choisir entre `while`, `do ... while` et `for`

Ces trois instructions sont équivalentes. Voici quelques considérations pour vous aider à choisir laquelle utiliser selon la nature du bloc que vous souhaitez programmer :
- Lorsque le nombre d'itérations dépend d'un paramètre dont les valeurs initiale et finale et l'incrémentation sont connus avant l'execution de la boucle, on utilise plutôt une boucle `for`.

- Dans le cas oú le test de sortie, et donc le nombre d'itérations, dépendent d'un calcul fait dans le corps de la boucle ou que celui-ci ne peut s'apparenter à une incrémentation simple, on utilisera un boucle `while` ou `do ... while` plutot qu'une boucle `for`.
- Pour choisir entre `while` ou `do ... while`, tout dépend du programme : garder en tête que `do ... while` effectue une fois les instructions avant de faire le test (on rentre forcément pour au moins une itération dans la boucle) alors que `while` effectue en premier lieu le test (les instructions peuvent ne jamais être effectuées).

2.7.5 Instructions de branchement non conditionnel dans les boucles

L'instruction `break;` peut être employé à l'interieur d'une boucle (`while`, `do ... while` ou `for`) pour interrompre son déroulement. Il interompt l'execution des instructions du bloc et termine la boucle la plus interne (comme si le test de continuation n'était plus vérifié). Par exemple le bloc :

```
int i;
for(i=1;i<5;i++){
  printf("i=%d, ",i);
  if(i==3)
    break;
  }
printf("\n valeur de i en sortie de boucle: %d",i);
```

va imprimer à l'écran la chose suivante :

```
i=1, i=2, i=3
valeur de i en sortie de boucle: 3
```

L'instruction `continue;` quant à elle permet d'interompre l'execution des instructions du bloc et de retourner au test de continuation. Par exemple, le bloc d'instruction :

```
int i;
for(i=1;i<5;i++){
  if(i==3)
    continue;
  printf("i=%d, ",i);
  }
printf("\n valeur de i en sortie de boucle: %d",i);
```

va imprimer à l'écran la chose suivante :

```
i=1, i=2, i=4, i=5
valeur de i en sortie de boucle: 5
```

2.8 Fonctions d'entrées-sorties

Chaque unité d'entrée ou de sortie constitue un *flux de données*. Les fonctions de lecture et d'écriture sur l'entrée et les sorties standards agissent exactement comme les fonctions de lecture et d'écriture dans des fichiers (que l'on détaillera au Chapitre **??**). L'en-tête `<stdio.h>` (abréviation de STandarD In - Out) fournit trois flux que l'on peut utiliser directement :

- `stdin`, l'entrée standard,
- `stdout`, la sortie standard,

– `stderr`, la sortie standard des erreurs.

Souvent, l'entrée standard envoie au programme les données issues du clavier, et les sorties standard envoient les données que le programme génère à l'écran. Mais d'où viennent et où vont ces données dépend étroitement du contexte et de l'implantation.

2.8.1 Les fonctions `putchar` et `getchar`

Les opérations d'entrées-sorties les plus simples sont celles qui lisent et écrivent un caractère. Il s'agit de fonctions d'entrées-sorties non formatées. La lecture d'un caractère sur le flux d'entrée standard est réalisée par la fonction `getchar`. Cette fonction n'a pas d'argument et retourne un `int` qui correspond au code du caractère lu : pour mettre le caractère lu dans la variable `caractere` (de type `int`), on utilise l'instruction suivante :

```
caractere=getchar();
```

Puisque le flux d'entrée est toujours considéré comme un fichier, cette fonction peut éventuellement détecter la fin du fichier et retourner l'entier `EOF` (End Of File) qui, par convention, termine toujours un fichier et dont la valeur est définie dans la bibliothèque `stdio.h` (en général, cet entier vaut -1). L'entier `EOF` est aussi renvoyé en cas d'erreur de lecture.

L'écriture d'un caractère sur l'unité de sortie standard est réalisée par la fonction `putchar` qui prend en argument une variable de type `int`. Pour afficher le caractère dont la conversion en `int` est `caractere`, on utilise l'instruction suivante :

```
putchar(caractere);
```

La valeur de retour est l'entier `caractere` ou `EOF` en cas d'erreur.

Un petit exemple tout simple de bloc qui lit un caractère unique sur le flux d'entrée standard et l'écrit sur le flux de sortie standard :

```
{
  int c;
  c=getchar();
  if(c!=EOF)
   putchar(c);
}
```

Un autre exemple : le bloc suivant lit l'entrée standard et la recopie, caractère par caractère dans la sortie standard jusqu'au premier retour-chariot (si l'entrée standard est le clavier, le retour chariot correspond au caractère `'\n'`) :

```
{
  char c;
  while((c=getchar())!='\n')
    putchar(c);
}
```

Petite Remarque : A propos du test `(c=getchar())!='\n'` : d'une part, l'affectation `c=getchar()` convertit implicitement l'entier renvoyé par `getchar` en `char`. D'autre part, la valeur de l'expression `(c=getchar())` est l'entier renvoyé par `getchar`, c'est à dire l'entier codant le caractère lu dans l'entrée standard. Le test `(c=getchar())!='\n'` compare deux objets de type `int` car la valeur de l'expression `'\n'` est l'entier correspondant au caractère de retour chariot.

2.8.2 Ecriture avec format : `printf`

La fonction `printf`, contenue dans la bibliothèque `stdio.h` écrit des évaluations d'expressions sur l'unité de sortie standard, selon un format spécifié. Son appel a la forme suivante :

`printf("`*`chaine-de-controle-du-format`*`", exp1,exp2,...,expN)`

Le parametre *chaine-de-controle-du-format* contient les caractères à afficher et les spécifications de format d'écriture correspondants aux paramètres *exp1*, *exp2*, ..., *expN*. La chaine de controle du format est indispensable car la fonction `printf` a un nombre variable d'arguments.

Ces spécifications sont introduites par le signe `%` (pour afficher simplement le signe pourcentage à l'écran, on utilisera `%%`). Les formats sont donnés dans le Tableau 2.5.

Format	Type de donnée	Ecriture
`%d`	`int`	décimale signée
`%ld`	`long int`	décimale signée
`%u`	`unsigned int`	décimale non signée
`%o`		octale non signée
`%x`		héxadécimale non signée
`%lu`	`unsigned long int`	décimale non signée
`%lo`		octale non signée
`%lx`		héxadécimale non signée
`%f`	`double`	décimale, virgule fixe
`%e`		décimale, notation exponentielle
`%g`		décimale, représentation la plus compacte parmi `%f` et `%e`
`%lf`	`long double`	décimale, virgule fixe
`%le`		décimale, notation exponentielle
`%lg`		décimale, représentation la plus compacte parmi `%f` et `%e`
`%c`	`char`	caractère
`%s`	`char*`	chaîne de caractères
`%p`	pointeur	héxadécimale non signée (équivalent à `%lu`)

TAB. 2.5 – Tableau des formats pour la fonction `printf`

Les chaines du type *%zz* apparaissant dans *chaine-de-controle-du-format* seront remplacées à l'impression par les valeurs des expressions *exp1*, *exp2*, ..., *expN* (dans l'odre d'apparition). Si la valeur d'une expression n'est pas du type indiqué par le format, celle-ci sera convertie.

Attention : Vérifier bien que vous avez autant d'expressions en paramètre que de signes `%` dans votre chaine de contrôle....

On peut eventuellement préciser certains paramètres du format d'impression en insérant une précision entre le symbole `%` et le ou les caractères précisant le format :

- Pour les entiers, on peut spécifier la largeur du champs minimal d'impression.
  Ex : l'instruction `%10b` indique qu'au moins 10 caractères seront réservés à l'affichage de la valeur d'une variable de type `int`. Par défaut, la donnée est cadrée à droite du champs de 10 caractères, pour qu'elle soit cadrées à gauche, on utilise `%-10d`.
- Pour les flottants, on peut raffiner l'utilisation de `%f` et `%lf` en choisissant la précision (nombre de chiffres après la virgule).
  Ex : si on veut afficher uniquement les 4 premières décimale de la valeur d'un variable de type `double`, on utilise le format `%.4f`. SI la précision n'est pas spécifiée, elle est de 6 chiffres après

la virgule.
De même, le format `%6.4f` indique que l'on réserve 12 caractères à l'écriture de la valeur d'une variable de type `double`, avec une précision de 4 chiffres après la virgule.

– Pour les chaînes de caractères, on peut également spécifier la plage réservée à l'écriture des caractères d'une chaine ainsi que le nombre de caractères qui seront imprimé.
Ex : en utilisant le format `%30.4s` On réserve un champs de 30 caractères pour écrire les 4 premiers caractères d'une chaine de caractères (s'afficheront alors les 4 premiers "lettres" composant la chaîne, suivis de 26 blancs).

Voici un petit exemple de programme regroupant plusieurs impressions à l'écran en utilisant différentes spécifications de format :

```
#include<stdio.h>

int main(){
  int n=45;
  double x=1e-8+100;
  char lettre='A';
  char * chaine="est le double de";

  printf("\n1. petite chaine toute simple sans argument");
  printf("\n2. Le code ASCII du caractère %c est %d ",lettre,lettre);
  printf("\n3. L'entier %d a pour code octal %o et pour code héxadécimal %x",n,n,n);
  printf("\n4. %s %d %s %d","Le nombre",n*2,chaine,n);
  printf("\n5. %.3f est le nombre flottant %.10e précisé au millième",x,x);
  printf("\n6. Utiliser le format %%f n'est pas suffisant pour afficher x:\n son utili
sation donne %f",x);
  printf("\n7. |%-6s||%6s|\n", "un","deux");

  return 0;
}
```

L'éxecution de ce programme affichera à l'écran les lignes suivantes :

```
1. petite chaine toute simple sans argument
2. Le code ASCII du caractère A est 65
3. L'entier 45 a pour code octal 55 et pour code héxadécimal 2d
4. Le nombre 90 est le double de 45
5. 100.000 est le nombre flottant 1.0000000001e+02 précisé au millième
6. Utiliser le format %f n'est pas suffisant pour afficher x:
son utilisation donne 100.000000
7. |un    ||  deux|
```

2.8.3 Lecture avec format : `scanf`

La fonction `scanf`, également définie dans la bibliothèque `stdio.h` permet de lire des valeurs sur l'unité d'entrée standard, selon un format spécifié en argument et les inscrire dans des cases mémoires dont les adresses sont fournies en arguments. Son appel a la forme suivante :

`scanf("`*chaine-de-controle-du-format*`", `*arg-1,arg-2,...,arg-N*`)`

En fait la plupart du temps, les arguments *arg-i* des adresses mémoires où sont stockées des variables et sont donc de la forme `&var-i` où *var-i* est l'identificateur d'une variable et `&` est l'opérateur d'adressage. Ainsi, le plus souvent, on fait appel à la fonction `scanf` de la manière suivante :

`scanf("chaine-de-controle-du-format", &var-1,&var-2,...,&var-N)`

Comme pour `printf`, la *chaine-de-controle-du-format* contient les spécifications de format des caractères à récupérer à partir du flux d'entrée et pour leur mise en mémoire aux adresses `&var-1`, `&var-2`,..., `&var-N`. Ces spécifications sont introduites par le signe `%`. Les formats sont donnés dans le Tableau 2.6.

Format	**Type de donnée**	**Représentation de la donnée saisie**
`%d`		décimale signée
`%o`	`int`	octale
`%x`		héxadécimale
`%hd`		décimale signée
`%ho`	`short int`	octale
`%hx`		héxadécimale
`%hd`		décimale signée
`%ho`	`long int`	octale
`%hx`		héxadécimale
`%u`	`unsigned int`	décimale non signée
`%hu`	`unsigned short int`	décimale non signée
`%lu`	`unsigned long int`	décimale non signée
`%f`		décimale, virgule fixe
`%e`	`float`	décimale, notation exponentielle
`%g`		décimale, représentation la plus compacte parmi `%f` et `%e`
`%lf`		décimale, virgule fixe
`%le`	`double`	décimale, notation exponentielle
`%lg`		décimale, représentation la plus compacte parmi `%f` et `%e`
`%Lf`		décimale, virgule fixe
`%Le`	`long double`	décimale, notation exponentielle
`%Lg`		décimale, représentation la plus compacte parmi `%f` et `%e`
`%c`	`char`	caractère
`%s`	`char*`	chaîne de caractères

TAB. 2.6 – Tableau des formats pour la fonction `scanf`

Lorsqu'une fonction fait appel à `scanf`, elle se met en attente d'une saisie sur le flux d'entrée standard (le clavier, a priori). Tant que la lecture de la *chaine-de-controle-du-format* n'est pas terminée, ses caractères sont lus les uns après les autres et comparés à ceux du flux d'entrée et les actions suivantes sont réalisées :

- Si le caractère courant de la chaine de contrôle n'est ni un `%`, ni un caractère d'espacement, alors le caractère courant du flux d'entrée doit être lui être identique. Si ce n'est pas le cas, la lecture est interrompue. Les valeurs saisies dans la suite du flux d'entrée ne seront pas mises en mémoire aux adresses des variables passées aux arguments suivants.
- Si le caractère courrant de la chaine de contrôle est un caractère d'espacement (espace, tabulation), alors tous les caractères d'espacement et les retour chariot du flux d'entrée sont lus. Tous les caractères d'espacement de la chaine de contrôle du format sont également lus.
- Si le caractère courant de la chaine de contrôle est un `%`, alors la spécification que ce symbole introduit est lue puis la suite des caractères du flux d'entrée est lue jusqu'au prochain caractère

d'espacement. Si cette chaine de caractères est conforme à la spécification, alors elle est mise en mémoire au bon format à l'adresse précisée en argument. sinon, la lecture est interrompue.

La saisie se termine par un retour chariot.

Attention : Lorsque la lecture du flux d'entrée est interrompue de manière "inopinée", les éléments qui n'ont pas encore étés lus sont toujours en file d'attente dans le flux d'entrée. Cela peut donner des choses étranges si le format n'est pas respecté par l'utilisateur, surtout si le programme comporte plusieurs appels à `scanf`. D'autre part, la fonction `scanf` gère mal les caractères d'epacement et les occurences de retour-chariot. Elle est à éviter si on doit lire des caractères et des chaines de caractères.

Ci-dessous un petit exemple de programme et des problèmes que la saisie peut engendrer.

```
#include<stdio.h>

int main(){
  int n=0;
  int m=0;
  char caractere='A';

  printf("\nEntrez 2 nombres au format \"** et **\":");
  scanf("%d et %d",&n,&m);
  printf("\nEntrez un caractère:");
  scanf("%c",&caractere);
  printf("\nNombres choisis: %d et %d. Caractère choisi: %c.\n",n,m,caractere);

  return 0;
}
```

Voici plusieurs executions de ce programme (les caractères saisis par l'utilisateur sont en gras, le symbôle ⌟ représentera la saisie d'une touche ENTREE). L'ordre d'affichage est indiqué :

Exécution avec saisie correcte :

1. `Entrez 2 nombres au format "** et **": ` **12 et 13** ⌟
2. `Entrez un caractère: ` **B**⌟
3. `Nombres choisis: 12 et 13. Caractère choisi: B.`

Execution avec saisie correcte :

1. `Entrez 2 nombres au format "** et **": ` **12** ⌟ **et** ⌟ **13** ⌟
2. `Entrez un caractère:` ⌟ **B**⌟
3. `Nombres choisis: 12 et 13. Caractère choisi: B.`

Exécution avec saisie incorrecte :

1. `Entrez 2 nombres au format "** et **": ` **12 te 13** ⌟
2. `Entrez un caractère:`
   `Nombres choisis: 12 et 0. Caractère choisi: t.`

Ici, la lecture de **12 te 13** ⌟ s'est arrétée au `t`. Dans le flux d'entré, il reste donc **te 13** ⌟ à lire. Le 2^{e} appel à `scanf` met donc en mémoire le caractère `t` à l'adresse de `lettre`. cela peut également arriver si on saisit par exemple une lettre au lieu d'un chiffre :

1. `Entrez 2 nombres au format "** et **": ` **12 et B** ⌟
2. `Entrez un caractère:`
   `Nombres choisis: 12 et 0. Caractère choisi: B.`

ou encore si on sait trop de parametres :

1. `Entrez 2 nombres au format "** et **":` **12 et 13 B** ␣
2. `Entrez un caractère:`
   `Nombres choisis: 12 et 13. Caractère choisi:  .`

Il se peut également que sur un coup de bol, la saisie soit correcte, mais il ne faut pas y compter... Exemple :

1. `Entrez 2 nombres au format "** et **":` **12 et 13B** ␣
2. `Entrez un caractère:`
   `Nombres choisis: 12 et 13. Caractère choisi: B.`

	déc.	oct.	Hex.	Bin.	Signification
NUL	0	0	0	0	Null (nul)
SOH	1	1	1	1	Start of Header (début d'en-tête)
STX	2	2	2	10	Start of Text (début du texte)
ETX	3	3	3	11	End of Text (fin du texte)
EOT	4	4	4	100	End of Transmission (fin de transmission)
ENQ	5	5	5	101	Enquiry (demande)
ACK	6	6	6	110	Acknowledge (accusé de réception)
BEL	7	7	7	111	Bell (Cloche)
BS	8	10	8	1000	Backspace (espacement arrière)
HT	9	11	9	1001	Horizontal Tab (tabulation horizontale)
LF	10	12	0A	1010	Line Feed (saut de ligne)ut de ligne)
VT	11	13	0B	1011	Vertical Tab (tabulation verticale)
FF	12	14	0C	1100	Form Feed (saut de page)
CR	13	15	0D	1101	Carriage Return (Retour chariot)
SO	14	16	0E	1110	Shift Out (fin d'extension)
SI	15	17	0F	1111	Shift In (démarrage d'extension)
DLE	16	20	10	10000	Data Link Escape
DC1	17	21	11	10001	Device Control 1 to 4 (contrôle des périphériques)
DC2	18	22	12	10010	
DC3	19	23	13	10011	
DC4	20	24	14	10100	
NAK	21	25	15	10101	Negative Acknowledge (Accusé de réception négatif)
SYN	22	26	16	10110	Synchronous Idle
ETB	23	27	17	10111	End of Transmission Block (fin du bloc de transmission)
CAN	24	30	18	11000	Cancel (annulation)
EM	25	31	19	11001	End of Medium (fin de support)
SUB	26	32	1A	11010	Substitute (substitution)
ESC	27	33	1B	11011	Escape (échappement)
FS	28	34	1C	11100	File Separator (séparateur de fichier)
GS	29	35	1D	11101	Group Separator (séparateur de groupe)
RS	30	36	1E	11110	Record Separator (séparateur d'enregistrement)
US	31	37	1F	11111	Unit Separator (séparateur d'unité)
SP	32	40	20	100000	Space(caractère Espace) en anglais

TAB. 2.7 – Correspondance entre caractères non imprimables et code ASCII

	déc.	oct.	Hex.	Bin.		déc.	oct.	Hex.	Bin.
!	33	41	21	100001	Q	81	121	51	1010001
”	34	42	22	100010	R	82	122	52	1010010
#	35	43	23	100011	S	83	123	53	1010011
$	36	44	24	100100	T	84	124	54	1010100
%	37	45	25	100101	U	85	125	55	1010101
&	38	46	26	100110	V	86	126	56	1010110
’	39	47	27	100111	W	87	127	57	1010111
(	40	50	28	101000	X	88	130	58	1011000
)	41	51	29	101001	Y	89	131	59	1011001
*	42	52	2A	101010	Z	90	132	5A	1011010
+	43	53	2B	101011	[	91	133	5B	1011011
,	44	54	2C	101100	\	92	134	5C	1011100
-	45	55	2D	101101	]	93	135	5D	1011101
.	46	56	2E	101110	^	94	136	5E	1011110
/	47	57	2F	101111	_	95	13	7 5F	1011111
0	48	60	30	110000	‘	96	140	60	1100000
1	49	61	31	110001	a	97	141	61	1100001
2	50	62	32	110010	b	98	142	62	1100010
3	51	63	33	110011	c	99	143	63	1100011
4	52	64	34	110100	d	100	144	64	1100100
5	53	65	35	110101	e	101	145	65	1100101
6	54	66	36	110110	f	102	146	66	1100110
7	55	67	37	110111	g	103	147	67	1100111
8	56	70	38	111000	h	104	150	68	1101000
9	57	71	39	111001	i	105	151	69	1101001
:	58	72	3A	111010	j	106	152	6A	1101010
;	59	73	3B	111011	k	107	153	6B	1101011
<	60	74	3C	111100	l	108	154	6C	1101100
=	61	75	3D	111101	m	109	155	6D	1101101
>	62	76	3E	111110	n	110	156	6E	1101110
?	63	77	3F	111111	o	111	157	6F	1101111
@	64	100	40	1000000	p	112	160	70	1110000
A	65	101	41	1000001	q	113	161	71	1110001
B	66	102	42	1000010	r	114	162	72	1110010
C	67	103	43	1000011	s	115	163	73	1110011
D	68	104	44	1000100	t	116	164	74	1110100
E	69	105	45	1000101	u	117	165	75	1110101
F	70	106	46	1000110	v	118	166	76	1110110
G	71	107	47	1000111	w	119	167	77	1110111
H	72	110	48	1001000	x	120	170	78	1111000
I	73	111	49	1001001	y	121	171	79	1111001
J	74	112	4A	1001010	z	122	172	7A	1111010
K	75	113	4B	1001011	{	123	173	7B	1111011
L	76	114	4C	1001100	—	124	174	7C	1111100
M	77	115	4D	1001101	}	125	175	7D	1111101
N	78	116	4E	1001110	~	126	176	7E	1111110
O	79	117	4F	1001111	DEL	127	177	7F	1111111
P	80	120	50	1010000					

Tab. 2.8 – Correspondance entre caractères imprimables et code ASCII

Chapitre 3

Fonctions et programmes

Tout langage de programmation offre un moyen de découper un programme en unitées indépendantes qui peuvent partager des données communes et s'appeller les unes les autres. En langage C, ces unitées sont appellées **fonctions** et les données partagées sont appellées **variables globales**.

Le langage C, et contrairement à d'autre langages comme le Pascal, les définitions de fonctions ne peuvent pas être imbriquées : une déclaration de fonction ne peut pas contenir d'autre déclarations de fonctions. Elle peut cependant contenir des déclarations d'objets qui sont appelées **variables locales**. Nous reviendrons sur la visibilité des variables au paragraphe 3.3.

Par convention, l'execution d'un programme commence par l'appel de la fonction `main` qui doit exister OBLIGATOIREMENT. C'est cette fonction qui fera appel à d'eventuelles autres fonctions auxiliaires. Chaque fonction auxiliaire peut également faire appel à d'autres fonctions, ou s'appeller elle-même (dans ce dernier cas, on dit que la fonction est **récursive**).

3.1 Notion de fonction

3.1.1 Définition d'une fonction

En mathématiques, une fonction f de n variables associe à chaque n-uplet $(x_1, x_2, \ldots, x_n)$ appartenant une partie E d'un ensemble produit $E_1 \times E_2 \times \cdots \times E_n$ un et un seul élément d'un autre ensemble F, appellé ensemble d'arrivée. Cet élément est noté en général $f(x_1, x_2, \ldots, x_n)$. L'ensemble E est appellé *domaine de définition* et F est appellé l'*ensemble d'arrivée*. La définition de la fonction f se résume en langage mathématiques comme suit :

$$\begin{array}{rccl} f: & E & \rightarrow & F \\ & x & \mapsto & f(x_1, x_2, \ldots, x_n) \end{array}$$

Dans un langage de programmation comme le C, le principe est le même. Pour définir une fonction, on a besoin de :

- son nom,
- son domaine de définition (un ou plusieurs types de données),
- son ensemble d'arrivée (un type de données),
- l'algorithme qui permet de calculer l'élément de l'ensemble d'arrivé associé à un élément du domaine de définition.

Voici la structure d'une définition de fonction en langage C :

```
type Nom_fonction (type-1 arg-1, type-2 arg2, ... type-N argN) {
  [déclaration des variables locales]
  liste d'instruction
  }
```

La première ligne : *type Nom_fonction (type-1 arg-1, type-2 arg2, ... type-N argN)* est l'**en-tête de la fonction**. Elle comprend :

- le type de la **valeur de retour** : (*type*) , qui précise l'ensemble d'arrivée de la fonction. On l'appelle le **type de la fonction**.
  Si la fonction ne doit retourner aucune valeur, on remplace le type par le mot-clé `void`. On parle dans ce cas de **procédure** plutot que de fonction.
- le **nom de la fonction** : *Nom_fonction*,
- les types des variables, qui précise le domaine de définition de la fonction. Les variables sont appellées **arguments formels de la fonction**. Leurs identificateurs n'ont d'importante qu'a l'interieur même de la fonction.
  Si la fonction n'a aucun argument, on remplace la liste d'arguments par le mot-clé `void`.
  D'autre part, si deux arguments sont du même type, on ne peut PAS grouper leurs déclaration. Chaque argument doit avoir sa propre déclaration avec son type.

Lorsqu'on rencontre l'expression *Nom_fonction*`(exp-1, exp-2, ..., exp-N)` où `exp-1`, `exp-2`, ..., `exp-N` sont des expressions de type respectifs *type-1*, *type-2*, ... , *type-N* dans le corps d'une autre fonction (dite **fonction appelante**), on dit que l'on fait **appel à la fonction** *Nom_fonction*. les arguments `exp-1`, `exp-2`, ..., `exp-N` sont appellés **arguments effectifs** ou **paramètres effectifs**.

Le bloc d'instruction qui suit l'en-tête de la fonction est appellé **corps de la fonction**. C'est là qu'est explicité l'algorithme qui permet de calculer la valeur de retour de la fonction, en fonction des arguments. Le corps de la fonction débute éventuellement par la déclaration de variables dites locales pour stocker la valeur à retourner et si on a besoin de variables annexes autres que les arguments. Il doit se terminer par une **instruction de retour à la fonction appelante** qui utilise le mot-clé `return`. La syntaxe de cette instruction est :

`return (`*exp*`) ;`

La valeur de r*exp* est la valeur que retournera la fonction *Nom_fonction* à la fonction qui a fait appel à elle. Son type doit être le type de la fonction. Si le type de la fonciton est `void`, on utilise simplement l'instruction `return;`.

Plusieurs instruction `return` peuvent apparaître dans le corps d'une même fonction. Dans ce cas, le retour au programme appelant sera provoqué par le premier `return` recontré lors que l'éxecution.

Par exemple, la fonction `min` qui calcule le minimum de deux nombres entiers peut être définie en langage C comme suit :

```
int min(int a, int b) {
 if(a < b)
   return a;
 else
   return b;
 }
```

la valeur de l'expression `min(15,4)` est la valeur retournée par le corps de la fonction `min` en affectant la valeur `15` à `a` et la valeur `4` à `b`.

Voici une fonctions qui, à partir d'un nombre flottant a et d'un entier n, calcule a^n.

```
float puissance(float a, int n) {
  float p=1;
  int i;
  if (n > 0){
```

```
    for(i = 1; i<=n ; i++)
      p=p*a;
    }
  else if (n < 0){
    for (i = -1; i>=n ; i--)
      p=p/a;
    }
  return p;
}
```

Il n'y a pas qu'une unique manière de définir une même fonction. Tout d'épend de l'algorithme choisi. La fonction `power` décrite ci-dessous fait exactement le même calcul que la fonction `puissance` décrite au dessus mais utilise un algorithme récursif : elle fait appel à elle-même.

```
float power(float a, int n) {
if (n == 0)
  return(1);
else if (n > 0)
  return(a*power(a,n-1));
else
  return(power(a,n+1)/a);
}
```

3.1.2 Prototype d'une fonction

Le langage C n'autorise pas les fonctions imbriquées. les définitions des fonctions auxiliaires doivent donc être placées avant ou après la fonction `main`. Il est cependant necéssaire que le compilateur "connaisse" la fonction avant son premier appel. A priori, les fonctions doivent donc être définies avant que l'on fasse appel à elles. Cela peut être problématique si on utilise une première fonction qui fait appel à une deuxième fonction qui fait elle-même appel à la première... la quelle des deux doit-on définir en premier ?

Pour éviter ce genre de désagréments, on peut déclarer un fonction par son **prototype**, qui donne seulement son nom, son type et celui de ces arguments, sous la forme :

type Nom_fonction (type-1 arg-1, type-2 arg2, ... type-N argN) ;

et donner plus loin, après la fonction `main` par exemple, la définition de complète de la fonction *Nom_fonction*. Les informations présentes dans le prototype sont suffisantes pour manipuler formellement les appels à la fonction *Nom_fonction* : elle permet au compilateur de vérifier le nombre et le type des paramètres utilisés dans la définition concordent bien avec ceux de la définition, de mettre en place d'éventuelles conversions des paramètres effectifs lorsque la fonction est appelée avec des paramètres dont les types ne correspondent pas aux types annoncés par le prototype...

Les fichiers d'en-tête (d'extension `.h` pour fichier Headers) de la bibliothèque standard contiennent notamment des prototypes de fonctions (ils peuvent également contenir les définitions de constantes globales). Leur définitions complètes sont stockées dans une bibliothèque logicielle qui sera liée au reste du programme lors que la phase d'édition des liens.

Par exemple, Supposons que l'on souhaite utiliser la fonction `printf()`. Le compilateur ne connaît pas, a priori, cette fonction bien qu'il s'agisse d'une fonction standard du langage C. Le prototype de la fonction `printf()` se trouve dans le fichier `stdio.h`.

Il s'agit donc de préciser au compilateur dans quel fichier se trouve le prototype de la fonction `printf()`. Pour cela, on ajoute la directive au microprocesseur `#include<stdio.h>`.

3.2 La fonction `main`

Jusqu'a présent, dans les exemples de programme complet qu'on a recontré dans ce cours, la fonction `main` avait pour définition quelque chose du type :

```
int main(void) {
  ...
  return 0;
}
```

L'instruction `return` *`statut`* `;`, qui renvoie un entier *`statut`* est utilisée dans le main pour fermer le programme.

En fait, la valeur de *`statut`* est transmise à l'environnement d'exécution (terminal, programme appellant, Système d'exploitation... selon d'où est lancé le programme). La valeur de retour 0 correspond à une terminaison du programme correcte, et la valeur 1 à une terminaison sur une erreur.

A la place de 0 ou 1, on peut utiliser comme valeur de retour la constante symbolique `EXIT_SUCCESS` (pour 0) et la constante symbolique `EXIT_FAILURE` (pour 1) qui sont définies dans `stdlib.h`, ou encore d'autres valeurs qu'on souhaite faire passer à l'environnement d'execution.

Une autre fonction permettant de quitter le programme est la fonction `exit` de la bibliothèque `stdlib.h`, dont le prototype est :

`void exit(`*`statut`*`) ;`

où *`statut`* a le même role et les mêmes valeurs possibles que lors de l'instruction `return` *`statut`* `;`.

Si `return` renvoie une valeur a la fonction appelante, donc ne quitte le programme que si lorsqu'elle est dans le main. L'appel à la fonction `exit()` quitte le programme quelque soit la fonction dans laquelle a lieu l'appel. La fonction `exit()` est donc à éviter si on fait des appels récursifs à la fonction `main` : `return` fera terminer l'appel courrant à `main` alors que `exit()` fera terminer le premier appel au `main` et donc tous les suivants (l'appel recursif a la fonction `main` n'est pas quelque chose de courrant ou souhaitable dans un programme en C, mais on sait jamais...).

La fonction `main` peut avoir des paramètres formels. Pour recevoir une liste d'arguments au lancement de l'exécution du programme, la ligne de commande qui sert à lancer le programme est alors composée du nom du fichier exécutable, suivi par les valeurs des paramètres. L'interpréteur de commande passe alors à la fonction `main` ces arguments.

Par convention, la fonction `main` son prototype standard est par convention :

```
int main(int argc,char* argv[]);
```

Elle possède deux paramètres formels :

- un entier `argc` (pour ARGument Count) qui compte le nombre de mots qui constituent la ligne de commande d'éxecution,
- un tableau de chaines de caractères de longueur variable `argv` (pour ARGument Vector) où sont stockés les mots de la ligne de commande : `argv[0]` contient le nom de l'executable et les chaines `argv[1]`... `argv[argc-1]` contiennent les paramètres.

Remarque : Le parametre `argv` est en fait un tableau d'élément de type `char *` (pointeur sur des éléments de type `char`). Ces objets seront étudiés au Chapitre 5.

On utilisera dans la suite de ce cours l'écriture standard de la fonction `main` :

```
int main(int argc,char * argv[]){
   ...
   return EXIT_SUCCESS;
}
```

L'utilisation dans le code de l'entier `argc` permet d'avoir un controle sur le nombre de parametres rentrés par l'utilisateur. De plus, les paramètres passés par l'utilisateur sont utilisables dans le code. Ces paramètres qui sont stockés sous forme de chaines de caractère (voir Chapitre suivant), doivent être converties si elles représentent un nombre, entier ou flottant. Pour cela, on utilise les fonctions suivantes, qui prennent toutes en argument une chaine de caractère `s` :

- `atoi(s)` qui renvoie une variable de type `int` dont l'écriture décimale, octale ou héxadécimale est donnée par `s`.
- `atol(s)` qui renvoie une variable de type `long` dont l'écriture décimale, octale ou héxadécimale est donnée par `s`.
- `atof(s)` qui renvoie une variable de type `double` dont l'écriture décimale, octale ou héxadécimale est donnée par `s`.

Voici comme exemple un programme qui calcule le produit de deux entiers que l'utilisateur passe en argument de l'éxecutable :

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

int main (int argc,char* argv[]){
  int a,b;

  if(argc!=3){
     printf("Nombres d'arguments invalide.\n Usage:%s int int\n",argv[0]);
     return EXIT_FAILURE;
     }
  a=atoi(argv[1]);
  b=atoi(argv[2]);
  printf("Le produit de %d et %d vaut %d. \n",a,b,a*b);
  return EXIT_SUCCESS;
}
```

Pour lancer le programme, on utilise par exemple la ligne de commande

```
./a.out 12 8
```

Dans cet exemple, l'argument `argv` est alors un tableau de 3 chaines de caractères :

- `argv[0]` est `"./a.out"`,
- `argv[1]` est `"12"`,
- `argv[2]` est `"8"`.

Ce sont des chaines de caractères : pour utiliser les deux nombres en tant qu'entier, il est nécessaire d'utiliser la fonction `atoi()` qui à une chaine de caractères renvoie l'entier dont elle est l'écriture décimale (fonction de la bibliothèque `stdlib`). Cette exexution renvoie donc :

```
Le produit de 12 et 8 vaut 96.
```

Par contre, la commande `./a.out 12` renvoie à l'utilisateur :

```
Nombres d'arguments invalide.
Usage: ./a.out int int
```

3.3 Stockage des variables et Visibilité des identificateurs

Les variables manipulées dans un programme en C ne sont pas toutes logées à la même enseigne. En particulier, elle n'ont pas toutes la même façon d'être stockée, ni la même "durée de vie".

3.3.1 Variables permanentes et variables temporaires

Il y a deux catégories de variables, selon la manière dont le compilateur définit leur stockage.

On a d'une part les **les variables permanentes (ou statiques)**. Une variable permanente occupe un emplacement mémoire qui reste fixe pendant toute l'execution du programme. Cet emplacement est alloué une fois pour toute lors de la compilation. La partie de la mémoire contenant ces variables est appellée **segment de données**. Par défaut, les variables permanentes sont initialiées à zéro par le compilateur. Elles sont caractérisées par le mot-clé `static`. L'effet de la classe `static` dépend de l'endroit où l'objet est déclaré (voir les deux paragraphes ci-dessous).

D'autre part, il y a les **les variables temporaires**. Les variables temporaires se voient allouer un emplacement mémoire de façon dynamique durant l'exécution du programme. Elle ne sont pas initialisée par défaut et l'emplacement mémoire est libéré à la fin de l'exécution de la fonction dans laquelle elle est située. La partie de la mémoire contenant ces variables est appellée **segment de pile**, il existe un spécificateur de type correspondant à ces variables : `auto` pour **variable de classe automatique**, mais ce spécificateur est inutile puisque il s'applique uniquement aux variables temporaires qui sont des variables automatiques par défaut (c'est un vestige du langage B, prédécesseur du langage C).

Une variable temporaire peut également être stockée dans le registre de la machine, qui est beaucoup plus rapide d'acces qui le reste de la mémoire (c'est utile lorsqu'une variable est utilisée fréquamment dans un programme). Pour stocker une variable temporaire dans le registre, on spécifie le type avec le mot-clé `register`. Comme le nombre de registres est limité, cette requête n'est satisfaite que s'il reste des emplacements disponibles. Cela dit, les options d'optimisation du compilateur `gcc` (comme l'option `-O`) sont maintenant plus efficaces que le forçage "manuel" de stockage dans le registre effectué grace à `register`.

D'autre part, il peut éventuellement être nécessaire de pouvoir, à des moments arbitraires de l'exécution, demander au système l'allocation de nouvelles zones de mémoire, et de pouvoir restituer au système ces zones (allocation et désallocation dynamique de la mémoire), comme par exemple, lorsque la taille des variables à stocker est variable elle aussi (dépendant d'un parametre que l'utilsateur doit choisir). Dans ce cas, l'allocation et la libération de la mémoire sont sous la responsabilité du programmeur et ces zones sont prises dans une partie de la mémoire appelé **segment de tas**.

La compilation d'un programme crée des fichiers binaires contenant les instructions des fonctions (et les liens vers les différents autres segments) qui seront placés dans une zone mémoire appelée **segment de code**. Lors du lancement de l'exécutable, le processeur ira chercher ses premières instructions dans le segment de code de la fonction `main` du programme.

les segments de pile et les segments de tas sont alloués au programme à chaque execution puis libérés lorsque le programme se termine. Une fois la compilation effectuée, le segment de code et le segment de données sont fixes.

La figure 3.1 représente l'organisation dans la mémoire des différents segments pour un programme `prog.c` contenant la fonction `main` et une fonction auxiliaire `aux`.

3.3.2 Variables globales et variables locales

La durée de vie des variables est liée à leur **portée**, c'est-à-dire à la portion du programme qui les contient.

On appelle **variable globale** une variable déclarée en dehors de toute fonction. Elle est alors connue du compilateur dans toute la portion de code qui suit sa déclaration et modifiable par toute fonction qui l'utilise dans cette portion de code. Les variables globales sont systèmatiquement permanentes et donc initialisées par défaut à 0.

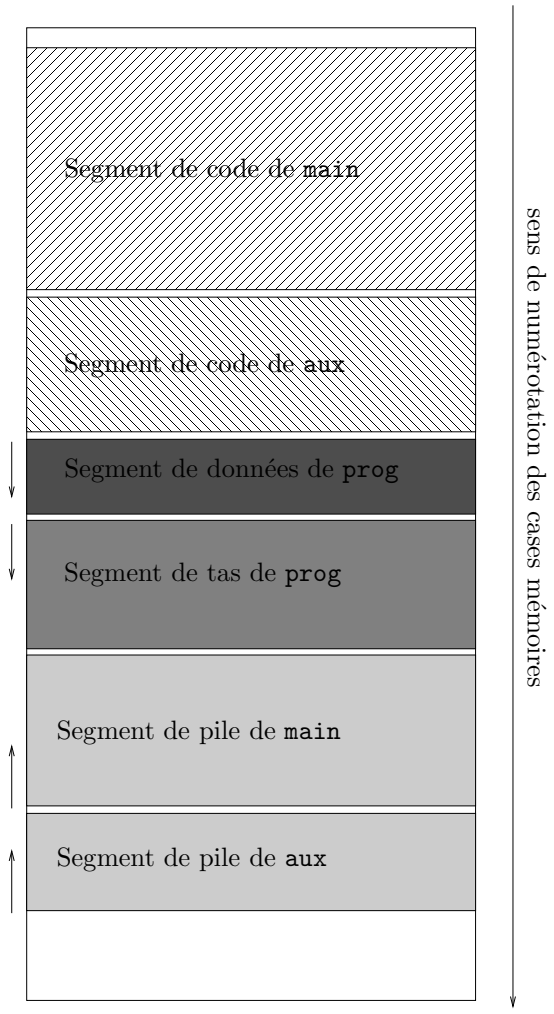


FIG. 3.1 – Schéma de l'organisation mémoire

Comme une variable globale est déjà permanente, le mot-clé `static` aura pour effet de limiter la portée de la variable ou de la fonction au seul fichier où elle est déclarée. Toutes les variables globales et fonctions qui ne sont pas déclarées (ou définies) en `static` sont externes par défaut. Par exemple, le programme suivant :

```
#include<stdio.h>

int n;
void compteur() {
  n++;
  printf(" %d -",n);
  return;
  }
int main(int argc, char *argv[]) {
  int i;
  for(i = 0;i < 5; i++) compteur();
  return EXIT_SUCCESS;
  }
```

va afficher à l'écran : `1 - 2 - 3 - 4 - 5 -`. L'initialisation de `n` à zéro est automatique car `n` est globale donc permanente et chaque appel de la fonction `compteur` incrémente sa valeur de 1.

On appelle **variable locale** une variable déclarée à l'interieur d'une fonction (ou d'un bloc d'instructions) du programme. Elles sont temporaires par défaut. Lorsqu'une fonction est appellée, elle place ces variables locales dans la pile. A la sortie de la fonction, la pile est vidée et les valeurs des variables locales sont perdues. Leurs valeurs ne sont pas conservées pour les prochains appels de la fonction.

Les variables locales n'ont aucun lien avec les variables globales du même nom. **La variable locale prime sur la variable globale**. Par exemple, le programme suivant :

```
#include<stdio.h>

int n=500000;
void compteur() {
  int n=0;
  n++;
  printf(" %d -",n);
  return;
  }
int main(int argc, char *argv[]) {
  int i;
  for(i = 0;i < 5; i++)  compteur();
  printf(" %d\n",n);
  return EXIT_SUCCESS;
  }
```

va afficher à l'écran : `1 - 1 - 1 - 1 - 1 - 500000`. A chaque appel de la fonction `compteur`, une variable locale `n` est initialisée à zéro puis incrémentée de 1. C'est la variable locale `n` qui est utilisée durant tout l'appel à la fonction `compteur` : la variable globale `n` n'a pas été modifiée.

On peut cependant forcer une variable locale à être permanente, en utilisant le spécificateur de type `static`. La valeur de la variable sera alors permanente entre les différents appels de la fonction. La variable ne sera visible que dans la fonction, mais ne sera pas réinitialisée à chaque appel de la fonction. L'intérêt est de garantir une certaine encapsulation, afin d'éviter des usages multiples d'une variable globale. Qui plus est, cela permet d'avoir plusieurs fois le même nom, dans des fonctions différentes.

Le programme suivant :

```
#include<stdio.h>

int n=500000;
void compteur() {
  static int n;
  n++;
  printf(" %d -",n);
  return;
  }
int main(int argc, char *argv[]) {
  int i;
  for(i = 0;i < 5; i++) compteur();
  printf(" %d\n",n);
  return EXIT_SUCCESS;
  }
```

va afficher à l'écran : `1 - 2 - 3 - 4 - 5 -500000`. La variable locale `n` est permanente, initialisée implicitement à 0 et sa dernière valeur est conservée pour les appels suivants à la fonction `compteur`. Cependant, en dehors de la fonction `compteur`, seule la variable globale `n` est visible.

3.4 Transmission des arguments par valeurs et par adresses

Les paramètres d'une fonction sont traités de la même manière que les variables locales de classe automatique : Lors de l'appel d'une fonction, les paramètres effectifs sont copiés dans le segment de pile et la fonction travaille uniquement sur ces copies, qui disparaissent lors du retour à la fonction appelante. Par défaut, si la fonction modifie un de ses paramètres, seule la copie sera en fait modifiée, la variable de la fonction appelante restant inchangée. On dit que les paramètres d'une fonction sont **transmis par valeurs**. Par exemple, si on fait tourner le programme suivant, sensé intervertir les valeurs de deux variables :

```
#include<stdio.h>

void flip(int a, int a) {
  int t;
  printf("debut flip: a=%d et b=%d\n", a,b);
  t=a;
  a=b;
  b=t;
  printf("fin flip: a=%d et b=%d\n", a,b);
  return;
 }
int main(int argc, char *argv[]) {
  int x=2;
  int y=5;
  printf("debut main: x=%d et y=%d\n", x,y);
  flip(x,y);
  printf("fin main: x=%d et y=%d\n", x,y);
  return EXIT_SUCCESS;
 }
```

on voit s'afficher :

```
debut main: x=2 et y=5
debut flip: a=2 et b=5
fin flip: a=5 et b=2
fin main: x=2 et y=5
```

Les valeurs de `x` et `y` n'ont pas été échangées...

Pourquoi donc ? Tout simplement car à l'appel de la fonction `flip` dans la fonction `main`, les valeurs de `x` et `y` sont copiées dans le segment de pile dévolu au stockage des variables `a` et `b`. Leurs valeurs sont ensuite interverties mais la fonction `flip` ne touche pas aux valeurs contenues au emplacements mémoires dévolus à `x` et `y`... les valeurs qu'ils contiennent sont donc inchangées.

Pour qu'une fonction modifie la valeur d'un de ces arguments, il faut qu'elle ait pour paramètre l'adresse mémoire de l'objet et non sa valeur. On dit que les arguments sont **transmis par adresses**. Connaissant leurs adresses, le programme ira modifiera les cases mémoire qui contiennent les variables à modifier.

On verra comment construire des fonctions modifiant leur paramètres plus tard, en utilisant les pointeurs, au Chapitre 5, Paragraphe 5.5.

3.5 Qualificateurs de type `const` et `volatile`

Les qualificateurs `const` et `volatile` permettent d'*encadrer* les possibilités de modifications d'une variable. Il se place juste avant le type de la variable.

La classe `const` ne déclare pas une vraie constante, mais indique au compilateur que la valeur de la variable ne peut pas être changée par le programme. Il est donc impératif d'assigner une valeur à la déclaration de la variable, sans quoi toute tentative de modification ultérieure entrainera une erreur de la part du compilateur.

Attention toutefois à son utilisation avec les pointeurs :

```
char * const p;
```

définit un pointeur constant sur un caractère : le pointeur `p` est garanti constant mais pas son contenu.

```
const char * p;
```

définit un pointeur sur un caractère constant : Le caractère pointé `p` est garanti constant mais le pointeur `p` peut être modifié.

```
const char * const p;
```

définit un pointeur constant sur un caractère constant : le pointeur `p` et son contenu garanti constant.

Le mot-clé `volatile` sert à spécifier au compilateur que la variable peut être modifiée à son insu. Cette classe de variable sert lors de la programmation système. Elle indique qu'une variable peut être modifiée en arrière-plan par un autre programme (par exemple par une interruption, par un thread, par un autre processus, par le système d'exploitation ou par un autre processeur dans une machine parallèle). Cela nécessite donc de recharger dans un registre du processeur cette variable à chaque fois qu'on y fait référence et ce même si elle se trouve déjà dans un de ces registres (ce qui peut arriver si on a demandé au compilateur d'optimiser le programme).

Remarque : On peut combiner const et volatile dans certaines situations. Par exemple :

```
extern const volatile int horloge_temps_reel;
```

déclare une variable entière, qu'on ne peut modifier à partir du programme, mais dont la valeur peut changer quand même. Elle pourrait désigner une valeur incrémentée régulièrement par une horloge interne.

3.6 Directives au préprocesseur et compilation conditionnelle

3.6.1 la directive `#include`

La directive au préprocesseur `#include` permet d'inorposer dans le fichier source un texte d'un autre fichier. Ce fichier peut-être une en-tête de bibliothèque (`stdlib.h`, `math.h`...) où n'importe quel autre fichier (bibliothèque personnelle de fonctions...). Deux syntaxes voisines peuvent être utilisées :

`#include<`*`nom_fichier`*`>`

pour inclure des fichiers qui se trouvent dans des dossiers définis par l'implémentation (ex : `/usr/include` ou `/usr/local/include`).

`#include"`*`nom_fichier`*`"`

pour inclure des fichiers du répertoire courant. On peut specifier d'autres repertoire à l'aide de l'option `-I` du compilateur `gcc`.

EN général, on utilise la première syntaxe pour les fichiers en-tête des bibliothèques standard et la deuxième syntaxe pour les fichiers crées par l'utilisateur.

3.6.2 Définitions de macros : la directive `#define`

Une **macro** est une association entre un indentificateur et un "texte". Cette directive permet de définir des **pseudo-constantes** ou des **pseudo-fonctions**. Le préprocesseur remplacera chaque occurrence de l'identificateur rencontrée dans le programme par le "texte" spécifié. On définit une macro grâce à la directive suivante au préprocesseur :

`#define` *`NOM texte`*

L'usage est de mettre les identificateurs des pseudo-constantes en majuscules. Exemple :

```
#define TAILLE 100
#define SAUT_DE_LIGNE putchar('\n')
```

Attention : Quelques règles pour la définition des macros :

1. Eviter de mettre un point virgule à la fin de la directive. Par exemple, si on définit :

   ```
   #define TAILLE 100;  /*  ; de trop  */
   ```

   L'instruction `i = TAILLE - 2;` serait remplacée par le préprocesseur par :

   ```
    i = 100;  - 2;
   ```

   Notez que l'instrcution `- 2;` est une expression valide en C...

2. Bien parenthéser les expressions composées. Par exemple, si on définit :

   ```
   #define TAILLE 10
   #define MAXI   2 * TAILLE + 3
   ```

   L'instruction `int p=MAXI*2;` fixe la valeur de l'entier `p` à 26 (`2 * 10 + 3 *2`) au lieu de 46. La bonne définition de `MAXI` est

   ```
   #define (MAXI 2 * TAILLE + 3)
   ```

On peut également définit des pseudo-fonctions, en utilisant des directives au préprocesseur avec la syntaxe suivante :

`#define` *NOM_Macro*`(`*param-1*`,`*param-2*`,... )` *texte*

Le préprocesseur remplace toute occurrence de *NOM_Macro*`(X-1,X-2,...)` par *texte*. Les paramètres formels *param-i* apparaissant dans *texte* sont remplacés par les paramètres effectifs `X-i`.

Par exemple si l'on définit :

```
#define CARRE(X)    X*X
#define SOMME(X,Y)  X+Y
```

Les instructions

```
z = CARRE(a);
y = CARRE(c+1);
```

seront remplacées par le préprocesseur remplace par :

```
z = a*a;
y = c+1*c+1;    /* étonnant !!!  équivalent de :  c+(1*c)+1 */
```

Comme pour la définitions des pseudo-constantes, il faut utiliser des parenthèses :

```
#define CARRE(X) (X)*(X)

y = CARRE(c+1);
       /*  --> y = (c+1)*(c+1);  */
z = (short) CARRE(c+1);
       /*  --> z = (short) (c+1)*(c+1); */
       /* équivalent  ((short) (c+1)) * (c+1) */
```

Ce qui n'est pas encore satisfaisant. Il faut définir la pseudo-fonction `CARRE` ainsi :

```
#define CARRE(X) ((X)*(X))

z = (short) CARRE(c+1);
       /*  --> z = (short) ((c+1)*(c+1)); */
```

Attention : Quelques règles pour la définition des macros :

1. *NOM_Macro* doit être immédiatement suivi d'une parenthèse ouvrante.
2. Il ne faut pas utiliser dans les paramètres réels des opérateurs pouvant générer des "effets de bord". Par exemple, si on définit :

   ```
   #define abs(X) ( ((X) < 0) ? -(X) : (X) )
   ```

   L'appel `abs(i++);` incrémenterait 2 fois `i` car dans la macro `abs`, `i` est évalué 2 fois.
3. Une macro ne peut pas être utilisée comme paramètre d'une fonction, car il n'y a pas d'adresse de point d'entrée.
4. Une macro peut tenir sur plusieurs lignes si chaque ligne de source est terminée par le symbole `\` (dans la signification UNIX, `\` annule le caractère suivant, ici c'est la fin de ligne qui est annulée). Exemple :

```
#define erreur(msg) {                    \
         fprintf(stderr, "%s\n", msg); \
         exit(1);                        \
         }
```

5. Comme pour la définition des pseudo-constante, eviter de mettre un point virgule à la fin de la directive.

Pour supprimer une définition, on utilise la directive `#undef` dont la syntaxe est :
`#undef` *NOM_Macro*
Cette directive annule le symbole spécifié par *NOM_Macro* qui avait été défini auparavant à l'aide de la directive `#define`.

```
#define max(a,b) ( ((a) > (b)) ? (a) : (b) )

     z = max(x,y);     /* appel de la macro max */
     #undef max
     z = max(x,y);     /* appel de la fonction max */
```

3.6.3 Compilation conditionnelle

La **compilation conditionnelle** permet d'inclure ou d'exclure des parties de code source dans le texte généré par le préprocesseur. La compilation du programme pourra alors s'adapter au matériel ou à l'environnement dans lequel le programme doit être exécuté selon les choix faits par l'utilisateur, passés en option de la commande de compilation.

Deux type de conditionnement sont pris en compte par les directives de compilation conditionnelle :
– les conditions sur la valeur d'une expression,
– les conditions d'existence (ou pas...) de symbôles.

Les valeurs des expressions et l'existence (ou pas) des symbôles en questions sont passés en options de `gcc` dans la ligne de commande de compilation : Certaines options de cette ligne de commande permettent de définir ou non des constantes symbolique utilisées par le préprocesseur et ainsi d'orienter ses traitements.

– `-D` *const* permet de définir la constante symbolique *const*,
– `-D` *const=valeur* permet en plus, lui donner la valeur *valeur*,
– `-U` *const* permet d'annuler la définition de la constante symbolique *const*.

La syntaxe d'une instruction de compilation conditionnelle concernant la valeur d'une ou plusieurs constantes symboliques est la suivante :

```
#if condition-val1
  /* Code à compiler si la condition 1 est vraie */
#elif condition-2
  /* Sinon si la condition 2 est vraie compiler ce bout de code */
...
#elif condition-N
  /* Sinon si la condition N est vraie compiler ce bout de code */
#else
  /* Sinon on compile ce bout de code */
#endif
```

Les valeurs des expressions `#elif` *condition-i* sont les résultats d'une opération booléenne mettant en jeu des tests sur des constantes symboliques.

Les opérateurs booléens utilisables ont la même syntaxe que pour ceux du C : `&&`, `||`, `!`, de même pour les opérateurs de comparaison : `==`, `<`, `<=`, `>`, `>=`.

Il existe également un opérateur booléen `defined(`*`SYMBOLE`*`)` qui est VRAI si *`SYMBOLE`* est défini.

Voici un exemple de directive de compilation conditionnelle au preprocesseur

```
#if (DEBUG==2) && !defined(ESSAI)
   (void)puts("DEBUG defini a 2 et ESSAI non defini");
#endif
```

Dans cet exemple, issu d'un programme `prog.c`, si la constante symbolique `DEBUG` est définie et contient la valeur `2` et la constante symbolique `ESSAI` n'est pas définie, alors la ligne

`(void)puts("DEBUG defini a 2 et ESSAI non defini");`

sera conservée par le préprocesseur dans le code à compiler.

La ligne de compilation suivante remplit ces conditions :

```
gcc -c -D DEBUG=2 prog.c
```

Lors de l'execution de la ligne de commande `gcc -c -D DEBUG=2 -D ESSAI prog.c`, par contre, la ligne

`(void)puts("DEBUG defini a 2 et ESSAI non defini");`

ne sera pas conservée par le préprocesseur dans le code à compiler.

On peut également utiliser la commande `#ifdef`, qui marche un peu comme un `#if` à la seul différence qu'il vérifie seulement si une constante a été définie. Par exemple :

```
#define Linux

#ifdef Linux
    /* code pour Linux */
#endif
```

est équivalent à :

```
#define Linux

#if defined(Linux)
    /* code pour Linux */
#endif
```

Attention l'opérateur `defined` ne peux être utilisé que dans le contexte d'une commande `#if` et `#elif`. L'opérateur defined combiné à des commandes `#if` et `#elif` permet de construire des conditions logiques plus complexes.

D'autre part, la commande `#ifndef` *`SYMBOLE`* permet de tester si une constante symbolique ici SYMBOLE n'est pas définie. C'est un équivalent de `#if !defined(`*`SYMBOLE`*`)`.

`#ifndef` est très utilisé dans les fichiers d'entête (ceux d'extension `.h`) pour éviter les inclusions infinies ou multiples. En effet, si on suppose le cas de figure suivant :

Un programme fait appel à deux fichiers d'en-tête `bib1.h` et `bib2.h`. On suppose que le fichier `bib1.h` contient la directive au préprocesseur `#include"bib2.h"` et le fichier `bib2.h` contient à son tour la directive au préprocesseur `#include"bib1.h"` : Le premier fichier a besoin du second pour fonctionner, et le second a besoin du premier. Ce qui va se passer au moment de la précompilation :

1. L'ordinateur lit `bib1.h` et voit qu'il faut inclure `bib2.h`
2. Il lit `bib2.h` pour l'inclure, et là il voit qu'il faut inclure `bib1.h`

3. Donc il inclut `bib1.h` dans `bib2.h`, mais dans`bib1.h`, on lui indique qu'il doit inclure `bib2.h`...

Ce scénario va boucler à l'infini. Pour éviter cela, il faut spécifier au préprocesseur d'inclure le fichier `bib1.h` s'il n'a pas été inclu avant. Cela s'effectue dans le fichier `bib1.h` avec la syntaxe suivante :

```
#ifndef BIB1_H
  #define BIB1_H
  /* Contenu de bib1.h */
#endif
```

D'une manière générale, on utilise la syntaxe :

```
#ifndef FICHIER_H
#define FICHIER_H
   /* Contenu de fichier.h */
 #endif
```

où `FICHIER_H` représente le l'en-tête `fichier.h` en majuscule. Ce mécanisme va éviter les inclusion en boucle.

Si plusieurs fichiers d'entête inclus demandent tous les deux d'inclure le même troisième. Toutes les définitions, prototypes... contenues dans ce dernier fichier vont être répétés dans le résultat produit par le préprocesseur.

Remarque : Il existe aussi une directive `#error` qui sert à arrêter la compilation lorsque vous jugez que votre programme ne pourra pas fonctionner, si par exemple une plate-forme n'est pas supportée, une ressource n'a pas été trouvée, une constante symbolique n'a pas la bonne valeur. .

Elle est souvent placée dans la partie `#else` d'une instruction de compilation conditionnelle `#if ... #else ... #endif`.

Sa syntaxe est la suivante : `#error "message d'erreur"`. Lorsque le compilateur arrive à cette ligne, il arrête la compilation et affiche message d'erreur. Un petit exemple :

```
#if defined(HAVE_DIRENT_H) && defined(HAVE_SYS_TYPES_H)
   #include <dirent.h>
   #include <sys/types.h>
#else
   /* Arret : dirent.h et sys/types.h non trouves sur ce systeme */
   #error "Readdir non implemente sur cette plateforme"
#endif
```

Si les fichiers d'en-tête `dirent.h` et `sys/types.h` sont présents dans le système, alors on les inclut, sinon, on renvoie un message d'erreur : ça ne sers à rien d'aller plus loin dans la compilation.

Chapitre 4

Types composés

Dans un programme, il est souvent nécessaire de manipuler des valeurs structurées. Par exemple, on peut vouloir manipuler un relevé de mesure composée d'un nombre fixe ou variable de nombres entiers ; un point de l'espace qui a un nom, une abscisse et une ordonnée, c'est-à-dire qu'on peut le voir comme un trplet formé d'un caractère et deux nombres flottants ; une figure géométrique composée elle-même de plusieurs points...

Pour cela, les types de base du langage C ne sont pas suffisants. Pour construire de telles valeurs, le langage C offre deux constructions possibles :

- Les **tableaux**, pour les structures dont tous les éléments sont du même type,
- Les **structures**, pour lesquelles les composantes peuvent être de types différents.

4.1 Tableaux

Un **tableau** est une suite de variables du même type (type de base ou type structuré), appellés **éléments du tableau**. On distinguera les tableaux monodimensionnels dont les éléments ont pour valeurs des nombres (entiers ou flottants), des adresses ou des structures, et les tableaux multidimensionnels dont les éléments sont eux-même des tableaux.

La déclaration d'un **tableau monodimensionnel** a la forme suivante :

type Nom_Tableau[*NOMBRE_ELEMENTS*] ;

Cette déclaration crée du tableau, composé de *NOMBRE_ELEMENTS* éléments de type *type* et désigné par l'indentificateur *Nom_Tableau*. *NOMBRE_ELEMENTS* est la *taille* du tableau et doit être un entier strictement positif.

Une telle déclaration alloue un espace mémoire de `sizeof(`*type*`) * `*NOMBRE_ELEMENTS* octets consécutifs pour stocker le tableau.

Les éléments du tableau sont numérotés de `0` à *NOMBRE_ELEMENTS*`-1`. On peut accéder à chaque élément en utilisant l'opérateur `[]`. Par exemple, pour affecter la valeur `0` au 3^{e}élément d'un tableau `Tab` de 10 entiers, on utilise l'instruction `Tab[2]=0;`.

Pour mettre à 1 tous les éléments de même tableau `Tab`, on utilise le bloc suivant :

```
int i;
for(i=0; i<10 ; i++){
  Tab[i] = 1;
  }
```

On peut également initialiser un tableau lors de sa déclaration par une liste de constantes :

type Nom_Tableau[N] = {*const-1,const-2,...,const-N* };

Si le nombre de données dans la liste d'initialisation est inférieur à la taille du tableau, seuls les premiers éléments seront initialisés.

On peut omettre la taille du tableau dans le cas d'une déclaration avec initialisation et utiliser l'instruction suivante :

type Nom_Tableau[] = {*const-1,const-2,...,const-N* };

Pour résumer, on a trois manières équivalentes de déclarer et initialiser un tableau `tab` représentant par exemple la série d'entiers $[2, 3, 4, 5]$:

- `int tab[]={2,3,4,5};`,
- `int tab[4]={2,3,4,5};`,
-
```
int tab[4];
tab[0]=2;
tab[1]=3;
tab[2]=4;
tab[3]=5;
```

Le langage C permet également de déclarer des tableaux **multidimensionnels** (des tableaux de tableaux en quelque sorte). Par exemple, on déclare un tableau bidimensionnel de la manière suivante :

type Nom_Tableau[NB_lignes][NB_colonnes] ; C'est en fait un tableau dont les éléments sont eux-mêmes des tableaux délements de type *type*.

On accède à l'élément ligne `i`, colonne `j` par l'intermédiaire de l'expression `Nom_Tableau[i][j]`.

L'initialisation d'un tableau bidimensionnel, comme par exemple un tableau d'entiers représentant la matrice $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$ se fait pas le biais d'une liste de listes comme suit

```
int TAB[2][3] = {{1,2,3},{4,5,6}};
```

Plus simplement, on pourra déclarer ce tableau ainsi :

```
int TAB[2][3] = {1,2,3,4,5,6};
```

ou en initialisant ses éléments un à un comme cela :

```
int TAB[2][3];
TAB[0][0] = 1;
TAB[0][1] = 2;
TAB[0][2] = 3;
TAB[1][0] = 4;
TAB[1][1] = 5;
TAB[1][2] = 6;
```

On peut bien évidemment déclarer des tableaux de dimension plus grande que 2. Par exemple, on déclare un tableau de dimension 4 et de taille `taille-1`$\times$ `taille-2`$\times$`taille-3` $\times$ `taille-4` de la manière suivante :

type Nom_Tableau[taille-1][taille-2][taille-3][taille-4] ;

Dans ce tableau, on accède à l'élément de coordonnées (`i`,`j`,`k`,`m`) par l'intermédiaire de l'expression `Nom_Tableau[i][j][k][m]`.

Remarque : On peut, comme pour les types simples, regrouper les déclarations de tablaux de diffréentes trailles d'éléments du même type (MAIS C'EST PAS BIEN...). Par exemple, l'instruction :

```
int i, ligne[4], matrice[2][3];
```

déclare simultanément une variable `i` de type `int`, une liste `ligne` de 4 `int` et un tableau `matrice` à 2 lignes et 3 colonnes.

ATTENTION : Un tableau ne peut pas figurer à gauche d'un opérateur d'affectation (il ne peut pas être une Lvalue). Par exemple, si `Tab1` et `Tab2` sont deux tableaux, ON NE PEUT PAS écrire "`Tab1 = Tab2;`".

Pour remplir le tableau `Tab1` avec les éléments de ceux du tableau `Tab2`, en supposant qu'il s'agit de deux tableaux de `10` entiers, on devra effectuer l'affectation élément par élément comme suit :

```
int Tab1[10],Tab2[10];
int i;
for(i=0;i<10;i++){
  Tab1[i] = Tab2[i];
  }
```

Les phrases suivantes, expliquant le pourquoi de ce phénomène prendront leur sens après le Chapitre 5 :

Un tableau est en fait un pointeur constant, vers le premier élément du tableau. La valeur de l'expression `Tab1` est donc l'adresse mémoire du premier élément de ce tableau. D'autre part, le pointeur est constant, ce qui implique en particulier qu'il ne peut être modifié : aucune opération globale n'est autorisée sur un tableau.

4.2 Chaines de caractères

La plupart des applications informatiques nécessitent le traitement de données textuelles. Pour représenter ces données, les langages de programmation mettent à disposition le type caractère et les chaînes de caractères. Comme ces objets sont très frequemment utilisés, la bibliothèque standard du C offre de nombreuses fonctions spécifiques pour ces objets.

On appelle **chaîne de caractère** un tableau de caractères dont la fin est marquée par le caractère de code ASCII égal à zéro (symbole `\0`). Une chaine de caractère est donc désignée par l'adresse de son premier caractère (ou par l'adresse du caractère `\0` si la chaine est vide).

Ainsi, une chaine `s` contenant le mot "Bonjour" est initialisée ainsi :

```
char s[8] = {'B','o','n','j','o','u','r','\0'};
```

ou simplement

```
char s[] = {'B','o','n','j','o','u','r','\0'};
```

Cette écriture étant assez lourde, on peut utiliser l'écriture abrégée suivante :

```
char s[8] = "Bonjour";
```

ou encore plus compact :

```
char s[] = "Bonjour";
```

C'est en général cette dernière façon d'initialiser les chaines de caractères que l'ont utilisera.

Remarque : Attention à ne pas confondre un caractère (ex : `'A'`) et une chaine de caractère de longueur 1 (ex : `"A"`).

Nous avons déja rencontré les chaines de caractère lors de l'utilisation des fonctions d'entrées/sorties formatées (Chapitre 2, Paragraphe 2.8) : les chaines de controle du format sont des chaines de caractères.

Pour afficher une chaine de caractères, via un appel à la fonction `printf`, on utilise le spécificateur de format `%s` à l'interieur de la chiane de format elle même. Ce spécificateur de format écrit dans la sortie standard la chaine de caractère jusqu'au symbole `'\0'` qu'il n'écrit pas. On peut également utiliser une chaine de caractère pour stocker une chaine de controle du format, comme par exemple dans le bout de code suivant :

```
int j;
char s1[] = "%d";
char s2[] = "Journée";
char s3[] = "Bonne";
printf("%s %s\n",s3,s2);
printf(s1,j);
```

ATTENTION 1 : Si on a déclaré une chaine de caractère sans l'initialiser `char s[8];` on ne pourra PAS effectuer à posteriori une affectation du type : `s = "Bonjour";` car `s` est le nom d'un tableau et donc ne peut pas être une Lvalue. Chaque lettre de la chaine de caractère doit être affectée séparément.

On peut cependant initialiser une chaine de caractères avec une chaine plus courte :

```
char ch1[20] = "bonjour";
char ch2[20] = {'b','o','n','j','o','u','r','\0'};
```

les 12 derniers caractères sont initialisés à 0 ou non en fonction de la classe d'allocation.

Une autre erreur classique consiste à comparer deux chaines via leurs identificateurs en pensant comparer les chaines elles-mêmes (alors que l'on compare uniquement les adresses mémoire adresses de leur premiers éléments). Par exemple, si on définit deux chaines par :

```
char s1[] = "Bonjour", s2[] = "Bonjour";
```

Le test `s1 == s2` sera faux (valeur 0), car `s1` et `s2` sont les adresses des débuts des chaines qu'elles désignent. Pour comparer deux chaines de caractères, il faudra les comparer caractère par caractère...

Le module `string.h` de la bibliothèque standard regroupe pleins de fonctions utiles à la manipulation des chaines de caractères : calculer leur longueurs, les comparer, les concatener, extraire des sous-chaines, etc... Ci-dessous 4 fonctions sur les chaines qui penvent être utiles (mais il en existe beaucoup d'autres) :

- Obtenir la longueur d'une chaine de caractères :

  ```
  size_t strlen(const char *s);
  ```

  (le type `size_t` est un dérivé du type entier)
- Comparer une chaine de caractères à une autre :

  ```
  int strcmp(const char *s1, const char *s2);
  ```
- concaténer deux chaines de caractères :

  ```
  char *strcat(char *s1, const char *s2);
  ```
- copier une chaine de caractères dans une autre...

  ```
  char *strcpy(char *s1, const char *s2);
  ```

A propose de lecture des chaînes de caractères

Au chapitre 2, Paragraphe 2.8, nous avons vu les fonctions d'entrée-sortie formatées `scanf` et `printf` qui peuvent lire et écrire des chaines de caractères.

La fonction `scanf` utilisée avec le format `%s`, lit sur l'entrée standard une suite de caractères jusqu'à — et c'est là son principal inconvenient— la rencontre d'un caractère d'espacement espace, tabulation ou saut de ligne, la range à l'adresse indiquée et la complète par un symbôle `\0`.

Voici ce que cela donne, avec deux exécutions du bloc d'instructions suivant :

```
char s[20];
scanf("%s",&s);
printf("s = %s", s);
```

Si l'utilisateur saisi au clavier : `Bonjour␣`

il obtiendra l'affichage :

```
s = Bonjour
```

Si maintenant l'utilisateur saisi au clavier : `Bonne journée␣`

il obtiendra l'affichage :

```
s = Bonne
```

Le caractère d'espacement entre `Bonne` et `journée` met fin à la lecture. Or il est fréquent d'avoir à lire une ligne, c'est-à-dire une chaine de caractères contenant plusieurs mots espacées pas des espaces ou des tabulations et se terminant par un caractère de saut de ligne `\n`.

Pour lire une ligne complète, on peut utiliser la fonction `fgets`, dont le prototype est le suivant :

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

C'est en fait une fonction qui permet de lire dans les fichiers (voir Chapitre **??**) mais que l'on peut utiliser pour lire dans le flux d'entrée standard (`stdin`). Cette fonction lit au plus *`size-1`* depuis le flux *`stream`* (que nous pouvons remplacer par `stdin`) et les stocke dans la chaine de caractère *`s`*. La lecture s'arrête après `EOF` ou un saut de ligne. Un caractère nul `\0` est ajouté en fin de chaine.

Cette fonction renvoie le pointeur *`s`* rempli ou un pointeur `NULL` si il y a eu un problème.

Ainsi, en utilisant le bloc :

```
char s[20];
fgets(s,20,stdin);
printf("s = %s", s);
```

La saisie au clavier de `Bonne journée␣` provoquera bien l'affichage :

```
s = Bonne journée
```

4.3 Structures

Une **structure** est une suite finie d'objets de différents types, stockées de manières contigüe. Chacun de ces objets est appellé **champs** ou **membre** de la structure et possède un identificateur.

La déclaration d'un **modèle de structure** permet de définir le nom du modèle ainsi ques les types et les identificateurs des différents champs que comporte le modèle. La déclaration d'une structure suit la syntaxe suivante :

```
struct Nom_Structure {
  type1 Nom_champs-1 ;
```

```
    type2 Nom_champs-2 ;
    ...
    typeN Nom_champs-N ;
};
```

Par exemple, si on souhaite créer une structure qui représente un point du plan, décrit par son nom (une lettre), son abcisse et son ordonnée, on définit la structure `Point` comme suit :

```
struct Point {
  char  nom;
  float   x, y;
};
```

Lorsqu'une structure a été déclarée, on peut créer des objets de ce "nouveau type". Par exemple, si la structure *Nom_Structure* a été précédemment déclarée, on crée un objet *Nom_variable* de type structure correspondant à la structure `modele` via la déclaration :

```
struct Nom_Structure Nom_variable ;
```

Si la structure *Nom_Structure* n'a pas été précédemment déclarée, on peut faire d'une pierre deux coups (déclaration de la structure ET déclaration de la variable de ce type structuré) en utilisant la syntaxe suivante :

```
struct Nom_Structure {
  type1 Nom_champs-1 ;
  type2 Nom_champs-2 ;
  ...
  typeN Nom_champs-N ;
} Nom_variable ;
```

Losrqu'une variable d'un type structuré est déclarée, on peut accéder à ses différents champs via l'opérateur **champs de structure**, matérialisé par un point ".". Ainsi, le $\texttt{i}^{\text{e}}$ champs de l'objet *Nom_variable* de type `struct` *Nom_Structure* est désigné par l'expression :

```
Nom_variable.Nom_champs-i
```

Chacun de ces champs peut ensuite être modifié séparément et sera toujours accessible via cette expression. On peut effectuer sur cet objet toutes les opérations valides sur les éléments de type *typei*.

On peut initialiser les variables structurées lors de leur déclarations. Par exemple, si la structure `Point` est définie comme en début de paragraphe, on peut écrire :

```
struct Point p1 = {'A',2.5,1.};
```

Mais on peut aussi remplir les champs un à un :

```
struct Point p1;
p1.nom = 'A';
p1.x = 2.5;
p1.y = 1.0;
```

Par exemple, le programme suivant définit la structure `complexe`, calcule et affiche le module d'un nombre complexe :

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

struct complexe{
  double re,im;
 };
double module(struct complexe z){
  return sqrt(z.re*z.re + z.im*z.im);
 }
int main(){
  struct complexe z;
  double r;
  ...
  r=module(z);
  printf(" le module du nombre complexe %.2f+i%.2f est %f\n", z.re,z.im,r);
  return 0;
 }
```

Remarque concernant le match Structure VS Tableau :

Dans le petit programme précédent, faisant intervenir la structure `complexe`, on aurait pu éventuellement representer un nombre complexe comme un tableau `double[2]`. Cependant le fait d'en faire un type structuré rend leurs manipulation plus souple : contrairement aux tableaux, on peut appliquer l'opérateur d'affectation aux structures. Par exemple, si `z1` et `z2` sont deux objets de type `struct complexe`, on peut écrire l'instruction `z1 = z2;`.

Remarque sur la visibilité des structures :

0n peut déclarer des nouvelles structures à l'exterieur de toute fonction. Cela permet de pouvoir s'en servir dans tous les fonctions du programme. Cette nouvelle structure a une portée globale. Si par contre, elle est définie à l'interieur d'une fonction (ce qui est le cas dans la déclaration de la structure et déclaration de la variable de ce type structuré simultanée), les autres fonctions du programme ne pourront pas s'en servir.

Remarque sur l'utilisation des structures dans les fonctions :

Les nouveaux types, définis par des structures, peuvent être utilisés comme type de paramètre pour des fonctions mais également comme type des valeurs de retour des fonctions. Définir une fonction dont la valeur de retour est une structure permet en quelque sorte de définir une fonction ayant "plusieurs valeurs de retour" (matérialisés par les champs de la structure).

4.4 Enumérations

Les énumérations permettent de définir un type par la liste (finie) des valeurs qu'il peut prendre. On définit un objet de type énumération via le mot-clé `enum` et un identificateur de modèle, suivi de la liste des valeurs qu'il peut prendre :

`enum` *Nom_Enum* `{` *Constante-1, Constante-2, ..., Constante-N*`};`

La liste de noms symboliques `{` *Constante-1, Constante-2, ..., Constante-N*`}`correspond en fait à des valeurs entières : Tout type construit sur ce modèle est de fait une "copie" du type `int` dans laquelle un certain nombre de constantes symboliques ont été définies par exemple à des fins de lisibilité des applications. Définir une énumération ne définit donc pas un domaine de définition mais simplement quelques constantes symboliques pour des entiers (un entier peut d'ailleurs posséder

plusieus constantes symboliques dans une même énumération). Par défaut, les entiers sont attribués aux constantes par ordre croissant en commençant par 0. Par exemple, si on déclare le type `Couleurs` comme suit :

```
enum Couleurs { BLEU, ROUGE, JAUNE};
```

L'execution de l'instruction `printf("%d - %d - %d", BLEU, ROUGE, JAUNE);` renvoie à l'écran :
`0 - 1 - 2`.

Dans le programme, on pourra ensuite utiliser indifféremment l'entier `1` ou la constante symbolique `ROUGE`.

On peut également forcer l'attribution des entiers aux constantes symboliques.

```
enum Couleurs { BLEU = 1, ROUGE = 3, JAUNE = 1};
```

L'execution de l'instruction `printf("%d - %d - %d", BLEU, ROUGE, JAUNE);` renvoie alors à l'écran :
`1 - 3 - 1`.

4.5 Unions

Une **union** désigne un ensemble de variables de types différents succeptibles d'occuper alternativement une même plage mémoire. Cela permet de définir des objets qui peut être d'un type au choix parmi un ensemble fini de types. Si les membres d'une union sont de tailles différentes, la place réservée en mémoire pour représenter l'union est la taille un membre le plus grand. Modifier l'un des champ "écrase" donc tous les champs de l'union. Typiquement, une union s'utilise lorsqu'un enregistrement peut occuper plusieurs fonctions bien distinctes et que chaque fonction ne requiert pas l'utilisation de tous les champs.

```
union Nom_union {
  type1 Nom_champs-1 ;
  type2 Nom_champs-2 ;
  ...
  typeN Nom_champs-N ;
};
```

On accède au champs d'une union de la même manière que pour une structure.

Dans l'exemple suivant, on crée une variable `Note_Exam` de type `union Note` qui peut être soit un entier, soit une lettre (selon le système de notation choisi) :

```
union Note {
 char lettre;
 int nombre;
};

union note Note_Exam;
```

Par exemple, pour rentrer une note en lettre, on utilise l'instruction `Note_Exam.lettre='A';` pour rentrer une note en chiffre, on utilise l'instruction `Note_Exam.nombre=20;`.

Les unions peuvent être utiles lorsqu'on a besoin de voir un objet sous des types différents (mais en général de même taille). Voici par exemple ce que l'on peut faire pour représenter les nombres complexes :

```
/*Structure représentant les nombres complexes en notation cartésienne*/
struct Cart{
  float x,y;

};
/*Structure représentant les nombres complexes en notation exponentielle*/
struct Exp{
  float rho,theta;
};

/*Structure pour les nombres complexe en notation exp. ou cartésienne*/
struct complexe{
  char test;
        /* test='c' -> Notation cartésienne, test='e' -> Notation exponentielle*/
  union {
    struct Cart cart;
    struct Exp exp;
  } Not;
};
```

Ainsi, lorsqu'on initialise une variable de type `struct complexe` on peut indifféremment l 'initialiser sous forme cartésienne ou sous forme exponentielle. Pour entrer les nombres $z_1 = 2 + 2i$ et $z2 = e^{0,3i}$, on écrira :

```
struct complexe z1 = {'c', {1.,2.}};
struct complexe z2 = {'e',{1., 0.3}};
```

Par exemple, pour afficher une variable de type `struct complexe`, on effectuera un `switch` sur le caractère `test` :

```
struct complexe z;
...

switch(z.test){
  case 'c':
    printf("z= %.2f+i%.2f\n",z.Not.cart.x,z.Not.cart.y);
    break;
  case 'e':
    printf("z= %.2f e^(i%.2f)\n",z.Not.exp.rho,z.Not.exp.theta);
    break;
  }
```

4.6 Indentification des types composés avec typedef

Pour alléger les programmes, on peut affecter un nouvel identificateur à un type composé (structures, tableaux ou chaines de caractère) à l'aide de `typedef`, dont la syntaxe est :

`typedef` *type_Structure synonyme* ;

Cela peut aider grandement à simplifier les déclaration de structures compliquées, imbriquant tableaux, structures, unions...

Par exemple

```
typedef struct Point[3] TRIANGLE;
```

déclare un type dont les instantes seront des tableaux de 3 structures de type `struct Point`.

L'instruction `TRIANGLE T1;` crée un tableau de taille 3, dont les élements sont des instances de `struct Point`.

D'autre part, on peut également combiner la déclaration de structure et l'utilisation de `typedef` :

```
typedef struct {
  char   nom;
  float  x,y;
} POINT;

POINT X;
```

est équivalent à :

```
struct Point {
  char   nom;
  float  x,y;
};

struct Point X;
```

4.7 A propos des valeurs des identificateurs de tableaux et des chaines de caractères

On a vu, pour les tableaux et les chaines de caractères, qu'aucune opération "globale" n'etait autorisée. En particulier, un tableau ne peut pas figurer à gauche d'un opérateur d'affectation : on ne peut pas écrire "`Tab1 = Tab2;`" si `Tab1` et `Tab2` sont deux tableaux ou deux chaines de caractères.

La valeur de l'expression `Tab1` est en fait l'adresse mémoire de la portion de mémoire contenant le premier élément du tableau. On dit que `Tab1` **pointe vers** `Tab1[0]`. Il est donc logique, dans un sens, de ne pas vouloir modifier la valeur de `Tab1`, car cela reviendrai à perdre l'adresse où ont été stockés les valeurs des éléments du tableau. Un identificateur de tableau est ce qu'on appelle un **pointeur constant**.

Cela nous amène donc naturellement au chapitre suivant...

Chapitre 5

Pointeurs

5.1 Adresses et pointeurs

Lorsqu'on manipule une variable dans un programme, sa valeur est stockée quelque part dans la mémoire centrale. Celle-ci est constituée d'octets, qui sont identifiées de manière unique par un numéro appellé **adresse**. Lorsqu'on déclare une variable, le compilateur lui associe un ou plusieurs octets consécutifs où sera stockée sa valeur. La taille de cette portion mémoire associé étant déterminée par le type de la variable. Pour retrouver la valeur d'une variable, il suffit en fait de connaitre le type de la variable ainsi que l'adresse du premier des octets où cette variable est stockée.

Bien qu'il soit *a priori* plus intuitif de désigner des variables par leur identificateurs plutôt que par leurs adresses, laissant le soin au compilateur de faire le lien entre variable et adresse-mémoire, il est de temps en temps très pratique et parfois nécessaire de manipuler directement les adresses...

Tout objet pouvant être placé à gauche d'un opérateur d'affectation (on appelle ces objets des *Lvalues*) possède :

- une adresse, qui indique l'octet à partir duquel les valeurs de cet objet seront stockées,
- une valeur, qui est stockée dans la mémoire à partir de l'octet indiqué par son adresse.

On accède à l'adresse d'une variable via l'opérateur d'adresse `&` (voir Chapitre 2, Paragraphe 2.5.8).

Voici un exemple de portion de code :

```
int n,m;
char a;
n=3;
m=n;
a='a';
printf("adresse de n: %lu\nadresse de m: %lu\nadresse de a: %lu\n",&n,&m,&a);
```

Une éxecution de cette portion de programme renvoie par exemple (Notez que chaque éxecution donne lieu à une allocation d'adresses différentes) :

```
adresse de n: 3214431708
adresse de m: 3214431704
adresse de a: 3214431715
```

L'affectation des adresses aux variables est schématisée Tableau 5.1.

Deux variables ayant des identificateurs différents ont deux adresses différentes. Les variables `n` et `m` sont de type `int` donc codées sur 4 octets, tandis que la variable `a` qui est de type `char` est stocké sur un seul octet.

Objet	Valeur	Adresse	Numéros d'octet
`m`	3	`&m=3214431704`	3214431704 à 3214431707
`n`	3	`&m=3214431708`	3214431708 à 3214431711
			3214431712
			3214431713
`a`	a	`&a=3214431715`	3214431715

TAB. 5.1 – Allocation-mémoire

L'instruction `m=n;` n'opère pas sur les adresses mais sur les valeurs : Les cases mémoires dévolues à `m` sont mise à la même valeur que les cases mémoires dévolues à `n`.

Notez que l'adresse d'un objet étant un numéro d'octet, c'est donc un entier. Son format dépend de l'architecture c'est le format interne (16,32 ou 64 bits). D'autre part, si on peut accéder à l'adresse d'une variable `n` avec l'opérateur d'adresse `&`, la valeur de `&i` est une constante : **on ne peut pas faire figurer `&i` à gauche d'un opérateur d'affectation ni la modifier**.

Cependant, comme il est souvent plus pratique de travailler avec les adresses, plutôt qu'avec les valeurs et les objets. Pour pouvoir manipuler les adresses, on doit recourir à une autre classe d'objet : les pointeurs.

5.1.1 Définition de variable de type pointeur

Un **pointeur** est un objet (qui peut être une Lvalue) dont la valeur est égale à l'adresse d'un autre objet. Cela revient en quelque sorte à déclare un identificateur d'adresse. On déclare un pointeur en utilisant l'instruction :

type `*`*Nom_pointeur* `;`

L'indentificateur *Nom_pointeur* est associé à un entier (en général de type `unsigned long int`) dont la valeur pourra être l'adresse d'une variable de type *type* (qui peut éventuellement être un type structuré). On dit que le pointeur *Nom_pointeur* pointe vers un objet de type *type*.

Ainsi, par exemple, les instructions suivantes :

```
int n, *p1;
char *p2;
struct personne *p3;
```

définissent successivement :

- une variable `n` de type `int` et un pointeur `p1` vers une variable de type `int`,
- un pointeur `p2` vers une variable de type `char`,
- un pointeur `p3` vers une variable de type `struct personne`

Par défaut, lorsqu'on définit un pointeur sans l'initialiser, il ne pointe sur rien : la valeur d'un pointeur est alors égale à une constante symbolique noté `NULL` définie dans l'en-tête `<stdio.h>` qui vaut `'\0'` en général. Le test *Nom_pointeur*`==NULL` permet donc de savoir si un pointeur pointe vers quelque chose ou pas.

Même si la valeur d'un pointeur est toujours un entier, **le type d'un pointeur dépend de l'objet vers lequel il pointe**. Cette distinction est indispensable à l'interprétation de la valeur d'un pointeur car il renseigne sur la taille de l'objet pointé. S'il pointe sur un caractère, sa valeur est celle de l'adresse de l'octet ou est stocké le caractère, ou s'il pointe sur un entier, sa valeur est celle de l'adresse du premier des 4 octets où est stocké l'entier.

Par exemple, lorsque le code suivant est executé :

```
int *p, n;
p=&n;
n=3;
```

On se trouve dans la configuration suivante (les emplacements mémoires attribués change à chaque exécution) :

Objet	Valeur	Adresse	Numéros d'octet
`n`	3	`&m=3218448700`	3218448700 à 3218448703
`p`	3218448700	`&p=3218448704`	3218448704 à 3218448707

Notez bien que l'atout principal des pointeurs réside dans le fait suivant : **contrairement à une adresse, la valeur d'un pointeur est initialisable et modifiable**.

Les pointeurs peuvent également être des éléments d'un tableau, ou un champs d'une structure. Par exemple, l'instruction :

```
struct personne *liste[50];
```

définit une liste de 50 pointeurs vers des variables de type `struct personne`, et l'instruction :

```
struct livre {
  char titre[30];
  struct personne *auteur;
  }
```

déclare une structure `livre`, dont le champs `auteur` est la donnée d'un pointeur sur une variable de type `struct personne`.

L'utilisation des pointeurs peut également permettre de déclarer un type de structure de manière récursive. Par exemple :

```
struct personne {
  char nom[30];
  struct personne *mere;
  struct personne *pere;
  }
```

Les pointeurs sur des structures seront détaillés au Paragraphe 5.4. Les pointeurs peuvent également être passés comme arguments ou comme type de retour de fonctions, permettant de définir des fonctions qui modifient leurs paramètres. Nous détailleront cela au Paragraphe 5.5.

5.1.2 Opérateur d'indirection

Pour accéder à la valeur d'une variable pointée par un pointeur, on utilise l'**opérateur unaire d'indirection** : `*`. Si par exemple, un pointeur `p` pointe vers une variable `n` de type `double`, on peut accéder à partir `p` à la valeur courante de `n` en utilisant `*p`. Par exemple, le morceau de code :

```
double *p;
double n;
n=3;
p=&n;
printf("*p= %d\n",*p);
```

imprimera à l'écran : `*p=3`.

En fait les objets `*p` et `n` sont identiques : ils ont même valeur et même adresse. Modifier la valeur de `*p` revient donc à modifier la valeur de `n`. Les relations entre `n` et `p` sont décrites à la Figure 5.1.

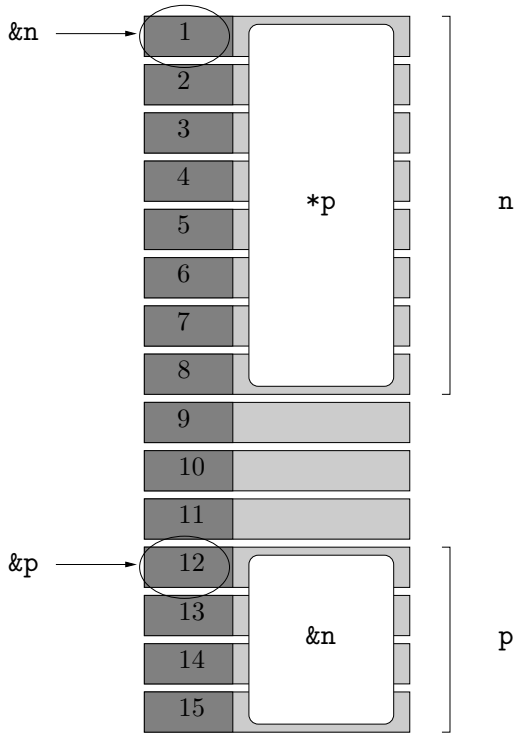


FIG. 5.1 – Relations entre une variable `double n` et un pointeurs `double *p=&n`.

On peut manipuler dans un même programme à la fois un pointeur `p` et son indirection `*p` mais il faut faire attention car ils réagissent différemment.

Si par exemple, après éxecution du code suivant :

```
int *p1, *p2;
int n = 3;
int  m = 100;
p1=&n;
p2=&m;
```

On se retrouve dans la configuration mémoire suivante :

Objet	Valeur	Adresse	Numéros d'octet
`n`	3	`&n=3218448700`	3218448700 à 3218448703
`m`	100	`&m=3218448704`	3218448704 à 3218448707
`p1`	3218448700	`&p1=3218448710`	3218448710 à 3218448713
`p2`	3218448704	`&p2=3218448714`	3218448714 à 3218448717

L'instruction `*p1=*p2;` modifierait la configuration mémoire comme suit :

Objet	Valeur	Adresse	Numéros d'octet
`n`	**100**	`&n=3218448700`	3218448700 à 3218448703
`m`	100	`&m=3218448704`	3218448704 à 3218448707
`p1`	3218448700	`&p1=3218448710`	3218448710 à 3218448713
`p2`	3218448704	`&p2=3218448714`	3218448714 à 3218448717

Tandis que l'instruction `p1=p2;` modifierait la configuration mémoire comme suit :

Objet	Valeur	Adresse	Numéros d'octet
`n`	3	`&n=3218448700`	3218448700 à 3218448703
`m`	4	`&m=3218448704`	3218448704 à 3218448707
`p1`	3218448700	`&p1=3218448710`	3218448710 à 3218448713
`p2`	**3218448700**	`&p2=3218448714`	3218448714 à 3218448717

5.1.3 Arithmétique des pointeurs

Les valeurs des pointeurs étant des entiers, on peut leur appliquer certains opérations. Les seules opérations valides pour les pointeurs sont :

- l'**addition d'un entier à un pointeur**, qui renvoie un pointeur de même type,
- la **soustraction d'un entier à un pointeur**, qui renvoie un pointeur de même type,
- la **différence entre deux pointeurs de même type** qui renvoie un entier.

On ne peut pas additionner des pointeurs.

Si `p` est un pointeur sur un objet de type *`type`* et `i` un entier, l'expression `p+i` désigne un pointeur sur un objet de type *`type`* et dont la valeur est celle de `p` incrémentée de `i*sizeof(`*`type`*`)`. C'est le même principe pour la soustraction et les opérateurs d'incrémentation `++` et `--`.

Si on déclare par exemple un pointeur `p1` sur un `int`, et que `p1` a la valeur 3218448700, alors `p1+10` a pour valeur $3218448700 + 4 \times 10 = 3218448740$ puisqu'un `int` est codé sur 4 octets.

Si par contre le pointeur `p1` pointait sur un objet de type `double`, alors `p1+10` aurait eu pour valeur $3218448700 + 8 \times 10 = 3218448780$puisqu'un `double` est codé sur 8 octets.

D'autre part, si `p1` et `p2` sont deux pointeurs sur des objets de type *`type`*, l'expression `p1-p2` désigne l'entier égal à $\frac{\texttt{p-q}}{\texttt{sizeof(}\textit{type}\texttt{)}}$.

On peut également appliquer à un pointeur l'**opérateur d'indexation** : `[ ]`. Il est lié à l'opérateur d'indirection via la formule `p[i] = *(p+i)`.

Ces opérations peuvent être utiles pour parcourir des tableaux, cela sera détaillé Paragraphe 5.3.

5.2 Allocation dynamique

Les déclarations de variables en C et dans beaucoup d'autres langages ont une limitation très importante : la taille des variables doit être connue à la compilation. On parle dans ce cas d'**allocation automatique** : le programme se charge de tout : il alloue de la mémoire à la déclaration de la variable et à la fin du bloc supprime automatiquement la variable.

Cela pose problème quand on ne sait le nombre de données qu'on doit traiter et leurs tailles qu'au moment de l'exécution. Pour résoudre ce problème, et pouvoir décider durant l'exécution d'un programme du nombre de variables à créer, il faudra nécessairement passer par de l'**allocation dynamique** de mémoire. Dans ce cas, le programmeur dispose de fonctions qui permettent de demander

au système une zone mémoire d'une certaine taille, qu'il pourra utiliser comme il le souhaite. En C, ces fonctions sont disponibles dans l'en-tête `<stdlib.h>`.

Si l'on utilise beaucoup plus souvent l'allocation automatique (où c'est le programme qui ce charge de tout), l'allocation dynamique s'avère parfois être l'unique solution. Voici un exemple où l'on doit utiliser l'allocation dynamique : vous souhaitez que l'utilisateur entre une série de chiffres, qui soit stockée dans la mémoire mais ne savez pas combien d'élément comportera cette série. Vous pourriez préparer un certain nombre de place n, et occuper uniquement celles qui vous servent mais cela vous limiterait à n entrées, et utiliserait de toute manière la place mémoire pour n données. L'allocation dynamique de mémoire vous permet de redéfinir la taille du tableau en cours d'exécution, d'ajouter ou supprimer des entrées, sans limites ou presque.

5.2.1 Allouer de la mémoire : les fonctions `malloc` et `calloc`

Lorsqu'on définit un pointeur sans l'initialiser, il pointe sur la constante symbolique `NULL`. L'initialisation d'un pointeur peut s'effectuer

- par une affectation (du type `pointeur=&variable`), si on souhaite faire pointer le pointeur sur une variable existant déjà dans le programme,
- ou en reservant pour `*pointeur` un espace mémoire de taille adéquate, puis en affectant directement une valeur à `*pointeur` (sans avoir besoin d'indentificateur de variable).

L'allocation dynamique est l'opération consistant à reserver une place mémoire pour stocker l'objet pointé par `pointeur`. Cette opération s'effectue à l'aide de la fonction `malloc` dont le prototype est le suivant :

```
void* malloc (size_t size) ;
```

Cette fonction renvoie un pointeur pointant vers un objet de taille *size*) octets. Le type `void*` est un passe-partout, il permet de remplacer n'importe quel autre type. Pour initialiser les pointeurs vers des objets de type différents, on remplace `void*` par le type souhaité ou on effectue un transtypage (voir Chapitre 2, Paragraphe 2.5.2) au moment de l'initialisation.

Pour allouer une plage mémoire à un tableau de *N* éléments de type *type* (soit un espace mémoire de taille *N*`*sizeof(`*type*`)`) qui sera pointé par *Nom_pointeur*, on appelle la fonction `malloc` ainsi :

```
type *Nom_pointeur = malloc(N*sizeof(type)) ;
```

Notez que l'affectation effectue direment le transtypage du type `void*` vers *type*`*`.

Si le pointeur *Nom_pointeur* a déja été déclaré précédemment comme pointant sur des éléments de type *type*, la synthaxe est un peu différente, puisqu'un transtypage doit être effectué :

```
type *Nom_pointeur ;
Nom_pointeur =(type*) malloc(N*sizeof(type)) ;
```

Remarques sur l'utilisation de `sizeof` :

La fonction `sizeof` peut prendre en argument un type ou l'identifiacteur d'une variable :

- `sizeof(`*type*`)` retourne le nombre d'octets nécessaire à la mise en mémoire d'un élément de type *type*.
- L'instruction `sizeof(`*Indentificateur*`) ;` retourne le nombre d'octets nécessaire à la mise en mémoire d'un élément de même type que *Identificateur*.

Il faut donc faire attention avec les tableaux. Si `Tab` est un tableau de `N` éléments de type *type*, l'instruction `sizeof(Tab);` renvoie `N*` `sizeof(`*type*`)` et pas `N`. Pour obtenir le nombre d'éléments du tableau `Tab`, on utilisera `sizeof(Tab)/sizeof(`*type*`)` ou encore `sizeof(Tab)/sizeof(Tab[0])`.

La fonction `calloc`, également disponible dans le fichier d'en-tête `<stdlib.h>` permet d'allouer dynamiquement de la place mémoire mais initialise également toutes les valeurs de cet espace mémoire à zéro. Son prototype est le suivant :

```
void* calloc(size_t Nb_Elements, size_t Taille_Element) ;
```

Elle renvoie un pointeur sur une plage mémoire de taille *Nb_Elements*Taille_Element*.

Pour allouer une plage mémoire à un tableau de *N* éléments de type *type* (soit un espace mémoire de taille *N*sizeof(type)*) qui sera pointé par *Nom_pointeur* et dont les valeurs sont initialisées à zéro, on appelle la fonction `calloc` ainsi :

```
type *Nom_pointeur = calloc(N,sizeof(type)) ;
```

Si le pointeur *Nom_pointeur* a déja été déclaré précédemment comme pointant sur des éléments de type *type*, comme pour `malloc`, la synthaxe est un peut différente, puisqu'un cast doit être effectué :

```
type *Nom_pointeur ;
Nom_pointeur =(type*) malloc(N,sizeof(type)) ;
```

5.2.2 Reaffecter la mémoire : la fonction `realloc`

La fonction `realloc` sert modifier une taille réservée par un appel à la fonction `malloc` (généralement pour agrandir bien que ce ne soit pas obligatoire le cas). Et —petit gadget supplémentaire— on peut aussi s'en servir pour une simple réservation (et non une modification) si on lui passe un pointeur `NULL` en premier paramètre (elle fait dans ce cas double emploi avec la fonction `malloc` mais est moins rapide puisqu'il y a un test supplémentaire).

Bien que souvent décriée pour sa lenteur, elle offre une alternative intéressante pour gérer des tableaux de taille variable. Bien sûr il ne faut pas allouer les objets un par un, mais par blocs (doublage, par exemple).

Son prototype est le suivant :

```
void * realloc(void * Nom_pointeur , size_t Nouvelle_taille) ;
```

Voici un bout de code utilisant `malloc` et `realloc` :

```
/*Allocation d'un tableau de 10 int*/

   size_t size = 10;
   int *p = malloc (size * sizeof(int));

/* doublement du stockage, stocké d'abord dans un pointeur temporaire*/

   size*=2;
   int *p_tmp = realloc (p, size * sizeof(int));


   if (p_tmp != NULL) {
      p = p_tmp;
     }
    else {
       /*l'ancien bloc pointé par p est valide, mais il n'a pas été agrandi*/
     }
```

5.2.3 Allocation automatique VS Allocation dynamique

Ce paragraphe illustre les différences entre les deux processus d'allocation mémoire.

	Allocation Automatique		Allocation Dynamique
$\ell 1$	`int n = 3 ;`	VERSUS	`int n = 3 ;`
$\ell 2$	`int *p ;`		`int *p=malloc(sizeof(int)) ;`

Après la ligne $\ell 2$, supposons qu'on se trouve dans la configuration suivante dans les deux cas :

Allocation Automatique			Allocation Dynamique		
Objet	Valeur	Adresse	Objet	Valeur	Adresse
`n`	3	`3218448700`	`n`	3	`3218448700`
`p`	? ? ?	`3218448704`	`p`	`3218448708`	`3218448704`
`*p`	? ? ?	? ? ?	`*p`	? ? ?	`3218448708`

A ce stade-là, la manipulation de `*p` n'a aucun sens dans les deux cas. Cependant, coté allocation automatique, l'appel à la valuer `*p` peut génèrer une "violation mémoire" : la valeur de `p` peut faire référence à une case mémoire ne faisant pas partie des segments du programme, cela est détecté par lesystème d'exploitation qui n'aime pas ça du tout, entrainant l'interruption du programme et le doux message d'erreur `segmentation fault`.

Coté allocation dynamique, l'allocation mémoire a crée une place d'octets nécessaires pour skocker un entier sur laquelle pointe `p` (dans le segment de tas). A ce moment là, `*p` vaut ce que la mémoire vaut à cet instant sur ces cases mémoires-là, converti en `int`.

Pour illustrer cela, voyons comment deux instructions différentes modifient l'état de la mémoire dans les deux cas :

→ Instruction `*p=n;`

Allocation Automatique			Allocation Dynamique		
Objet	Valeur	Adresse	Objet	Valeur	Adresse
`n`	3	`3218448700`	`n`	3	`3218448700`
`p`	? ? ?	`3218448704`	`p`	`3218448708`	`3218448704`
`*p`	? ? ?	? ? ?	`*p`	3	`3218448708`

Si on a utilisé l'allocation automatique, cette instruction génère une violation mémoire et une erreur `segmentation fault`. Dans le cas de l'utilisation de l'allocation dynamique, cela a parfaitement un sens et remplace la valeur de `*p` par la valeur de la variable `n`.

L'avantage d'utiliser ici l'allocation dynamique réside dans le fait que `*p` et `n` n'ont d'égal que leur valeurs : modifier `*p` n'affecte pas `n` et vice-versa.

→ Instruction `p=&n;`

Allocation Automatique			Allocation Dynamique		
Objet	Valeur	Adresse	Objet	Valeur	Adresse
`n`	3	`3218448700`	`n`	3	`3218448700`
`p`	`3218448700`	`3218448704`	`p`	`3218448700`	`3218448704`
`*p`	3	`3218448700`	`---`	? ? ?	`3218448708`

Dans les deux cas, cette instruction initalise le pointeur `p` à la valeur de l'adresse `&n` : Manipuler `*p` revient à manipuler `n` : ces deux expressions ont même valeur et même adresse et toute modification de l'une entraine la même modification sur l'autre. Cependant, si on utilisé l'allocation dynamique, la plage-mémoire stockant `*p` avant l'instruction n'est pas libérée... Le programme ne pourra plus utiliser cette place : c'est ce qu'on appelle une **fuite mémoire**.

En fait, lorsqu'on a plus besoin des données présentes dans une plage-mémoire allouée dynamiquement à un pointeur, il faut la libérer...

5.2.4 Libérer la mémoire : la fonction `free`

Pour libérer l'espace mémoire alloué à un pointeur *Nom_Pointeur*, on utilise la fonction `free`, dont le prototype est le suivant :

`void free(void *`*Nom_Pointeur*`).`

Comme pour la fonction `malloc` le type générique `void *` indique que cette fonction peut prendre en argument un pointeur de n'importe quel type. Cette fonction doit être utilisée lorsqu'on n'a plus besoin d'utiliser les données vers lesquelles pointent le pointeur *Nom_Pointeur* et permet à les cases mémoires contenant *Nom_Pointeur* de pouvoir être de nouveau utilisées en cas d'autre affectation.

Le pointeur *Nom_Pointeur* existe toujours et peut être reutilisé dans le programme. Par contre la valeur de *Nom_Pointeur* n'est pas conservée.

Le programme suivant illustre les possibilités et les conséquences possibles de l'utilisation des identifiants de pointeurs ayant étés libéré :

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

3        int main(int argc,char* argv[]) {
4          int n=3;
5          int *p=malloc(sizeof(int));
6          int *q=malloc(sizeof(int));
7          *p=12;
8          *q=40;
9          free(p);
10         free(q);
11         p=&n;
12         int *p1=malloc(sizeof(int));
13         int *q1=malloc(sizeof(int));
14         *p1=5;
15         *q=3;
16         return EXIT_SUCCES;
17       }
```

Après la ligne 8, la mémoire est dans l'état suivant (les adresses sont notés en héxadécimal) :

Objet	Valeur	Adresse
p	0x804a008	0xbfb52aec
q	0x804a018	0xbfb52ae8
*p	12	0x804a008
*q	40	0x804a018

Après la ligne 10, les valeurs de *p et *q ne sont plus garanties mais les pointeurs conservent leurs valeurs (adresses des premières cases mémoires des anciens *p et *q) :

Objet	Valeur	Adresse
p	0x804a008	0xbfb52aec
q	0x804a018	0xbfb52ae8
*p	0	0x804a008
*q	134520832	0x804a018

A la ligne 11, le pointeur libéré p est initialisé comme pointant vers une autre variable, ici n :

Objet	Valeur	Adresse
p	0xbfb52af0	0xbfb52aec
*p	3	0xbfb52af0
n	3	0xbfb52af0

Ligne 12 et 13, deux nouveaux pointeurs p1 et q1 sont crées et la mémoire est allouée dynamiquement ; les même plages mémoires précédemment utilisées pour p et q sont utilisées : Remarquez que

Objet	Valeur	Adresse
n=*p	3	0xbfb52af0
p	0xbfb52af0	0xbfb52aec
q	0x804a018	0xbfb52ae8
p1	0x804a018	0xbfb52ae4
q1	0x804a008	0xbfb52ae0
*q	134520832	0x804a018
*p1	134520832	0x804a018
*q1	0	0x804a008

p1 et q pointent toujours vers la même case mémoire : les modifications de *p1 affecterons *q et vice versa.

Ainsi, la ligne 14 transforme à la fois *p1 et *q :

Objet	Valeur	Adresse
q	0x804a018	0xbfb52ae8
p1	0x804a018	0xbfb52ae4
*q	5	0x804a018
*p1	5	0x804a018

La ligne 15 modifie également *p1 et *q en mettant leurs valeurs à 3.

5.2.5 Quelques bons reflexes

– **Initialiser les pointeurs** : Lorsqu'on déclare une variable ou qu'on alloue une zone mémoire, on ne dispose d'aucune garantie sur le contenu de cette zone ou de cette variable. Initialiser systématiquement les variables dès leur déclaration, en particulier lorsqu'il s'agit de pointeurs, fait partie des bons réflexes à prendre et pour la plupart des applications ne dégrade pas le temps d'exécution de manière significative. Les erreurs provenant d'une non-initialisation de pointeurs peuvent être extrèmement difficile à localiser à l'intérieur d'un programme complexe.

- **Un seul pointeur par zone mémoire** : En règle générale, il faut éviter (autant que faire se peut...) que plusieurs variables pointent la même zone mémoire allouée.
- **Eviter les fuites de mémoire** : La perte du pointeur associé à un secteur mémoire rend impossible la libération du secteur à l'aide de free. On qualifie cette situation de fuite mémoire, car des secteurs demeurent réservés sans avoir été désalloués. Cette situation perdure jusqu'à la fin du programme principal.

5.3 Pointeurs, tableaux et chaines de caractères

L'usage des pointeurs en C est essentiellement tourné vers la manipulation des tableaux et des chaines de caractères.

5.3.1 Tableaux unidimensionnels

Un tableau en C est un **pointeur constant** (c'est-à-dire non modifiable). Dans la déclaration :

```
int Tab[10];
```

`Tab` est un pointeur constant vers une plage-mémoire de taille `10*sizeof(int)`, dont la valeur est l'adresse du premier élément du tableau : `Tab` a pour valeur `&Tab[0]`. On peut donc se servir d'un pointeur pour parcourir le tableau comme dans le code suivant :

```
int Tab[5]={1,2,3,6,4};
int *p;
p=Tab;
for (int i = 0; i<N;i++) {
  printf("%d \n",*p);
  p++;
  }
```

qui est équivalent à :

```
int Tab[5]={1,2,3,6,4};
for (int i = 0; i<N;i++) {
  printf("%d \n",Tab[i]);
  }
```

qui est encore équivalent à :

```
int Tab[5]={1,2,3,6,4};
int *p;
p=Tab;
for (int i = 0; i<N;i++) {
  printf("%d \n",p[i]);
  }
```

Les deux différences principales entre tableaux est pointeurs sont les suivantes :

- un pointeur doit toujours être initialisé, soit par un appel aux fonctions malloc— ou `calloc`, soit par affectation d'une expression d'adresse (du type `p=&i;`),
- un tableau ne peut pas être une Lvalue (ne peut jamais figurer à gauche d'un opérateur d'affectation). En particulier, il ne supporte pas l'arithmétique des pointeurs (on ne peut pas écrire `Tab++;`).

D'autre part, la manipulation de tableaux plutôt que de pointeurs a deux inconvenients majeurs, du fait qu'ils sont des pointeurs constants :

– On ne peut pas créer des tableaux dont la taille est une variable du programme,

– on ne peut pas créer de tableaux bidimensionnels dont les lignes n'ont pas toutes la même taille.

Ces opérations deviennent possibles lorsque la mémoire est allouée dynamiquement, avec `malloc` ou `calloc`, et donc que l'on a à faire à de vrais pointeurs.

Ainsi, pour définir un tableau à `n` éléments, où `n` est une variable du programme, on utilise :

```
int n;
int *Tab;
Tab=(int *)malloc(n*sizeof(int));
```

Le pointeur `Tab` s'utilisera ensuite comme un tableau de `n` éléments de type `int`. En particulier, son i^e élément est donnée par l'expression `Tab[i]`.

5.3.2 Tableaux et pointeurs multidimensionnels

Un tableau multidimensionnel est un tableau de tableaux. Il s'agit donc d'un pointeur constant qui pointe vers un pointeur constant. Par exemple, dans la déclaration de tableau suivante :

```
int Tab[N][M];
```

L'élément `Tab` est un pointeur vers un objet qui est lui-même de type pointeur sur des entiers. `Tab` a une valeur constante, égale à l'adresse du premier élément du tableau : `&Tab[0][0]`. Les pointeurs `Tab[i]`, pour `i` variant de `0` à `N-1` sont des pointeurs constants pointant respectivement sur le premier élément des lignes `i` : `Tab[i]` a donc pour valeur `&Tab[i][0]`.

Par exemple, supposons que l'on souhaite représenter à l'aide d'un tableau la matrice à coefficients entiers suivante :

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

On initialise par exemple le tableau ainsi :

```
int Tab[2][3]={{1,2,3},{4,5,6}};
```

La mémoire sera par exemple organisée ainsi :

Objet	Valeur	Adresse
`Tab`	4687351800	3218448700
`Tab[0]`	4687351800	3218448704
`Tab[1]`	4687351812	3218448708
`Tab[0][0]`	1	4687351800
`Tab[0][1]`	2	4687351804
`Tab[0][2]`	3	4687351808
`Tab[1][0]`	4	4687351812
`Tab[1][1]`	5	4687351816
`Tab[1][2]`	6	4687351820

On peut également définir des tableaux tridimensionnels et même à plus de 3 dimensions (en utilisant une déclaration du type *`type`* `tab[N1][N2]...[Nn];`). Par exemple, si on souhaite décrire la famille de matrices suivante :

$$A_0 = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}, \quad A_2 = \begin{bmatrix} 3 & 2 & 1 \\ 6 & 5 & 4 \end{bmatrix}, \quad A_3 = \begin{bmatrix} 1 & 1 & 1 \\ 4 & 1 & 1 \end{bmatrix}, \quad A_4 = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}.$$

à l'aide d'un tableau multidimensionnel, on peut la déclarer ainsi :

```
int A[4][2][3];
A[0]={{1,2,3},{4,5,6}};
A[1]={{3,2,1},{6,5,4}};
A[2]={{1,1,1},{4,1,1}};
A[3]={{0,0,0},{1,0,0}};
```

Les tableaux multidimensionnels sont cependant contraignants : tous les sous-tableaux doivent avoir la même taille. On ne pourrait par exemple pas utiliser un tableau pour stocker proprement la famille de matrices suivante :

$$A_0 = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}, \quad A_2 = \begin{bmatrix} 2 & 1 \\ 5 & 4 \end{bmatrix}, \quad A_3 = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}, \quad A_4 = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}.$$

Pour stocker ce type de données, on peut en revanche utiliser des **pointeurs multidimensionnels**. Ces pointeurs possèdent plusieurs avantages par rapport aux taleau multidimensionnels. Outre le fait que l'on puisse stocker des éléments de taille différentes, on peut également allouer dynamiquement la mémoire dédiée à chaque "sous-pointeur".

Pour déclarer un pointeur multidimensionnel, on procède comme pour les tableaux. Définir un pointeur bidimensionnel *Nom_pointeur* sur des objets de type *type*, c'est définir un pointeur sur des objets de type *type* *. On déclarera donc un pointeur didimensionnel via l'instruction :

*type ** Nom_pointeur* ;

De même, on déclarera un pointeur tridimensionnel ainsi :

*type *** Nom_pointeur* ;

Par exemple, si on souhaite stocker et initialiser, à l'aide d'un pointeur de pointeurs, les premières lignes triangle de Pascal, qui contient les coefficiants binomiaux :

$$\begin{array}{ccccccccc} & & & & 1 & & & & \\ & & & 1 & & 1 & & & \\ & & 1 & & 2 & & 1 & & \\ & 1 & & 3 & & 3 & & 1 & \\ 1 & & 4 & & 6 & & 4 & & 1 \\ & 1 & & 5 & & 10 & & 5 & & 1 \\ 1 & & 6 & & 15 & & 20 & & 15 & & 6 & & 1 \\ & & & & \dots & & & & \end{array}$$

où chaque coefficient est obtenu en faisant la somme des deux qui sont au dessus de lui.

Pour déclarer et initialiser les 10 premières lignes du triangle, on procède comme suit :

```
int i,j;
int n=10;
int ** Pascal;
```

```
/*Allocation de la mémoire*/
Pascal =(int **)malloc(n*sizeof(int *));
for (i=0;i<n;i++){
    Pascal[i]=(int *) malloc((i+1)*sizeof(int));
   }
/*Initialisation des valeurs*/
Pascal[0][0]=1;
for (i=1;i<n;i++){
   Pascal[i][0]=1;
   for(j=1;j< i;j++){
      Pascal[i][j]=Pascal[i-1][j-1]+Pascal[i-1][j];
      }
   Pascal[i][i]=1;
  }
```

5.3.3 Pointeurs et chaines de caractères

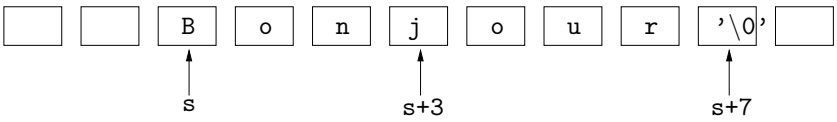
Les chaines de caractères sont des tableaux à une dimension d'objets de type `char` qui se terminent par le caractère nul (`'\0'`). On peut manipuler toute chaine de caractère à l'aide d'un pointeur vers des objets de type char. A une chaine définie par :

```
char * s;
```

on pourra par exemple affecter la valeur suivante :

```
s="Bonjour";
```

et on pourra faire subir à `s` toute opération valide sur les pointeurs, comme par exemple `s++`. La quantité `s+3` désignera la chaine de caractères `jour` et `s+7` la chaine de caractères vide.



5.4 Pointeurs et structures

5.4.1 Pointeur sur une structure

Contrairement aux tableaux, les objets de type structure sont des Lvalues (qui peuvent être à gauche d'un opérateur d'affectation). Ils possèdent une adresse, correspondant à l'adresse du premier élément du premier champs (ou membre) de la structure.

Si une structure a été définie comme suit :

```
struct Structure {
type1 champs-1 ;
type2 champs-2 ;
...
typeN champs-N ;
};
```

et que `p` est un pointeur sur un élément de type `struct Structure` (déclaré par l'instruction `struct Structure *p;`), on peut accéder au champs `champs-i` de la structure pointée par `p` via l'expression :

```
p->champs-i
```

L'opérateur `->` est l'opérateur de **pointeur de membre de structure**. Cette expression est équivalente à l'expression suivante :

```
(*p).champs-i
```

Notez ici que **les parenthèses sont indispensables** car l'opérateur d'indirection a une priorité plus élevée que l'opérateur de membre de structure.

Voici un exemple de programme qui manipule des pointeurs sur des structures.

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

struct Etudiant {
  char nom[30];
  int note;
};

typedef struct Etudiant * Groupe;

int main(int argc,char *argv[]){
  int n,i;
  Groupe examen;

  printf("Entrez le nombre d'étudiants: ");
  scanf("%d",&n);
  examen=(Groupe)malloc(n*sizeof(struct Etudiant));

  for(i=0;i<n;i++){
    printf("\n Saisie du nom de l'élève numéro %d: ",i+1);
    scanf("%s",&examen[i].nom);
    printf("\n Saisie de la note de %s: ",examen[i].nom);
    scanf("%d",&examen[i].note);
  }

  printf("\n Entrez un numéro d'étudiant: ");
  scanf("%d",&i);
  if(i<n+1){
    printf("\n Etudiant numéro %d: %s Note: %d",i,examen[i-1].nom,examen[i-1].note);
    }
  else{
    printf("\n Il n'y a pas d'étudiant numéro %d",i);
    }
  return EXIT_SUCCESS;
}
```

Remarque les expressions `examen[i].nom` et `examen[i].note` auraient pu être remplacées respectivement par `(examen+i)->nom` et `(examen+i)->note`.

5.4.2 Structures auto-référencées et listes chainées

En C, on a souvent besoin de structures dont l'un ou plusieurs des champs sont des pointeurs vers une structure du même type comme dans les deux exemples suivants :

```
struct cellule{
  double valeur;
  struct cellule *cellulesuivante;
  };

struct personne {
   int dateNaissance;
   struct personne *pere;
   struct personne *mere;
  };
```

Utiliser des structures auto-référencées permet en particulier de créer des **listes chainées**. Pour représenter des listes d'éléments de même type, on peut utiliser un tableau ou un pointeur mais cette représentation (dite représentation contigüe) suppose a priori que la taille maximale de la liste soit donnée (pour l'allocation dynamique). Ce problème peut être résolu en utilisant la représentation des données par une liste chainée : Chaque élément de la liste est représenté dans une structure appellée **cellule** qui contient un champs pour représenter la donnée à stocker et un autre champs qui est un pointeur sur l'élément suivant de la liste. La liste est un pointeur sur la première cellule et le pointeur de la dernière cellule pointe sur l'élément `NULL`.

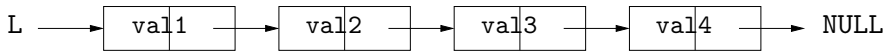


FIG. 5.2 – Schéma d'une liste chainée

Par exemple, pour représenter une liste d'éléments de type `double` àl'aide d'une liste chainée, on définit le modèle de structure `cell` de la maière suivante :

```
struct cell {
   double value;
   struct cell *nextcell;
  };
```

On peut également définir le type `liste` comme étant un pointeur sur une structure `cell` :

```
typedef struct cell *list;
```

Un fois cela effectué, on peut insérer un élément en début de liste assez facilement, en définissant par exemple la fonction `insere` :

```
liste insereDebut(double element, list L) {
   liste Q;
   Q = (liste)malloc(sizeof(struct cell));
   Q->value=element;
   Q->nextcell=L;
   return Q;
  };
```

ou après le n^e élément de la liste :

```
liste insereMilieu(double element, list L, int n) {
   int i=1;
   list P,Q;
   P = (list)malloc(sizeof(struct cell));
   Q = (list)malloc(sizeof(struct cell));
   P=L;
   while(P->nextcell!=NULL && i<n) {
       P=P->nextcell;
       i++;
      }
   Q->nextcell=P->nextcell;
   Q->value=element;
   P->nextcell=Q;
   return L;
 };
```

Pour afficher la liste chainée `L`, on pourra utiliser la fonction suivante :

```
void afficherListe(list L) {
  list P=L;
   while(P!=NULL){
       printf("%f\t",P->value);
       P=P->nextcell;
      }
  }
```

5.5 Pointeurs et fonctions

5.5.1 Fonctions modifiant leur paramètres : Passage d'arguments par adresses

Comme il avait été suggéré au au Chapitre 3, Paragraphe 3.4, si on cherche par exemple à fabriquer une fonction qui modifie les valeurs d'un couple de paramêtres (un flottant double et un entier), le prototype est INADAPTÉ :

```
void modif(double x, int a);
```

car le passage des arguments se fait par valeurs. Les valeurs des arguments sont copiés dnas le segment de pile de la fonction `modif` et ce sont ces copies-là qui pourront être modifiées et pas les valeurs des variables originales (celles de la fonction appellante).

Lorsqu'on veut qu'une fonction puisse modifier ces paramètres, il est nécessaire de passer les adresses des variables et non leurs valeurs. Dans la fonction, on pourra ensuite demander de modifier les valeurs rangées aux adresses mémoire dont les valeurs sont passées en paramètres : les paramètres d'une telle fonction sont de type pointeurs. On utilisera le prototype suivant :

```
void modif(double *adr_x, int *adr_a);
```

En procédant ainsi, les valeurs des adresses sont copiées dans le segment de pile de la fonction `modif`, et par le biais de l'opérateur d'indirection, on pourra alors modifier les valeurs des variables originales.

En utilisant le passage d'argument par adresse, on peut modifier par exemple le programme donné au Paragraphe 3.4 pour qu'il échange bien les valeurs de deux variables :

```
#include<stdio.h>

void flip(int *adr_x, int *adr_y) {
  int t;
  printf("debut flip: x=%d et y=%d\n",  *adr_x, *adr_y);
  t=*adr_x;
  *adr_x=*adr_y;
  *adr_x=t;
  printf("fin flip: x=%d et y=%d\n",  *adr_x, *adr_y);
  return;
 }

int main(int argc,char *argv[]) {
  int x=2;
  int y=5;
  printf("debut main: x=%d et y=%d\n", x,y);
  flip(&x,&y);
  printf("fin main: x=%d et y=%d\n", x,y);
  return EXIT_SUCCESS;
 }
```

affichera bien :

```
debut main: x=2 et y=5
debut flip: a=2 et b=5
fin flip: a=5 et b=2
fin main: x=5 et y=2
```

A l'appel de la fonction `flip` avec les paramètres `&x` et `&y`, voici ce qu'il se passe :

- On stocke dans la pile dévolue à `flip` deux pointeurs sur des entiers `adr_x` et `adr_y` pointant vers les cases mémoires d'adresses respectives `&x` et `&y` (la valeur de `adr_x` est `&x` et la valeur de `adr_y` est `&y`),
- La variable `t` est ensuite initialisée avec la valeur de la variable pointée par `adr_x`, c'est à dire la valeur de `x`,
- On affecte à la variable pointée par `adr_x` (valeur de `x`) la valeur de la variable pointée par `adr_y` (valeur de `y`).
- On affecte à la variable pointée par `adr_y` (valeur de `y`) la valeur de `t`.

5.5.2 Retour sur la pile, le tas, les segments de données et les segments de code...

Le programme décrit ci-dessous permet de visualiser l'organisation mémoire décrite au Paragraphe 3.3 et schématisée à la Figure 3.1.

```
#include <stdio.h>
#include <stdlib.h>
int Max=4;
int Min=-4;
int Aux(int a){
  int b;
  int c=1;
  int * r=malloc(Max*sizeof(int));
  printf("------Segments de pile de Aux()------\n");
  printf("Adresse b: %d\n",&b);
  printf("Adresse c: %d\n",&c);
  printf("Adresse r: %d\n",&r);
  printf("------Segments de tas------\n");
  printf("Cases pointées par r: %d\n",r);
  free(r);
  return a;
}
int main(int argc,char *argv[]){
  int i,j;
  int n=0;
  int m=1;
  int * p=malloc(Max*sizeof(int));
  int * q=malloc(Max*sizeof(int));
  printf("------Segments de code------\n");
  printf("Adresse de la fonction main(): %d\n",(*main));
  printf("Adresse de la fonction Aux() : %d\n",(*Aux));
  printf("------Segments de données------\n");
  printf("Adresse Max: %d\n",&Max);
  printf("Adresse Min: %d\n",&Min);
  printf("------Segments de pile de main()------\n");
  printf("Adresse i: %d\n",&i);
  printf("Adresse j: %d\n",&j);
  printf("Adresse n: %d\n",&n);
  printf("Adresse m: %d\n",&m);
  printf("Adresse p: %d\n",&p);
  printf("Adresse q: %d\n",&q);
  Aux(i);
  printf("Cases pointées par q: %d\n", q);
  printf("Cases pointées par p: %d\n",p);

  free(p);
  free(q);
  return EXIT_SUCCESS;
}
```

Ce programme affiche à l'écran les valeurs suivantes :

```
------Segments de code------
Adresse de la fonction main(): 134513832
Adresse de la fonction Aux(): 134513684
------Segments de données------
Adresse Max: 134519356
Adresse Min: 134519360
------Segments de pile de main()------
Adresse i: -1075017872
Adresse j: -1075017876
Adresse n: -1075017880
Adresse m: -1075017884
Adresse p: -1075017888
Adresse q: -1075017892
------Segments de pile de Aux()------
Adresse b: -1075017932
Adresse c: -1075017936
Adresse r: -1075017940
------Segments de tas------
Cases pointées par r: 134520888
Cases pointées par q: 134520864
Cases pointées par p: 134520840
```

Remarque à propos des pointeurs sur des fonctions et du passage de fonctions en argument :

Il est parfois utile de passer des fonctions comme paramètres d'une autre fonction. C'est ce qui a été fait dans le programme ci-dessus, lors des appels

```
printf("Adresse de la fonction main(): %d\n",(*main));
printf("Adresse de la fonction Aux() : %d\n",(*Aux));
```

Comme pour les variables, on a utilisé un mécanisme de pointeur, car comme toute chose mise en mémoire dans l'ordinateur, une fonction a une adresse de début de code. Un pointeur sur une fonction contiendra l'adresse de la première instruction du code de la fonction.

Un pointeur sur une fonction dont le prototype est

type Nom_fonction (type-1 arg-1, type-2 arg2, ... type-N argN) ;

est de type *type (*) (type-1 arg-1, type-2 arg2, ... type-N argN) ;*.

Par exemple, une fonction `Operateur` prenant par exemple deux entiers ainsi qu'une fonction qui prend elle-même deux entiers en paramètres sera déclarée ainsi :

```
int Operateur( int, int, int (*)(int,int));
```

Pour l'appliquer à deux entiers `a` et `b` et à une fonction `f`, on utilise l'appel `Operateur(a,b,f)`. On n'utilise pas la notation `&f`.

Dans le corps d'une fonction dont un paramètre `f` est de type fonction, pour faire référence à ce paramètre `f`, on utilise la notation `(*f)`. Par exemple, on peut définir la fonction `Operateur` comme suit :

```
int Operateur(int a, int b, int (*f)(int,int)){
  int x=(*f)(a,b);
```

```
  int y=(*f)(b,a);
  if (x<y){
    return y;
   }
  else{
    return x;
   }
}
```

Chapitre 6

Quelques conseils...

Quelques conseils et règles à ne pas oublier pour permettre une programmation sereine....

Des conseils pour facilité la **lisibilité des programmes** :

- **Commentaires** : Les commentaires sont souvent utiles lorsqu'on reprend le code d'un programme après un certain temps... Ils doivent expliquer la logique de traitement et justifier les choix effectués (pourquoi tel algorithme, telle structure de données, ...). et ils permettent de retrouver facilement des parties de code que l'on pourrait vouloir réutiliser...
  On peut également se servir des commentaire pour fabriquer des **cartouche d'en-tête** : Chaque fichier source (`.c`, `.h`) et chaque fonction C doivent être précédés d'un cartouche qui pourra contenir le nom de l'unité ou de la fonction, son rôle et ses points de programmation critiques, pour une fonction la description des arguments et des codes retours, la date l'auteur et le rôle de chaque modification...
  Le cartouche de l'unité permet de faciliter sa compréhension d'un point de vue général. Les cartouches de fonctions permettent de comprendre leur rôle et conditions d'appel sans se plonger dans le code source. Ces entêtes sont indispensables à la gestion de la configuration et à la maintenance.
- **Indentation** : J'indente correctement les blocs... ce n'est pas seulement pour faire joli, mais pour s'y retrouver facilement. D'autre part, l'utilisation de la tabulation est a éviter... Il vaut mieux utiliser des indentations de 2 ou 3 caractères "espace".
- **Nomenclature des identificateurs** : Je me fixe une nomenclature précise ET cohérente pour les indentificateurs des fonctions, ceux des variables et ceux des programmes et je n'en change pas...
  J'utilise si possible des identificateurs dont je peux me souvenir facilement.
  Je ne mets NI d'accents NI d'espaces dans les identificateurs, les noms de mes fichiers, les fonctions....
  Je n'utilise pas non plus des noms commençant pas des chiffres...
- **Nomenclature des constantes symboliques et des macros** : Les noms des constantes symboliques et des macros instructions doivent être écrits en majuscule. Dans un fichier source, cela permet des distinguer rapidement les constantes des variables.
- **Accolades** : Lorsque j'ouvre une accolade... je la ferme directement. On préfèrera aussi les accolades sur des lignes seules, histoire de pouvoir vérifier facilement qu'il ne manque ni parenthèse ouvrante, ni parenthèse fermante...
- **Parenthésage** : Je n'abuse pas des parenthésages inutiles...
- **Lignes** : N'hésitez pas à aller à la ligne... L'idéal étant d'avoir une seule instruction par ligne, une déclaration de variable par ligne.

Des conseils pour facilité la **modification des programmes** :

- **Editeur de texte** : Il vaut mieux privilégier un éditeur de texte minimaliste, du type emacs. Certains éditeurs de textes utilisant des caractères spéciaux masqués, il se peut alors que des surprises vous attendent lors de la compilation...
- **Éviter les constantes littérales** : Les constantes chaîne, numérique doivent être représentées par des symboles. Les constantes littérales sont "interdites" car les constantes littérales dupliquées et réparties dans le code source rendent les modifications plus difficiles : risque d'altération, d'oubli d'une occurrence. Il faut les regrouper sous formes de constantes symboliques (définition du pré-processeurs) en tête de fichier source ou dans des fichiers inclus.
- **Pas d'éléments inutiles ou non initialisé :** Toute fonction définie doit être utilisée et toute variable déclarée doit être initialisée. La valeur par défaut d'une variable dépend de facteurs qui peuvent échapper au programmeur. Cette règle évite les fonctionnements aléatoires de certains programmes. En général, les compilateurs et les outils de vérifications émettent des avertissements en cas de problème.
- **Limiter le nombre des variables externes ou globales :** L'utilisation de variables externes ou globales doit être limité au maximum. Les variables globales peuvent être modifiées par plusieurs unités de compilation, le contrôle des modifications est donc très délicat, les effets de bord sont possibles (modifications non désirées d'une variable ou d'une zone mémoire par du code lié). Un trop grand nombre de variables externes dénote une mauvaise analyse des interfaces entre unités de compilation ou une mauvaise gestion de la portée des variables.
- **Ne pas réinventer la roue** : Penser à utiliser les fonctions et bibliothèques écrites par d'autres programmeurs, principalement dans les bibliothèques standards du langage et des bibliothèques reconnues
- **Pas de code dans les** `.h` : Ne jamais écrire de fonctions ou d'instructions dans les fichiers d'entête .h (excepté les macros instructions). Écrire le corps de fonction dans un fichier `.h` et l'inclure dans plusieurs sources dupliquerait ce code et pourrait provoquer des erreurs due aux symboles définis plusieurs fois. Plutôt que d'utiliser le pré-processeur, il faut répartir le code source dans différents fichiers `.c` qui seront compilés séparément et réunit lors de l'édition de lien.

Pour d'autres pièges à éviter, on se refèrera à l'*échelle du Goret*, liste graduée des choses qu'il vaut mieux éviter de faire lorsqu'on programme en C :

`http ://www.bien-programmer.fr/goret.htm`

Chapitre 7

Bibliographie

Langage C, Patrick TRAU
`http://www-ipst.u-strasbg.fr/pat/program/tpc.htm`

Programmation C -Wikibook
`http://fr.wikibooks.org/wiki/Programmation_C`

Programmation en langage C, Anne CANTEAUT
`http://www-roq.inria.fr/codes/Anne.Canteaut/COURS_C`

Initiation à la programmation procédurale, à l'algorithmique et aux structures de données par le langage C, Maude MANOUVRIER
`http://www.lamsade.dauphine.fr/~manouvri/`

Langage C, Patrick CORDE
`http://www.idris.fr/data/cours/lang/c/IDRIS_C_cours.html`

C, Un premier langage de programmation, Jacques LE MAITRE
`http://lemaitre.univ-tln.fr/supports-cours.htm`

... et un nombre incalculable de visites sur des forums...