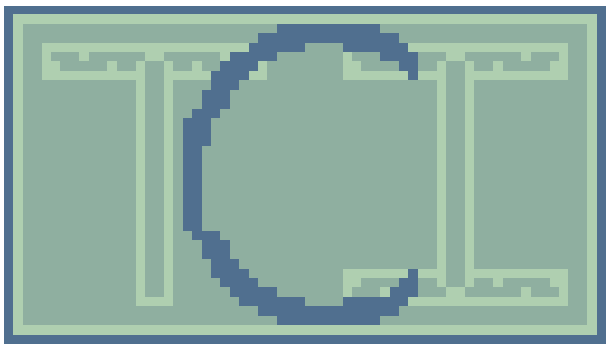
TI-92plus, V-200 & TI-89



Tutorial C

Rédigé par Pascal MARTIN
http://www.squalenet.net/

Sommaire

Introduction

Le C est un langage de programmation qui a été, à l'origine, développé pour ré-écrire le système d'exploitation UNIX, afin de le rendre plus portable. Pour cela, ce langage doit permettre d'accéder à tout ce que propose la machine, mais cela ne s'est pas fait au détriment de la clarté du langage ; en effet, le C est l'un des langages structurés de référence, et fait parti des langages de programmation les plus répandus ; si, un jour, vous faites des études d'informatiques, il est fort probable que vous appreniez à programmer en C !
Il est, depuis quelques années maintenant, possible de programmer en C pour les calculatrices Texas Instruments, modèles TI-89, TI-92+, et V-200. C'est de cela que ce tutorial traitera.

Un programme écrit en langage C doit être "traduit", afin d'être compréhensible par la machine. Pour cette "traduction", on utilise un logiciel, nommé compilateur. Il existe, pour nos calculatrices, deux compilateurs, fonctionnant tous deux sur un ordinateur de type PC. Le premier, historiquement parlant, est TIGCC, qui est un portage de GCC, le célèbre compilateur GNU. C'est celui que nous utiliserons au cours de ce tutorial, puisque c'est le plus fonctionnel, et le plus utilisé ; il permet aussi de programmer en langage d'Assembleur, comme expliqué dans l'un des autres tutoriaux de la TCI.
Le second a été développé plus récemment par Texas Instrument eux-même, et s'appelle TI-Flash Studio ; sa particularité est de permettre le développement d'applications flash. cela dit, il est moins stable, et moins puissant, que TIGCC ; de plus, son nombre d'utilisateurs est extrêmement plus limité.
TIGCC peut être téléchargé sur le site de l'équipe qui se charge de son développement : http://tigcc.ticalc.org

Le pack TIGCC inclut une IDE (Integrated Developpement Environment - Environnement intégré de Développement) ; nous considérerons, dans ce tutorial, sauf lorsque le contraire sera explicitement précisé, que vous utilisez l'IDE.
Cela dit, il nous arrivera parfois d'utiliser le compilateur en ligne de commande, lorsque l'IDE ne nous permettra pas de faire ce que nous voulons.
Notons que le compilateur en ligne de commande peut être utilisé sous Linux, à condition, bien évidemment, de télécharger la version Linux, et non pas la version Windows. Une partie importante des exemples présentés tout au long de ce tutorial a d'ailleur été développée en utilisant la version Linux de TIGCC.

Ce tutorial a été conçu pour les versions 0.95 et 0.96 de TIGCC. Il est possible, étant donné les modifications apportées entre la version 0.94 et la version 0.95, que certains exemples que nous vous proposerons ne fonctionnent pas avec des versions antérieures. De même, bien que la TIGCC-Team essaye au maximum de conserver une compatibilité maximale avec les anciennes versions, il est possible que certains exemples ne fonctionnent pas avec des versions plus récentes de TIGCC (cela est fortement improbable, mais, on ne sait jamais, ce qui explique pourquoi nous précisons que cela peut se produire).
Au cours des premiers chapitres du tutorial, nous essayerons de présenter le mode de fonctionnement des versions antérieures à la 0.95, afin que vous soyez à même de comprendre des codes sources écrits pour une ancienne version, mais, rapidement, nous ne nous consacrerons plus qu'à la version 0.95 et à la version 0.96.

Les diverses parties présentées dans ce tutorial ont été écrites dans leur intégralité par les membres de la TCI®, parfois aidés par d'autres programmeurs. (Dans de tels cas, leur nom est naturellement cité).
Nous remercions tous ceux qui nous ont aidé, notamment en nous envoyant des programmes, ou des remarques.

Écrire un tutorial est chose difficile : il faut parvenir à être progressif dans le niveau de difficulté, des sujets abordés ; il faut aussi réussir à être clair, de façon à ce que nos explications soient comprises par le plus grand nombre, et, surtout, il faut rédiger des exemples suffisamment brefs pour être aisément compréhensibles, tout en étant suffisamment complets afin de bien mettre en évidence leur sujet.
Nous vous demandons donc d'être indulgent par rapport aux erreurs que nous pourrions être amené à commettre, et nous en excusons d'avance.
Pour toute suggestion et/ou remarque, n'hésitez pas à nous contacter via l'adresse E-mail que vous trouverez en bas de chaque page de notre tutorial.

Bon courage !

Pour fini, en tant qu'auteur de ce tutorial, je tiens à remercier toutes les personnes qui m'ont aidées, sans qui ce tutorial ne serait pas ce qu'il est à présent. En particulier :

- Toute l'équipe de TIGCC pour leur formidable outil de développement.
- Verstand et Hibou, pour leur travail sur les deux premières versions de ce tutorial, qui ont en parti resservies à l'élaboration de cette version.
- Les membres, et le webmaster, du forum www.yaronet.com , qui m'ont de nombreuses fois aidés à résoudre des problèmes lorsque j'ai débuté la programmation en langage C, et qui m'ont plusieurs fois conseillés lors de l'élaboration de chacune des versions de ce tutorial.
- Mes parents, qui, depuis que je suis tout petit, m'ont habitué à avoir un ordinateur à la maison, ce qui m'a permis de prendre goût à l'informatique.
- Vincent MARCHAND, pour le travail qu'il a effectué sur l'interface de ce tutorial, afin de l'intégrer au site ti-fr.com , ainsi que pour sa patience face à mes exigences au sujet de sa distribution.
- Vous, qui lisez et utilisez ce tutorial, et qui par là m'encouragez à continuer à travailler dessus.

Licence

Chapitre 1

Premiers Pas sous TIGCC

Comme chacun sait, toute machine informatique ne travaille qu'avec des 0 et des 1 (en règle générale, le courant passe ou ne passe pas). Ainsi le langage de plus bas niveau (celui qui agit directement sur le processeur) n'est qu'une manipulation de 0 et de 1 : le langage de programmation qui s'en approche le plus est l'assembleur. En concevant un programme en assembleur, on n'écrit bien évidement pas avec des 0 et des 1 ; sur un ordinateur, on crée un fichier dans lequel on écrit notre programme avec des instructions de base telles que multiplier, comparer des valeurs, déplacer des données en mémoire... Ensuite, un logiciel appelé " Assembleur " va assembler le fichier (appelé le fichier source) dans le langage de la TI : les 0 et les 1. Mais l'assembleur est souvent difficile à comprendre et à manipuler, en particulier pour les débutants en programmation. Des langages d'un niveau plus haut (plus proche du "langage naturel", mais en anglais dans la majeure partie des cas) ont donc été créé : le TI-BASIC est un exemple parmi d'autres. Pour ce type de langage, il n'y a plus besoin de compilateur, la machine (la TI dans la cas qui nous intéresse) lit le programme tel quel (on dit qu'elle l'interprète). Le TI-BASIC a l'avantage d'être relativement stable (ne plante pratiquement jamais) mais est extrêmement lent. L'assembleur a les propriétés totalement inverses : rapide mais la difficulté à programmer entraîne trop souvent des plantages pour les débutants.

Une alternative à ces difficultés est le C : un peu plus complexe que le TI-BASIC mais pratiquement aussi rapide que l'assembleur.
Son secret est qu'après que vous ayez écrit votre programme en C dans un fichier, le compilateur va s'occuper de vérifier sa cohérence, puis le traduire en langage Assembleur ; celui-ci sera enfin assemblé en langage machine. Le langage C, parmi les trois disponibles pour programmer sur nos calculatrices, offre ainsi, au yeux d'un grand nombre de programmeurs, le meilleur rapport entre facilité de programmation et performances.
De plus, le langage C offre beaucoup plus de possibilités que ne le propose le langage TI-BASIC : par exemple les structures, les tableaux à plusieurs dimensions, ainsi que l'accès à quasiment tout ce que la machine permet de faire.
Notons tout de même une chose dont l'importance n'est pas grande lorsqu'il s'agit d'écrire des programmes de taille raisonnable, mais qui croit avec la complexité, et les besoins de rapidité et d'optimisation : en règle générale, pour des architectures limitées du genre de celle dont nous traitons dans ce tutorial, un bon programmeur en C produira un programme plus rapide et plus optimisé qu'un mauvais programmeur en langage d'Assembleur, mais un bon programmeur en ASM produira toujours un programme de meilleure qualité qu'un bon programmeur C. Cela tenant au fait que le programmeur ASM dit exactement à la machine quoi faire, alors que le programmeur C doit passer par un traducteur, le compilateur. Une solution souvent retenue par les programmeurs connaissant à la fois le C et l'Assembleur, est de profiter des avantages des deux langages, en écrivant la majeure partie de leur programme en C, pour la facilité et rapidité de développement qu'il permet, mais en programmant les routines critiques en ASM. (Mais cela implique de connaître le C, et de bien connaître l'ASM !).

I:\ Création d'un projet sous TIGCC IDE

Sous l'IDE fournie dans le pack de TIGCC, à chaque fois que l'on veut développer un programme, il faut commencer par créer un projet, qui contiendra les différents fichiers utiles à son écriture. Celui-ci regroupera bien sûr votre (ou vos) fichier(s) source(s) (*.c) ; il pourra, et nous le conseillons, contenir des fichiers textes (*.txt) dans lesquels vous pourrez mettre quelques notes ou des fichiers lisez-moi. Vous pourrez y ajouter vos propres headers (*.h), ou encore des fichiers contenant du code Assembleur (*.s pour de l'Assembleur GNU, et *.asm pour de l'Assembleur A68k, qui est traité dans le tutorial de la TCI sur le langage ASM).

A: Avec TIGCC 0.94 et versions précédentes

Pour créer ce projet, cliquez sur *File*, puis sur *New*, et enfin sur *Projet*.
Une fois le projet créé, il convient de lui ajouter au minimum un fichier source, dans lequel nous pourrons écrire le texte de notre programme. Pour cela, encore une fois, cliquez sur *File*, puis *New*, et ensuite, cette fois-ci, puisque nous souhaitons écrire un programme en langage C, choisissez "*C Source File*".
Le logiciel va alors vous présenter une série d'écrans avec des cases à cocher, qui vous permettrons d'affiner quelques options. Pour l'instant, laissez les options par défaut, qui permettent de créer des programmes tout à fait raisonnables, et cliquez sur *Next* à chaque fois. Au final, le fichier C est créé ; il ne reste plus qu'à lui donner un nom. Étant donné que c'est le fichier source principal du projet, vous pouvez, par exemple, le nommer "*main*" (l'extension est automatiquement rajoutée par l'IDE, et n'apparait pas).

A présent, il faut enregistrer le projet sur le disque dur; Pour cela, cliquez sur *File*, puis "*Save All...*". Choisissez ensuite où enregistrer le projet, et donnez lui le nom que vous voulez que votre programme ait au final, une fois envoyé sur la TI (8 caractères maximum, le premier étant une lettre, et les suivants des lettres ou chiffres).

Avec la version 0.94 SP4 de TIGCC, le code généré avec les options par défaut est le suivant :

```
// C Source File
// Created 03/07/2003; 18:46:33

#define USE_TI89 // Compile for TI-89
#define USE_TI92PLUS // Compile for TI-92 Plus
#define USE_V200 // Compile for V200

// #define OPTIMIZE_ROM_CALLS // Use ROM Call Optimization

#define MIN_AMS 100 // Compile for AMS 1.00 or higher

#define SAVE_SCREEN // Save/Restore LCD Contents

#include <tigcclib.h> // Include All Header Files

// Main Function
void _main(void)
{
  // Place your code here.
}
```

Exemple Complet

B: Avec TIGCC 0.95

Avec la version 0.95 de TIGCC, sortie début octobre 2003, on commence par créer un projet, en cliquant sur File, New, Project.
Une fois le projet créé, il nous faut créer, puisque nous souhaitons programmer en langage C, un fichier source C. Pour cela, cliquez sur File, New, "*C Source File*". Donnez un nom à ce fichier ; étant donné que c'est le fichier source principal du projet, vous pouvez, par exemple, le nommer "*main*" (l'extension est automatiquement rajoutée par l'IDE, et n'apparait pas).

A présent, il faut enregistrer le projet sur le disque dur; Pour cela, cliquez sur *File*, puis "*Save All...*". Choisissez ensuite où enregistrer le projet, et donnez lui le nom que vous voulez que votre programme ait au final, une fois envoyé sur la TI (8 caractères maximum, le premier étant une lettre, et les suivants des lettres ou chiffres).

Avec TIGCC 0.95, le code généré est le suivant :

```
// C Source File
// Created 08/10/2003; 14:17:20

// Delete or comment out the items you do not need.
#define COMMENT_STRING         "Place your comment here."
#define COMMENT_PROGRAM_NAME   "Place your program name here."
#define COMMENT_VERSION_STRING "Place your version string here."
#define COMMENT_VERSION_NUMBER 0,0,0,0
/* major, minor, revision, subrevision */
#define COMMENT_BW_ICON \
    {0b0000000000000000, \
     0b0000000000000000, \
     0b0000000000000000, \
     0b0000000000000000, \
     0b0000000000000000, \
     0b0000000000000000, \
     0b0000000000000000, \
     0b0000000000000000, \
     0b0000000000000000, \
     0b0000000000000000, \
     0b0000000000000000, \
     0b0000000000000000, \
     0b0000000000000000, \
     0b0000000000000000, \
     0b0000000000000000, \
     0b0000000000000000}
#define COMMENT_GRAY_ICON \
    {0b0000000000000000, \
     0b0000000000000000, \
     0b0000000000000000, \
     0b0000000000000000, \
     0b0000000000000000, \
     0b0000000000000000, \
     0b0000000000000000, \
     0b0000000000000000, \
     0b0000000000000000, \
     0b0000000000000000, \
     0b0000000000000000, \
     0b0000000000000000, \
     0b0000000000000000, \
     0b0000000000000000, \
     0b0000000000000000, \
     0b0000000000000000}, \
    {0b0000000000000000, \
     0b0000000000000000, \
     0b0000000000000000, \
```

```
      0b0000000000000000, \
      0b0000000000000000, \
      0b0000000000000000, \
      0b0000000000000000, \
      0b0000000000000000, \
      0b0000000000000000, \
      0b0000000000000000, \
      0b0000000000000000, \
      0b0000000000000000, \
      0b0000000000000000, \
      0b0000000000000000, \
      0b0000000000000000, \
      0b0000000000000000}

#include <tigcclib.h>

// Main Function
void _main(void)
{
    // Place your code here.
}
```

Exemple Complet

Beaucoup de choses définies dans ce code ne nous servirons pas avant quelques temps... Nous allons donc les supprimer, afin d'avoir un code source plus petit (et donc un programme moins gros), ce qui nous donnera des exemples plus court, et plus lisibles !
En ne conservant que ce qui nous est actuellement utile, et nécessaire, nous obtenons le code source qui suit :

```
// C Source File
// Created 06/10/2003; 00:02:40

#include <tigcclib.h>

// Main Function
void _main(void)
{
    // Place your code here.
}
```

Exemple Complet

Lorsque nous fournirons des exemples de source dans ce tutorial, nous supposerons que vous avez créé un projet, ainsi qu'un fichier source C, et que vous avez, comme nous venons de le faire, supprimé tout ce qui est inutilement, à notre niveau, inséré par TIGCC.

Notez que, au cours de ce tutorial, nous supposerons que vous utilisez au minimum la version 0.95 de TIGCC.
Il sera assez rare, je pense, que des explications soient données concernant les versions antérieures, principalement du fait que la version 0.95 apporte beaucoup de nouveautés, et énormément de changements.
En fait, jusqu'au chapitre 8 de ce tutorial, il est possible que nous parlions encore de la version 0.94, en annexe de ce que nous dirons pour la version 0.95. Une fois le chapître 8 passé, il est fort peu probable que l'on fasse encore mention de la version 0.94, et, si le cas se produit, ce sera sûrement de façon anecdotique.
Cela est du au fait que les huit premiers chapitres de ce tutorial ont été rédigé avant la sortie de la version 0.95 (ils ont naturellement été revus, une fois la version 0.95 sortie, afin de lui correspondre), alors que les suivants l'ont été après sa sortie.

II:\ Compilation d'un projet sous TIGCC IDE

Enfin, pour créer votre programme exécutable pour la TI (une fois votre programme fini bien sûr), cliquez sur l'icone "*Make*", ou, si vous êtes un grand adepte des raccourcis clavier, appuyez sur la combinaison de touches [Alt+F9].
Le compilateur transformera votre fichier source en fichier exécutable pour TI, si votre source est sans erreur. En cas d'erreur, ou de choses que le compilateur juge douteuses, il apparaîtra peut-être des Warning ou des Errors. Les Errors indiquent qu'il y a une faute de programmation et que la compilation est impossible. Warning indique seulement que le programme présente un problème, mais que cela n'empêche pas la création du fichier exécutable. Cela dit, souvent, en cas de Warning, le programme plantera ou ne fera pas ce qu'il faut (En réalité, certains warnings sont sans importance, ou leur affichage peut être affiché supprimé en ajoutant quelques options de compilation. Cependant, tant que vous ne maîtriserez pas parfaitement la technique de compilation, et surtout dans les cas où vous ignorez la signification des warnings, nous vous encourageons fortement à prêter attention à TOUS ceux qui peuvent apparaître !). N'ayez par contre pas peur si vous voyez afficher une multitude d'erreurs : bien souvent une seule erreur très simple de programmation entraîne une dizaine d'incompréhension pour le compilateur.

III:\ Exécution du programme

Si la compilation s'est déroulée sans erreur, TIGCC aura créé un programme exécutable (fichier comportant l'extension *.89z, ou *.9xz, selon la machine). Vous pouvez à présent l'exécuter.
Pour cela, vous avez naturellement la solution d'envoyer le programme sur votre calculatrice, et de le lancer. Cela dit, lorsque l'on développe, en particulier lorsque l'on débute la programmation, il est fréquent que les programmes que l'on écrit soient buggués, et puisse planter la calculatrice.
C'est pour cette raison que nous vous encourageons à toujours utiliser un émulateur, tel VTI (Virtual TI Emulator) si vous êtes sous Windows, ou TI-Emu si vous êtes sous Linux, pour tester vos programmes, directement sur l'ordinateur.
De plus, envoyer un programme à l'émulateur est nettement plus rapide que de l'envoyer à la calculatrice.

Sous TIGCC-IDE, pour lancer un programme sous VTI, vous pouvez cliquer sur l'icone représentant une flèche verte, ou presser la touche F9. Naturellement, VTI doit être lancé pour que cela fonctionne ; et, de plus, il vaut mieux que vous ne touchiez pas au PC avant que le programme ne soit lancé sur VTI (utiliser le PC en cours de transfert de TIGCC vers VTI peut parfois empécher le bon déroulement de ce transfert).

Nous n'entrerons pas plus dans les détails, et nous n'expliquerons pas ce que fait le code source présenté en exemple plus haut. Nous en reparlerons plus tard, mais avons besoin d'expliquer d'autres choses auparavant, je pense.

Nous avons, au cours de ce premier chapitre, vu comment créer un projet, un fichier source C, comment le compiler et l'exécuter. Au cours du chapitre suivant, nous discuterons de différents modes de programmation qui s'offrent à nous, afin de vous permettre de choisir celui que vous utiliserez généralement.
Une fois ceci fait, nous passerons à la création de notre premier, et simple, programme.

Chapitre 2

Kernel vs Nostub

Ce chapitre va nous permettre, brièvement je l'espère, de parler des deux modes de programmation qu'il existe pour nos calculatrices. Dit comme cela, vous devez probablement vous demander ce que je veux dire, mais vous comprendrez bientôt ; avant d'entrer dans les explications, je devine que vous ne vous sentirez peut-être pas très concerné par ce qui va suivre... Certes, pour des programmes que vous écrivez pour vous-même, que ce soit au cours de votre apprentissage, ou plus tard, pour votre usage personnel, le mode de programmation n'a pas une grande importance ; mais, si un jour vous êtes amené à diffuser votre, son type peut avoir une influence sur ses utilisateurs.

Au cours de ce chapitre, nous allons étudier les différences majeures entre la programmation en mode "nostub", et la programmation en mode "kernel", ainsi que ce qui peut avoir une importance pour l'utilisateur du programme.

I:\ Définitions

Tout d'abord, il serait utile de savoir ce que signifient ces deux termes.

Un programme "nostub" est un programme qui ne nécessite aucun "kernel" (noyau, en français) pour fonctionner. Cela signifie, plus clairement, que, pour utiliser ce type de programmes, il n'est pas utile d'avoir un programme tel que DoorsOS, UniversalOS, TeOS, ou encore PreOS, installé sur sa calculatrice.
Le mode nostub s'est répandu à peu près en même temps que la possibilité de coder en langage C pour nos TIs.

Un programme de type "kernel", parfois aussi dit de type "Doors", du nom du kernel qui a fixé le standard le plus utilisé, est exactement le contraire : il a besoin d'un kernel installé pour fonctionner. Notez qu'il est généralement bon d'utiliser un kernel récent, et à jour, afin de profiter de ce qui se fait de mieux.
Ce mode est, historiquement parlant, le premier à avoir été utilisé, puisqu'il remonte au temps des TI-92 simples ; notez qu'il a, naturellement, vécu des améliorations depuis.

II:\ "Kernel vs Nostub" : Que choisir ?

Chaque mode, quoi qu'en disent certains, présente des avantages, et des inconvénients. Je vais ici tenter de vous décrire les plus importants, afin que vous puissiez, par la suite, faire votre choix entre ces deux modes, selon vos propres goûts, mais aussi (et surtout !) selon ce dont vous aurez besoin pour votre programme.

Mode Kernel :

Avantages	Inconvénients
● Permet une utilisation simple de librairies dynamiques (équivalent des Dll sous Windows) déjà existantes (telles Ziplib, Graphlib, Genlib, ...), ou que vous pourriez créer par vous-même. ● Le Kernel propose de nombreuses fonctionnalités destinées à vous faciliter la vie, ainsi qu'un système d'anticrash parfois fort utile. (Une fois le kernel installé, l'anticrash l'est aussi, pour tous les programmes que vous exécutez sur la machine ; pas uniquement le votre !)	● Nécessite un programme (le Kernel) installé avant que vous ne puissiez lancer le votre. ● L'utilisation de librairies dynamiques fait perdre de la RAM lors de l'exécution du programme (parfois en quantité non négligeable) si toutes les fonctions de celle-ci ne sont pas utilisées. Cependant, notez qu'il est tout à fait possible de programmer en mode Kernel sans utiliser de librairies dynamiques ! Naturellement, la mémoire RAM est récupérée une fois l'exécution du programme terminée.

Mode Nostub :

Avantages	Inconvénients
● Ne nécessite pas de Kernel installé (Fonctionne même, normalement, sur une TI "vierge" de tout autre programme). ● En théorie, si le programme n'a pas besoin des fonctionnalités proposées par les Kernels (qu'il lui faudrait ré-implémenter !), il pourra être plus petit que s'il avait été développé en mode Kernel (car les programmes en mode Kernel sont dotés d'un en-tête de taille variable, qui peut monter à une bonne cinquantaine d'octets, et jamais descendre en dessous de environ 20-30 octets) Cela dit, en pratique, c'est loin de toujours être le cas, en particulier pour les programmes de taille assez importante.	● Ne permet pas, en ASM, la création et l'utilisation de librairies dynamiques (du moins, pas de façon aussi simple qu'en mode Kernel !) ; cela est actuellement permis en C, mais pas encore en ASM. ● En cas de modifications majeures (par Texas Instrument) dans les futures versions d'AMS, certaines fonctions d'un programme Nostub peuvent se révéler inaccessibles, et alors entraîner des incompatibilités entre la calculatrice et le programme. Il faudra alors que l'auteur du programme corrige son code et redistribue la nouvelle version du programme (Sachant que la plupart des programmeurs sur TI sont des étudiants, qui stoppent le développement sur ces machines une fois leurs études finies, ce n'est que rarement effectué !). Ce n'est pas le cas en mode Kernel, pour les fonctions des librairies dynamiques : l'utilisateur du programme n'aura qu'à utiliser un Kernel à jour pour que le programme fonctionne de nouveau.

Dans ce tutorial, nous travaillerons en mode Nostub. Non que je n'apprécie pas le mode Kernel, mais le mode Nostub est actuellement le plus "à la mode". Je me dois donc presque dans l'obligation de vous former à ce qui est le plus utilisé...
Cela dit, il est fort probable que, dans quelques temps,nous étudions pendant quelques chapitres le mode Kernel, ceci non seulement à cause de son importance historique, mais pour certaines des fonctionnalités qu'il propose. A ce moment là, nous le signalerons explicitement.
Bien que n'étudiant pas tout de suite le mode Kernel, je tiens à préciser, pour ceux qui liraient ce tutorial sans trop savoir quel Kernel installer sur leur machine (s'ils souhaitent en installer un, bien entendu), que le meilleur Kernel est actuellement PreOS, disponible sur www.timetoteam.fr.st. C'est le seul qui soit à jour : DoorsOS est totalement dépassé, TeOS l'est encore plus, de même que PlusShell, et UniversalOS n'est plus à jour. De plus, PreOS propose nettement plus de fonctionnalité que DoorsOS ou UniversalOS ! (Notons que PreOS permet naturellement d'exécuter les programmes conçus à l'origine pour DoorsOS ou UniversalOS).

Si vous faites un tour sur les forums de la communauté, vous aurez sans doute l'occasion de croiser des "fanatiques" de l'un, ou de l'autre, mode. Ne leur prêtez pas particulièrement attention ; de toute façon, si l'un vous donne dix arguments en faveur du mode nostub, l'autre vous en donnera vingt en faveur du mode kernel, et vice versa... C'est un débat sans fin, et, malheureusement, récurrent...
Comme je l'ai déjà dit, chaque mode présente ses avantages, et inconvénients ; c'est à vous, et à vous seul, de faire votre choix entre les deux, programme par programme, en pesant le pour, et le contre, de chacun.

Je n'ai pas l'intention de rendre ce chapitre plus long ; l'idée y est déjà, même si nous pourrions débattre sans fin.
Je commencerai le prochain chapitre par une brève réflexion sur la forme à donner à ce tutorial, puis nous verrons notre premier programme, ne faisant absolument rien, si ce n'est rendre le contrôle à la calculatrice une fois terminé, ce qui est une indispensable base !

Chapitre 3

Programmons en C

I:\ Quelques remarques concernant ce tutorial

Nous allons commencer ce chapitre par quelques remarques sur la forme que nous avons choisi de donner à ce tutorial, et peut-être aussi quelques remarques sur son fond, afin d'expliquer les choix que nous avons été amené à effectuer.

A: Un tutorial

Lorsque l'on souhaite écrire un document traitant d'un langage de programmation, tel le C, plusieurs approches s'offrent à nous. En particulier, il faut choisir entre deux possibilités :

- Une approche chapitre par chapitre, chacun d'entre eux traitant d'un sujet différent,
- ou une approche plus linéaire.

Dans le second cas, le tutorial commencerait par énormément de théorie, et finirait par présenter quelques fonctions de base... Autrement dit, nous commencerions par un bla-bla sans fin, extrêmement lassant et complexe à assimiler sans pratiquer, et finirions par ce qui est, pour ainsi dire, le plus simple... De plus, il nous faudrait attendre la fin, ou presque, du tutorial, pour parvenir à réaliser des programmes faisant quelque chose d'utiles, ce qui n'est pas notre but, et probablement pas le votre non plus !
Dans le premier cas, nous présenterions la théorie au fur et à mesure que nous en aurions besoin, ce qui nous permettrait d'immédiatement la mettre en application, de façon à mieux vous la faire assimiler. Certes, une telle approche entrerait certainement moins dans les détails de chaque sujet, mais rien ne nous empêchera d'y revenir plus tard, lorsque nous aurons besoin de ces détails...
Nous estimons que la première approche est la plus adaptée pour l'apprentissage d'un langage tel le C, en particulier pour des débutants en programmation, de par le fait qu'elle nous permet de couvrir différents sujets, qu'elle vous permettra rapidement d'écrire des programmes simples, et qu'elle est, sans nulle doute, la moins lassante. Nous estimons que la seconde approche serait plus adaptée pour l'écriture d'une référence, non d'un tutorial.

Ce tutorial existe en version à consulter en ligne, mais aussi à consulter hors-ligne, téléchargeable en version PDF (c'est la version hors-ligne qui est imprimable via les icones d'imprimante en haut, et en bas, de chaque page du tutorial). Il n'y a pas de différence majeure de contenu entre ces deux versions, mais il peut y avoir quelques différences de forme ou de mise en page.
Par exemple, la version on-line présente en en-tête de chaque chapître une citation, sélectionnée aléatoirement. La version hors-ligne ne les présente pas. Notez que ces citations touchent à la programmation en général, et pas uniquement au langage C, ce qui explique le fait que, par exemple, un nombre non négligeable de citations soient de Bjarne Stroustrup, créateur du langage C++, mais aussi le fait que ces même citations soient aussi utilisées pour le tutorial de la TCI traitant du langage d'Assembleur.

En plus du contenu chapitre par chapitre de ce tutorial, vous avez à votre disposition une page de conseils, qu'il serait bon, à mon avis, que vous lisiez de temps en temps, sachant que plus vous avancerez dans le tutorial, mieux vous comprendrez le contenu de cette page, et une FAQ, dans laquelle j'ai regroupé quelques questions fréquemment posées, avec leurs réponses.
Avant de me poser des questions par mail, j'aimerai que vous utilisiez ces deux pages, ainsi que les divers forums que vous pourrez trouver sur Internet, notamment du fait que je n'ai que très peu de

temps pour m'occuper de mon courrier...

B: Des exemples, indispensables dans un tutorial

Tout au long de ce tutorial, vous trouverez des exemples de codes sources. Au maximum, j'essayerai de les présenter avec la coloration syntaxique par défaut de l'IDE de TIGCC, afin de ne pas changer vos habitudes, à une différence prêt : je ne mettrai pas, dans les exemples de ce tutorial, pour raison de lisibilité, en gras les types de donnés, écris en bleu. Je tenterai aussi de respecter certaines habitudes d'écriture et de présentation, par exemple en indentant, ou en plaçant des caractères d'espacement à certains endroits, afin de rendre le source plus lisible, même si cela n'est nullement une obligation pour le langage C.
Je ne peux que vous encourager à bien présenter votre code source, afin de le rendre plus lisible, ce qui ne peut que faciliter sa compréhension, que ce soit pour vous, si vous êtes amené à le reprendre quelques temps après, ou pour n'importe quel lecteur, si vous diffusez vos programmes en open-source.

Tout au long de ce tutorial, en plus de la coloration syntaxique, j'ai choisi de respecter une norme de présentation pour tous les exemples que je donnerai.
Dans un cadre de couleur violette, vous trouverez les exemples d'algorithmes, ainsi que les notations théoriques conformes à la grammaire du C. Dans un cadre de couleur noire, les exemples de codes sources ; lorsque ceux-ci seront compilables par un simple copier-coller, ils seront suivi de la mention "Exemple Complet" ; si cette mention est absente, c'est que le code source que je vous proposerai ne sera pas entier : il s'agira simplement d'une portion de code source, mettant l'accent sur LE point précis que j'ai choisi de mettre en valeur par cet exemple.

Pour bien vous montrer comment les exemples sont présentés, en voici :

```
Ceci est un algorithme, ou une notion théorique de grammaire.
```

```
Ceci est un exemple, une portion de programme ciblant un point précis.
```

```
Et ceci est un exemple complet, de code source compilable sous TIGCC v0.95
```

Exemple Complet

Vous remarquerez probablement qu'il est assez rare que je mette des exemples complets... En effet, je préfère ne mettre que des portions de sources, plus courtes que des exemples complets, afin de mieux cibler LE point important que l'exemple doit montrer. De plus, je considère qu'il vaut mieux pour vous que vous ayez un peu à réfléchir pour pouvoir utiliser le code que je vous propose en exemple, afin que vous cherchiez à comprendre commet il fonctionne et comment il peut interagir avec le reste d'un programme, plutôt que de vous présenter un programme complet, que vous vous contenterez d'exécuter sans même réfléchir à son fonctionnement.

C: Programmation en C, et normes internationales

Si vous souhaitez en savoir plus sur la programmation en C, je vous conseille vivement d'aller dans une librairie (une grande librairie, de préférence, disposant d'un rayon informatique, afin que vous ayez un choix important ; une FNAC fait généralement parfaitement l'affaire), de feuilleter quelques livres traitant du C, et d'en acheter un. J'insiste sur le fait qu'il est bon de feuilleter plusieurs livres, afin que celui que vous retiendrez vous plaise, ait une présentation agréable, ... , afin que vous ayez envie de l'étudier. Si je peux me permettre un conseil de plus, ne choisissez pas un livre par sa taille : un livre trop petit risque de survoler des notions importantes à force de vouloir les résumer, et un livre trop gros risque d'être indigeste et de vous lasser.
Certes, ces livres traiteront de la programmation en C pour PC ; mais le C est un langage universel : sa logique, que ce soit pour PC (préférez un livre traitant de la programmation en ligne de commande à un livre traitant de programmation en interface graphique style Windows... La programmation en interface graphique est extrêmement complexe pour un débutant en programmation, à mon avis, et s'éloigne grandement de la programmation pour TI), ou pour TI, est la même.
Pour écrire ce tutorial, il m'est arrivé, m'arrive, et m'arrivera, d'utiliser le livre Le Langage C, seconde édition (The C Programming Language, 2nd edition), écrit par Brian W. Kernighan et Denis M. Ritchie, les deux fondateurs du langage C. Ce livre est généralement appelé "K&R", du nom de ses auteurs, et constitue en quelque sorte la référence. Cela dit, je ne le conseille pas à des débutants en programmation... Bien que conforme à l'esprit du langage dont il est la Bible, il ne me paraît pas vraiment adapté à des néophytes... (Ceci est un avis personnel ; libre à quiconque me lisant de penser le contraire, bien entendu).

Je finirai cette partie en disant qu'il existe deux standards concernant la programmation en C :

- Le standard ANSI, qui est admis par tous les compilateurs C, et qui est LE standard,
- et le standard GNU, qui est admis par le compilateur GCC et ses dérivés, dont TIGCC.

Le standard GNU englobe les fonctionnalités du standard ANSI, et rajoute des "extensions GNU", qui le rendent plus riche de par certains aspects.
Cela dit, bon nombre d'extensions GNU ne sont valables que sous le compilateur GCC (et donc TIGCC), et pas sous d'autres compilateurs non-GNU, tels Microsoft Visual C++ (qui est aussi capable de compiler du code C), par exemple, qui suit la norme ANSI.
Notons tout de même que la logique du GNU-C est la même que celle de l'ANSI-C, et que seules des petits extensions ont été rajoutées ; ce ne sont pas des évolutions majeures, mais juste des facilités pour le programmeur, en général. Si vous avez l'habitude d'utiliser du GNU-C, il ne vous sera pas difficile de vous adapter à un compilateur ANSI (Ayant moi-même commencé par apprendre le GNU-C, je n'ai eu aucune difficulté à utiliser des compilateurs ANSI, et parle donc d'expérience vécue).
Une partie des extensions GNU est même généralement acceptée par les compilateurs qui se disent ANSI, même si elles ne sont pas (pas encore, devrai-je dire) officialisées !

II:\ Mais qu'est-ce qu'une TI ?

Avant de commencer à programmer, nous allons, rapidement, voir ce qu'est une TI. Nous n'avons pas besoin, pour programmer en C, de connaître autant de détails que pour programmer, par exemple, en Assembleur, ce qui nous permettra d'être assez bref, et ne ne voir que la couche superficielle.

Ce tutorial est prévu pour les calculatrices Texas Instrument modèles TI-89/92/V-200. A peu de choses près, ces trois machines sont identiques : leurs différences majeures sont au niveau de la taille de l'écran, et du clavier.

Il existe deux versions matérielles différentes : les HW1, et HW2.
Les HW1 sont les plus anciennes, les HW2 les plus récentes. Les HW2 comportent quelques fonctionnalités supplémentaires par rapport aux HW1 (par exemple, les HW1 n'ont pas de support d'horloge). La V-200 est considérée comme une HW2 (Il n'existe pas de V-200 HW1).
Au cours de notre apprentissage de la programmation en Assembleur, il sera rare que nous ayons à nous soucier des différences entre les différentes versions matérielles, mais, quand il le faudra, nous préciserons que c'est le cas.

Il existe aussi plusieurs versions de "cerveaux". Ce "cerveau" est le logiciel qui vous permet de faire des mathématiques, de programmer en TI-BASIC, de dessiner, de lancer des programmes en Assembleur, ... ; bref, ce "cerveau" est ce grâce à quoi votre machine est plus qu'un simple tas de composants électroniques.
Différentes versions allant de 1.00 sur TI-92+, 1.01 sur TI-89, et 2.07 sur V-200, à, pour le moment, 2.09 (sortie durant le second trimestre 2003) ont été diffusées par Texas Instrument. En règle générale, plus la version est récente, plus elle comporte de fonctions intégrées, directement utilisables en Assembleur.
Ce "cerveau" est généralement appelé "AMS" (pour Advanced Mathematic Software), ou, par abus de langage, "ROM" (Pour Read Only Memory), puisque l'AMS est stocké dans un mémoire flashable de ce type.
Le plus souvent possible, nous essayerons de rédiger des programmes compatibles entre les différentes versions de ROM, mais, dans les rares cas où ce ne sera pas possible (par exemple, parce que nous aurons absolument besoin de fonctions qui ne sont pas présentes sur certaines anciennes ROM), nous le signalerons.

Nos TIs sont dotées d'un processeur Motorola 68000, cadencé à 10MHz sur HW1, et à 12MHz sur HW2.

III:\ Un premier programme

Maintenant que nous avons beaucoup parlé, nous allons pouvoir passer à quelque chose d'un peu plus intéressant, notre premier programme.
En fait, nous allons reprendre le code généré par défaut par TIGCC, que nous avons pu voir au premier chapitre, et expliquer ce qu'il contient qui soit, pour le moment, interessant. Nous n'étudierons probablement pas chacun des détails de ce code source, ce serait vouloir aller trop vite, je pense...

Pour nous remettre ce code en mémoire, le voici :

```
// C Source File
// Created 06/10/2003; 00:02:40

#include <tigcclib.h>

// Main Function
void _main(void)
{
    // Place your code here.
}
```

Exemple Complet

Ou alors, si vous utilisez une version de TIGCC antérieure à la 0.95, ce que je vous déconseille, puisque la 0.95 offre plus de fonctionnalités, et que c'est celle dont nous traiterons au cours de ce tutorial, vous aurez quelque chose approchant le code source présenté ci-dessous :

```
// C Source File
// Created 03/07/2003; 18:46:33

#define USE_TI89 // Compile for TI-89
#define USE_TI92PLUS // Compile for TI-92 Plus
#define USE_V200 // Compile for V200

// #define OPTIMIZE_ROM_CALLS // Use ROM Call Optimization

#define MIN_AMS 100 // Compile for AMS 1.00 or higher

#define SAVE_SCREEN // Save/Restore LCD Contents

#include <tigcclib.h> // Include All Header Files

// Main Function
void _main(void)
{
  // Place your code here.
}
```

Exemple Complet

A: Un peu d'anglais, avant tout

La première chose que l'on remarque si on parcoure ce code source est que beaucoup de texte est en anglais. Il faut que vous compreniez une chose : en informatique, l'anglais est la langue de référence ! Si vous cherchez de la documentation sur Internet, vous finirez par tomber sur de l'anglais, si vous lisez la documentation de TIGCC, ce que je vous conseille vivement, vous constaterez qu'elle est en anglais, si vous avez l'intention de parler programmation sur Internet, vous aboutirez à des chans en anglais, un jour ou l'autre, ...
Même si vous n'aimez pas beaucoup l'anglais, si vous voulez vous lancer dans l'informatique, il vous faudra apprendre, comprendre, savoir lire, écrire, et parler l'anglais. Si vous souhaitez effectuer vos études dans un des domaines de l'informatique, l'anglais vous sera quasiment indispensable, croyez-moi.
Étant donné l'importance de l'anglais, je ne prendrai pas la peine de traduire chaque phrase ou mot que nous rencontrerons dans cette langue ; je traduirai parfois certaines notions capitales, mais il vous faudra vous débrouiller pour le reste : je fais parti de ceux qui pensent qu'ils vaut mieux vous pousser à vous améliorer, plutôt que de tout vous mettre entre les mains.

B: Commentons, commentons

La seconde chose que l'on remarque, si l'on prête un tant soit peu attention à la coloration syntaxique, est la forte proportion de vert. Sous l'IDE de TIGCC, cette couleur est utilisée pour représenter ce qu'on appelle "commentaires". Un commentaire est du texte, qui ne sera pas pris en compte par le compilateur lors de la compilation ; les commentaires vous permettent, comme leur nom l'indique, de commenter votre code source, afin de faciliter sa relecture, et sa compréhension. Par exemple, si une ligne vous a posé de gros problèmes à l'écriture, vous pouvez expliquer en commentaire ce qu'elle fait. Si vous utilisez un algorithme particulier, vous pouvez expliquer comment il fonctionne.

Je vous encourage à fortement commenter vos codes sources ; cela vous permettra de mieux les comprendre si vous les relisez plus tard. Certes, au moment où vous écrivez votre programme, vous parvenez sans mal à comprendre ce qu'il fait... Mais six mois plus tard, quand vous aurez oublié la façon dont vous l'avez écrit et les algorithmes que vous avez utilisé, je vous assure que vous aurez beaucoup plus de mal à le comprendre ! Certes, pour un petit programme de quelques centaines de lignes, il n'est pas très difficile de retrouver son fonctionnement en lisant le code source... Mais quand vous avez un gros projet de plus de vingt mille lignes pour vos études, d'une durée de quelques mois, plus deux projets de cinq mille lignes chacun pour TI sur lesquels vous travaillez quand vous avez un peu de temps libre, et qui s'étalent donc eux-aussi sur plusieurs mois, plus quelques TP d'un ou deux milliers de lignes en trois ou quatre langages différents, je suis en mesure de vous assurer que vous comprenez vite l'intérêt des commentaires !
Et ceci est encore plus vrai lorsque vous travaillez à plusieurs sur un projet : commenter permet d'éviter que, toutes les cinq minutes, les gens avec qui vous travaillez ne viennent vous demander ce que fait telle ou telle fonction, que vous n'avez pas commenté en vous disant que c'était évident ! Cela dit, ne tombez pas non plus dans l'extrême : si vous avez une ligne de programme contenant "2+2", il n'est probablement pas très judicieux de la commenter en notant que "celle ligne permet de faire l'addition de deux et deux" ! Ce serait une perte de temps totale que de commenter ce genre de choses !
Pour résumer, le plus difficile avec les commentaires est sûrement de juger où est-ce qu'ils sont utiles, et où est-ce qu'ils ne le sont pas... Mais cela vient avec l'expérience :-)

Il existe deux manières d'écrire des commentaires :

La première, qui est utilisée dans le code source présenté plus haut, est de commencer le commentaire par une suite de deux caractères slash : //
Tout ce qui suit, jusqu'à la fin de la ligne, fera parti du commentaire.
Cette solution pour marquer les commentaires fait parti de la norme C99 (commentaires de style C++), mais je n'ai jamais rencontré de compilateur la refusant, même pour les compilateurs plus anciens.

```
// Ceci est un commentaire
```

La seconde permet d'écrire des commentaires couvrant plusieurs lignes, ou seulement une portion de ligne.
Pour commencer un commentaire de ce type, il faut écrire un slash suivit d'une étoile (symbole de la multiplication) : /*
Et un commentaire de ce type se termine par une étoile suivie d'un slash : */
Autrement dit, ce qui est compris entre /* et */ est le commentaire.
Notez que ces commentaire ne peuvent s'imbriquer : ouvrir un commentaire de ce genre à l'intérieur d'un autre est une erreur, et sera refusé par le compilateur.

```
/* Ceci est
un autre
commentaire */
```

C: La fonction _main

Nous n'étudierons pas au cours de ce chapitre ce qui écrit en **vert foncé gras** dans le code source présenté plus haut ; disons que cette ligne est très bien comme elle est (ces lignes, dans le cas de versions antérieures à la 0.95, ou dans le cas où vous n'auriez pas supprimé ce que nous vous avons conseillé de supprimer, au chapitre 1), pour des programmes simples, et que nous n'avons pas vraiment besoin de nous en préoccuper pour l'instant : mieux vaut se soucier de ce qui est réellement important. Naturellement, nous expliquerons ce que tout ceci signifie, mais dans quelques chapitres, quand nous aurons vu quelques bases avant.
Cela dit, pour la version 0.94 de TIGCC, rien ne vous empêche d'essayer de comprendre les lignes inclues par défaut de par vous-même ; certaines d'entre elles sont aisément compréhensibles grêce à leurs commentaires. D'autres le sont moins... si la curiosité vous brule, et que vous ne pouvez attendre, libre à vous de consulter la documentation de TIGCC (Ces commandes sont toujours documentées, même si elles ne sont normalement plus utilisées directement, depuis la version 0.95 de TIGCC).

Ce que nous allons ici étudier est la fonction principale du programme, la fonction _main, qui correspond à la portion de code reproduite ci-dessous :

```
void _main(void)
{
  // Place your code here.
}
```

Tout d'abord, essayons de définir ce qu'est une fonctions :
Une fonction est un "morceau" de code source, qui permet de regrouper certains traitement dans une sorte de boite, dont on peut ensuite se servir sans même savoir comment elle a été programmée. Avec une fonction correctement conçue, on peut ne pas se soucier de la façon dont un traitement est effectué ; il nous suffit de savoir quel est ce traitement. D'ailleurs, au cours de ce tutorial, et même dans quasiment tous les programmes que vous écrirez, vous utiliserez des fonctions dont vous ne connaîtrez pas le code source : vous saurez comment les utiliser et ce qu'elles font, mais pas comment elles le font. Cette possibilité est une des grandes forces du C.
Une fonction est définie de la façon suivante :

```
type_de_retour nom_de_la_fonction(liste_des_arguments_éventuels)
{
  // code de la fonction.
}
```

La fonction _main est celle qui est appelée automatiquement au lancement du programme ; c'est l'équivalent de la fonction main, sans underscore, sur la majorité des plate-formes.
Pour nos calculatrices, le type_de_retour de cette fonction doit être void, ce qui signifie qu'elle ne retourne rien, et la liste_des_arguments_éventuels doit être void aussi, ce qui signifie qu'elle ne prend aucun argument. Nous verrons plus tard ce que sont les arguments et le type de retour, ainsi que comment les utiliser ; pour l'instant, tout ce que vous devez savoir est que la fonction _main attend void pour les deux.

Dans notre exemple, la fonction _main ne contient qu'un commentaire, c'est-à-dire, aux yeux du compilateur, rien. Donc, lors de l'exécution du programme, rien ne se passera (Ou plutôt devrai-je dire que rien de visible ne se passera... Nous verrons plus tard pourquoi, mais cela est du aux lignes en **vert foncé gras** disparues depuis la version 0.95 de TIGCC, remplacées par des cases à cocher dans les options du projet).

Au cours des programmes que nous serons amené à écrire, nous placerons le code que nous

souhaitons exécuter dans la fonction _main, à la place du commentaire nous disant de "placer notre code ici". Puis, lorsque nos programmes commencerons à être suffisamment important, ou lorsque cela sera judicieux, nous les découperons en plusieurs fonctions, appelées par la fonction _main, et pouvant s'appeler les unes les autres.

Au cours du prochain chapitre, nous étudierons comment appeler une fonction intégrée à la ROM, et n'attendant pas de paramètre, puis nous verrons comment utiliser des fonctions attendant des paramètres.

 hapitre 4

Appel d'un ROM_CALL

Maintenant que nous savons écrire, et compiler, des programmes ne faisant rien, nous allons pouvoir (mieux vaut tard que jamais !) réaliser un programme... faisant "quelque chose".
Principalement, pour commencer du moins, nous nous intéresserons à l'utilisation des fonctions intégrées à la ROM de la TI. Lorsque nous appellerons l'une de ces fonctions, nous réaliserons un "appel à la ROM", traditionnellement appelé "ROM_CALL". Par extension, et abus de langage (ça ne fait qu'un de plus... ça doit être ce qu'on appelle l'informatique :-)), nous emploierons généralement ce terme pour désigner les fonctions inclues à l'AMS en elles-mêmes.
L'ensemble des ROM_CALLs constitue une librairie de fonctions extrêmement complète, et qui, en règle générale, s'enrichit à chaque nouvelle version de ROM. Cet ajout de nouvelles fonctions peut être source d'incompatibilités entre votre programme et des anciennes versions d'AMS. (Jusqu'à présent, il n'y a que de très rares fonctions qui aient disparues, et celles-ci ont été supprimées parce qu'elles présentaient un danger potentiel pour la machine, tout en n'ayant qu'une utilisé limitée.) ; cependant, le plus souvent possible, nous veillerons à conserver la plus grande compatibilité possible.
A titre d'exemple, les ROM 2.0x sont toutes dotées de plus d'un millier de ROM_CALLs, qui permettent de faire tout ce que le TIOS fait ! Nous n'utiliserons qu'une infime partie de ces fonctions pour ce tutorial, et il est quasiment certain que vous n'utiliserez jamais plus du tiers de tous les ROM_CALLs ! (tout simplement parce que vous n'aurez pas l'usage des autres, à moins de développer des programmes un peu particuliers).

Il est possible de passer des paramètres à un ROM_CALL, si celui-ci en attend. Par exemple, pour une fonction affichant un texte à l'écran, il sera possible de préciser quel est ce texte ; pour un ROM_CALL traçant une ligne entre deux points, il faudra passer en paramètres les coordonnées de ces points.
Pour savoir quels sont les paramètres attendus par un ROM_CALL, je vous invite à consulter la documentation de TIGCC, fournie dans le pack que vous avez installé.
Nous verrons tout ceci au fur et à mesure de notre avancée dans ce chapitre...

I:\ Appel d'un ROM_CALL sans paramètre

A: Un peu de théorie

Comme nous l'avons dit plus haut, un ROM_CALL est une fonction inclue dans la ROM de la machine. Un ROM_CALL ne présente aucune différence, aux yeux du programmeur, qu'une fonction qu'il aurait écrit lui-même, si ce n'est qu'il ne l'a pas écrit.
Un ROM_CALL a donc un type de retour, et une liste d'arguments, exactement comme la fonction _main que nous avons étudié au chapitre précédent. Un ROM_CALL exécute une, ou plusieurs, action(s), sans que vous sachiez exactement comment il le fait (à moins d'être très curieux, et de désassembler la ROM... et encore vous faudra-t-il comprendre le code ASM donné par le désassembleur) : tout ce que vous avez besoin de savoir, c'est comment l'utiliser dans votre programme.

Une fonction n'attendant pas d'argument, est appelée en utilisant la syntaxe suivante :

```
nom_de_la_fonction();
```

Veillez à ne pas oublier le point-virgule en fin d'instruction, qui permet au compilateur de, justement, repérer la fin de l'instruction.

B: Un exemple simple

Sur nos TIs, il existe un ROM_CALL, nommé ngetchx, qui attend un appui sur une touche, et qui ne prend pas de paramètre. Nous allons écrire un programme, dans lequel nous utiliserons ce que nous avons dit au chapitre précédent, ainsi que la théorie que nous venons d'expliquer. Ce programme, une fois lancé, attendra que l'utilisateur appuie sur une touche (sauf les modificateurs, tels que [2nd], [<>], [shift], [alpha] sur 89, et [HAND] sur 92+/V200), et rend le contrôle au système d'exploitation de la TI (TI Operating System, abrégé en TIOS, qui est généralement utilisé comme synonyme d'AMS, ou de ROM).
Voila le code source de ce programme :

```
// C Source File
// Created 08/10/2003; 14:17:20

#include <tigcclib.h>

// Main Function
void _main(void)
{
    ngetchx();
}
```

Exemple Complet

Si vous compilez et exécutez ce programme, vous pourrez constater que, comme nous l'attendions, il attend une pression sur une touche, et, une fois la touche pressée, se termine.

II:\ Appel d'un ROM_CALL avec paramètres

A: Un peu de théorie

Appeler un ROM_CALL en lui passant des paramètres est chose extrêmement facile, une fois qu'on a compris le principe. Pour chaque ROM_CALL connu (et documenté), la documentation de TIGCC vous fourni la liste des paramètres qu'on doit lui passer. Il vous suffit de respecter ce qui est indiqué dans la documentation.
Pour appeler une fonction attendant des paramètres (on utilise aussi bien le terme de "paramètre" que celui "d'argument"), il suffit d'écrire son nom, suivi, entre parenthèses, de la liste des arguments.

B: Appel d'un ROM_CALL travaillant avec un quelque_chose *

Comme premier exemple, nous allons utiliser un ROM_CALL permettant d'afficher une ligne de texte dans la barre de status, en bas de l'écran. Ce ROM_CALL s'appelle ST_helpMsg. Il faudra, naturellement, préciser à la fonction quel message afficher ; nous le passerons en paramètre, comme la documentation de TIGCC précise que nous devons agir.
D'ailleurs, si l'on regarde ce qu'on appelle le "prototype" de ce ROM_CALL, on peut voir qu'il est comme reproduit ci-dessous :

```
void ST_helpMsg (const char *msg);
```

Comme nous pouvons le constater, ce ROM_CALL retourne void, c'est-à-dire rien.
Par contre, il attend un const char * en paramètre, ce qui correspond à une chaîne de caractères. Ne prêtez pas attention au nom msg donné à cette chaîne de caractère, il ne sert absolument à rien (et ne servira pas plus dans la suite de ce tutorial), et n'est là que pour que l'écriture soit plus "jolie".
En C, ce qu'on appelle chaîne de caractère correspond à du texte, délimité par des guillemets double.
Par exemple :

```
"Ceci est une chaîne de caractères"
```

Donc, pour appeler le ROM_CALL ST_helpMsg en lui demandant d'inscrire le message "Hello World !", il nous faudra utiliser cette syntaxe :

```
ST_helpMsg("Hello World !");
```

Afin de laisser le temps à l'utilisateur, on effectuera un appel au ROM_CALL ngetchx, afin que le programme ne se termine pas immédiatement.
Au final, notre code source sera celui-ci :

```
// C Source File
// Created 08/10/2003; 14:17:20

#include <tigcclib.h>

// Main Function
void _main(void)
{
    ST_helpMsg("Hello World !");
    ngetchx();
}
```

Exemple Complet

C: Appel d'un ROM_CALL attendant plusieurs paramètres, de type entier

Maintenant que nous avons vu les bases des appels de ROM_CALL avec paramètres, nous allons étendre notre connaissance au passage de plusieurs paramètres, et, pour varier, nous les prendrons cette fois de type entier.
Nous travaillerons ici avec le ROM_CALL `DrawLine`, qui permet de tracer une ligne entre deux points de l'écran, et qui nous permet de choisir le mode d'affichage que nous souhaitons.

Voici la façon dont ce ROM_CALL est déclaré dans la documentation de TIGCC :

```
void DrawLine (short x0, short y0, short x1, short y1, short Attr);
```

Pour l'appeler, nous agirons exactement de la même façon qu'avec le ROM_CALL ST_helpMsg : nous préciserons les arguments, dans l'ordre indiqué.
Par exemple, pour tracer une ligne entre les points de coordonnées (10 ; 30) et (70 ; 50), en mode Normal, c'est-à-dire en noir sur blanc, avec une épaisseur de trait de un pixel, nous utiliserons cette instruction :

```
DrawLine(10, 30, 70, 50, A_NORMAL);
```

Les différents modes de dessin pour cette fonction sont précisés dans la documentation de TIGCC, justement à l'entrée correspondant au ROM_CALL DrawLine.
Les coordonnées sont données en pixels, à partir du coin supérieur gauche de l'écran, qui a pour coordonnées (0 ; 0).
Le coin inférieur droit de l'écran a pour coordonnées, sur TI-89, (159 ; 99), et, sur TI-92+ et V-200, (239 ; 127).

Je pense que nous en avons assez vu pour ce chapitre. Vous pouvez, et je vous encourage à le faire, vous entraîner à utiliser d'autres ROM_CALL, tels, par exemple, DrawStr, qui permet d'afficher un texte à l'écran à la position que l'on désire (il est fort probable que, de toute façon, nous étudions ce ROM_CALL plus loin dans ce tutorial, de part sa grande importance), en vous basant sur la documentation de TIGCC ; cela ne peut vous faire que du bien. Je vous conseille cependant de tester vos programmes sur VTI, afin d'éviter toute mauvaise surprise.
Au chapitre prochain, nous commencerons par voir comment effacer l'écran avant de dessiner quelque chose, afin de rendre nos affichages plus jolis, puis nous verrons comment il se fait que l'écran soit redessiné en fin de programme, et comment empêcher ceci.

 hapitre 5

Effacer l'écran ; Afficher un message en quittant

Ce chapitre va nous permettre de voir comment effacer l'écran, afin que ce que nous y affichons y apparaisse de façon plus satisfaisante, sans avoir le fond du TIOS ; ensuite, nous verrons comment il se fait que l'écran soit restauré à la fin du programme, comment empêcher ceci, et pourquoi. Finalement, nous verrons ce que signifient les commandes commençant par des # qui sont mises par défaut au début du fichier source, à sa création.

I:\ Quelques remarques au sujet des bases de la syntaxe du C

Même si ce n'est pas le sujet de ce chapitre, j'estime qu'il est temps de faire quelques petites remarques concernant la syntaxe du langage C. En fait, j'en ferai deux, pour être précis.

Au cours des brefs exemples que nous avons jusqu'ici rencontré, vous avez sans doute pu remarque la présence de caractères point-virgule en certains endroits du code. Peut-être même avez-vous aussi pu remarquer que leur absence posait problème au compilateur, qui générait alors une erreur...
En C, le point-virgule constitue le marquer de fin d'instruction : le compilateur a besoin qu'on lui indique quand est-ce qu'une instruction prend fin, grêce à ce symbole. Cela permet, par exemple, d'écrire une instruction sur plusieurs lignes, avec ou sans espaces, afin d'améliorer la lisibilité (par exemple, pour que l'instruction tienne sur la largeur de l'écran).

La seconde remarque que j'aimerai faire ici, et que nous illustrerons par un exemple dans quelques chapitres, est qu'il est indispensable de bien faire la différence entre les majuscules et les minuscules ! Par exemple, _main n'est pas la même chose que _MAIN !

II:\ Effacer l'écran, et constater qu'il est restauré

A: Effacer l'écran

Effacer l'écran est une opération particulièrement simple, puisqu'il suffit d'appeler un ROM_CALL, nommé ClrScr, qui signifie "Clear Screen".
C'est une instruction particulièrement utile, qui permet d'afficher des graphismes à l'écran sans conserver le fond du TIOS, qui, rappelez-vous un des exemples vu précédemment, gâche terriblement l'effet...

Nous pourrions utiliser cet exemple-ci, qui trace une ligne après avoir effacé l'écran, et qui attend ensuite une pression sur une touche, afin que l'utilisateur ait le temps de voir la ligne :

```
// C Source File
// Created 03/07/2003; 18:46:33

#include <tigcclib.h>          // Include All Header Files

// Main Function
void _main(void)
{
    ClrScr();
    DrawLine(10, 30, 70, 50, A_NORMAL);
    ngetchx();
}
```

Exemple Complet

Il n'y a rien de particulier à dire sur cet exemple ; si vous tentez de l'exécuter, vous pourrez constater qu'il fonctionne comme nous le souhaitions...

B: Sauvegarde et Restauration "automatique"

Comme vous pouvez le remarquer, une fois le programme terminé, le contenu de l'écran est restauré, automatiquement pourrions-nous dire. En fait, cette restauration n'est pas automatique, dans le sens où ce n'est pas le TIOS qui le re-dessine, et dans le sens où c'est une commande de notre code source qui donne l'ordre de le sauvegarder au lancement, et de le restaurer au moment de quitter.

1: Avec TIGCC 0.95

En fait, une option permet d'indiquer au compilateur si l'on souhaite ou non que l'écran soit redessiné à la fin du programme.
Pour la trouver, cliquez sur Project, Options, Onglet Compilation, Bouton Program Options, et enfin, Onglet Home Screen. Là, vous trouverez une case à cocher intitulée Save/Restore LCD Contents.
Si cette case est cochée, ce qui est le cas par défaut, l'écran sera sauvegardé au lancement du programme, et restauré au moment où celui-ci se termine. Si vous dé-cochez cette case, vous pourrez constater que l'écran n'est plus restauré à la fin du programme : la barre de menus en haut ne devrait plus être visible (mais le redeviendra si vous appuyez sur une des touches de fonction, ou que vous ouvrez, par exemple, le var-link), et la ligne séparant la status line (en bas de l'écran) et le reste de l'écran aura disparu, et ne ré-apparaîtra pas (c'est la seule partie de l'écran qu'il n'est pas possible de récupérer sans utiliser un programme en C ou Assembleur pour la redessiner, ou sans faire de reset).

2: Avec TIGCC 0.94 ou antérieurs

En fait, c'est la ligne suivante :

```
#define SAVE_SCREEN // Save/Restore LCD Contents
```

qui indique, comme nous le prouve le commentaire, au compilateur qu'il va devoir ajouter au programme le code nécessaire pour sauvegarder et restaurer l'écran.
Si vous supprimez cette ligne, ou la passez en commentaire, vous pourrez constater que l'écran n'est plus restauré à la fin du programme : la barre de menus en haut ne devrait plus être visible (mais le redeviendra si vous appuyez sur une des touches de fonction, ou que vous ouvrez, par exemple, le var-link), et la ligne séparant la status line (en bas de l'écran) et le reste de l'écran aura disparu, et ne ré-apparaitra pas (c'est la seule partie de l'écran qu'il n'est pas possible de récupérer sans utiliser un programme en C ou Assembleur pour la redessiner, ou sans faire de reset).

Même si vous ne pensez pas un jour utiliser une version de TIGCC antérieure à la 0.95, je vous conseille de lire ce que je dis à son sujet : cela peut vous aider à comprendre des sources que vous aurez l'occasion de lire, et qui ont été écrites par des programmeurs ayant utilisé une ancienne version de TIGCC, ou n'ayant pas pris la peine de changer leurs habitudes.
Par exemple, définir SAVE_SCREEN a été la méthode conseillée pendant plus de deux ans, il me semble... ce qui correspond à un nombre assez impressionnant de programmes !!!
Cette remarque est vraie pour la sauvegarde de l'écran... elle l'est aussi pour pas mal d'autres choses, même si je ne prendrai pas la peine de la reformuler à chaque fois.

C: Supprimer la sauvegarde et restauration automatique de l'écran

Il n'est pas difficile de supprimer la sauvegarde et restauration automatique de l'écran : nous l'avons fait juste au-dessus, en dé-cochant l'option correspondante (version 0.95 de TIGCC), ou en en supprimant la ligne correspondante, ou en la passant en commentaire (versions 0.94 et antérieures).

Ce sur quoi je voudrai surtout insister, c'est sur l'utilité de ne pas restaurer automatiquement l'écran. Pour le moment, pour les petits programmes que nous avons réalisé, la restauration automatique est bien pratique, et il est vrai qu'elle le restera dans l'avenir, même pour des programmes plus importants. Cela dit, parfois, par exemple, on peut avoir envie de laisser un message à l'écran, une fois le programme terminé, et cela est impossible si la restauration automatique est activée, puisque l'écran sera redessiné une fois que toutes nos instructions auront été exécutées.
Nous verrons dans la prochaine partie de ce chapitre comment faire pour pouvoir afficher un message à l'écran, qui reste une fois le programme terminé, mais sans que l'écran ne soit pas redessiné.

III:\ Laisser un message une fois le programme terminé

A la fin de votre programme, vous pouvez souhaiter laisser, par exemple, un message dans la barre de status en bas de l'écran, renvoyant vers votre site web, ou annonçant votre nom... tout en voulant que l'écran soit redessiné, puisque le fait ne pas restaurer l'écran est, comme nous l'avons vu, assez peu esthétique !
Pour cela, il va vous falloir sauvegarder l'écran "à la main" au début du programme, et le restaurer, toujours "à la main", à la fin de celui-ci. Une fois l'écran restauré, vous pourrez afficher votre message, qui ne sera donc pas écrasé par la restauration de l'écran.
Du temps où j'ai commencé à programmer en C, il fallait d'ailleurs toujours faire comme ça, car l'instruction SAVE_SCREEN n'existait pas encore.

A: Sauvegarder l'écran

Pour sauvegarder l'écran, il vous faut déclarer une variable de type LCD_BUFFER, de la façon suivante :

```
LCD_BUFFER sauvegarde_ecran;
```

(Nous verrons plus en détails au chapitre suivant la déclaration de variables ; pour l'instant, sachez juste que c'est ceci qu'il faut faire, mais que vous pouvez remplacer 'sauvegarde_ecran' par autre chose de plus évocateur à vos yeux, si vous le souhaitez)
Ensuite, il faut sauvegarder l'écran ; pour cela, nous utiliserons la macro LCD_save, en lui passant en argument le nom de la variable que nous venons de déclarer, comme indiqué ci-dessous :

```
LCD_save(sauvegarde_ecran);
```

A présent, nous pouvons effacer l'écran, et dessiner ce que nous souhaitons dessus, comme nous l'avons déjà fait auparavant.

B: Restaurer l'écran

Pour restaurer l'écran, c'est aussi extrêmement simple : il nous suffit, comme montré dans la portion de code reproduite ci-dessous, d'appeler la macro LCD_restore, en lui passant en argument, un fois encore, la variable que nous avons déclaré plus haut.

```
LCD_restore(sauvegarde_ecran);
```

Ce n'est pas plus difficile que cela.

C: Exemple de programme

Maintenant que nous savons sauvegarder et restaurer l'écran, passons à un exemple d'utilisation :

```
// C Source File
// Created 08/10/2003; 14:17:20

#include <tigcclib.h>

// Main Function
void _main(void)
{
    LCD_BUFFER sauvegarde_ecran;
    LCD_save(sauvegarde_ecran);
    ClrScr();
    DrawLine(10, 30, 70, 50, A_NORMAL);
    ngetchx();
    LCD_restore(sauvegarde_ecran);
    ST_helpMsg("Chapitre 5 du tutorial C");
}
```

Exemple Complet

Avec une version de TIGCC antérieure à la 0.95, nous aurions ceci :

```
// C Source File
// Created 03/07/2003; 18:46:33

#define USE_TI89              // Compile for TI-89
#define USE_TI92PLUS          // Compile for TI-92 Plus
#define USE_V200              // Compile for V200

// #define OPTIMIZE_ROM_CALLS // Use ROM Call Optimization

#define MIN_AMS 100           // Compile for AMS 1.00 or higher

//#define SAVE_SCREEN         // Save/Restore LCD Contents

#include <tigcclib.h>         // Include All Header Files

// Main Function
void _main(void)
{
    LCD_BUFFER sauvegarde_ecran;
    LCD_save(sauvegarde_ecran);
    ClrScr();
    DrawLine(10, 30, 70, 50, A_NORMAL);
    ngetchx();
    LCD_restore(sauvegarde_ecran);
    ST_helpMsg("Chapitre 5 du tutorial C");
}
```

Exemple Complet

Comme nous pouvons le voir, le programme s'exécute exactement de la même façon que plus haut (nous avons, une fois de plus, repris le même exemple, en le complétant), et, une fois le programme fini, un message reste affiché dans la barre de status.
Ce message s'effacera dès que l'utilisateur appuiera sur une touche.

IV:\ Les commandes inclues par TIGCC avec les options par défaut

Pour terminer ce chapitre, nous allons parler des commandes inclues par TIGCC au début de notre code : les lignes commençant par un caractère #. Nous n'examinerons que celles qui sont incluses avec les options par défaut ; c'est-à-dire celles que nous avons utilisées dans nos codes sources, jusqu'à présent sans réellement se soucier de ce qu'elles faisaient.
Dans cette partie, la plupart des commandes ont été remplacées par des cases à cocher dans les options du projet lors de la sortie de TIGCC 0.95... en fait, seule la dernière "commande", le #include, n'a pas été remplacé de la sorte. Cela dit, vous serez probablement amené à rencontrer ces options si vous lisez des codes sources écrits par d'autres que vous ; je vous encourage donc à ne pas ignorer cette partie de ce chapitre...

A: Modèles de calculatrices pour lesquels le programme doit être compilé

```
#define USE_TI89                // Compile for TI-89
#define USE_TI92PLUS            // Compile for TI-92 Plus
#define USE_V200                // Compile for V200
```

Ces trois lignes indiquent au compilateur pour quelles machines il doit créer le programme. Vous pouvez créer votre programme pour l'une, les deux, ou les trois machines.
Pour votre usage personnel, autant ne pas se fatiguer à créer des programmes compatibles entre les différentes calculatrices ; cela dit, si vous destinez votre programme à la diffusion, il peut être bon qu'il fonctionne sur chaque modèle de machine, afin de toucher le plus d'utilisateurs possible.

B: Optimisation des ROM_CALLs

```
#define OPTIMIZE_ROM_CALLS // Use ROM Call Optimization
```

Cette directive permet ordonne au compilateur d'optimiser les appels de ROM_CALLs. Cela dit, il faut quelques instructions pour que cette optimisation se fasse, ce qui explique que cette directive fasse grossir la taille du programme si peu d'appels aux fonctions inclues dans la ROM sont effectués.
Cette optimisation réserve aussi un registre, pour toute la durée du programme ; ce registre ne peut donc pas être utilisé pour les calculs. Les registres n'étant que très peu nombreux, et étant les mémoires les plus rapides de la machine, dans le cas d'un programme n'utilisant que peu de ROM_CALL (un jeu évolué, par exemple), on préférera souvent ne pas utiliser cette "optimisation", de façon à accélérer les calculs.

C: Version minimale d'AMS requise

```
#define MIN_AMS 100           // Compile for AMS 1.00 or higher
```

Chaque nouvelle ROM, c'est-à-dire chaque nouvelle version d'AMS (Advanced Mathematical Software, le "cerveau" logiciel de la calculatrice) sortie par Texas Instrument apporte de nouveaux ROM_CALLs ; il est donc des ROM_CALLs qui ne sont pas disponibles sur les anciennes ROMs... Si vous utilisez certains de ceux-là, il vous faut préciser à partir de quelle version de ROM votre programme peut fonctionner, afin qu'il ne puisse pas être lancé sous des versions plus anciennes, où il aurait un comportement indéterminé.

Par exemple, si vous souhaitez utiliser le ROM_CALL AB_getGateArrayVersion, défini uniquement à partir de la ROM 2.00, d'après la documentation, il vous faudra définir ceci :

```
#define MIN_AMS 200
```

Si vous ne le définissez pas, en laissant, par exemple, la valeur par défaut de 100, le compilateur vous renverra un message d'erreur, disant qu'il ne connaît pas ce ROM_CALL.

Cela dit, je vous encourage fortement à n'utiliser que des ROM_CALL existant depuis le plus longtemps possible, afin que votre programme puisse fonctionner sur le maximum de versions de ROM !

D: Sauvegarde/Restauration automatique de l'écran

```
#define SAVE_SCREEN           // Save/Restore LCD Contents
```

Nous avons déjà vu ceci plus haut, puisque c'était le sujet primaire de ce chapitre ; nous ne reviendrons pas dessus.

E: Include standard

```
#include <tigcclib.h>         // Include All Header Files
```

Pour finir ce chapitre, parlons rapidement de la directive #include.
Elle permet, comme son nom l'indique, d'ordonner au compilateur d'inclure du texte, tel, par exemple, du code, depuis un fichier, pris dans le répertoire par défaut (c:\Program Files\TIGCC\Include\C si vous avez suivi l'installation par défaut) si le nom de ce fichier est écrit entre chevrons (entre le caractère '<' et le caractère '>'), ou dans le répertoire courant s'il est écrit entre guillemets doubles ("").

Le fichier tigcclib.h contient les prototypes des ROM_CALLs, ainsi que de nombre autres fonctions. Le fait de l'inclure permet au compilateur de savoir quels paramètres les ROM_CALLs doivent recevoir, quelles sont leurs valeurs de retour, ...
Il est généralement nécessaire d'inclure ce fichier pour parvenir à compiler un programme sous TIGCC.

Nous voila parvenu à la fin de ce chapitre. Le prochain nous permettra de parler de la notion de variables, et d'apprendre comment en déclarer, ainsi que les bases de leur utilisation.

 hapitre 6

Les variables (types, déclaration, portée)

Nous allons à présent parler de ce qu'on appelle, en C, les variables.
Qu'est-ce qu'une variable ? Pour faire court et simple, c'est un emplacement mémoire, de taille limitée, qui est à la disposition du programme, pour que le programmeur puisse y mémoriser des données dont la valeur n'est pas fixe.
Au cours de ce chapitre, nous verrons comment créer des variables, quels sont les types de variables que l'on peut utiliser, quelles sont les valeurs maximales et minimales qu'elles peuvent contenir, ...

Avant toute chose, je tiens à attirer votre attention sur le fait que ce que l'on appelle "variable", en C, est différent de ce qui est appelé de la même façon en TI-BASIC. Aux yeux du BASIC, une variable est un fichier, qui apparaît dans le menu VAR-LINK, qui peut être utilisé par plusieurs programmes, et même en dehors de tout programme.
En C, une variable est interne au programme, n'apparaît pas en dehors du programme, et ne peut pas être utilisée ailleurs que dans le programme. C'est de ce genre de variable dont nous allons parler dans ce chapitre, et c'est ce que nous nommerons "variables" dans la suite de ce tutorial.

Il existe plusieurs types de variables, en C ; au cours de ce chapitre, nous nous limiterons aux variables arithmétiques, qui sont celles dont nous avons l'usage pour des programmes simples. Plus tard, nous parlerons des tableaux, puis des pointeurs, tant redouté par les débutants, et enfin des structures, qui permettent de regrouper plusieurs données dans une seule variable, mais je pense qu'il vaut mieux ne pas apporter trop de nouveautés à la fois, et commencer par le plus simple. Les chaînes de caractères ne constituent pas un type en elles-même ; à cause de leur importance, nous leur consacrerons un chapitre entier, une fois que nous aurons étudié les tableaux et les pointeurs.

I:\ Les différents types de variables, leurs modificateurs, et leurs limites

En C, il existe deux familles de variables concernées par ce chapitre : les entiers, et les réels (souvent appelés "nombres en virgule flottante", communément abrégé en "flottants"). Ces deux familles sont découpées en plusieurs tailles, permettant de stocker des nombres plus ou moins grands.

A: Les entiers

1: Ce que dit la norme

Tout d'abord, précisons que l'écriture suivante :

```
sizeof(type)
```

renvoie la taille, en octets, que prend une variable du type précisé ; cela pourra nous servir dans la suite de ce chapitre. Je vous encourage d'ailleurs, lorsque vous avez besoin d'utiliser le nombre d'octets que fait un type de variables, à toujours employer sizeof plutôt que la taille que vous pensez que fait une variable. Ainsi, votre code sera plus facilement portable sur d'autres machines. (d'autant plus que la taille, en octet de chaque type n'est pas fixé par la norme !).
Sachez aussi que sizeof n'est pas réservé aux variables entières seulement.

La taille en octets des types de variables entières n'est pas fixé par la norme ANSI, ni par la norme GNU.
La seule chose qui est précisée, c'est que `sizeof(char)` doit être inférieure ou égale à `sizeof(short)`, qui doit être inférieure à `sizeof(int)`, qui doit être inférieure ou égale à `sizeof(long)`, qui doit elle-même être inférieure ou égale à `sizeof(long long)`, sachant que int correspond généralement au mot-machine.

2: Ce qu'il en est pour nos calculatrices

Maintenant que nous avons vu ce que dit la norme, voyons ce qu'il en est pour nos calculatrices, et ce que signifie les différents types énoncés ci-dessus.

Écriture complète	Écriture concise, généralement utilisée	Taille, avec les options par défaut	Intervalle de valeurs
`signed char`	`char`	8 bits, 1 octet	de –128 à 127
`unsigned char`	`unsigned char`	8 bits, 1 octet	de 0 à 255
`signed short int`	`short`	16 bits, 2 octets	de -32 768 à 32 767
`unsigned short int`	`unsigned short`	16 bits, 2 octets	de 0 à 65 535
`signed int`	`int`	16 bits, 2 octets	de -32 768 à 32 767
`unsigned int`	`unsigned int`	16 bits, 2 octets	de 0 à 65 535
`signed long int`	`long`	32 bits, 4 octets	de -2 147 483 648 à 2 147 483 647
`unsigned long int`	`unsigned long`	32 bits, 4 octets	de 0 à 4 294 967 296
`long long int`	`long long`	64 bits, 8 octets	de -9223372036854775808 à 9223372036854775807
`unsigned long long int`	`unsigned long long`	64 bits, 8 octets	de 0 à 18446744073709551615

Comme on peut le remarquer, avec les options par défaut, ou, plutôt, sans options particulières, les int sont codés comme des short. Cela dit, il est possible de passer au compilateur une option lui indiquant de considérer les int comme des long...
Je vous conseille donc de toujours utiliser des short à la place des int (autrement dit, lorsque vous voulez une valeur sur 16 bits), afin d'être plus précis, et, lorsque vous voulez une valeur sur 32 bits, utilisez des long. C'est ce qui se fait toujours ou presque parmi les développeur pour TI ; autant que vous suiviez cette habitude.

Une autre habitude qu'il peut être bon de prendre est d'utiliser un type correspondant aux valeurs que vous pouvez vouloir mémoriser dedans. Par exemple, n'utilisez par des long long pour stocker un nombre compris entre 0 et 10 ; ce serait une perte de temps ridicule, surtout sachant que les long long ne sont pas utilisables directement par le microprocesseur (il travaille sur 32 bits au maximum), et doivent être "traduit" afin d'être utilisables, ce qui prend du temps.
Les nombres sur un ou deux octets sont ceux avec lesquels le processeur travaille le plus rapidement ; ensuite viennent les nombres sur 4 octets, et enfin ceux sur plus, qui reçoivent un traitemant particulier.

Aussi, si vous savez que vous ne travaillerez qu'avec des valeurs positives ou nulles, autant utiliser une variable de type unsigned. Cela permettra au compilateur de vous prévenir si, par mégarde, vous utiliser une valeur négative, et votre code source sera plus compréhensible : la personne qui le lira saura que la variable qu'elle a sous les yeux ne passe jamais en dessous de 0, et qu'il n'est pas nécessaire de réfléchir à ce qu'il se passerait si cela arrivait.

Enfin, le tableau ci-dessus vous présente une écriture que j'ai surnommé "complète", et une écriture plus concise. Comme noté en tête de colonne, ce sont les écritures concises qui sont généralement utilisées : un programmeur C ne se fatiguera pas à écrire quelque chose d'inutile, et profitera au maximum des fonctionnalités et de la souplesse du langage !

B: Les flottants

1: Ce que dit la norme

Pour ce qui est des nombres flottants, la norme définit ceci : `sizeof(float)` doit être inférieure ou égale à `sizeof(double)`, qui doit elle même être inférieure ou égale à `sizeof(long double)`.

2: Ce qu'il en est pour nos calculatrices

Sur nos machines, les trois types float, double, et long double sont tous équivalents.
On utilisera généralement le type `float`, car c'est celui qui correspond le mieux aux noms de fonctions du TIOS.

Un float, comme nous pourrons les utiliser sur TI, est compris entre 1e-999 et 9.999999999999999e999, avec une précision de 16 chiffres significatifs.
Pour les curieux, le format d'encodage de flottants sur TI n'est pas celui généralement utilisé sur PC, qui demande trop de calculs pour l'encodage et le décodage, mais SMAP II BCD.

II:\ Déclaration, et Initialisation, de variables

A: Déclaration de variables

Déclarer une variable est une opération extrêmement simple, qui nécessite deux choses :

- Le type de de variable que l'on souhaite créer,
- et le nom que l'on souhaite lui donner.

Le type de variable que l'on souhaite créer dépend, bien évidemment, de nos besoins, et nous le choisirons parmi ceux proposés plus haut.
Le nom de la variable, quand à lui, doit répondre à quelques règles, que nous étudierons plus loin. Pour l'instant, il nous suffit de savoir qu'il peut contenir des lettres, et le caractère underscore (généralement, Alt-Gr + 8, sur les claviers azerty).

On commence par écrire le type de la variable souhaitée, et on le fait suivre par le nom que l'on veut lui donner, en terminant le tout, bien entendu, par un point virgule, pour marquer la fin de l'expression.
Par exemple, pour déclarer une variable contenant un entier de type short, nommée 'a', nous utiliserons la syntaxe suivante :

```
short a;
```

Pour déclarer une variable de type nombre flottant, nommée 'pi', nous utiliserons la même syntaxe, comme suit :

```
float pi;
```

Il en va de même pour les autres types de variables ; pour cette raison, nous ne donnerons pas d'exemples supplémentaires.

Notez qu'il est parfaitement possible de définir plusieurs variables, du même type, sur une seule ligne logique. Pour cela, vous précisez le type de variables, et ensuite, la liste des noms de variables, séparés par des virgules, comme indiqué ci-dessous :

```
char a, b, c,
           d, e;
```

GCC, et donc, TIGCC, permet de déclarer des variables un peu n'importe où dans son code source, du moment qu'on les déclare avant de les utiliser. Cela dit, je vous conseille de regrouper vos déclarations de variables en début de bloc, c'est-à-dire après les accolades ouvrantes, afin de plus facilement les retrouver. Si vous avez dix variables à déclarer, autant toutes les déclarer au même endroit, plutôt qu'une parci et une parlà ! De la sorte, si vous devez modifier quelque chose, vous vous y retrouver plus facilement. De même, vous n'aurez pas à vous demander au moment d'utiliser une variable que vous savez avoir déclaré si sa déclaration est avant ce que vous voulez écrire (ce qui correspond à ce que le compilateur attend), ou après (ce qui générerait une erreur à la compilation).

B: Initialisation de variables

On appelle "initialisation" d'une variable le fait de lui donner une valeur pour la première fois. Pour donner une valeur à une variable (on dit "affecter" une valeur à une variable), on utilise l'opérateur d'affectation, '=' (Le symbole égal).
Cet opérateur affecte à la variable placée à sa gauche le résultat de l'expression placée à sa droite. Au cours de ce chapitre, nous n'utiliserons que des nombres comme expression, mais nous verrons plus tard que le terme d'expression regroupe bien plus que cela.

Notez que l'opérateur d'affectation, en C, fonctionne dans la sens inverse de celui que vous avez pu utiliser si vous avez programmé en TI-BASIC ; celui du TI-BASIC affecte la valeur à sa gauche dans la variable à sa droite.

Il est possible d'initialiser une variable en deux endroits : au moment de sa déclaration, et ailleurs dans le source.
En fait, seule l'affectation à la déclaration est un cas particulier, et constitue véritablement une initialisation. Donner une valeur à une variable à un autre moment qu'à sa déclaration se fait toujours de la même façon, que ce soit ou non la première fois.

1: A la déclaration

Comme nous l'avons laissé entendre, il va nous falloir utiliser l'opérateur d'affectation, en ayant à sa gauche notre variable, et à sa droite la valeur que nous souhaitons donner à celle-ci.
Cela dit, nous voulons que cette affectation se fasse à la déclaration de la variable, et, pour déclarer une variable, nous avons vu qu'il fallait écrire son type à gauche de son nom.

En combinant les deux, nous obtenons pour, par exemple, déclarer une variable de type long, nommée 'mavar', en l'initialisation à 234567, cette syntaxe :

```
long mavar = 234567;
```

De la même façon, pour déclarer une valeur de type flottant nommée pi, et approximativement égale à la valeur de PI, nous utiliserons ceci :

```
float pi = 3.141592;
```

Comme vous pouvez le remarquer, le séparateur entre la partie entière, et la partie décimale n'est pas, en C, la virgule, mais le point (que nous appelons alors "point décimal", afin de bien le faire correspondre à sa fonction).

Ici encore, il est possible de déclarer, et d'initialiser, plusieurs variables sur une même ligne logique, toujours en les séparant par des virgules :

```
short a=20, b=40, c, d=56;
```

Les variables a, b, et d seront initialisées ; la variable c ne le sera pas. Ceci est parfaitement possible, et ne pose absolument aucun problème au compilateur.

2: Ailleurs

Pour affecter une valeur à une variable ailleurs dans le code source, que ce soit ou non la première fois que nous le faisons, nous utiliserons la même syntaxe, mais sans préciser le type de la variable (ce qui reviendrait à la redéclarer, et une variable ne peut pas être déclarée deux fois !).
Une chose importante : en C, pour pouvoir utiliser une variable, il faut qu'elle ait été déclarée. Affecter une valeur à une variable revient à l'utiliser ; il faut donc, avant de l'initialiser, bien l'avoir déclarée ! Si vous essayez d'utiliser une variable non déclarée, le compilateur générera une erreur.
Une autre remarque est que le type d'une variable est fixé à sa déclaration, et ne peut pas changer. Par exemple, si vous avez déclaré une variable comme étant d'un des types entiers, vous ne pouvez pas lui affecter une valeur contenant un point décimal ! Cela aussi provoquerait une erreur de la part du compilateur. De la même façon, si vous avez déclaré une variable comme étant unsigned, vous ne pouvez pas lui affecter une valeur négative.

La syntaxe étant similaire à celle déjà étudiée, nous ne donnerons qu'un seul exemple. Supposons que nous ayons une variable de type unsigned short, nommée varshort, qui ait été déclarée correctement.
Pour lui affecter la valeur 10, nous utiliserons la syntaxe suivante :

```
varshort = 10;
```

Il en va de même pour les autres types simples, ceux que nous avons vu au cours de ce chapitre.

III:\ Quelques remarques concernant les bases, et le nommage des variables

A: Une histoire de bases

Tous les jours, nous sommes amenés à utiliser des nombres, et nous le faisons généralement en base 10, c'est à dire en utilisant des chiffres allant de 0 à 9.

En C, il est possible d'utiliser la base 2 (binaire), la base 8 (octale), la base 10 (décimale), et la base 16 (hexadécimale).
Les nombres en binaire sont préfixés de 0b (Le chiffre zéro, suivi de la lettre 'b'), les nombres en base 16 de 0x ou 0X, et les nombres en base 8 d'un 0 tout seul. La base 10 est la base par défaut, ce qui explique pourquoi les nombres exprimés en décimal ne sont pas préfixés. Les nombres que nous avons utilisés pour haut dans ce chapitre sont donc, comme la logique le voudrait, en base 10, qui est la base que nous avons l'habitude d'utiliser.

Le binaire est souvent utilisé car il permet de représenter de façon claire des données conformément à la façon dont elles sont codées dans la machine ; par exemple, le courant passe, ou ne passe pas, la charge magnétique est positive, ou négative...
A chaque fois, on a deux états possibles. Le binaire code ses nombres à l'aide de 0 ou de 1, c'est-à-dire deux états possibles, qui correspondent aux deux états possibles.
Chaque chiffre d'un nombre est appelé "digit". Pour le langage binaire, il s'agit de "binary digit", communément abrégé en "bit". Un bit correspond à la plus petite unité d'information possible.

L'héxadécimal, est utilisé pour faciliter l'écriture et la recopie de données binaires. En effet, un digit hexadécimal peut coder 16 valeurs, ce qui correspond à 4 bits. Ainsi, un mot d'un octet sera codé sur deux digits, en hexadécimal, au lieu de huit en binaire, diminuant fortement le risque d'erreurs à la recopie.
Les nombres en base 16 sont codés en utilisant les dix chiffres habituels, de 0 à 9, et les six premières lettres de l'alphabet, de A à F.

La base octale n'est que rarement utilisée. Elle code ses nombres en utilisant les chiffres allant de 0 à 7.

Nous ne détaillerons pas ici comment convertir un nombre d'une base dans une autre, afin de ne pas rendre ce chapitre trop long, d'autant plus que ce n'est pas le sujet. Cela dit, si vous n'avez jamais étudié cela en cours, ou que vous pensez avoir besoin de révisions, vous pouvez consulter cette page.

B: Nommage des variables

La norme C est assez souple pour ce qui concerne le nommage des variables, mais aussi assez stricte sur certains points :

Un nom de variable

- Peut contenir des lettres de A à Z, et de a à z, ainsi que les dix chiffres de 0 à 9, et le caractère underscore (tiret souligné : '_').
- Doit commencer par une lettre.

Vous devez préter attention au fait que le C est sensible à la case, ce qui signifie que les majuscules et les minuscules sont reconnues comme des caractères différents. Par exemple, mavar, MAVAR, et mAvAr sont trois variables différentes !

Je vous conseille d'adopter une habitude concernant le nommage des variables, et de vous y tenir ; vous vous y retrouverez plus facilement de la sorte.
Voici plusieurs exemples d'habitudes de nommages ; libre à vous d'en retenir une, ou d'en choisir une autre.

- Séparer les 'mots' constituant le nom de la variable par des underscore ; par exemple : ceci_est_une_variable.
- Mettre la première lettre de chaque 'mot' en majuscule : CeciEstUneVariable.
- Même chose, sauf pour le premier mot (il me semble que c'est ce qui est généralement fait en langage JAVA) : ceciEstUneVariable.
- Préfixer le nom de la variable par quelques lettres indiquant son type ; par exemple, l_ pour un long, f_ pour un float, ... (On se rapproche de la notation hongroise beaucoup utilisée par les programmeurs sous interface windows utilisant les MFC).

Cela dit, je vous conseille aussi de choisir des noms qui ne soient pas trop longs (car fatiguant à taper, et vous risquer d'en oublier la moitié), ni trop court (afin que vous sachiez à quoi sert la variable).
Une habitude généralement prise est d'utiliser des noms de variables en une lettre (en particulier 'i', puis 'j', 'k', ...) en tant que variables de boucles ; nous étudierons dans quelques chapitres ce que cela signifie ; si j'y pense, je repréciserai ceci à ce moment là.

IV:\ Portée des variables, et espace de vie

Comme nous l'avons déjà dit, nous devons impérativement déclarer les variables avant de pouvoir les utiliser. Cela dit, il y a quelques éléments supplémentaires qu'il peut être bon de connaître.

Plus haut, nous avons, très brièvement, parlé de la notion de bloc, qui correspond à tout ce qui est compris entre une accolade ouvrante et l'accolade correspondante. Par exemple, une fonction, telle _main, que nous avons déjà utilisé, est un bloc.
Pour bien illustrer ce fait, voici comment nous l'avons écrite il y a quelques chapitres :

```
void _main(void)
{ // Début d'un bloc
    ST_helpMsg("Hello World !");
    ngetchx();
} // Fin du bloc
```

Cela dit, un bloc ne correspond pas nécessairement à une fonction. Il est d'ailleurs possible d'imbriquer les des blocs, et nous le ferons extrêmement souvent dans l'avenir.

Il existe deux genres différents de variables, dont la portée, c'est-à-dire la portion de programme où elles sont visibles, est différente : les variables locales, et les variables globales.

A: Variables locales

Une variable locale est une variable qui est déclarée dans une fonction, ou, plus généralement dans un bloc.
Plusieurs règles définissent les zones où les variables locales existent. Nous allons voir chacune de ces règles, en leur associant à chaque fois un exemple, afin de bien les illustrer.

Une variable locale existe à partir du moment où vous la déclarez, que ce soit en début de bloc si vous avez suivi le conseil que j'ai donné plus haut, ou plus loin dans le corps du bloc si vous avez préféré déclarer votre variable seulement au moment de son utilisation, jusqu'à la fin du bloc dans lequel elle est déclarée :

```
{ // Début du bloc
    // Instructions diverses, ou aucune instruction.
    // la variable a n'a pas été déclarée, et ne peut donc pas être utilisée.
    short a;
    // La variable a existe, à présent.
} // Fin du bloc => La variable a cesse d'exister.
```

Il en va de même quelque soit la profondeur du bloc ; qu'il s'agisse ou non d'un bloc enfant n'a absolument aucune influence sur ce point, comme nous pouvons le remarquer ci-dessous :

```
{
    // La variable a n'existe pas ; il est impossible de l'utiliser
    {
        // La variable a n'existe toujours pas.

        short a;
        // A présent, la variable a existe ; il devient possible de l'utiliser.

        // Fin du bloc dans lequel la variable a a été déclarée.
        // La variable a cesse donc d'exister.
    }

    // La variable a n'existe pas :
    // Puisqu'elle a été crée dans un bloc enfant, elle a été détruite
    // à la fin du bloc enfant qui l'avait créé.
}
```

Une variable locale est visible depuis les blocs enfants de celui dans lequel elle a été déclarée, à partir du moment où ces blocs enfants sont localisés après sa déclaration :

```
{
    short a;
    // La variable a, déclarée, peut être utilisée dans ce bloc
    {
        // On est dans un bloc enfant de celui dans lequel
        // la variable a a été créée
        // On peut donc utiliser la variable a ici.


        // La variable a a été créée dans un bloc père de celui dont on est
        // à la fin. La variable a n'est donc pas détruite.
    }
    // La variable a existe toujours, et peut donc être utilisée.

    // On arrive à la fin du bloc dans lequel la variable a a été déclarée
    // La variable a est donc détruite.
}
```

Cela dit, si vous définissez dans un bloc une variable de même nom qu'une variable définie dans le bloc père, la variable du bloc père sera masquée par la variable du bloc courant, ce qui signifie que la variable que vous utiliserez sera celle du bloc enfant, et non celle du bloc père. Ceci est un comportement susceptible de vous induire en erreur, par exemple si vous ne pensez pas au fait que vous avez déclaré une variable de même nom que dans le bloc père !
L'exemple ci-dessous illustre bien ceci :

```
{
    short a;
    // On a déclaré une variable a, de type short.
    // On considérera que cette variable est la variable a n°1
    // On peut, naturellement, utiliser la variable a
    // ce sera a n°1 qui sera utilisée.
    {
        float a;
        // On a déclaré, dans ce bloc, une autre variable a, de type float
        // On considérera que cette variable est la variable a n°2

        // Si, dans ce bloc, on utilise une variable nommée a,
        // le compilateur utilisera la variable a n°2,
        // car la déclaration de a dans ce bloc
        // masque la déclaration de a déclarée dans le bloc père


        // On arrive à la fin du bloc qui a créé la variable
        // a n°2. Celle-ci est donc détruite.
    }
    // Si, maintenant, on utilise la variable a, ce sera la n°1,
    // puisqu'on est dans le bloc où c'est celle-ci qui a été déclarée.
    // Précisons que son contenu n'aura nullement été affecté par la
    // déclaration et l'utilisation de la variable a n°2 dans le bloc fils.

    // Fin du bloc dans lequel la variable a n°1 a été déclarée.
    // Elle va donc être détruite.
}
```

Le seul cas présentant un danger pour le programmeur est le dernier ; les autres ne sont qu'une question de logique et d'habitude, à partir du moment où l'on sait qu'une variable doit être déclarée avant de pouvoir être utilisée.

B: Variables globales

Nous allons à présent pouvoir étudier le cas des variables globales. Une variable "globale" est une variable qui est définie en dehors de tout bloc.
Attention, il ne s'agit pas d'une variable qui n'est définie dans aucun bloc, bien au contraire ! En effet, une variable globale est utilisable dans tous les blocs du programme, ou, du moins, dans tous les blocs qui suivent sa définition.

Par exemple, si nous définissons une variable globale avant la fonction _main, cette variable sera valable dans le code de la fonction _main, comme on pourrait s'y attendre. Mais, si l'on définit d'autres fonctions que _main, comme nous apprendrons à le faire dans quelques chapitres, cette variable sera aussi accessible depuis ces autres fonctions, ce qui n'était pas possible avec des variables locales.

Une variable globale se définit exactement de la même façon qu'une variable locale, mis à part le fait que sa déclaration se fait en dehors de tout bloc. Son initialisation peut se faire à la déclaration, comme pour les variables locales.
Ensuite, on peut l'utiliser de la même façon que les variables locales : la seule différence est, j'insiste, que sa déclaration se fait en dehors de tout bloc.

Deux choses sont à noter, concernant les variables globales :

- Si le programme n'est ni archivé, ni compressé, les variables globales concervent leur valeur entre chaque exécution du programme.
- Utiliser des variables globales fait augmenter la taille du programme, car elles sont mémorisées dans celui-ci, contrairement aux variables locales, qui sont créées au moment de leur déclaration.

Je finirai par une petite remarque : utiliser des variables globales est souvent considéré comme assez sale. En effet, le fait que ces variables puissent être masquées par des variables locales (exactement de la même façon qu'une variable locale d'un bloc peut masquer une variable déclarée dans un bloc père) et soient accessibles depuis n'importe quel point du programme rend leur utilisation assez "risquée", si je puis me permettre d'utiliser un terme aussi fort.
Conjointement avec la seconde remarque que j'ai fait juste au dessus, cela a pour conséquence que les variables globales sont assez peu utilisées ; je vous conseille de suivre cette habitude, et de ne les utiliser que lorsqu'il n'est pas possible de faire autrement, c'est-à-dire, rarement !

Il reste encore des éléments concernant les variables dont je n'ai pas parlé, en particulier, l'utilité et l'utilisation des mots-clés auto, register, static, extern, volatile, et const, mais je ne pense pas qu'ils soient utiles au niveau où nous en sommes, et vous n'en n'aurez pas besoin avant quelques temps. Je n'ai pas envie de rendre ce chapitre, déjà assez long, et probablement quelque peu rébarbatif, encore plus long, surtout pour quelque chose qui n'est pas extrêmement utile ; nous les présenterons donc lorsque nous en en aurons l'usage.

Chapitre 7

Affichage d'un nombre à l'écran ; Valeur de retour d'une fonction

Ce chapitre va nous permettre d'étudier comment utiliser la valeur de retour d'une fonction, ou, plus particulièrement dans l'exemple que nous prendrons, d'un ROM_CALL. Nous verrons auparavant comment afficher une valeur à l'écran, montrerons l'importance du respect des majuscules et minuscules, et finiront ce chapitre par une remarque concernant l'imbrication d'appels de fonctions.

I:\ Afficher un nombre à l'écran

Il existe plusieurs façon d'afficher une valeur à l'écran... Certaines sont standards, d'autres non. Certaines sont simples d'emploi, d'autres non.
Au cours de ce chapitre, et des suivants, nous utiliserons la fonction printf.
Plus loin dans ce tutorial, il est possible que nous voyons d'autres méthodes, peut-être plus flexibles, mais moins pratiques pour le débutant...

A: Utilisation de printf

La fonction printf permet d'afficher des chaînes de caractères formatées ; cela signifie que nous pourrons lui demander d'insérer des valeurs au milieu de la chaîne de caractère, et que nous pourrons préciser comment les formater, c'est-à-dire combien de chiffres afficher au minimum, afficher ou non le signe, ou encore utiliser du binaire ou de l'hexadécimal plutôt que du décimal.

Notons que cette fonction est ANSI, ce qui signifie que tous les compilateurs qui se disent respectant la norme ANSI-C la fournissent ; la norme GNU-C suivie par GCC étant une extension à l'ANSI, il aurait été étonnant que TIGCC ne la fournisse pas. remarquez qu'il peut être bon que vous reteniez comment utiliser cette fonction : si vous êtes un jour amené à programmer sous un autre compilateur que TIGCC, il est quasiment certain que vous la retrouverez (je n'ai jamais rencontré un compilateur C ne la définissant pas !).

printf est une fonction admettant un nombre variable d'arguments ; le premier, obligatoire, est une chaîne de caractère, qui contient ce que l'on veut afficher, plus des sortes de "codes" permettant de définir les valeurs à insérer. Les suivants (qui peuvent être au nombre de 0, 1, ou plus) correspondent aux valeurs à insérer dans la chaîne à l'affichage.
Notez que le nombre d'arguments suivant la chaîne de caractères doit être égal au nombre de codes dans celle-ci ! (S'il y a plus d'arguments que de codes, les arguments en trop seront ignorés ; s'il n'y en n'a pas assez, le résultat est indéfini, ce qui, pour dire les choses clairement signifie qu'il y a un risque non négligeable de plantage. Pour résumer, respectez le nombre d'arguments attendu, c'est la meilleure chose à faire.).

Par exemple, pour afficher une chaîne de caractères, sans insérer la moindre valeur, on pourra utiliser cette syntaxe :

```
printf("Hello World !");
```

Je ne vais pas reproduire ici la liste de toutes les options de formatage possibles, cela rallongerait inutilement ce chapitre, puisqu'elles sont toutes fournies, ainsi que leurs explications, dans la documentation de TIGCC, à l'entrée printf.

Sachez juste que les options de formatage commencent par un caractère '%'. La suite de caractère '\n' (antislash, accessible sur un clavier azerty par la combinaison de touches alt-gr + 8) permet un retour à la ligne.

Notez que, lorsque nous utiliserons printf, nous emploierons la fonction clrscr pour effacer l'écran (voir partie suivante pour plus d'explications).
Sachant que l'option de formatage permettant d'afficher un entier (signed short, pour être précis) est %d, voici un exemple :

```
// C Source File
// Created 08/10/2003; 14:17:20

#include <tigcclib.h>

// Main Function
void _main(void)
{
    clrscr();
    printf("la valeur est : %d !", 100);
    ngetchx();
}
```

Exemple Complet

Lorsque nous exécuterons ce programme, nous aurons à l'écran le message suivant : "La valeur est : 100 !".

Sachant qu'il est possible de remplacer une valeur par une variable contenant une valeur, que nous avons vu comment créer et initialiser des variables, et que l'option de formatage permettant d'afficher un flottant est %f, voici un autre exemple :

```
// C Source File
// Created 09/10/2003; 12:33:35

#include <tigcclib.h>

// Main Function
void _main(void)
{
    float a = 13.456;
    clrscr();
    printf("la valeur est : %f !", a);
    ngetchx();
}
```

Exemple Complet

A l'écran sera affiché... ce à quoi nous pourrions nous attendre.

B: Remarque au sujet de l'importance de la case

Comme vous avez pu le remarquer, nous avons ici utilisé la fonction clrscr, alors que, quelques chapitre auparavant, nous avions employé le ROM_CALL ClrScr (notez l'absence de majuscules dans le premier cas, et leur présence dans le second !).

Le ROM_CALL ClrScr permet, comme nous l'indique la documentation de TIGCC, d'effacer l'écran.
La fonction clrscr, qui n'est pas intégrée à la ROM, et qui prend donc un peu de place dans le programme, à partir du moment où on l'utilise, fait plus que cela : elle réinitialise la position d'écriture utilisée par printf au coin supérieur gauche de l'écran, soit, au pixel de coordonnées (0 ; 0).

Pour avoir la preuve de cette différence, essayez ce programme :

```
// C Source File
// Created 09/10/2003; 12:33:35

#include <tigcclib.h>

// Main Function
void _main(void)
{
    clrscr();
    printf("Salut");
    ngetchx();
    clrscr();
    printf("toi.");
    ngetchx();
}
```

Exemple Complet

Et, maintenant, faites la même chose, en remplaçant les deux appels à clrscr par des appels à ClrScr. Comme vous pouvez le constater, la différence est... visible.

Certes, vous pourriez être tenté d'utiliser toujours clrscr, et jamais ClrScr... Cela dit, clrscr effectuant plus d'opérations, il est possible qu'elle soit plus lente... et, puisqu'elle n'est pas inclue à la ROM, il est évident qu'elle fera grossir la taille du programme.
Pour résumer mes pensées, utilisez ce dont vous avez besoin ; ni plus, ni moins. Si vous avez besoin d'utiliser printf, il est fort probable que vous souhaitiez réinitialiser la position d'affichage... Dans ce cas, utilisez clrscr. Si vous ne comptez pas utiliser printf, autant appeler ClrScr, qui est tout aussi adaptée, dans ce cas.

Je tenais juste à profiter de cette occasion pour prouver l'intérêt qu'il y a à prêter attention aux majuscules, afin que vous compreniez bien que ce que j'ai dit plus haut n'était pas du vent.

II:\ Utilisation de la valeur de retour d'une fonction

A: Généralités

Une fonction, quelle qu'elle soit, ROM_CALL, fonction définie par vous-même, ou inclue dans TIGCC, a un type de retour. Ce type de retour est précisé devant le nom de la fonction à sa déclaration.
Ce type de retour peut être void, ce qui signifie "vide", ou, plutôt, "néant", auquel cas la fonction de retournera pas de valeur, ou un autre type, parmi ceux que nous avons déjà vu ou ceux qu'il nous reste à étudier.

Par exemple, la fonction ClrScr, dont nous avons parlé juste au dessus, ne retourne rien : son prototype est, si l'on se fie à la documentation, le suivant :

```
void ClrScr(void);
```

La fonction ngetchx, elle, renvoie une valeur de type short, comme nous l'indique son prototype, ainsi défini :

```
short ngetchx(void);
```

Pour une fonction retournant quelque chose, il est possible de récupérer la valeur retournée dans une variable de même type que celle-ci, au moment où on l'appelle (ou, pour être plus précis, au moment où elle s'achève).
Pour cela, nous écrirons le nom de la variable, le symbole d'affectation, et l'appel de la fonction, comme nous aurions fait pour une simple valeur.

Notons que, comme nous l'avons déjà fait, il est possible d'appeler une fonction retournant une valeur sans chercher à utiliser cette valeur !

B: Exemple : ngetchx

Par exemple, pour la fonction ngetchx, nous pourrions agir ainsi :

```
short key;
key = ngetchx();
```

Nous déclarons une variable de type short, ce qui correspond au type de retour du ROM_CALL ngetchx, et, ensuite, nous appelons ngetchx, en précisant que sa valeur de retour doit être mémorisée dans cette variable.

Ensuite, il nous est possible d'afficher le code de cette touche, comme dans l'exemple ci-dessous, dont est tiré l'extrait que nous venons de citer :

```
// C Source File
// Created 08/10/2003; 14:17:20

#include <tigcclib.h>

// Main Function
void _main(void)
{
    clrscr();
    printf("Pressez une touche...");

    short key;
    key = ngetchx();

    printf("\nCode de la touche : %d.", key);

    ngetchx();
}
```

Exemple Complet

Le programme résultat va demander à l'utilisateur d'appuyer sur une touche ; une fois que ce sera fait, il affichera le code correspondant à cette touche.
La code touche renvoyé par ngetchx est généralement le même que celui renvoyé par GetKey en BASIC, mais il diffère pour quelques touches ; en particulier, pour les touches directionnelles, il me semble. D'ailleurs, pour ces touches-là, les codes sont inversés selon que le programme tourne sur une TI-89 ou sur une TI-92+ !

C: Remarque concernant l'imbrication d'appels de fonctions

Pour clore ce chapitre, nous finirons par remarquer qu'il est possible d'imbriquer des appels de fonctions.
Pour l'exemple que nous avons pris juste au-dessus, cela nous permettrait d'éviter l'utilisation de la variable key.
Cela dit, le code deviendra peut-être moins lisible... tout en regroupant mieux les parties logiques... question de goût, je suppose.
Toujours est-il qu'il est parfaitement possible d'écrire quelque chose de ce genre :

```
// C Source File
// Created 08/10/2003; 14:17:20

#include <tigcclib.h>

// Main Function
void _main(void)
{
    clrscr();
    printf("Pressez une touche...");

    printf("\nCode de la touche : %d.", ngetchx());

    ngetchx();
}
```

Exemple Complet

Le résultat sera exactement le même que celui obtenu avec l'écriture précédemment présentée.
Comme vous pouvez le remarquer, dans ce genre de situation, c'est l'appel de fonction le plus interne qui est effectué en premier, et l'on termine par le plus externe, puisque celui-ci à besoin du premier.

Voila un chapitre de plus d'achevé. Un chapitre assez simple, et assez facile à supporter, je pense, même s'il nous a permis d'apprendre les bases de l'utilisation d'une fonction extrêmement utile, ainsi que l'emploi de la valeur de retour d'une fonction, chose qu'il est nécessaire de maîtriser.
Je ne peux que vous encourager à consulter la documentation de printf, afin de découvrir les nombreuses options de formatage que cette fonction propose ; nous travaillerons sans nulle doute avec certaines d'entre-elles dans le futur.
Le chapitre prochain nous permettra d'apprendre à calculer en C, maintenant que nous sommes en mesure d'afficher le résultat d'une opération...

Chapitre 8

Opérateurs arithmétiques et bit à bit

Maintenant que nous savons ce que sont les variables, quels sont les différents types arithmétiques, que nous avons appris comment en déclarer et y mémoriser des données, et que nous sommes en mesure d'afficher le contenu d'une de ces variables, nous pouvons à présent étudier les opérateurs arithmétiques, et bits à bits, permettant de travailler avec, et sur, ces variables.

Tout d'abord, nous étudierons les opérateurs arithmétiques les plus souvent utilisés, ceux permettant d'additionner, diviser, ... Puis, nous verrons ceux qu'il est moins fréquent de rencontrer, mais dont l'utilité ne peut être niée. Naturellement, nous n'oublierons pas de faire mention des formes concises de ces opérateurs.
Ensuite, après un bref rappel sur la logique booléenne, nous étudierons les opérateurs bits à bits, et les opérateurs de décalage de bits.
Pour finir, nous résumerons les règles de priorité entre les différents opérateurs que nous aurons ici étudié.

I:\ Opérateurs arithmétiques

Puisque l'on dispose de variables à même de mémoriser des valeurs, il apparait comme nécessaire de pouvoir manipuler ces valeurs : les additionner, les soustraire, ou leur faire subir des traitements plus... exotiques.

A: Les cinq opérations

Le C utilise les mêmes opérateurs que les mathématiques pour les quatre opérations standards :

- Addition : +
- Soustraction : - (tiret ; touche 6 du clavier azerty)
- Multiplication : * (étoile, à gauche de la touche entrée, sur un clavier azerty)
- Division : / (slash)

En plus de ceux-ci, un cinquième opérateur est défini, qui permet de calculer le modulo, c'est-à-dire le reste de la division entière entre deux nombres. Le symbole utilisé pour cela est le pourcent : %

Ces cinq opérateurs sont des opérateurs binaires. Cela signifie qu'ils travaillent avec deux éléments, un à leur gauche, et un à leur droite.

Voici un petit exemple d'utilisation de ces opérateurs :

```
// C Source File
// Created 09/10/2003; 12:33:35

#include <tigcclib.h>

// Main Function
void _main(void)
{
    clrscr();

    short a = 10;
    short b = 20;
    printf("\n%d+%d=%d", a, b, a+b);
    printf("\n%d-%d=%d", a, b, a-b);
    printf("\n%d*%d=%d", a, b, a*b);
    printf("\n%d/%d=%d", a, b, a/b);
    printf("\n%d%%%d=%d", a, b, a%b); // Pour afficher un symbole '%',
                                      // il faut en mettre deux,
                                      // puisque '%' indique,
                                      // normalement, l'option
                                      // de formattage

    printf("\n%d+%d-50/2=%d", a, b, a+b-50/2);

    ngetchx();
}
```

Exemple Complet

Comme nous pouvons le remarquer, on peut utiliser ces opérateurs avec des variables, ou directement sur des valeurs ; cela revient au même. Cela dit, on utilise la plupart du temps des variables, contenant le résultat d'appels de fonctions, par exemple : un code statique, faisant toujours la même chose, ne servirait pas à grand chose, en dehors des exemples !

Naturellement, ces opérateurs peuvent être séparé des variables par des espaces, des tabulations, des retours à la ligne... enfin, tout ce qui vous semble améliorer la lisibilité, en particulier dans le cas d'expressions longues et complexes !

Les quatre opérateurs "standard" peuvent travailler aussi bien avec des entiers qu'avec des flottants (notez que la division de deux entiers est en fait une division entière : elle renvoie le quotient, arrondi à la valeur inférieure (par exemple, 7/4 renverra un entier valant 1, et non pas un flottant valant 1.75)).
L'opérateur modulo, lui, ne peut travailler qu'avec des nombres entiers.

Notez qu'une division, ou un modulo, par 0 a un résultat non défini par la norme ; sur nos calculatrices, une division par 0 entraîne un plantage.

B: Altération de la priorité des opérateurs

Comme en mathématiques, encore une fois, il est possible d'altérer la priorité des opérateurs, en utilisant des parenthèses. En premier sera évalué ce qui est dans les parenthèses les plus internes.
Par exemple, 3+2*2 donnera 3+(2*2), soit 3+4, soit 7
Alors que (3+2)*2 donnera 5*2, soit 10.
Il est, naturellement, possible d'imbriquer plusieurs niveaux de parenthèses.

D'ailleurs, lorsque vous travaillez avec des expressions complexes, je vous conseille d'utiliser des parenthèses pour clarifier votre code ; elles permettront qu'au premier coup d'oeil, on comprenne l'ordre d'évaluation, sans avoir à réfléchir sur les priorités d'opérateurs !

C: Forme concise des opérateurs

Il nous arrive parfois d'effectuer une opération sur une variable, et d'affecter le résultat de ce calcul dans cette variable...
Par exemple, nous pourrions penser à une écriture de ce type :

```
a = a+20;
```

Ceci peut être écrit de façon plus concise, en utilisant la syntaxe présentée ci-dessous :

```
a += 20;
```

En fait, la plupart des opérateurs arithmétiques admettent une syntaxe de ce genre :
Une expression de la forme "A = A opérateur (B);" équivaut à "A opérateur= B", mais à part le fait que A ne sera évalué qu'une fois.
Les cinqs opérateurs binaires +, -, *, / et % que nous avons ici étudié admettent cette forme, et les cinq opérateurs binaires de manipulation de bits que nous verrons plus bas au cours de ce chapitre l'admettent aussi. Notez que les opérateurs unaires, ceux ne travaillant que sur une seule variable, que nous allons voir juste au-dessous, ne peuvent pas utiliser une syntaxe concise de la sorte !

Encore une fois, j'insiste sur le fait que le C est un langage concis, et que cette particularité est particulièrement appréciée par ses utilisateurs ; il est donc certain que vous rencontrerez ces formes si vous parcourez des codes sources, et, donc, il serait utile que vous les reteniez...

D: Opérateurs arithmétiques unaires

Il existe deux opérateurs, en C, permettant de forcer le signe d'une valeur. Ce sont tous deux des opérateurs unaire, ce qui signifie qu'ils ne travaillent que sur une donnée.

Pour obtenir l'opposé d'une valeur, on utilise celle-ci :

```
-a;
```

Notez que cela revient à soustraire la valeur à 0... mais en l'écrivant de façon plus propre.

De façon symétrique, l'opérateur + unaire a été défini. Il renvoie la valeur de l'opérande sur laquelle il travaille.
Notez qu'il ne renvoie pas la valeur absolue ! Utiliser l'opérateur + unaire sur une valeur négative renverra... une valeur négative !

Voici un petit exemple illustrant l'utilisation de ces deux opérateurs, sur une valeur positive, et sur une donnée négative :

```
// C Source File
// Created 08/10/2003; 14:17:20

#include <tigcclib.h>

// Main Function
void _main(void)
{
    clrscr();

    short a = 10;
    short b = -20;

    printf("+a=%d", +a);
    printf("\n+b=%d", +b);

    printf("\n-a=%d", -a);
    printf("\n-b=%d", -b);

    ngetchx();
}
```

Exemple Complet

Si vous exécutez cet exemple, prêtez tout particulièrement attention à la seconde ligne affichée ! Et retenez en mémoire la remarque que j'ai fait juste avant de présenter ce code source !

E: Incrémentation, et Décrémentation

Lorsqu'il s'agit d'ajouter un à une valeur, c'est-à-dire de l'incrémenter, ou de lui retrancher un, c'est-à-dire la décrémenter, il est possible d'utiliser respectivement les opérateurs ++, ou -- (deux plus à la suite, ou deux moins à la suite).
Ces opérateurs peuvent être placés avant, ou après, leur opérande. Dans le premier cas, on parlera d'incrémentation préfixée, et, dans le second, d'incrémentation postfixée.

Lorsque l'on utilise un opérateur postfixé, la valeur de la variable nous est renvoyée, et, seulement ensuite, elle est incrémentée (ou décrémentée, selon le cas).
Lorsque l'on travaille avec un opérateur préfixé, la variable est incrémentée (ou décrémentée), et, ensuite, sa valeur nous est renvoyée.

Pour que vous compreniez bien ce qui se passe, et comment, voici un exemple :

```
// C Source File
// Created 08/10/2003; 14:17:20

#include <tigcclib.h>

// Main Function
void _main(void)
{
    clrscr();

    short a;

    a = 10;
    printf("1: a++ => %d", a++);
    printf("\n2: a = %d", a);

    a = 10;
    printf("\n3: a-- => %d", a--);
    printf("\n4: a = %d", a);

    a = 10;
    printf("\n5: a = %d", a);
    printf("\n6: ++a => %d", ++a);

    a = 10;
    printf("\n7: a = %d", a);
    printf("\n8: --a => %d", --a);

    ngetchx();
}
```

Exemple Complet

Ce que je vous recommande de faire est de l'exécuter, et de suivre en parallèle le résultat de son exécution et son code source.

F: A ne pas faire !

Il est tout à fait possible d'utiliser des écritures telles que celle-ci sans que le compilateur ne s'en offusque :

```
c = a++ + ++b;
```

Ou que celles-là :

```
c = a++ + +(++b);
c = a++ + + ++b;
c = a-- - -(--b);
```

Ces écritures nous montrent bien que l'on peut mettre un paquet de plus ou de moins à la suite... tout en sachant qu'il faut, par endroit, utiliser des espaces ou des parenthèses, pour que le compilateur soit à même de faire la différence entre les opérateurs d'incrémentation, d'addition, et de signe...
Cela dit, c'est assez peu lisible ! Je vous conseille donc de bannir totalement ce genre de syntaxe, et de préférer étaler ceci sur plusieurs lignes, que vous n'aurez pas de mal à relire ! Parce que pour comprendre ces expressions, c'est quelque peu complexe, et on ne peut s'empêcher d'hésiter quand à l'ordre dans lequel les opérations seront menées (notamment, a sera incrémenté avant, ou après de faire la somme ? Je dois reconnaître que je suis incapable de le dire avec certitude... si j'ai bonne mémoire, la norme ne le dit même pas, et on se retrouve dans la même situation que celle que je présente en dessous)

Pour l'écriture suivante, je suis sûr de moi : elle est totalement indéfinie par la norme (ce n'est pas la seule, mais c'est la plus évidente) :

```
c = a++ + a++;
```

L'ordre d'évaluation n'est pas clairement défini, et on ne sait pas dans quel ordre les incrémentations et la somme seront effectuées...
D'ailleurs, GCC vous préviendra que L'opération concernant a peut être indéfinie.
Dans un programme, c peut au final valoir une certaine valeur... et dans un autre programme, une autre valeur !
Ce type d'écriture, présentant ce genre d'effets de bords est à bannir !

II:\ Opérateurs travaillant sur les bits

En plus des opérateurs arithmétiques usuels, que nous utilisons tout le temps, le C définit des opérateurs travaillant sur les bits. Certains de ces opérateurs permettent de travailler bit à bit sur des valeurs ; d'autres permettent de décaler, vers la droite ou la gauche, les bits d'une donnée.
Nous commencerons cette partie par un bref rappel de logique booléenne, puis nous présenterons les opérateurs correspondants.

A: Rappels de logique booléenne

La logique booléenne n'utilise que deux valeurs, 0, et 1, qui correspondent tout à fait à ce que l'on emploie lorsque l'on travaille en binaire.
Trois opérations, voire quatre, d'algèbre de Boole nous seront utiles lorsque nous programmerons en C : le OU (or, en anglais), le OU exclusif (xor, ou eor, selon les habitudes du programmeur ; en C, on utilise généralement xor, alors qu'en Assembleur, on aura plus tendance à utiliser eor, par exemple ; mais ce sont deux appellations qui renvoient à la même chose), et le ET (and). En plus de cela, il est possible de définir le NON (not).

Voici les tables de ces opérations :

a	**b**		**a OR b**		**a XOR b**		**a AND b**		**NOT a**
0	0		0		0		0		1
0	1		1		1		0		1
1	0		1		1		0		0
1	1		1		0		1		0

a OR b vaut 1 si a, ou b, ou les deux, valent 1.
a XOR b vaut si a ou b, mais pas les deux à la fois, valent 1.
a AND b vaut 1 si a et b valent tous les deux 1.
Et NOT a vaut 1 si a vaut 0, et, inversement, 0 si a vaut 1.

B: Opérateurs bits à bits

Le C propose des opérateurs bits à bits permettant de faire ceci directement sur les bits composant une, ou deux, valeurs, selon l'opérateur. Notez que ces opérateurs ne travaillent qu'avec des valeurs, on des variables, de type entier : ils ne peuvent pas être utilisés sur des flottants !

Pour effectuer un OR bit à bit, vous utiliserez l'opérateur | (le pipe sous les systèmes UNIX, accessible par Alt-Gr 6 sur un clavier azerty).
Pour le XOR, il convient d'utiliser le ^ (accent ciconflexe).
Pour le AND, c'est un & qu'il revient d'écrire (é commercial, touche 1 du clavier azerty).
Et enfin, pour le NOT, c'est le ~ (tilde, soit Alt-Gr 2).

Notez que |, ^, et & sont des opérateurs binaires, alors que le ~ est un opérateur unaire. Les trois premiers, en tant qu'opérateurs binaires, disposent d'une écriture abrégée pour les affectations, telle &=, par exemple.

Ci-dessous, un extrait de code source présentant quelques emplois de ces quatre opérateurs :

```
short a=10,
      b=20,
      c;
c = a | b;
c = a & b;
c = a ^ b;
c = ~a;
```

Je reconnais qu'il est assez rare pour un débutant d'employer ces opérateurs, mais ils sont souvent fort utiles pour certains types de programmes ; ne voulant pas avoir à revenir dessus plus tard, au risque de les oublier, j'ai préféré les présenter dans ce chapitre.

C: Opérateurs de décalage de bits

Un autre type permet de manipuler directement les bits d'une variable, ou d'une donnée. Ils s'agit des deux opérateurs de décalage de bits.

Le premier, >> (deux fois de suite le signe supérieur) permet de décaler les bits d'un nombre vers la droite, et le second, << (deux signes inférieur successifs), est sa réciproque, c'est-à-dire qu'il décale vers la gauche.

Ces opérateurs s'utilisent de la façon suivante :
valeur OP N
Où valeur est une donnée, une variable, une valeur, ..., et N le nombre de bits dont on souhaite décaler les bits de la donnée.
Notez que décaler de N bits vers la droite revient à diviser par 2 puissance N, et décaler de N bits vers la gauche correspond à une multiplication par 2 puissance N.

Voici un petit exemple, pour vous donner la syntaxe d'utilisation, en clair :

```
short a = 0b00011100;
short b;
b = a << 2;
// b vaut maintenant 0b01110000
b = a >> 4;
// b vaut maintenant 0b00000001
a <<= 3;
// a vaut maintenant 0b11100000
```

La remarque faite plus haut pour les opérateurs bits à bits est ici aussi valable.

III:\ Résumé des priorités d'opérateurs

Pour terminer ce chapitre, je pense que récapituler les priorités des opérateurs que nous avons ici vu peut être une très bonne chose...
C'est pour ce genre de choses qu'un livre utilisé comme référence peut être une bonne chose : même si aucun livre ne traite de la programmation en C pour TI, le C pour PC et le C pour TI sont le même langage ! Moi-même, malgré plusieurs années d'expérience en programmation en langage C, j'ai préféré me référer à un livre pour écrire cette partie, afin de ne pas écrire quoi que ce soit d'erroné.
Dans le doute, il demeure possible d'utiliser des parenthèses pour forcer un ordre de priorité ; je vous conseille d'en utiliser assez souvent, dès que les choses ne sont pas tout à fait certaines dans votre esprit, afin de rendre votre source plus clair, et plus facile à comprendre si vous êtes un jour amené à le relire ou à le distribuer...

Le tableau ci-dessous présente les différents opérateurs que nous avons vu (j'espère ne pas en oublier, ni en rajouter que nous n'ayions pas encore étudié). La ligne la plus haute du tableau regroupe les opérateurs les plus prioritaire, et les niveaux de priorité diminuent au fur et à mesure que l'on descend dans le tableau. Lorsque plusieurs opérateurs sont sur la même ligne, cela signifie qu'ils ont le même niveau de priorité.

Opérateurs, du plus prioritaire au moins prioritaire

```
()
~ ++ -- + - sizeof
* / %
+ -
<< >>
&
^
|
= += -= *= /= %= &= ^= |= <<= >>=
```

Notez que les opérateurs + et - unaires sont plus prioritaires que leurs formes binaires.
N'oubliez pas que le C n'impose pas l'ordre dans lequel les opérandes d'un opérateur sont évaluées, du moins, pour les opérateurs que nous avons jusqu'à présent vu ; le compilateur est libre de choisir ce qu'il considère comme apportant la meilleure optimisation. Il en va de même pour l'ordre d'évaluation des arguments passés à une fonction.

Maintenant que nous en avons terminé avec tous ces opérateurs, nous allons passer à un chapitre qui nous permettra d'étudier ce que sont les structures de contrôles, lesquelles sont proposées en C, et comment les utiliser.
Cependant, avant de pouvoir nous lancer dans le vif du sujet, il nous faudra étudier encore quelques nouveaux opérateurs, qui nous seront indispensables pour la suite...

Chapitre 9

Structures de contrôle (if, while, for, switch, ...)

Ce chapitre, le neuvième de ce tutorial, va nous permettre d'apprendre ce que sont les structures de contrôle, leur utilité, et, surtout, comment les utiliser.
Nous profiterons de ce chapitre pour parler des opérateurs de comparaison, qui sont à la base de la plupart des conditions.
En plus de cela, nous devrons, avant de pouvoir nous attaquer au sujet principal de ce chapitre, étudier deux autres nouveaux opérateurs, qui nous seront utiles pour combiner des conditions simples, pour on obtenir de plus complexes.

I:\ Quelques bases au sujet des structures de contrôle

Nous commencerons ce chapitre par une partie au cours de laquelle nous définirons grossièrement les structures de contrôle, puis au cours de laquelle nous étudierons les opérateurs de comparaison, pour finir par les opérateurs de logique booléenne.

A: Qu'est-ce que c'est, et pourquoi en utiliser ?

Un programme ayant un fonctionnement linéaire, un comportement déterminé précisément et invariant, n'est généralement pas très utile ; certes, cela permet d'effectuer un grand nombre de fois une tâche répétitive, simplement en lançant le programme le nombre de fois voulu... Mais, généralement, il faut que le programme s'adapte à des cas particuliers, à des conditions, qu'il exécute certaines fois une portion de code, d'autres fois une autre portion de code, qu'il soit capable de répéter plusieurs fois la même tâche sur des données successives...
Tout ceci nécessite ce qu'on appelle "structures de contrôle".

Le C propose plusieurs structures de contrôle différentes, qui nous permettent de nous adapter à tous les cas possibles. Il permet d'utiliser des conditions (structures alternatives), des boucles (structures répétitives), des branchements conditionnels ou non, ...
Certaines de ces structures de contrôle sont quasiment toujours utilisées ; d'autres le sont moins. Certaines sont très appréciées, d'autres sont à éviter...

Une fois la première partie de ce chapitre terminée, nous passerons à l'étude de ces différentes structures de contrôle.

B: Les opérateurs de comparaison

Lorsque l'on travaille avec des conditions, il s'agit généralement pour nous de faire des comparaisons.
Par exemple, "est-ce que a est plus petit que b ?" ou "est-ce que la touche appuyée au clavier est égale à ESC ?" ou encore "est-ce que l'expression _le_fichier_X_existe_ est égale à vrai ?".

En C, pour exprimer une comparaison, on utilise, tout comme en mathématiques, des opérateurs. Ces opérateurs sont quasiment tous des opérateurs binaires, ce qui signifie qu'ils prennent une donnée à gauche, une donnée à droite, et renvoie le résultat de la comparaison.
Le résultat de la comparaison est une valeur booléenne, notée TRUE (vrai) ou FALSE (faux). On associe à TRUE toute valeur non nulle, et à FALSE la valeur nulle, ce qui signifie que toute expression dont le résultat est différent de 0 sera considérée comme valant TRUE si on l'emploie dans une condition. Il en va de même, respectivement, pour FALSE et la valeur 0.

1: Egalité, Différence

Pour déterminer si deux expressions sont égales, on utilise l'opérateur == (deux fois le symbole égal à la suite, sans rien entre !).
Notez bien que l'opérateur = (égal seul) est utilisé pour les affectations, pas pour la comparaison ! C'est un fait que les débutants ont trop souvent tendance à oublier.

Pour déterminer si deux expressions sont différentes, on utilise l'opérateur != (un point d'exclamation, immédiatement suivi d'un symbole égal).

Par exemple,
- `10 == 20` renverra FALSE,
- `15 != 20` renverra TRUE,
- `12 == 12` renverra TRUE,
- et `14 != 14` renverra FALSE.

(Notez que ce ne sont que des exemples théoriques : en réalité, il est complètement absurde d'effectuer de telles comparaisons, et ces opérateurs permettent de faire plus que cela !)

2: Supérieur strict, Supérieur ou Égal

Il est aussi, naturellement, possible de comparer des données de façon à déterminer si l'une est plus grande que l'autre.
Pour une comparaison stricte, en excluant l'égalité, on utilise, comme en mathématiques, le symbole >.
Pour une comparaison non stricte, incluant le égal, on utilisera l'opérateur >= (constitué d'un symbole supérieur immédiatement suivi d'un symbole égal).

Par exemple,
- `10 > 20` renverra FALSE,
- `20 >= 20` renverra TRUE,
- `12 > 11` renverra TRUE,
- et `24 >= 34` renverra FALSE.

3: Inférieur strict, Inférieur ou Égal

Pour comparer des données de façon à déterminer si l'une est plus petite que l'autre, on agira exactement de la même manière, mais en utilisant les opérateurs < et <= selon que l'on veut une comparaison stricte ou non.

4: Négation

Il existe un dernier opérateur de comparaison, qui lui, est un opérateur unaire, ce qui signifie qu'il ne travaille qu'avec une seule donnée, celle que l'on place à sa droite. Il ne permet de faire une comparaison qu'avec 0.
Cela signifie que si son opérande, la donnée sur laquelle il travaille, vaut 0, la comparaison vaudra TRUE, et que si la donnée a une valeur non nulle, la comparaison vaudra FALSE.

Par exemple,
`!0` renverra TRUE,
`!10` renverra FALSE.

Notez que, comme nous l'avons dit plus haut, FALSE vaut 0, et que TRUE vaut une valeur non nulle... L'opérateur ! peut donc être utilisé pour obtenir la négation d'une condition...
Par exemple,
`!(10 > 20)` renverra TRUE,
`!(20 == 20)` renverra FALSE.

C: Les opérateurs logiques Booléens

Le C propose deux opérateurs binaires de logiques booléenne, qui permette notamment de regrouper plusieurs conditions, afin de former une condition plus complexe.
Ces opérateurs sont :

- OU inclusif, dont l'opérateur est `||` (deux barres verticales l'une à la suite de l'autre ; barres verticales accessibles par Alt-Gr + 6 sur un clavier azerty).
- ET, dont l'opérateur est `&&`

Ces deux opérateurs fonctionnent un peu comme les opérateurs bit à bit `|` et `&` que nous avons étudié au chapitre précédent, à cela près qu'ils ne travaillent pas avec les bits, mais avec des données dans leur intégralité.
Notez aussi qu'il n'existe pas d'opérateur OU exclusif de logique booléenne. Si vous en avez l'utilité, il vous faudra procéder "à la main", avec une succession de `&&` et de `||`.

Par exemple,
`TRUE || TRUE` renverra TRUE,
`TRUE || FALSE` renverra TRUE,
`FALSE || TRUE` renverra TRUE,
`FALSE || FALSE` renverra FALSE.
Ou bien :
`TRUE && TRUE` renverra TRUE,
`TRUE && FALSE` renverra FALSE,
`FALSE && TRUE` renverra FALSE,
`FALSE && FALSE` renverra FALSE.

Notez que les expressions de ce type sont évaluées de gauche à droite, et ne sont évaluées dans leur intégralité que si nécessaire.
Par exemple, dans l'expression `TRUE || FALSE`, le programme verra que le premier terme est à TRUE, et ne prendra donc pas la peine d'évaluer le second, puisqu'il suffit que l'un des deux soit à TRUE pour que toute l'expression soit à TRUE ; l'expression vaudra TRUE.
Respectivement, dans l'expression `FALSE && TRUE`, le programme verra que le premier terme est à FALSE, et n'évaluera pas le second, puisqu'il faut que les deux soient à TRUE pour que l'expression soit à TRUE ; celle-ci sera donc à FALSE.
Cela dit, ceci tient un peu du détail, et l'ordre d'évaluation n'est souvent pris en compte par les programmeurs que lorsqu'ils optimisent leur programme, par exemple, en plaçant l'expression dont le résultat est le plus intéressant en premier. Le résultat global est le même, que vous placiez les opérandes dans un ordre, ou dans l'autre !

D: Résumé des priorités d'opérateurs

Exactement comme nous l'avons fait en fin de chapitre précédent, nous allons, puisque nous venons d'étudier plusieurs nouveaux opérateurs, dresser un tableau résumant leurs priorités respectives. Tout comme pour le tableau du chapitre précédent, la ligne la plus haute du tableau regroupe les opérateurs les plus prioritaire, et les niveaux de priorité diminuent au fur et à mesure que l'on descend dans le tableau. Lorsque plusieurs opérateurs sont sur la même ligne, cela signifie qu'ils ont le même niveau de priorité.

Opérateurs, du plus prioritaire au moins prioritaire

```
()
! ~ ++ -- + - sizeof
* / %
+ -
<< >>
< <= > >=
== !=
&
^
|
&&
||
= += -= *= /= %= &= ^= |= <<= >>=
```

Notez que les opérateurs + et - unaires sont plus prioritaires que leurs formes binaires.

II:\ Structures conditionnelles

Maintenant que nous avons appris ce dont nous avions besoin pour écrire des conditions, nous allons l'appliquer. Nous commencerons par apprendre à utiliser des structures de contrôle conditionnelles, aussi appelées alternatives, du fait que leur forme généralisée permet de faire un choix entre plusieurs portions de code à exécuter.

A: if...

La forme la plus simple de structure conditionnelle est d'exécuter quelque chose dans le cas où une condition est vraie.
Pour cela, on utilisera la structure de contrôle if, dont la syntaxe est la suivante :

```
if ( condition )
    Instruction à exécuter si la condition est vraie.
```

Par exemple, pour afficher "a vaut 10" dans le cas où la variable a contient effectivement la valeur 10, on utilisera le code suivant :

```
if(a == 10)
    printf("a vaut 10");
```

Notez que l'indentation (le fait que l'instruction à exécuter si la condition est vraie soit décallée vers la droite) n'influe en rien le comportement du programme : on peut tout à fait ne pas indenter. Cela dit, indenter correctement permet de rendre le code source beaucoup plus lisible ; je vous conseille de toujours prendre le soin d'indenter votre source !
Sachant qu'un bloc peut prendre la place d'une instruction, on pourra exécuter plusieurs commande dans le cas où une condition est vraie, simplement en les regroupant à l'intérieur d'un bloc, c'est-à-dire en les plaçant entre une accolade ouvrante, et une accolade fermante. Notez que cette remarque est vraie dans tous les cas où l'on attend une instruction ; en particulier, ceci est vrai pour les structures de contrôle que nous verrons dans la suite de ce chapitre.
Voici un exemple en utilisant un bloc à la place d'une seule instruction :

```
if(a == 10)
{
    clrscr();
    printf("a vaut 10");
    ngetchx();
}
```

Dans ce cas, si la variable a contient la valeur 10, l'écran sera effacé, on affichera un message, et on attendra une pression sur une touche.

B: if... else...

Il est souvent nécessaire de pouvoir exécuter une série d'instructions dans le cas où une condition est vraie, et d'exécuter une autre série d'instructions dans le cas contraire... Bien évidemment, on pourrait utiliser une construction de cette forme, en utilisant l'opérateur de logique booléenne de négation :

```
if ( condition )
    Instruction à exécuter si la condition est vraie.
if ( !condition )
    Instruction à exécuter si la condition est fausse.
```

Cela dit, le langage C fourni une forme de structure de contrôle alternative qui rend les choses plus simples et plus lisibles ; elle a la syntaxe qui suit :

```
if ( condition )
    Instruction à exécuter si la condition est vraie.
else
    Instruction à exécuter si la condition est fausse.
```

Ainsi, on ne teste même qu'une seule fois la condition, alors qu'on l'évaluait deux fois si l'on utilise l'idée donnée juste au-dessus (idée qui est à bannir, j'insiste !).

Par exemple, pour afficher un message dans le cas où une variable contient la valeur 10, et un autre message dans tous les autres cas, on peut utiliser l'extrait de code-source qui suit :

```
if(a == 10)
    printf("a vaut 10");
else
    printf("a est différent de 10");
```

C: if... else if... else...

On peut aussi être amené à vouloir gérer plusieurs cas particuliers, et un cas général, correspondant au fait qu'aucun des autres cas n'a été vérifié. Pour cela, nous emploierons la structure de contrôle dont la syntaxe est la suivante, et qui est la forme la plus complète de if :

```
if ( condition1 )
    Instruction à exécuter si la condition1 est vraie.
else if ( condition2 )
    Instruction à exécuter si la condition2 est vraie.
else if ( condition3 )
    Instruction à exécuter si la condition3 est vraie.
...
...
else
    Instruction à exécuter si les condtions 1, 2, et 3 sont toutes les trois fausses.
```

Avec une telle structure de contrôle, on n'évalue une condition que si toutes les précédentes sont fausses, et on finit par le cas le plus général, qui est celui où toutes les conditions énoncées précédemment sont fausses.

Notez que le cas `else` est optionnel : il est parfaitement possible de définir une structure alternative de ce type sans mettre de cas par défaut.
Remarquez aussi que l'on peut positionner autant de `else if` que l'on veut. La forme que nous avons vu au point précédent de ce chapitre n'est rien de plus qu'un cas où l'on n'emploie pas de `else if` !

Finissons cette partie concernant les structures de contrôle conditionnelles par un dernier exemple, utilisant la dernière forme que nous venons de définir, la plus complète des trois :

```
if(a == 10)
    printf("a vaut 10");
if(a == 15)
    printf("a vaut 15");
if(a == 20)
    printf("a vaut 20");
else
    printf("a est différent de 10, de 15, et de 20");
```

Si vous avez compris ce qui précède, cet exemple n'a pas besoin d'explications ; dans le cas contraire, je ne peux que vous inciter à relire ce que nous avons étudié plus haut...

D: Opérateur (condition) ? (vrai) : (faux)

Le C fournit aux programmeurs un opérateur permettant d'obtenir une valeur si une condition est vraie, et une autre si la condition est fausse ; il s'agit de l'opérateur ?:, qui s'utilise comme ceci :

```
condition ? valeur_si_vrai : valeur_si_faux
```

Et tout cette expression prend la valeur valeur_si_vrai si la condition est vraie, ou la valeur valeur_si_faux si la condition est fausse.
Cela explique que le nom de cet opérateur soit "expression conditionnelle" : il se comporte véritablement comme une expression, lorsqu'il est utilisé dans un calcul.

Par exemple, pour affecter à une variable une valeur en fonction du résultat de l'évaluation d'une condition, on pourra utiliser ceci :

```
short a;
short result;

// on initialise a d'une façon ou d'une autre

result = (a==10 ? 100 : 200);
```

Notons que cette écriture n'est que la forme concise de celle-ci :

```
short a;
short result;

// on initialise a d'une façon ou d'une autre

if(a == 10)
    result = 100;
else
    result = 200;
```

Mais utiliser l'opérateur ?: permet d'utiliser une valeur en fonction d'une condition là où on ne pourrait pas mettre un bloc if...else.

Par exemple, l'opérateur ?: est extrêmement pratique dans ce genre de situation :

```
short a;
short result;

// on initialise a d'une façon ou d'une autre

result = 2*(a==10 ? 100 : 200);
```

Alors que si on avait voulu utiliser une construction if...else, il aurait fallu utiliser une variable temporaire, comme ceci :

```
short a;
short result;
short temp;

// on initialise a d'une façon ou d'une autre

if(a == 10)
    temp = 100;
else
    temp = 200;

result = 2*temp;
```

(Ou effectuer la multiplication dans le bloc correspondant au if, et dans celui correspondant au else, ce qui n'est pas toujours possible, et qui grossirait inutilement notre programme.)

L'opérateur ?: est, puisqu'il fonctionne avec trois opérandes, un opérateur "ternaire".
Cela dit, étant donné que c'est le seul opérateur de ce genre que fournit le C, on a généralement tendance à l'appeler "l'opérateur ternaire", plutôt que "expression conditionnelle".

Et, puisque nous venons de voir un nouvel opérateur, voici la liste des priorités d'opérateurs, mise à jour en le prenant en compte :

Opérateurs, du plus prioritaire au moins prioritaire

```
()
! ~ ++ -- + - sizeof
* / %
+ -
<< >>
< <= > >=
== !=
&
^
|
&&
||
?:
= += -= *= /= %= &= ^= |= <<= >>=
```

Notez que les opérateurs + et - unaires sont plus prioritaires que leurs formes binaires.

III:\ Structures itératives

Le C présente trois types de structures de contrôle itératives, c'est-à-dire, de structures de contrôle permettant de réaliser ce qu'on appelle des boucles ; autrement dit, d'exécuter plusieurs fois une portion de code, généralement jusqu'à ce qu'une condition soit fausse.
Le plus grand danger que présentent les itératives est que leur condition de sortie de boucle ne soit jamais fausse... dans un tel cas, on ne sort jamais de la boucle (à moins d'utiliser une opération d'altération de contrôle de boucle, que nous étudierons au cours de cette partie), et on réalise une "boucle infinie". La seule façon de sortir d'une boucle infinie est de force la mort du programme, si vous en avez la possibilité (par exemple, si un kernel permettant de tuer le programme courant est installé sur la calculatrice), ou de réinitialiser la calculatrice : le C ne propose pas une "astuce", contrairement au ti-basic et sa touche ON, permettant de quitter le programme de manière propre en cas de fonctionnement incorrect.
Nous verrons dans ce chapitre trois formes de répétitives ; pour chacune d'entre-elle, nous donnerons au minimum deux exemples, qui seront volontairement assez proche pour chaque structure, afin de montrer comment chacune permet de faire ce que font les autres... ou leurs différences !

A: while...

La première des itératives que nous étudierons au cours de ce chapitre est la boucle "`while`", appelée "tant que" en français, lorsque l'on fait de l'algorithmie.
Avec cette structure de contrôle, tant qu'une condition est vraie, les instructions lui correspondant sont exécutées.
Sa syntaxe est la suivante :

```
while ( condition )
    Instruction à effectuer tant que la condition est vraie.
```

Naturellement, ici aussi, de la même façon que pour les conditionnelles que nous avons étudié et que pour les itératives que nous verrons dans quelques instants, il est possible d'utiliser un bloc d'instructions entre accolades à la place d'une simple instruction.

Voici un premier exemple, qui va afficher la valeur de la variable a :

```
a = 0;
while(a <= 2)
{
    printf("a=%d\n", a);
    a++;    // Il ne faut pas oublier de changer la valeur de a !
            // (dans le cas contraire, a n'aurait jamais la valeur 12,
            //  et on serait dans un cas de boucle infinie !!!)
}
```

On aura à l'écran, dans l'ordre, 0, 1, et 2. Une fois la valeur 2 affichée, la variable a sera incrémentée, et vaudra 3 ; la condition de boucle deviendra alors fausse, et on n'exécutera plus le code correspondant à la boucle : le programme continuera son exécution après la structure de contrôle.

Ci-dessous, un second exemple ; essayez de le comprendre, et, surtout, de comprendre ce qu'il fera, avant de lire le commentaire qui le suit...

```
a = 0;
while(a > 10)
{
    printf("On est dans la boucle");
}
```

Ici, la variable a vaut 0, et ne peut donc pas être supérieure à 10. La condition de boucle est fausse. Etant donné que cette condition est, avec les boucles `while`, testée avant d'exécuter le code correspondant à la boucle, on n'exécutera jamais celui-ci, et n'affichera donc rien à l'écran. Naturellement, cet exemple est ridicule, puisque l'on fixe la variable a juste avant la répétitive, mais il montre un des principes de ce type de boucle. (Notez qu'un compilateur idéal détecterait que la condition est toujours fausse, et pourrait supprimer toute la structure répétitive du programme, puisqu'elle ne sert finalement à rien ! GCC le fait peut-être même parfois, je ne saurai dire)

B: do... while

La seconde structure de boucle, que nous allons maintenant étudier, est "`do...while`", que l'on pourrait, en français, appeler "faire... tant que".
Ici encore, les instructions constituant la boucle sont exécutées tant que la condition de boucle est vraie. Cela dit, contrairement à `while`, avec `do...while`, la condition est évaluée à la fin de la boucle ; cela signifie que les instructions correspondant au corps de la structure de contrôle seront toujours exécutées au moins une fois, même si la condition est toujours fausse !

La syntaxe correspondant à cette structure répétitive est la suivante :

```
do
    Instruction à exécuter tant que la condition est vraie.
while ( condition );
```

Reprenons l'exemple que nous avions utilisé précédemment, qui affiche les différentes valeurs que prend une variable au fur et à mesure qu'on l'incrémente, et qui quitte une fois que la variable en question atteint une certaine valeur, mais en utilisant une itérative de la forme `do...while`, cette fois-ci :

```
a = 0;
do
{
    printf("a=%d\n", a);
    a++;    // Il ne faut pas oublier de changer la valeur de a !
            // (dans le cas contraire, a n'aurait jamais la valeur 12,
            //  et on serait dans un cas de boucle infinie !!!)
}while(a <= 2);
```

Le résultat sera exactement le même que celui que nous avions précédemment obtenu, lorsque nous utilisions un `while`.

Par contre, si nous essayons de faire la même pour notre second exemple, en écrivant un code tel celui-ci :

```
a = 0;
do
{
    printf("On est dans la boucle");
}while(a > 10);
```

... nous pourrons constater que l'on rentre dans la boucle, alors que la condition n'est pas vraie... Ce qui nous montre bien que la condition de boucle est testée en fin de boucle, et que les instructions correspondant à celle-ci sont toujours exécutées, au moins une fois.
Certes, dans un cas tel celui-ci, c'est plus un inconvénient qu'autre chose... Mais il est des cas lorsque l'on programme, où ce comportement correspond à ce que l'on recherche, et, dans ces occasions, il est plus simple d'utiliser un `do...while` qu'un `while` !

C: for

La troisième, et dernière, structure de contrôle itérative est celle que l'on appelle boucle "`for`". Elle est généralement utilisée lorsque l'on veut répéter un nombre de fois connu une action.
Sa syntaxe générale est la suivante :

```
for ( initialisation ; condition ; opération sur la variable de boucle )
    Instruction à exécuter tant que la condition est vraie.
```

Les trois expressions que j'ai nommé intialisation, condition, et opération sur la variable de boucle sont toutes trois optionnelle ; si aucune condition n'est précisée, le compilateur supposera que la condition est toujours vraie. Les points-virgule, eux, par contre, sont obligatoires !
Les noms que j'ai employé ne sont que purement indicatifs, mais ils correspondent à l'usage que l'on fait généralement de la boucle for, à savoir répéter un nombre de fois connu une suite d'instructions. Pour cela, il faut :

- Disposer d'une variable de boucle, qui sera considérée comme un compteur du nombre de fois dont on est passé dans la boucle.
- Initialiser cette variable, ce que l'on fait grâce à la première expression du for.
- Avoir une condition de boucle : une fois cette condition devenue fausse, on cessera de boucler. En règle générale, il est recommandé que cette condition porte sur la variable de compteur !
- Modifier la valeur de la variable de boucle utilisée comme compteur.

Par exemple, nous pourrions ré-écrire, une fois de plus, notre premier exemple, qui correspond tout à fait au cas d'utilisation le plus courant d'une boucle for, de la forme suivante :

```
for(a=0 ; a<=2 ; a++)
{
    printf("a=%d\n", a);
}
```

Le fonctionnement de ceci est tout simple : on initialise à 0 notre variable, on vérifie qu'elle est inférieure ou égale à deux, on affiche sa valeur, on l'incrémente, on vérifie qu'elle est inférieure ou égale à deux, on affiche sa valeur, on l'incrémente, ...
L'initialisation se fait une et une seule fois, au tout début, avant de rentrer dans la boucle ; la condition est évaluée en début de boucle, exactement comme pour `while`, et l'opération sur la variable de fin de boucle est effectuée en fin de boucle, avant de boucler.

Pour vérifier que le test de la condition se fait avant le passage dans la boucle, vous pouvez essayer avec l'exemple qui suit :

```
for(a=0 ; a>10 ; a--)
{
    printf("On est dans la boucle");
}
```

... et vous constaterez que l'on n'affiche pas de message à l'écran.

Voyons à présent trois exemples, réalisant exactement la même chose que le premier, mais en n'ayant pas mis certaines des trois expressions de l'instruction for ; ces instructions ont été sorties du for, placées soit avant celui-ci, soit dans la boucle, afin de montrer clairement où est-ce que le for les place dans le cas où on le laisse faire (ce que je ne peux que conseiller !)
Tout d'abord, en sortant l'initialisation :

```
a = 0;
for( ; a<=2 ; a++)
{
    printf("a=%d\n", a);
}
```

Ou en plaçant l'opération sur la variable de boucle en fin de boucle :

```
for(a=0 ; a<=12 ; )
{
    printf("a=%d\n", a);
    a++;
}
```

Ou encore en ne laissant que la condition :

```
a = 0;
for( ; a<=2 ; )
{
    printf("a=%d\n", a);
    a++;
}
```

Dans ce dernier cas, on revient exactement à une itérative de type `while`, qu'il vaut mieux utiliser, pour des raisons de facilité de compréhension...

Notez que la forme où on ne place ni initialisation, ni condition, ni opération, est souvent utilisée comme boucle infinie, dont il est possible de se sortir en utilisant certaines instructions d'altération de contrôle de boucle, que nous allons voir très bien ; pour illustrer ce propos, voici la syntaxe correspondant à une boucle infinie :

```
for ( ; ; )
    Instruction à exécuter sans cesse.
```

Naturellement, utiliser une structure de type "`while`" ou "`do...while`" avec une valeur non nulle à la place de la condition revient exactement au même ; mais c'est cette forme qui est censée être utilisée.

D: Instructions d'altération de contrôle de boucle

Le langage C propose plusieurs instructions qui permettent d'altérer le contrôle de boucles itératives, soit en forçant le programme à passer à l'itération suivante sans finir d'exécuter les instructions correspondant à celle qui est en cours, soit en forçant le programme à quitter la boucle, comme si la condition était fausse.
C'est dans cet ordre que nous étudierons ces instructions d'altération de répétitives.

1: continue

L'instruction `continue` permet de passer au cycle suivante d'une boucle, sans exécuter les instructions restantes de l'itération en cours.
Considérez l'exemple suivant :

```
for(a=0 ; a<=5 ; a++)
{
    if(a == 3)
        continue; // On passe directement à l'itération suivante,
                  // sans effectuer la fin de la boucle cette fois-ci.
                  // Donc, lorsque a vaudra 3, on n'affichera pas
                  // sa valeur.
    printf("a=%d\n", a);
}
```

Lorsque a sera différent de 3, l'appel à printf permettra d'afficher sa valeur. Mais, lorsque a vaudra 3, on exécutera l'instruction `while`. On retournera immédiatement au début de la boucle, en incrémentant a au passage.

Naturellement, pour un cas de ce genre, il serait préferable d'utiliser quelque chose de plus simple, et de plus explicite, dans le genre de ceci :

```
for(a=0 ; a<=5 ; a++)
{
    if(a != 3)
        printf("a=%d\n", a);
}
```

Notez que l'instruction `continue` rompt la logique de parcours de la boucle, et rend le programme plus difficilement compréhensible ; elle est donc à éviter autant que possible, et ne devrait être utilisée que lorsque c'est réellement la meilleure, voire la seule, solution possible !

2: break

L'instruction `break`, elle, met fin au parcours de la boucle, sitôt qu'elle est rencontrée, comme si la condition d'itération était devenue fausse, mais sans même finir de parcourir les instructions correspondant au cycle en cours.
Etudions l'exemple qui suit :

```
for(a=0 ; a<=5 ; a++)
{
    if(a == 3)
        break;  // Lorsque a vaut 3, on met fin à la répétitive,
                // sans même terminer la boucle courante.
                // On n'affichera donc, avec cet exemple,
                // que 0, 1, et 2 ; c'est tout.
    printf("a=%d\n", a);
}
```

Ce programme affichera la valeur de a lorsque a est inférieur à 3. Lorsque a vaudra 3, on exécutera l'instruction `break`... On quittera alors la boucle, sans même afficher la valeur 3, puisque l'appel à printf suit le `break`.

Ici encore, nous avons choisi un exemple extrêmement simple, qui pourrait être écrit de manière plus judicieuse (Si l'on ne veut pas aller jusque 3, pourquoi est-ce qu'on ne limiterait pas à 3 la valeur de a en condition d'itération ?), comme suit :

```
for(a=0 ; a<3 ; a++)
{
    printf("a=%d\n", a);
}
```

De la même façon que pour `continue`, `break` rompt la logique de parcours de la boucle... Cette instruction est donc elle aussi à éviter, dans la mesure du possible.

Un peu plus loin dans ce chapitre, nous parlerons de l'instruction `return`, qui peut aussi être utilisée de façon à altérer le contrôle de boucle, mais qui est d'un usage plus général.

IV:\ Structure conditionnelle particulière

Le langage C présente une structure conditionnelle particulière, le `switch`.

Cette structure est particulière dans le sens où elle ne permet que de comparer une variable à plusieurs valeurs, entières.

```
switch(nom_de_la_variable)
{
    case valeur_1:
            Instructions à exécuter dans le cas où la variable vaut valeur_1
        break;
    case valeur_2:
            Instructions à exécuter dans le cas où la variable vaut valeur_2
        break;
    default:
            Instructions à exécuter dans le cas où la variable vaut une valeur autre
que valeur_1 et valeur_2
        break;
}
```

Une structure `switch` peut avoir autant de case que vous le souhaitez. Le cas '`default`' est optionnel : si vous le mettez, les instructions lui correspondant seront exécutée si la variable ne vaut aucune des valeurs précisée dans les autres cas ; si vous ne le mettez pas et que la variable est différente des valeurs précisées dans les autres cas, rien ne se passera.
Je me permet d'insister sur le fait que `switch` ne permet de comparer une variable qu'à des valeurs ENTIERES ! Il est impossible d'utiliser cette structure conditionnelle pour comparer une variable à un flottant, par exemple !

Et voici un exemple d'utilisation de la structure conditionnelle `switch`, sans le cas `default` :

```
short a = 10;
switch(a)
{
    case 5:
            printf("a vaut 5\n");
        break;
    case 10:
            printf("a vaut 10\n");
        break;
    case 15:
            printf("a vaut 15\n");
        break;
}
```

Étant donné que la variable a vaut 10, et que l'on a un cas qui correspond à cette valeur, on affichera un message disant "a vaut 10".

A présent, si la variable ne correspond à aucune des valeurs proposées, toujours sans cas default :

```
short b = 20;
switch(b)
{
    case 5:
            printf("b vaut 5\n");
        break;
    case 10:
            printf("b vaut 10\n");
        break;
}
```

Ici, b vaut 20... mais on n'a aucun cas correspondant à cette valeur... On n'affichera donc rien à l'écran.

La même chose, avec un cas `default` :

```
short b = 20;
switch(b)
{
    case 5:
            printf("b vaut 5\n");
        break;
    case 10:
            printf("b vaut 10\n");
        break;
    default:
            printf("b ne vaut ni 5 ni 10\n");
        break;
}
```

Ici encore, aucun cas ne correspond de manière précise à la valeur 20... Puisque l'on a un cas default, c'est donc dans celui-ci qu'on se trouve, et on affichera un message disant que "b ne vaut ni 5 ni 10".

Notez que l'instruction `break`, que nous avons déjà étudié un petit peu plus haut dans ce chapitre, à la fin de chaque cas est optionnelle ; elle permet d'éviter que les cas suivants celui correspondant à la valeur de la variable soient exécutés, puisque le `break` permet de quitter une structure de contrôle. Si on ne le met pas, il se passera quelque chose dans le genre de ce que nous propose cet exemple :

```
short c = 5;
switch(c)
{
    case 5:
            printf("c vaut 5\n");
    // Volontairement, on omet ici le break !
    case 10:
            printf("c vaut 10\n");
        break;
    case 15:
            printf("c vaut 15\n");
        break;
}
```

Qu'est-ce qui se passe ici ?
La variable c vaut 5. On entrera donc dans le cas correspondant, et on affichera le message "c vaut 5". Cela dit, puisqu'on n'a pas d'instruction `break` à la fin de ce cas, on ne quittera pas la structure de contrôle ; on continuera donc à exécuter les instructions... du cas 10 ! Et on affichera aussi le message "c vaut 10" ! Une fois ceci fait, on parviendra à une instruction `break`, et on quittera la structure `switch`.
Il arrive parfois que l'on ne mette pas une instruction `break`... Parfois, on le fait volontairement, et cela correspond à ce que nous voulons faire... Mais, souvent, en particulier pour les débutants, c'est un oubli qui entraîne de drôles de résultats à l'exécution du programme ! Soyez prudents.

V:\ Branchement inconditionnel

Pour finir ce chapitre, nous allons rapidement parler des opérateurs de branchement inconditionnels. Un opérateur de branchement inconditionnel permet de "sauter" vers un autre endroit dans le programme, sans qu'il n'y ait de condition imposée par l'opérateur, au contraire des boucles ou des opérations conditionnelles, par exemple.

Les opérateurs `continue` et `break`, dont nous avons parlé plus haut, ont tout à fait leur place ici, même si nous avons choisi de les présenter dans le contexte où ils sont utilisés.

A: L'opérateur goto

Lorsque l'on parle d'opérateurs de branchement inconditionnels, le premier qui vienne généralement à l'esprit des programmeurs est le `goto`. Il permet de brancher sur ce qu'on appelle une "étiquette" (un "label", en anglais), déclaré comme suit :

```
nom_de_l_etiquette:
```

C'est à dire un identifiant, le nom de l'étiquette, qui doit être conforme aux normes concernant les noms de variables, suivi d'un caractère deux-points.

Et l'instruction `goto` s'utilise de la manière suivante :

```
goto nom_de_l_etiquette_sur_laquelle_on_souhaite_brancher;
```

Notez cependant que `goto` ne peut brancher que sur une étiquette placée dans la même fonction que lui. (Nous verrons au chapitre suivant ce que sont précisément les fonctions, notion que nous avons déjà eu l'occasion d'évoquer).

Voici un petit exemple simple utilisant un `goto` :

```
    printf("blabla 1\n");
    goto plus_loin;
    printf("blabla 2\n"); // Cette instruction ne sera
                          // jamais exécutée...
plus_loin:
    printf("blabla 3\n");
```

Comme vous pourrez le constater si vous essayez d'exécuter ce programme, le message "blabla 2" ne sera pas affiché... En effet, l'instruction `goto` sautera l'instruction lui correspondant...

Notez que j'ai l'habitude d'indenter les étiquettes d'un cran de moins que le reste du programme, afin de pouvoir les repérer plus facilement... Cette habitude me vient fort probablement de la programmation en langage d'Assembleur, et vous n'êtes nullement tenu de la respecter : comme je l'ai sûrement déjà dit, l'indentation ne sert à rien du point de vue du compilateur... Elle permet juste de rendre vos programmes plus lisibles ; à vous de déterminer quelles sont les habitudes d'indentation qui vous correspondent.

Pour finir, je me permet d'ajouter que l'utilisation massive de l'opérateur `goto` n'est pas vraiment une programmation propre. Je vous conseille ne l'utiliser que le plus rarement possible. En particulier, pensez à utiliser tout ce que nous avons déjà vu au cours de ce chapitre avant de vouloir utiliser `goto` !

En théorie, il est toujours possible de se passer de l'opérateur `goto`... même si, je le reconnais, dans certains cas (pour se sortir d'une triple boucle imbriquée ou d'une situation joyeuse dans ce genre, par exemple), il est bien pratique !

B: L'opérateur return

Pour finir ce chapitre, juste deux petits mots sur l'opérateur de branchement inconditionnel return.

Cet opérateur permet de quitter la fonction dans laquelle on l'appelle. Si la fonction courante est la fonction _main, alors, return quittera le programme.

Nous verrons probablement au chapitre suivant, traitant des fonctions, une utilisation plus générale de l'opérateur return, mais, en attendant, voici une façon de l'utiliser :

```
return;
```

Utilisé juste comme ceci, cet opérateur n'a pas une grande utilité... Mais nous verrons bientôt que ses possibilités sont supérieures à ce que nous présentons ici !

 hapitre 10

Ecriture, Appel, et Utilisation de fonctions

A présent, nous allons étudier les fonctions. Vous savez déjà comment en utiliser, puisque les ROM_CALLs ne sont rien de plus que des fonctions, mais nous n'avons pas encore réellement appris à en écrire, si ce n'est la fonction _main, qui est un cas particulier de fonctions.
Au cours de ce chapitre, nous verrons qu'elle est l'utilité d'écrire des fonctions, puis nous apprendrons à en écrire, et nous finirons par montrer comment on les appelle.

I:\ Mais pourquoi écrire, et utiliser des fonctions ?

Lorsque l'on programme, en C comme dans bien d'autres langages, il est fréquent d'avoir à utiliser plusieurs fois une portion de code dans un programme, ou, même, d'utiliser la même portion de code dans plusieurs programmes. Cela dit, nous ne voulons pas avoir à réécrire ce code à chaque fois ; ce serait d'ailleurs une ridicule perte de temps, et d'espace mémoire !
C'est pour cela que les notions de "fonction", ou de "procédure", ont été créées.
Qu'est-ce qu'une fonction ? Pour faire bref, on peut dire que c'est une portion de code, qui peut travailler sur des données que l'on lui fourni, qui peut renvoyer un résultat, et qui est souvent destinée à être utilisée plusieurs fois, par son créateur ou par quelqu'un d'autre, sans avoir à être ré-écrite à chaque fois.

Certaines fonctions, couramment appelées ROM_CALLs sont inclues à la ROM ; nous avons déjà appris comment les utiliser. D'autres sont inclues dans les bibliothèques partagées de TIGCC.
Toutes ces fonctions, vous pouvez les appeler sans avoir à les ré-écrire à chaque fois ; admettez que c'est bien pratique : imaginiez qu'il vous faille recopier plusieurs dizaines, voire centaines, de lignes de code C ou Assembleur, selon les cas, à chaque fois que vous voulez afficher un message, ou attendre qu'une touche soit pressée au clavier... rien qu'en pensant à cela, je suis certain que vous comprenez aisément l'utilité, et même la nécessité, des fonctions !
Vous pouvez aussi, et c'est le sujet de ce chapitre, écrire vos propres fonctions, que vous pourrez appeler tout à fait de la même manière que celles que nous avons jusqu'à présent utilisé. D'ailleurs, sans vraiment le savoir, vous en avez déjà écrit : `_main` est une fonction, obligatoire certes, mais une fonction tout de même !

Si vous lisez des documents traitants d'algorithmie, vous pourrez peut-être constater qu'ils font souvent, tout comme certains langages de programmation, une distinction entre ce qui est appelé "procédures", qui sont des portions de code effectuant un traitement à partir de une ou plusieurs données, et ne retournant aucun résultat et ce qui est appelé "fonctions", qui sont des portions de code effectuant elles aussi un traitement à partir de une ou plusieurs données, mais retournant un résultat. Le langage C ne fait pas de différence entre les notions de "fonctions" et de "procédures" : le terme de fonction est généralisé, et une fonction peut retourner quelque chose... ou rien. Lorsque l'on programme dans ce langage, le terme de procédure n'est donc que rarement employé.

Avant que l'on passe à l'écriture de fonctions, notez que ce qui est important pour une fonction, c'est de savoir ce qu'elle fait, à partir de quoi, et ce qu'elle renvoie : l'utilisateur de la fonction n'a pas à savoir comment elle effectue son traitement, ni par quel algorithme !
Si une fonction est bien pensée, il doit être possible de changer totalement son mode de fonctionnement sans avoir à changer ni ce qu'elle retourne, ni ce qu'elle attend comme données ; autrement dit, il doit rester possible d'utiliser la fonction exactement de la même manière, sans

même avoir à savoir que son mode de fonctionnement a été modifié. Ceci est d'autant plus vrai si vous avez l'intention de diffuser une fonction !

II:\ Écrire une fonction

Maintenant que nous avons vu l'utilité des fonctions, voyons comment en écrire.

A: Écriture générale de définition d'une fonction

En C, une fonction a toujours un type de retour, qui correspond au type du résultat qu'elle peut renvoyer et qui peut être n'importe lequel des types que nous avons précédemment étudié ; ce type de retour peut être `void` si on souhaite que la fonction ne renvoie rien. Elle a aussi un nom, qui respecte les conventions de nommage des variables (des lettres, chiffres, ou '_', en commençant par une lettre ou un underscore). Et, enfin, elle a une liste de zéro, un, ou plusieurs, paramètres.
Voici ce à quoi ressemble la définition d'une fonction :

```
type_de_retour nom_de_la_fonction(type_param_1 nom_param_1, type_param_2 nom_param_2)
{
    Contenu de la fonction
}
```

Comme je l'ai dit juste au-dessus, on peut n'avoir aucun paramètre, ou un, ou deux, ou autant qu'on veut. Ici, j'en ai mis deux, mais j'aurai pu en mettre un nombre différent.
Tant que nous en sommes à parler des paramètres, ils peuvent prendre pour type n'importe lequel de ceux que nous avons jusqu'à présent étudié et de ceux que nous étudieront plus tard.
Il est des gens qui utilisent le terme de "paramètre" lorsque l'on déclare la fonction et de "argument" lorsqu'on l'utilise ; d'autres personnes font exactement le contraire... pour simplifier, et n'ayant pas trouvé de norme à ce sujet, j'utilise les deux indifféremment.

B: Cas d'une fonction retournant une valeur

Une fonction déclarée comme ayant un type de retour différent de `void` doit retourner quelque chose. Pour cela, nous utiliserons l'instruction `return`, dont nous avons brièvement parlé au chapitre précédent.
Cette instruction, utilisée seule, permet de quitter une fonction (ou le programme, si la fonction est `_main`) ; si on la fait suivre d'une donnée, elle permet de quitter la fonction, en faisant en sorte que celle-ci renvoie la valeur précisée. Notez que le type de la donnée utilisée avec l'instruction `return` doit être le même que le type de retour de la fonction !

Lorsque l'on veut simplement quitter la fonction, on utilise cette écriture :

```
return;
```

Si l'on veut quitter une fonction renvoyant un entier, et que l'on souhaite retourner à l'appelant la valeur 150, on utilisera cette écriture :

```
return 150;
```

Notez que, pour une fonction dont le type de retour est `void`, qui, donc, ne renvoie rien, l'instruction `return` est optionnelle : une fois arrivé à la fin de la fonction, on retournera à l'appelant, même s'il n'y a pas d'instruction `return`.

Par contre, pour une fonction dont le type de retour est différent de `void`, il est obligatoire d'utiliser l'instruction `return`, en lui précisant quoi retourner. Dans le cas contraire, le compilateur vous signifiera qu'il n'est pas d'accord avec ce que vous avez écrit.

C: Quelques exemples de fonctions

Tout d'abord, commençons par un exemple de fonction ne prennant pas de paramètre (autrement dit, elle prend `void` en paramètre, c'est-à-dire néant), et qui ne retourne rien non plus. Cette fonction ne fait rien de plus qu'afficher un message.

```
void fonction1(void)
{ // Cette fonction ne prend pas de paramètre,
  // et ne renvoie rien
    printf("Ceci est la fonction 1");
}
```

Passons à une fonction qui prend deux entier, un `short` et un `long`, en paramètres, et qui affiche leurs valeurs... en appelant la fonction `printf`, fournie dans les bibliothèques partagées de TIGCC. Comme nous pouvons le remarquer, nous pouvons, au sein de la fonction, utiliser les paramètres tout à fait comme nous utilisons des variables ; en fait, les paramètres ne sont rien de plus que des variables, auxquelles on affecte une valeur au moment de l'appel de la fonction.

```
void fonction2(short param1, long param2)
{ // Cette fonction prend deux paramètres,
  // et ne renvoie rien
    printf("Param1 = %d,\nParam2 = %ld", param1, param2);
}
```

Un peu plus utile, une fonction qui prend en paramètre un entier, signé, sur 32 bits, qui calcule son carré, le stocke dans une variable, et retourne la valeur de cette variable :

```
unsigned long fonction_carre1(long a)
{ // Cette fonction calcule le carré du nombre
  // qui lui est passé en paramètre
    unsigned long result = a*a;
    return result;
}
```

A présent, exactement la même chose, sauf que, cette fois, on n'utilise plus de variable pour stocker le résultat : on retourne directement le résultat du calcul :

```
unsigned long fonction_carre2(long a)
{ // Cette fonction calcule le carré du nombre
  // qui lui est passé en paramètre
  // (Ecriture plus courte)
    return a*a;
}
```

Il est possible d'écrire des fonctions qui, tout comme `printf`, admettent un nombre variable d'arguments. Cela dit, il est rare d'avoir à écrire de telles fonctions, et cela est assez complexe. Nous ne traiterons donc pas ce sujet ici.

Naturellement, les fonctions utilisées ici en exemples sont volontairement très courtes, et ne contiennent que le code nécessaire à leur bon fonctionnement. En réalité, il serait ridicule d'utiliser des fonctions pour réaliser des tâches aussi courtes et simples, ne serait-ce du fait que l'on passe un petit peu de temps à appeler la fonction ! Remplacer une seule instruction par une fonction n'est généralement guère utile !
Cela dit, des exemples plus long n'auraient pas mieux montré les méthodes d'écriture de fonctions, et auraient risqué d'attirer votre attention sur des points qui n'entrent pas dans le sujet de ce chapitre, ce qui explique le choix que j'ai fait de ne montrer que des exemples brefs.

Pour finir, notez que même si appeler une fonction prend un certain temps, ce temps est vraiment minime, en particulier dans un langage fonctionnel tel le C, surtout à partir du moment où la fonction fait plus de quelques lignes. De plus, plus on appelle la fonction un grand nombre de fois, plus le gain en espace mémoire est réel et sensible.
Autrement dit, n'hésitez pas à utiliser des fonctions... sans toutefois en abuser dans des cas tels ceux que j'ai ici pris en exemple.

D: Notion de prototype d'une fonction

Pour en finir avec l'écriture des fonctions, et faire le lien avec leur utilisation, nous allons étudier la notion de prototype d'une fonction.
Un prototype de fonction permet au compilateur de savoir ce que la fonction attend en paramètre, et le type de ce qu'elle retournera, même si le code correspondant à la fonction est écrit ailleurs, ou plus loin dans le fichier source.

1: Pourquoi ?

Lorsque nous voulons appeler une fonction, il faut que le compilateur sache ce qu'elle retourne, et ce qu'elle prend en paramètre.
Puisque le compilateur lit les fichiers source du haut vers le bas, une solution, à laquelle on pense souvent lorsque l'on débute, est de déclarer toutes les fonctions avant leur utilisation... et, donc, de placer la fonction `_main` tout en bas du fichier source, en supposant que les fonctions ne s'appellent pas les unes les autres. Cela dit, cette méthode est loin d'être optimale ; en effet, elle ne fonctionnera pas si on a plusieurs fichiers sources, ou si les fonctions s'appellent les unes les autres. De plus, placer la fonction `_main` tout en fin de fichier ne facilite pas la lecture, puisque, nous aussi, nous avons tendance à lire de haut en bas, du début vers la fin.

C'est pour cela que le langage C introduit la notion de prototype de fonction.

2: Comment ?

Le prototype d'une fonction reprend exactement l'en-tête de la fonction, mais pas son corps, qui est remplacé par un point-virgule.
Un prototype a donc une écriture de la forme suivante :

```
type_de_retour nom_de_la_fonction(type_param_1 nom_param_1, type_param_2 nom_param_2);
```

Dans l'écriture d'un prototype de fonction, les noms de paramètres sont optionnels. Si vous donnez à vos paramètres des noms évocateurs (ce que je ne peux que vous inciter à faire !), il peut être bon de les conserver dans les prototypes, afin que vous sachiez à quoi correspond chaque paramètre... A vous de voir, encore une fois.

En règle générale, on place les prototypes de fonctions en début de fichier source, et les fonctions en fin... Ou même, on place les fonctions dans d'autres fichiers sources, lorsque l'on travaille sur un projet suffisamment important.

3: Exemples

Pour en finir avec les prototypes de fonctions, voici les prototypes des fonctions que nous avons précédemment utilisées en exemples :

```
void fonction1(void);
void fonction2(short param1, long param2);
unsigned long fonction_carre1(long a);
unsigned long fonction_carre2(long a);
```

III:\ Appel d'une fonction

Appeler une fonction est fort simple ; nous l'avons d'ailleurs déjà fait maintes fois pour des ROM_CALLs qui, même si elles n'ont pas été écrites par vous, ne sont rien d'autre que des fonctions. Cela explique pourquoi nous serons aussi bref dans cette partie.
Il vous faut écrire le nom de la fonction, suivi de, entre parenthèses, les différents paramètres, s'il y en a. Naturellement, on fait suivre d'un point-virgule si on est en fin d'instruction.

Juste pour la forme, puisque nous savons déjà comment faire, voici quelques exemples ; tout d'abord, des fonctions ne retournant rien :

```
fonction1();

short a = 12;
fonction2(a, 345);  // Il faut, naturellement, que le paramètre passé
                    // soit du type attendu par la fonction.
```

Et maintenant, utilisons des fonctions retournant quelque chose ; Si l'on souhaite récupérer le résultat renvoyé par une fonction, on peut tout à fait déclarer une variable du type de la donnée renvoyée par la fonction, et lui affecter le résultat renvoyé.
Ou alors, on peut utiliser directement le résultat renvoyé, que ce soit dans un calcul, dans un autre appel de fonction, ...
En somme, une fonction renvoyant un résultat de type X peut être utilisée partout où l'on peut utiliser directement une valeur, ou une variable, de type X.

```
unsigned long a;
a = fonction_carre1(2);

unsigned long b = fonction_carre1(fonction_carre1(4));
```

Utiliser une fonction n'est pas plus difficile que cela... Et cela nous a évité d'avoir à écrire deux fois le code contenu par la fonction.
(Même si, avec une fonction aussi courte que celle-ci, il aurait mieux valu se passer de fonction, et utiliser directement le code qu'elle contient... du moins, si l'on n'était pas dans un cas d'école)

IV:\ Quelques mots au sujet de la récursivité

Le langage C permet aux fonctions d'être récursives ; cela signifie qu'il est possible, en C, pour une fonction, de s'appeler elle-même.
Cela dit, il faut penser, à un moment ou à un autre, à inclure une condition permettant de stopper la récursion, afin de ne pas entrer dans une récursion infinie.

La récursion est parfois utilisée là où employer des boucles serait extrêmement compliqué.
En général, la récursion consiste à partir d'un état d'origine, à effectuer un traitement sur cet état, et de recommencer sur l'état obtenu.
En somme, la récursion sert souvent à illustrer la façon dont l'homme raisonne.

Si vous avez déjà suivi des cours d'algorithmie concernant les tris de données, vous avez fort probablement utilisé la récursion lorsque vous avez étudié le QuickSort (algorithme permettant de trier rapidement un tableau, dans la plupart des cas, en se basant sur le principe du "Diviser pour Régner"). Si vous avez déjà suivi des cours d'algorithmie avancée concernant les arbres, vous aurez utilisé la récursion pour parcourir un arbre (notez qu'il est probablement possible de parcourir un arbre avec des boucles... Mais je n'ose même pas imaginer à quel point cela doit être complexe par rapport à l'utilisation de la récursivité !).

Pour la forme, voici un bref exemple utilisant une fonction permettant de calculer la factorielle d'un nombre, de manière récursive

```
#include <tigcclib.h>

long factorielle(long a);

void _main(void)
{
    clrscr();
    printf("%ld", factorielle(5));
    ngetchx();
}

long factorielle(long a)
{
    if(a<=1) return 1;
    return a*factorielle(a-1);
}
```

Exemple Complet

Pour bien comprendre cet exemple, il faut se rappeler que la factorielle d'un nombre se défini comme étant le produit de ce nombre avec tous ceux qui lui sont inférieurs. Ainsi, 5! = 5*4*3*2*1.
Ceci pourrait se programmer avec une boucle de type `while`.
Cela dit, on peut aussi procéder ainsi :

5! = 5*4!
4! = 4*3!
3! = 3*2!
2! = 2*1

Avec ce raisonnement, on suit une logique récursive : pour calculer la factorielle d'un nombre, on calcule la factorielle de celui qui lui est inférieur, et on recommence... Tout en sachant que l'on doit s'arréter à 1. C'est exactement ce fonctionnement qui est utilisé par notre second exemple :
On commence par appeler la fonction factorielle pour calculer la factorielle de 5, et celle-ci s'auto-appelle jusqu'à ce qu'on arrive à 1.

Étudiez bien cet exemple, qui est une application idéale de la notion de récursivité.

V:\ Passage de paramètres par valeurs

En langage C, les paramètres de fonctions sont passés à celle-ci par valeur (on dit aussi parfois qu'ils sont passés par copie), ce qui signifie que la fonction appelée ne travaille pas sur les données mêmes que la fonction appelante lui a passé en paramètre, mais sur une copie de ces données, qui est propre à la fonction.
Cela implique qu'une fonction ne peut pas modifier des variables déclarées dans la fonction appelante.

Pour illustrer mes propos, prenons une fonction simple, qui prend en paramètre deux entiers, et essaye d'échanger leurs valeurs. Voici le coude-source correspondant :

```
#include <tigcclib.h>

void echange(short a, short b);

void _main(void)
{
    short a = 10;
    short b = 20;

    clrscr();

    printf("AVANT: a=%d ; b=%d\n", a, b);

    echange(a, b);

    printf("APRES: a=%d ; b=%d\n", a, b);

    ngetchx();
}


void echange(short a, short b)
{
    short temp;

    printf("DEBUT: a=%d ; b=%d\n", a, b);

    temp = a;
    a = b;
    b = temp;

    printf("FIN  : a=%d ; b=%d\n", a, b);
} // Fin echange
```

Exemple Complet

Si nous exécutons ce programme, nous aurons comme affichage :

```
AVANT: a=10 ; b=20
DEBUT: a=10 ; b=20
FIN  : a=20 ; b=10
APRES: a=10 ; b=20
```

Cela nous montre que, dans la fonction echange, les valeurs des deux variables a et b ont bien été échangées, mais que cela n'a eu absolument aucune influence sur les deux variables a et b de la fonction _main, qui avait appelé la fonction echange.
En effet, comme nous l'avons dit, les variables a et b de la fonction _main ont été passées par copie à la fonction echange, ce qui signifie que celle-ci n'a pas travaillé sur a et b de _main, mais sur ses propres variables a et b... que nous aurions d'ailleurs tout à fait pu appeler autrement sans que cela n'ait la moindre influence sur le déroulement du programme.

Au chapitre suivant, nous allons étudier ce que sont les pointeurs, et nous verrons qu'ils nous permettent de régler ce problème, en nous permettant de passer des paramètres à une fonction autrement que par valeur.

Chapitre 11

Les Pointeurs

Une des particularités extrêmement puissante du langage C est qu'il propose la notion de "pointeur". La notion de pointeur est souvent assez mal comprise par les débutants en programmation, et, à cause de cela, quelque peu redoutée.
Cela dit, utilisés judicieusement, les pointeurs rajoutent au langage C un grand nombre de fonctionnalités dont on ne peut pas se passer à partir du moment où on sait comment les utiliser, et qu'on commence à développer des programmes de taille non négligeable.
Au cours de ce chapitre, nous commencerons par voir ce que sont les pointeurs. Ensuite, nous verrons comment en créer et comment les utiliser pour quelques cas simples, et nous finirons par quelques exemples montrant leur utilité.

I:\ Qu'est-ce qu'un pointeur ?

Lorsque l'on jette un coup d'oeil dans un livre traitant de programmation en C au chapitre abordant le sujet des pointeurs (ce que je vous conseille de faire, naturellement), on constate qu'un pointeur est souvent représenté par une case et une flèche, qui indique une autre case, un peu comme ceci :

Si un jour, vous vous promenez dans la rue, que vous passez devant deux boulangeries côtes à côtes, et qu'un passant vous demande laquelle des deux fait le meilleur pain, vous allez en indiquer une des deux du doigt. Vous serez alors un pointeur, qui pointe vers l'une des deux boulangeries.

Pour parler en termes de langage informatique, sachez que chaque case mémoire peut être désigné par son adresse, c'est-à-dire, par son emplacement.
Un pointeur, tout simplement, est une case mémoire, qui contient l'adresse d'une autre case mémoire. Naturellement, l'adresse contenue par le pointeur peut changer, et pointer, selon le moment, sur une autre case mémoire.
De la sorte, en consultant une seule case mémoire, le pointeur, on peut, selon les circonstances, obtenir l'adresse d'autres cases mémoires nous intéressant.
En langage C, ces "cases mémoire" sont généralement appelées "variables". Un pointeur est donc une variable d'un type particulier, destinée à contenir non pas une valeur, mais l'adresse d'une autre variable.

Par exemple, si on a un pointeur nommé A, une variable entière, nommée X, qui contient la valeur X, et que A pointe sur X, on aura tendance à représenter cela de la sorte :

Dans la suite de ce chapitre, nous verrons comment ceci se représente en langage C, et quelle utilité cela peut avoir.

II:\ Déclaration d'un pointeur

Après cette introduction, nous allons voir comment, en langage C, déclarer des pointeurs.

Tout d'abord, il est à noter qu'il n'existe pas, en langage C, de "type pointeur", au sens où il y a un type int, un type float, ou d'autres types de ce genre.
Par contre, lorsque l'on crée une variable jouant le rôle de pointeur, il nous faut indiquer au compilateur vers quel type de données elle pourra pointer.
Ainsi, de la même façon qu'un float et un short sont différents, un pointeur déclaré comme pouvant pointer sur un short ne pourra pas pointer sur un float ; la réciproque étant, bien entendue, vraie ; et cela s'applique à tout type de variable.

Pour déclarer un pointeur sur une donnée de type donné, on procède comme si on voulait déclarer une variable de ce type là, mais en précédant le nom de celle-ci d'une étoile, comme ceci :

```
TYPE *nom_du_pointeur;
```

Par exemple, pour déclarer un pointeur, nommé 'a', qui puisse pointer sur un short, et un pointeur nommé 'b', qui puisse pointer sur un float, on écrira :

```
short *a;
float *b;
```

Le C permettant de mettre des espaces un peu où on le souhaite, il est naturellement possible d'écrire ceci :

```
short* a;
```

Cela dit, je vous conseille de préférer la première écriture que je vous ai proposé, à savoir, d'accoler l'étoile au nom de la variable pointeur.
En effet, lorsque vous serez amenés à déclarer plusieurs variables de type pointeur, il vous faudra vous souvenir que l'étoile indiquant que la variable que vous déclarez est un pointeur ne porte que sur la variable qu'elle précède.
Ainsi, si vous utilisez une des deux syntaxes suivantes :

```
short *a, b, c;   // seul a sera un pointeur !
short* a, b, c;   // Même chose !
```

Seul 'a' sera un pointeur sur une donnée de type short. b et c, en revanche seront des variables entières de type short, comme nous en avons utilisé dans les chapitres précédents.
Si vous voulez déclarer plusieurs pointeurs, il vous faut, j'insiste, précéder chacun de leurs noms par une étoile, comme ceci :

```
short *a, *b, *c;
```

Je vous conseille donc de prendre l'habitude, lorsque vous déclarez une variable de type pointeur, de toujours placer l'étoile immédiatement avant son nom, afin de ne pas croire que celle-ci affecte plus d'une variable, et, en même temps, afin de rendre votre code plus lisible.

III:\ Utilisation de base des pointeurs

Maintenant que nous savons comment déclarer des pointeurs, voyons comment les utiliser. Nous commencerons par voir comment faire pour obtenir l'adresse d'une variable, puis nous étudierons la méthode à suivre pour accéder à la donnée pointée par un pointeur.

A: Adresse d'une variable

Le C fournit un opérateur qui permet de récupérer l'adresse d'une variable, de façon à ce qu'on puisse la stocker dans un pointeur, qui pointera alors sur la variable donnée.
Il s'agit de l'opérateur unaire &, qui se se place avant le nom de la variable dont on souhaite obtenir l'adresse.

Par exemple, pour obtenir l'adresse de la variable a, on utilisera ceci :

```
&a
```

Maintenant que l'on sait comment obtenir l'adresse d'une variable, nous allons pouvoir mémoriser cette adresse dans un pointeur déclaré comme pouvant pointer sur ce type de variable, afin qu'il pointe effectivement vers quelque chose.
Pour cela, tout naturellement, nous utilisons l'opérateur d'affectation =, que nous avons déjà eu l'occasion d'étudier précédemment.
Par exemple, déclarons une variable de type float, et un pointeur sur float, affectons une valeur à notre première variable, et faisons pointer le pointeur sur celle-ci ; cela se fera de la manière suivante :

```
float b;
float *p2;

b = 3.14156;
p2 = &b;
```

Nous avons déjà expliqué que, tant qu'elle n'a pas été initialisée (c'est-à-dire, tant que nous ne lui avons pas encore affecté de valeur), une variable contient une valeur indéterminée, ce qui revient à dire qu'elle contient absolument n'importe quoi.
Il en est exactement de même pour les pointeurs, qui, finalement, ne sont rien de plus que des variables un peu particulières : tant qu'on ne leur a rien affecté, ils pointent sur une zone mémoire indéterminée. Prenez garde, essayer d'utiliser un pointeur qui n'a pas encore été initialisé et qui ne pointe donc vers "rien" est une source de bugs que l'on retrouve fréquemment, et qui, dès que le programme grossit un peu, est difficile à localiser.

Lorsque l'on utilise l'opérateur & en tant qu'opérateur unaire sur une variable, on dit qu'on référence cette variable, & étant alors l'opérateur de référence.
Il est à noter que cet opérateur ne peut s'appliquer qu'à des entités présentes en mémoire, c'est-à-dire, aux variables, et aux éléments de tableaux, que nous étudierons au chapitre suivant. Il ne peut, en particulier, pas s'appliquer à une expression, une constante, ou une variable déclarée comme register.

B: Valeur pointée

A présent, puisque nous savons comment obtenir un pointeur pointant sur une donnée, nous allons voir comment faire pour accéder à celle-ci, via le pointeur.

1: Valeur pointée, en C

Pour cela aussi, le C nous fournit un opérateur : l'étoile, que nous avions déjà utilisé pour déclarer un pointeur.
Pour obtenir l'adresse d'une variable, nous utilisions l'opérateur unaire & placé devant la nom de la variable dont nous souhaitions connaître l'adresse. Pour accéder à la valeur pointée par un pointeur, nous utiliserons l'opérateur * de la même façon, devant le nom du pointeur.

En utilisant la syntaxe *pointeur, nous avons accès à la donnée, aussi bien en lecture qu'en écriture, exactement comme si nous avions travaillé directement sur la variable pointée.
Par exemple, supposons que nous avons déclaré une variable de type entier, et un pointeur sur entier, de la façon suivante :

```
short a;
short *p;

a = 10;
p = &a;   // p pointe maintenant sur la variable a.
```

Nous pourrons mémoriser une valeur à l'adresse de la variable pointée par p, c'est-à-dire la variable a, de la façon suivante :

```
*p = 150;
```

Si nous essayons d'afficher la valeur contenue dans la variable a, nous constaterons que celle-ci est à présent 150, preuve que le pointeur p pointait bien vers a, et que nous pouvons y accéder de cette manière.
Et, naturellement, nous aurons accès à cette valeur en lecture, comme ceci par exemple :

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

Lorsque l'on applique l'opérateur * à un pointeur, on dit qu'on déréférence ce pointeur, l'opérateur * étant alors l'opérateur de déréférence, aussi parfois appelé opérateur d'indirection.

Comme on pourrait s'en douter, lorsque l'on déclare un pointeur comme pouvant pointer sur des données d'un certain type, la valeur obtenue par déréférencement de ce pointeur est du type pointé. Par exemple, avec un pointeur p déclaré comme pointant sur des données de type short, *p sera de type short.

2: Exemple illustré

Nous allons maintenant, en quelques schémas, illustrer ce que nous venons de présenter.
A chaque fois, nous ferons suivre la portion de code C par le schéma, et un bref commentaire décrivant ce que nous avons fait.

Tout d'abord, nous déclarons une variable de type short, nommée a, et un pointeur pouvant pointer sur des short :

```
short  a;
short  *p;
```

Ensuite, nous initialisons la variable a, en lui affectant la valeur 10 :

```
a  =  10;
```

Puis nous initialisons le pointeur p, en lui affectant l'adresse de la variable a :

```
p  =  &a;
```

Et enfin, nous affectons la valeur 150 à la zone mémoire pointée par le pointeur p, c'est-à-dire, étant donné ce que nous avons effectué juste au-dessus, à la variable a :

```
*p  =  150;
```

C: Pointeur NULL

Le standard C défini la constante NULL, qui correspond à l'adresse 0.
Par convention, l'adresse NULL étant considérée comme "impossible", cette valeur est donc considérée comme permettant de coder une erreur, ou le fait que le pointeur ne pointe sur rien.
Notamment, lorsque nous utilisons des fonctions retournant un pointeur, il est fréquent que celles-ci soient conçues de manière à retourner NULL en cas d'erreur.

Il est à noter que l'écriture vers un pointeur NULL est interdite, et causera un plantage de votre programme. Lorsqu'une fonction est susceptible de retourner NULL, ce qui est généralement indiqué dans sa documentation, il vous faut donc impérativement vérifier, avant d'utiliser le pointeur retourné, que celui-ci n'est pas NULL.

Lorsque l'on utilise la valeur NULL dans un contexte où on attendrait vrai ou faux, celle-ci est évaluée comme fausse.
Cela permet notamment des écritures comme celle-ci :

```
#include <tigcclib.h>

short *p;

// Travail avec p ; en particulier, on le fait pointer sur quelque chose.

if(!p)
{
     printf("p est NULL");
}
```

Au lieu de :

```
short *p;

// Travail avec p ; en particulier, on le fait pointer sur quelque chose.

if(p == NULL)
{
     printf("p est NULL");
}
```

Certes, cela ne fait que quelques caractères de moins dans le premier cas... Mais les programmeurs C ayant tendance à apprécier les formes concises, ils auront généralement tendance à éviter la seconde écriture.
En effet, pourquoi tester si "(p est NULL) est vrai ?", alors qu'il suffit de tester si "p est faux ?" ?

D: Exemple complet

Pour finir avec ces deux opérateurs, voici un petit exemple complet les mettant en oeuvre :

```
#include <tigcclib.h>

void _main(void)
{
    short a;
    short *p;

    clrscr();

    a = 150;
    p = &a;

    printf("a=%d ; *p=%d\n", a, *p);

    a += 10;
    printf("a=%d ; *p=%d\n", a, *p);

    *p += 40;
    printf("a=%d ; *p=%d\n", a, *p);

    ngetchx();
}
```

Exemple Complet

Ce petit programme affecte la valeur 150 à une variable entière, et déclare un pointeur sur celle-ci. Ensuite, nous ajoutons 10 à notre variable, puis 40, en y accédant, la seconde fois, par l'intermédiaire du pointeur.
Entre chaque opération, nous affichons la valeur de la variable, et la valeur pointée par le pointeur, ce qui nous permet de constater que nos manipulations ont le même effet, que nous accédions à la variable directement, ou par un pointeur pointant sur elle.

E: Résumé des priorités d'opérateurs

Exactement comme nous l'avons fait à chaque fois que nous avons vu de nouveaux opérateurs, nous allons, puisque nous venons d'en étudier deux nouveaux, dresser un tableau résumant leurs priorités respectives. Tout comme pour les tableaux des chapitres précédents, la ligne la plus haute du tableau regroupe les opérateurs les plus prioritaire, et les niveaux de priorité diminuent au fur et à mesure que l'on descend dans le tableau. Lorsque plusieurs opérateurs sont sur la même ligne, cela signifie qu'ils ont le même niveau de priorité.

Opérateurs, du plus prioritaire au moins prioritaire

```
()
! ~ ++ -- + - * & sizeof
* / %
+ -
<< >>
< <= > >=
== !=
&
^
|
&&
||
?:
= += -= *= /= %= &= ^= |= <<= >>=
```

Notez que les opérateurs + et - unaires présentés sur la seconde ligne sont plus prioritaires que leurs formes binaires présentes sur la quatrième ligne. Il en va de même pour les opérateurs & et * unaires que nous venons d'étudier, présents en seconde ligne, par rapport à leurs formes binaires vues précédemment, qui sont en troisième ligne.

IV:\ Quelques exemples d'utilisation

Pour terminer ce chapitre, nous allons dire quelques mots sur l'arithmétique de pointeurs, que nous utiliserons dès le chapitre prochain, et nous verrons un cas où nous n'avons pas d'autre solution que l'emploi de pointeurs.

A: Arithmétique de pointeurs

On appelle "arithmétique de pointeurs" le fait d'appliquer des opérateurs mathématiques sur des pointeurs.
Un pointeur contenant, en guise de valeur, une adresse, faire de l'arithmétique de pointeurs revient à effectuer des calculs portant sur des adresses mémoire.
Nous commencerons par constater que nous pouvons ajouter des entiers à des pointeurs, et nous finirons en parlant brièvement de la soustraction de pointeurs.

1: Ajouter un entier à un pointeur

Lorsque l'on dispose d'un pointeur sur un emplacement mémoire, il est possible, en lui ajoutant une valeur entière, de le faire pointer sur un des emplacements mémoire qui suit (ou qui précéde, si la valeur que l'on ajoute est négative).
Cela n'est pas extrêmement utile à savoir pour l'instant, mais nous utiliserons cette propriété des pointeurs dès le prochain chapitre, où nous verrons comment demander au compilateur de réserver des emplacements mémoire successifs.

Il est à noter que, lorsqu'on ajoute 1 à un pointeur, l'adresse sur laquelle il pointe n'est pas incrémenté de 1, mais du nombre d'octets correspondant au type pointé, c'est à dire, à 1*sizeof(TYPE).
Ainsi, si on incrémente de 1 un pointeur déclaré comme pointant sur des short, l'adresse ne sera pas incrémentée de 1, mais de 2, puisque les short sont codés, sur nos machines, sur deux octets.
Cela explique pourquoi, lorsque l'on veut travailler directement avec la mémoire, octets par octets, on utilise généralement des pointeurs déclarés comme pouvant pointer sur des char, qui sont codés sur un octet.

2: Soustraction de pointeurs

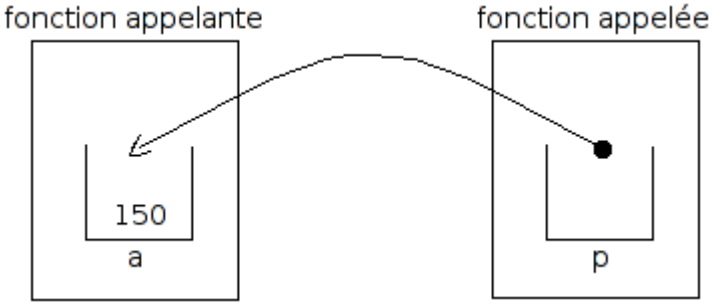
Rapidement, pour mémoire, notons qu'il est possible de soustraire des pointeurs, afin, par exemple, de savoir combien d'emplacements mémoire les séparent, mais c'est la seule opération que l'on peut effectuer sur deux pointeurs.
En particulier, il n'est pas possible d'en additionner, multiplier, diviser... Et, de toute façon, cela ne serait guère utile, après tout.

B: Pointeurs en paramètres de fonctions

Nous avons eu l'occasion de voir, lorsque nous travaillions sur les fonctions, au chapitre précédent, que leurs paramètres étaient passés par valeurs, et que cela empêchait les fonctions de modifier des variables appartenant à la portion de code appelante.
Nous avions dit que les pointeurs nous permettraient de trouver une solution à ce problème. C'est ce dont nous allons à présent discuter.

1: Un peu de théorie

Nous ne pouvons pas changer le fait que le passage de paramètres se fait par copie ; c'est fixé par le standard C, et il ne peut en être autrement. Cela dit, si plutôt que de passer en paramètre à la fonction une copie de la variable que nous voulons qu'elle puisse modifier, nous lui passons un pointeur sur celle-ci, elle aura accès, par le biais de ce pointeur, à la variable déclarée dans la fonction appelante. Nous pourrions représenter cette idée par un schéma tel celui-ci :



Dans notre fonction appelante, nous avons déclaré une variable nommée a, et nous avons passé à la fonction appelé l'adresse de cette variable. De la sorte, la fonction appelée a accès à l'emplacement mémoire correspondant à la variable a.

2: Un exemple

Etant donné que les pointeurs font parti des éléments du langage C qu'il est quasiment impossible de comprendre sans la pratique, nous allons reprendre l'exemple que nous avions utilisé en fin de chapitre précédent, à savoir, l'écriture d'une fonction dont le rôle est d'échanger les valeurs des deux variables passées en paramètres.
Nous avions vu que, puisque le C passe les paramètres par copie, cela n'était pas possible, avec les connaissances dont nous disposions avant de passer à ce chapitre...
Nous allons donc maintenant utiliser ce que nous venons d'apprendre, pour résoudre ce problème.
Voici le code-source d'un exemple complet, afin que vous puissiez voir par vous-même comment cela fonctionne :

```
#include <tigcclib.h>

void echange(short *a, short *b);

void _main(void)
{
    short a = 10;
    short b = 20;

    clrscr();

    printf("AVANT:   a=%d ;   b=%d\n", a, b);

    echange(&a, &b);

    printf("APRES:   a=%d ;   b=%d\n", a, b);

    ngetchx();
}


void echange(short *pa, short *pb)
{
    short temp;

    printf("DEBUT: *pa=%d ; *pb=%d\n", *pa, *pb);

    temp = *pa;
    *pa = *pb;
    *pb = temp;

    printf("FIN  : *pa=%d ; *pb=%d\n", *pa, *pb);
} // Fin echange
```

Exemple Complet

Que fait ce programme ?
Tout d'abord, dans la fonction _main, nous déclarons deux variables et leur affectons des valeurs.
Ensuite, nous passons les adresses de ces deux variables à la fonction echange.
Cette fonction travaille ensuite sur les pointeurs, et échange les valeurs pointées.
Et nous finissons par revenir à la fonction _main, où nous pourrons constater que les valeurs de a et de b ont bien été échangées.

Et voici le résultat qui sera affiché à l'écran :

```
AVANT:   a=10 ;   b=20
DEBUT: *pa=10 ; *pb=20
FIN  : *pa=20 ; *pb=10
APRES:   a=20 ;   b=10
```

Il est possible que, une fois arrivé à la fin de ce chapitre, les pointeurs vous paraissent encore un peu énigmatiques... Je dirais presque que c'est normal : d'après ce que j'ai pu observer, et d'après mon expérience personnelle, on ne maîtrise la notion de pointeur, et on ne sait véritablement les utiliser, qu'au bout d'un certain temps, une fois qu'on s'y est habitué... En somme, le savoir-faire vient avec la pratique, une fois encore.
Au cours des chapitres suivant, nous serons amenés à utiliser les pointeurs plus d'une fois. En particulier, nous les utiliserons pour les tableaux, et, donc, les chaînes de caractères, et pour tout ce qui se rapporte à l'allocation dynamique de mémoire.

 hapitre 12

Les Tableaux (Listes, Matrices)

Le langage C permet de créer des tableaux, nommés parfois listes, parfois matrices. Ce chapitre va nous permettre d'apprendre à les créer, et à les utiliser, tout en nous montrant quelques exemples possibles de leur utilité.

I:\ Notions de bases au sujet des tableaux

Au cours de cette partie, nous commencerons par définir le terme de "tableaux" appliqué au langage de programmation C. Ensuite, nous verrons à quoi ils peuvent nous servir lorsque nous développons un programme.

A: Tout d'abord, quelques mots de vocabulaire

Un tableau est une variable constituée de plusieurs cases d'un type donné, chacune capable de mémoriser une information du type en question.

Un tableau à une dimension est constitué d'une seule ligne et de plusieurs colonnes (ou, selon le point de vue, de l'inverse). Un tableau à deux dimensions est constitué de plusieurs lignes et de plusieurs colonnes.

Le terme de liste est souvent utilisé pour désigner un tableau à une dimension. Le terme de matrice est parfois utilisé, comme en mathématiques, pour désigner un tableau à plus de une dimension

B: Quelques exemples classiques d'utilisation possible de tableaux

Il est assez fréquent d'avoir à utiliser de nombreuses données d'un même type... et il n'est pas vraiment pratique d'utiliser un grand nombre de variables différentes : premièrement, il faut trouver des noms pour toutes ces variables, mais, en plus, avoir des variables différentes ne permet pas de toutes les traiter dans une boucle.

Un tableau permet de remédier à ces deux problèmes : plutôt que d'utiliser de nombreuses variables nommées a1, a2, a3, ..., a150, autant utiliser une seule variable, nommée, par exemple, 'a', qui contienne les 150 données que nous aurions placé dans nos variables aX.

Une application possible des tableaux, que nous prendrons d'ailleurs comme exemple à la fin de ce chapître, est le tri de données : on place des données dans le désordre dans un tableau, et on lance un algorithme de tri sur celui-ci ; à la fin de l'exécution de l'algorithme, le tableau contiendra les données triés, que nous pourrons récupérer grâce à une boucle parcourant les cases du tableau, les unes après les autres.

II:\ Tableaux à une dimension : Listes

Nous allons commencer par étudier la création, ainsi que l'utilisation, de tableaux à une dimension, communément appelés Listes.

A: Déclaration

Pour pouvoir utiliser un tableau, il nous faut commencer par, comme pour les autres types de variables, le définir. Pour cela, on utilise la syntaxe suivante :

```
TYPE nom_variable[NOMBRE_ELEMENTS];
```

TYPE désigne le type des éléments que l'on souhaite ranger dans notre tableau.
NOMBRE_ELEMENTS correspond au nombre de cases que contiendra le tableau ; notez que ce nombre doit être fixé au moment de la compilation, du moins en standard ANSI.

Par exemple, pour déclarer une liste capable de contenir 10 entiers, nous utiliserons cette syntaxe :

```
short  tab1[10];
```

Naturellement, cette syntaxe peut être réutilisée pour tous les types de variables que nous avons déjà vu, ainsi que pour ceux que nous serons amené à étudier dans le futur.

B: Initialisation

Il est possible, comme pour les autres types de variables que nous avons jusqu'à présent eu l'occasion d'étudier, d'initialiser un tableau lors de sa création, c'est-à-dire, d'affecter des valeurs à ses cases, en partant de la gauche.
Pour cela, on utilise une syntaxe de la forme suivante :

```
TYPE nom_variable[NOMBRE_ELEMENTS] = {case1, case2, case3, case4};
```

Naturellement, il faut que le nombre d'éléments compris dans la liste d'initialisation soit inférieur ou égal au nombre d'éléments que peut contenir le tableau. Dans le cas contraire, le compilateur nous renverra un warning.
Si la liste d'initialisation comporte moins d'éléments que le tableau ne peut en contenir, celui-ci sera rempli à partir du début, et les dernières cases, non initialisées, contiendront une valeur indéterminée.

Voici un exemple correspondant à l'initialisation d'un tableau d'entiers lors de sa déclaration :

```
short  tab3[3]  =  {1,  2,  3};
```

Comme on peut selon toute logique s'y attendre, la première case du tableau tab3 prendra pour valeur 1, la seconde pour valeur 2, et la troisième et dernière aura pour valeur 3.

C: Utilisation

L'accès aux éléments d'un tableau se fait par indice : lorsque l'on le numéro de la case dans laquelle se trouve une information, le C nous permet d'accéder à celle-ci.
Notez que l'indice d'un élément dans un tableau est toujours compris entre 0 pour le premier élément du tableau, et (nombre d'éléments moins un) pour le dernier. Je me permet d'insister sur le fait que le premier élément d'un tableau est à la case d'indice 0, et non pas 1 comme dans d'autres langages, tels le TI-BASIC !
Si vous dépassez ces bornes, vous sortirez du tableau ; en lecture, vous obtiendrez des données indéterminées ; en écriture, cela peut conduire à un plantage de votre programme. Soyez donc extrêmement prudents, notamment lorsque vous utiliserez une boucle pour parcourir un tableau.

Pour accéder à une case d'un tableau, on utilise la syntaxe suivante :

```
nom_du_tableau[indice]
```

C'est l'opérateur "`[]`" (un crochet ouvrant, l'indice, un crochet fermant) qui est utilisé pour signifier l'indexation de tableau. "indice" correspond à un nombre entier, supérieur ou égal à 0, et désigne la case du tableau sur laquelle on souhaite travailler.

Voici quelques exemples d'opérations que nous pouvons effectuer avec un tableau :

```
// Création d'un tableau de 6 entiers.
// On n'initialise que les 4 premières cases du tableau.
short tab[6] = {1, 2, 3, 4};
short i = 1;

tab[4] = 15; // on met 15 dans l'avant dernière case
tab[5] = 16; // et 16 dans la dernière
                //(qui a pour indice nombre de cases moins un = 6-1 = 5)

if(tab[2] == 3)
{
    // Code à exécuter si la 3ème case du tableau vaut 3.
    // (ce qui est ici le cas)
    printf("Coucou !\n");
}

// Affichage de la valeur de la case d'indice i
// Autrement dit, ici, de la case d'indice 1
// (la seconde case du tableau)
printf("Valeur de la case d'indice %d : %d\n", i, tab[i]);
```

Comme nous pouvons le constater, on utilise toujours la même syntaxe ; la seule chose à laquelle penser est qu'il faut éviter le débordement d'indice, pour ne pas sortir du tableau, et que, en C, les indices commencent à 0 !

D: Tableaux et pointeurs

En fait, si l'on se penche d'un peu plus près sur ce qui est en mémoire, un tableau n'est rien de plus qu'une série d'emplacements mémoire consécutifs.
Donc, si on connaît l'adresse de la première case du tableau, et qu'on la stocke dans un pointeur, il est possible, en incrémentant ce pointeur comme nous l'avons vu au chapitre précédent, de le faire pointer successivement sur toutes les cases du tableau.
Pour les exemples que nous utiliserons dans cette partie, nous considérerons que nous avons déclaré le tableau tab et le pointeur p comme ceci :

```
short tab[5] = {1, 2, 3, 4, 5};
short *p;
```

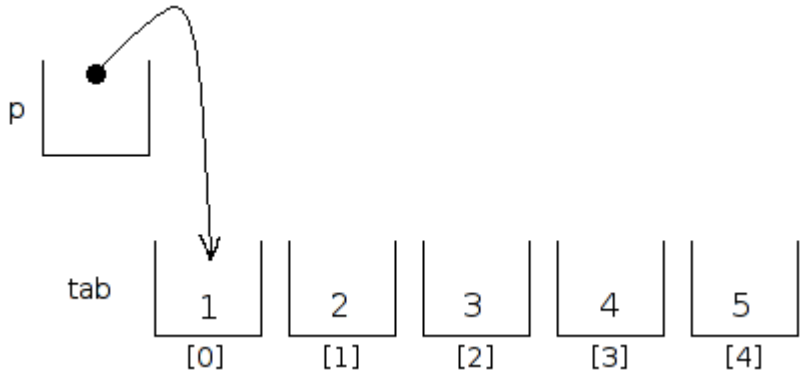
Commençons par faire pointer le pointeur p sur une des cases du tableau ; pour notre exemple, nous choisissons de le faire pointer sur la première case de celui-ci.
Pour cela, il nous suffit d'appliquer l'opérateur de référence à la première case du tableau, comme ceci :

```
p = &tab[0];    // p pointe sur la première case du tableau
```

Et si nous essayons d'afficher la donnée pointée par le pointeur, comme ceci :

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

Nous constaterons que la valeur affichée est 1, ce qui correspond effectivement au contenu de la première case de notre tableau.
Voici un schéma représentant ce que nous venons de faire :

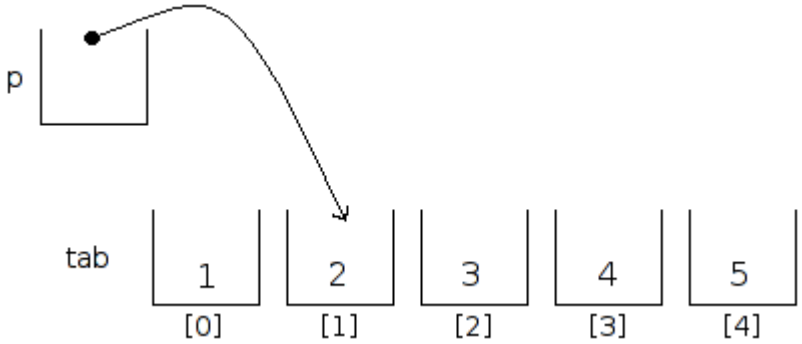


A présent, effectuons un petit peu d'arithmétique de pointeurs, et ajoutons 1 à notre pointeur, ce qui revient à le faire pointer sur l'emplacement suivant en mémoire.
Le C garantissant qu'un tableau est représenté en mémoire par une suite d'emplacements mémoires successifs, incrémenter de un notre pointeur, qui pointait sur la première case du tableau, revient à le faire pointer sur la seconde case du même tableau.
Voici la portion de code source correspondante, suivie d'un affichage, pour vérifier que *p vaut bien la valeur contenue dans la seconde case :

```
p++;
printf("*p = %d\n", *p);
```

Nous obtiendrons 2 comme valeur affichée, c'est-à-dire, la valeur de la seconde case du tableau, comme nous le souhaitions.

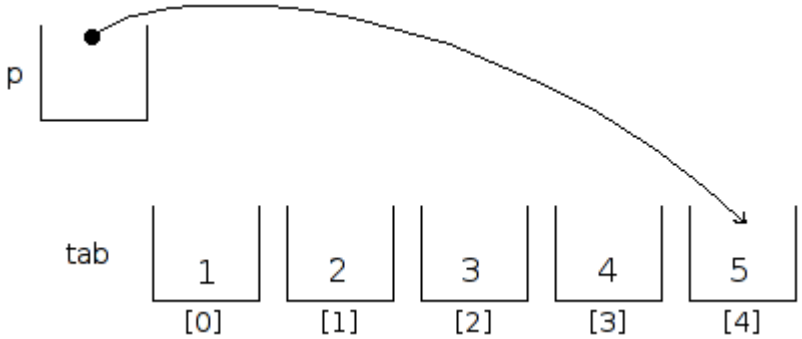
Et la représentation schématique correspondant à ce que nous venons de faire :



A présent, nous allons nous prouver que nous ne sommes pas forcé de n'incrémenter que de 1, et que nous pouvons utiliser d'autres valeurs, pour nous déplacer ailleurs dans le tableau.
Par exemple, allons voir trois cases plus loin :

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

A l'affichage, nous verrons la valeur 5, qui correspond à la case d'indice 4... c'est-à-dire, à la cinquième, et dernière, case de notre tableau. Ce qui, schématiquement représenté, équivaut à ceci :



Naturellement, il est possible, au lieu d'incrémenter le pointeur pour passer à l'une des cases suivantes, de le décrémenter pour passer à une des cases précédentes.

Lorsque vous utilisez un pointeur pour accéder aux différentes cases d'un tableau, il vous faudra penser à vous assurer que vous ne sortez pas du tableau. Autrement, le comportement de votre programme sera indéfini.
(Sur nos machines, un accès en dehors d'un tableau en lecture se traduit par l'obtention de données invalides, et, en écriture, en règle général, par un plantage dans le futur.)
Cela dit, lorsque l'on travaille avec les indices des cases du tableau, il nous faut nous assurer exactement de la même chose.

Pour en finir avec l'accès aux listes par pointeurs, je tiens à préciser que la variable déclarée comme un tableau est en fait elle-même un pointeur sur la première case de celui-ci.
Dans notre exemple, donc, l'écriture

```
*tab = 10;
```

est strictement équivalente à l'écriture

```
p = &tab[0];
*p = 10;
```

Dans les deux cas, on mémorise la valeur 10 dans la première case du tableau.
Notons que la seconde écriture pourrait avoir été écrite sans utiliser de pointeur temporaire, de cette manière :

```
*(&tab[0]) = 10;
```

Mais cela n'est pas forcément très lisible, en particulier pour un débutant non encore habitué à la syntaxe des pointeurs.

Nous étudierons, au chapitre suivant, un genre particulier de tableaux, que nous aurons tendance à parcourir en utilisant un pointeur, plutôt que des indices de cases, du fait que, dans ce cas là, l'écriture que nous venons d'introduire a tendance à être plus concise que la seconde.

III:\ Deux exemples complets utilisant des Listes

Pour terminer notre étude concernant les tableaux à une dimension, nous allons étudier deux petits exemples,

A: Exemple 1

Pour notre premier exemple, nous allons remplir un tableau avec des nombres entiers générés aléatoirement, puis nous afficherons le contenu des cases de celui-ci.

La génération de nombres aléatoires se fait en utilisant la fonction dont le prototype est le suivant :

```
short random (short num);
```

Cette fonction prend en paramètre un entier, et renvoie un entier dont la valeur sera supérieure ou égale à 0, et inférieure strictement à celle passée en paramètre.
Par exemple, pour générer un nombre aléatoire compris entre 0 et 9 inclus, on utilisera ceci :

```
random(10);
```

Voici à présent le code source de notre exemple ; quelques explications suivent celui-ci, mais ne devraient pas être utiles, étant donné que nous avons déjà étudié tout ce qui est utilisé au sein de l'exemple :

```
#include <tigcclib.h>

void _main(void)
{
    short tab[6];
    int i;

    for(i=0 ; i<6 ; i++)
    { // Parcours de tout le tableau, case par case.
        tab[i] = random(100);  // On affecte à la case
courante un nombre aléatoirement sélectionné entre 0 et 99 compris.
    }

    clrscr();
    for(i=5 ; i>=0 ; i--)
    { // Parcours du tableau en sens inverse
        printf("case %d => %d\n", i, tab[i]);
    }

    ngetchx();
}
```

Exemple Complet

Pour résumer en quelques mots ce que fait ce programme : on commence par déclarer un tableau de six entiers. On affecte ensuite à chacune des cases de celui-ci une valeur aléatoire comprise entre 0 et 99 compris.
Puis on finit par une boucle qui parcourt le tableau en partant de la fin, en affichant pour chaque case son indice, et la valeur contenue.

B: Exemple 2

Notre second exemple va partir de l'idée illustrée au cours du précédent, à savoir, remplissage d'un tableau à l'aide de valeurs générées aléatoirement, mais ira plus loin, puisque nous trierons ces valeurs avant de les afficher.
L'algorithme de tri que nous utiliserons est un des moins efficaces qui soit, mais il présente l'avantage d'être extrêmement simple, et rapide à écrire.
Voici le code source de l'exemple :

```
#include <tigcclib.h>

void _main(void)
{
    short tab[10];
    short i, j;
    short temp;

    for(i=0 ; i<10 ; i++)
    { // On remplit le tableau avec des données aléatoires
        tab[i] = random(20);
    }

    clrscr();
    printf("Données non triées :\n");
    for(i=0 ; i<10 ; i++)
    { // On affiche les données (non triées) contenues dans le tableau
        printf("%d ", tab[i]);
    }
    ngetchx();


    // Tri des données par ordre croissant :
    for(i=0 ; i<10 ; i++)
    {
        for(j=i+1 ; j<10 ; j++)
        {
            if(tab[j] < tab[i])
            { // On inverse
                temp = tab[i];
                tab[i] = tab[j];
                tab[j] = temp;
            }
        }
    }

    printf("\nDonnées triées :\n");
    for(i=0 ; i<10 ; i++)
    { // On affiche les données (qui, maintenant, sont triées) contenues dans le tablea
u
        printf("%d ", tab[i]);
    }
    ngetchx();

}
```

Exemple Complet

Le fonctionnement du programme est assez basique, divisé en trois parties : tout d'abord, on crée un tableau, et le remplit de valeurs générées aléatoirement ; ensuite, on l'affiche, pour montrer que les valeurs sont bien dans le désordre ; puis, on le trie ; et finalement, on le réaffiche, pour montrer que les valeurs sont à présent dans l'ordre.
Le principe de l'algorithme de tri utilisé est, dans les grandes lignes, le suivant : on parcoure le

tableau case par case, et, à chaque itération de la boucle de parcours, on s'assure que la case suivante contienne la plus petite valeur restant en partie droite du tableau ; autrement dit, à chaque itération, les cases à gauche de la case courante sont triées, et celles à droite de la case courante restent à trier.

IV:\ Tableaux à plusieurs dimensions

Le C ne permet pas de créer que des listes : il est en effet possible de créer des tableaux à "plusieurs dimensions", qui rentrent donc véritablement dans la définition de tableaux.
Bien qu'il soit possible de créer des tableaux à autant de dimensions que l'on veut, il est rarement que l'on ait besoin de plus de deux dimensions ; nous nous consacrerons donc, au cours de cette partie, aux tableaux à deux dimensions, communément appelés matrices.

A: Création d'un tableau à plusieurs dimensions

Voici la syntaxe qu'il convient d'utiliser pour déclarer un tableau à plusieurs dimensions :

```
TYPE
nom_du_tableau[nombre_elements_dimension_1][nombre_elements_dimension_2][nombre_element
s_dimension_3];
```

Cette syntaxe correspond au cas d'un tableau à trois dimensions, mais, naturellement, il suffit de l'adapter pour déclarer un tableau du nombre de dimensions que l'on souhaite. Par exemple, pour déclarer une matrice composée d'entiers, organisés en 10 lignes et 3 colonnes, on écrira ceci :

```
short tab[10][3];
```

Cela dit, il nous aurait été tout à fait possible d'écrire l'inverse, soit ceci :

```
short tab[3][10];
```

On aurait aussi obtenu un tableau de 10 lignes et 3 colonnes... organisées autrement en mémoire, mais capable de contenir la même quantité de données.
En fait, le C ne défini pas les termes de "ligne" et de "colonne" : c'est à nous, lorsque nous travaillons avec des tableaux, de nous souvenir de ce que nous considérons comme "ligne" et comme "colonne", notamment lorsque nous souhaitons y accéder. Personnellement, j'ai l'habitude de ceci :

```
TYPE nom_du_tableau[nombre_de_lignes][nombre_de_colonnes];
```

A vous de choisir la logique que vous préférez... et de vous y tenir pour ne pas vous embrouiller.

B: Utilisation d'un tableau à plusieurs dimensions

Pour utiliser un tableau à plusieurs dimensions, on utilisera le même type de syntaxe que pour les listes ; la seule différence sera que l'on utilisera non pas un indice désignant la case dans la liste, mais plusieurs ; à savoir, un par dimension.
Par exemple, pour accéder à la seconde case de la troisième ligne d'un tableau à deux dimensions (en considérant que la première dimension correspond à la ligne, et la seconde à la colonne) et y stocker la valeur 123, on utilisera cette syntaxe :

```
short tab[10][3];   // Tableau à 10 lignes et 3 colonnes
                        // (puisque je considère que la première dimension correspond
aux lignes, et la seconde aux colonnes)

tab[2][1] = 123;    // On enregistre la valeur 123
                        // à la seconde case de la 3ème ligne
```

Naturellement, tout comme pour les listes, les indices commencent à 0, et se terminent à nombre_d_éléments moins 1. Ce qui, après tout, est logique, puisqu'une matrice n'est finalement rien de plus, en mémoire, qu'une liste de listes.

C: Un petit exemple d'utilisation d'une matrice

Ci-dessous, un petit exemple qui, dans une matrice de 10 lignes et 3 colonnes, mémorise les tables de multiplication de 5, 6, et 7. Encore une fois, nous aurions pu nous dispenser d'utiliser un tableau pour un programme aussi simple, et calculer la table de multiplication lors de l'affichage (plutôt que la calculer à un moment, et l'afficher plus tard)...
Cela dit, encore une fois, un exemple bref, même s'il est simple et qu'il n'utilise pas forcément les tableaux de façon judicieuse, permet, à mon avis, de mieux présenter un sujet qu'un long exemple dans lequel on se perdrait.
Voici le code source de cet exemple :

```
#include <tigcclib.h>

void _main(void)
{
    short tab[10][3];  // Tableau à 10 lignes et 3 colonnes
                       // (puisque je considère que la première dimension correspond au
x lignes, et la seconde aux colonnes)
    short i=0,
          j=0;

    for(j=0 ; j<3 ; j++)
    {
        for(i=0 ; i<10 ; i++)
        {
            tab[i][j] = (i+1)*(j+5);   // i+1 car on veut que la table de multiplicatio
ns
                                       //aille de 1 à 10 (et non de 0 à 9)
                                       // j+5 car on veut les tables de multiplication d
e 5, 6, et 7
                                       // (j allant de 0 à 2 compris)
        }
    }

    // tab[i][1] (i variant de 0 à 9 compris) vaut
    // maintenant {6, 12, 18, 24, 30, 36, 42, 48, 54, 60)
    // Pour vérifier, affichons tab[i][1] (table de multiplication de 6) :
    clrscr();
    for(i=0 ; i<10 ; i++)
    {
        printf("6*%d = %d\n", i+1, tab[i][1]);
    }


    ngetchx();
}
```

Exemple Complet

Pour les curieux qui se demandent ce qu'aurait donné un programme n'utilisant pas de tableaux, voici une solution possible, qui affiche les tables de multiplication de 5, 6, et 7 au fur et à mesure qu'elles sont calculées :

```
#include <tigcclib.h>

void _main(void)
{
    short i=0,
          j=0;

    clrscr();
    for(i=0 ; i<10 ; i++)
    {
        printf("% 2d* ", i+1);
        for(j=0 ; j<3 ; j++)
        {
            printf("%d=% 2d ", j+5, (i+1)*(j+5));
        }
        printf("\n");
    }

    ngetchx();
}
```

Exemple Complet

Ce code source ne présentant rien de bien nouveau, et n'étant pas en rapport avec le sujet du chapître, je ne m'étendrai pas dessus. Juste à titre d'information, "% 2d" est une balise printf qui permet d'afficher un nombre entier sur au minimum deux caractères, en mettant des blancs à gauche si nécessaire.

D: Utilisation de pointeurs pour accéder aux matrices

Tout comme pour les listes, il est possible d'utiliser des pointeurs pour accéder aux différents éléments d'une matrice.
Je ne dirai que quelques mots à ce sujet, car ce n'est pas extrêmement utilisé, et il est beaucoup plus difficile de s'y retrouver que lorsqu'il s'agit de listes.
La principale différence est que vous avez à gérer plusieurs dimensions. Par exemple, pour passer à la ligne suivante d'un tableau à deux dimensions, il vous faut ajouter non pas 1, mais le nombre d'éléments que le tableau peut contenir par ligne, puisque les cases sont organisées de manière successive en mémoire.

V:\ Travail avec des tableaux

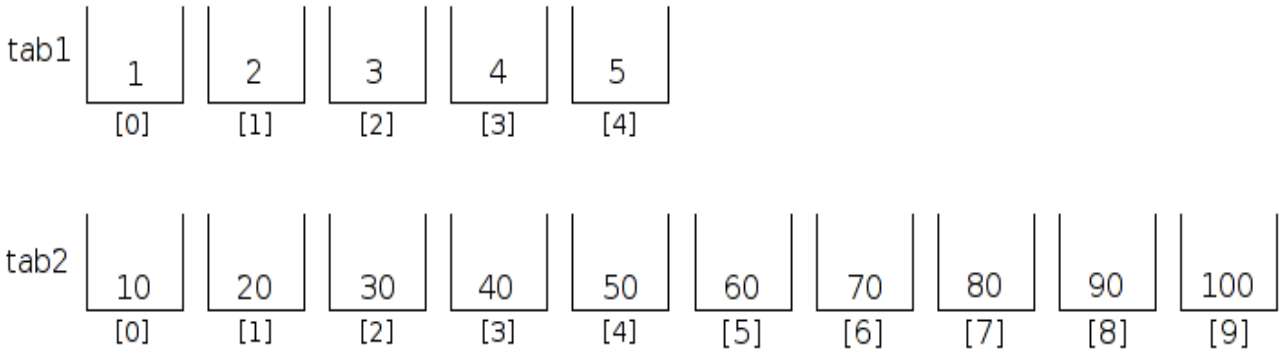
Pour finir ce chapitre, nous allons voir comment manipuler d'un seul coup plusieurs cases de tableaux ; par exemple, pour copier les données d'un tableau vers un autre tableau.

A: Copie de données d'un tableau vers un autre

Tout d'abord, pour cette sous-partie, nous allons considéré que nous avons déclaré deux tableaux d'entiers, nommés tab1 et tab2, comme ceci :

```
short tab1[5] = {1, 2, 3, 4, 5};
short tab2[10] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
```

A titre d'information, pour y voir plus clair, voici le schéma correspondant :



Si nous voulons copier le contenu du tableau tab1 vers le tableau tab2, nous serions tenté d'utiliser une syntaxe telle que ceci :

```
tab2 = tab1;
```

Cela dit, ceci n'est pas permis en C. En effet, l'opérateur d'affectation ne peut être appliqué à des tableaux, qui, bien que n'étant que des zones mémoires auxquelles le compilateur fait correspondre un pointeur, ne sont pas directement considéré comme des pointeurs par celui-ci.
Cette écriture est considérée une erreur au niveau du compilateur, qui refusera donc de compiler votre programme.

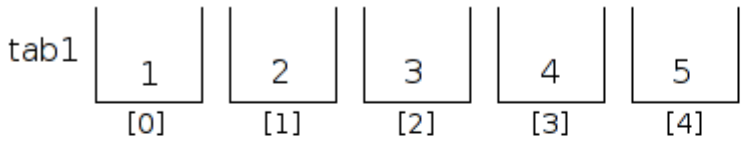
Si vous voulez copier des données d'une zone mémoire vers une autre autre mémoire, il va vous falloir employer la fonction memcpy, qui s'utilise comme ceci :

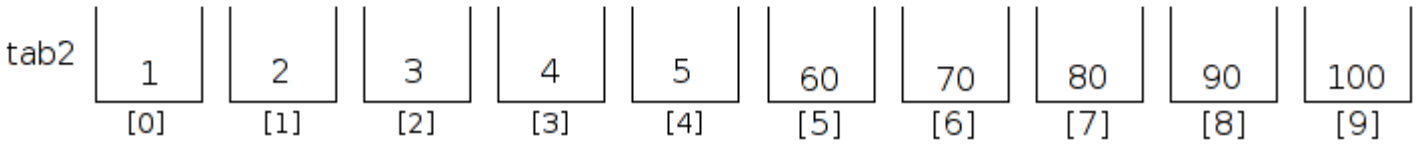
```
memcpy(destination, origine, nombre_d_octets);
```

Si vous souhaitez plusieurs cases d'un tableau vers un autre tableau, c'est ceci qu'il vous faut utiliser, puisqu'il est garanti, en C, que les cases successives d'un tableau seront organisées de manière séquentielle en mémoire. Par exemple, si nous souhaitons copier les 5 premières cases du tableau tab1 vers les 5 premières cases du tableau tab2, sachant que ces tableaux contiennent des éléments de type short, nous utiliserons ceci :

```
memcpy(tab2, tab1, 5*sizeof(short));
```

Et voici le résultat, représenté sous forme schématique :





Le dernier paramètre que l'on doit fournir à memcpy correspond à la taille en octets du bloc à copier. Il nous faut donc multiplier le nombre d'éléments que l'on souhaite copier par la taille de chacun de ces éléments, le plus propre pour obtenir la taille d'un élément étant d'utiliser sizeof. Considérer qu'on sait qu'un short est codé sur 2 octets n'est pas une bonne idée. En effet, cela risque de poser problème si vous essayez de recompiler votre programme pour une autre plate-forme, telle un PC par exemple. Alors que sizeof(short) vaudra toujours le nombre d'octets sur lesquels un short est codé, quelque soit la plate-forme.

Naturellement, il est possible de copier une portion de tab1 vers, par exemple, le milieu de tab2. Par exemple, pour copier vers les 6ème et 7ème cases de tab2 les 3ème et 4ème cases de tab1, nous utiliserions ceci :

```
memcpy(&tab2[5], &tab1[2], 2*sizeof(short));
```

Ou, en faisant de l'arithmétique de pointeurs, ce qui serait probablement plus naturel pour un programmeur C disposant d'un minimum d'expérience :

```
memcpy(tab2+5, tab1+2, 2*sizeof(short));
```

Naturellement, à chaque fois que nous faisons appel à la fonction memcpy, il nous faut nous prendre garde à ne pas dépasser des bornes du tableau.
En particulier, il faut nous assurer qu'on ne cherche pas à copier vers le tableau de destination plus qu'il ne peut contenir, ce qui causerait un plantage à plus ou moins long terme, et aussi, qu'on ne cherche pas à copier depuis le tableau d'origine plus de données qu'il n'en contient, ce qui reviendrait à copier des données indéterminées.
Soyez encore plus prudent si vous cherchez à copier des portions de tableaux comme nous venons juste de le faire.

Si, par curiosité, vous souhaitez afficher le contenu des deux tableaux, vous pourrez utiliser, par exemple, cette portion de code source :

```
unsigned short i;
for(i=0 ; i<5 ; i++)
     printf("tab1[%u]=%d\n", i, tab1[i]);
for(i=0 ; i<10; i++)
     printf("tab2[%u]=%d\n", i, tab2[i]);
```

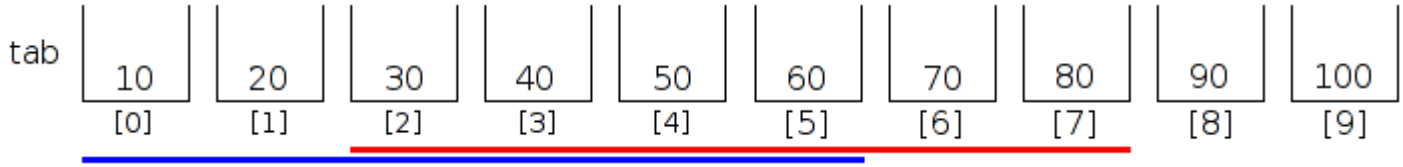
(Eventuellement, il peut être utile, selon la taille de votre écran et la taille des caractères utilisés, d'insérer un appel à ngetchx entre les deux boucles for)

B: Déplacement de données

Dans le cas où les zones mémoire d'origine et de destination sont superposées, la fonction memcpy ne doit pas être employée.
Cela se produit si, par exemple, on cherche à décaler de quelques cases les données d'un tableau.
Pour cette sous-partie, nous considérerons que nous avons déclaré le tableau tab de la façon qui suit :

```
short tab[10] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
```

Ce qui, graphiquement, correspond à ceci :



A présent, considérons le cas où nous voulons copier les données comprises dans les cases allant de la 3ème à la 8ème (zone en rouge sur le schéma) vers le début du tableau (zone en bleu sur le schéma).
De toute évidence, la région d'origine et celle de destination se recouvrent, ne serait-ce qu'en partie.
Si on en croit la documentation, memcpy ne doit pas être utilisée dans une situation telle que celle-ci. Nous ferons donc appel à la fonction memmove, qui s'emploie de la même façon :

```
memmove(destination, origine, nombre_d_octets);
```

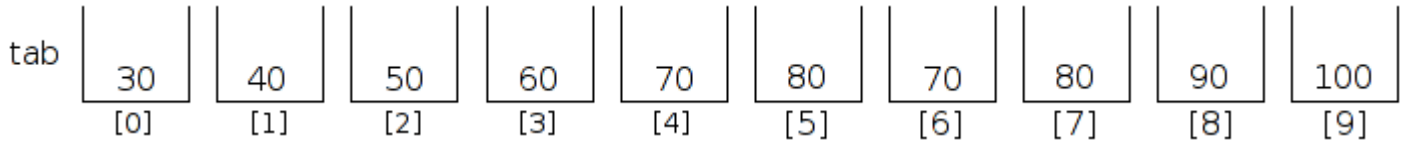
Dans le cas qui nous intéresse, il nous faudra donc écrire :

```
memove(&tab[0], &tab[2], 6*sizeof(short));
```

Ou plutôt, si vous commencez à penser comme la majorité des programmeurs C :

```
memove(tab, tab+2, 6*sizeof(short));
```

Ce qui nous permettra d'atteindre le résultat souhaité, que nous pourrions ainsi représenter, sous forme schématique :



C: Inversion de pointeurs

Pour terminer, nous allons voir une petite astuce qui peut être utile dans le cas où l'on souhaite échanger les contenus de deux tableaux.

Naturellement, on pourrait créer un tableau "temporaire", copier les données du premier tableau dedans, copier les données du second tableau dans le premier, et finir par copier les données du tableau temporaire vers le second tableau.
Un peu comme ceci :

```
short tab1[5] = {1, 2, 3, 4, 5};
short tab2[5] = {10, 20, 30, 40, 50};
short temp[10];

memcpy(temp, tab1, 5*sizeof(short));
memcpy(tab1, tab2, 5*sizeof(short));
memcpy(tab2, temp, 5*sizeof(short));
```

Cela dit, cette façon de procéder présente plusieurs inconvénients :

- Tout d'abord, il nous faut avoir un tableau temporaire suffisamment grand pour pouvoir contenir les données des deux tableaux (il faut donc que la taille du tableau temporaire soit supérieure ou égale à celle du plus grand des deux autres tableaux)
- Ensuite, si tab1 est plus petit que tab2, il ne pourra pas contenir toutes les données qui étaient dans tab2. tab2, lui, pourra contenir les données de tab1, mais il ne sera pas "rempli" : il restera, en fin de tableau, des cases inoccupées.
- Et enfin, copier des données prend du temps ; si les tableaux sont de grande taille, cela risque de ralentir notre programme.

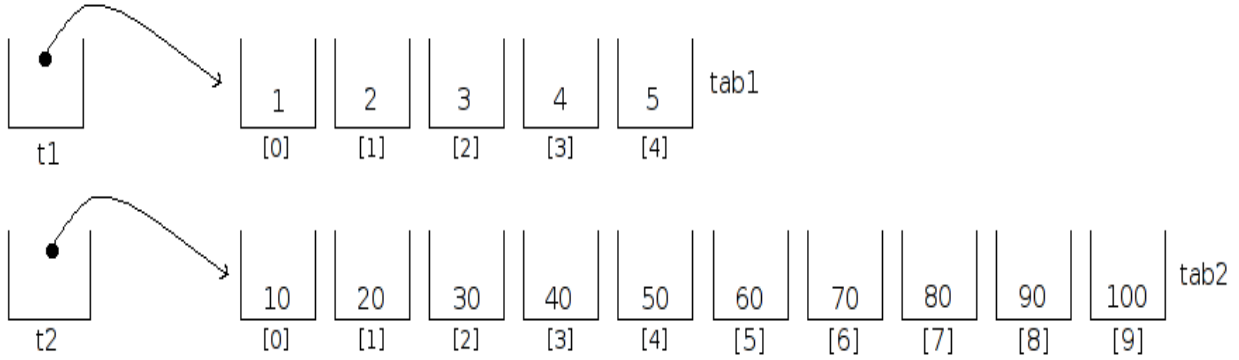
Une autre solution est de, dès le départ, travailler avec des pointeurs.
On commence par déclarer nos deux tableaux, et, en même temps, on déclare des pointeurs sur le type des données du tableau. Comme ceci :

```
short tab1[5] = {1, 2, 3, 4, 5};
short tab2[10] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};

short *t1;
short *t2;
```

Et dès le début du programme, on fait correspondre ces deux pointeurs à nos deux tableaux :

```
t1 = tab1;
t2 = tab2;
```

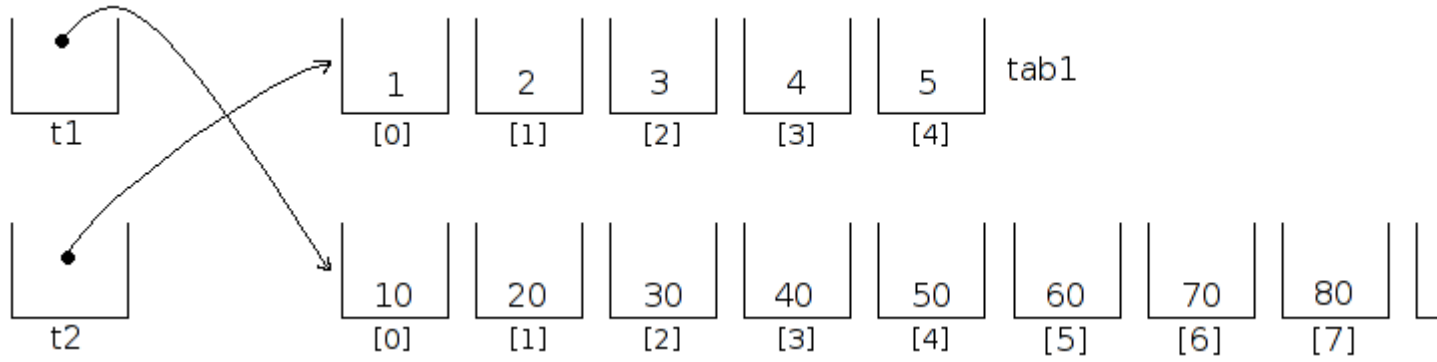
Schématiquement, on peut représenter cela ainsi :



Par la suite, au lieu de travailler avec nos tableaux, nous travaillons avec les deux pointeurs, et, au lieu d'échanger les contenu des deux tableaux, on échange les deux pointeurs :

```
short *temp;
temp = t1;
t1 = t2;
t2 = temp;
```

De la sorte, t2 pointera sur le tableau tab1, et t1 pointera sur le tableau tab2.
Si nous continuons à travailler avec nos deux pointeurs, nous aurons l'impression que les contenus des deux tableaux ont été échangés., ce qui peut se représenter ainsi :



Si on procède à un affichage des contenus pointés par les deux pointeurs, on verra que l'on peut accéder aux tableaux, exactement comme s'ils avaient été échangés.

Voici un exemple de code source complet illustrant les propos que nous venons de tenir :

```
#include <tigcclib.h>

void _main(void)
{
    unsigned short i;
    short tab1[5] = {1, 2, 3, 4, 5};
    short tab2[10] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};

    short *t1;
    short *t2;

    t1 = tab1;
    t2 = tab2;

    // Affichage des tableaux pointés
    // par les deux pointeurs, au départ :
    clrscr();
    printf("Tableau correspondant à t1:\n");
    for(i=0 ; i<5 ; i++)
        printf("t1[%u]=%d\n", i, t1[i]);
    ngetchx();
    clrscr();
    printf("Tableau correspondant à t2:\n");
    for(i=0 ; i<10; i++)
        printf("t2[%u]=%d\n", i, t2[i]);
    ngetchx();

    // Echange des deux pointeurs :
    short *temp;
    temp = t1;
    t1 = t2;
    t2 = temp;

    // Et affichage, après échange :
    clrscr();
    printf("Tableau correspondant à t1:\n");
    for(i=0 ; i<10 ; i++)
        printf("t1[%u]=%d\n", i, t1[i]);
    ngetchx();
    clrscr();
    printf("Tableau correspondant à t2:\n");
    for(i=0 ; i<5; i++)
        printf("t2[%u]=%d\n", i, t2[i]);
    ngetchx();

}
```

Exemple Complet

En premier, voici les deux premiers affichage, avant l'échange des pointeurs :

```
Tableau correspondant à t1:
t1[0]=1
t1[1]=2
t1[2]=3
t1[3]=4
t1[4]=5
```

```
Tableau correspondant à t2:
t2[0]=10
t2[1]=20
t2[2]=30
t2[3]=40
t2[4]=50
t2[5]=60
t2[6]=70
t2[7]=80
t2[8]=90
t2[9]=100
```

Et voici les deux suivants, après échange des pointeurs :

```
Tableau correspondant à t1:     Tableau correspondant à t2:
t1[0]=10                        t2[0]=1
t1[1]=20                        t2[1]=2
t1[2]=30                        t2[2]=3
t1[3]=40                        t2[3]=4
t1[4]=50                        t2[4]=5
t1[5]=60
t1[6]=70
t1[7]=80
t1[8]=90
t1[9]=100
```

Comme nous pouvons le constater, nous avons l'impression que les contenus des deux tableaux ont été échangés, alors que, finalement, nous n'avons fait qu'échanger deux pointeurs.
Nous avons gagné en rapidité (il est nettement plus rapide d'échanger deux pointeurs, ce qui revient à effectuer trois affectations, que d'échanger les contenus de deux gros tableaux !), mais aussi en espace mémoire, puisque nous n'avons plus besoin d'un (éventuellement grand) tableau temporaire.

Le chapitre suivant, traitant des chaînes de caractères, va nous permettre de travailler avec des tableaux dont le contenu est organisé de façon quelque peu particulière. Il nous permettra de mettre en application certains des points que nous venons d'étudier.

Chapitre 13

Chaînes de caractères

Au cours de ce chapitre, nous traiterons de ce qui est appelé, en C, « Chaînes de caractères ».
Nous commencerons par étudier les caractères, qui sont l'élément de base des chaînes de caractères, puis nous verrons comment déclarer une chaîne de caractères. Ensuite, nous apprendrons à les manipuler grâce aux fonctions que fournit TIGCC, et, enfin, nous verrons comment nous pourrions les manipuler par pointeurs.

Tout d'abord, présentons, en quelques mots, ce qu'est une chaîne de caractères, pour ceux qui n'auraient pas l'habitude de ce terme technique.
Le terme de "chaîne de caractères" correspond à un type de données composé d'une suite ordonnée de caractères, et utilisé pour représenter du texte, un caractère étant une lettre, un chiffre, ou un symbole, visible ou non, tel, par exemple, les parenthèses, la ponctuation, ou l'espace.
Nous en avons déjà utilisé sans vraiment savoir qu'il s'agissait de ce dont nous allons parler au cours de ce chapitre.
Par exemple, "ceci est une chaîne de caractères".

I:\ Caractères

Ce que l'on appelle "caractère" est le constituant de base de la chaîne de caractères ; il s'agit du terme utilisé en informatique pour désigner n'importe quel symbole pouvant apparaître dans du texte, que ce soit des lettres, des chiffres, ou n'importe quel autre symbole tel la ponctuation par exemple.
Nos TI codent les caractères sur un octet, en utilisant un codage proche de l'ASCII étendu.
Cela signifie que les 8 bits de chaque octet sont utilisé, ce qui permet 256 codes de caractères différents.
Vous trouverez, page 18 de l'annexe B de votre manuel, disponible en PDF ici si vous ne l'avez pas sous la main, la correspondance entre chaque caractère son code.

A: Notation d'un caractère

En C, pour représenter le code d'un caractère (on dit généralement code ASCII, même si ce n'est pas forcément de l'ASCII "pur"), on écrit le symbole correspondant à celui-ci en le plaçant entre guillemets simples.
Par exemple, le caractère 'a' a pour code, d'après la table de caractères dont nous avons parlé plus haut, 97 en décimal, soit 0x61 en hexadecimal.

Les caractères sont mémorisés, en C, sous forme de variables de type char.
Par exemple, pour déclarer la variable c, de type char, et contenant le caractère 'Y', nous utiliserons cette syntaxe :

```
char c = 'Y';
```

En regardant la table de caractères, nous pouvons observer que les lettres de l'alphabet en majuscules sont toutes à la suite les unes des autres, et qu'il en va de même pour les lettres en minuscules.
Ceci va nous faciliter la vie : pour savoir si un caractère est, par exemple, une lettre majuscule, il nous suffira de tester si son code ASCII est compris entre celui de 'A' et celui de 'Z'.
A titre de curiosité, nous pouvons noter que les lettres minuscules ont des codes ASCII supérieurs à ceux de leurs équivalentes minuscules ; d'ailleurs, l'écart entre le code ASCII d'une lettre minuscule et celui de la lettre majuscule correspondante est, si l'on observe la table, toujours égal à 'a'-'A'.
A partir de là, il nous est facile d'écrire une fonction qui convertisse une lettre majuscule en la lettre minuscule correspondante : tout d'abord, nous testons si le caractère passé en paramètre est bien une lettre majuscule, et, si c'est le cas, nous renvoyons le code de la lettre minuscule correspondante. Si le caractère passé en paramètre n'était pas une lettre majuscule, nous renvoyons ce caractère.
Voici le code source correspondant à un programme implémentant et utilisant une telle fonction :

```
#include <tigcclib.h>

char MAJ_to_min(char c)
{
     if(c>='A' && c<='Z')
          return c+('a'-'A');
     else
          return c;
} // Fin maj_to_min

void _main(void)
{
     clrscr();

     printf("'A' => '%c'\n", MAJ_to_min('A'));
     printf("'F' => '%c'\n", MAJ_to_min('F'));

     ngetchx();
}
```

Exemple Complet

La fonction MAJ_to_min aurait aussi pu être écrite de manière plus concise, comme ceci :

```
char MAJ_to_min(char c)
{
     return c>='A' && c<='Z' ? c+('a'-'A') : c;
} // Fin maj_to_min
```

Notons qu'il existe déjà une fonction dans les librairies de TIGCC qui serve à convertir des caractères de majuscule à minuscule ; il s'agit de la fonction tolower. Sa réciproque est la fonction toupper.

B: Caractères Spéciaux

Les Caractères allant de 0x00 à 0x1F sont généralement appelés caractères non imprimables.
Historiquement, ils correspondent à des codes de contrôle utilisés pour, par exemple, les machines à écrire ou les imprimantes en mode texte.
Parmi ceux-ci, on peut par exemple citer le caractère 0x0D, nommé "Carriage Return" (Retour chariot), qui correspond, sur certains systèmes, à un retour à la ligne.
La plupart de ces caractères, sur des machines tels nos ordinateurs, ne sont normalement pas visibles à l'écran, ce qui explique l'appellation de "caractères non imprimables".

Cela dit, la plupart d'entre-eux sont, sur TI, du fait qu'il n'y a pas de périphérique en mode texte à contrôler, représentés par un pictogramme visible à l'écran.

Les caractères de contrôle les plus fréquent peuvent être représentés de manière simple en C : il leur est fait correspondre, au niveau du compilateur, une lettre, précédée d'un antislash pour signifier qu'elle ne doit pas être interprétée telle qu'elle.
En particulier, nous pouvons citer ces quelques caractères :

Caractère	Signification
`'\t'`	Tabulation
`'\n'`	Retour à la ligne
`'\\'`	Antislash
`'\"'`	Guillemets doubles
`'\''`	Guillemet simple

Il en existe d'autres, qui ne sont pas vraiment utile sur TI, puisque nous ne travaillons pas sur un matériel en mode texte ; nous ne les présenterons pas ici.

Nous pouvons remarquer plus particulièrement les trois derniers caractères de notre tableau : ils doivent être précédés d'un antislash, afin que leur signification première soit annulée. Par exemple, le caractère '\", sans l'antislash, correspondrait à la fin du caractère ; on aurait, autrement dit, ceci '", soit deux guillemets n'encadrant aucun caractère, et un guillemet tout seul ne servant à rien... Cela entraînerait une erreur de compilation.
En somme, l'antislash permet de donner une signification différente au caractère qui le suit, ou, comme nous le verrons un peu plus bas, aux caractères qui le suivent. On dit généralement qu'il agit comme caractère d'échappement.
Le fait "d'échapper" un caractère revient donc à le faire précéder d'un antislash.

C: Désigner des caractères par leur code

Il est des caractères qu'il est particulièrement difficile de saisir au clavier. Par exemple, si vous voulez utiliser le caractère '©', vous pouvez avoir quelques difficultés à le trouver.
C'est pour cela que le C permet de désigner des caractères directement par leur code, en octal en préfixant le code par un antislash, ou en hexadécimal en précédant le code par un antislash et un x.
Par exemple, le caractère '©' correspond, d'après la table des codes ASCII trouvée dans le manuel de votre calculatrice, à la valeur 169. 169 en décimal correspond à A9 en hexadécimal, et à 251 en octal.
Au lieu d'écrire '©' dans votre programme, il vous est possible d'utiliser '\251', ou '\xA9', selon la base numérique que vous préférez.
Notez que cela est indispensable pour les caractères non imprimables qui n'ont pas de code textuel.
Par exemple, pour afficher un petit verrou, vous utiliserez ceci :

```
#include <tigcclib.h>

void _main(void)
{
	clrscr();

	printf("\16\n");
	printf("\x0E\n");

	ngetchx();
}
```

Exemple Complet
(Du moins, vous utiliserez une des deux écritures, selon votre préférence)

II:\ Chaînes de caractères en C

Maintenant que nous avons vu ce que sont les caractères, nous allons pouvoir commencer à étudier les chaînes de caractères, généralement nommées "string" en anglais.
La norme C défini une chaîne de caractères comme un tableau de chars, dont le dernier élément vaut 0. C'est ce marqueur de fin qui est caractéristique des chaînes de caractères, et toutes les fonctions permettant d'en manipuler supposent que ce 0 final est présent.
Autrement dit, lorsque vous avez une chaîne de caractères contenant 10 lettres, elle occupe en fait 11 emplacements mémoires ; pensez-y lorsque vous avez à déclarer de l'espace mémoire pour stocker une chaîne de caractères.

A: Création de chaînes de caractères

Une chaîne de caractères étant un liste de caractères, elle peut se déclarer comme telle. Par exemple :

```
char str[] = {'S', 'a', 'l', 'u', 't', '\0'};
```

Cela dit, il est extrêmement fastidieux d'utiliser ce type de syntaxe. C'est pourquoi le C fourni les guillemets doubles, qui indiquent au compilateur que ce qu'ils entourent est une chaîne de caractères.
On pourrait ré-écrire notre exemple précédent de cette manière :

```
char str[] = "Salut";
```

Vous remarquerez que, avec cette syntaxe, le 0 de fin de chaîne est automatiquement généré par le compilateur.
Cette écriture est équivalente à celle que nous avons employé juste au-dessus.

Il est aussi possible de déclarer une chaîne de caractères en utilisant la syntaxe par pointeurs, comme ceci :

```
char *str = "Salut";
```

Tant que l'on n'essaye d'accéder à la chaîne de caractères qu'en lecture, ce type de déclaration aura les mêmes conséquences que les deux précédents.
Cela dit, il existe une différence importante entre la déclaration par tableau, et celle par pointeur.

En effet, dans le premier cas (déclaration par tableau), str est un tableau de taille juste suffisamment grande pour pouvoir contenir la chaîne de caractères et son 0 de fin de chaîne. Quelque soit les manipulations effectuées sur la chaîne, str désignera toujours la même zone mémoire. De plus, à chaque fois que vous entrerez dans le bloc au sein de laquelle str est déclarée, la valeur de celle-ci sera réinitialisée.
En revanche, dans le cas de la déclaration par pointeur, le compilateur va créer une zone dans laquelle le texte sera stocké, et initialisera un pointeur pointant sur cette zone. A chaque fois qu'on rentrera dans le bloc contenant cette déclaration, ce sera le pointeur qui sera réinitialisé, et non le texte !

Pour illustrer mes propos, observons deux exemples, dans lesquels nous déclarons, au sein d'une boucle se répétant deux fois, une chaîne de caractères, l'affichons, et modifions son premier caractère.
Dans le premier cas, nous déclarons notre chaîne de caractères par tableau :

```
#include <tigcclib.h>

void _main(void)
{
    short i;

    clrscr();
    for(i=0 ; i<2 ; i++)
    {
        char str[] = "Salut";
        printf("%s\n", str);
        str[0] = '!';
    }
    ngetchx();
}
```

Exemple Complet
Et dans le second cas, nous la déclarons par pointeur :

```
#include <tigcclib.h>

void _main(void)
{
    short i;

    clrscr();
    for(i=0 ; i<2 ; i++)
    {
        char *str = "Salut";
        printf("%s\n", str);
        str[0] = '!';
    }
    ngetchx();
}
```

Exemple Complet
L'écriture str[0] permet d'accéder au premier caractère de la chaîne, puisque, après tout, une chaîne de caractères n'est rien de plus qu'un tableau, quelque soit la façon dont elle est déclarée : ce que nous avons concernant les tableaux au chapitre précédent continue de s'appliquer ici.

Nous obtenons, à l'exécution, les affichages suivants :
Dans le premier cas :

```
Salut
Salut
```

Et dans le second :

```
Salut
!alut
```

Autrement dit, dans le premier cas, la chaîne de caractères a été initialisée à chaque passage dans la boucle, alors qu'au second, elle ne l'a été qu'une et une seule fois ; par la suite, c'est le pointeur, et non la chaîne elle-même, qui a été initialisée.
Ces deux exemples illustrent bien la différence qu'il existe entre les deux écritures.

Cela dit, veuillez noter qu'il **n'est** généralement **pas** une bonne idée que de modifier une chaîne de caractères crée en la plaçant entre guillemets doubles.
En effet, il arrive que le compilateur choisisse, si deux chaînes sont identiques, de les fusionner en une seule. Dans ce cas vous modifiez l'une, cela modifiera aussi l'autre, puisqu'elles ne font finalement qu'une !
Par exemple, regardez cet exemple :

```
#include <tigcclib.h>

void _main(void)
{
    char *str1 = "salut";
    char *str2 = "salut";

    clrscr();

    printf("%s %s\n", str1, str2);

    str1[0] = '!';

    printf("%s %s\n", str1, str2);

    ngetchx();
}
```

Exemple Complet
En premier, on affiche

```
salut salut
```

Mais une fois qu'on modifie str1, on affiche

```
!alut !alut
```

En fait, str2 a aussi été modifiée !
Ce qui nous prouve que le compilateur avait remarqué que str1 et str2 étaient déclarées comme pointant sur le même texte, et avait choisi de ne stocker celui-ci qu'une seule fois dans le programme.
Notez que ce comportement peut varier selon le compilateur. En effet, la norme dit que le comportement en cas de modification d'une chaîne déclarée par pointeur est indéterminé.

Pour en finir avec la déclaration de chaînes de caractères, j'en arrive à la forme qu'on utilise le plus fréquemment lorsque l'on souhaite en manipuler.
Il s'agit d'une déclaration de tableau, dont on précise la taille, mais sans l'initialiser. Encore une fois, il faut penser au 0 de fin de chaîne.
Par exemple, si on veut déclarer une chaîne de caractères qui puisse contenir 10 lettres, on déclarera un tableau de 11 chars, de la manière suivante :

```
char str[11];
```

Maintenant que nous savons comment déclarer des chaînes de caractères, nous allons apprendre, grâce à la suite de ce chapitre, à les manipuler.

B: Manipulations de chaînes de caractères

Les chaînes de caractères étant très souvent employées, et leur format étant strictement défini par la norme C, de nombreuses fonctions permettant de les manipuler sont généralement inclues à aux bibliothèques des compilateurs.
Ainsi, TIGCC et AMS nous fournissent de nombreuses fonctions, dont un bon nombre sont des ROM_CALLs, dédiées au travail avec des chaînes de caractères. Notez que la plupart d'entre elles sont des fonctions ANSI, ce qui garantit que vous les retrouverez avec quasiment tous les compilateurs C existants.
Sous TIGCC, les fonctions de manipulation de chaînes de caractères sont regroupées dans le fichier d'en-tête <string.h>.
Au cours de cette partie, nous allons voir comment copier des chaînes de caractères vers d'autres chaînes de caractères, comment en concaténer, comment en formater, ou encore comment en obtenir la longueur...

1: Longueur d'une chaîne de caractères

Tout d'abord, la fonction strlen permet de déterminer la longueur, en nombre de caractères, de la chaîne de caractères qu'on lui passe en argument. Voici son prototype :

```
unsigned long strlen(const char *str);
```

Notez que c'est le nombre de caractères "utiles" qui est retourné : le 0 de fin n'est pas compté.
Par exemple, considérons cette portion de code :

```
printf("%lu", strlen("salut"));
```

La valeur affichée sera 5 : la chaîne de caractères "salut" contient 5 caractères, même si elle occupe un sixième espace mémoire pour le 0 de fin de chaîne.

2: Copie de chaînes de caractères

Pour copier une chaîne de caractères vers une autre, il convient d'utiliser la fonction strcpy, qui prend en paramètre un pointeur sur la zone mémoire de destination, et la chaîne d'origine :

```
char *strcpy(char *destination, const char *origine);
```

Par exemple :

```
char str[30];
strcpy(str, "Hello World !");
printf("%s", str);
```

Cette portion de code affichera "Hello World !".
Notez que la zone de mémoire de destination doit être de taille suffisamment important pour pouvoir contenir la chaîne de caractères que l'on essaye d'y placer, 0 de fin compris. Dans le cas contraire, votre programme risque de corrompre des données, et d'entraîner un plantage.
Cette fonction retourne un pointeur sur la chaîne de destination.

3: Concaténation de chaînes de caractères

A présent, passons à la concaténation de chaînes de caractères.
En informatique, on utilise le terme "concaténer", pour signifier que l'on veut mettre bout à bout deux chaînes de caractères. Par exemple, la concaténation de "Hello " et de "World !" donnera "Hello World !".
C'est la fonction strcat qui remplit ce rôle. Elle accole à la chaîne de caractères passée en second paramètre à la fin de la première.

```
char *strcat(char *destination, const char *origine);
```

Par exemple :

```
char str[30];
strcpy(str, "Hello");
printf("%s", str);
strcat(str, " World !");
printf("%s", str);
```

Cette portion de code commencera par afficher "Hello", et finira en affichant "Hello World !".
Ici encore, zone de mémoire correspondant à la chaîne de destination doit être de taille suffisamment importante pour pouvoir contenir les textes des deux chaînes de départ, plus le 0 final.

La fonction strcat retourne un pointeur sur la chaîne de destination ; celle qui contiendra le résultat de la concaténation.

Notez que les deux paramètres doivent être des chaînes de caractères valides ! En particulier, le code suivant, où la chaîne de caractères de destination n'a pas été initialisée, ne fonctionnera pas, et risque de causer un plantage :

```
char str[30];
strcat(str, "Hello World !");
```

En effet, strcat parcourt la première chaîne jusqu'à trouver son 0 final, puis parcourt la seconde chaîne jusqu'à son 0 final, en ajoutant les caractères de celle-ci à la suite de ceux de la première. Donc, si une des deux chaînes n'a pas de 0 final (autrement dit, si un des paramètres n'est pas une chaîne de caractères !), cela ne fonctionnera pas.

4: Comparaison de chaînes de caractères

Assez souvent, lorsque l'on travaille avec des chaînes de caractères, il nous arrive d'avoir à les comparer, pour savoir laquelle est la plus "grande", la plus "petite", ou si elles sont "égales".
Dans ce but, le C fournit la fonction strcmp, qui prend en paramètres deux chaînes de caractères, et retourne :

- 0 si les deux chaînes sont égales.
- Une valeur supérieure à 0 si la première chaîne est plus "grande" que la seconde.
- Une valeur inférieure à 0 si la première chaîne est plus "petite" que la première.

La conversion se faisant sur les codes ASCII de chaque caractères, l'ordre est alphabétique, les majuscules étant plus "petites" que les minuscules.
Voici le prototype de cette fonction :

```
short strcmp(const unsigned char *str1, const unsigned char *str2);
```

Par exemple, pour tester si deux chaînes sont égales, nous pourrons utiliser une écriture telle que celle-ci :

```
if(!strcmp("salut", "salut"))
{
      // Les deux chaînes sont égales
}
```

Je me permet d'insister sur le fait que la valeur retournée en cas d'égalité entre les deux chaînes de caractères est 0.

Pour la forme, voici un petit exemple ou nous testons les 3 cas : égalité, et plus "petit" ou plus "grand" :

```
char str1[] = "aaaab";
char str2[] = "aahbag"
short comp = strcmp(str1, str2);
if(!comp)
    printf("str1 == str2");
else if(comp < 0)
    printf("str1 < str2");
else
    printf("str1 > str2");
```

Si vous êtes curieux, sachez que, en cas de différence entre les deux chaînes, la valeur retournée est égale à la différence entre les codes ASCII du premier caractère différent de chaque chaîne.
Par exemple, si on appelle strcmp sur les chaînes "aaax" et "aazb", la valeur retournée sera 'a'-'z', c'est-à-dire -25 : la chaîne "aaax" est plus petite que la chaîne "aazb".

Si vous avez besoin d'effectuer une comparaison qui ne soit pas sensible à la casse (c'est-à-dire, qui considère que les majuscules et minuscules sont égales), vous pouvez utiliser la fonction suivante :

```
short cmpstri(const unsigned char *str1, const unsigned char *str2);
```

Elle fonctionne exactement comme strcmp, sauf qu'elle est un peu plus lente, étant donné qu'elle convertit tous les caractères de majuscule vers minuscule avant d'effectuer la comparaison.

5: Fonctions travaillant uniquement sur les n premiers caractères

La librairie C fourni aussi des fonctions permettant de copier, concaténer, ou comparer des chaînes, mais en se limitant à un certain nombre de caractères, que l'utilisateur peut définir, en comptant à partir du premier de la chaîne.
Ces fonctions sont nommées comme leurs équivalents travaillant sur des chaînes entières, mais en intercalant un 'n' entre "str" et ce que fait la fonction. Il faut aussi leur passer un troisième paramètre, qui correspond au nombre de caractères que la fonction devra traiter.
Voici leurs prototypes :
Pour la copie des n premiers caractères d'une chaîne vers un bloc mémoire :

```
char *strncpy(char *destination, const char *origine, unsigned long n);
```

Pour accoler à la fin d'une chaîne les n premiers caractères d'une autre :

```
char *strncat(char *destination, const char *origine, unsigned long n);
```

Et enfin, pour comparer les n premiers caractères de deux chaînes :

```
short strncmp(const unsigned char *str1, const unsigned char *str2, unsigned long n);
```

Tout comme celles que nous avons vu précédemment, ces fonctions supposent que les chaînes qu'elles doivent lire sont valides (terminées par un 0), et que les zones mémoires où elles doivent écrire sont suffisamment grandes pour contenir ce que l'on essaye d'y placer.
De plus, comme on peut s'y attendre, si le nombre de caractères que vous spécifiez est plus grand que la taille effective de la chaîne, ces fonctions ignoreront le nombre de vous avez spécifié, et s'arrêteront à la fin de la chaîne.
Notez, et c'est important, que, pour strncpy, si le nombre de caractères dans la chaîne d'origine est supérieur ou égal à n, la chaîne de destination **ne sera pas** terminée par un '\0' ! Dans ce cas, pour que la chaîne de destination soit considérée comme valide, ce qui est indispensable si vous voulez appeler une autre fonction dessus, il faudra que vous rajoutiez le '\0' de fin de chaîne vous-même !

6: Conversion d'une chaîne de caractères vers un entier ou un flottant

Parfois, on dispose d'une chaîne de caractères, et on sait qu'elle contient une écriture de type numérique. On peut alors être amené à vouloir obtenir ce nombre sous forme d'une variable sur laquelle on puisse faire des calculs, telle, par exemple, un entier ou un flottant.
Il existe donc des fonctions qui permettent de convertir des chaînes de caractères en un type numérique. Les voici :
Pour convertir une chaîne de caractères en entier short :

```
short atoi(const char *str);
```

Pour convertir une chaîne de caractères en entier long :

```
long atol(const char *str);
```

Pour ces deux fonctions, on passe en paramètre la chaîne de caractères, et on obtient l'entier en retour. L'analyse de la chaîne de caractères s'arrête au premier caractère non valide pour un entier.
Notez que ces fonctions retourneront un résultat indéterminé si vous essayez de les faire travailler sur une valeur plus grande que ce qu'un short (respectivement, un long) peut contenir.
Si vous avez besoin de plus de flexibilité que celle proposée par ces fonctions, je vous conseille de jeter un coup d'oeil sur la documentation de la fonction strtol. Etant donné qu'il est rare que les débutants aient besoin de ce qu'elle permet de faire, je ne la présenterai pas plus ici.

Et, pour convertir une chaîne de caractères en flottant, on utilisera cette fonction :

```
float atof(const char *str);
```

Elle s'utilise comme les deux fonctions retournant des entiers.
Si la conversion ne peut se faire, atof renverra la valeur NAN (Not A Number). Vous pourrez vérifier si l'appel a échoué en appelant la fonction is_nan sur la valeur retournée par atof.

7: Formatage de chaînes de caractères

Au cours de ce tutorial, nous avons souvent utilisé la fonction printf, qui permet d'afficher à l'écran du texte, en le formatant : insertion du contenu de variables au cours du texte, nombre de caractères fixes, ...
Le C fourni la fonction sprintf, qui fonctionne exactement de la même façon que printf, à la différence prêt que la chaîne de caractères résultante, au lieu d'être affichée à l'écran, est mémorisée dans une variable.
Voici le prototype de cette fonction :

```
short sprintf(char *buffer, const char *format, ...);
```

Le premier paramètre, buffer, correspond à un pointeur vers un tableau de caractères suffisamment grand pour que la fonction puisse y mémoriser la chaîne de caractères formatée.
Le second paramètre, format, correspond à une chaîne de caractères, au sein de laquelle sont utilisés le même type de codes de formatage que pour la fonction printf. Je vous invite à consulter la documentation de printf pour la liste de ces codes.
Ensuite, viennent, dans l'ordre, les paramètres correspondant aux codes de formatage utilisés dans le second paramètre.
Par exemple :

```
char str[30];
short a = 10;
sprintf(str, "a vaut %d\n", a);
```

Naturellement, on peut ensuite travailler avec la chaîne de caractères obtenue ; par exemple, pour l'afficher avec une fonction autre que printf.

III:\ Un peu d'exercice !

Pour terminer ce chapitre, nous allons profiter du fait que certaines fonctions de manipulation de chaînes de caractères soient très simples pour faire un peu d'exercice concernant les tableaux et les pointeurs : nous allons voir comment il est possible de réécrire certaines des fonctions que nous avons ici appris à utiliser.
Pour certaines des fonctions que je choisirai de vous proposer de ré-implémenter, je donnerai une ou plusieurs solutions possibles. En particulier, du moins au début, je proposerai une solution qui pourrait correspondre à du code écrit par un débutant non habitué aux manipulations de pointeurs, et je fournirai une autre solution, correspondant plus à du code de programmeur C ayant de l'expérience dans ce langage. Naturellement, j'expliquerai les différences entre les différentes écritures, et pourquoi on peut passer de l'une à l'autre.
Je ne pense pas que cette partie soit indispensable à pour ce qui est de l'utilisation de base des chaînes de caractères, mais si, un jour, vous avez besoin d'écrire des fonctions travaillant sur du texte, ce que nous allons pourra probablement vous aider. De plus, travailler un peu avec les pointeurs ne peut que vous aider à mieux comprendre leurs principes, si ce n'est pas encore fait. Je vous conseille donc de ne pas négliger les quelques fonctions que nous allons à présenter développer.

J'insiste sur le fait que je ne réimplémente ici ces fonctions qu'à titre d'exercice, et que c'est chose qu'il est absolument ridicule de faire dans un programme utile ! Si des fonctions sont fournies dans les bibliothèques, ce n'est pas pour que vous les redéveloppiez dans vos programmes !

A: strlen

La première fonction que j'ai choisi de réimplémenter est la plus simple de celles que nous avons étudié au cours de ce chapitre ; il s'agit de strlen.
Pour rappel, strlen prend en paramètre une chaîne de caractères, et retourne le nombre de caractères qu'elle contient, sans compter le 0 de fin de chaîne.

Voici la première implémentation que je vous propose de découvrir :

```
unsigned long my_strlen_1(char *str)
{
    unsigned long len = 0;
    while(str[len] != '\0')
    {
        len++;
    }
    return len;
}
```

Le principe est simple, on déclare une variable qui va servir à mémoriser la longueur de la section de chaîne déjà parcourue, et on l'initialise à 0, puisqu'on va commencer le parcours par le début de la chaîne.
Ensuite, on boucle en incrémentant notre compteur de 1 à chaque fois passage dans la boucle, ce qui nous permet de parcourir la chaîne caractère par caractère, du début vers la fin.
Cette boucle dure jusqu'à ce que le caractère courant soit '\0', c'est-à-dire 0. Ce caractère étant le marqueur de fin de chaîne, on met fin au parcours.
Et enfin, on retourne la valeur du compteur de nombre de caractères, qui correspond au nombre de caractères qui précédent le 0 de fin de chaîne.

C'est l'algorithme que nous continuerons d'utiliser pour strlen. Nous ne changerons que la façon d'écrire, afin d'arriver à du code légèrement plus concis.

Première petite modification : le caractère '\0' valant, comme le nom "0 de fin de chaîne" l'indique, zéro, et le 0 étant considéré comme faux dans les tests, on peut ré-écrire la condition de la boucle pour supprimer le test inutile :

```
unsigned long my_strlen_2(char *str)
{
    unsigned long len = 0;
    while(str[len])
    {
        len++;
    }
    return len;
}
```

Le reste de la fonction ne change pas encore.

A présent, dans la condition de la boucle, plutôt que d'accéder au caractère courant de la chaîne en indiçant un tableau, accédons-y par pointeur : à chaque cycle, on incrémente le pointeur correspondant à la chaîne, de sorte à ce qu'il pointe successivement sur le premier, puis le second, puis le troisième, et ainsi de suite.
Et on accède au caractère courant en déréférençant ce pointeur.

```
unsigned long my_strlen_3(char *str)
{
    unsigned long len = 0;
    while(*str++)
    {
        len++;
    }
    return len;
}
```

On a pour l'instant conservé la variable servant à calculer la longueur de la chaîne de caractères, en continuant à l'incrémenter à chaque itération de la boucle.

Il reste une idée que l'on pourrait mettre en application. En effet, en regardant le code de la fonction que nous venons d'écrire, nous réalisons encore deux incrémentations par boucle : celle du pointeur, et celle de la variable destinée à contenir la longueur de la portion de chaîne déjà parcourue.
Pourquoi n'essayerions-nous pas de supprimer une de ces deux incrémentations ?
Après tout, cela est possible : supprimons l'incrémentation de la variable len... et la variable aussi, par la même occasion (si on ne l'incrémente plus, elle ne sert à rien).
Maintenant, il nous faut un moyen de déterminer de combien de cases le pointeur a été avancé...
Une solution pour cela est de mémoriser, au début de la fonction, sur quelle case il pointe. Et, à la fin de la fonction, nous calculons la différence entre le pointeur str, qui correspond à l'adresse de fin de chaîne, et le pointeur qui a mémorisé l'adresse de début de celle-ci.
Et cette différence, d'après les règles d'arithmétique de pointeurs, correspond au nombre de cases qui séparent les deux pointeurs, c'est-à-dire, au nombre de caractères que la chaîne comporte.
Etant donné que nous avons choisi d'écrire l'instruction d'incrémentation du pointeur directement dans la condition de la boucle, le corps de celle-ci est désormais vide. Nous le remplaçons donc par un point-virgule, qui signifie "instruction vide", plus significatif qu'une paire d'accolades ne contenant rien.

```
unsigned long my_strlen_4(char *str)
{
    char *debut = str;
    while(*str++);
    return str-debut;
}
```

En utilisant cette méthode par rapport à celle que nous employions avec l'implémentation précédente, nous réalisons une soustraction de plus, certes, mais nous écomisons autant d'incrémentations (d'additions, donc) qu'il y a de caractères dans la chanîe de caractères.
Autrement dit, la méthode que nous employons ici a toutes les chances d'être plus rapide que celle utilisée auparavant.

Pour les autres fonctions que nous allons ré-implémenter, nous utiliserons à nouveau ce que nous avons employé ici. Nous ne ré-expliquerons pas tout en détail, puisque nous venons déjà de le faire.

B: strcpy

A présent, nous allons nous pencher sur la fonction de copie de chaînes, strcpy.
Comme nous l'avons vu plus haut, cette fonction prend en paramètres un pointeur sur un bloc mémoire et une chaîne de caractères, et copie la chaîne de caractères vers le bloc mémoire, qui est supposé suffisamment grand pour pouvoir contenir la copie de la chaîne de caractères. Cette fonction retourne un pointeur sur la chaîne de destination.
Globalement, l'algorithme utilisé est simple : on parcourt la chaîne de caractères d'origine caractère par caractère, et, au fur et à mesure, on copie ces caractères vers le bloc mémoire de destination, sans oublier, bien entendu, le 0 de fin de chaîne.

Notre première implémentation est la suivante :

```
char *my_strcpy_1(char *dest, char *orig)
{
    char *debut = dest;
    unsigned short position = 0;
    while(orig[position] != '\0')
    {
        dest[position] = orig[position];
        position++;
    }
    dest[position] = '\0';
    return debut;
}
```

Tout d'abord, nous déclarons un pointeur que nous faisons pointer sur le bloc mémoire de destination, puisque c'est l'adresse de celui-ci que nous devons retourner. Etant donné que, dans cette implémentation, nous ne modifions pas l'adresse de dest, nous aurions pu ne pas utiliser le pointeur debut, et directement retourner dest. Cela dit, par la suite, nous modifierons dest... alors, après tout, autant prendre de bonnes habitudes de suite.
Ensuite, nous déclarons une variable entière qui va servir à mémoriser la position dans les deux chaînes du caractère sur lequel nous travaillons actuellement. Etant donné qu'on parcourt la chaîne en allant du début vers la fin, cette variable de position est initialisée à 0.
La boucle while de notre fonction a un fonctionnement simple : comme pour strlen, nous parcourons la chaîne de caractères caractère par caractère, et nous nous arrêterons au '\0' de fin de chaîne. Dans le corps de la boucle, nous copions le caractère présent à la position courante dans la chaîne d'origine vers la même position dans la chaîne de destination.
Une fois arrivé à la fin de notre boucle, il nous faut ajouter à la fin de la chaîne de destination de '\0' de fin de chaîne, puisque cela n'a pas été fait dans la boucle (une fois qu'on a atteint le 0 de fin de chaîne dans la chaîne d'origine au niveau de la condition de la boucle, on n'entre pas dans son corps, et on ne fait donc pas l'affectation du 0 de fin de chaîne de la chaîne d'origine vers la chaîne de destination... il convient donc de faire cette affectation après la boucle).
Et enfin, on retourne le pointeur sur la chaîne de destination.

Pour notre seconde implémentation de strcpy, nous allons profiter d'une propriété du C, qui dit qu'une affectation a pour valeur celle qu'elle permet de stocker dans son opérande gauche. Autrement dit,

```
a = b+c
```

vaut a : la somme de b et c est calculée, mémorisée dans a, et toute l'affectation vaut a.

Nous allons déplacer l'affectation depuis le corps de la boucle vers la condition. De la sorte, l'affectation sera fera maintenant avant de tester si on est en fin de chaîne, le test lui-même se faisant en tirant profit de la propriété que nous venons d'énoncer.

```
char *my_strcpy_2(char *dest, char *orig)
{
    char *debut = dest;
    unsigned short position = 0;
    while((dest[position] = orig[position]) != '\0')
    {
        position++;
    }
    return debut;
}
```

De la sorte, ce n'est qu'après avoir copié un caractère de la chaîne d'origine vers la chaîne de destination qu'on vérifiera si ce caractère correspondait au '\0' de fin de chaîne.
Ainsi, nous n'avons plus à placer le '\0' à la fin de la chaîne de destination après la fin de la boucle, comme nous le faisions dans notre implémentation précédente.
Rien d'autre n'a été modifié pour cette version.

A présent, passons à l'utilisation de pointeurs, plutôt que de tableaux et d'indexes.
Les modifications sont proches de celles que nous avons effectué précédemment pour strlen ; je ne les détaillerai donc pas. L'algorithme reste exactement le même que pour notre implémentation précédente.

```
char *my_strcpy_3(char *dest, char *orig)
{
    char *debut = dest;
    while((*dest++ = *orig++) != '\0');
    return debut;
}
```

La variable de position disparaît ; elle n'est plus nécessaire, puisque nous procédons par incrémentation de pointeurs.

Enfin, exactement de la même manière que pour strlen, le test d'égalité entre le caractère courant (résultat de l'affectation) et '0\' est redondant, puisque '\0' vaut 0, et que 0 est équivalent à faux dans un contexte de condition.

```
char *my_strcpy_4(char *dest, char *orig)
{
    char *debut = dest;
    while((*dest++ = *orig++));
    return debut;
}
```

Aucune autre modification n'a été apportée par rapport à l'implémentation précédente.

Encore une fois, utiliser des pointeurs nous a permis d'utiliser de réduire le nombre de lignes de notre fonction, la rendant plus concise.
Cet exemple nous a aussi permis de réaliser l'importance que de petites "astuces" peut prendre. J'ajouterai que ce sont elles qui permettent souvent de faire la différence entre un programmeur C, et un programmeur C expérimenté : ce n'est généralement que lorsque l'on connaît bien le langage C que l'on est à même de se souvenir de certaines de ses particularités de ce genre, qui, même si elles sont loin d'être indispensables à l'écriture de programmes évolués, permettent parfois de se simplifier quelque peu la vie.
(Il aurait probablement été possible de se passer de ce genre d'astuce en utilisant une boucle do...while au lieu d'une boucle while... Mais je souhaitais attirer votre attention sur le fait que le C est un langage plein de subtilités, créées pour permettre des écritures concises, même si leur maîtrise n'est pas nécessairement indispensable.

C: strcat

A présent, je vais vous proposer une implémentation de strcat, qui permet de concaténer deux chaînes de caractères.
Etant donné que le fonctionnement de cette fonction est proche de celles que nous venons de voir, je ne donnerai qu'une seule écriture, et peu d'explications.
En effet, voici ce que strcat fait :

- Tout d'abord, on parcourt la première chaîne de caractères jusqu'à son 0 de fin.
- Ensuite, on recule d'une case, pour que la première écriture de celles qui suivront écrase le 0 de fin de la première chaîne, puisque l'on veut ajouter le texte de la seconde chaîne à la fin de la première.
- A présent, on parcourt la seconde chaîne de caractères, en la copiant au fur et à mesure à la suite des caractères qui étaient déjà présents dans la première.
- Et enfin, on retourne l'adresse du début de la chaîne de destination, qu'on avait pris soin de sauvegarder au début de la fonction.

En somme, la première étape correspond à ce que l'on a vu dans strlen, et les suivantes, à ce qu'on a fait pour strcpy.

Voici l'implémentation de que je vous propose :

```
char *my_strcat_1(char *dest, char *orig)
{
    char *debut = dest;
    while(*dest++); // On se place sur le 0 de fin de chaine
    dest--; // On revient avant le 0 de fin de chaine
    while((*dest++ = *orig++)); // et on copie de orig vers dest
    return debut;
}
```

Si vous ne comprenez pas cet exemple de code, je vous conseille de relire ce que nous avons étudié pour strlen et strcpy : strcat n'est quasiment rien de plus qu'une combinaison du code écrit pour ces deux autres fonctions.

D: strcmp

La dernière fonction que j'ai choisi de vous proposer pour cette partie est la fonction strcmp, qui permet, comme nous l'avons vu précédemment, de comparer deux chaînes de caractères.
Comme nous l'avons dit, si les deux chaînes sont égales, strcmp renvoie 0. Sinon, elle renvoie la différence entre les deux premiers caractères différents d'une chaîne à l'autre.

Pour les quatre implémentations que je vous propose, nous retournons toujours la même chose, que ce soit en écriture avec des tableaux indicés, ou des pointeurs ; je ne reviendrai donc pas dessus : il s'agit simplement de la soustraction évoquée plus haut.

Pour notre première implémentation, nous allons utiliser, pour parcourir nos deux chaînes, des tableaux indicés. Le parcours de chaînes en lui-même n'est pas une nouveauté, puisque nous en avons déjà effectué pour les fonctions précédentes ; je ne reviendrai donc pas dessus en détails.
Voici le code correspondant à notre première version de strcat:

```
short my_strcmp_1(const unsigned char *str1, const unsigned char *str2)
{
    unsigned short position = 0;
    while(str1[position]!='\0' && str2[position]!='\0' && str1[position]==str2[position
])
    {
        position++;
    }
    return str1[position] - str2[position];
}
```

Ce qui est intéressant ici est la condition de la boucle : celle-ci s'exécute tant que :

- `str1[position]!='\0'` : on n'a pas atteint la fin de la première chaîne de caractères.
- `str2[position]!='\0'` : et tant qu'on n'a pas atteint la fin de la seconde chaîne de caractères.
- `str1[position]==str2[position]` : et tant que les caractères des deux chaînes sont égaux.

Autrement dit, on met fin à la boucle si on atteint la fin d'une des deux chaînes, ou si le caractère à la position courante dans la première chaîne est différent du caractère à la position courante dans la seconde.

Pour notre seconde implémentation, nous allons réfléchir un peu plus au sujet de la condition de notre boucle, en vue de l'optimiser. Tout d'abord, nous supprimons les tests avec '\0', qui sont redondants, puisque '\0' est faux.
Mais ensuite, demandons-nous pourquoi faire ces trois tests ? Est-ce qu'on ne pourrait pas n'en faire que deux ?
Certes, il faut vérifier qu'on n'est à la fin ni de la première chaîne, ni de la seconde... Mais pourquoi tester si str1[position] et str2[position] sont non-nuls, alors que nous testons aussi si str1[position] est égal à str2[position] ?
Nous avons plusieurs cas devant nous, en réfléchissant :

- str1[position] est nul. On quitte la boucle sans rien tester d'autres.
- str1[position] est non nul. On teste l'égalité entre str1[position] et str2[position], et s'ils sont différents, on quitte la boucle.
- Mais dans le cas où str2[position] est nul, soit str1[position] sera nul aussi, auquel cas on aura quitté la boucle à cause du premier test, soit str1[position] ne sera pas nul, auquel cas str1[position] sera différent de str2[position], et on quittera la boucle à cause du second test. En somme, il n'est pas utile de tester directement si str2[position] est ou non nul ; cela sera de toute façon fait indirectement.

Voici le code correspondant :

```
short my_strcmp_2(const unsigned char *str1, const unsigned char *str2)
{
    unsigned short position = 0;
    while(str1[position] && str1[position]==str2[position])
    {
        position++;
    }
    return str1[position] - str2[position];
}
```

Certes, supprimer **une** condition, ce n'est pas grand chose... Mais **une** condition **sur trois**, c'est déjà une optimisation conséquente, surtout sur une fonction aussi courte !
Encore une fois, nous voyons que sans trop réfléchir, nous pouvons parfaitement écrire des programmes qui fonctionnent... Mais que réfléchir un peu nous permet de les rendre efficaces.

A présent, passons à un accès aux chaînes par pointeurs plutôt que par tableaux. La seule chose qui change est au niveau des écritures, qui nous permettent de nous débarrasser de la variable de position :

```
short my_strcmp_3(const unsigned char *str1, const unsigned char *str2)
{
    while(*str1 && *str1==*str2)
    {
        str1++;
        str2++;
    }
    return *str1 - *str2;
}
```

Aucune autre modification n'a été apportée pour cette version.

Et enfin, uniquement à titre de curiosité, voici comment, en utilisant une boucle for plutôt qu'une boucle while, nous pourrions écrire cette fonction en deux lignes :

```
short my_strcmp_4(const unsigned char *str1, const unsigned char *str2)
{
    for( ; *str1 && *str1==*str2 ; str1++, str2++);
    return *str1 - *str2;
}
```

Il s'agit en fait exactement du même code source, sauf que nous avons changé le type de boucle, tirant profit du fait que la boucle for comprend une partie de test et une autre "d'incrémentation" au sein même de son instruction de boucle.
Cela dit, l'implémentation précédente, utilisant une boucle while, était tout aussi efficace, et sans le moindre doute plus lisible. J'aurais donc, personnellement, tendance à vous conseiller de la préférer à celle-ci : mieux vaut un code source plus lisible, quitte à ce qu'il fasse quelques lignes de plus !

Nous voici arrivé à la fin de ce chapitre du tutorial. Le suivant va nous permettre d'étudier comme regrouper plusieurs données au sein d'une même variable, en utilisant ce que le langage C appelle "structure".

Chapitre 14

Regrouper des Données : Structures, Unions, et Champs de Bits

Ce chapitre va nous permettre d'étudier les structures, les unions, et les champs de bits, qui sont plusieurs méthodes que le C met à notre disposition pour regrouper des données.

I:\ Structures

Lorsque l'on programme, il arrive que l'on ressente le besoin de regrouper des données au sein d'une seule variable.
Au cours de notre Tutorial, nous avons eu l'occasion de parler des tableaux, qui permettent de regrouper en une variable des données qui sont toutes du même type.
A présent, nous allons étudier les structures, qui sont le moyen que fournit le langage C pour regrouper plusieurs données de types qui peuvent être différents au sein d'une seule variable, d'un seul "enregistrement".

A: Déclaration et utilisation

Pour déclarer une structure, on utilise le mot-clef `struct`, de la manière qui suit :

```
struct
{
    type1 variable1;
    type2 variable2;
    type3 variable3;
} nom_structure;
```

Ceci va déclarer une variable nommmée `nom_structure`, qui correspondra à une structure contenant trois variables, de types que vous demandez.

Ensuite, pour utiliser cette variable, et les éléments la composant, on utilisera une syntaxe de ce type :

```
nom_structure.variable
```

Ceci pouvant être utilisé aussi bien en partie gauche, qu'en partie droite, d'une expression, tout comme n'importe quel autre type de variable que nous avons jusqu'à présent étudié.

B: Exemple

Par exemple, pour déclarer une variable nommée `ma_structure`, et contenant deux entiers `a` et `b`, nous utiliserons la syntaxe suivante :

```
struct
{
      short  a;
      short  b;
}  ma_structure;
```

Ensuite, il nous est possible d'utiliser cette structure en accèdant à ses composantes de la manière dont nous l'avons indiqué un peu plus haut.
Pour affecter des valeurs aux deux variables composant la structure :

```
ma_structure.a  =  10;
ma_structure.b  =  20;
```

Et, pour relire ces valeurs, nous utilisons la même écriture :

```
printf("%d ; %d",  ma_structure.a,  ma_structure.b);
```

Ce qui nous permettra d'afficher ces deux valeurs.

C: Soyons brefs !

Devoir écrire tout ce qui compose la structure à chaque fois que nous souhaitons en déclarer une est extrêmement long... surtout si nous avons un grand nombre de variables à déclarer.
Bien évidemment, nous pourrions utiliser ceci :

```
struct
{
      short a;
      short b;
} var_1, var_2, var_3, var_4;
```

Ce qui nous permet de déclarer, ici, quatre variables correspondant à notre structure.
Mais si nous souhaitons déclarer des variables de ce type dans, par exemple, deux fonctions, il nous faudra écrire la description complète de la structure deux fois, comme ceci :

```
void fonction_1(void)
{
    struct
    {
        short a;
        short b;
    } ma_variable_1;

    ma_variable_1.a = 10;
    ma_variable_1.b = 20;

    // ...

} // Fin fonction_1

void fonction_2(void)
{
    struct
    {
        short a;
        short b;
    } ma_variable_2;

    ma_variable_2.a = 135;
    ma_variable_2.b = 246;

    // ...

} // Fin fonction_2
```

Si nous souhaitons apporter une modification à cette structure, il nous faudra la modifier en deux endroits, ce qui peut rapidement devenir une source d'erreurs.

Voila pourquoi il est possible de donner un nom à la strucure, comme ceci :

```
struct ma_structure
{
	short a;
	short b;
};
```

Et ensuite, chaque fois que nous voudrons déclarer une variable de type correspondant à cette structure, nous utiliserons ce type de syntaxe :

```
struct ma_structure ma_variable_3;
```

Ce qui est plus court, et, bien évidemment, plus sécurisé, puisqu'il n'y a plus, en cas de besoin, qu'à modifier la structure en un seul endroit.

Cependant, le C, avec ses habitudes de concision, permet d'aller encore plus loin.
En effet, il nous est possible d'indiquer au compilateur qu'il doit considérer notre structure comme un type, au même sens que `short` ou `float`, par exemple.
Pour cela, nous utiliserons l'instruction `typedef`, de la manière suivante :

```
typedef struct
{
	short a;
	short b;
} MON_NOM_TYPE;
```

Ceci va indiquer au compilateur qu'il doit considérer `MON_NOM_TYPE` comme un nouveau type de données, correspondant à notre structure.
Notons que, en général, nous écrivons en majuscules les noms de types que nous définissons à l'aide d'un `typedef`, afin de plus rapidement pouvoir les identifier, dans le but de s'y retrouver au sein de notre code source.
Ensuite, pour déclarer une variable du type que nous avons choisi pour notre structure, nous agirons exactement de la même manière que pour tout autre type de variable :

```
MON_NOM_TYPE ma_variable_5;
```

Ceci va nous permettre de déclarer une variable, nommée `ma_variable_5`, de type correspondant à notre structure.
Par la suite, nous pourrons l'utiliser exactement de la même manière que ce que nous avons fait jusqu'à présent :

```
ma_variable_5.a = 89;
ma_variable_5.b = 90;
```

Le choix d'utiliser un `typedef` ou d'en rester à la solution que je proposais juste au dessus (conserver le mot-clef `struct`) dépend généralement des programmeurs...
Personnellement, je préfére utiliser un `typedef` ; mais le choix final vous revient.

D: Affectation de structures

Le langage C permet d'effectuer des affectations de structures, en utilisant l'opérateur d'affectation =, comme pour n'importe quel autre type de données.
Notez que celui-ci affecte l'ensemble des champs de la structure ; si vous ne souhaitez copier que quelques uns de ses champs, il vous faudra les copier un par un.
De plus, il n'est possible d'affecter que des structures de même type !

Voici un exemple d'affectations de structures :

```c
#include <tigcclib.h>

typedef struct
{
    short a;
    float y;
    long d;
} MA_STRUCTURE;

void affiche(MA_STRUCTURE a);

void _main(void)
{
    clrscr();

    MA_STRUCTURE a, b;

    a.a = 10;
    a.y = 3.14;
    a.d = 567;

    affiche(a);

    b = a;

    affiche(b);

    b.y = 35.6;

    affiche(b);

    ngetchx();
}

void affiche(MA_STRUCTURE a)
{
    printf("%d - %f - %ld\n", a.a, a.y, a.d);
}
```

Exemple Complet

En quelques mots, voici ce que nous faisons au cours de cet exemple :
Nous commençons par déclarer un type correspondant à une structure : `MA_STRUCTURE`.
Ensuite, nous déclarons le prototype d'une fonction que nous avons choisi de nommer `affiche`.
Cette fonction, dont la définition est placée en fin de notre code source, n'a d'autre but que l'affichage du contenu d'une variable de type de structure avec lequel nous avons choisi de travailler.

Après cela, notre fonction principale efface l'écran, déclare deux variables du type correspondant à notre structure, initialise les éléments de l'une d'entre elle, affecte celle-ci à la seconde, puis modifie l'un des champs de la seconde.
Le tout en affichant, à chaque étape, le contenu de la structure sur laquelle nous venons de travailler.

F: Initialisation d'une structure

Le langage C nous permet d'initialiser une structure entière en une seule affectation, de la manière qui suit :

```
MA_STRUCTURE a = {10, 3.14, 567};
```

Notez toutefois que ce type d'affectation ne peut se faire qu'à la déclaration, et nulle part ailleur. De plus, en utilisant cette syntaxe, vous devez initialiser tous les champs de la structure.

Si vous ne souhaitez initialiser que certains des champs de la structure, vous pouvez, sous TIGCC (en tant que dérivé de GCC), utiliser la syntaxe illustrée ci-dessous :

```
MA_STRUCTURE a = {.a=10, .d=567};
```

L'effet sera exactement le même que si vous initialisiez vous-même à 0 les champs non initialisés, y compris en occupation mémoire - en fait, le compilateur fera l'initialisation des champs non précisés lui-même, sans vous l'indiquer.
En somme, cette syntaxe n'est, par rapport à la précédente, qu'un raccourci d'écriture, prévue pour vous éviter d'avoir à préciser des valeurs pour les champs qui ne vous intéressent pas.

E: Structures et pointeurs

Pour en terminer avec les structures, nous allons traiter de l'utilisation de structures via des pointeurs.
Pour cette partie, nous allons considérer que nous avons déclaré une structure, et un pointeur sur une structure, de la manière suivante ; toujours, naturellement, en travaillant sur le type de structure que nous utilisons depuis tout à l'heure :

```
MA_STRUCTURE a = {10, 3.14, 89};
MA_STRUCTURE *p;
```

Exactement comme avec les types avec lesquels nous avons déjà travaillé au cours de ce tutorial, la seconde ligne permet de déclarer un pointeur, nommé `p`, à même de pointer sur une structure du type qui nous intéresse.
A présent, faisons pointer notre pointeur sur la variable de type structure que nous avons déclaré :

```
p = &a;
```

Encore une fois, cela se fait de la même manière que pour les autres types de variables : en utilisant l'opérateur unaire &.

Pour ce qui est d'accèder à un champ de la structure, il nous faut commencer par accèder à celle-ci, en utilisant, comme pour n'importe quel pointeur, l'opérateur `*` ; ensuite, comme nous l'avons vu précédemment, nous utilisons l'opérateur `.` pour accèder au champ qui nous intéresse.
Par exemple :

```
(*p).a = 16;
```

Cela dit, ce type d'écriture est particulièrement lourd, notamment du fait que les parenthèses sont nécessaires, en raison de priorité des opérateurs...
Voila pourquoi le C, fidèle à ses habitudes de concision, propose un moyen alternatif : l'opérateur `->` que nous pouvons utiliser de la manière qui suit :

```
p->a = 16;
```

Ceci correspondant exactement à la même écriture que la précédente.

II:\ Unions

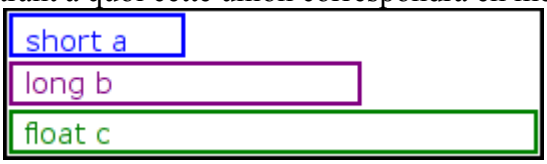
Comme nous l'avons vu un peu plus haut, une structure permet d'enregistrer plusieurs données, éventuellement de types différents, en les regroupant au sein d'une seule variable.
Une `union` permet de désigner un seul espace mémoire, et de permettre d'y accèder de plusieurs moyens différents. En somme, une union est un regroupement de plusieurs types de données, qui permettent tous d'accèder au même emplacement mémoire.

A: Déclaration et Utilisation

Voici comment on déclare une `union` ; exactement de la même manière que pour une structure :

```
union
{
    short a;
    long b;
    float c;
} mon_union;
```

Voici un rapide schéma montrant à quoi cette union correspondra en mémoire :



Le type `float` étant, des trois types que nous avons groupé au sein de notre union, celui qui occupe le plus de place en mémoire, une variable du type de cette union occupera en mémoire la taille d'un `float`, dans ce cas.
Ensuite, si nous adressons une variable du type de cette union par le champ de type `short`, nous n'utiliserons qu'une partie de l'espace mémoire occupé par cette variable, puisqu'un `short` est moins gros en mémoire qu'un `float`.

Pour ce qui est de l'utilisation, il en va exactement de la même manière que pour les structures.
Par exemple :

```
mon_union.a = 10;
mon_union.b = 70900;
mon_union.c = 3.14;
```

(En travaillant avec l'union que nous avons déclaré juste au dessus)

B: Soyons plus brefs !

De la même façon que pour les structures, il existe des syntaxes plus concises pour la déclaration d'unions :

```
union union_2
{
    short a;
    float b;
};

union union_2 a;

a.a = 10;
a.b = 3.14;
```

Et, en utilisant un typedef :

```
typedef union
{
    short a;
    float b;
} MON_UNION;

MON_UNION b;
b.a = 10;
b.b = 3.14;
```

Comme nous le voyons, dans un cas comme dans l'autre, la syntaxe est exactement la même que ce que nous avons présenté précédemment pour les structures.

C: Démonstration

A présent, pour que vous comprenniez bien le fonctionnement d'une union, et pour vous prouver que les différents champs la composant sont bien des alias du même espace mémoire, j'aimerais attirer votre attention sur la portion de code suivante :

```
typedef union
{
    short a;
    long b;
} MY_UNION;

MY_UNION test;

test.b = 0xA0B0C0D0;

clrscr();
printf("%X - %lX", test.a, test.b);
ngetchx();
```

Au cours de cet exemple, nous affectons la valeur 0xA0B0C0D0 au champ de type `long` de notre variable de type `union` capable de contenir un `long` et un `short`.
Ensuite, nous affichons le contenu de cette variable, en y accédant par le champ de type `long`, ce qui nous affiche bien la valeur à laquelle nous nous attendions, à savoir, 0xA0B0C0D0, ainsi que le contenu de cette même variable, en y accédant, cette fois-ci, par le champ de type `short` ; et là, la valeur affichée est 0xA0B0, ce qui correspond à la première moitié de l'espace mémoire correspondant à notre variable.
En somme, nous venons de prouver que les deux champs composant notre variable correspondent bien au même espace mémoire, de la taille du plus grand de ceux-ci.

D: Utilité ?

Maintenant que nous savons comment déclarer, et utiliser, des unions, je pense qu'un petit exemple concret de cas où l'emploi d'une union peut être bénéfique s'avére nécessaire, pour que vous comprenniez bien à quoi cela sert... Cet exemple nous permettra aussi d'utiliser une structure en "situation réelle".

Nous allons nous placer dans la situation d'un programmeur qui commence à réfléchir à la manière dont il pourait coder en mémoire les niveaux de son futur jeu de type Shoot'em Up (Pour quelques exemples correspondant à ce à quoi je pense, en sachant que j'ai volontairement simplifié mon exemple ici, pensez à des titres tels Solar Striker, ou Krypton - Cf TimeToTeam)

Nos niveaux peuvent contenir, sur chaque ligne, soit rien, soit un ennemi, soit un power-up, un niveau étant composé d'autant de lignes que nous le souhaitons, à raison d'un power-up ou ennemi par ligne du niveau. Ce niveau pourrait donc être enregistré sous forme d'un tableau d'unions représentant soit un ennemi, soit un power-up ; mais ceci ne nous regarde pas ici : c'est au niveau de l'implémentation du jeu lui-même que ceci devrait être décidé.

Dans notre niveau, un ennemi peut être décrit comme une structure ainsi définie :

```
typedef struct
{
	short nombre_canons;
	short vitesse;
	short energie;
}ENNEMI;
```

Comme nous pouvons le constater, nos ennemis disposent d'un nombre variable de canons, d'une vitesse donnée, et d'un certain nombre de points d'énergie, tout ceci étant défini par le créateur du niveau.

Pour ce qui est des power-ups, leur seul spécificité est le type de bonus dont il s'agit ; voici comment un power-up peut donc être défini au sein de notre niveau :

```
typedef struct
{
	short type_bonus;
}POWER_UP;
```

Ici aussi, donc, un typedef d'une structure.

Finalement, pour notre niveau, il pourrait nous suffir d'une union contenant soit un ENNEMI, soit un POWER_UP.
Cela dit, il nous faut un moyen de déterminer si l'espace mémoire correspondant à une ligne de notre niveau doit être interprété comme un descripteur d'ennemi, ou comme un descripteur de power-up. Pour cela, nous ajoutons un champ "type", pour lequel nous acceptons les valeurs 0 pour rien, 1 pour un ennemi, et 2 pour un power-up.
Voici la structure que nous utiliserons :

```
typedef struct
{
	short type;	// 1 => ennemi ; 2 => power-up ; 0 => rien
	short x, y;
	union
	{
		ENNEMI ennemi;
		POWER_UP power_up;
	};
}ENTITE;
```

Notons l'utilisation de deux champs nommés `x` et `y`, qui définiront la position de création de notre ennemi ou power-up ; pourquoi les placer à la fois dans l'union ENNEMI et dans l'union POWER_UP ? alors qu'il est si simple de ne les éfinir qu'une seule fois ?
Notons aussi que nous avons groupé nos deux types de descripteur au sein d'une union non nommée. Ceci est possible dans le cas ou notre union non nommée (ou une structure non nommée) est placée au sein d'une autre union ou structure, à condition qu'il n'y ait pas de définition ambigüe de noms de champs.

III:\ Champs de bits

Ce qu'on appelle, en C, un Champ de Bits, est une structure un peu particulière : au lieu que les champs la composant soient différentes variables de différents types, comme pour une structure "normale", les champs qui la composent sont en fait des portions d'un entier : comme son nom l'indique, un champ de bit est une structure dont les différents champs font le nombre de bits que vous voulez.
Pour déclarer un champ de bit, voici le type d'écriture que vous utiliserez :

```
typedef struct
{
      unsigned short
            champ_1: 6,
            champ_2: 6,
            champ_3: 1,
            champ_4: 1,
            champ_5: 2;
}BIT_FIELD_1;
```

Votre structure ne contient qu'un seul type entier, dont elle fera la taille (`short` ou `long`, par exemple), et vous découpez ce type en plusieurs champs, chacun faisant le nombre de bits que vous souhaitez.
Dans le cas de l'exemple que nous venons de présenter, notre champ de bits est composé de deux champs de 6 bits de long, de deux champs de 1 bit, et d'un champ de 2 bits, ce qui fait bien les 16 bits qui composent un `unsigned short`.

Pour accèder aux champs composant un champ de bits, vous agissez exactement de la même manière que pour accèder à ceux qui composeraient une structure : après tout, un champ de bits est une structure !
Par exemple :

```
BIT_FIELD_1 mon_champ;

mon_champ.champ_1 = 16;
mon_champ.champ_4 = 0;
```

Cette portion de code déclarant une variable du type du champ de bits que nous avons défini plus haut, et affectant des valeurs à deux des champs de celle-ci.

En deux mots, la principale utilité des champs de bits est de vous permettre de tirer profit de tous les bits d'un espace mémoire, ce qui est utile dans le cas où vous avez besoin de mémoriser beaucoup d'informations, en ne disposant que peu d'espace mémoire.

Nous venons d'atteindre la fin de ce chapître de notre tutorial, destiné à nous apprendre à regrouper des données par l'utilisation de structures, d'unions, et de champs de bits. Le prochain chapître va nous permettre de découvrir ce qu'est l'allocation dynamique de mémoire, et comment en tirer parti.

Chapitre 15

Allocation dynamique de Mémoire

Nous allons à présent aborder un chapitre qui peut paraître quelque peu déroutant au départ ; mais ce que nous y apprendrons se révèle rapidement être extrêmement utile, et important. Je ne peux donc que vous inciter à persévérer si vous ne saisissez pas tout dès la première lecture, car il est rare de ne pas être amené à rencontrer des "allocations dynamiques de mémoire" dans les programmes que l'on trouve sur le net, ou que l'on écrit, à partir du moment où l'on arrive à des programmes de taille non négligeable.

I:\ Allocation Dynamique de Mémoire ?

Il est relativement fréquent, lorsque l'on commence à développer des programmes réellement interactifs, d'avoir besoin, par moment, de disposer d'un espace mémoire dont la taille n'est soit pas connue à l'écriture du programme, soit trop grande pour pouvoir être allouée sur la pile.
Par exemple, puisque la pile ne fait que environ 16Ko, il n'est pas possible de déclarer un tableau de la manière qui suit, dans le corps d'une fonctions :

```
unsigned long tab[5000];
```

En effet, un `unsigned long` occupe 4 octets en mémoire ; cette déclaration demande 5000*4, soit 20000 octets... sur une pile qui n'en fait que 16000 et des miettes.
Cette déclaration posera donc, comme vous pouvez vous y attendre, des problèmes lors de l'exécution de votre programme.

La solution à ce problème est de demander de la mémoire ailleurs que sur la pile, c'est-à-dire, sur ce qu'on appelle, en programmation, le tas ("Heap" en anglais, par opposition à "Stack", la pile).
Sur nos calculatrice, la taille du tas est au maximum d'environ 180Ko, lorsque toute la mémoire RAM est libre ; naturellement, si l'utilisateur a laissé des variables en mémoire RAM sans les archiver, moins de mémoire sera disponible ; autrement dit, une allocation dynamique de mémoire peut échouer ; il vous faudra donc **toujours**, lorsque vous en effectuerez une, vérifier si elle a réussi.

D'autre part, la mémoire RAM est découpée en blocs de 64Ko ; il n'est donc pas possible d'allouer plus de 64Ko (moins quelques octets) à la fois.

Effectuer une allocation dynamique de mémoire, c'est demander au système de vous réserver un bloc de mémoire, que vous pourrez utiliser à votre convenance.
Cela dit, c'est à vous d'indiquer, avant la fin de votre programme, que vous souhaitez libérer ce bloc mémoire. Dans le cas contraire, il restera marqué comme réservé, et le système d'exploitation de la TI ne pourra plus l'utiliser ; au final, si vous allouez beaucoup de mémoire sans jamais la libérer, votre TI n'en n'aura plus de libre... ni pour vos programmes, ni pour son propriétaire... Et, dans ce cas de figure, la seule solution permettant de récupérer la mémoire perdue est d'effectuer un reset de la calculatrice.
Vous vous doutez bien que cette solution n'est pas appréciable pour les utilisateurs de votre programme... qui ne l'utiliseront probablement pas longtemps, s'il cause de telles fuites de mémoire.
Donc, lorsque vous allouez dynamiquement de la mémoire, pensez à toujours la libérer lorsque vous n'en n'avez plus besoin !

II:\ Comment faire, sur TI

Maintenant que nous avons vu en quoi l'allocation dynamique de mémoire consiste, ainsi qu'à quoi elle sert, nous allons étudier les différentes fonctions mises à notre disposition pour travailler avec le tas.

A: Allocation Mémoire

Tout d'abord, allouons de la mémoire.
Pour cela, nous utilisons la fonction `malloc`, dont le prototype est le suivant :

```
void *malloc(unsigned long taille);
```

Cette fonction prend en paramètre la quantité de mémoire, en octets, que l'on souhaite allouer, et retourne, en cas de succès, un pointeur vers la zone mémoire qui nous est nouvellement réservée. En cas d'échec, c'est un pointeur `NULL` qui nous est renvoyé.
La fonction `malloc` est un alias, conforme à la norme ANSI-C, de la fonction `HeapAllocPtr`, dont le nom provient du TIOS. Naturellement, cette fonction est définie exactement de la même manière que la première :

```
void *HeapAllocPtr(unsigned long taille);
```

Généralement, j'ai tendance à préférer la fonction `malloc`, du fait que c'est une fonction standard de la bibliothèque C, sous tous les compilateurs non préhistoriques. Cela dit, d'autres programmeurs, en particulier ceux ayant commencé la programmation sur TI par le langage d'Assembleur, préférent `HeapAllocPtr`. Encore une fois, c'est à vous de choisir selon vos préférences.

Je me permet d'insister sur la nécessité de vérifier que l'allocation a réussi : si la fonction d'allocation a renvoyé `NULL`, le bloc mémoire que vous souhaitiez réserver ne vous a pas été alloué ; il ne faut pas, en ce cas, travailler avec le pointeur retourné, sous peine de plantage (écriture vers l'adresse nulle, ou vers une zone mémoire non définie).

Par exemple, pour allouer un tableau de 5000 `unsigned long`, comme nous souhaitions le faire au début de ce chapître :

```
unsigned long *tab;

tab = malloc(5000*sizeof(unsigned long));
if(tab != NULL)
{
    // Utilisation de l'espace mémoire alloué
    // ...
    // /!\ Penser à libérer la mémoire !!!
}
else
{
    // Echec de l'allocation mémoire
}
```

Notez l'utilisation de `5000*sizeof(unsigned long)` : malloc attend une taille en octets... Puisque nous voulons allouer l'espace mémoire nécessaire pour 5000 `unsigned long`, il nous faut multiplier 5000 par la taille d'un `unsigned long`. Et c'est cette taille qui est retournée par `5000*sizeof(unsigned long)`.
N'oubliez pas ceci, ou il sera fréquent que nous allouiez moins de mémoire que ce que vous pensiez... et que cela soit à l'origine de plantages de votre programme.

B: Réallocation

Il arrive que l'on ait besoin de modifier la taille d'un bloc mémoire que nous avions alloué grâce à la fonction `malloc`.
Pour cela, il nous faudra utiliser la fonction `realloc`, dont le prototype est le suivant :

```
void *realloc(void *pointeur, unsigned long nouvelle_taille);
```

Le premier paramètre que prend cette fonction correspond à un pointeur vers un espace mémoire qui avait été dynamiquement alloué au préalable, via la fonction `malloc` ou une fonction équivalente. Si ce pointeur est nul, la fonction `realloc` agira de la même manière que la fonction `malloc`, que nous avons étudié plus haut.
Le second paramètre correspond à la nouvelle taille que nous souhaitons pour notre bloc mémoire. Celle-ci peut être inférieure, ou supérieure, selon vos besoins, à la taille d'origine.
En cas de succès, la fonction `realloc` retourne un pointeur sur la nouvelle zone mémoire, qui peut ou non être la même que celle qui était déjà à votre disposition, selon l'état dans lequel la mùmoire se trouvait. En cas d'échec à la réallocation, la fonction retourne `NULL` ; notez que, dans ce cas, vous ne devez pas considérer que les données pointées par le pointeur que vous aviez passé en premier paramètre soient toujours valides !

A titre d'illustration, voici un petit exemple de réallocation de l'espace mémoire que nous avons alloué un peu plus haut :

```
tab = realloc(tab, 6000*sizeof(unsigned long));
if(tab != NULL)
{
	// Utilisation de l'espace mémoire ré-alloué
	// ...
	// /!\ Penser à libérer la mémoire !!!
}
else
{
	// Echec de la ré-allocation mémoire
}
```

La remarque que j'ai fait pour `malloc` au sujet de `sizeof(unsigned long)` est valide ici aussi, bien entendu, puisque la taille attendue par `realloc` est exprimée en octets.

C: Libération de mémoire allouée

J'ai déjà parlé plusieurs fois de l'importance qui réside dans le fait de bien libérer l'espace mémoire que vous avez alloué... Il serait peut-être temps d'étudier comment cela se fait.
Pour libérer un espace mémoire que vous aviez demandé précédemment à l'aide de la fonction `malloc`, il convient d'utiliser la fonction `free`, dont le prototype est le suivant :

```
void free(void *pointeur);
```

L'emploi de cette fonction est des plus simple : elle prend en paramètre un pointeur sur l'espace mémoire à libérer, pointeur qui correspond à la valeur de retour de `malloc`, et ne retourne rien.
De la même façon que `HeapAllocPtr` est un alias de `malloc`, la fonction `free` est un alias de la fonction `HeapFreePtr`.

```
void HeapFreePtr(void *pointeur);
```

Ici encore, choisir entre l'une ou l'autre de ces deux fonctions est une histoire de goûts et d'habitudes... Personnellement, je préfère la fonction `free`, du fait qu'il s'agit d'une fonction ANSI-C ; mais libre à vous d'agir autrement.

Je me permet d'attirer votre attention sur le fait qu'il est nécessaire que le paramètre que vous passez à la fonction `free` soit valide, et corresponde bien au pointeur qui avait été retourné par une fonction d'allocation mémoire.
Si le pointeur que vous passez en paramètre n'est pas valide, il est fort probable que votre programme plantera.

Et voici un petit exemple, même si je doute de son utilité :

```
unsigned long *tab = malloc(5000*sizeof(unsigned long));
if(tab != NULL)
{
	// Utilisation de l'espace mémoire alloué
	// ...
	free(tab);
}
else
{
	// Echec de l'allocation mémoire
}
```

Comme précédemment, nous allouons un espace mémoire ; et si l'allocation a réussi, nous libérons le bloc mémoire que nous avions obtenu.

III:\ Exemple

Pour finir, voici un petit exemple de programme au sein duquel nous réalisons une allocation dynamique de mémoire :

```
#include <tigcclib.h>

void _main(void)
{
    short nombre_elements = random(10)+1;
    clrscr();

    unsigned long *tab = malloc(nombre_elements*sizeof(unsigned long));
    if(tab != NULL)
    {
        short i;
        for(i=0 ; i<nombre_elements ; i++)
        { // Remplissage du tableau...
            tab[i] = random(100000);
        }

        printf("%d éléments :\n", nombre_elements);
        for(i=nombre_elements-1 ; i>=0 ; i--)
        { // Affichage du tableau (en ordre inverse)...
            printf("%lu\n", tab[i]);
        }

        free(tab);
    }
    else {printf("Echec de l'allocation mémoire\n");}

    ngetchx();
}
```

Exemple Complet

Tout d'abord, nous utilisons la fonction `random`, qui retourne un nombre choisi aléatoirement entre 0 et la valeur passée en argument (exclue), de manière à obtenir un nombre valant entre 1 et 10 compris.
Ensuite, nous allouons une zone mémoire pouvant contenir ce nombre d'`unsigned long`.
Après quoi, nous initialisons chacune des cases de ce tableau à une valeur aléatoire, choisie entre 0 et 99999 compris, puis nous parcourons ce tableau en sens inverse, dans le but d'afficher les valeurs qu'il contient à présent.
Et enfin, nous libérons l'espace mémoire que nous avions alloué.
Naturellement, nous n'avons effectué nos manipulations sur l'espace mémoire alloéu uniquement si l'allocation mémoire n'avait pas échoué.

Pour cet exemple comme pour celui que nous avions étudié il y a quelques chapîtres de ça, il aurait été possible de se passer d'un tableau... Mais, après tout... c'était le but de notre exemple.

Notons, qu'il existe de nombreuses autres fonctions permettant de travailler avec la mémoire ; on peut par exemple déterminer combien de mémoire est disponible, verrouiller des blocs en mémoire, et pas mal d'autres choses encore.
Cela dit, ces fonctionnalités ne sont pas ANSI, et il n'est pas nécessaire, à mon avis, de les connaître, sauf si l'on souhaite effectuer des manipulations bien particulières ; si le besoin s'en fait sentir plus loin au cours de ce tutorial, nous étudierons certaines de ces fonctions.
Bien entendu, en plus de cela, rien ne vous empêche de consulter la documentation du fichier `alloc.h`.

Chapitre 16

Graphismes en Noir et Blanc

Nous avons déjà vu au cours de certains exemples de ce Tutorial, comment effacer l'écran, afficher une ligne, ou comment sauvegarder puis restaurer l'écran...
Mais ce chapître va nous permettre d'aller une étape plus loin, puisqu'il va nous permettre de découvrir une partie des fonctions du TIOS permettant de réaliser simplement des affichages graphiques, en noir et blanc.

I:\ Afficher du Texte

Nous avons déjà utilisé la fonction `printf` pour afficher du texte à l'écran... Mais nous allons voir, au cours de cette partie, qu'il y a d'autres possibilités, plus avancées.

A: Afficher une Chaîne de Caractères

Tout d'abord, rappelons rapidement que la fonction `DrawStr` permet d'afficher une chaîne de caractères à la position souhaitée sur l'écran ; voici son prototype :

```
void DrawStr(short x, short y, const char *str, short Attr);
```

Le dernier paramètre correspond à la façon dont la chaîne de caractères sera affichée : en noir sur blanc, en blanc sur noir, ...
Par exemple, exécutez la portion de code suivante, pour avoir quelques idées de ce qui est possible :

```
DrawStr(0, 0, "Hello World !", A_NORMAL);
DrawStr(0, 10, "Hello World !", A_REVERSE);
DrawStr(0, 20, "Hello World !", A_SHADED);
```

Et pour plus d'informations, notamment sur les autres types d'affichage possible, n'hésitez pas à consulter la documentation...

B: Taille de la Police

Cela dit, pourquoi se limiter à la taille de police par défaut, qui, qui plus est, n'est pas toujours la même, puisqu'elle dépend souvent de ce que vous faisiez avant de lancer votre programme, de la façon dont il a été lancé, et de la plate-forme sur laquelle il s'exécute ?
En effet, le TIOS vous propose une fonction qui permet de déterminer quelle est la police actuellement sélectionnée ; voici son prototype :

```
unsigned char FontGetSys(void);
```

La valeur retournée est soit `F_4x6`, qui correspond à la petite police, `F_6x8`, qui correspond à la police de taille moyenne, soit `F_8x10`, qui correspond à la grande taille de police.
Et, naturellement, voici la fonction qui permet de modifier la taille de police courante :

```
unsigned char FontSetSys(short Font);
```

Elle prend en paramètre la nouvelle taille de police, parmis les trois présentées juste au dessus, et retourne la taille de police précédemment sélectionnée.
Pour illustrer ceci, voici un bref exemple :

```
unsigned char save_taille = FontSetSys(F_4x6);
DrawStr(0, 0, "Bonjour !", A_NORMAL);
FontSetSys(F_6x8);
DrawStr(0, 10, "Bonjour !", A_NORMAL);
FontSetSys(F_8x10);
DrawStr(0, 20, "Bonjour !", A_NORMAL);
FontSetSys(save_taille);
```

Cette portion de code va commencer par sauvegarder la police courante, puis en changer trois fois, en affichant, à chaque fois, un bref message avec la police courante.
Enfin, nous restaurons la police de caractères qui avait été, au départ, sauvegardée.

C: Quelques autres fonctions

A présent, principalement pour montrer qu'il existe d'autres fonctions, en voici deux qui peuvent vous être utiles.

Tout d'abord, la fonction `DrawChar`, qui permet d'afficher le caractère que vous voulez à la position que vous voulez ; voici son prototype :

```
void DrawChar(short x, short y, char c, short Attr);
```

On spécifie le caractère par son code (vous trouverez la liste des codes de cacactères dans le manuel de votre calculatrice), et le dernier paramètre fait parti de la liste de ceux qui sont acceptés par la fonction `DrawStr`, que nous venons d'étudier.
Par exemple, pour afficher un caractère ©, nous pourrions utiliser l'instruction suivante :

```
DrawChar(10, 10, 169, A_NORMAL);
```

Sachant que le caractère © a pour code 169.

Une autre fonction qui peut vous être utile, par exemple si vous souhaitez centrer une chaîne de caractères sur l'écran, est la fonction `DrawStrWidth`, dont le prototype est le suivant ; elle permet de déterminer la longueur d'une chaîne de caractère en fonction d'une des trois tailles de police que nous avons déjà évoqué :

```
short DrawStrWidth(const char *str, short Font);
```

Et voici un petit exemple illustrant l'emploi de cette fonction :

```
char str[] = "Hello World !";
unsigned char save_taille_2 = FontSetSys(F_4x6);
printf("%s => %d\n", str, DrawStrWidth(str, FontGetSys()));
FontSetSys(F_6x8);
printf("%s => %d\n", str, DrawStrWidth(str, FontGetSys()));
FontSetSys(F_8x10);
printf("%s => %d\n", str, DrawStrWidth(str, FontGetSys()));
FontSetSys(save_taille_2);
```

Cet exemple affiche à l'écran, pour chacun des trois tailles de police, la longueur d'un bref message.

II:\ Afficher des Graphismes Simples

A présent, nous allons voir quelques fonctions nous permettant d'afficher des graphismes simples à l'écran. Nous commencerons par une fonction permettant d'afficher des pixels, puis nous passerons à l'affichage de lignes.

Notez que les fonctions que nous verrons au cours de cette partie ne fonctionnent que si leurs coordonnées sont valides : vous ne **devez pas** essayer d'afficher quelque chose en dehors de l'écran avec ces fonctions ! Dans le cas contraire, votre programme risque de planter !
Nous verrons au cours de la partie suivante qu'il existe d'autres fonctions, pour lesquelle ce problème ne se pose pas.

A: Afficher des Pixels

Tout d'abord, commençons par afficher des pixels seuls. Pour cela, on utilise cette fonction :

```
void DrawPix(short x, short y, short Attr);
```

Les deux premiers paramètres correspondent aux coordonnées à l'écran du pixel que l'on souhaite afficher, et le dernier correspond au mode d'affichage : A_NORMAL pour allumer le pixel, A_REVERSE pour l'éteindre, et A_XOR pour inverser son état.

Et voici un petit exemple :

```
DrawPix(10, 20, A_NORMAL);
DrawPix(11, 20, A_REVERSE);
DrawPix(12, 20, A_XOR);
```

Ici, on allume le pixel de coordonnées (10, 20), on éteint celui de coordonnées (11, 20), et on inverse l'état de celui de coordonnées (12, 20).

Simple, n'est-ce pas ?
Souvenez-vous juste que les coordonnées doivent correspondre à des points valides, ce qui signifie que x doit varier entre 0 et 239 compris, et que y doit être compris entre 0 et 127 inclus.

B: Afficher des Lignes

A présent, nous allons voir comment il nous est possible d'afficher des lignes à l'écran. Puisque vous connaissez le nom de la fonction d'affichage de pixels, il devrait vous être relativement simple de deviner celui de la fonction de tracés de lignes, dont le prototype est le suivant :

```
void DrawLine(short x0, short y0, short x1, short y1, short Attr);
```

Cette fonction prend en paramètres les coordonnées des deux points formants les extrémités de la ligne, ainsi que le mode dans lequel elle doit être affichée.
Pour plus de détails au sujet des modes disponibles, qui sont assez nombreux, je vous invite à consulter la documentation de TIGCC.

Et voici un petit exemple de programme traçant trois lignes en utilisant des modes différents :

```
#include <tigcclib.h>

void _main(void)
{
    ClrScr();

    DrawLine(10, 10, 50, 50, A_NORMAL);
    DrawLine(10, 20, 60, 60, A_THICK1);
    DrawLine(30, 10, 30, 80, A_XOR);

    ngetchx();
}
```

Exemple Complet
Vous remarquerez ici que le mode A_XOR a pour effet d'inverser la couleur de la ligne : les pixels qui étaient blancs deviennent noir, et vice-versa.

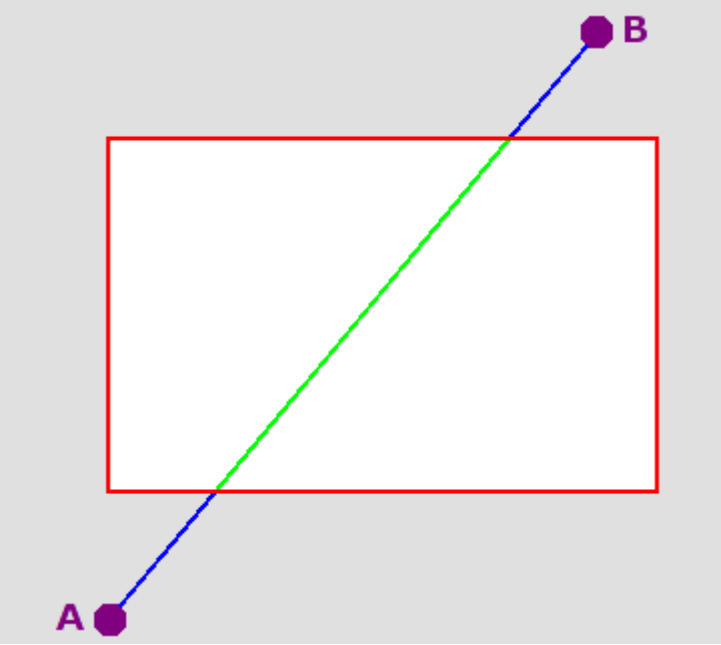
III:\ Graphismes et Clipping

Les fonctions que nous venons d'étudier nous permettent déjà d'afficher quelques graphismes simples... Cela dit, nous devons, en les employant, prendre garde à ne pas dépasser de l'écran, sous peine de plantages de notre programme.
Pour remédier à ce problème, nous allons voir qu'il existe la notion de "clipping", et des fonctions utilisant ce principe.

A: Notion de Clipping

"Clipper" un graphisme par rapport à une "zone de clipping" signifie ne conserver de ce graphisme que les parties qui se trouvent sur ladite zone de clipping.

Je pense que ceci sera plus compréhensible avec un exemple et un petit schéma...
Admettons que l'on veuille tracer une ligne entre deux points A et B. Ces deux points sont en dehors de la zone de clipping que l'on accepte, celle-ci étant souvent l'écran.
Etant donné qu'afficher des graphismes en dehors de l'écran peut provoquer un plantage de notre programme, il va nous falloir éliminer les portions de la ligne qui ne sont pas sur celui-ci, comme ceci :



La rectangle rouge correspond à notre zone de clipping ; les deux points A et B, les extrémités de la ligne, sont en dehors de celle-ci...
Le fait de clipper par rapport à notre zone de clipping aura pour effet d'éliminer les deux parties bleues de la ligne, ce qui nous permettra de ne conserver que la partie verte, que nous pourrons alors afficher sans danger.

B: Quelques fonctions graphiques avec clipping

Le TIOS nous fourni de nombreuses fonctions graphiques avec clipping. Nous n'en verrons que quelques unes, afin d'illustrer leur principe de fonctionnement, et je vous laisse vous reporter à la documentation pour toutes les connaître.

1: Clipping par défaut

Certaines fonctions du TIOS travaillent avec une zone de clipping par défaut. C'est par exemple le cas de la fonction suivante, qui permet d'afficher un pixel :

```
void DrawClipPix(short x, short y);
```

Pour ces fonctions, la zone de clipping doit être fixée en utilisant la fonction `SetCurClip`, dont le prototype est le suivant :

```
void SetCurClip(const SCR_RECT *clip);
```

Cette fonction prend en paramètre un pointeur sur une union de type `SCR_RECT`, qui est définie de la manière suivante :

```
typedef union {
    struct {
        unsigned char x0, y0, x1, y1;
    } xy;
    unsigned long l;
} SCR_RECT;
```

Donc pour définir une zone de clipping pour les fonctions telles `DrawClipPix`, il nous est possible d'utiliser quelque chose comme ceci (Dans ce cas, on définit comme zone de clipping une surface correspondant à l'écran d'une TI-92+ ou v200 ; on aurait pu choisir d'autres valeurs, naturellement) :

```
SCR_RECT rect;
rect.xy.x0 = 0;
rect.xy.y0 = 0;
rect.xy.x1 = 239;
rect.xy.y1 = 127;

SetCurClip(&rect);
```

Cela dit, si vous n'avez pas l'intention d'utiliser plusieurs fois votre variable que nous avons ici appelé "rect", vous pouvez, du fait que TIGCC est un compilateur GNU-C, utiliser ce type de syntaxe, plus concise :

```
SetCurClip(&(SCR_RECT){{0, 0, 239, 127}});
```

2: Clipping par fonction

En revanche, une bonne partie des fonctions de `graph.h` - la plupart, en fait - prennent la zone de clipping en paramètre, sous forme, ici aussi, d'une union de type `SCR_RECT`.
Par exemple, pour afficher un caractère, une élipse, ou un triangle, vous pourrez utiliser une des fonctions qui suit :

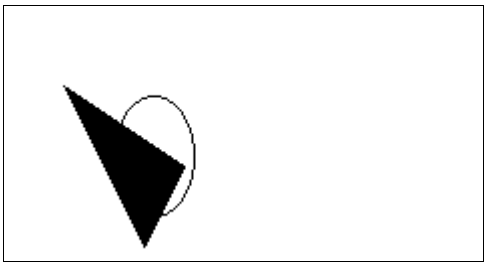
```
void DrawClipChar(short x, short y, short c, const SCR_RECT *clip, short Attr);

void DrawClipEllipse(short x, short y, short a, short b, const SCR_RECT *clip, short
Attr);

void FillTriangle(short x0, short y0, short x1, short y1, short x2, short y2, const
SCR_RECT *clip, short Attr);
```

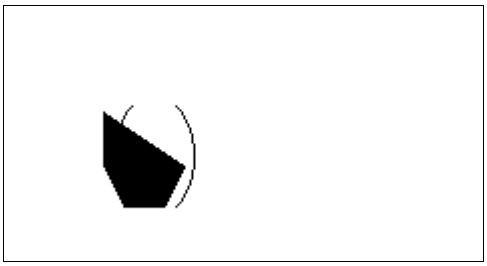
Et voici un petit exemple utilisant deux de ces fonctions :
La variable `ScrRect` est définie au niveau de TIGCC comme un pointeur sur un `SCR_RECT` faisant la taille de l'écran ; plus rapide à utiliser qu'à re-définir !

```
#include <tigcclib.h>

void _main(void)
{
    SCR_RECT rect;

    ClrScr();
    DrawClipEllipse(75, 75, 20, 30, ScrRect, A_NORMAL);
    FillTriangle(30, 40, 90, 80, 70, 120, ScrRect, A_NORMAL);
    ngetchx();


    rect.xy.x0 = 50;
    rect.xy.y0 = 50;
    rect.xy.x1 = 100;
    rect.xy.y1 = 100;

    ClrScr();
    DrawClipEllipse(75, 75, 20, 30, &rect, A_NORMAL);
    FillTriangle(30, 40, 90, 80, 70, 120, &rect, A_NORMAL);
    ngetchx();
}
```

Exemple Complet

Et voici les captures d'écran correspondantes : sur la première, la zone de clipping est l'écran. Sur la seconde, elle est plus petite ; les graphismes sont donc "coupés" :





Ce qui nous montre toute l'efficacité du clipping.

Pour finir, deux dernières fonctions que je souhaite vous présenter ici :

```
void DrawClipLine(const WIN_RECT *Line, const SCR_RECT *clip, short Attr);

void DrawClipRect(const WIN_RECT *rect, const SCR_RECT *clip, short Attr);
```

Comme leurs noms respectifs l'indiquent, ces fonctions permettent d'afficher une ligne et un rectangle, avec clipping.
Leur particularité, qui explique qu'elles méritent de figurer ici est qu'elles prennent en paramètre un pointeur sur une struture de type `WIN_RECT` pour désigner les deux points constituant les extrémités de la droite, ou les deux coins supérieur gauche et inférieur droit du rectangle.
Cette structure est définie ainsi :

```
typedef struct {
    short x0, y0, x1, y1;
} WIN_RECT;
```

Comme pour les unions de type `SCR_RECT`, on peut utiliser deux types de syntaxe :
Tout d'abord, la syntaxe la plus "conventionnelle", si je puis dire :

```
WIN_RECT rect;
rect.x0 = 50;
rect.y0 = 50;
rect.x1 = 100;
rect.y1 = 100;
DrawClipRect(&rect, ScrRect, A_NORMAL);
```

Ou alors, en utilisant les cast-constructor du GNU-C :

```
DrawClipLine(&(WIN_RECT){30, 40, 90, 78}, ScrRect, A_NORMAL);
```

Vous ne manquerez pas de noter que, une fois encore, on a utilisé `ScrRect` pour signifier que nous voulions utiliser tout l'écran comme zone de clipping.

IV:\ Manipulation d'Images

Pour terminer ce chapitre, nous allons voir une série de fonctions dont le but est de nous permettre d'enregistrer ou d'afficher des images complètes.

A:\ Afficher une image définie dans le programme

Tout d'abord, nous allons voir comment encoder une image directement dans notre programme, afin de l'afficher par la suite.
La fonction que nous utiliserons au cours de cette partie pour afficher des images est `BitmapPut` ; son prototype est le suivant :

```
void BitmapPut(short x, short y, const void *BitMap, const SCR_RECT *clip, short Attr);
```

Le troisième argument attendu par cette fonction est un pointeur sur une struture de type `BITMAP`, définie comme ceci :

```
typedef struct
{
    unsigned short NumRows, NumCols;
    unsigned char Data[];
} BITMAP;
```

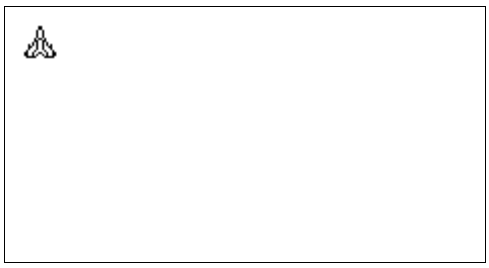
Cette structure correspond à un descripteur d'image : elle permet de mémoriser, en nombre de pixels, le nombre de lignes et le nombre de colonnes de l'image ; ces informations sont ensuite suivies des données de l'image elle-même, ligne par ligne, de gauche à droite.
Chaque pixel de l'image est encodé sous forme d'un bit, valant 1 si le pixel est noir et 0 sinon. Si la largeur de l'image n'est pas multiple de 8, les données des derniers octets de droite doivent être complétées par des 0, afin que l'on ait des octets complets.

Et voici un petit exemple où on déclare puis affiche une image de 16 pixels de coté :

```
#include <tigcclib.h>

void _main(void)
{
    ClrScr();

    static BITMAP bitmap =
    {
        16, 16,
        {
            0b00000001, 0b10000000,
            0b00000001, 0b10000000,
            0b00000010, 0b01000000,
            0b00000010, 0b01000000,
            0b00000010, 0b01000000,
            0b00000010, 0b01000000,
            0b00000101, 0b10100000,
            0b00000110, 0b01100000,
            0b00001010, 0b01010000,
            0b00101010, 0b01010100,
            0b00110010, 0b01001100,
            0b00100010, 0b01000100,
            0b01000100, 0b00100010,
            0b11001001, 0b10010011,
            0b11001010, 0b01010011,
            0b01111100, 0b00111110,
        }
    };

    BitmapPut(10, 10, &bitmap, ScrRect, A_NORMAL);

    ngetchx();
}
```

Exemple Complet

Et une capture d'écran de notre programme lors de son exécution :



Notons que BitmapPut est surtout adaptée à l'affichage de grandes images : pour des images de 8x8, 16x16, ou 32x32, nous aurions plutôt tendance, en pratique, à utiliser des fonctions d'affichage de sprites, telles celles que nous étudierons plus loin au cours de ce tutorial.

B:\ Sauvegarder une image, et l'afficher

A présent, nous allons voir comment sauvegarder une portion d'image affichée à l'écran, afin de pouvoir, par la suite, l'afficher avec `BitmapPut`, que nous avons déjà utilisé.
Pour enregistrer une portion d'image, nous utiliserons la fonction `BitmapGet`, dont le prototype est le suivant :

```
void BitmapGet(const SCR_RECT *rect, void *BitMap);
```

Cette fonction va sauvegarder les données contenues dans le rectangle décrit par son premier paramètre vers la zone mémoire sur laquelle pointera son second paramètre.
Cela signifie que nous allons avoir besoin d'une zone mémoire de taille suffisante... Pour calculer cette taille, nous allons utiliser la fonction `BitmapSize`, qui, elle aussi, prend en paramètre la zone de l'écran que l'on veut sauvegarder, et retourne le nombre d'octets dont nous allons avoir besoin :

```
unsigned short BitmapSize(const SCR_RECT *rect);
```

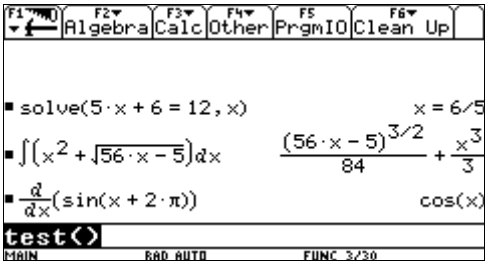
Maintenant que nous connaissons la taille du bloc mémoire dont nous avons besoin pour sauvegarder notre portion d'image, nous pouvons, par exemple, l'allouer grâce à un `malloc`, puis, appeler `BitmapGet`, en lui passant en paramètre l'adresse de ce bloc mémoire.
Pour réafficher notre image, nous faisons comme plus haut. Voici un exemple illustrant mes propos :

```
#include <tigcclib.h>

void _main(void)
{
    unsigned short taille = BitmapSize(&(SCR_RECT){{100, 40, 200, 110}});

    unsigned char *buffer = malloc(taille);
    if(buffer)  // Important !
    {
        BitmapGet(&(SCR_RECT){{100, 40, 200, 110}}, buffer);

        ClrScr();

        BitmapPut(10, 10, buffer, ScrRect, A_NORMAL);

        ngetchx();
        free(buffer);  // Important !
    }
}
```

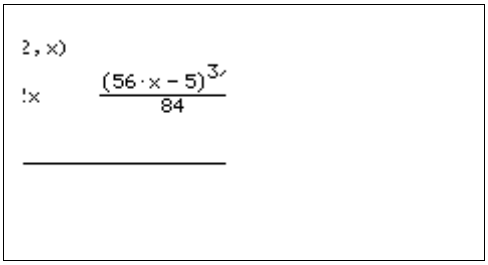
Exemple Complet

Vous noterez que, comme nous l'avons appris un peu plus tôt, nous avons pensé à vérifier que l'allocation mémoire avait bien réussi, et nous n'avons pas oublié de libérer la mémoire allouée.

Etant donné que le programme que nous venons d'écrire sauvegarde une portion de l'écran, il peut être intéressant de savoir ce que celui-ci contenait au moment où j'ai exécuté le-dit programme :



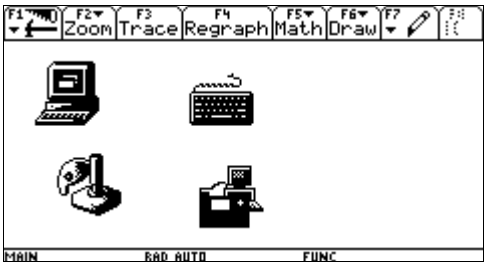
Et voici l'affichage obtenu au cours de l'exécution de notre programme : on reconnait aisément la portion d'écran sauvegardée, et ré-affichée :

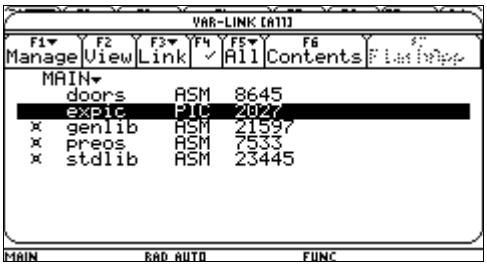


Naturellement, selon ce que vous avez d'affiché à l'écran au moment où vous lancez ce programme, il est probable que vous obteniez une sortie différente...
Notez aussi que le programme que j'ai choisi pour illustrer mes propos a été conçu pour une taille d'écran de TI-92+ ou de Voyage 200. Si vous avez une TI-89, je vous laisse le loisir de l'adapter - arrivé à ce niveau du Tutorial, vous devriez y parvenir sans difficulté.

C:\ Afficher une image contenue dans un fichier PIC

Pour le dernier point de cette partie, nous allons voir comment afficher à l'écran le contenu d'un fichier de type "PIC" - les images du TIOS. Pour l'exemple, je vais utiliser une des images fournie avec txtrider (notez que les images de txtrider sont compressées ; il m'a donc fallu la décompresser auparavant). Voici comment elle apparait une fois affichée dans l'écran graph, et dans le VAR-LINK :





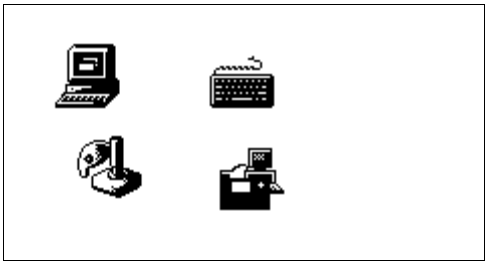
Pour afficher une image contenue dans un fichier PIC, nous allons utiliser une fonction fournie dans la FAQ de TIGCC. Tout simplement, cette fonction prend en paramètre le nom du fichier sous forme d'un `SYM_STR`, ce qui implique d'utiliser la macro `SYMSTR` pour l'obtenir depuis une chaine de caractères, ainsi que les coordonnées où l'on veut afficher l'image, et le mode dans lequel on veut réaliser la sortie écran.

Et voici un petit exemple utilisant la fonction `show_picvar` qui nous est proposée dans la documentation, qui affiche à l'écran l'image que je vous ai présenté un peu plus haut :

```
#include <tigcclib.h>

short show_picvar(SYM_STR SymName, short x, short y, short Attr);

void _main(void)
{
    ClrScr();

    show_picvar(SYMSTR ("expic"), 10, 10, A_NORMAL);

    ngetchx();
}


short show_picvar(SYM_STR SymName, short x, short y, short Attr)
{
    SYM_ENTRY *sym_entry = SymFindPtr(SymName, 0);
    if(!sym_entry)
        return FALSE;
    if(peek(HToESI(sym_entry->handle)) != PIC_TAG)
        return FALSE;
    BitmapPut(x, y, HeapDeref(sym_entry->handle) + 2, ScrRect, Attr);
    return TRUE;
}
```

Exemple Complet

Et voici une capture d'écran prise pendant l'exécution du programme :



Naturellement, libre à vous de modifier cette fonction si vous souhaitez obtenir un comportement différent !

Le prochain chapitre va nous permettre de voir ce qu'est la "VAT", et comment s'en servir, voire la manipuler.
Plus loin au cours de ce tutorial, nous aurons l'occasion de revenir sur l'affichage de graphismes, que ce soit en niveaux de gris, ou en utilisant des sprites...

 onclusion

Cette page n'a pas encore été rédigée.

Y'a pas le feu...

Reste une vingtaine de chapitres à écrire avant d'en arriver ici...

M'enfin, on verra un de ces jours pour rédiger un minimum, afin de donner l'impression d'un tout :D