École de technologie supérieure

THÈME 4
Des caractères qui parfois n'en sont pas

INF125 Introduction à la programmation
Sylvie Ratté et Hugues Saulnier

Ce thème résume le chapitre 5 du livre de référence.

1. Vol d'identité! Des caractères qui sont des nombres

COMPÉTENCE VISÉE:
Être capable d'utiliser le type "char" et comprendre le lien qui l'unit au type "int". Savoir utiliser les fonctions "getchar" et "putchar" et prendre conscience du phénomène de tamporisation ("buffering") qui a lieu lors de la saisie.

A. Représentation et stockage: parlez-vous ASCII couramment?

Puisque nos machines ne peuvent comprendre que des séquences de bits, la représentation des caractères devra donc évidemement faire aussi appel à des séquences de bits. Qu'est-ce qui fait en sorte qu'une séquence soit interprétée comme un entier, comme un réel ou comme un caractère? C'est le TYPE associé à ces séquences qui indique clairement comment nous désirons que cette séquence soit interprétée par notre programme. De là toute l'importance des types et du système qui permet de les interpréter correctement.

Le type "char" représente les caractères de toute sorte. Une variable de type "char" se déclare évidemment comme tout autre variable. Pour représenter des valeurs littérales (ex. la lettre A), on utilise l'apostrophe. Ainsi,

```
char rep;  /* déclaration d'une variable de type caractère */
rep = 'A'; /* assignation d'un caractère à la variable     */
```

L'ensemble des caractères comprend les lettres, majuscules et minuscules, les chiffres de 0 à 9, les symboles de ponctuation (ex. ?, &, != , +, etc.) et même les caractères qui ne s'affichent pas directement: beep sonore, changement de ligne, saut de page, etc. Pour savoir quel caractère est représenté par une séquence de bits données, il faut avoir à sa disposition une table de correspondance. Il est très facile de s'en créer une. Le tout consiste à inventer une correspondance entre ces caractères et une séquence de bits donnée. Ainsi, si vous avez trois bits à votre disposition, vous pourriez vous inventer le code suivant pour représenter les lettres de A à H:

```
000 A
001 B
010 C
011 D
100 E
101 F
110 G
111 H
```

Il faudrait évidemment que les autres utilisateurs acceptent votre code et l'utilise. Le code ASCII constitue justement cela. Il s'agit d'une table de correspondance entre des séquences de bits et des caractères. Ce code, édicté par une organisation américaine, est désormais présents sur tous les ordinateurs de type PC à quelques différences près. Le code de base contient 128 caractères représentés par les nombres 0 à 127. L'espace porte le numéro 32, la lettre A, le numéro 65, le retour à la ligne, le code 13, etc. Ces 128 caractères forment le coeur standard du code ASCII (American Standard Code for Information Interchange).

Il existe aussi différent code ASCII dits étendus (256 caractères) afin de tenir compte des caractères qui s'avèrent pour certains essentiels (les accents du français par exemple). Ces codes ASCII ne sont cependant pas standards. Vous en avez certainement fait l'expérience en transférant un fichier d'un PC vers un MACINTOSH ou même un fichier écrit sous DOS vers WINDOWS!

On trouve aussi le code EBCDIC (Extended Binary Coded Decimal Interchange Code) disponibles essentiellement sur les gros ordinateurs IBM. Ce code contient 256 caractères codés et ces caractères ne se trouvent pas exactement dans le même ordre que ceux du code ASCII.

Un texte écrit sur un ordinateur utilisant l'encodage EBCDIC aura toutes les chances d'être en partie illisible sur un ordinateur qui utilise un code ASCII.

L'annexe B du livre présente les tables des codes ASCII et EBCDIC. Vous pouvez aussi retrouver la première ici en format PDF.

B. Le monde où 'A' + 4 = 'E'

La chaîne de caractères bien identifiée par ses guillements comme "dommage" est un suite de "char". Un caractère littéral unique, type "char", est TOUJOURS indiqué par des apostrophes comme 'Y'. Pour lire avec "scanf" un caractère, on peut utiliser %c et il en est de même pour l'afficher avec "printf". Mais il faut bien comprendre ici que le %c dans un "scanf" considère que tous les caractères saisis par

l'utilisateur sont pertinents et seront donc lus. Ainsi, la boucle suivante s'effectue 6 fois si l'utilisateur saisit "2 4 6" suivi d'un retour de chariot puis de ^Z. Il y a bien 6 caractères lus: le '2', l'espace, le '4', l'espace, le '6' et le changement de ligne.

```
int main(){
   char car;
   while (scanf("%c", &car)!= EOF)
      printf("%c\n", car);
   return 0;
}
```

Par contre, elle sera effectuée 3 fois si le programme devient:

```
int main(){
   int n;
   while (scanf("%i", &n)!= EOF)
      printf("%i\n", n);
   return 0;
}
```

puisque, dans ce cas, ni les espaces, ni le changement de ligne ne sont considérés comme étant pertinents.

Si on le désire, on peut aussi afficher un caractère comme un entier. L'entier ainsi affiché correspond évidemment au code ASCII.

```
printf("%i", 'A'); /* affiche 65. */
```

Vous pouvez d'ailleurs vérifier les caractères lus dans la première boucle en complétant le "printf" comme suit:

```
printf("%c %i", car, car);
```

Vous aurez ainsi une idée des caractères lus par les "scanf" successifs.

Un petit mot sur "printf"

L'affichage de certains caractères avec "printf" pose parfois certains problèmes (le changement de ligne par exemple) et c'est pourquoi nous avons des séquences spéciales pour les représenter. Nous connaissons déjà le \n et aussi le \t. Voici la table complète de ces séquences bien pratiques:

Séquence	Signification	Séquence	Signification
`\0`	le caractère nul	`\a`	alerte (beep)
`\b`	retour arrière (back space)	`\f`	saut de page (form feed)
`\n`	nouvelle ligne	`\r`	retour de chariot
`\t`	tabulation horizontale	`\v`	tabulation verticale
`\'`	apostrophe	`\"`	guillemet
`\\`	barre oblique inversée	`\?`	point d'interrogation
`\`*ddd*	caractère correspondant au code octal indiqué par *ddd*	`\x`*ddd*	caractère correspondant au code hexadécimal indiqué par *ddd*

Vous pouvez d'ailleurs essayer le programme WAKEUP.C pour vous familiariser avec quelques unes de ces séquences.

Conversion

Puisque la représentation des caractères implique des nombres entre 0 et 127, la conversion du type "int" au type "char" et vice versa est tout à fait naturelle en C. La conversion automatique se charge d'effectue le travail. Il n'y a donc pas de fonctions de conversions comme en Pascal ou en Basic. Si vous désirez le code ASCII d'un caractère, il suffit de l'utiliser comme un entier. Ainsi l'expression suivante permet de transformer un caractère minuscule en majuscule:

```
(carac - 'a') + 'A'
```

Si la variable "carac" contient le caractère "p", on obtient:

```
                  ('p' - 'a') + 'A'
interprété comme : (112 - 97)  + 65
ce qui donne     : 80
soit le caractère: 'P'
```

Cette technique simple ne fonctionne cependant pas en EBCDIC car les caractères ne sont pas ordonnés de la même façon.

Ce va-et-viens entre les entiers et les caractères est parfois subtile. Il faut simplement se rappeler que le type "char" accepte des nombres de 0 à 127 ou de 0 à 255. Si le programmeur de la boucle suivante pense que celle-ci permettra d'afficher tous les caractères, il se trompe:

```
int main(void){
   char car;
```

```
            for (car=0; car< 256; car++)
               printf("%i : %c\n", car, car);
         }
```

Certains compilateurs offriront d'ailleurs le petit avertissement suivant concernant la condition du "for":

```
warning: comparison is always 1 due to limited range of data type
```

En clair ce message signifie que la condition sera toujours vraie. La variable "car", déclarée de type "char", n'accepte que des valeurs plus petites ou égales à 255 il est donc impossible de sortir de la boucle puisque lorsque l'on additionnera 1 à la variable "car" qui vaudra 255, on retombera sur 0! Pour produire le résultat désiré originalement, la variable "car" devrait être déclarée de type "int" afin qu'elle puisse prendre la valeur 256 nécessaire pour sortir de la boucle!

Entrée/sortie de caractères

On peut évidemment utiliser "scanf" et "printf" pour effectuer la lecture et l'affichage de caractères. Cependant, deux autres fonctions sont beaucoup plus efficaces pour effectuer ce travail. Il s'agit de "getchar" et "putchar". Leurs prototypes se présentent comme suit:

```
int getchar(void);
int putchar(int c);
```

Elles appartiennent toutes les deux à librairie "stdio".

La fonction "getchar" retourne un "int" plutôt qu'un "char" afin de nous permettre de tester le EOF qui vaut -1 et qui n'est pas un caractère appartenant au type "char" comme tel. De fait, il est très courant de manipuler les caractères carrément avec des variables "int". Si la valeur de cet entier se situe parmi les valeurs acceptables, le caractère est correctement obtenu.

Le programme DISPLAY.C qui produit un écho de tous les caractères lus illustre le fonctionnement de ces deux fonctions. Comme pour "scanf" et "printf", les fonctions "getchar" et "putchar" font usage de la tamporisation ("buffering"). Ainsi, ce n'est qu'après le changement de ligne, que la lecture comme telle de chaque caractère peut commencer. "getchar" saisit donc chaque caractère dans le tampon d'entrée ainsi remplie. Elle n'effectue donc pas sa tâche en direct. L'écho n'est donc pas immédiat. L'avantage évidemment c'est que l'utilisateur peut effectuer des corrections tant et aussi longtemps qu'il n'a pas saisi un changement de ligne.

En fait, la saisie directe en temps réel de chaque caractère est difficile à coder et dépend largement de la plateforme utilisée. En Turbo C, il existe deux fonctions (non standards évidemment) qui permettent d'effectuer de telles saisies. Elles se nomment "getch" et "getche" et se retrouvent dans la librairie "conio". La première saisit un caractère sans produire un écho de ce caractère à l'écran (de sorte que l'utilisateur ne voit pas ce qu'il tappe) tandis que la seconde, produit un écho du caractère saisi.

Voici un petit programme LINENO.C à essayer avec une redirection de l'entrée standard. il permet de numéroter les lignes d'un fichier texte. Vous pouvez même l'essayer sur le texte du code source lui-même.

```
rappel pour la redirection: LINENO < fichier.txt
ou même: LINENO < LINENO.C > LINENO.TXT
```

La dernière commande produira une copie du programme source dont les lignes seront numérotées. Évidemment, le fichier ainsi produit ne peut être compilé!

Vous êtes désormais en mesure d'examiner l'étude de cas (A).

2. Manipulation de tous ses oiseaux

COMPÉTENCE VISÉE:
Savoir utiliser les fonctions offertes dans la librairie "ctype". Être capable d'effectuer des lectures caractère par caractère et être en mesure de construire un nombre entier en procédant de la sorte.

A. Quelques tests sur les caractères

La librairie "ctype" contient un groupe de fonctions permettant d'effectuer des tests simples sur les caractères et deux fonctions permettant de passer des majuscules aux minuscules et vice-versa. Les fonctions du premier groupe retournent un entier différent de 0 lorsque le caractère fournit en argument correspond au critère testé, et 0 sinon.

`isalnum`	une lettre ou un chiffre (a..z, A..Z, 0..9)
`isalpha`	une lettre (a..z, A..Z)
`iscntrl`	un caractère de contrôle ou la touche d'effacement (codes 0 à 31 et code 127)
`isdigit`	un chiffre (0..9)
`isgraph`	un caractère imprimable mais n'inclut pas l'espace (codes 33..126)
`islower`	une lettre minuscule (a..z)
`isprint`	un caractère imprimable en incluant l'espace (codes 32..126)
`ispunct`	un caractère de ponctuation c'est-à-dire est "isprint" ET n'est pas "isalnum"
`isspace`	le caractère espace
`isupper`	une lettre majuscule (A..Z)
`isxdigit`	un caractère pouvant représenter un chiffre hexadécimal (0..9, A..F)

`tolower`	Si le caractère fourni est déjà une lettre minuscule, retourne ce caractère intact sinon, retourne la lettre correspondante en minuscule. Si le caractère fourni n'est pas une lettre, retourne ce caractère intact.
`toupper`	Si le caractère fourni est déjà une lettre majuscule, retourne ce caractère intact sinon, retourne la lettre correspondante en majuscule. Si le caractère fourni n'est pas une lettre, retourne ce caractère intact.

Toutes ces fonctions ne considèrent aucunement les codes supérieurs à 127. Ainsi, les caractères accentués sont totalement évacués.

Les fonctions pour traiter les accents dépendent évidemment de l'environnement dans lequel vous travaillez. Un bon exercice ici consiste à faire afficher les codes et les caractères correspondant des caractères de 128 à 255 pour ensuite, réécrire vos propres fonctions isalpha, isctrl, tolower, toupper, etc. Vous pourriez ainsi vous créez une petite librairie où l'on trouvera de nouvelles fonctions tenant compte des caractères français. Votre version personnelle de "toupper" devrait transformer le "é" en "É".

Le petit programme suivant affiche justement tous les caractères entre 128 et 255 inclusivement. Faites-le exécuter et redirigez son résultat vers une fichier texte afin de pouvoir examiner les caractères à votre aise.

Ouvrez le fichier grâce à Turbo C et notez le code ASCII correspondant au "é". Fermez le fichier puis ouvrez-le cette fois avec le bolc-notes de Windows ou encore "Wordpad" ou "Word". Notez à nouveau le code du "é".

Le petit programme BRKWD.C illustre une utilisation possible de la fonction "isalpha". Il extrait les mots d'un texte. Essayez-le avec en interactif mais également avec des redirections.

B. Des caractères aux nombres

Nous connaissons désormais les limites de la fonction "scanf". Examinons, pour nous en convaincre, la lecture de quelques entiers avec "scanf" et la lecture de quelques réels:

BOUT DE CODE (1)	`int nombre;` `scanf("%i", &nombre);`
SÉQUENCE SAISIE PAR L'UTILISATEUR	CONTENU DE LA VARIABLE "nombre"
45 65 72	45 (scanf retourne 1) et le tampon contient encore " 65 72"
17.5	17 (scanf retourne 1) et le tampon contient encore ".5"
22abc14	22 (scanf retourne 1) et le tampon contient encore " abc"
.45	? (scanf retourne 0) et le tampon contient encore ".45"
abc	? (scanf retourne 0) et le tampon contient encore "abc"

BOUT DE CODE (2)	`double x;` `scanf("%lf", &x);`
SÉQUENCE SAISIE PAR L'UTILISATEUR	CONTENU DE LA VARIABLE "x"
45 65 72	45.0 (scanf retourne 1) et le tampon contient encore " 65 72"
17.5	17.5 (scanf retourne 1) et le tampon est vide
22abc14	22.0 (scanf retourne 1) et le tampon contient encore "abc14"
.45	0.45 (scanf retourne 1) et le tampon est vide
23.56abc	23.56 (scanf retourne 1) et le tampon contient encore "abc"
abc45	? (scanf retourne 0) et le tampon contient encore "abc45"

Si l'on désire effectuer une validation extrêmement robuste des entrées, "scanf" ne peut nous aider de façon élégante. Que faire, par exemple, pour extraire tous les entiers d'une séquence telle que: 45abc17.5 90.123tuh24.

On pourrait, dans un premier temps, profiter du fait que "getchar" peut extraire un caractère du tampon à la fois et profiter de cette particularité pour effectuer des tentatives de lecture d'entiers de la manière suivante:

`#include <stdio.h>` `#include <ctype.h>`	Les libraries nécessaires.
`int main(void){` `  int nombre, encore, r, car;`	le nombre entier lu, un drapeau pour indiquer la fin, la variable "r" pour capter le résultat du "scanf" et la variable "car" pour capter le résultat du "getchar".
`  encore = car = 1;`	Initialisation du drapeau et de la variable "car" puisque nous avons besoin de les référer immédiatement.
`  while (encore) {`	tant qu'il y a encore des données dans le tampon d'entrée
`   while ((car != EOF) && (r = scanf("%i", &nombre)) == 0)`	tant que le caractère lu n'est pas EOF et que le résultat du "scanf" sur un entier n'est pas possible
`      car = getchar();`	enlever un caractère du tampon d'entrée
`   if (r != EOF && car != EOF)` `      printf("%i\n", nombre);`	si nous n'avons pas rencontré EOF ni lors du "scanf" ou lors du "getchar", on affiche le nombre entier lu.
`   else` `      encore = 0;`	si nous avons rencontré EOF lors du "scanf" ou lors du "getchar", nous avons terminé.

`  }`
`  return 0;`
`}`
	fin du "main"

Le problème évidemment c'est que le programme ne distinguera pas les nombres entiers des nombres réels. Ainsi, pour la chaîne précédente, les nombres affichés seront, dans l'ordre, 45, 17, 5, 90, 123, 24. Si l'on désire uniquement les nombres entiers, il faut procéder autrement.

De fait, lorsqu'une telle situation est rencontrée, on procède uniquement avec "getchar". Notre algorithme s'apparente alors à celui utilisé dans BRKWD.C sauf que cette fois, nous devons extraire les entiers. L'algorithme pourrait alors s'écrire comme suit:

1. lire un caractère à la fois jusqu'à ce que le caractère lu soit un chiffre
2. lire un caractère à la fois jusqu'à ce que le caractère lu ne soit pas un chiffre (tout en faisant cela construire le nombre entier correspondant)
3. si le caractère qui vient d'être lu est un point alors
   rejeter le nombre construit et lire les caractères suivants jusqu'à ce que le caractère lu ne soit plus un chiffre.
4. sinon afficher le nombre entier construit
5. recommencez le traitement à partir de (1)

Comment traduisons-nous cet algorithme en C? Nous avons besoin d'une variable entière pour contenir le dernier caractère lu et d'une variable entière dans laquelle on placera l'entier construit. Nous les nommerons respectivement "car" et "nombre".

NOTE
Dans la traduction qui va suivre, nous utilisons le fait que C interprète la valeur 0 comme le faux et TOUTE VALEUR DIFFÉRENTE DE ZÉRO, comme le vrai. De plus, nous utilisons l'opérateur ! signifiant la négation. Les opérateurs logiques sont présentés dans la table qui suit. On trouvera également des explications plus approfondies aux sections 6.2 et 6.3 du livre ainsi qu'à la section 1 du thème 5.

`&&`	Opérateur ET (AND) `condition1 && condition 2`
`||`	Opérateur OU (OR) `condition1 || condition2`
`!`	Opérateur NON (NOT) `!condition`

Voici maintenant la traduction de l'algorithme:

`#include <stdio.h>` `#include <ctype.h>`	Les libraries nécessaires.
`int main(void){` `  int nombre, encore, car;`	le nombre entier lu, un drapeau pour indiquer la fin et la variable "car" pour capter le résultat du "getchar".
`  encore = 1;`	Initialisation du drapeau
`  while (encore) {`	tant qu'il y a encore des données dans le tampon d'entrée
`   while (!isdigit(car = getchar()));`	1. lire un caractère à la fois jusqu'à ce que le caractère lu soit un chiffre. Remarquez le ";" à la fin. Le "while" n'est formé que de la condition. Il n'y a donc pas d'instructions comme telle dans le corps de la boucle.

`   nombre = 0;`
`   while (isdigit(car)){`
`     nombre = nombre*10 + (car - '0');`
`     car = getchar();`
`   }`
	2. lire un caractère à la fois jusqu'à ce que le caractère lu ne soit pas un chiffre (tout en faisant cela construire le nombre entier correspondant)

`   if (car == '.')` `      while (isdigit(car = getchar()));`	3. si le caractère qui vient d'être lu est un point alors rejeter le nombre construit et lire les caractères suivants jusqu'à ce que le caractère lu ne soit plus un chiffre.
`   else` `      printf("%i", nombre);`	4. sinon afficher le nombre entier construit

`  }`
`  return 0;`
`}`
	fin du "main"

Le seul problème (!) avec ce programme (que vous trouverez sous le nom SCAN3.C) c'est qu'il ne s'arrête jamais. En effet la boucle "while" externe est contrôlée par la variable "encore" qui prend la valeur 1 au début du programme mais qui ne change jamais de valeur. Pour traiter la fin des données avec EOF, il nous faudrait insérer des tests sur EOF à chaque fois que nous effectuons un "getchar", ce qui allourdira considérablement le programme on en conviendra.

La version correcte vous est laissée en exercice. Pour les curieux, voici une première solution intitulée SCAN4.C.

ÉTUDE DE CAS

A. VOS ACCOLADES SONT-ELLES ÉQUILIBRÉES?

Nous ferons usage ici de la redirection pour examiner le source d'un programme C. Notre programme affichera, après l'examen, un simple message. Si le nombre d'accolades ouvrantes est égal au nombre d'accolades fermantes, le programme affiche: "Tout est en equilibre." Autrement, le programme affichera un message de votre choix.

Le traitement est très simple. Il suffit de lire sur "stdin" caractère par caractère jusqu'à la fin. On a déjà un début de boucle en tête:

```
       while ((car = getchar()) != EOF)
```

Ensuite, pour compter le nombre d'accolades ouvrantes et fermantes, on peut avoir recours à deux variables entières ou encore à une seule comme dans la solution suivante:

```
int main(void){
```

```
    int nombre, car;
    nombre = 0;
    while ((car = getchar()) != EOF) {
      if (car == '{') nombre++;
      else if (car == '}') nombre--;
    }

    if (nombre == 0)
      printf("Tout est en equilibre.\n");
    else if (nombre > 0)
      printf("Trop d'accolades ouvrantes, %i en surplus.\n", nombre);
    else
      printf("Trop d'accolades fermantes, %i en surplus.\n", -1 * nombre);

   return 0;
}
```

Le fichier peut être récupéré ici sous SCAN5.C. Évidemment cette solution ne permet pas d'afficher le nombre d'accolades de chaque type rencontrées. Pour cela, il aurait fallu utiliser deux variables entières pour compter.

B. De la base 10 vers la base 16

Version 1: le principe de base consiste à effectuer des modulos et des divisions successives par 16

`int lireEntierPositif(void);` `void ecrireChiffreBase16(int chiffre);`	Le programme utilise deux fonctions pour effectuer sa tâche. La fonction "lireEntierPositif" effectue des lectures répétitives tant et aussi longtemps que l'utilisateur n'a pas saisi un nombre entier positif. La fonction "ecrireChiffreBase16" affiche le chiffre fourni en argument en base 16.

`int main(void){`
`  int nombre;`
`  nombre = lireEntierPositif();`
`  while (nombre){`
`  ecrireChiffreBase16(nombre % 16);`
`  nombre /= 16;`
`  }`
`  return 0;`
`}`
	Modulos et divisions successives entières par 16 et affichage des chiffres ainsi récupérés.

`int lireEntierPositif(void){`
`  int valeur;`
`  int fini;`
`  fini = 0;`
`  while (!fini) {`
`  printf("\nDonnez-moi un entier positif : ");`
`  if (scanf("%i", &valeur) == 1) {`
`  if (valeur >= 0)`
`  fini = 1;`
`  }`
`  fflush(stdin);`
`  }`
`  return valeur;`
`}`
	Le code de cette fonction est évident.

`void ecrireChiffreBase16(int chiffre){`
`  if (chiffre <= 9)`
`  printf("%i", chiffre);`
`  else`
`  printf("%c", chiffre + 55);`
`}`
	Lorsque le chiffre est plus petit que 10, on affiche le chiffre comme un entier sinon, on doit récupérer la lettre qui représente ce chiffre. Inutile d'effectuer des tests sur 10, 11, 12, etc. Il suffit de se rapporter au code ASCII. Si le chiffre est 10, on désire afficher A, si c'est 11, on désire afficher B, etc. En additionnant 55 au chiffre, on obtient le code ASCII de la lettre correspondante.

Cette version souffre cependant d'un problème majeure. Le nombre en base 16 résultant est écrit à l'envers! Essayez-le! Vous trouverez le code complet sous le nom BASE10.C.

Version 2: Écrivons tout cela à l'endroit.

Pour écrire le nombre résultant correctement, il faut savoir d'avance combien de chiffres composent le résultat final. En fouillant un peu dans notre mémoire de matheux, on peut récupérer la formule suivante:

Le nombre de chiffres nécessaires pour représenter un nombre N en base B est: $\lfloor \log_B N \rfloor + 1$

La fonction "floor" de la librairie "math" nous permettra de calculer le "plancher" du résultat mais ne fournit pas de "log" en base 16. Par contre, elle fournit le "log" en base 10 (la fonction "log10") et nous savons, parce que nous avons tous fait un petit cours de mathématiques, que:

$$\log_B N = \frac{\log_{10} N}{\log_{10} B}$$

Si nous savons maitenant combien de chiffres composeront le résultat, disons "k", nous savons que le chiffre situé à l'extrême gauche est de l'ordre de 16 exposant (k-1) puisque le chiffre à l'extrême droite est de l'ordre de 16 exposant 0.

Notre boucle démarrera en divisant par 16 exposant (k - 1), puis par 16 exposant (k - 2) et ainsi de suite jusqu'à 16 exposant 0. Les modifications n'affectent que le "main", celui devient:

```
int main(void){
  int nombre, k, diviseur;
  nombre = lireEntierPositif();
  k = floor(log10(nombre)/log10(16)) + 1;
  diviseur = pow(16, k - 1);
  while (diviseur){
    ecrireChiffreBase16(nombre / diviseur);
    nombre %= diviseur;
    diviseur /= 16;
  }
  return 0;
}
```

Le programme complet se trouve ici sous le nom BASE10P.C.

C. De la base 10 vers celle que vous voulez

Il ne manque pas grand chose pour faire une version un peu plus générale. En fait, avec toutes les lettres de l'alphabet (A à Z), on peut convertir un nombre en base 10 vers un nombre en base 36! On lira donc la base résultante désirée puis, dans notre code, il nous suffira de remplacer les 16 par notre variable contenant la base. Le programme complet se trouve sous le nom BASE10G.C.

D. De la base 16 vers celle que vous voulez

Le problème ici consiste à lire le nombre fourni par l'utilisateur. Ce nombre peut être composé de lettres. Bien sûr, nous pourrions utiliser la séquence %x pour effectuer la saisie (voir la table du thème 3 concernant les scanf et les printf). Cela fonctionne d'ailleurs très bien. Le "scanf" de la fonction "lireEntierPositif" deviendrait:

```
scanf("%x", &valeur)
```

Mais cette méthode se généralise assez mal puisqu'il n'existe pas de séquences spéciales pour d'autres bases (sauf la base 8). Nous n'avons donc pas le choix, pour lire un nombre exprimé en base 16 (ou tout autre base d'ailleurs), on lira le nombre caractère par caractère avec "getchar" tout en construisant le nombre final. La fin du nombre sera indiqué par le changement de ligne (soit "\n"). La lecture est simple et s'apparente à la boucle que nous avons déjà vue plus haut (programme SCAN3.C). Cependant, il faut distinguer la lecture des chiffres entre 0 et 9 de celles des caractères entre A et F. Dans le premier cas, on devra soustraire 48 de l'entier ainsi obtenu puisque le caractère '0' EST l'entier 48. Dans le second cas, on soustrait la valeur 55 (ainsi 'A' - 55 = 10). La fonction de lecture a été renommée "lireEntierEnBase16" et sa définition se présente comme suit:

```
int lireEntierEnBase16(void){
  int valeur, car;
  valeur = 0;
  printf("\nDonnez-moi un entier positif en base 16 : ");
  while ((car = getchar()) != '\n')
     if (isdigit(car))
        valeur = valeur * 16 + (car - '0');
     else
        valeur = valeur * 16 + (car - 55);
  return valeur;
}
```

Le code complet du programme se trouve sous le nom BASE16.C.

E. De la base que vous désirez à celle que vous voulez

Si l'on examine la dernière fonction, on se rend rapidement compte que l'on peut sans problème la généraliser. En effet, en modifiant les 16 pour des 2, on peut lire un entier en base 2. En remplacant les 16 par des 32, on peut lire des entiers exprimés en base 32 et ainsi de suite.

Cette version générale pourrait donc admettre un argument qui sera la base dans laquelle sera écrit le nombre. On obtient:

```
int lireEntierEnBase(int base){
  int valeur, car;
  valeur = 0;
  printf("\nDonnez-moi un entier positif en base %i : ", base);
  while ((car = getchar()) != '\n')
     if (isdigit(car))
        valeur = valeur * base + (car - '0');
     else
        valeur = valeur * base + (car - 55);
  return valeur;
}
```

Le code complet se trouve dans le fichier BASEG.C.

Il est à noter qu'aucune de ces versions n'effectuent de validation sur les nombres saisis par l'utilisateur. Rien n'empêche notre utilisateur d'entrer des 3 ou des 5 lorsqu'il saisit un nombre en base 2. Cette validation peut être rajoutée dans chaque fonction de lecture de manière relativement simple.

EXERCICE

Écrire un programme capable de lire une séquence de notes littérales (A+, A, A-, B+, B, B-, etc.) appartenant à un étudiant et de calculer sa moyenne. La séquence se termine par le caractère '*'. Chaque note peut être éparée de la suivante par un ou plusieurs changement de ligne, un ou plusieurs espaces ou n'importe quel caractère non alphabétique (sauf évidemment +, - et *). Ce problème n'est pas aussi simple qu'il en a l'air. La table de correspondance suivante pourra vous aider:

A+	4.3	B+	3.3	C+	2.3	D+	1.3
A	4.0	B	3.0	C	2.0	D	1.0
A-	3.7	B-	2.7	C-	1.7	E	0.0

Voici une solution possible (NOTES.C)