Chapitre 4

Les fonctions

Comme dans la plupart des langages, on peut en C découper un programme en plusieurs fonctions. Une seule de ces fonctions existe obligatoirement ; c'est la fonction principale appelée `main`. Cette fonction principale peut, éventuellement, appeler une ou plusieurs fonctions secondaires. De même, chaque fonction secondaire peut appeler d'autres fonctions secondaires ou s'appeler elle-même (dans ce dernier cas, on dit que la fonction est *récursive*).

4.1 Définition d'une fonction

La définition d'une fonction est la donnée du texte de son algorithme, qu'on appelle corps de la fonction. Elle est de la forme

```
 type  nom-fonction ( type-1 arg-1,..., type-n arg-n)
{
  [ déclarations de variables locales ]
   liste d'instructions
}
```

La première ligne de cette définition est l'*en-tête* de la fonction. Dans cet en-tête, ***type*** désigne le type de la fonction, c'est-à-dire le type de la valeur qu'elle retourne. Contrairement à d'autres langages, il n'y a pas en C de notion de procédure ou de sous-programme. Une fonction qui ne renvoie pas de valeur est une fonction dont le type est spécifié par le mot-clef `void`. Les arguments de la fonction sont appelés *paramètres formels*, par opposition aux *paramètres effectifs* qui sont les paramètres avec lesquels la fonction est effectivement appelée. Les paramètres formels peuvent être de n'importe quel type. Leurs identificateurs n'ont d'importance qu'à l'intérieur de la fonction. Enfin, si la fonction ne possède pas de paramètres, on remplace la liste de paramètres formels par le mot-clef `void`.

Le corps de la fonction débute éventuellement par des déclarations de variables, qui sont locales à cette fonction. Il se termine par l'*instruction de retour à la fonction appelante*, `return`, dont la syntaxe est

```
return(expression);
```

La valeur de ***expression*** est la valeur que retourne la fonction. Son type doit être le même que celui qui a été spécifié dans l'en-tête de la fonction. Si la fonction ne retourne pas de valeur (fonction de type `void`), sa définition s'achève par

```
return;
```

Plusieurs instructions `return` peuvent apparaître dans une fonction. Le retour au programme appelant sera alors provoqué par le premier `return` rencontré lors de l'exécution. Voici quelques exemples de définitions de fonctions :

```
int produit (int a, int b)
{
  return(a*b);
}

int puissance (int a, int n)
{
  if (n == 0)
    return(1);
  return(a * puissance(a, n-1));
}

void imprime_tab (int *tab, int nb_elements)
{
  int i;
  for (i = 0; i < nb_elements; i++)
    printf("%d \t",tab[i]);
  printf("\n");
  return;
}
```

4.2 Appel d'une fonction

L'appel d'une fonction se fait par l'expression

`nom-fonction(para-1,para-2,...,para-n)`

L'ordre et le type des paramètres effectifs de la fonction doivent concorder avec ceux donnés dans l'en-tête de la fonction. Les paramètres effectifs peuvent être des expressions. La virgule qui sépare deux paramètres effectifs est un simple signe de ponctuation ; il ne s'agit pas de l'opérateur *virgule*. Cela implique en particulier que l'ordre d'évaluation des paramètres effectifs n'est pas assuré et dépend du compilateur. Il est donc déconseillé, pour une fonction à plusieurs paramètres, de faire figurer des opérateurs d'incrémentation ou de décrémentation (`++` ou `--`) dans les expressions définissant les paramètres effectifs (*cf.* Chapitre 1, page 23).

4.3 Déclaration d'une fonction

Le C n'autorise pas les fonctions imbriquées. La définition d'une fonction secondaire doit donc être placée soit avant, soit après la fonction principale `main`. Toutefois, il est indispensable que le compilateur "connaisse" la fonction au moment où celle-ci est appelée. Si une fonction est définie après son premier appel (en particulier si sa définition est placée après la fonction `main`), elle doit impérativement être déclarée au préalable. Une fonction secondaire est déclarée par son *prototype*, qui donne le type de la fonction et celui de ses paramètres, sous la forme :

`type nom-fonction(type-1,...,type-n);`

Les fonctions secondaires peuvent être déclarées indifféremment avant ou au début de la fonction `main`. Par exemple, on écrira

```
int puissance (int, int );

int puissance (int a, int n)
{
  if (n == 0)
    return(1);
  return(a * puissance(a, n-1));
}

main()
{
  int a = 2, b = 5;
  printf("%d\n", puissance(a,b));
}
```

Même si la déclaration est parfois facultative (par exemple quand les fonctions sont définies avant la fonction `main` et dans le bon ordre), elle seule permet au compilateur de vérifier que le nombre et le type des paramètres utilisés dans la définition concordent bien avec le protype. De plus, la présence d'une déclaration permet au compilateur de mettre en place d'éventuelles conversions des paramètres effectifs, lorsque la fonction est appelée avec des paramètres dont les types ne correspondent pas aux types indiqués dans le prototype. Ainsi les fichiers d'extension `.h` de la librairie standard (fichiers headers) contiennent notamment les prototypes des fonctions de la librairie standard. Par exemple, on trouve dans le fichier `math.h` le prototype de la fonction `pow` (élévation à la puissance) :

```
extern  double pow(double , double );
```

La directive au préprocesseur

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

permet au préprocesseur d'inclure la déclaration de la fonction `pow` dans le fichier source. Ainsi, si cette fonction est appelée avec des paramètres de type `int`, ces paramètres seront convertis en `double` lors de la compilation.

Par contre, en l'absence de directive au préprocesseur, le compilateur ne peut effectuer la conversion de type. Dans ce cas, l'appel à la fonction `pow` avec des paramètres de type `int` peut produire un résultat faux !

4.4 Durée de vie des variables

Les variables manipulées dans un programme C ne sont pas toutes traitées de la même manière. En particulier, elles n'ont pas toutes la même *durée de vie*. On distingue deux catégories de variables.

Les variables permanentes (ou statiques) Une variable permanente occupe un emplacement en mémoire qui reste le même durant toute l'exécution du programme. Cet emplacement est alloué une fois pour toutes lors de la compilation. La partie de la mémoire contenant les variables permanentes est appelée *segment de données*. Par défaut, les variables permanentes sont initialisées à zéro par le compilateur. Elles sont caractérisées par le mot-clef `static`.

Les variables temporaires Les variables temporaires se voient allouer un emplacement en mémoire de façon dynamique lors de l'exécution du programme. Elles ne sont pas initialisées par défaut. Leur emplacement en mémoire est libéré par exemple à la fin de l'exécution d'une fonction secondaire.

Par défaut, les variables temporaires sont situées dans la partie de la mémoire appelée *segment de pile*. Dans ce cas, la variable est dite *automatique*. Le spécificateur de type correspondant, `auto`, est rarement utilisé puisqu'il ne s'applique qu'aux variables temporaires qui sont automatiques par défaut.

Une variable temporaire peut également être placée dans un registre de la machine. Un registre est une zone mémoire sur laquelle sont effectuées les opérations machine. Il est donc beaucoup plus rapide d'accéder à un registre qu'à toute autre partie de la mémoire. On peut demander au compilateur de ranger une variable très utilisée dans un registre, à l'aide de l'attribut de type `register`. Le nombre de registres étant limité, cette requête ne sera satisfaite que s'il reste des registres disponibles. Cette technique permettant d'accélérer les programmes a aujourd'hui perdu tout son intérêt. Grâce aux performances des optimiseurs de code intégrés au compilateur (*cf.* options `-O` de `gcc`, page 10), il est maintenant plus efficace de compiler un programme avec une option d'optimisation que de placer certaines variables dans des registres.

La durée de vie des variables est liée à leur *portée*, c'est-à-dire à la portion du programme dans laquelle elles sont définies.

4.4.1 Variables globales

On appelle *variable globale* une variable déclarée en dehors de toute fonction. Une variable globale est connue du compilateur dans toute la portion de code qui suit sa déclaration. Les variables globales sont systématiquement permanentes. Dans le programme suivant, `n` est une variable globale :

```
int n;
void fonction();

void fonction()
{
  n++;
  printf("appel numero %d\n",n);
  return;
}

main()
{
  int i;
```

```
  for (i = 0; i < 5; i++)
    fonction();
}
```

La variable `n` est initialisée à zéro par le compilateur et il s'agit d'une variable permanente. En effet, le programme affiche

```
appel numero 1
appel numero 2
appel numero 3
appel numero 4
appel numero 5
```

4.4.2 Variables locales

On appelle *variable locale* une variable déclarée à l'intérieur d'une fonction (ou d'un bloc d'instructions) du programme. Par défaut, les variables locales sont temporaires. Quand une fonction est appelée, elle place ses variables locales dans la pile. A la sortie de la fonction, les variables locales sont dépilées et donc perdues.

Les variables locales n'ont en particulier aucun lien avec des variables globales de même nom. Par exemple, le programme suivant

```
int n = 10;
void fonction();

void fonction()
{
  int n = 0;
  n++;
  printf("appel numero %d\n",n);
  return;
}

main()
{
  int i;
  for (i = 0; i < 5; i++)
    fonction();
}
```

affiche

```
appel numero 1
appel numero 1
appel numero 1
appel numero 1
appel numero 1
```

Les variables locales à une fonction ont une durée de vie limitée à une seule exécution de cette fonction. Leurs valeurs ne sont pas conservées d'un appel au suivant.

Il est toutefois possible de créer une variable locale de classe statique en faisant précéder sa déclaration du mot-clef `static` :

static *type nom-de-variable*;

Une telle variable reste locale à la fonction dans laquelle elle est déclarée, mais sa valeur est conservée d'un appel au suivant. Elle est également initialisée à zéro à la compilation. Par exemple, dans le programme suivant, `n` est une variable locale à la fonction secondaire `fonction`, mais de classe statique.

```
int n = 10;
void fonction();

void fonction()
{
  static int n;
  n++;
  printf("appel numero %d\n",n);
  return;
}

main()
{
  int i;
  for (i = 0; i < 5; i++)
    fonction();
}
```

Ce programme affiche

```
appel numero 1
appel numero 2
appel numero 3
appel numero 4
appel numero 5
```

On voit que la variable locale `n` est de classe statique (elle est initialisée à zéro, et sa valeur est conservée d'un appel à l'autre de la fonction). Par contre, il s'agit bien d'une variable locale, qui n'a aucun lien avec la variable globale du même nom.

4.5 Transmission des paramètres d'une fonction

Les paramètres d'une fonction sont traités de la même manière que les variables locales de classe automatique : lors de l'appel de la fonction, les paramètres effectifs sont copiés dans le segment de pile. La fonction travaille alors uniquement sur cette copie. Cette copie disparaît lors du retour au programme appelant. Cela implique en particulier que, si la fonction modifie la valeur d'un de ses paramètres, seule la copie sera modifiée ; la variable du programme appelant, elle, ne sera pas modifiée. On dit que les paramètres d'une fonction sont *transmis par valeurs*. Par exemple, le programme suivant

```
void echange (int, int );

void echange (int a, int b)
{
  int t;
  printf("debut fonction :\n a = %d \t b = %d\n",a,b);
  t = a;
  a = b;
  b = t;
  printf("fin fonction :\n a = %d \t b = %d\n",a,b);
  return;
}

main()
{
  int a = 2, b = 5;
  printf("debut programme principal :\n a = %d \t b = %d\n",a,b);
  echange(a,b);
  printf("fin programme principal :\n a = %d \t b = %d\n",a,b);
}
```

imprime

```
debut programme principal :
 a = 2   b = 5
debut fonction :
 a = 2   b = 5
fin fonction :
 a = 5   b = 2
fin programme principal :
 a = 2   b = 5
```

Pour qu'une fonction modifie la valeur d'un de ses arguments, il faut qu'elle ait pour paramètre l'adresse de cet objet et non sa valeur. Par exemple, pour échanger les valeurs de deux variables, il faut écrire :

```
void echange (int *, int *);

void echange (int *adr_a, int *adr_b)
{
  int t;
  t = *adr_a;
  *adr_a = *adr_b;
  *adr_b = t;
  return;
```

```
}

main()
{
  int a = 2, b = 5;
  printf("debut programme principal :\n a = %d \t b = %d\n",a,b);
  echange(&a,&b);
  printf("fin programme principal :\n a = %d \t b = %d\n",a,b);
}
```

Rappelons qu'un tableau est un pointeur (sur le premier élément du tableau). Lorsqu'un tableau est transmis comme paramètre à une fonction secondaire, ses éléments sont donc modifiés par la fonction. Par exemple, le programme

```
#include <stdlib.h>

void init (int *, int );

void init (int *tab, int n)
{
  int i;
  for (i = 0; i < n; i++)
    tab[i] = i;
  return;
}

main()
{
  int i, n = 5;
  int *tab;
  tab = (int*)malloc(n * sizeof(int));
  init(tab,n);
}
```

initialise les éléments du tableau `tab`.

4.6 Les qualificateurs de type `const` et `volatile`

Les qualificateurs de type `const` et `volatile` permettent de réduire les possibilités de modifier une variable.

`const` Une variable dont le type est qualifié par `const` ne peut pas être modifiée. Ce qualificateur est utilisé pour se protéger d'une erreur de programmation. On l'emploie principalement pour qualifier le type des paramètres d'une fonction afin d'éviter de les modifier involontairement.

`volatile` Une variable dont le type est qualifié par `volatile` ne peut pas être impliquée dans les optimisations effectuées par le compilateur. On utilise ce qualificateur pour les variables susceptibles d'être modifiées par une action extérieure au programme.

Les qualificateurs de type se placent juste avant le type de la variable, par exemple

```
const char c;
```

désigne un caractère non modifiable. Ils doivent toutefois être utilisés avec précaution avec les pointeurs. En effet,

```
const char *p;
```

définit un pointeur sur un caractère constant, tandis que

```
char * const p;
```

définit un pointeur constant sur un caractère.

4.7 La fonction `main`

La fonction principale `main` est une fonction comme les autres. Nous avons jusqu'à présent considéré qu'elle était de type `void`, ce qui est toléré par le compilateur. Toutefois l'écriture

```
main()
```

provoque un message d'avertissement lorsqu'on utilise l'option `-Wall` de `gcc` :

```
% gcc -Wall prog.c
prog.c:5: warning: return-type defaults to 'int'
prog.c: In function 'main':
prog.c:11: warning: control reaches end of non-void function
```

En fait, la fonction `main` est de type `int`. Elle doit retourner un entier dont la valeur est transmise à l'environnement d'exécution. Cet entier indique si le programme s'est ou non déroulé sans erreur. La valeur de retour 0 correspond à une terminaison correcte, toute valeur de retour non nulle correspond à une terminaison sur une erreur. On peut utiliser comme valeur de retour les deux constantes symboliques `EXIT_SUCCESS` (égale à 0) et `EXIT_FAILURE` (égale à 1) définies dans `stdlib.h`. L'instruction `return(`*`statut`*`);` dans la fonction `main`, où *`statut`* est un entier spécifiant le type de terminaison du programme, peut être remplacée par un appel à la fonction `exit` de la librairie standard (`stdlib.h`). La fonction `exit`, de prototype

```
void exit(int statut);
```

provoque une terminaison normale du programme en notifiant un succès ou un échec selon la valeur de l'entier `statut`.

Lorsqu'elle est utilisée sans arguments, la fonction `main` a donc pour prototype

```
int main(void);
```

On s'attachera désormais dans les programmes à respecter ce prototype et à spécifier les valeurs de retour de main.

La fonction main peut également posséder des paramètres formels. En effet, un programme C peut recevoir une liste d'arguments au lancement de son exécution. La ligne de commande qui sert à lancer le programme est, dans ce cas, composée du nom du fichier exécutable suivi par des paramètres. La fonction main reçoit tous ces éléments de la part de l'interpréteur de commandes. En fait, la fonction main possède deux paramètres formels, appelés par convention argc (argument count) et argv (argument vector). argc est une variable de type int dont la valeur est égale au nombre de mots composant la ligne de commande (y compris le nom de l'exécutable). Elle est donc égale au nombre de paramètres effectifs de la fonction + 1. argv est un tableau de chaînes de caractères correspondant chacune à un mot de la ligne de commande. Le premier élément argv[0] contient donc le nom de la commande (du fichier exécutable), le second argv[1] contient le premier paramètre....

Le second prototype valide de la fonction main est donc

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

Ainsi, le programme suivant calcule le produit de deux entiers, entrés en arguments de l'exécutable :

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

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

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

On lance donc l'exécutable avec deux paramètres entiers, par exemple,

```
a.out 12 8
```

Ici, argv sera un tableau de 3 chaînes de caractères argv[0], argv[1] et argv[2] qui, dans notre exemple, valent respectivement "a.out", "12" et "8". Enfin, la fonction de la librairie standard atoi(), déclarée dans stdlib.h, prend en argument une chaîne de caractères et retourne l'entier dont elle est l'écriture décimale.

4.8 Pointeur sur une fonction

Il est parfois utile de passer une fonction comme paramètre d'une autre fonction. Cette procédure permet en particulier d'utiliser une même fonction pour différents usages. Pour cela, on utilise un mécanisme de pointeur. Un pointeur sur une fonction correspond à l'adresse du début du code de la fonction. Un pointeur sur une fonction ayant pour prototype
type `fonction(`*type_1*`,...,`*type_n*`);`
est de type
type `(*)(`*type_1*`,...,`*type_n*`);`

Ainsi, une fonction `operateur_binaire` prenant pour paramètres deux entiers et une fonction de type `int`, qui prend elle-même deux entiers en paramètres, sera définie par :

```
int operateur_binaire(int a, int b, int (*f)(int, int))
```

Sa déclaration est donnée par

```
int operateur_binaire(int, int, int(*)(int, int));
```

Pour appeler la fonction `operateur_binaire`, on utilisera comme troisième paramètre effectif l'identificateur de la fonction utilisée, par exemple, si `somme` est une fonction de prototype

```
int somme(int, int);
```

on appelle la fonction `operateur_binaire` pour la fonction `somme` par l'expression

```
operateur_binaire(a,b,somme)
```

Notons qu'on n'utilise pas la notation `&somme` comme paramètre effectif de `operateur_binaire`.

Pour appeler la fonction passée en paramètre dans le corps de la fonction `operateur_binaire`, on écrit `(*f)(a, b)`. Par exemple

```
int operateur_binaire(int a, int b, int (*f)(int, int))
{
  return((*f)(a,b));
}
```

Ainsi, le programme suivant prend comme arguments deux entiers séparés par la chaîne de caractères `plus` ou `fois`, et retourne la somme ou le produit des deux entiers.

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

void usage(char *);
int somme(int, int);
int produit(int, int);
int operateur_binaire(int, int, int(*)(int, int));

void usage(char *cmd)
```

```
{
  printf("\nUsage: %s int [plus|fois] int\n",cmd);
  return;
}

int somme(int a, int b)
{
  return(a + b);
}

int produit(int a, int b)
{
  return(a * b);
}

int operateur_binaire(int a, int b, int (*f)(int, int))
{
  return((*f)(a,b));
}

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


  if (argc != 4)
    {
      printf("\nErreur : nombre invalide d'arguments");
      usage(argv[0]);
      return(EXIT_FAILURE);
    }
  a = atoi(argv[1]);
  b = atoi(argv[3]);
  if (!strcmp(argv[2], "plus"))
    {
      printf("%d\n",operateur_binaire(a,b,somme));
      return(EXIT_SUCCESS);
    }
  if (!strcmp(argv[2], "fois"))
    {
      printf("%d\n",operateur_binaire(a,b,produit));
      return(EXIT_SUCCESS);
    }
  else
    {
      printf("\nErreur : argument(s) invalide(s)");
      usage(argv[0]);
```

```
      return(EXIT_FAILURE);
    }
}
```

Les pointeurs sur les fonctions sont notamment utilisés dans la fonction de tri des éléments d'un tableau `qsort` et dans la recherche d'un élément dans un tableau `bsearch`. Ces deux fonctions sont définies dans la libriarie standard (`stdlib.h`).

Le prototype de la fonction de tri (algorithme quicksort) est

```
void    qsort(void *tableau, size_t nb_elements, size_t taille_elements,
int(*comp)(const void *, const void *));
```

Elle permet de trier les `nb_elements` premiers éléments du tableau `tableau`. Le paramètre `taille_elements` donne la taille des éléments du tableau. Le type `size_t` utilisé ici est un type prédéfini dans `stddef.h`. Il correspond au type du résultat de l'évaluation de `sizeof`. Il s'agit du plus grand type entier non signé. La fonction `qsort` est paramétrée par la fonction de comparaison utilisée de prototype :

```
int comp(void *a, void *b);
```

Les deux paramètres `a` et `b` de la fonction `comp` sont des pointeurs génériques de type `void *`. Ils correspondent à des adresses d'objets dont le type n'est pas déterminé. Cette fonction de comparaison retourne un entier qui vaut 0 si les deux objets pointés par `a` et `b` sont égaux et qui prend une valeur strictement négative (resp. positive) si l'objet pointé par `a` est strictement inférieur (resp. supérieur) à celui pointé par `b`.

Par exemple, la fonction suivante comparant deux chaînes de caractères peut être utilisée comme paramètre de `qsort` :

```
int comp_str(char **, char **);

int comp_str(char **s1, char **s2)
{
  return(strcmp(*s1,*s2));
}
```

Le programme suivant donne un exemple de l'utilisation de la fonction de tri `qsort` pour trier les éléments d'un tableau d'entiers, et d'un tableau de chaînes de caractères.

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

#define NB_ELEMENTS 10

void imprime_tab1(int*, int);
void imprime_tab2(char**, int);
int comp_int(int *, int *);
int comp_str(char **, char **);

void imprime_tab1(int *tab, int nb)
```

```
{
  int i;
  printf("\n");
  for (i = 0; i < nb; i++)
    printf("%d \t",tab[i]);
  printf("\n");
  return;
}

void imprime_tab2(char **tab, int nb)
{
  int i;
  printf("\n");
  for (i = 0; i < nb; i++)
    printf("%s \t",tab[i]);
  printf("\n");
  return;
}

int comp_int(int *a, int *b)
{
  return(*a - *b);
}
int comp_str(char **s1, char **s2)
{
  return(strcmp(*s1,*s2));
}

int main()
{
  int *tab1;
  char *tab2[NB_ELEMENTS] = {"toto", "Auto", "auto", "titi", "a", "b",\
"z", "i , "o","d"};
  int i;

  tab1 = (int*)malloc(NB_ELEMENTS * sizeof(int));
  for (i = 0 ; i < NB_ELEMENTS; i++)
    tab1[i] = random() % 1000;
  imprime_tab1(tab1, NB_ELEMENTS);
  qsort(tab1, NB_ELEMENTS, sizeof(int), comp_int);
  imprime_tab1(tab1, NB_ELEMENTS);
  /************************/
  imprime_tab2(tab2, NB_ELEMENTS);
  qsort(tab2, NB_ELEMENTS, sizeof(tab2[0]), comp_str);
  imprime_tab2(tab2, NB_ELEMENTS);
  return(EXIT_SUCCESS);
}
```

La librairie standard dispose également d'une fonction de recherche d'un élément dans un tableau *trié*, ayant le prototype suivant :

```
void    *bsearch((const void *clef, const void *tab, size_t nb_elements,
size_t taille_elements, int(*comp)(const void *, const void *)));
```

Cette fonction recherche dans le tableau trié `tab` un élément qui soit égal à l'élément d'adresse `clef`. Les autres paramètres sont identiques à ceux de la fonction `qsort`. S'il existe dans le tableau `tab` un élément égal à celui pointé par `clef`, la fonction `bsearch` retourne son adresse (de type `void *`). Sinon, elle retourne le pointeur `NULL`.

Ainsi, le programme suivant prend en argument une chaîne de caractères et détermine si elle figure dans un tableau de chaînes de caractères prédéfini, sans différencier minuscules et majuscules. Rappelons que `bsearch` ne s'applique qu'aux tableaux triés ; il faut donc appliquer au préalable la fonction de tri `qsort`.

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

#define NB_ELEMENTS 4

int comp_str_maj(char **, char **);

int comp_str_maj(char **s1, char **s2)
{
  int i;
  char *chaine1, *chaine2;

  chaine1 = (char*)malloc(strlen(*s1) * sizeof(char));
  chaine2 = (char*)malloc(strlen(*s2) * sizeof(char));
  for (i = 0; i < strlen(*s1); i++)
    chaine1[i] = tolower((*s1)[i]);
  for (i = 0; i < strlen(*s2); i++)
    chaine2[i] = tolower((*s2)[i]);
  return(strcmp(chaine1,chaine2));
}

int main(int argc, char *argv[])
{
  char *tab[NB_ELEMENTS] = {"TOTO", "Auto", "auto", "titi"};
  char **res;

  qsort(tab, NB_ELEMENTS, sizeof(tab[0]), comp_str_maj);
  if ((res = bsearch(&argv[1],tab,NB_ELEMENTS,sizeof(tab[0]),comp_str_maj)) ==\
NULL)
    printf("\nLe tableau ne contient pas l'element %s\n",argv[1]);
  else
```

```
    printf("\nLe tableau contient l'element %s sous la forme %s\n",argv[1], \
*res);
  return(EXIT_SUCCESS);
}
```

4.9 Fonctions avec un nombre variable de paramètres

Il est possible en C de définir des fonctions qui ont un nombre variable de paramètres. En pratique, il existe souvent des méthodes plus simples pour gérer ce type de problème : toutefois, cette fonctionnalité est indispensable dans certains cas, notamment pour les fonctions `printf` et `scanf`.

Une fonction possédant un nombre variable de paramètre doit posséder au moins un paramètre formel fixe. La notation ... (obligatoirement à la fin de la liste des paramètres d'une fonction) spécifie que la fonction possède un nombre quelconque de paramètres (éventuellement de types différents) en plus des paramètres formels fixes. Ainsi, une fonction ayant pour prototype

```
int f(int a, char c, ...);
```

prend comme paramètre un entier, un caractère et un nombre quelconque d'autres paramètres. De même le prototype de la fonction `printf` est

```
int printf(char *format, ...);
```

puisque `printf` a pour argument une chaîne de caractères spécifiant le format des données à imprimer, et un nombre quelconque d'autres arguments qui peuvent être de types différents.

Un appel à une fonction ayant un nombre variable de paramètres s'effectue comme un appel à n'importe quelle autre fonction.

Pour accéder à la liste des paramètres de l'appel, on utilise les macros définies dans le fichier en-tête `stdarg.h` de la librairie standard. Il faut tout d'abord déclarer dans le corps de la fonction une variable pointant sur la liste des paramètres de l'appel; cette variable a pour type `va_list`. Par exemple,

```
va_list liste_parametres;
```

Cette variable est tout d'abord initialisée à l'aide de la macro `va_start`, dont la syntaxe est

```
va_start(liste_parametres, dernier_parametre);
```

où `dernier_parametre` désigne l'identificateur du dernier paramètre formel fixe de la fonction. Après traitement des paramètres, on libère la liste à l'aide de la `va_end`:

```
va_end(liste_parametres);
```

On accède aux différents paramètres de liste par la macro `va_arg` qui retourne le paramètre suivant de la liste:

```
va_arg(liste_parametres, type)
```

où `type` est le type supposé du paramètre auquel on accède.

Notons que l'utilisateur doit lui-même gérer le nombre de paramètres de la liste. Pour cela, on utilise généralement un paramètre formel qui correspond au nombre de paramètres de la liste, ou une valeur particulière qui indique la fin de la liste.

Cette méthode est utilisée dans le programme suivant, où la fonction `add` effectue la somme de ses paramètres en nombre quelconque.

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

int add(int,...);

int add(int nb,...)
{
  int res = 0;
  int i;
  va_list liste_parametres;

  va_start(liste_parametres, nb);
  for (i = 0; i < nb; i++)
    res += va_arg(liste_parametres, int);
  va_end(liste_parametres);
  return(res);
}

int main(void)
{
  printf("\n %d", add(4,10,2,8,5));
  printf("\n %d\n", add(6,10,15,5,2,8,10));
  return(EXIT_SUCCESS);
}
```