ENSEIRB 2006-2007 - Première année
PG101 - Programmation en C

TD 3 : Tableaux et chaînes de caractères (commentaires)

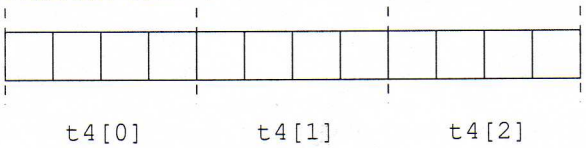
1 Tableaux

Exercice 1:

Explicitons tout d'abord la différence entre un tableau et les variables que nous avons vues jusqu'ici. Une variable `x` de type `int` est une seule zone mémoire pouvant contenir une valeur de type (taille) `int`. Un tableau `t` de 10 éléments de type `int` est donc une zone mémoire pouvant contenir 10 valeurs de type (taille) `int`. Nous pouvons aussi le décrire comme 10 zones mémoire consécutives pouvant chacune contenir une valeur de type (taille) `int`.

La définition d'un tableau diffère donc de celle d'une variable par le fait qu'elle précise le nombre de valeurs que stocke ce tableau. Ceci donne donc :

- `int t1[3]`
- `char t2[10]`
- l'initialisation se fait en même temps que la définition du tableau en donnant la liste des valeurs qu'il doit stocker : `unsigned int t3[4] = {0,1,2,3}`.
  Notons que nous pouvons omettre la taille de `t3` lors de sa définition : elle sera automatiquement déduite de son initialisation. Ainsi, la définition `unsigned int t3[] = {0,1,2,3}` est équivalente à la précédente.
- les éléments stockés par un tableau peuvent eux-mêmes être des tableaux, permettant ainsi de créer des tableaux multidimensionnels : `float t4[3][4]`. Notez cependant qu'il s'agit d'une structure linéaire comme le montre le schéma suivant :



- puisqu'un tableau multidimensionnel est un tableau de tableaux, l'initialisation reflète cette hiérarchie. Elle est donc elle-aussi constituée d'un tableau de tableaux : `int t5[2][2] = { {1,0} , {0,1} }`

Nous obtenons alors le programme suivant :

```
#include <stdio.h>

int main() {
  int t1[3];
  char t2[10];
  unsigned int t3[] = {0,1,2,3};
  float t4[3][4];
```

```
  int t5[2][2] = {{1,0},
                  {0,1}};

  printf("taille de t1: %u\n", sizeof(t1));
  printf("taille de t2: %u\n", sizeof(t2));
  printf("taille de t3: %u\n", sizeof(t3));
  printf("taille de t4: %u\n", sizeof(t4));
  printf("taille de t5: %u\n", sizeof(t5));

  printf("taille de t4[0]: %u\n", sizeof(t4[0]));
  printf("taille de t5[0]: %u\n", sizeof(t5[0]));

  return 0;
}
```

Nous voyons à l'exécution de ce programme que les tailles des différentes variables montrent qu'il s'agit de tableaux. puisqu'elles

Pour afficher les tailles de `t4[0]` et `t5[0]`, il suffit d'ajouter les deux lignes suivantes avant l'instruction `return 0 ;` dans le programme précédent :

```
printf("taille de t4[0]: %u\n", sizeof(t4[0]));
printf("taille de t5[0]: %u\n", sizeof(t5[0]));
```

Nous constatons d'après ces affichages que `t4` et `t5` sont bien des tableaux de tableaux. En effet, la taille de `t4[0]` est égale à 4 fois la taille d'un `float`, ce qui confirme que `t4` est un tableau de 3 tableaux de 4 `float`. De même, la taille de `t5[0]` est égale à 2 fois la taille d'un `int`, ce qui illustre le fait que `t5` est un tableau de 2 tableaux de valeurs de type `int`.

On modifie la taille de `t3` en replaçant sa définition dans le programme ci-dessus par :

```
unsigned int t3[32] = {0,1,2,3};
```

Nous constatons à l'exécution du programme que `t3` a maintenant la taille de 32 fois la taille d'un `unsigned int`. Notons que l'initialisation de ce tableau n'affecte une valeur déterminée qu'aux 4 premiers éléments de `t3` laissant indeterminée la valeurs des autres éléments.

Exercice 2:

La file considérée ici est la suite des 10 valeurs contenues dans le tableau `t`. Il s'agit d'une file dont la longueur est connue, nous allons donc utiliser un traitement structuré sur l'instruction `for`. Nous reprenons point à point les éléments constitutifs du traitement de file :

- `N` vaut 10
- `<obtenir l'élément i>` l'élément de rang `i` dans `t` s'obtient facilement au moyen de l'opérateur `[.]` par : `t[i]`
- `<traiter l'élément i>` revient à afficher cet élément. Pour cela, nous utilisons `printf`. Rappelons que `t` est un tableau d'entiers, donc le format utilisé pour afficher un élément de `t` est `%d` : `printf("%d\n", t[i])`
- `<initialisations>` aucune initialisation n'est nécessaire ici
- `<présentation du résultat>` le résultat se limite à l'affichage des éléments, il n'y a donc rien à faire de plus en fin de boucle

Nous obtenons ainsi le programme suivant :

```
#include <stdio.h>

int main() {
  int i, t[10] = {0,1,2,3,4,5,6,7,8,9};

  for (i=0; i<10; i++)
    printf("%d\n", t[i]);

  return 0;
}
```

Exercice 3:

Nous sommes en présence d'une file de longueur connue : celle des 1000 premiers entiers. Nous allons donc utiliser une structure de traitement de file basée sur une boucle `for`.

Le langage C permet de définir des constantes symboliques pour rendre les programmes plus génériques. Ainsi, ici nous pouvons définir la constante `K` :

```
#define K 1000
```

Nous pouvons ensuite nous servir de `K` partout où la taille du tableau `t` est nécessaire : pour sa définition, dans la boucle `for`, etc. `K` est ensuite remplacé par la valeur 1000 (conformément à sa définition) lors de la compilation du programme. Ceci nous permet de faire fonctionner le programme pour des tableaux plus petits ou plus grands en ne modifiant que la définition de `K` dans le programme.

Nous complétons maintenant le schéma de traitement de file :

- `N` vaut ici `K`
- `<obtenir l'élément i>` le ième élément de la file est le ième entier contenu dans la variable `i`
- `<traiter l'élément i>` revient à donner à `t[i]` la valeur du ième entier : `t[i] = i`
- `<initialisations>` aucune initialisation n'est nécessaire ici
- `<présentation du résultat>` le résultat se limite à l'inversion des éléments de `t`

Le traitement de file pour l'affichage de `t` a déjà été étudié lors de l'exercice précédent. Nous obtenons donc le programme suivant :

```
#include <stdio.h>

#define K 1000

int main() {
  int i, t[K];

  /* Remplissage */
  for (i=0; i<K; i++)
    t[i] = i;
```

Nous obtenons ainsi le programme suivant :

```
#include <stdio.h>

int main() {
  int i, t[10] = {0,1,2,3,4,5,6,7,8,9};

  for (i=0; i<10; i++)
    printf("%d\n", t[i]);

  return 0;
}
```

Exercice 3:

Nous sommes en présence d'une file de longueur connue : celle des 1000 premiers entiers. Nous allons donc utiliser une structure de traitement de file basée sur une boucle `for`.

Le langage C permet de définir des constantes symboliques pour rendre les programmes plus génériques. Ainsi, ici nous pouvons définir la constante `K` :

```
#define K 1000
```

Nous pouvons ensuite nous servir de `K` partout où la taille du tableau `t` est nécessaire : pour sa définition, dans la boucle `for`, etc. `K` est ensuite remplacé par la valeur 1000 (conformément à sa définition) lors de la compilation du programme. Ceci nous permet de faire fonctionner le programme pour des tableaux plus petits ou plus grands en ne modifiant que la définition de `K` dans le programme.

Nous complétons maintenant le schéma de traitement de file :

- `N` vaut ici `K`
- `<obtenir l'élément i>` le ième élément de la file est le ième entier contenu dans la variable `i`
- `<traiter l'élément i>` revient à donner à `t[i]` la valeur du ième entier : `t[i] = i`
- `<initialisations>` aucune initialisation n'est nécessaire ici
- `<présentation du résultat>` le résultat se limite à l'inversion des éléments de `t`

Le traitement de file pour l'affichage de `t` a déjà été étudié lors de l'exercice précédent. Nous obtenons donc le programme suivant :

```
#include <stdio.h>

#define K 1000

int main() {
  int i, t[K];

  /* Remplissage */
  for (i=0; i<K; i++)
    t[i] = i;
```

```
#include <stdio.h>

int main() {
  char s[] = "Bonjour!";
  int i;

  /* Affichage direct */
  printf("%s\n", s);

  /* Affichage caractere par caractere */
  i = 0;
  while (s[i] != '\0') {
    printf("%c", s[i]);
    i++;
  }
  printf("\n");

  return 0;
}
```

Remarque : l'affichage caractère par caractère reproduit fidèlement le comportement de `printf` en présence du format `%s`.

Exercice 7:

Le calcul de la longueur de la chaîne de caractères correspond à un traitement de file avec butée non traitée. Nous devons donc compléter le schéma de programme :

- `<obtenir le 1er élément>` pour obtenir un élément de la chaîne, nous utilisons l'opérateur `[.]`. Afin de décrire chaque élément successivement, nous introduisons une variable `i` de type `int` qui vaut successivement 0, 1, 2, etc. Le premier élément est donc obtenu par `s[i]` pour `i` valant 0
- `<obtenir l'élément suivant>` l'obtention de l'élément suivant se fait en incrémentant la valeur de `i` : `i++`, puis en accédant à l'élément par `s[i]`
- `<élément != butée>` il s'agit de vérifier que l'élément courant `s[i]` est différent de la butée `'\0'`, donc : `s[i] != '\0'`
- `<traiter l'élément courant>` consiste à compter le nombre de caractères dans la chaîne (hors la butée). La variable `i` décrit un à un les indices des caractères dans la chaînes. Le premier indice étant 0, lorsque la butée est rencontrée, `i` contient la longueur de la chaîne. Le traitement consiste ici à compter les caractères en incrémentant la valeur de `i`, ce qui se confond ici avec l'obtention de l'élément suivant.
- `<initialisations>` pour accéder au premier élément de la chaîne par `s[i]`, `i` doit être initialisée à 0
- `<présentation du résultat>` une fois que nous avons compté tous les caractères de la chaîne, il nous reste à afficher cette valeur : `printf("Longueur de '%s' : %d\n", s, i)`

Nous obtenons ainsi le programme suivant :

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

```
  }

  return 0;
}
```

2 Chaînes de caractères

Exercice 6:

La définition d'une chaîne de caractères se fait comme pour tous les tableaux, par exemple :

```
char s[10];
```

Pour son initialisation, nous avons deux choix. Soit comme nous l'avons fait pour les tableaux jusqu'ici :

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

en prenant garde de ne pas oublier la butée `'\0'`. Soit au moyen des constantes de type chaîne de caractères :

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

dans ce dernier cas, la butée `'\0'` est automatiquement placée en fin de chaîne.

Pour l'affichage de la chaîne, nous avons deux choix là encore. Nous pouvons soit utiliser le format `%s` reconnu par `printf` :

```
printf("%s\n", s);
```

Soit, nous optons pour un affichage caractère par caractère de cette chaîne. Nous nous retrouvons alors face à un traitement de file avec butée non traitée comme les autres éléments :

- `<obtenir le 1er élément>` pour obtenir un élément de la chaîne, nous utilisons l'opérateur `[.]`. Afin de décrire chaque élément successivement, nous introduisons une variable `i` de type `int` qui vaut successivement 0, 1, 2, etc. Le premier élément est donc obtenu par `s[i]` pour `i` valant 0
- `<obtenir l'élément suivant>` l'obtention de l'élément suivant se fait en incrémentant la valeur de `i` : `i++`, puis en accédant à l'élément par `s[i]`
- `<élément != butée>` il s'agit de vérifier que l'élément courant `s[i]` est différent de la butée `'\0'`, donc : `s[i] != '\0'`
- `<traiter l'élément courant>` consiste à afficher le caractère courant dans la chaîne : `printf("%c", s[i])`
- `<initialisations>` pour accéder au premier élément de la chaîne par `s[i]`, `i` doit être initialisée à 0
- `<présentation du résultat>` une fois que nous avons affiché tous les caractères de la chaîne, nous pouvons passer à la ligne pour plus de lisibilité : `printf("\n")`

Ceci nous donne finalement le programme :

```
  /* Affichage */
  for (i=0; i<K; i++)
    printf("%d\n", t[i]);

  return 0;
}
```

Remarque : il suffit de changer la définition de `K` pour traiter une file de taille différente.

Exercice 4:

L'inversion des éléments de `t` se fait «en place». Une fois que `t[0]` et `t[K-1]` (`K` étant le nombre d'éléments de `t`) ont été inversés lors du traitement de `t[0]`, il n'y a pas lieu de le refaire lors du traitement de `t[K-1]`.

Nous en déduisons alors que la file considérée ici est la suite des indices des éléments de `t` de `0` à `K/2`. Il s'agit d'une file dont on connaît le nombre d'éléments (la valeur de `K` est fixée à sa définition), nous mettons donc en place un traitement de file basée sur une boucle `for`.

Comme pour l'exercice précédent, nous définissons une constante `K` pour rendre notre programme facilement adaptable. Nous complétons maintenant le schéma de traitement de file :

- `N` vaut ici `K/2`
- `<obtenir l'élément i>` le ième élément de la file est simplement la valeur de `i`
- `<traiter l'élément i>` revient à échanger les valeurs de `t[i]` et `t[K-1-i]`. Pour cela, nous utilisons une nouvelle variable `x` de type `int` (comme les éléments de `t`) :

  ```
  x = t[i];
  t[i] = t[K-1-i];
  t[K-1-i] = x;
  ```
- `<initialisations>` aucune initialisation n'est nécessaire ici
- `<présentation du résultat>` le résultat se limite à l'inversion des éléments de `t`

Nous ne détaillons pas les traitements de file liés au remplissage et à l'affichage du tableau `t` qui ont été vus précédemment. Nous obtenons alors le programme :

```
#include <stdio.h>

#define K 11

int main() {
  int i, x, t[K];

  /* Remplissage */
  for (i=0; i<K; i++)
    t[i] = i;

  /* Affichage avant inversion */
  printf("Avant inversion : \n");
  for (i=0; i<K; i++)
    printf("%d  ", t[i]);
```

- `<initialisations>` aucune initialisation n'est nécessaire ici
- `<présentation du résultat>` aucun résultat n'est à présenter ici.

Nous complétons alors le schéma de programme précédent :

```
for (i=0; i<K; i++) {
  for (j=0; j<M; j++) {
    m3[i][j] = m1[i][j]+m2[i][j];
  }
}
<présentation du résultat>
```

Il nous reste à attaquer la `<présentation du résultat>`, c'est à dire l'affichage de `m3`. La file que nous considérons ici est à longueur `K` connue : il s'agit de la suite des lignes de `m3`. Nous avons alors le traitement de file :

- `N` vaut ici `K`
- `<obtenir l'élément>` il s'agit de `m3[i]`
- `<traiter l'élément>` revient à afficher la ligne `m3[i]`. Il s'agit là encore d'une file de valeur dont le traitement sera considéré ultérieurement
- `<initialisations>` aucune initialisation n'est nécessaire ici
- `<présentation du résultat>` une fois la ligne affichée, nous passons à la ligne suivante (à l'écran) : `printf("\n")` pour plus de lisibilité.

Ceci nous permet de préciser un peu plus le schéma de programme :

```
/* Calcul de la somme */
for (i=0; i<K; i++) {
  for (j=0; j<M; j++) {
    m3[i][j] = m1[i][j]+m2[i][j];
  }
}

/* Affichage de m3 */
for (i=0; i<K; i++) {
  <traiter l'élément m3[i]>
  printf("\n");
}
```

Pour finir, nous devons considérer `<traiter l'élément m3[i]>` qui consiste à afficher la ième ligne de la matrice `m3`. Celle-ci peut être vue comme une file dont on connaît la longueur `M`. Nous devons donc compléter le schéma de traitement de file :

- `N` vaut ici `M`
- `<obtenir l'élément>` revient à obtenir le nombre flottant en ligne `i`, colonne `j` dans `m3` : `m3[i][j]`
- `<traiter l'élément>` consiste en l'affichage de cette valeur : `printf("%f ", m3[i][j])`
- `<initialisations>` aucune initialisation n'est nécessaire ici
- `<présentation du résultat>` néant

Nous pouvons alors compléter notre programme :

```
/* Calcul de la somme */
for (i=0; i<K; i++) {
```

```
  for (j=0; j<M; j++) {
    m3[i][j] = m1[i][j]+m2[i][j];
  }
}

/* Affichage de m3 */
for (i=0; i<K; i++) {
  for (j=0; j<M; j++) {
    printf("%f ", m3[i][j]);
  }
  printf("\n");
}
```

Afin d'obtenir un programme complet, il nous reste à définir les variables et à initialiser `m1` et `m2`. Ceci se fait sur un double traitement de file, d'une manière similaire à l'affichage de `m3`. Voici le programme obtenu finalement (ici, nous avons choisi d'initialiser `m1[i][j]` à la valeur `i*j` et `m2[i][j]` à la valeur `i+j`) :

```
#include <stdio.h>

#define K 3
#define M 2

int main() {
  float m1[K][M], m2[K][M], m3[K][M];
  int i, j;

  /* Initialisations */
  for (i=0; i<K; i++) {
    for (j=0; j<M; j++) {
      m1[i][j] = i*j;
      m2[i][j] = 1+j;
    }
  }

  /* Calcul de la somme */
  for (i=0; i<K; i++) {
    for (j=0; j<M; j++) {
      m3[i][j] = m1[i][j]+m2[i][j];
    }
  }

  /* Affichage de m3 */
  for (i=0; i<K; i++) {
    for (j=0; j<M; j++) {
      printf("%f ", m3[i][j]);
    }
    printf("\n");
```

```
int main() {
  int i;
  char s[] = "Bonjour!";

  i=0;
  while (s[i] != '\0') {
    i++;
  }
  printf("Longueur de '%s' : %d\n", s, i);

  return 0;
}
```

Remarque : la longueur de la chaîne est bien le nombre de caractères hors `'\0'`.

Exercice 8:

Nous disposons donc de deux chaînes `s1` et `s2` que nous devons afficher dans l'ordre lexicographique. Nous devons donc comparer ces deux chaînes caractère par caractère pour déterminer cet ordre. Notre file est donc constituée de paires d'éléments `s1[i],s2[i]` à comparer pour déterminer l'ordre entre ces deux chaînes. Le traitement s'arrête lorsque celui-ci a été trouvé, c'est à dire, soit lorsqu'un couple `s1[i],s2[i]` tel que `s1[i]` $\neq$ `s2[i]` a été trouvé, soit lorsque `s1[i] = s2[i] = '\0'`. Nous sommes donc en présence d'une file avec butée non traitée comme les autres éléments puisque le traitement se limite à considérer l'élément suivant, alors qu'il ne faut pas le faire une fois la butée trouvée.

Nous pouvons donc compléter notre schéma de traitement de file :

- `<obtenir le 1er élément>` le premier élément est `s1[i],s2[i]` pour `i` valant 0. Nous utilisons donc une variable `i` pour décrire les différents éléments de la file
- `<obtenir l'élément suivant>` l'obtention de l'élément suivant se fait en incrémentant la valeur de `i` : `i++`, puis en accédant au nouvel élément `s1[i],s2[i]`
- `<élément != butée>` il s'agit de vérifier que `(s1[i] == s2[i]) && (s1[i] != '\0')` (il n'est pas nécessaire d'ajouter `s2[i] != '\0'`)
- `<traiter l'élément courant>` il n'y a pas ici de traitement à proprement parler. Le parcours de la file a pour seul but de déterminer l'ordre lexicographique entre `s1` et `s2`
- `<initialisations>` pour accéder au premier élément de la file, `i` doit être initialisée à 0
- `<présentation du résultat>` une fois que nous sommes sortis de la boucle, il suffit de comparer `s1[i]` et `s2[i]` pour connaître l'ordre entre les deux chaînes. En effet, soit `s1[i]` $\neq$ `s2[i]`, alors le plus petit caractère donne la plus petite chaîne. Notons que si une chaîne est plus courte que l'autre, alors ceci est toujours vrai puisqu'alors `s1[i]` $= 0$ et `s2[i]` > 0 (ou inversement). Soit, `s1[i]` et `s2[i]` sont égaux à `'\0'`, alors les deux chaînes sont égales :

  ```
  if (s1[i] < s2[i]) {
    printf("%s\n", s1);
    printf("%s\n", s2);
  }
  ```

```
    else {
      printf("%s\n", s2);
      printf("%s\n", s1);
    }
```

Nous obtenons donc le programme suivant :

```
#include <stdio.h>

int main() {
  int i;
  char s1[] = "abcde";
  char s2[] = "abc";

  i = 0;
  while ((s1[i] == s2[i]) && (s1[i] != '\0')) {
    i++;
  }
  if (s1[i] < s2[i]) {
    printf("%s\n", s1);
    printf("%s\n", s2);
  }
  else {
    printf("%s\n", s2);
    printf("%s\n", s1);
  }

  return 0;
}
```

Exercice 9:

Nous définissons donc une variable chaîne de caractères `s` que nous supposons initialisée. Nous nous intéressons ici à la transformation de `s` en majuscules. La file considérée ici est la suite des caractères de `s`. Il s'agit d'une file avec butée non traitée comme les autres : il n'y a pas lieu de la passer en majuscules.

- `<obtenir le 1er élément>` le premier élément est `s[i]` pour `i` valant 0. Nous utilisons donc une variable `i` pour décrire les différents éléments de la file
- `<obtenir l'élément suivant>` l'obtention de l'élément suivant se fait en incrémentant la valeur de `i` : `i++`, puis en accédant au nouvel élément `s[i]`
- `<élément != butée>` il s'agit de vérifier que `(s[i] != '\0')`
- `<traiter l'élément courant>` le traitement consiste à passer `s[i]` en majuscule, soit : `s[i] = toupper(s[i])` (il faudra inclure la librairie `ctype.h` pour utilier `toupper`)
- `<initialisations>` pour accéder au premier élément de la file, `i` doit être initialisée à 0
- `<présentation du résultat>` une fois que nous sommes sortis de la boucle, il faut afficher `s` à l'aide de `printf`, ce qui donne : `printf("%s\n", s)`

Nous obtenons maintenant le programme suivant :

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

int main() {
  char s[] = "a mettre en majuscules !";
  int i;

  i = 0;
  while (s[i] != '\0') {
    s[i] = toupper(s[i]);
    i++;
  }
  printf("%s\n", s);

  return 0;
}
```

Exercice 10:

Nous nous intéressons à la recopie de `s1` dans `s2`. La file considérée ici est la suite des caractères de `s1`. Il s'agit d'une file dont la butée (`'\0'`) n'est pas traitée comme les autres éléments : nous recopions simplement les caractères de `s1`. Nous nous orientons donc sur une structure de contrôle `while` que nous complétons :

- `<obtenir le 1er élément>` le premier élément est `s1[i]` pour `i` valant 0. Nous utilisons donc une variable `i` pour décrire les différents éléments de la file
- `<obtenir l'élément suivant>` l'obtention de l'élément suivant se fait en incrémentant la valeur de `i` : `i++`, puis en accédant au nouvel élément `s1[i]`
- `<élément != butée>` il s'agit de vérifier que `(s[i] != '\0')`
- `<traiter l'élément courant>` le traitement de la file consiste à copier `s1` dans `s2` caractère par caractère : `s2[i] = s1[i]`
- `<initialisations>` pour accéder au premier élément de la file, `i` doit être initialisée à 0
- `<présentation du résultat>` une fois que nous sommes sortis de la boucle, il faut d'une part ajouter la butée `'\0'` à `s2` : `s2[i] = '\0'` (`i` contient la position où la butée doit être ajoutée), et d'autre part afficher `s2` à l'aide de `printf`, ce qui donne : `printf("%s\n", s2)`

Nous obtenons alors le programme suivant :

```
#include <stdio.h>

int main() {
  char s1[] = "abcd", s2[5];
  int i;

  i = 0;
```

```
while (s1[i] != '\0') {
    s2[i] = s1[i];
    i++;
  }
  s2[i] = '\0';
  printf("%s\n", s2);

  return 0;
}
```

Il se pose un gros problème si la taille de `s2` est plus petite que celle de `s1` (`s2` faisant moins de 5 caractères dans le programme précédent par exemple). Dans ce cas, nous allons recopier autant de caractères qu'il y en a dans `s1` ce qui va nous conduire à écrire en dehors de `s2`, dans des emplacements mémoire qui ne nous sont pas attribués.

La solution consiste à borner le nombre de caractères de `s1` copiés dans `s2`. Nous en copierons au plus `sizeof(s2)-1` car il faut conserver une case libre pour la butée `'\0'` de la chaîne. La file considérée maintenant est donc constituée de paires `i,s1[i]`. Les butées de cette file sont les couples tels que $\texttt{i} \geq \texttt{sizeof(s2)}$ ou `s1[i] = '\0'`. La butée de cette file n'est pas traitée comme les autres éléments : le caractère ne doit pas être ajouté à `s2` sous peine de débordement.

Nous pouvons maintenant compléter notre programme :

- `<obtenir le 1er élément>` le premier élément est `i,s1[i]` pour `i` valant 0. Nous utilisons donc une variable `i` pour décrire les différents éléments de la file
- `<obtenir l'élément suivant>` l'obtention de l'élément suivant se fait en incrémentant la valeur de `i` : `i++`, puis en accédant au nouvel élément `i,s1[i]`
- `<élément != butée>` il s'agit de vérifier que `(i < sizeof(s2)-1) && (s1[i] != '\0')`
- `<traiter l'élément courant>` le traitement consiste à copier `s1[i]` en même place dans `s2` : `s2[i] = s1[i]`
- `<initialisations>` pour accéder au premier élément de la file, `i` doit être initialisée à 0
- `<présentation du résultat>` une fois que nous sommes sortis de la boucle, il faut ajouter la butée `'\0'` à `s2` pour que celle-ci soit bien une chaîne de caractères. Notons qu'à la sortie de la boucle `i` contient nécessairement la position où la butée doit être ajoutée à `s2`. Enfin, nous devons afficher `s2` :

  ```
  s2[i] = '\0';
  printf("%s\n", s2);
  ```

Ceci nous conduit au programme :

```
#include <stdio.h>

int main() {
  char s1[] = "abcd", s2[3];
  int i;

  i = 0;
  while ((i < sizeof(s2)-1) && (s1[i] != '\0')) {
    s2[i] = s1[i];
    i++;
```

```
  }
  s2[i] = '\0';
  printf("%s\n", s2);

  return 0;
}
```

Remarque : dans ces deux programmes, la butée `'\0'` est finalement ajoutée à `s2`. Celle-ci étant présente à la fin de `s1`, on pourrait considérer que la butée de `s1` est traitée comme les autres caractères de la file. Ceci est possible, mais nous avons fait un choix différent. Pourquoi ?

Exercice 11:

Le problème considéré ici est la concaténation de `s2` à `s1`. Le schéma suivant montre ce que nous souhaitons obtenir :

```
Avant :
     ------------------
s1 : |B|o|n|\0|?|?|?|?|
     ------------------

     ------------
s2:  |j|o|u|r|\0|
     ------------

Après :
     ------------------
s1 : |B|o|n|j|o|u|r|\0|
     ------------------

     ------------
s2:  |j|o|u|r|\0|
     ------------
```

Nous voyons donc le principe : tout d'abord nous devons nous placer à la fin de la chaîne `s1`, plus précisément sur la case du tableau qui contient le cacarctère `'\0'`, puis recopier `s2` à partir de cette position dans `s1`. Nous avons donc deux traitements de file à réaliser.

Le premier concerne le positionnement à la fin de `s1`. Il s'agit de parcourir la file des caractère de `s1` un à un. Lorsque le caractère considéré est différent de `'\0'`, on passe au caractère suivant. Lorsque le caractère `'\0'` est atteint, on s'arrête. Notre file est donc la suite des caractères constituant `s1`, terminée par la butée `'\0'` qui n'est pas traitée comme les autres caractères. Nous procédons à l'analyse de traitement de file :

- `<obtenir le 1er élément>` le premier élément est `s1[i]` pour `i` valant 0. Nous utilisons donc une variable `i` pour décrire les différents éléments de la file
- `<obtenir l'élément suivant>` l'obtention de l'élément suivant se fait en incrémentant la valeur de `i` : `i++`, puis en accédant au nouvel élément `s1[i]`
- `<élément != butée>` il s'agit de vérifier que `(s1[i] != '\0')`

- `<traiter l'élément courant>` il n'y a pas de traitement de l'élément courant : on se contente de passer à l'élément suivant puisqu'on recherche la position de la butée dans `s1`
- `<initialisations>` pour accéder au premier élément de la file, `i` doit être initialisée à 0
- `<présentation du résultat>` une fois que nous sommes sortis de la boucle, `i` contient la position de `'\0'` dans `s1` qui est la valeur recherchée. Il n'y a donc pas de résultat à présenter

Nous obtenons alors le début de programme suivant :

```
i = 0;
while (s1[i] != '\0')
  i++;
```

Il nous reste ensuite à recopier la chaîne `s2` à partir de la position `i` dans `s1`. Cette recopie fonctionne exactement comme à l'exercice précédent : nous ne procédons donc pas à nouveau à l'analyse de traitement de file. Notez cependant une légère variation puisque nous devons copier le caractère `s2[0]` dans `s1[i]`, le caractere `s2[1]` dans `s1[i+1]` et ainsi de suite. Nous introduisons donc une variable `j` qui parcourt la chaîne `s2` (elle remplace la variable `i` dans l'exercice précédent), et la recopie s'écrit donc : `s1[i+j] = s2[j]`.

Nous obtenons donc le programme suivant :

```
#include <stdio.h>

int main() {
  char s1[8] = "Bon", s2[] = "jour";
  int i, j;

  /* Placement sur la butee '\0' dans s1 */
  i = 0;
  while (s1[i] != '\0')
    i++;

  /* Recopie de s2 a partir de la position i dans s1 */
  j = 0;
  while (s2[j] != '\0') {
    s1[i+j] = s2[j];
    j++;
  }
  s1[i+j] = '\0';
  printf("%s\n", s1);

  return 0;
}
```

Il se pose le même problème de dépassement de la taille de `s1` que dans l'exercice précédent (pour `s2`). Nous ne détaillons pas à nouveau le traitement de file. Voici le programme obtenu :

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

```
  printf("\n");

  /* Inversion */
  for (i=0; i<K/2; i++) {
    x = t[i];
    t[i] = t[K-1-i];
    t[K-1-i] = x;
  }

  /* Affichage apres inversion */
  printf("Apres inversion : \n");
  for (i=0; i<K; i++)
    printf("%d  ", t[i]);
  printf("\n");

  return 0;
}
```

Exercice 5:

Une matrice est un tableau de tableau, ce qui en terme de matrice se traduit par un tableau de lignes. Nous avons donc une première file constituée de paires de lignes de `m1` et `m2`, dont la longueur est connue (définie par un constante symbolique `K`). Nous pouvons donc compléter une partie du schéma de programme :

- `N` vaut ici `K`
- `<obtenir l'élément i>` il s'agit de la paire constituée par la ième ligne de `m1` et la ième ligne de `m2`, c'est à dire `m1[i],m2[i]`
- `<traiter l'élément i>` revient à sommer ces deux lignes. Ceci constitue un second traitement de file que nous analyserons par la suite
- `<initialisations>` aucune initialisation n'est nécessaire ici
- `<présentation du résultat>` le résultat se limite à afficher `m3`. Là encore, il s'agit d'un autre traitement de file que nous analyserons par la suite

Nous obtenons donc dans un premier temps un schéma de programme partiellement completé :

```
for (i=0; i<K; i++) {
  <traiter m1[i] et m2[i]>
}
<présentation du résultat>
```

Nous nous intéressons maintenant à `<traiter m1[i] et m2[i]>`. Il s'agit d'une file de longueur `M` (constante symbolique) connue dont chaque élément est une paire `m1[i][j],m2[i][j]` où `m1[i][j]` est l'élément en ligne `i`, colonne `j` de `m1` (de même pour `m2[i][j]`). Nous pouvons donc écrire le schéma de traitement de file correspondant :

- `N` vaut ici `M`
- `<obtenir l'élément>` il s'agit de la paire `m1[i][j],m2[i][j]`
- `<traiter l'élément>` revient à sommer ces deux valeurs et à placer le résultat dans `m3` aux coordonnées ligne,colonne correspondantes : `m3[i][j] = m1[i][j]+m2[i][j]`

```c
int main() {
  char s1[8] = "Bon", s2[] = "jour";
  int i, j;

  /* Placement sur la butee '\0' dans s1 */
  i = 0;
  while (s1[i] != '\0')
    i++;

  /* Recopie de s2 a partir de la position i dans s1 */
  j = 0;
  while ((i+j < sizeof(s1)-1) && (s2[j] != '\0')) {
    s1[i+j] = s2[j];
    j++;
  }
  s1[i+j] = '\0';
  printf("%s\n", s1);

  return 0;
}
```