ENSEIRB 2007-2008 - Première année
PG101 - Programmation en C - `http://www.enseirb.fr/~herbrete/PG101/`

TD 2 : Traitements de files (commentaires)

1 Files de longueur connue

Les files de longueur connue peuvent être traitées en énumérant leurs éléments du premier au dernier. Chaque élément est traité de manière identique aux autres.

Puisqu'on connaît par avance le nombre d'éléments à traiter, la structure de contrôle usuelle pour ce type de file est le `for`. Nous allons donc écrire un programme C qui a la structure suivante :

```
<initialisations>
for (i=0; i<N; i++) {
  <obtenir l'élément i>
  <traiter l'élément i>
}
<présentation du résultat>
```

où `N`
est la longueur de la file. Il nous reste donc à spécifier, pour chaque file à traiter, les différents éléments inconnus du schéma de programme présenté ci-dessus :

- `N` qui est la longueur de la file
- `<obtenir l'élément i>` qui correspond au «calcul» de l'élément de rang *i* dans la file
- `<traiter l'élément i>`
  qui correspond au traitement d'un élément de la file (ils sont tous traités de la même façon)
- `<initialisations>` qui correspond à des traitements à réaliser avant celui de la file
- `<présentation du résultat>`
  qui correspond à la production du résultat attendu à l'issue du traitement de la file

Exercice 1:

Comment est définie la file vide ?

Une file vide est simplement une file de longueur 0.

Exercice 2:

La file considérée ici est la suite des 15 premiers entiers naturels, donc de 0 à 14.

Pour écrire le programme, nous allons donc remplir le schéma décrit ci-dessus.

- `N` vaut donc 15
- `<obtenir l'élément i>` revient à utiliser la valeur de `i`. Le passage à l'élément suivant sera automatique du fait de l'instruction `i++` dans la boucle `for`.
- `<traiter l'élément i>`
  consiste à ajouter sa valeur à la somme des entiers précédents. Nous allons donc utiliser une variable `s` pour stocker la somme des entiers de 0 à `i`. Le traitement consiste donc en l'ajout de `i` à `s`, soit `s = s+i`
- `<initialisations>` il faut initialiser la variable `s` à zéro : `s = 0` puisqu'avant le début du traitement de la file, la somme des valeurs lues vaut zéro
- `<présentation du résultat>` consiste simplement à afficher la valeur de `s` : `printf("s=%d\n", s)`

Ceci donne le programme (il suffit de remplacer les éléments) :

```
#include <stdio.h>

int main() {
  int i, s;

  s = 0;
  for (i=0; i<15; i++) {
    s = s+i;
  }

  printf("s=%d\n", s);

  return 0;
}
```

Exercice 3:

La file considérée dans cet exercice est une suite de 10 entiers.

- `N` vaut 10
- `<obtenir l'élément i>` nous allons utiliser `scanf` pour lire des entiers au clavier. Pour cela, nous avons besoin d'une variable `x` de type `int` qui stocke l'entier lu : `scanf("%d", &x)`.

  Remarque : il est nécessaire d'ajouter le symbole `&` devant `x` dans l'instruction `scanf`. Ceci sera expliqué dans la suite des TD, pour le moment nous en admettrons simplement la nécessité.
- `<traiter l'élément i>` le traitement consiste ajouter l'entier `x` à la somme des entiers précédemment lus. Pour cela, nous utilisons une variable `s` qui stocke la somme des entiers précédemment lus : `s = s+x`
- `<initialisations>` il faut initialiser la variable `s` à zéro : `s = 0` puisque n'ayant lu aucune valeur de la file, leur somme vaut zéro
- `<présentation du résultat>` il s'agit simplement d'afficher la valeur de `s` : `printf("s=%d\n", s)`

Ce qui donne :

```
#include <stdio.h>

int main() {
  int i, x, s;

  s = 0;
  for (i=0; i<10; i++) {
    scanf("%d", &x);
    s = s+x;
  }

  printf("s=%d\n", s);

  return 0;
}
```

Exercice 4:

La file considérée ici est une suite de 8 entiers.

Nous remplissons donc les trous de notre schéma de programme :

- `N` vaut 8
- `<obtenir l'élément i>` nous allons le lire au clavier en utilisant `scanf`. La valeur lue sera stockée dans la variable `x` : `scanf("%d", &x)`
- `<traiter l'élément i>` nous devons afficher l'opposé de la valeur de `x`, donc : `printf("%d\n", -x)`
- `<initialisations>` aucune initialisation n'est nécessaire
- `<présentation du résultat>` aucun traitement du résultat n'est nécessaire

Ceci donne le programme :

```
#include <stdio.h>

int main() {
  int i, x;

  for (i=0; i<8; i++) {
    scanf("%d", &x);
    printf("%d\n", -x);
  }

  return 0;
}
```

Exercice 5:

La file considérée est une suite de 12 entiers.

Nous complétons le schéma de programme :

- `N` vaut 12
- `<obtenir l'élément i>` par `scanf` en utilisant une variable `x` pour stocker la valeur lue : `scanf("%d", &x)`
- `<traiter l'élément i>` nous devons utiliser une variable `nb_pos` pour compter le nombre de valeurs lues positives ou nulles. Le traitement d'un élément de la file consiste à ajouter 1 à la valeur de `nb_pos` si la valeur de `x` est positive ou nulle :

  ```
  if (x >= 0)
    nb_pos++;
  ```

- `<initialisations>` il faut initialiser `nb_pos` à zéro, puisque lorsqu'on n'a pas commencé à traiter la file, il y a zéro valeurs lues positives ou nulles : `nb_pos = 0`
- `<présentation du résultat>` il faut afficher la valeur de `nb_pos` : `printf("nb_pos=%d\n", nb_pos)`

Ce qui donne le programme :

```
#include <stdio.h>

int main() {
  int i, x, nb_pos;

  nb_pos = 0;
  for (i=0; i<12; i++) {
    scanf("%d", &x);
    if (x >= 0)
      nb_pos++;
  }
  printf("nb_pos=%d\n", nb_pos);

  return 0;
}
```

Exercice 6:

La file considérée ici est la suite des monômes $x^i/i!$ des rangs i=0 à i=10

Nous pouvons alors compléter notre schéma de programme :

- `N` vaut 10
- `<obtenir l'élément i>` nous avons une variable `x`, de type `float` initialisée à `3.5`. Nous allons stocker la valeur du monôme de rang i dans une variable `m` de type `float`. Notons que le monôme de rang i+1 s'obtient depuis celui de rang i en le multipliant par `x` et en le divisant par i, le monôme de rang 0 étant 1. Ainsi, nous l'obtenons par : `m = m*x/i`
- `<traiter l'élément i>` le calcul de $exp(x)$ consiste à sommer les monômes jusqu'au rang souhaité. Pour cela, nous utilisons une variable `s` de type `float`
  pour stocker la somme des monômes précédents. Le traitement consiste à ajouter la valeur de `m` à celle de `s` : `s = s+m`
- `<initialisations>` il convient d'initialiser la somme à 1 qui correspond au monôme de rang 0 : `s = 1` puisqu'avant tout traitement de la file, c'est la valeur approchée de $exp(x)$. De même, `m` est initialisée à 1 puisque c'est la valeur du monôme de rang 0
- `<présentation du résultat>` il suffit d'afficher la valeur de `s` : `printf("s=%d\n", s)`

Ce qui nous donne :

```
#include <stdio.h>

int main() {
  float x=3.5, m, s;
  int i;

  s = 1;
  m = 1;
  for (i=1; i<=10; i++) {
    m = m*x/i;
    s = s+m;
  }
  printf("exp(%f)=%f\n", x, s);

  return 0;
}
```

Remarque : nous avons décalé les valeurs de *i*
dans la boucle de façon à les faire correspondre au rang des monômes. Nous aurions tout aussi bien pu modifier l'expression du calcul de `m`.

2 Files avec butée non traitée

Les files avec butée non traitée sont constituée d'une suite d'éléments de même type dont le dernier (la *butée*) est distingué des autres. Chaque élément est traité de la même façon que les autres, à l'exception du dernier élément.

Ici, nous ne connaissons pas à l'avance le nombre d'éléments. De plus, la *butée* ne doit pas être traitée comme les autres éléments, donc à chaque obtention d'un élément de la file, il faut tester si celui-ci est la butée ou non avant d'entreprendre tout traitement. La structure de contrôle qui s'applique naturellement dans ce cas est le `while`, ce qui donne le schéma de programme suivant :

```
<initialisations>
<obtenir le 1er élément>
while (<élément != butée>) {
  <traiter l'élément courant>
  <obtenir l'élément suivant>
}
<présentation du résultat>
```

Il nous reste maintenant à spécifier pour chaque file à traiter les éléments suivants :

- `<obtenir le 1er élément>`
  qui consiste en l'obtention du premier élément de la file (qui peut tout à fait être la butée)
- `<obtenir l'élément suivant>`
  qui consiste à obtenir l'élément suivant dans la file. Elle peut différer de l'obtention du premier élément (pour des files complexes)
- `<élément != butée>` comment teste-t-on si l'élément courant de la file est égal ou différent de la butée ?
- `<traiter l'élément courant>`
  quel traitement est appliqué à un élément de la file ? Rappelons qu'ils sont tous traités de la même façon, sauf le dernier (la butée)
- `<initialisations>` correspond à des traitements qui doivent être réalisés avant celui de la file
- `<présentation du résultat>` qui, une fois la file traitée, produit le résultat attendu. Ceci peut inclure un traitement de la butée (différent de celui des autres éléments)

Exercice 7:

Comment est définie la file vide ?

Une file avec butée non traitée contient nécessairement la butée. Une file vide est donc réduite à une seul élément : la butée

Exercice 8:

La file considérée ici est une suite d'entiers qui se termine par l'entier 0.

Nous complétons notre schéma de programme :

- `<obtenir le 1er élément>` les entiers sont lus au clavier. Nous allons donc utiliser `scanf` pour lire cet entier, et une variable `x` de type `int` pour stocker la valeur lue : `scanf("%d", &x)`
- `<obtenir l'élément suivant>`
  il est obtenu de la même façon que le premier : il n'y a pas de différenciation ici
- `<élément != butée>` la butée est l'entier 0. Le test correspond donc à `x != 0`
- `<traiter l'élément courant>`
  le but du programme est de calculer le produit des entiers lus. Il nous faut pour cela une variable `p` de type `int` qui stocke le produit des entiers de la file traités jusqu'ici. Le traitement de l'élément courant consiste à faire le produit de `p` par `x`, soit : `p = p*x`
- `<initialisations>` nous devons initialiser la valeur de `p`. Le mode de traitement des éléments de la file indique de `p` doit être initialisé à 1 (élément neutre pour la multiplication). Notons que si la file est vide, le schéma de programme nous montre que le produit vaut 1
- `<présentation du résultat>` il s'agit simplement d'afficher la valeur de `p` : `printf("p=%d\n", p)`

Ceci nous donne le programme suivant :

```
#include <stdio.h>

int main() {
  int x, p;

  p = 1;
  scanf("%d", &x);
  while (x != 0) {
    p = p*x;
    scanf("%d", &x);
  }
  printf("p=%d\n", p);

  return 0;
}
```

Exercice 9:

La suite considérée ici est une suite de caractères qui se termine par le caractère `'\n'`

Nous complétons le schéma de programme :

- `<obtenir le 1er élément>` l'obtention du premier élément se fait par la lecture d'un caractère à l'aide de `scanf`. Nous utilisons une variable `c` de type `char` pour stocker la valeur lue : `scanf("%c", &c)`
- `<obtenir l'élément suivant>` il n'y a pas de différence pour la lecture des éléments : ils sont tous lus au clavier comme indiqué ci-dessus.
- `<élément != butée>` la file se termine par le caractère `'\n'`. Le test de fin de file s'écrit donc `c != '\n'`
- `<traiter l'élément courant>`
  pour stocker le nombre de caractères lus, nous avons besoin d'une variable `nb_c`. Le traitement d'un élément de la file consiste simplement en l'incrémentation de la valeur de la variable `nb_c` : `nb_c++`
- `<initialisations>`
  lorsque la file n'a pas encore été traitée, nous n'avons lu aucun caractère jusque là, donc nous initialisons `nb_c` à zéro : `nb_c = 0`
- `<présentation du résultat>` il s'agit d'afficher la valeur de `nb_c` : `printf("nb_c=%d\n", nb_c)`

Ceci nous donne le programme :

```
#include <stdio.h>

int main() {
  char c;
  int nb_c;

  nb_c = 0;
  scanf("%c", &c);
  while (c != '\n') {
    nb_c++;
    scanf("%c", &c);
  }
  printf("nb_c=%d\n", nb_c);

  return 0;
}
```

Exercice 10:

La file considérée ici est une suite de caractères terminée par la butée `'\n'`.

Nous devons maintenant répondre aux questions suivantes :

- `<obtenir le 1er élément>` les caractères sont lus au clavier en utilisant la fonction `scanf`. Nous devons utiliser une variable `c` de type `char` pour stocker le caractère lu : `scanf("%c", &c)`
- `<obtenir l'élément suivant>`
  il n'y a pas de distinction entre la lecture du premier des éléments et les suivants
- `<élément != butée>` la file est terminée par le caractère `'\n'`, donc ce test est : `c != '\n'`
- `<traiter l'élément courant>`
  nous devons afficher le caractère lu en convertissant les voyelles de minuscules en majuscules :

  ```
  switch (c) {
    case 'a' :
    case 'e' :
    case 'i' :
  ```

```
      case 'o' :
      case 'u' :
      case 'y' :
        printf("%c\n", toupper(c));
        break;
      default :
        printf("%c\n", c);
        break;
    }
```

Pour l'utilisation de la fonction `toupper`, il faudra penser à inclure le fichier d'en-tête `ctype.h`

Nous pouvons écrire ce code de façon différente :

```
switch (c) {
  case 'a' :
    printf("A\n");
    break;
  case 'e' :
    printf("E\n");
    break;
  case 'i' :
    printf("I\n");
    break;
  case 'o' :
    printf("O\n");
    break;
  case 'u' :
    printf("U\n");
    break;
  case 'y' :
    printf("Y\n");
    break;
  default :
    printf("%c\n", c);
    break;
}
```

Ou encore :

```
if ((c == 'a') || (c == 'e') || (c == 'i') || (c == 'o') ||
    (c == 'u') || (c == 'y'))
  printf("%c\n", toupper(c));
else
  printf("%c\n", c);
```

- `<initialisations>` il n'y a pas d'initialisation ici
- `<présentation du résultat>` il n'y a pas non plus de résultat à présenter à l'issu du traitement de file

Ceci nous donne le programme C suivant :

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

int main() {
  char c;

  scanf("%c", &c);
  while (c != '\n') {
    switch (c) {
      case 'a' :
      case 'e' :
      case 'i' :
      case 'o' :
      case 'u' :
      case 'y' :
        printf("%c\n", toupper(c));
        break;
      default :
        printf("%c\n", c);
        break;
    }
    scanf("%c", &c);
  }

  return 0;
```

```
}
```

Exercice 11:

La file est une suite de paires d'entiers qui se termine par le couple (0,0).

Nous complétons le schéma de traitement de file :

- `<obtenir le 1er élément>` pour obtenir le premier élément, il faut lire deux valeurs entières à l'aide de `scanf`. Pour cela, nous allons utiliser deux variables `x` et `y` de type `int` qui stockeront les valeurs lues : `scanf("%d %d", &x, &y)`
- `<obtenir l'élément suivant>`
  il n'y a pas de différence ici entre l'obtention du premier élément de la file et leurs suivants
- `<élément != butée>` la butée de la file est le couple (0,0). Ce test s'écrit donc : `! ((x == 0)&&(y == 0))` ou encore `(x != 0) || (y != 0)`
- `<traiter l'élément courant>`
  nous devons calculer la longueur du vecteur somme des vecteurs définis par la file. Pour cela, nous utilisons deux variables `sx` et `sy` de type `int`
  qui calculent les coordonnées du vecteur somme. À chaque nouveau vecteur lu, il faut ajouter ses coordonnées à celles du vecteur somme :

  ```
  sx = sx+x;
  sy = sy+y;
  ```

- `<initialisations>` nous devons initialiser `sx` et `sy` à zéro puisque lorsque la file n'a pas encore été traitée, le vecteur somme est (0,0)
- `<présentation du résultat>`
  à l'issue du traitement de la file, nous avons les coordonnées du vecteur somme dans `sx` et `sy`. Nous devons donc calculer la longueur de ce vecteur, puis l'afficher. La longueur du vecteur somme est stockée dans une variable `l` de type `float` :

  ```
  l = sqrt(sx*sx + sy*sy);
  printf("Longueur %f\n", l);
  ```

  Notons qu'il faudra inclure la bibliothèque `math.h` pour la définition de la fonction `sqrt`. Par ailleurs, il faudra lier cette bibliothèque lors de la compilation, qui se fera donc par la commande :

  ```
  gcc prog.c -Wall -lm -o prog
  ```

Ceci nous donne le programme :

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

int main() {
  int x, y, sx, sy;
  float l;

  sx = 0;
  sy = 0;
  scanf("%d %d", &x, &y);
  while ((x != 0) || (y != 0)) {
    sx = sx+x;
    sy = sy+y;
    scanf("%d %d", &x, &y);
  }
  l = sqrt(sx*sx + sy*sy);
```

```
  printf("Longueur %f\n", l);

  return 0;
}
```

Exercice 12:

Le problème qui nous intéresse est le calcul de la longueur de la plus longue sous-suite croissante d'une suite d'entiers terminée par 0. Bien que la suite des valeurs lues soit par exemple celle-ci :

```
1 2 1 7 9 -1 3 8 4 2 0
```

elle ne constitue pas la file. Reformattons la suite d'entiers pour y faire apparaître les sous-suites croissantes :

```
1 2   1 7 9   -1 3 8   4   2   0
```

Ce qui nous intéresse ici est la suite des longueurs de ces sous-suites, soit, sur notre exemple :

```
2 3 3 1 1 0
```

Notons que seule la sous-suite constituée uniquement de l'élément 0 a la longueur 0.

La file *f* est donc ici constituée d'une suite d'entiers terminée par 0, l'élément de rang *i* étant la longueur de la sous-suite croissante de même rang dans la suite d'entiers lus au clavier.

Nous pouvons maintenant compléter notre schéma de programme :

- `<obtenir le 1er élément>` le premier élément de notre file *f* est la longueur de la première sous-suite. Or, cette sous-suite est elle-même une file *f'*. Sur notre exemple, il s'agit de la file `1 2 1` de butée `1` (le second), non traitée comme les autres éléments puisqu'elle ne contribue pas à définir la longueur de cette sous-suite.

  Nous nous trouvons donc face à un autre traitement de file à mettre en oeuvre pour l'obtention des éléments de *f*. Nous considérerons ce problème ultérieurement, et nous supposerons pour le moment que la variable `l` de type `int`
  contient la longueur de la première sous-suite croissante, c'est à dire, le premier élément de notre file *f*
- `<obtenir l'élément suivant>`
  il n'y a pas de distinction entre l'obtention du 1er élément et des autres éléments. Là aussi, nous repondrons donc à ce point lors du traitement de *f'*, et nous considérons maintenant que `l` contient l'élément de la file
- `<élément != butée>`
  lors de l'analyse de la file, nous avons dit que la butée et la longueur de suite 0. Ce test s'écrit donc : `l != 0`
- `<traiter l'élément courant>`
  nous voulons calculer la longueur de la plus grande sous-suite. Pour cela, nous avons besoin d'une variable `l_max`
  qui stocke cette longueur maximale. Le traitement de l'élément courant (la longueur dernièrement obtenue) consiste donc à modifier la valeur de `l_max` au cas où la longueur courante lui est supérieure :

  ```
  if (l > l_max)
    l_max = l;
  ```

- `<initialisations>` il faut que nous initialisions la variable `l_max` en vue de sa comparaison avec `l`. Avant tout traitement de la file, la plus longue sous-suite a la longueur 0, ce qui est cohérent avec le cas de la file vide. Nous avons donc pour initialisation : `l_max = 0`

- `<présentation du résultat>` il s'agit simplement ici d'afficher la valeur de `l_max` comme cela nous est demandé : `printf("longueur max=%d\n" l_max)`

Ceci nous donne le schéma de programme suivant, où il substiste des éléments à remplacer ultérieurement par le traitement de *f'* :

```
l_max = 0;
<obtenir le 1er élément>
while (l != 0) {
  if (l > l_max)
    l_max = l;
  <obtenir l'élément suivant>
}
printf("longueur max=%d\n", l_max);
```

Il nous reste maintenant à décrire le mode d'obtention des éléments de *f*. Comme nous l'avons dit précédemment, il s'agit de mettre en oeuvre un autre traitement d'une file *f'*, définie comme une suite croissante d'entiers dont la butée est un entier inférieur à l'entier le précédent dans la suite, ou zéro (butée de la suite d'entiers lus au clavier)

- `<obtenir le 1er élément>` de *f* nous conduit à une analyse de la file *f'* :
  - `<obtenir le 1er élément>` l'obtention d'un entier lu au clavier se fait simplement via `scanf`. La valeur lue est stockée dans la variable `x` : `scanf("%d", &x)`
  - `<obtenir l'élément suivant>` pas de distinction ici entre le premier élément et les suivants
  - `<élément != butée>` la butée est double dans notre cas : soit l'entier lu est 0, et nous avons la fin de la suite d'entiers lue au clavier, soit l'entier lu est plus petit que l'entier le précédent, et nous avons la fin de la sous-suite croissante. Nous utilisons une variable `x_prec` pour stocker l'entier précédemment lu. Le test s'écrit donc : `(x!=0)&&(x>=x_prec)`
  - `<traiter l'élément courant>`
    le traitement de la sous-suite d'entiers consiste simplement à en calculer la longueur. Pour cela, nous utilisons une variable `l` de type `int`. Le traitement consiste alors à incrémenter la valeur de `l`. Il faut par ailleurs modifier la valeur de `x_prec` pour que cette variable contienne toujours l'avant dernier élément de *f'* :

    ```
    l++;
    x_prec = x;
    ```

  - `<initialisations>`
    avant que nous ne lisions le moindre caractère de la file, la longueur de celle-ci est nulle, donc `l` est initialisée à 0. Il nous faut aussi initialiser `x_prec` qui sert à tester la croissance de la file. Avant que toute lecture de la file ne débute, il n'y a pas d'élément précedemment lu, la seule solution consiste donc à initialiser `x_prec` au plus petit entier possible afin de ne pas altérer le test de croissance de la file :

    ```
    l = 0;
    x_prec = INT_MIN;
    ```

    Notons qu'il faudra include la bibliothèque `limits.h` pour obtenit la définition de la constante `INT_MIN`
  - `<présentation du résultat>` le résultat de ce traitement est stocké dans la variable `l`, comme souhaité pour le traitement de *f*. Il n'y a donc pas de traitement supplémentaire à effectuer

  Ceci nous donne le programme suivant pour l'obtention du premier élément de *f* :

  ```
  l = 0;
  x_prec = INT_MIN;
  scanf("%d", &x);
  while ((x != 0) && (x >= x_prec)) {
    l++;
    x_prec = x;
    scanf("%d", &x);
  }
  ```

- Revenons maintenant au traitement de la file *f*. `<obtenir l'élément suivant>` diffère ici de l'obtention du premier élément. En effet, la butée de la file *f'* (la sous-suite croissante) est le premier élément de la sous-suite croissante précédente. En conséquence :
  - `<obtenir le 1er élément>` le premier élément est déjà stocké dans la variable `x` puisque la lecture

a été faite lors de la lecture de la butée de la sous-suite croissante précédente
- `<obtenir l'élément suivant>` par contre, ici, il faut bien lire l'élément suivant dans la file : `scanf("%d", &x)`

Le reste est identique au cas de l'obtention du 1er élément, ce qui nous donne le programme :

```
l = 0;
x_prec = INT_MIN;
while ((x != 0) && (x >= x_prec)) {
  l++;
  x_prec = x;
  scanf("%d", &x);
}
```

Au final, nous avons donc le programme suivant :

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

int main() {
  int l, l_max, x, x_prec;

  l_max = 0;
  /* obtenir 1er elmt */
  l = 0;
  x_prec = INT_MIN;
  scanf("%d", &x);
  while ((x != 0) && (x >= x_prec)) {
    l++;
    x_prec = x;
    scanf("%d", &x);
  }
  /**/
  while (l != 0) {
    if (l > l_max)
      l_max = l;
    /* obtenir elemt suivant */
    l = 0;
    x_prec = INT_MIN;
    while ((x != 0) && (x >= x_prec)) {
      l++;
      x_prec = x;
      scanf("%d", &x);
    }
    /**/
  }
  printf("longueur max=%d\n", l_max);

  return 0;
}
```

3 Files avec butée traitée

Il s'agit des suites finies d'éléments de même type dont le dernier élément (la *butée*) est distingué des autres, tous les éléments étant traités de la même façon (y compris la butée).

Il faut donc traiter tout élément avant de tester s'il s'agit de la butée, ce qui nous amène vers la structure de contrôle `do...while`. Nous allons donc écrire des programmes C ayant la structure suivante :

```
<initialisations>
do {
  <obtenir élément>
  <traiter élément>
}
while (<élément != butée>)
<présentation du résultat>
```

Il nous reste plus qu'à définir les différents éléments inconnus de cette structure de programme en fonction des files à traiter.

Exercice 13:

Comment est définie la file vide ?

Une file avec butée traitée contient au moins un élément : la butée, et celle-ci est traitée comme tous les autres éléments. Il y a donc au moins un élément traité dans toute file de ce type.

Exercice 14:

La file à traiter ici est une suite d'entiers terminée par 0. Nous pouvons maintenant chercher à compléter la structure de programme :

- `<obtenir élément>` les éléments de la file sont lus au clavier. Nous allons donc utiliser `scanf` pour les lire et nous les stockons dans une variable `x` de type `int`. L'obtention d'un élément est donc : `scanf("%d", &x)`
- `<traiter élément>`
  le traitement de la file consiste à en sommer les éléments. Nous allons utiliser une variable `s` de type `int` pour stocker la somme des éléments de la file traités jusqu'ici. Le traitement s'écrit donc : `s = s+x`
- `<élément != butée>` la butée de la file est 0, le test est donc : `x != 0` puisque `x` contient l'élément courant de la file
- `<initialisations>` nous devons initialiser la variable `s` qui accumule la somme des éléments de la file. Avant que le traitement de la file débute la somme de ses événements lus jusqu'ici est 0, donc nous initialisons : `s = 0`
- `<présentation du résultat>` consiste en l'affichage de la valeur calculée : `printf("somme=%d\n", s)`

Nous obtenons donc le programme :

```
#include <stdio.h>

int main() {
  int s, x;

  s = 0;
  do {
    scanf("%d", &x);
    s = s+x;
  }
  while (x != 0);
  printf("somme=%d\n", s);

  return 0;
}
```

Exercice 15:

La file considérée ici est la suite des monômes $x^i/i!$, $i \geq 0$ du développement limité de $exp(x)$ jusqu'au premier élément inférieur à 0.01. Le traitement à appliquer à cette file est la somme de ses éléments.

Nous pouvons maintenant compléter le schéma de programme :

- `<obtenir élément>` l'élément de rang $i \geq 0$ correspond au monôme $x^i/i!$. Il s'obtient depuis le monôme de rang $i-1$, $i \geq 1$ en le multipliant par x et en le divisant par i. L'élément de rang 0 est le germe de la séquence, défini à 1. Nous utilisons la variable `m` de type `float` pour stocker la valeur du monôme courant :

  ```
  m = m*x/i;
  i = i+1;
  ```

- `<traiter élément>`
  le traitement de l'élément consiste à l'ajouter à la somme des éléments traités jusqu'alors. Nous devons donc stocker celle-ci dans une variable `s` de type `float`. Le traitement s'écrit alors : `s = s+m`
- `<élément != butée>` la butée est le premier élément de la file qui est inférieur à 0.01. Ce test s'écrit donc : `m >= 0.01`
- `<initialisations>`
  la séquence des monômes est définie récursivement. Il faut donc l'initialiser avec la valeur du monôme de rang 0, c'est à dire initialiser `m` à 1. Il faut de même initialiser `i` à 1 car c'est le rang du premier monôme calculé. Enfin, il faut aussi initialiser `s` à 1 qui est la valeur du premier monôme.

  ```
  m = 1;
  i = 1;
  s = 1;
  ```

- `<présentation du résultat>` il s'agit simplement d'afficher la valeur de `s` au moyen de l'instruction : `printf("exp(%f)=%f\n", x, s)`

Ceci nous donne le programme suivant :

```
#include <stdio.h>

int main() {
  float x=3.5, s, m;
  int i;

  m = 1;
  i = 1;
  s = 1;
  do {
    m = m*x/i;
    i = i+1;
    s = s+m;
  }
  while (m >= 0.01);
  printf("exp(%f)=%f\n", x, s);

  return 0;
}
```

4 Commentaires

Les exercices présentés ici ont pour but de donner des moyens de construire systématiquement des programmes

de traitement de flots de données identiques.

L'existence des 3 types de files explique l'existence des trois structures de contrôles : `for`, `while` et `do...while`. Cette caractérisation simplifie la construction et l'écriture des programmes, bien qu'à chaque fois nous puissions écrire les programmes en utilisant une autre structure itérative que celle qui est naturellement adéquat. Il est en effet aisé de traduire toute boucle `for` en boucle `while`, et de même pour les boucles `do...while`.

Maintenant vous savez comment écrire un programme pour chaque type de file. En pratique, la première difficulté face à un traitement de file est d'identifier à quel type de file nous avons à faire. La réponse n'est parfois pas unique (la file du dernier exercice pourrait très bien être vue comme une file à butée non traitée). Ce TD offre donc un cadre de raisonnement, sans qu'il faille lui donner trop de rigidité. Il nous semble intéressant d'utiliser cette technique de construction des programmes dans la suite des TD.

Ce document a été traduit de L^AT_EX par HEVEA