TP #2 – APL2

13 novembre 2018
Erwan Kerrien

Rappel : si vous avez des questions sur ce TP ou sur le cours, n'hésitez pas à m'envoyer un mail à Erwan.Kerrien@inria.fr (je consulte plus rarement mon mail Erwan.Kerrien@univ-lorraine.fr).

Nous travaillons sous Linux. Je suppose dans chaque TP que vous avez créé un répertoire adéquat pour le module APL2 et/ou chaque TP et que vous y travaillez. Il est admis que vous savez le créer et que vous savez y aller. Je ferai référence à ce répertoire par `REP` mais vous pouvez le nommer comme vous voulez.

Le TP est à rendre pour demain matin (14 novembre) 8h : seules les questions notées sont à me rendre (voir les sections "Questions notées". Vous devez m'envoyer au minimum les fichiers C Vous pouvez, si vous le souhaitez et si vous le jugez utile, m'envoyer un texte de commentaires. Ne pas le faire ne vous exposera à aucun retrait de points. En revanche, ne pas commenter vos codes d'une manière ou d'une autre le fera. Ces fichiers sont à déposer sur arche (séparés ou sous forme d'archive zip). En cas de souci, vous pouvez les envoyer son mon email Erwan.Kerrien@inria.fr

1 Objectifs du TP

Ce TP a pour objectif de vous présenter deux notions du langage C, plus avancées que celles que nous avons vues la dernière fois. Ces notions sont les *structures* et les *pointeurs*.

Le travail final demandé consistera à appliquer ces notions pour lire une base de données (dans un format texte très simple) et stocker et manipuler les informations qu'elle contient.

2 Pointeurs

Un pointeur permet de stocker une adresse de début d'une zone mémoire. On peut donc accéder, en lecture et en écriture, au contenu de cette zone par le biais du pointeur. Encore faut-il pouvoir décoder et encoder les bits qui y sont stockés. C'est pour cela qu'un pointeur pointe vers un type donné : le type définit sa taille (nombre d'octets nécessaires pour stocker les valeurs de ce type), ainsi que l'encodage des valeurs du type (un entier signé n'est pas stocké comme un entier non signé, qui n'est pas stocké comme un réel...). Ainsi, on peut connaître la taille de la plage mémoire pointée, et en décoder correctement le contenu. Les pointeurs offrent dès lors des mécanismes assez bas niveau pour manipuler la mémoire.

2.1 Déclaration et assignation d'un pointeur

Un pointeur se déclare en indiquant le type pointé, suivi d'une étoile "*". Par exemple :

```
1  int *pi;
2  char *pc;
```

déclare deux variables de types pointeurs : `pi` de type `int*`, qui est donc un pointeur vers un `int` ; et `pc` de type `char*` qui est donc un pointeur vers un `char`. On notera que les espaces avant et/ou après le caractère "*" importent peu. On aura aussi bien déclarer ces variables comme :

```
1  int* pi;
2  char* pc;
```

ou encore

```
1  int * pi;
2  char * pc;
```

On préférera cependant la première manière indiquée car elle permet de mieux comprendre une déclaration comme

```
1  int *pi, i;
```

qui, bien qu'horrible pour la lisibilité du code, est valide et déclare une variable `pi` de type pointeur vers un `int`, ainsi qu'une variable `i` de type `int`. De même, pour déclarer deux variables pointeurs vers un même type, il faudra les faire précéder à chaque fois d'un `*`. Ainsi pour déclarer `pi1` et `pi2`, deux pointeurs vers des `int`, on écrira :

```
1 int *pi1, *pi2;
```

Les deux opérateurs de base associés aux pointeurs sont la prise d'adresse et de déréférencement. Ces deux opérateurs sont inverses l'un de l'autre :

- la prise d'adresse permet de récupérer l'adresse mémoire d'une variable (préalablement déclarée) : il s'applique donc à une variable d'un type donné et produit un pointeur sur ce type. L'opérateur de prise d'adresse se nomme *opérateur adresse-de* et se note par le caractère `&` qui précède le nom de la variable.
- le déréférencement permet de décoder les bits stockés dans la zone mémoire pointée par une variable de type pointeur : il s'applique donc à un pointeur et produit donc une valeur du type pointé. L'opérateur de déréférencement se nomme *indirection* et se note par le caractère `*` qui précède le nom de la variable.

Le code suivant donne un exemple d'application des opérateurs adresse-de et d'indirection (compilez et lancez-le) :

```
1  // fichier pointeur1.c
2  #include <stdio.h>
3
4  int main()
5  {
6    int i;
7    int *pi;
8
9    i=3;
10   pi = &i;
11   printf("(via la variable) Valeur=%d; adresse=%p\n",i,&i);
12   printf("(via le pointeur) Valeur=%d; adresse=%p\n",*pi,pi);
13
14   *pi = 4;
15   printf("(via la variable) Valeur=%d; adresse=%p\n",i,&i);
16   printf("(via le pointeur) Valeur=%d; adresse=%p\n",*pi,pi);
17
18   return 0;
19 }
```

L'adresse de la variable `i`, de type `int` est prise en ligne 9 et stockée dans la variable `pi` de type `int*`, soit un pointeur sur un `int`. Les lignes 10 et 11 montrent que les deux variables se rapportent à la même zone mémoire et aux mêmes valeurs, et que les deux opérateurs adresse-de et indirection sont inverses l'un de l'autre. En ligne 14, on modifie la zone pointée par `pi` pour y stocker la valeur 4 via le mécanisme d'indirection. Les lignes 15 et 16 montrent que la variable `i` a bien été modifiée. Par ailleurs, on remarquera que le format `%p` permet d'afficher une adresse (au format hexadécimal) stockée dans un pointeur.

Le couplage de ces deux opérateurs est surtout employé pour implanter le mécanisme de passage par variable que nous avons vu dans la fonction `sommeProduit` dans le premier TP.

Enfin, on notera que, puisqu'un pointeur correspond à une variable, il est possible de prendre son adresse, qui correspond alors à... un pointeur de pointeur. Un tel pointeur se déclare avec une double `*`. Par exemple :

```
1 int i;
2 int *pi;
3 int **ppi;
4 i=1;
5 pi = &i; // *pi == i == 1
6 ppi = &pi; // *ppi == pi && **ppi == i == 1
```

C'est ainsi qu'on peut stocker et manipuler des tableaux bi-dimensionnels (tableaux de tableaux, voire multi-dimensionnels, tableaux de tableaux ... de tableaux, voir plus bas l'exemple des tableaux de chaînes de caractères).

2.2 Allocation d'un pointeur

Un programme ne peut pas accéder à une zone mémoire qui n'est pas attribuée au processus qui l'exécute. Dans le cas précédent, l'indirection était possible car de la mémoire avait été implicitement allouée pour stocker la valeur de la

variable `i` [1]. Cette mémoire est automatiquement libérée en fin de vie de la variable (sortie du bloc de code dans lequel elle a été définie). Cette volatilité et ce manque de contrôle de la mémoire peut être gênant. Pour le pallier, un mécanisme d'allocation et de libération explicite de la mémoire existe [2] en C à travers les fonctions `malloc` (et associées : `calloc`, qui initialise en plus la zone allouée à 0 et `realloc` qui réalloue de la mémoire déjà en partie allouée), et `free` qui libère la mémoire ainsi allouée. Ces fonctions sont déclarées dans le fichier `stdlib.h` qu'il est dès lors obligatoire d'inclure dans le code. Le code suivant vous en donne un exemple similaire à celui de `pointeur1.c` ci-dessus :

```
1  // fichier pointeur2.c
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  int main()
6  {
7    int i;
8    int *pi;
9
10   pi = malloc(sizeof(int));
11
12   *pi=3;
13   i=*pi;
14   printf("(via la variable) Valeur=%d; adresse=%p\n",i,&i);
15   printf("(via le pointeur) Valeur=%d; adresse=%p\n",*pi,pi);
16
17   *pi = 4;
18   printf("(via la variable) Valeur=%d; adresse=%p\n",i,&i);
19   printf("(via le pointeur) Valeur=%d; adresse=%p\n",*pi,pi);
20
21   free(pi);
22   return 0;
23 }
```

Ici, de la mémoire est allouée pour un `int` et stockée dans la variable pointeur `pi` (ligne 10). On remarque ici que l'adresse de `i` et celle stockée dans `pi` sont différentes. Ainsi, malgré la ligne 13, si on change la valeur stockée sous `pi` en ligne 17, seule la zone mémoire pointée par `pi` est modifiée et la valeur de `i` reste inchangée. En effet, en ligne 13, on a simplement recopier la valeur stockée à l'adresse `pi` dans la variable `i` qui, elle, est stockée ailleurs comme le prouve son adresse différente de `pi`.

2.3 Pointeurs et tableaux

Nous avons vu qu'une variable de type tableau indique en fait le début de la zone mémoire dans laquelle sont stockées les éléments du tableau. On peut donc définir un tableau comme un pointeur. Nous allons travailler sur les chaînes de caractères qui, en C, sont des tableaux de caractères (type `char`), dont le dernier est le caractère nul (code ASCII=0).

Les éléments d'un tableau étant stockés de manière contiguë, nous avons besoin de `n * sizeof(Type)` pour stocker `n` éléments de type `Type`. Le code suivant en donne un exemple (compilez et lancez-le) :

```
1  // fichier string.c
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5
6  int main()
7  {
8    char S[]="Bonjour";
9    char *ps;
10   int lenS,i;
11
12   lenS = strlen(S);
13   ps=malloc((lenS +1) * sizeof(char));
14
```

1. Cette mémoire est allouée sur la pile.
2. Cette mémoire est allouée dans le tas.

```
15    for (i=0; i<lenS; i++)
16      ps[i] = S[i];
17    ps[lenS] = '\0';
18
19    printf("Test: %s -> %s\n",S,ps);
20
21    free(ps);
22    return 0;
23  }
```

En ligne 8, la variable `S` de type chaîne de caractères est déclarée et initialisée à la chaîne “Bonjour”. Cette ligne donne un exemple d’allocation automatique d’un tableau sur la pile : la déclaration complète devrait être

```
1  char S[8] = "Bonjour";
```

où la taille (fixe) du tableau est indiquée lors de la déclaration pour permettre une allocation automatique [3]. La version de C actuelle accepte d’omettre cette taille lorsque la variable est initialisée dans la déclaration : cette initialisation indique au compilateur la taille à réserver pour l’allocation automatique du tableau. C’est ce que nous utilisons dans notre code [4]. En ligne 9, la variable `ps` est déclarée comme un pointeur sur des `char`. En ligne 12, la variable `lenS` reçoit la longueur de la chaîne `S` (la fonction `strlen` est déclarée dans le fichier `string.h` inclus en ligne 4). Puis en ligne 13, de la mémoire est allouée pour stocker `lenS + 1 char`. L’adresse de cette mémoire est stockée dans le pointeur `ps`. On notera le +1 afin de stocker le caractère nul de fin de chaîne. En lignes 15 et 16, on boucle sur les caractères de la chaîne `S` et on les stocker un à un dans `ps`. On notera que la notation avec des crochets est valide pour `ps`. La chaîne stockée dans `ps` est correctement terminée par le caractère nul en ligne 17. La ligne 19 permet de vérifier que tout s’est bien passé. Enfin, la mémoire allouée dans `ps` est libérée en ligne 21.

2.4 Et les tableaux de chaînes de caractères ?

Chaque chaîne étant un tableau, c’est-à-dire un pointeur, un tableau de chaînes de caractères est un pointeur sur un pointeur, ce qu’on appelle un double pointeur. Le code suivant permet de créer un tableau de chaînes de caractères (compilez et lancez-le) :

```
1  // fichier tabstring.c
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5
6  int main()
7  {
8    char S1[]="Bonjour";
9    char S2[]="et";
10   char S3[]="bienvenue";
11
12   char **tabs;
13   tabs=malloc(3*sizeof(char*));
14   tabs[0] = malloc((strlen(S1)+1)*sizeof(char));
15   tabs[1] = malloc((strlen(S2)+1)*sizeof(char));
16   tabs[2] = malloc((strlen(S3)+1)*sizeof(char));
17
18   strcpy(tabs[0],S1);
19   strcpy(tabs[1],S2);
20   strcpy(tabs[2],S3);
21
22   int i;
23   printf("Test: ");
24   for (i=0; i<3; i++)
25     printf("%s ",tabs[i]);
26   printf(" !!!\n");
```

3. Vous noterez que `Bonjour` comporte 7 lettres mais qu’on alloue de la place pour 8 caractères pour inclure le `\0` final

4. Nous pouvons aussi le faire pour, par exemple un tableau d’entier. Dans ce cas, le tableau constant qui sert à l’initialisation est défini par une liste de nombres, séparés par des virgules et encadrée par des accolade. Par exemple : `int tabint[] = {1,2,3,4};`

```
27
28    for (i=0; i<3; i++) free(tabs[i]);
29    free(tabs);
30    return 0;
31  }
```

Nous commençons par déclarer trois chaînes de caractères `S1`, `S2` et `S3` qui sont initialisées (lignes 8 à 10). Ensuite la variable `tabs` est un double pointeur, soit un tableau de tableaux. Il faut donc faire une double couche d'allocations. En ligne 13 on commence par allouer de la place pour 3 pointeurs : chaque élément du tableau stockera l'adresse du tableau correspondant. Ensuite, chaque tableau est alloué en lignes 14 à 16. Notez que d'une part il serait illicite de modifier `tab[i]` (`i`=0,1 ou 2) sans allocation préalable de `tabs`. D'autre part, on alloue toujours un caractères supplémentaire pour le caractère nul final. On voit bien que `tabs[0]` par exemple est un pointeur, donc un tableau, et on pourra accéder aux caractères, par exemple le premier, par `tabs[0][0]` (Attention `tabs[0,0]` n'est pas valide). La fonction `strcpy` permet de recopier une chaîne de caractères dans une autre (exactement ce que faisaient les lignes 15 à 17 du code précédent `string1.c` !). On les emploie lignes 18 à 20. Ensuite on affiche le résultat. Notez l'emploi de la boucle. N'hésitez pas à modifier l'affichage pour voir ce que ça donne. Enfin, en miroir de la double allocation lignes 13 à 16, il y a une double couche de libération lignes 28 et 29 et on fera toujours attention à bien libérer dans l'ordre inverse des allocations (si vous libérez d'abord `tabs`, vous ne pourrez plus libérer `tabs[i]` puisque l'expression `tabs[i]` n'est plus valide dès lors que `tabs` a été libéré !).

3 Arguments de la ligne de commande

Lorsque vous lancez un programme en ligne de commande, vous tapez le nom de l'exécutable (par exemple `./tabstring` pour le dernier code). Vous pouvez en fait ajouter des arguments à cette ligne de commande et C offre un mécanisme pour les récupérer facilement. Imaginons par exemple qu'on veuille faire un additionneur de deux nombres. L'exécutable va s'appeler `add` et on aimerait lui donner en arguments les deux nombres à ajouter, par exemple la commande `./add 3 4` devrait afficher `7`.

Pour ce faire, on utilise une autre version de la fonction `main` qui prendra deux arguments :

```
1   int main(int argc, char **argv)
```

Le premier argument est `argc` de type `int` : c'est le nombre de mots sur la ligne de commande (3 pour la commande `./add 3 4`). Le deuxième argument `argv` est un tableau de chaînes de caractères, une par mot sur la ligne de commande. Le tableau `argv` est donc de longueur `argc` et le nom de l'exécutable est toujours stocké dans `argv[0]`. Le code suivant permet de tester ce mécanisme (compilez et lancez-le avec autant de mots que vous voulez à la suite de l'exécutable sur la ligne de commande) :

```
1  // fichier args.c
2  #include <stdio.h>
3
4  int main(int argc, char **argv)
5  {
6    printf("Nom de l'executable: %s\n", argv[0]);
7    if (argc == 1)
8      printf("Aucun argument\n");
9    else
10     {
11       int i;
12       for (i=1; i<argc; i++)
13         printf("Argument #%d: %s\n", i, argv[i]);
14     }
15   return 0;
16 }
```

Ainsi l'addition de deux entiers peut se coder comme :

```
1  // fichier add.c
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  int main(int argc, char **argv)
```

```
{
  int a=strtol(argv[1],0,0);
  int b=strtol(argv[2],0,0);
  printf("Resultat: %d\n", a+b);
  return 0;
}
```

On notera la fonction `strtol`, déclarée dans `stdlib.h` qui permet de transformer une chaîne de caractères (ici `argv[0]` et `argv[1]` en entiers. Gardez 0 pour les deux autres arguments (voir `man strtol` pour plus d'informations).

Exercice : Question notée

Ecrivez les programmes suivants :

1. Modifiez `add.c` de manière à ce qu'on puisse donner autant de nombres qu'on veut en arguments et qu'ils soient tous additionnés.
   `[Bonus]` Vous pourrez étendre le code pour bien gérer les erreurs d'entrée, par exemple si je rentre `./add 3 4 toto 5`, je voudrais que s'affiche une erreur. Voir les documentations `man strtol` et `man errno` pour cela.
2. Écrivez le programme `verlan` qui affiche les mots entrés en ligne de commande **après l'exécutable**, à l'envers. Par exemple `./verlan Bonjour et bienvenue` affichera `bienvenue et Bonjour`.
3. Écrivez le programme `verlanfull` qui affiche les mots dans l'ordre de `verlan`, mais en plus affiche leurs caractères en ordre inversé. Ainsi `./verlanfull Bonjour et bienvenue` affichera `eunevneib te ruojnoB`.

4 Fichiers

4.1 Entrée/sortie standard

Nous avons déjà vu la fonction `printf` qui permet d'écrire une chaîne de caractères sur l'écran, autrement dit ce qu'on appelle la *sortie standard*. Le premier argument est appelé *format* et comprend des séquences commençant par `%` qui se verront remplacées par la valeur des variables mises dans la suite des arguments de `printf` (voir le TP1 pour les différents formats, et la commande shell `man -s 3 printf` pour une documentation extensive).

Sa contrepartie est la fonction `scanf` qui permet de lire des valeurs sur l'entrée standard, c'est-à-dire entrées au clavier par l'utilisateur (un peu comme `input` en python). `scanf` prend les mêmes arguments que `printf`, à savoir une chaîne de caractères qui indique le format, puis une suite d'arguments qui sont chacun associés à une séquence commençant par `%`. La différence, est qu'ici la valeur de ces variables est lue dans la chaîne de caractères entrées au clavier. Du coup, ces variables doivent être passées par variable : autrement dit on indique un pointeur vers ces variables à la `scanf`.

Par exemple, testez le petit code suivant

```
// fichier input.c
#include <stdio.h>

int main()
{
    char c;
    int i;
    float f;
    char s[10];
    int ret;
    ret=scanf("Un char %c, un entier %d, un reel %f, une chaine %s et c'est tout.", &c, &i, &f, s);
    printf("Resultat (%d): %c // %d // %f // %s //\n", ret, c, i, f, s);
    return 0;
}
```

`scanf` renvoie le nombre d'arguments correctement lus (stocké dans `ret`). Testez avec différentes entrées au clavier et vous verrez que cette fonction est assez délicate à manipuler. À noter toutefois que la forme `%*` permet d'indiquer que le format doit contenir un certain élément, dont on indique le format, mais dont la valeur ne nous intéresse pas : cette marque de formatage n'aura donc pas de variable associée dans la liste d'arguments. Par exemple :

```
char pays[256];
```

```
float temp;
scanf("%s %*s %f\n", pays, &temp);
```

Lira bien une ligne qui contient une première chaîne de caractères (sans espace, ni tabulation), qui correspond au nom d'un pays, puis une deuxième chaîne de caractères (sans espace, ni tabulation), qui correspond à un nom de ville, mais qui ne nous intéresse pas, puis un réel qui indique une température. On remarquera l'allocation automatique de la variable `pays`, suffisamment grande pour accueillir un nom de pays (dont la longueur est a priori variable et dont on ne connaît pas la longueur maximale, il faut donc ici utiliser une borne haute raisonnable).j

4.2 Fichiers

Il existe les mêmes fonctions pour les fichiers. Elles ont pour noms `fprintf` et `fscanf`. Elles prennent les mêmes arguments que `printf` et `scanf`, sauf qu'il faut mettre en premier un nouvel argument : un pointeur vers un fichier. Une variable fichier est de type `FILE` : elle se crée par la fonction `fopen` qui prend en paramètre une chaîne de caractères (nom d'un fichier), ouvre ce fichier, et renvoie un pointeur vers un `FILE`. Le pointeur est libéré par un appel à `fclose` qui ferme également le fichier précédemment ouvert. Un fichier peut être ouvert en lecture et/ou en écriture. Afin de spécifier le mode d'ouverture, la fonction `fopen` prend un deuxième argument sous forme de chaîne de caractères : "r" pour ouvrir en lecture seule, "w" pour ouvrir en écriture (écrase le contenu du fichier, ou crée le fichier s'il n'existe pas), ou "a" pour ouvrir en écriture (se positionne en fin de fichier, ce qui permet donc d'en étendre le contenu). `fopen` renvoie le pointeur `NULL` en cas d'échec. Voir l'aide en ligne du shell par la commande `man -s 3 fopen`.

L'exemple suivant fait simplement une recopie d'un fichier, passé en argument du programme, dans le fichier passé en second argument du programme : on suppose juste ici que le premier fichier contient une liste de réels, ces réels étant recopiés, un par ligne, dans le fichier en sortie.

```
// fichier copie.c
#include <stdio.h>

int main(int argc, char *argv[])
{
   FILE *fichierIn, *fichierOut;
   float val;
   // verifie que deux arguments ont bien ete passes au programme
   if (argc != 3)
     {
       fprintf(stderr, "Erreur de syntaxe: copie <fichierIn> <fichierOut>\n");
       return 1;
     }

   // ouvre le premier fichier en lecture
   fichierIn = fopen(argv[1], "r");

   // test si tout s'est bien passe
   if (!fichierIn)
     {
       fprintf(stderr, "Impossible d'ouvrir le fichier %s en lecture\n", argv[1]);
       return 2;
     }

   // si on est la, c'est que tout va bien. On ouvre le fichier en ecriture
   fichierOut = fopen(argv[2],"w");
   // on test encore si tout est ok
   if (!fichierOut)
     {
       fprintf(stderr, "Impossible d'ouvrir le fichier %s en ecriture\n", argv[2]);
       return 3;
     }

   // realise la copie
   while (fscanf(fichierIn, "%f", &val) == 1)
     fprintf(fichierOut, "%f\n", val);

```

```
38      // ferme les fichiers ouverts
39      fclose(fichierIn);
40      fclose(fichierOut);
41
42      return 0;
43  }
```

On remarquera l'usage de `stderr`, lignes 11, 21 et 30 : il s'agit d'un fichier ouvert par défaut vers la sortie d'erreur standard. Il existe deux autres fichiers ouverts par défaut : `stdout` pour la sortie standard et `stdin` pour l'entrée standard, ce qui fait que `printf(...)` est équivalent à `fprintf(stdout, ...)` et `scanf(...)` est équivalent à `fscanf(stdin, ...)`. On remarquera par ailleurs l'usage de valeurs non nulles renvoyées par le programme en cas d'erreur, ainsi que l'usage des paramètres `argc` et `argv` dans la fonction `main`. La boucle `while` en lignes 35 et 36 permet de réaliser la copie, celle-ci s'arrêtant quand on ne peut plus lire de réel. Une autre manière de tester si nous sommes arrivés à la fin d'un fichier serait d'appeler la fonction `feof(FILE)` qui renvoie 0 tant que nous ne sommes pas à la fin du fichier. Pour plus d'informations sur la lecture et écriture dans un fichier, je vous invite à aller aussi vous renseigner sur les fonctions `fwrite` et `fread`, et ne pas vous arrêter là car bien d'autres possibilités existent que nous n'avons pas le temps de voir ici.

Exercice : Question notée

Ecrivez un programme, que vous nommerez `readDB`, correspondant au fichier source `readDB.c`, qui prend en entrée un fichier contenant sur chaque ligne un nom, suivi d'un prénom, puis un âge et enfin un genre (M ou F). Le programme renverra :

1. Le nombre de personnes (autrement dit le nombre de lignes)
2. Le nombre d'hommes et de femmes
3. L'âge moyen de ces personnes

Par exemple si je donne le fichier suivant en entrée :

```
Enfant Hélène 44 F
Enfant Ludivine 47 F
Flaille Abdel 19 M
Flaille Akim 23 M
Flaille Yves 21 M
Neymar Jean 24 M
Titegoute Justine 78 F
Yapudebiairedenlefrigo Robin 67 M
```

Le programme répondra :

```
Le fichier contient 8 noms de personnes, dont 5 hommes et 3 femmes, avec un âge moyen de 40.375 ans.
```