



Votre numéro d'anonymat :

--	--	--

Programmation et structures de données en C– LU2IN018

Examen du 16 juin 2022

1 heure 30

Aucun document n'est autorisé.

Les calculatrices, baladeurs et autres appareils électroniques sont interdits. Les téléphones mobiles doivent être éteints et rangés dans les sacs. Le barème sur 51 points (11 questions) n'a qu'une valeur indicative.

Les appels à `malloc` seront supposés réussir. Il ne sera pas nécessaire de vérifier leur valeur de retour. De la même manière, l'ouverture d'un fichier sera supposée réussir. Il ne sera pas nécessaire de vérifier que c'est bien le cas.

Le mémento qui vous a été distribué est reproduit à la fin de l'énoncé.

L'ensemble des structures et prototypes de fonctions est également rappelé à la fin de l'énoncé, sur une page détachable.

Chasseur de tête

1 Gestion des compétences

Pour de nombreuses entreprises, il est critique de trouver la personne ayant les bonnes compétences. Pour les aider à faire ce choix, vous allez écrire du code permettant de gérer un ensemble d'individus et de trouver, dans cet ensemble, celles ou ceux qui ont les compétences attendues. Les compétences seront représentées sous la forme d'une chaîne de caractères stockée dans une liste chaînée. Les personnes seront représentées par leur nom et prénom ainsi que par la liste de leurs compétences. Les personnes seront également stockées dans une liste chaînée. Structures :

```
typedef struct _elt_comp {
    char *comp;
    struct _elt_comp *suiv;
} Elt_Comp;
```

```
typedef struct _pers {
    char *nom;
    char *prenom;
    Elt_Comp *lcomp;
} Pers;
```

```
typedef struct _elt_pers {
    Pers *pers;
    struct _elt_pers *suiv;
} Elt_Pers;
```

Question 1 (3 points)

Écrivez la fonction de création d'une personne. Le nom et le prénom doivent être dupliqués (i.e. une nouvelle zone mémoire doit être allouée), la liste de compétences doit être initialisée à une liste vide (NULL).

Prototype :

```
Pers *creer_pers(const char *nom, const char *prenom);
```

Solution:

```
Pers *creer_pers(const char *nom, const char *prenom) {
    Pers *pers = (Pers *)malloc(sizeof(Pers));
    pers->nom = strdup(nom);
    pers->prenom = strdup(prenom);
    pers->lcomp=NULL;
    return pers;
}
```

Question 2 (3 points)

Écrivez la fonction d'ajout d'une compétence à une personne. Vous utiliserez une fonction d'ajout de compétence à une liste de compétences que vous écrirez également. Prototypes :

```
void ajouter_comp_pers(Pers *pers, const char *comp);
```

```
Elt_Comp *ajouter_comp(Elt_Comp *list_comp, const char *comp);
```

La fonction d'ajout d'une compétence à une liste de compétences (`ajouter_comp`) prend en argument la liste initiale et renvoie la liste dans laquelle la compétence `comp` a été ajoutée. L'ordre des compétences dans la liste n'ayant pas d'importance, vous choisisrez d'effectuer l'insertion à l'endroit le plus pertinent afin de minimiser les opérations à effectuer lors de l'ajout d'une nouvelle compétence (vous justifierez votre réponse). La chaîne de caractères décrivant la compétence (`comp`) doit être dupliquée.

Solution: C'est l'ajout en tête qui est clairement le plus simple et le plus efficace car le nombre d'opérations à réaliser est indépendant de la taille de la liste.

```
void ajouter_comp_pers(Pers *pers, const char *comp) {
    pers->lcomp = ajouter_comp(pers->lcomp, comp);
}
```

```
Elt_Comp *ajouter_comp(Elt_Comp *lcomp, const char *comp) {
    Elt_Comp *ecomph=(Elt_Comp *)malloc(sizeof(Elt_Comp));
    ecomph->comp=strdup(comp);
    ecomph->suiv=lcomp;
    return ecomph;
}
```

Question 3 (9 points)

Écrivez les fonctions d'écriture d'une personne dans un fichier. Prototypes :

```
void ecrire_pers(const char *nom_fichier, Elt_Pers *list_pers);

void ecrire_pers_rec(FILE *f, Elt_Pers *list_pers);

void ecrire_comp_rec(FILE *f, Elt_Comp *list_comp);
```

ecrire_pers ouvre le fichier de nom nom_fichier et fait ensuite appel à la fonction récursive d'écriture d'une liste de personnes dans un fichier déjà ouvert (fonction ecrire_pers_rec). La fonction ecrire_comp_rec devra écrire une liste de compétences (toujours dans le fichier ouvert).

Le nom d'une personne sera écrit sur une ligne, puis, sur la ligne suivante sera écrit son prénom. La ligne suivante sera un '<<<', qui marquera le début des compétences. Chaque compétence sera écrite sur une seule ligne. Un '>>>' marquera la fin de la liste de compétences.

Exemple :

```
Batty
Roy
<<<
perspicace
fort
intelligent
>>>
```

Solution:

```
void ecrire_comp_rec(FILE *f, Elt_Comp *list_comp) {
    while(list_comp) {
        fprintf(f, "%s\n", list_comp->comp);
        list_comp=list_comp->suiv;
    }
}

void ecrire_pers_rec(FILE *f, Elt_Pers *list_pers) {
    while(list_pers) {
        fprintf(f, "%s\n", list_pers->pers->nom);
        fprintf(f, "%s\n", list_pers->pers->prenom);
        fprintf(f, "<<<\n");
        ecrire_comp_rec(f, list_pers->pers->lcomp);
        fprintf(f, ">>>\n");
        list_pers=list_pers->suiv;
    }
}

void ecrire_pers(const char *nom_fichier, Elt_Pers *list_pers) {
    FILE *f=fopen(nom_fichier, "w");
    ecrire_pers_rec(f, list_pers);
    fclose(f);
}
```

Question 4 (3 points)

Écrivez la fonction permettant de libérer toute la mémoire associée à une liste de compétences. Prototype :

```
void liberer_liste_comp(Elt_Comp *list_comp);
```

Solution:

```
void liberer_liste_comp(Elt_Comp *list_comp) {
    while(list_comp) {
        Elt_Comp *suiv=list_comp->suiv;
        free(list_comp->comp);
        free(list_comp);
        list_comp=suiv;
    }
}
```

Question 5 (6 points)

Écrivez une fonction qui, à partir d'une liste de personnes et d'une compétence donnée crée deux nouvelles listes de personnes : celles ayant une compétence donnée et celles qui ne l'ont pas. Prototype :

```
void sel_pers(Elt_Pers *list_pers, char *comp, Elt_Pers **pl_avec,
               Elt_Pers **pl_sans);
```

La fonction devra parcourir *list_pers* (qui ne devra pas être modifiée) et ajouter toutes les personnes ayant la compétence *comp* à la liste indiquée par *pl_avec*. Les personnes qui ne l'ont pas doivent être ajoutées à *pl_sans*. *pl_avec* et *pl_sans* sont des listes transmises par pointeurs pour pouvoir les modifier. Lors de l'ajout d'une personne à une de ces listes, un nouvel élément de liste de personne devra être créé, mais la personne ne sera pas dupliquée. On limitera les parcours de listes au strict nécessaire.

Solution:

```
void sel_pers(Elt_Pers *list_pers, char *comp, Elt_Pers **pl_avec,
               Elt_Pers **pl_sans) {
    int trouve;
    while(list_pers) {
        // faut-il inclure cette personne ?
        Elt_Comp *lcp=list_pers->pers->lcomp;
        trouve=0;
        while(lcp!=NULL) {
            if (strcmp(lcp->comp, comp)==0) {
                trouve=1;
                break;
            }
            lcp=lcp->suiv;
        }
        Elt_Pers *nlpers=(Elt_Pers*)malloc(sizeof(Elt_Pers));
        nlpers->pers=list_pers->pers;
        if (trouve==1) {
            nlpers->suiv=*pl_avec;
            *pl_avec=nlpers;
```

```

    }
else {
    nlpers->suiv=*pl_sans;
    *pl_sans=nlpers;
}
list_pers=list_pers->suiv;
}
}

```

2 Choix des bonnes personnes avec un arbre binaire de décision

La sélection des personnes sur la base d'une liste ne suffit pas lorsque la base de personnes est très grande. Pour aider les recruteurs à affiner leur recherche, vous allez écrire des fonctions s'appuyant sur des arbres de décision. Chaque noeud de l'arbre contient une liste de personnes qui correspondent aux personnes ayant validé tous les choix réalisés avant d'arriver à ce noeud. Un noeud contient également une chaîne de caractères qui permettra de distinguer les deux sous-arbres : le sous-arbre `avec_comp` contient les personnes ayant la compétence `comp`, le sous-arbre `sans_comp` contient les personnes n'ayant pas cette compétence.

Structure :

```

typedef struct _abd {
    Elt_Pers *list_pers;
    char *comp;
    struct _abd *avec_comp;
    struct _abd *sans_comp;
} ABD;

```

Question 6 (6 points)

Écrivez la fonction récursive de construction d'un arbre de décision. Prototype :

```
ABD *construire_ABD(Elt_Pers *list_pers, Elt_Comp *comp_a_ignorer);
```

Si la liste est vide, la fonction renvoie NULL, sinon, elle crée un noeud contenant la liste de personnes transmise en argument (sans la copier). Ce noeud contiendra aussi la compétence selon laquelle la liste de personnes sera divisée en deux dans les deux sous-arbres (celles ayant la compétence et celles ne l'ayant pas).

Pour que les listes de personnes ne soient pas segmentées plusieurs fois selon la même compétence, la compétence à chaque noeud sera choisie par la fonction `choix_competence`, dont le prototype est le suivant (cette fonction est fournie, elle ne sera pas à écrire) :

```
char *choix_competence(Elt_Pers *list_pers, Elt_Comp *comp_a_ignorer);
;
```

Elle renvoie une compétence présente chez au moins une personne mais absente de la liste de compétences `comp_a_ignorer`. Si elle ne trouve pas de compétence, elle renvoie NULL. Si cette fonction renvoie NULL, ou si la liste de personnes est de taille 1, `construire_ABD` renverra un noeud dont les champs `comp`, `avec_comp` et `sans_comp` sont initialisés à NULL (et le champ `list_pers` initialisé à l'argument `list_pers`). Dans les autres cas, il faut créer un noeud contenant la liste des

personnes et la nouvelle compétence, puis segmenter la liste des personnes avec cette compétence et construire récursivement les sous-arbres contenant les personnes avec et sans cette compétence.

Solution:

```

ABD *construire_ABD(Elt_Pers *list_pers, Elt_Comp *comp_a_ignorer) {
    // Construction d'un arbre binaire de décision
    // à chaque niveau de l'arbre, on propose la compétence
    // qui sépare le mieux l'ensemble de personnes.

    if (list_pers==NULL) {
        return NULL;
    }
    // Choix de la compétence pertinente
    char *comp = choix_competence(list_pers, comp_a_ignorer);
    if ((list_pers->suiv==NULL) || (comp==NULL)) {
        printf("Comp_est_NULL_ou_la_liste_est_réduite_à_1_personne_!\n"
               );
        ABD *abd=(ABD *)malloc(sizeof(ABD));
        abd->comp=NULL;
        abd->list_pers = list_pers;
        abd->avec_comp = NULL;
        abd->sans_comp = NULL;
        return abd;
    }
    Elt_Comp *nevite=ajouter_comp(comp_a_ignorer, comp);

    // segmentation de la liste
    Elt_Pers *l_avec=NULL;
    Elt_Pers *l_sans=NULL;
    sel_pers(list_pers,comp, &l_avec, &l_sans);

    ABD *abd=(ABD *)malloc(sizeof(ABD));
    abd->comp=strdup(comp);
    abd->list_pers = list_pers;
    abd->avec_comp = construire_ABD(l_avec, nevite);
    abd->sans_comp = construire_ABD(l_sans, nevite);
    free(nevite->comp);
    free(nevite);
    return abd;
}

```

Ceux qui pensent à faire la libération de la compétence ajoutée à la liste des compétences à effacer peuvent avoir un bonus !

Question 7 (3 points)

Écrivez la fonction d'utilisation de l'arbre de décision. Prototype :

```
void trouver_personnes(ABD *abd);
```

Si on est sur une feuille, la fonction affiche les noms de la liste de personnes portée par le noeud. Vous pourrez utiliser pour cela la fonction `afficher_noms` (vous n'aurez pas à l'écrire) :

```
void afficher_noms(Elt_Pers *list_pers);
```

Si le noeud n'est pas une feuille, la fonction demandera à l'utilisateur s'il souhaite trouver des gens ayant la compétence portée par le noeud ou non. L'utilisateur saisira au clavier la réponse (tout caractère qui n'est pas 'o' sera considéré comme 'n') et la recherche continuera avec le sous-arbre approprié. Vous pourrez écrire la fonction de façon itérative ou récursive.

Exemple de message affiché à l'écran si le champ `comp` du noeud en cours pointe sur la chaîne "programmeur" :

Voulez-vous des propositions ayant la compétence "programmeur" ? (o/n)

Solution:

```
void trouver_personnes(ABD *abd) {
    if(abd) {
        if ((abd->avec_comp==NULL) && (abd->sans_comp==NULL)) {
            printf("feuille...\n");
            afficher_noms(abd->list_pers);
            return;
        }

        printf("Voulez-vous des propositions ayant la compétence %s ? (o/n)", abd->comp);
        char clu;
        scanf("%c", &clu);
        if (clu=='o')
            trouver_personnes(abd->avec_comp);
        else
            trouver_personnes(abd->sans_comp);
    }
}
```

3 Main

Question 8 (6 points)

Ecrivez un main qui crée 4 personnes et qui les ajoute dans une liste de personnes 1p (vous pourrez ne détailler les instructions que d'une seule personne). La fonction devra ensuite écrire la liste dans le fichier "base_pers.txt", puis construire un arbre binaire et faire appel à la fonction d'utilisation de l'arbre. Pour finir, vous prendrez soin de libérer la mémoire allouée. Pour cela vous pourrez utiliser les fonctions (que vous n'aurez pas à écrire) :

```
void liberer_abd(ABD *abd);
```

```
void liberer_pers(Elt_Pers *list_pers, int liberer_p);
```

Les personnes et listes n'étant pas systématiquement dupliquées, la libération de la mémoire occupée par le programme peut nécessiter plusieurs variantes de la fonction de libération d'une liste de personnes. Pour toutes les gérer dans une fonction, `liberer_pers` prend un argument `liberer_p`. Il est utilisé de la façon suivante :

- s'il vaut 0, toute la mémoire occupée est libérée (éléments et personnes) ;
- s'il vaut 1, seuls les éléments sont libérés ;
- s'il vaut 2, seules les personnes sont libérées.

La fonction de libération d'un arbre libère les noeuds, les éléments de ses listes de personnes, mais ne libère pas les personnes. Ces fonctions sont fournies et ne sont donc pas à écrire.

Solution:

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

#include "cvtheque.h"

int main(void) {
    Pers *p1=creer_pers("Deckard", "Rick");
    ajouter_comp_pers(p1, "observateur");
    ajouter_comp_pers(p1, "rapide");
    ajouter_comp_pers(p1, "perspicace");

    Pers *p2=creer_pers("Tyrell", "Eldon");
    ajouter_comp_pers(p2, "intelligent");
    ajouter_comp_pers(p2, "programmeur");
    ajouter_comp_pers(p2, "perspicace");

    Pers *p3=creer_pers("Batty", "Roy");
    ajouter_comp_pers(p3, "intelligent");
    ajouter_comp_pers(p3, "fort");
    ajouter_comp_pers(p3, "agile");

    Pers *p4=creer_pers("Kowalsky", "Leon");
    ajouter_comp_pers(p4, "combatif");
    ajouter_comp_pers(p4, "fort");
    ajouter_comp_pers(p4, "violent");

    Elt_Pers *lp=NULL;
    lp=ajouter_pers(lp, p1);
    lp=ajouter_pers(lp, p2);
    lp=ajouter_pers(lp, p3);
    lp=ajouter_pers(lp, p4);

    ecrire_pers("base_pers.txt", lp);

    printf("\n====_Arbres_de_décision_===\n");
}
```

```

ABD *abd=construire_ABD(lp, NULL);

trouver_personnes(abd);

liberer_pers(lp,2);
liberer_abd(abd);

return 0;
}

```

4 Débogage

Question 9 (6 points)

Une autre personne a écrit le code de lecture d'une liste de personnes avec le main suivant pour la tester.

Fichier cvtheque.c (en partie) :

```

264 Elt_Pers *lire_pers(const char *nom_fichier) {
265     FILE *f=fopen(nom_fichier, "r");
266     char buffer[256];
267     char *nom;
268     char *prenom;
269     Elt_Pers *liste_lue=NULL;
270     /* boucle de lecture d'une personne */
271     while(1) {
272         if (fgets(buffer, 256, f)==NULL)
273             break;
274         /* pour enlever le '\n' final */
275         buffer[strlen(buffer)-1]='\0';
276         nom=strdup(buffer);
277
278         fgets(buffer, 256, f);
279         /* pour enlever le '\n' final */
280         buffer[strlen(buffer)-1]='\0';
281         prenom=strdup(buffer);
282
283         /* lecture du '<<<' initial */
284         fgets(buffer, 256, f);
285
286         Pers *p_lue=creer_pers(nom, prenom);
287
288         fgets(buffer, 256, f);
289         /* pour enlever le '\n' final */
290         buffer[strlen(buffer)-1]='\0';
291
292         while(strcmp(buffer, ">>>") !=0) {

```

```

293     ajouter_comp_pers(p_lue,buffer);
294     fgets(buffer, 256, f);
295     /* pour enlever le '\n' final */
296     buffer[strlen(buffer)-1]='\0';
297 }
298
299     liste_lue = ajouter_pers(liste_lue,p_lue);
300 }
301 return liste_lue;
302 }
```

Fichier main_cv.c :

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 #include "cvtheque.h"
6
7
8 int main(void) {
9
10    Elt_Pers *l_lue=lire_pers("base_pers.txt");
11    printf("Liste_lue!_\n");
12    liberer_pers(l_lue,0);
13
14    return 0;
15 }
```

En exécutant le programme avec valgrind, on obtient les messages suivants :

```

$ valgrind --leak-check=full ./main_cv
==43560== Memcheck, a memory error detector
==43560== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==43560== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==43560== Command: ./main_cv
==43560==
Liste lue !
==43560==
==43560== HEAP SUMMARY:
==43560==     in use at exit: 522 bytes in 9 blocks
==43560== total heap usage: 51 allocs, 42 frees, 10,250 bytes allocated
==43560==
==43560== 20 bytes in 4 blocks are definitely lost in loss record 1 of 3
==43560==     at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_mem
==43560==     by 0x48F038E: strdup (strdup.c:42)
==43560==     by 0x109F6D: lire_pers (cvtheque.c:281)
==43560==     by 0x109340: main (main_cv.c:10)
==43560==
==43560== 30 bytes in 4 blocks are definitely lost in loss record 2 of 3
==43560==     at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_mem
==43560==     by 0x48F038E: strdup (strdup.c:42)
```

```

==43560==      by 0x109F21: lire_pers (cvtheque.c:276)
==43560==      by 0x109340: main (main_cv.c:10)
==43560==
==43560== LEAK SUMMARY:
==43560==      definitely lost: 50 bytes in 8 blocks
==43560==      indirectly lost: 0 bytes in 0 blocks
==43560==      possibly lost: 0 bytes in 0 blocks
==43560==      still reachable: 472 bytes in 1 blocks
==43560==              suppressed: 0 bytes in 0 blocks
==43560== Reachable blocks (those to which a pointer was found) are not shown.
==43560== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==43560==
==43560== For lists of detected and suppressed errors, rerun with: -s
==43560== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)

```

Qu'est-ce que cela signifie ? Est-ce normal ? Est-ce que cela indique un problème ? Si c'est le cas, de quel type de problème s'agit-il et comment le corriger ? Vous justifierez votre réponse.

Solution: Il s'agit d'un problème de fuite mémoire. Il y a 50 octets de perdus (definitely lost : ...). Ces octets ont été perdus lors des strdup des noms et prénoms (respectivement aux lignes 276 et 281). Pour résoudre le problème, il faut passer à des chaînes allouées statiquement (tableaux de char au lieu de char *) et faire des strcpy, ou bien libérer la mémoire après la création d'une personne (les noms et prénoms sont copiés dans la fonction).

5 Listes génériques

Les utilisateurs ont fait remonter le besoin d'associer une valeur à la compétence, pour distinguer les débutants des experts. Nous allons pour cela utiliser un niveau représenté sous la forme d'un entier. Cela donne la structure suivante :

```

typedef struct _comp_niv {
    char *comp;
    int niveau;
} Comp_Niv;

```

On profite de cette demande de modification pour passer à des listes chaînées génériques. Pour rappel, pour utiliser ces listes il suffit de définir les fonctions permettant de manipuler les données. Ces fonctions sont ensuite transmises à la liste sous la forme de pointeurs de fonctions dans une structure de donnée dédiée. Grâce à cela, les fonctions de manipulations des listes n'ont plus besoin d'être modifiées si on change le type des données.

Question 10 (3 points)

Écrivez la fonction de duplication adaptée pour la structure Comp_Niv. Prototype :

```
void *dupliquer_comp_niv(const void *src);
```

Pour rappel, cette fonction prend en argument un pointeur sur la donnée que l'on souhaite manipuler (de type Comp_Niv donc) sous la forme d'un pointeur générique. Elle alloue une nouvelle donnée et recopie la donnée pointée par src dans celle-ci. La chaîne de caractères stockée dans le champ comp devra être dupliquée.

Solution:

```
void *dupliquer_comp_niv(const void *src) {
    const Comp_Niv *cn_src=(const Comp_Niv *)src;
    Comp_Niv *dst=(Comp_Niv *)malloc(sizeof(Comp_Niv));
    dst->comp=strdup(cn_src->comp);
    dst->niveau=cn_src->niveau;
    return (void *)dst;
}
```

Question 11 (3 points)

Écrivez la fonction de libération de la mémoire adaptée pour la structure Comp_Niv. Prototype :

```
void detruire_comp_niv(void *data);
```

Solution:

```
void detruire_comp_niv(void *data) {
    Comp_Niv *cn_data=(Comp_Niv *)data;
    free(cn_data->comp);
    free(cn_data);
}
```

Mémento de l'UE LU2IN018

Ce document a pour vocation de présenter un bref descriptif des fonctions offertes par les bibliothèques standards et qui sont susceptibles d'être utilisées dans l'UE.

Entrées - sorties

Prototypes disponibles dans `stdio.h`.

Entrées, sorties formatées

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

La fonction `printf` écrit sur la sortie standard (par défaut le terminal), un message correspondant au texte spécifié dans la chaîne `format`. Le texte affiché peut contenir des portions *variables* spécifiées par des codes de conversion précédés par le caractère `%`. Les arguments suivant `format` doivent correspondre (ordre et type) aux codes de conversion présents. Voici quelques codes de conversion courants :

- `%d` : entier
- `%c` : caractère
- `%s` : chaîne de caractères
- `%f` : nombre réel

`printf` renvoie le nombre de caractères imprimés et la chaîne de format peut contenir des codes de contrôle permettant le formatage comme `\n` pour forcer le passage à la ligne et `\t` pour insérer une tabulation.

```
int scanf (const char *format, ...);
int sscanf(const char *entree, const char *format, ...);
```

Les fonctions `scanf` et `sscanf` permettent de saisir et analyser un texte saisi sur l'entrée standard, par défaut le clavier (`scanf`) ou depuis une chaîne de caractères passée en argument (`sscanf`). Le texte devra respecter le `format` spécifié et les arguments suivants doivent correspondre à des pointeurs sur des variables de type appropriés. Les codes de conversion sont identiques à ceux de `printf`.

Entrées, sorties caractères

```
int getchar(void);
```

Lit un caractère dans le flux d'entrée standard (par défaut le clavier). La fonction retourne le code `EOF` en cas d'erreur, le caractère lu dans le cas contraire.

```
int putchar(int c);
```

Affiche un caractère dans le flux de sortie standard (par défaut le terminal). La fonction retourne le code `EOF` en cas d'erreur, le caractère lu dans le cas contraire.

```
int puts(const char *s);
```

Affiche une chaîne de caractères dans le terminal et passe à la ligne, renvoie `EOF` en cas d'erreur.

Fichiers

Prototypes disponibles dans `stdio.h`.

```
FILE *fopen (const char *path, const char *mode);
```

Ouvre un fichier dont le chemin est spécifié par la chaîne `path` et retourne un pointeur de type `FILE *` (`NULL` en cas d'échec). L'argument `mode` permet de spécifier le type d'accès à réaliser sur le fichier :

- `[r]` pour un accès en lecture,
- `[w]` pour un accès en écriture et le contenu précédent du fichier est écrasé,
- `[a]` pour un accès en écriture, le contenu du fichier est préservé et les écritures sont effectuées à la suite des contenus déjà présents.

En cas d'erreur la fonction retourne le code `NULL` sinon un pointeur vers le fichier ouvert.

```
int fclose(FILE *fp);
```

Cette fonction provoque la fermeture du fichier pointé par `fp`. En cas d'erreur la fonction retourne le code `EOF` sinon 0.

```
int fprintf(FILE *stream, const char *format, ...);
```

Identique à `printf` mais l'argument `stream` permet de spécifier le flux de sortie.

```
int fscanf(FILE *stream, const char *format, ...);
```

Identique à `scanf` mais l'argument `stream` permet de spécifier le flux d'entrée.

```
int fputc(int c, FILE *stream);
```

Identique à `putchar` mais l'argument `stream` permet de spécifier le flux de sortie.

```
int fputs(const char *s, FILE *stream);
```

Identique à `puts` mais l'argument `stream` permet de spécifier le flux de sortie et il n'y a pas d'ajout de passage à la ligne.

```
int getc(FILE *stream);
```

Identique à `getchar` mais l'argument `stream` permet de spécifier le flux d'entrée.

```
char *fgets(char *s, int size, FILE *stream);
```

Lit au plus `size-1` octets dans le flux `stream`. La lecture s'arrête dès qu'un passage à la ligne est rencontré. Les octets lus sont stockés dans `s`. La fonction retourne `s` en cas de succès et `NULL` en cas d'erreur.

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

Lecture binaire de **nmemb** éléments de **size** octets dans le fichier **stream**. Les données lues sont stockées en mémoire à partir de l'adresse **ptr**. La fonction retourne le nombre d'éléments effectivement lus.

```
size_t fwrite(const void *ptr, size_t size, size_t nmemb,
             FILE *stream);
```

Écriture de **nmemb** éléments de **size** octets dans le fichier **stream**. Les données à écrire sont lues en mémoire à partir de l'adresse **ptr**. La fonction retourne le nombre d'éléments effectivement écrits.

Chaînes de caractères

Prototypes disponibles dans **string.h**.

Une chaîne de caractères correspond à un tableau de caractère et doit contenir un marqueur de fin \0.

```
size_t strlen(const char *s);
```

Renvoie la longueur d'une chaîne de caractères (marqueur de fin \0 non compris).

```
int strcmp(const char *s1, const char *s2);
```

```
int strncmp(const char *s1, const char *s2, size_t n);
```

Comparaison entre chaînes de caractères éventuellement limité aux **n** premiers caractères. La valeur renournée est :

— 0 si les deux chaînes sont identiques,

— négative si **s1** précède **s2** dans l'ordre lexicographique (généralisation de l'ordre alphabétique),

— positive sinon.

```
char *strcpy(char *dest, const char *src);
```

```
char *strncpy(char *dest, const char *src, size_t n);
```

Copie le contenu de la chaîne **src** dans la chaîne **dest** (marqueur de fin \0 compris). La chaîne **dest** doit avoir précédemment été allouée. La copie peut être limitée à **n** caractères et la valeur renournée correspond au pointeur de destination **dest**.

```
void *memcpy(void *dest, const void *src, size_t n);
```

Copie **n** octets à partir de l'adresse contenue dans le pointeur **src** vers l'adresse stockée dans **dest**. **dest** doit pointer vers une zone mémoire préalablement allouée et de taille suffisante. **memcpy** renvoie la valeur de **dest**.

```
size_t strlen(const char *s);
```

Retourne le nombre de caractères de la chaîne **s** (marqueur de fin \0 non compris).

```
char * strdup(const char *s);
```

Cette fonction permet de dupliquer une chaîne de caractères, elle retourne un pointeur vers la chaîne nouvellement allouée. La nouvelle chaîne pourra être libérée avec la fonction **free**.

```
char * strcat(char *dest, const char *src);
char * strncat(char *dest, const char *src, size_t n);
```

Ajoute la chaîne **src** à la suite de la chaîne **dst**. La chaîne **dest** devra avoir été allouée et être de taille suffisante. La fonction retourne **dest**.

```
char * strstr(const char *haystack, const char *needle);
```

La fonction renvoie un pointeur sur la première occurrence de la sous-chaîne **needle** rencontrée dans la chaîne **haystack**. Si la chaîne recherchée n'est pas présente, la fonction retourne NULL.

Conversion de chaînes de caractères

Prototypes disponibles dans **stdlib.h**.

```
int atoi(const char *nptr);
```

La fonction convertit le début de la chaîne pointée par **nptr** en un entier de type **int**.

```
double atof(const char *nptr);
```

Cette fonction convertit le début de la chaîne pointée par **nptr** en un **double**.

```
long int strtol(const char *nptr, char **endptr, int base);
```

Convertit le début de la chaîne **nptr** en un entier long. L'interprétation tient compte de la **base** et la variable pointée par **endptr** est affectée avec l'adresse du premier caractère invalide (au sens de la conversion).

Allocation dynamique de mémoire

Prototypes disponibles dans **stdlib.h**.

```
void * malloc(size_t size);
```

Alloue **size** octets de mémoire et retourne un pointeur générique correspondant à l'adresse du premier octet de la zone, renvoie NULL en cas d'échec.

```
void * realloc(void *ptr, size_t size);
```

Permet de modifier la taille d'une zone de mémoire allouée dynamiquement. **ptr** doit correspondre à l'adresse du premier octet de la zone précédemment allouée par **malloc** ou **realloc**. **size** correspond à la taille en octet de la nouvelle zone allouée. **realloc** garantie que la nouvelle zone contiendra les données présentes dans la zone initiale.

```
void free(void *ptr);
```

Libère une zone mémoire allouée dynamiquement. **ptr** doit correspondre à l'adresse du premier octet de la zone précédemment allouée par **malloc** ou **realloc**.

La liste des fonctions du programme considéré est indiquée ci-après. Certaines fonctions ne sont pas à écrire. Ces fonctions peuvent tout de même être utilisées et considérées comme disponibles.

===== Fichier cvtheque.h =====

```
#ifndef _CVTHEQUE_H_
#define _CVTHEQUE_H_

typedef struct _elt_comp {
    char *comp;
    struct _elt_comp *suiv;
} Elt_Comp;

typedef struct _pers {
    char *nom;
    char *prenom;
    Elt_Comp *lcomp;
} Pers;

typedef struct _elt_pers {
    Pers *pers;
    struct _elt_pers *suiv;
} Elt_Pers;

/* Q1: Creer une personne */
Pers *creer_pers(const char *nom, const char *prenom
                 );

/* Q2: ajouter une compétence à un individu */
void ajouter_comp_pers(Pers *pers, const char *comp)
    ;

/* Q2: ajouter une compétence à une liste de compétences */
Elt_Comp *ajouter_comp(Elt_Comp *list_comp, const
                           char *comp);
```

```
/* Ajouter une personne dans une liste de personnes
 */
Elt_Pers *ajouter_pers(Elt_Pers *lp, Pers *p);

/* Q3: écrire une liste de compétences dans un
   fichier (déjà ouvert) */
void ecrire_comp_rec(FILE *f, Elt_Comp *list_comp);

/* Q3: écrire une liste de personnes dans un fichier
   (déjà ouvert) */
void ecrire_pers_rec(FILE *f, Elt_Pers *list_pers);

/* Q3: écrire une liste de personnes dans un fichier
   */
void ecrire_pers(const char *nom_fichier, Elt_Pers *
                 list_pers);

/* Q4: libérer une liste de compétences */
void liberer_liste_comp(Elt_Comp *list_comp);

/* Q4: libérer une personne */
void liberer_une_pers(Pers *pers);

/* libérer une liste de personnes */
void liberer_pers(Elt_Pers *list_pers, int liberer_p
                  );

/* Q6: créer la liste de personnes ayant une liste
   de compétences donnée */
void sel_pers(Elt_Pers *list_pers, char *comp,
              Elt_Pers **pl_avec, Elt_Pers **pl_sans);

typedef struct _abd {
    Elt_Pers *list_pers;
    char *comp;
    struct _abd *avec_comp;
```

```

struct _abd *sans_comp;
} ABD;

/* Q7: choix d'une compétence (fonction donnée, elle n'est pas à écrire) */
char *choix_competence(Elt_Pers *list_pers, Elt_Comp *comp_a_ignorer);

/* Q7: construction d'un arbre de décision */
ABD *construire_ABD(Elt_Pers *list_pers, Elt_Comp *comp_a_ignorer);

/* Q8: utilisation d'un arbre de décision */
void trouver_personnes(ABD *abd);

/* Q8: afficher les noms (fonction donnée, elle n'est pas à écrire) */
void afficher_noms(Elt_Pers *list_pers);

/* libération de l'arbre (fonction donnée, elle n'est pas à écrire) */
void liberer_abd(ABD *abd);

/* Q9 : lire une liste de personnes depuis un fichier */
Elt_Pers *lire_pers(const char *nom_fichier);

typedef struct _comp_niv {
    char *comp;
    int niveau;
} Comp_Niv;

/*Q10: fonctions génériques */
void *dupliquer_comp_niv(const void *src);
void copier_comp_niv(const void *src, void *dst);
void detruire_comp_niv(void *data);
void afficher_comp_niv(const void *data);
int comparer_comp_niv(const void *a, const void *b);
int ecrire_comp_niv(const void *data, FILE *f);
void * lire_comp_niv(FILE *);

#endif

```