Nom :	
Prénom :	
No. groupe :	
No. carte :	

Programmation et structures de données en C– LU2IN018

Partiel du 10 novembre 2022

1h30

Aucun document n'est autorisé. Le memento qui a été distribué est reproduit à la fin de cet énoncé.

Les calculatrices, baladeurs et autres appareils électroniques sont interdits. Les téléphones mobiles doivent être éteints et rangés dans les sacs.

Toutes les questions sont indépendantes. Pour les questions à choix multiples à 1 point, vous obtenez 1 point si vous avez coché toutes les cases correspondant à des réponses justes et seulement celles-ci. Pour les questions à 2 points ou plus acceptant plusieurs réponses, vous perdez 1 point par réponse erronée (case juste non cochée ou case fausse cochée). La note minimale à une question est 0. Le barème sur 25 points (15 questions) n'a qu'une valeur indicative.

ATTENTION : lisez le sujet dans son intégralité avant de commencer. Certaines questions sont à réponse libre (lecture ou écriture de code). Elles peuvent donc nécessiter plus de temps de réflexion que les questions à choix multiples.

Il ne vous est pas demandé de vérifier qu'un `malloc` a bien alloué la mémoire. De même il n'est pas demandé de vérifier qu'un `fopen` a ouvert correctement le fichier demandé.

Question 1 (2 points)

Cochez les réponses justes :

☐ L'instruction :

```
int tab1[10][50];
```

permet de déclarer un tableau statique d'entiers à deux dimensions, 10 lignes et 50 colonnes.

☐ La séquence d'instructions suivante :

```
int i;
int **tab2 = malloc(sizeof(int*)*10);
for (i=0 ; i<10 ; i++)tab2[i]= malloc(sizeof(int)*50);
```

permet de déclarer un tableau dynamique d'entiers à deux dimensions, 10 lignes et 50 colonnes.

☐ L'instruction :

```
int (int tab2[10][50]);
```

permet de déclarer un tableau dynamique d'entiers à deux dimensions, 10 lignes et 50 colonnes

☐ L'instruction :

```
dynamic int tab2[10][50];
```

permet de déclarer un tableau dynamique d'entiers à deux dimensions, 10 lignes et 50 colonnes

- ☐ La zone mémoire occupée par `tab1` est supérieure ou égale à celle occupée par `tab2`
- ☐ Si l'on a plus besoin de `tab1`, on peut libérer l'espace mémoire qu'il occupe avec l'instruction `free(tab1);`
- ☐ Si l'on a plus besoin de `tab2`, on peut libérer complètement l'espace mémoire qu'il occupe avec l'instruction `free(tab2);`

Question 2 (2 points)

Le présentateur de l'émission «The Voice» souhaite faire un catalogue des chansons chantées par les candidats de ce jeu avec les caractéristiques suivantes : numéro, année de création, titre, interprète original et commentaire.

Soient les structures suivantes :

```
typedef struct _chanson1 {
   int numero;
   int annee_creation;
   char titre[50];
   char interprete_original[50];
   char *commentaire;
   } Chanson1;

typedef struct _chanson2 {
   int numero;
   int annee_creation;
   char titre[50];
   char interprete_original[50];
   char *commentaire;
   struct _chanson2 *suivant;
   } Chanson2;

typedef struct _chanson3 {
   int annee_creation;
   char titre[50];
   char interprete_original[50];
   char *commentaire;
   } Chanson3;
```

On ne limite pas la longueur du catalogue et les chansons y sont classées par année de création croissante.

Le commentaire accompagnant une chanson peut atteindre 300 caractères.

Pour représenter le catalogue, on peut utiliser, soit une liste chaînée dont chaque maillon décrit une chanson, soit un tableau dont chaque case décrit une chanson.

Quels sont les avantages et les inconvénients des deux représentations (liste et tableau).

Laquelle utiliseriez-vous dans ce cas (justifiez) ?

Réponse :

Cochez la structure que vous utiliseriez :

☐ `Chanson1`, en déclarant le tableau ainsi : `Chanson1 tab1[100];`

☐ `Chanson1`, en déclarant le tableau ainsi :

```
Chanson1 *tab1=malloc(100*sizeof(Chanson1));
```

☐ Utiliser une liste chaînée dont les maillons sont de type `Chanson1`.

☐ `Chanson2`, en déclarant le tableau ainsi : `Chanson2 tab1[100];`

☐ `Chanson2`, en déclarant le tableau ainsi :

```
Chanson2 *tab1=malloc(100*sizeof(Chanson2));
```

☐ Utiliser une liste chaînée dont les maillons sont de type `Chanson2`.

☐ `Chanson3`, en déclarant le tableau ainsi : `Chanson3 tab1[100];`

☐ `Chanson3`, en déclarant le tableau ainsi :

```
Chanson3 *tab1=malloc(100*sizeof(Chanson3));
```

☐ Utiliser une liste chaînée dont les maillons sont de type `Chanson3`.

Question 3 (1 point)

Si l'on ne se préoccupe pas de classer les chansons par année de création, quelle va être la représentation (tableau ou liste) que l'on va choisir en mémoire et pourquoi ? Quelle structure de données utiliser ? Y aura-t-il des répercussions concernant l'insertion d'une nouvelle chanson ?

Réponse :

Question 4 (1 point)

Supposons que l'on ne stocke que 30 chansons et qu'on utilise à cet effet le tableau `tab` défini ainsi : `Chanson1 tab[30];`

Le tableau `tab` et les données qu'il contient occupent en mémoire :

(cochez la ou les réponses correctes)

- ☐ 30 fois la taille de 2 entiers + 30 fois 50 octets + 30 fois la taille d'un pointeur.
- ☐ 30 fois la taille d'un entier + 60 fois 50 octets + 30 fois la taille d'un pointeur.
- ☐ 30 fois la taille de 2 entiers + 30 fois 50 octets + 30 fois 25 octets + l'espace nécessaire pour stocker les chaînes de caractères représentant le champ `commentaire` .
- ☐ 30 fois la taille d'un entier + 60 fois 50 octets + l'espace nécessaire pour stocker les chaînes de caractères représentant le champ `commentaire`.
- ☐ 30 fois la taille de 2 entiers + 30 fois 100 octets + 30 fois la taille d'un pointeur + l'espace nécessaire pour stocker les chaînes de caractères représentant le champ `commentaire`.

Question 5 (2 points)

Soit la fonction suivante, qui crée une nouvelle chanson dans une liste chaînée de maillons de type `Chanson2`.

```
Chanson2 *creer_chanson(int num, int annee_creation, char *titre,
   char *interprete_original, char *comment){
 Chanson2 *chans=(Chanson2*)malloc(sizeof(Chanson2));
 chans->numero=num;
 chans->annee_creation=annee_creation;
 strcpy(chans->titre, titre); /* inst1 */
 strcpy(chans->interprete_original,interprete_original);
 chans->commentaire=strdup(comment); /* inst2 */
 chans->suivant=NULL;
 return chans;
  }
```

Expliquer l'effet de l'instruction `inst1`, puis celui de l'instruction `inst2`, en mettant clairement en évidence leurs différences.

Réponse :

Question 6 (2 points)

Soit la fonction main suivante, qui utilise la fonction `creer_chanson` de la question précédente :

```
1  int main(void) {
2     char titre1[60]="O␣sole␣mio";
3           char interprete_original1[60]="Giuseppe␣Anselmi";
4     char commentaire1[60]="chanson␣Italienne␣traditionnelle";

5     Chanson2 *chanson1=
      creer_chanson(10,1898,titre1,interprete_original1,commentaire1)
         ;
6     Chanson2 *chanson2=(Chanson2*) malloc(sizeof(Chanson2));

7     chanson2=chanson1;
8     printf("%s\n",chanson2->titre);
9     printf("%s\n", chanson2->interprete_original);
10    printf("%s\n",chanson2->commentaire);

11    strcpy(chanson1->titre,"Petit␣Papa␣Noel");
12    strcpy(chanson1->interprete_original,"Tino␣Rossi");
13    strcpy(chanson1->commentaire,"Chanson␣de␣Noel␣Francaise");

14    printf("%s\n",chanson2->titre);
15    printf("%s\n", chanson2->interprete_original);
16    printf("%s\n",chanson2->commentaire);

17    Chanson2 *chanson3=(Chanson2*)malloc(sizeof(Chanson2));
18    strcpy(chanson3->titre,chanson1->titre);
19    strcpy(chanson3->interprete_original,chanson1->
   interprete_original);
20    chanson3->commentaire=chanson1->commentaire;
21    printf("%s\n",chanson3->titre);
22    printf("%s\n", chanson3->interprete_original);
23    printf("%s\n",chanson3->commentaire);

24    strcpy(chanson1->titre,"Let␣it␣be");
25    strcpy(chanson1->interprete_original,"Les␣Beatles");
26    strcpy(chanson1->commentaire,"Chanson␣de␣legende");

27    printf("%s\n",chanson3->titre);
28    printf("%s\n", chanson3->interprete_original);
29    printf("%s\n",chanson3->commentaire);
30    return 0;}
```

Définissons respectivement les affichages `Affichage1`, `Affichage2`, `Affichage3`.

Affichage 1 :

O sole mio

Giuseppe Anselmi

chanson Italienne traditionnelle

Affichage 2 :

Petit Papa Noel

Tino Rossi
Chanson de Noel Francaise
Affichage 3 :
Let it be
Les Beatles
Chanson de legende
Cochez les réponses exactes.
L'exécution des lignes 8 à 10 produit :
☐ `Affichage1`
☐ `Affichage2`
☐ `Affichage3`
☐ Autre chose que `Affichage1` ou `Affichage2` ou `Affichage3`
L'exécution des lignes 14 à 16 produit :
☐ `Affichage1`
☐ `Affichage2`
☐ `Affichage3`
☐ Autre chose que `Affichage1` ou `Affichage2` ou `Affichage3`
L'exécution des lignes 21 à 23 produit :
☐ `Affichage1`
☐ `Affichage2`
☐ `Affichage3`
☐ Autre chose que `Affichage1` ou `Affichage2` ou `Affichage3`
L'exécution des lignes 27 à 29 produit :
☐ `Affichage1`
☐ `Affichage2`
☐ `Affichage3`
☐ Autre chose que `Affichage1` ou `Affichage2` ou `Affichage3`

Question 7 (2 points)
Ecrire la fonction de prototype :

```
void liberer(Chanson2 *liste_chansons);
```

qui libère l'espace mémoire occupé par une liste chaînée dont les maillons sont de type `Chanson2`.
Attention : il y aura une pénalisation si TOUS les maillons ne sont pas libérés.

Réponse :

Question 8 (2 points)

Soit le programme suivant :

```
#include <stdio.h>
void afficher(char *tab, int a1, int a2, float b, float *c){
   printf("tab=");
   for (int i=0;i<5;i++) printf("%c␣",tab[i]);
   printf("\na1=%d\n",a1);
   printf("a2=%d\n",a2);
   printf("b=%f\n",b);
   printf("*c=%f\n", *c);
}

void transformation(char *tab, int a1, int a2, float b, float *c) {
   *(tab+2)=*tab;
   tab[3]=*(tab+1);
   a1=a2/10;
   a2=a1*2;
   c=&b;
   afficher(tab,a1,a2,b,c);   /* affichage 2 */
}

int main(void){
   int a1, a2;
   float b,f=4.5,*c;
   char tab[]={'a','b','c','d','e'};
   a1=25;
   a2=17;
   b=5.5;
```

```
    c=&f;
    afficher(tab, a1,a2, b, c); /* affichage 1 */
    transformation(tab, a1, a2, b, c);
    afficher(tab, a1, a2, b, c); /* affichage 3 */
    return 0;
}
```

On exécute le main du programme.

Qu'affiche le programme pour les appels à la fonction `afficher` correspondant successivement à **affichage 1**, **affichage 2** et **affichage 3** ?

Réponse :

Question 9 (1 point)

Un marchand de jouets est spécialisé dans la vente des Playmobil et il vend toutes les boîtes possibles de la collection. Ces boîtes ont toutes un numéro particulier. Il y en a tant que leur gestion en devient vraiment compliquée et lorsque des clients lui demandent la boîte numéro x, le pauvre marchand se demande ce qu'il y a dans cette boîte. Il décide donc d'écrire un programme pour gérer un catalogue qu'il représentera en mémoire par un tableau dont chaque case stockera les caractéristiques d'une boîte Playmobil (c'est un passionné d'informatique !).

Voici la description de la structure décrivant une boîte :

```
typedef struct Boite_Playmobil {
         int numero;
         char collection[20];
         char *description;
      } Playmobil;
```

Il n'y a pas plus de 250 boîtes, aussi dimensionne-t-il son tableau à 250. Il y insère les boîtes de manière séquentielle en prenant soin de conserver un compteur courant du nombre de boîtes déjà insérées dans le tableau.

Voici un programme qui gère les boîtes de Playmobil de notre marchand de jouets :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5
6  typedef struct Boite_Playmobil{
7     int numero;
8     char collection[50];
9     char *description;
1     } Playmobil;
11
12   Playmobil *creer_boite(int num,char *collect, char *descript)
   {
13 Playmobil *p=(Playmobil*)malloc(sizeof(Playmobil));
14 p->numero=num;
15 strcpy(p->collection, collect);
16 strcpy(p->description, descript);
17 return p;
18    }
19
20    int main(void){
21    Playmobil Tab_Playmobil[250];
22    Playmobil *p;
23    int nb_boites=0;
24    p=creer_boite(78,"Princess", "Grand␣palais␣de␣princesse:␣
   magnifique␣palais␣de␣3␣etages␣relies␣par␣2␣escaliers␣monumentaux␣
   en␣colimacon.␣Le␣palais␣comporte␣plusieurs␣tours␣avec␣differentes␣
   pieces,␣en␣bas,␣cuisine,␣salle␣a␣manger,␣au␣premier,␣chambre␣des␣
   princesses␣et␣salle␣de␣bain,␣au␣deuxieme,␣salle␣de␣musique␣et␣
   boudoir.");
25    nb_boites=nb_boites+1;
26    Tab_Playmobil[nb_boites]=*p;
27    p=creer_boite(123,"PlaymobilPlus","Caleche␣du␣couple␣royal␣
   tiree␣par␣deux␣chevaux␣arnaches␣et␣conduite␣par␣un␣cocher␣
   moustachu,␣valises␣et␣panier␣de␣pique-nique␣du␣couple␣royal,␣deux␣
   chevaux␣et␣trois␣personnages.");
28    nb_boites=nb_boites+1;
29    Tab_Playmobil[nb_boites]=*p;
```

```
30    return 0;
31 }
```

On le compile et on l'exécute ainsi :

```
bash$ gcc -g -Wall -o playmobil playmobil.c
bash$ ./playmobil
```

On obtient le message :

```
Erreur de segmentation.
```

Cochez la ou les réponses correctes :

- ☐ C'est un problème de compilation.
- ☐ C'est un problème rencontré lors de l'exécution.
- ☐ `gcc` peut être utile pour résoudre le problème.
- ☐ `ddd` peut être utile pour résoudre le problème.
- ☐ il n'y a pas d'autre solution que de relire soigneusement son code pour trouver l'erreur, car on ne possède pas d'outil logiciel qui puisse aider.

Question 10 (2 points)

L'exécution de ce code avec `valgrind` donne :

```
==7294== Memcheck, a memory error detector
==7294== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==7294== Using Valgrind-3.16.1 and LibVEX; rerun with -h for copyright info
==7294== Command: ./playmobil
==7294==
==7294== Use of uninitialised value of size 8
==7294==    at 0x483BDE4: strcpy (vg_replace_strmem.c:511)
==7294==    by 0x10919C: creer_boite (playmobil.c:16)
==7294==    by 0x1091CD: main (playmobil.c:24)
==7294==
==7294== Invalid write of size 1
==7294==    at 0x483BDE4: strcpy (vg_replace_strmem.c:511)
==7294==    by 0x10919C: creer_boite (playmobil.c:16)
==7294==    by 0x1091CD: main (playmobil.c:24)
==7294==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
==7294==
==7294==
==7294== Process terminating with default action of signal 11 (SIGSEGV)
==7294==  Access not within mapped region at address 0x0
==7294==    at 0x483BDE4: strcpy (vg_replace_strmem.c:511)
==7294==    by 0x10919C: creer_boite (playmobil.c:16)
==7294==    by 0x1091CD: main (playmobil.c:24)
==7294==  If you believe this happened as a result of a stack
==7294==  overflow in your program's main thread (unlikely but
==7294==  possible), you can try to increase the size of the
==7294==  main thread stack using the --main-stacksize= flag.
==7294==  The main thread stack size used in this run was 8388608.
==7294==
==7294== HEAP SUMMARY:
==7294==     in use at exit: 64 bytes in 1 blocks
==7294==   total heap usage: 1 allocs, 0 frees, 64 bytes allocated
```

```
==7294==
==7294== LEAK SUMMARY:
==7294==    definitely lost: 0 bytes in 0 blocks
==7294==    indirectly lost: 0 bytes in 0 blocks
==7294==      possibly lost: 0 bytes in 0 blocks
==7294==    still reachable: 64 bytes in 1 blocks
==7294==         suppressed: 0 bytes in 0 blocks
==7294== Rerun with --leak-check=full to see details of leaked memory
==7294==
==7294== Use --track-origins=yes to see where uninitialised values come from
==7294== For lists of detected and suppressed errors, rerun with: -s
==7294== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

```
Erreur de segmentation
```

Que peut-on déduire de ce listing ? Où est l'erreur (ou où sont les erreurs ?)

Réponse :

Question 11 (1 point)

Proposez une ou plusieurs lignes de code permettant de corriger l'erreur (ou les erreurs) identifée(s) dans la question précédente, en précisant quelle(s) ligne(s) elle(s) remplace(nt).

Réponse :

Question 12 (2 points)

Une boîte Playmobil est ainsi décrite dans un fichier texte :

```
34
Princess
Palais de princesse : structure du type du Grand Palais de princesse,
    mais ne comporte qu'un␣etage␣et␣le␣mobilier␣des␣pieces␣est␣reduit
   ␣egalement.
```

La première ligne est le numéro, la deuxième la collection et la description tient sur une seule longue ligne (la 3eme) qui est coupée ici pour l'affichage.

Soit la fonction de lecture suivante :

```
Playmobil *lire_playmobil(FILE *f){
   char collection[50];
   char description[300];
   int numero;
   /* lecture de numero */
   /* lecture de collection */
   /* lecture de description */
   return creer_boite(numero, collection, description);
}
```

(On ignorera les retours à la ligne pendant la lecture).

et la fonction d'écriture suivante :

```
void ecrire_boite(FILE *f, Playmobil *p){
   /* écriture de la boite *.
}
```

Indiquez la (ou les) réponse(s) parmi les suivantes qui donne(nt) des instructions correctes pour :

— La lecture de numero
— La lecture de collection
— La lecture de description
— L'écriture de la boîte

qui figurent respectivement dans la fonction de lecture et la fonction d'écriture précédentes.

☐
```
fread(&numero,sizeof(int),1,f);
fscanf(f,"␣%s␣",&collection);
fgets(&description, 300,f);
fwrite(&p,sizeof(Playmobil),1,f);
```

☐
```
&numero=fgetc(f);
fgets(&collection,50,f);
fgets(&description,300,f);
fwrite(&p,sizeof(Playmobil),1,f);
```

☐
```
fscanf(f,"␣%d␣",&numero);
fgets(collection,50,f);
fgets(description,300,f);
fprintf(f,"%d\n%s\n%s\n", p->numero, p->collection, p->description
   );
```

☐
```
fread(numero,sizeof(int),1,f);
&collection=fgetc(f);
fgets(description, 300, f);
fwrite(p,sizeof(Playmobil),1,f);
```

☐
```
*numero=fgetc(f);
collection=fgetc(f);
description=fgetc(f);
fprintf(f,"%d\n%s\n%s\n", p->numero, p->collection, p->description
   );
```

Question 13 (2 points)

Soit la fonction main qui ouvre le fichier "playmobil_source.txt", lit les caractéristiques d'une boîte Playmobil et les écrit dans un fichier "playmobil_destination.txt". On fait l'hypothèse que l'ouverture et la fermeture des fichiers se passent sans problèmes, sans qu'il soit nécessaire de les tester.

```
int main(void){
   /* inst1 */
   Playmobil *p=lire_playmobil(fs);
   FILE *fd=fopen("playmobil_destination.txt","w");
   /* inst2 */
   fclose (fs);
   fclose(fd);
   /* inst3 */
   free(p);
   return 0;}
```

Cochez les réponses exactes :

☐ `/* inst1 */` doit être remplacée par:
```
FILE *fs=fopen("playmobil_source.txt","r");
```

☐ `/* inst1 */` doit être remplacée par:
```
FILE *fd=fopen("playmobil_source.txt","r");
```

☐ `/* inst2 */` doit être remplacée par:
```
lire_playmobil(fd,*p);
```

☐ `/* inst2 */` doit être remplacée par:
```
ecrire_boite(fd,p);
```

☐ `/* inst2 */` doit être remplacée par:
```
ecrire_boite(fd,*p);
```

☐ `/* inst3 */` doit être remplacée par:
```
free(p->description);
```

☐ `/* inst3 */` doit être remplacée par:
```
free(*collection);
```

☐ `/* inst3 */` doit être remplacée par:
```
free(p->collection);
```

Question 14 (1 point)

Les fonctions précédentes sont réparties entre deux fichiers : `playmobil.c` et `main_playmobil.c`.

Les prototypes des fonctions et la déclaration de la structure Playmobil sont dans `playmobil.h`.

Cochez, parmi les propositions suivantes, celle(s) qui permet(tent) de créer un exécutable à partir de ces fichiers.

☐
```
ddd -Wall -o main_playmobil main_playmobil.c playmobil.c
```

☐
```
ddd -c -Wall -o main_playmobil.o main_playmobil.c
ddd -c -Wall -o playmobil.o playmobil.c
ddd -Wall -o main_playmobil main_playmobil.o playmobil.o
```

☐
```
gcc -c -Wall -o main_playmobil.o main_playmobil.c
gcc -c -Wall -o playmobil.o playmobil.c
gcc -Wall -o main_playmobil main_playmobil.o playmobil.o
```

☐
```
gcc -Wall -o main_playmobil main_playmobil.c playmobil.c
```

☐
```
gcc -Wall -o main_playmobil main_playmobil.c playmobil.h
```

☐
```
gcc -Wall -o main_playmobil main_playmobil.c
```

Question 15 (2 points)

Cochez la ou les réponses correctes :

☐ Dans la trace d'une compilation, on peut systématiquement ignorer les warnings qui n'ont en fait aucune importance.

☐ "Segmentation fault" ou "Erreur de segmentation" est un diagnostic du compilateur gcc qui indique un problème relatif à la mémoire.

☐ Une fuite mémoire résulte de la non-désallocation de la mémoire qui a été précédemment allouée.

☐ Valgrind n'est pas utile pour détecter des fuites mémoire.

☐ ddd est utile pour debugger un programme et possède une interface graphique.

☐ Utiliser un fichier Makefile est intéressant, car il permet d'expliciter les dépendances de fichiers les uns par rapport aux autres et permet ainsi une cohérence dans leur gestion.

☐ Les fichiers headers de suffixe .h sont utilisés dès que les fonctions d'un programme sont situées dans plusieurs fichiers.

☐ Le pré-processeur intervient avant le compilateur et son rôle est de pré-traiter un programme en le complétant grâce aux directives précédées par `#`, `#define` ou `#include`.

☐ Un fichier header, outre les prototypes de certaines fonctions, peut contenir la définition de structures C.

☐ `#define Taille = 100` est un exemple de directive valide pour le pré-processeur.

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 son 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 retournée 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 retournée 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`.