Votre numéro d'anonymat : | | | |

Programmation et structures de données en C– 2I001

Examen du 16 décembre 2015
2 heures
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 60 points (12 questions) n'a qu'une valeur indicative.

Le mémento qui vous a été distribué est reproduit à la fin de l'énoncé. Les appels à `malloc` seront supposés réussir. Il ne sera pas nécessaire de vérifier la valeur de retour.

Le Compte est bon

L'objectif de ce sujet est de vous faire développer un programme permettant de gagner à coup sûr au célèbre jeu télévisé *Le Compte est bon*. Ce jeu consiste à obtenir un nombre (tiré au hasard dans l'intervalle 100 à 999) à partir d'opérations élémentaires (Addition $+$, Soustraction $-$, Multiplication $\times$, Division $\div$) sur des entiers naturels, en partant de nombres tirés au hasard (de 1 à 10, 25, 50, 75 et 100).

Lorsque l'émission n'était pas informatisée, le jeu comportait vingt-quatre plaques : les nombres de 1 à 10 présents en double exemplaire et les nombres 25, 50, 75 et 100 présents en un seul exemplaire. Le joueur tire au hasard 6 cartes et dispose de 45 secondes pour déterminer la combinaison utilisant le moins de cartes possible et qui permet en utilisant des opérations simples d'atteindre la valeur attendue.

Voici un exemple :
Nombres tirés : 3, 75, 2, 4, 1, 10
Valeur attendue : 888
Solution :
$75 - 1 = 74$
$3 \times 4 = 12$
$74 \times 12 = 888$

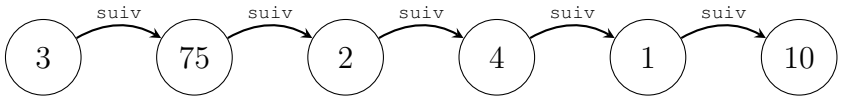
Pour cet exemple, le compte est bon, si ce n'est pas le cas le programme doit chercher la valeur la plus proche possible de celle recherchée (cette valeur approchée peut être inférieure au supérieure à celle recherchée).

1 Structures de données

Pour l'ensemble de ce sujet nous allons utiliser une unique structure de donnée. Dans un premier temps elle nous permettra de constituer des listes chaînées et dans un second temps nous l'utiliserons pour construire des arbres d'expression. Nous avons nommé cette structure `Lm_Nd` pour ***éLément_Noeud*** et en voici sa définition :

```
enum type_n {VALEUR, OPERATEUR};
enum oper {PLUS, MOINS, MULT, DIV};

typedef struct _eln
   {
   enum type_n type; // Valeur ou operateur
   int val; // valeur du noeud quelque soit le type
   enum oper op; // operateur si type == operateur
   struct _eln *g; //fils gauche
   struct _eln *d; //fils droit
   struct _eln *suiv; //element suivant
   } Lm_Nd;
```

Le champ `val` est affecté que le noeud corresponde à une valeur ou à un opérateur. Dans le premier cas la valeur correspond à celle du sous-arbre. Pour les valeurs de l'exemple précédent nous obtenons la liste suivante :



2 Liste de valeurs

Pour ce premier exercice les `Lm_Nd` manipulés seront exclusivement de type `VALEUR` et seront utilisés pour constituer des listes chaînées, les autres pointeurs (`g` et `d`) devront systématiquement être initialisés à `NULL`.

Question 1 (3 points)

Le jeu initial correspondant aux 6 cartes choisies aléatoirement sera représenté par une liste chaînée. Écrivez une fonction d'ajout d'une valeur à une liste chaînée (qui peut être vide). L'ordre des valeurs dans la liste n'a pas d'importance.

```
Lm_Nd *ajouter_val(Lm_Nd *liste, int val);
```

Solution:

```
Lm_Nd *ajouter_val(Lm_Nd *liste, int val)
   {
   Lm_Nd *nv = malloc(sizeof(Lm_Nd));
   nv->type = VALEUR;
   nv->val = val;
   nv->suiv = liste;
   return nv;
   }
```

Les 6 cartes initiales doivent être choisies aléatoirement parmi les 24 cartes disponibles, évidement une carte ne peut être sélectionnée qu'une fois, pour garantir cette condition les cartes du jeu vont être stockées dans un tableau de 24 entiers.

```
int pioche[24] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1, 2, 3, 4, 5, 6,
    7, 8, 9, 10, 25, 50, 75, 100};
```

Chaque case du tableau correspond à une carte, quand une carte a été tirée la case correspondante du tableau est mise à 0.

Question 2 (6 points)

Écrivez la fonction permettant de tirer aléatoirement 6 cartes/valeurs.

```
Lm_Nd *tirer_cartes ();
```

Cette fonction retourne une liste constituées de 6 éléments correspondant aux 6 valeurs tirées. Vous pourrez utiliser la fonction `int rand(void)` qui retourne un entier aléatoire entre 0 et `RAND_MAX`.

Solution:

```
Lm_Nd *tirer_cartes ()
   {
   int pioche[24] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1, 2, 3, 4, 5,
      6, 7, 8, 9, 10, 25, 50, 75, 100};
   int i;
   Lm_Nd *jeu = NULL;
   for (i=0; i<24; i++)
      {
      unsigned int vi;
      do
         vi = rand() % 24;
      while (pioche[vi] == 0);
      jeu = ajouter_val(jeu, pioche[vi]);
      pioche[vi] = 0;
      }
   return jeu;
   }
```

Question 3 (3 points)

Écrivez une fonction qui affiche les valeurs tirées :

```
void afficher_cartes(Lm_Nd *jeu);
```

Solution:

```
void afficher_cartes(Lm_Nd *jeu)
   {
   while (jeu)
      {
      printf("%d␣", jeu->val);
      jeu = jeu->suiv;
      }
   putchar('\n');
   }
```

Question 4 (3 points)

Plus tard vous aurez à libérer la mémoire correspondant à une telle liste chaînée, écrivez tout de suite la fonction correspondante :

```
void liberer_liste(Lm_Nd *plm);
```

Solution:

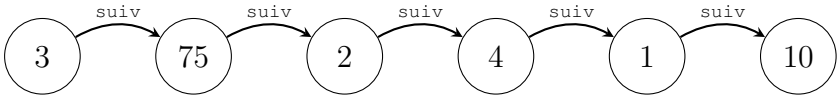
```
void liberer_liste(Lm_Nd *plm)
   {
   while (plm)
      {
      Lm_Nd *suiv = plm->suiv;
      free(plm);
      plm = suiv;
      }
   }
```

3 Existence d'une solution

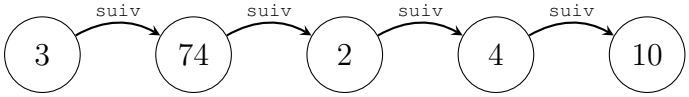
Nous allons maintenant chercher si il existe une solution, nous allons pour cela parcourir l'ensemble des expressions formables à partir des valeurs tirées. Nous allons pour cela utiliser un algorithme récursif qui à chaque étape va réduire de 1 le nombre de valeurs restant utilisable en en remplaçant 2 par leur somme, leur différence, leur produit ou le résultat de leur division entière.

Ce traitement devra être répété tant qu'il restera au moins 2 valeurs et que la valeur recherchée n' aura pas été rencontrée. Nous avons ainsi la garantie que toutes les solutions formables ont été testées.

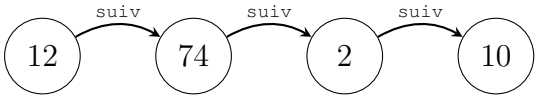
En partant des valeurs de l'exemple et en *trichant* pour trouver plus vite le résultat voici les étapes de notre algorithme.



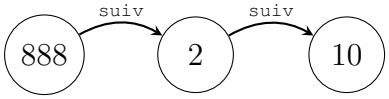
Première opération : $75 - 1 = 74$



Seconde opération : $3 \times 4 = 12$



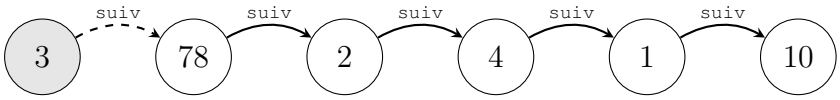
Seconde opération : $12 \times 74 = 888$



Nous n'avons pu tricher que parce que nous connaissions déjà la solution, l'algorithme réel que vous allez devoir implanter n'a pas cette possibilité, il va donc devoir explorer toutes les expressions formables. À chaque étape il va devoir pour tous les couples de valeurs les remplacer successivement par toutes les combinaisons de ces valeurs : leur somme, leur différence, leur produit et le résultat de leur division entière si elle est possible (reste égal à zéro).

Nous vous conseillons d'imbriquer trois boucles, les deux premières permettent d'établir toutes les combinaisons de valeurs, la troisième de parcourir les quatre opérations possibles.

La première boucle parcours toutes les valeurs de la liste, sélectionne une première valeur et teste si cette valeur correspond à la valeur recherchée (dans ce cas le travail est terminé). La seconde parcourt les valeurs restantes (la suite de la liste) et sélectionne une seconde valeur. La troisième remplace la seconde valeur (celle de la seconde boucle) par la somme des deux premières :



Puis appel récursif sur les 5 dernières valeurs. L'arc en pointillé signifie que le pointeurs `suiv` du premier élément reste valide mais que cette valeur a déjà été sélectionnée.

Question 5 (12 points)

Écrivez la fonction qui recherche si il existe une solution pour atteindre la valeur recherchée. Pour limiter la taille du code à écrire vous pourrez considérer que toutes les opérations ($+ - \times$) sont commutatives. Les soustractions peuvent produire des valeurs négatives.

```
int lecompte_v0(Lm_Nd *jeu, int val);
```

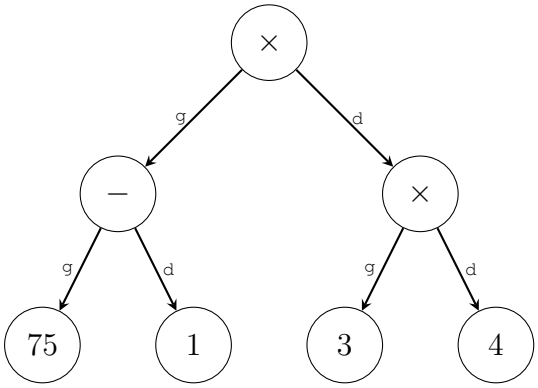
Cette fonction retourne 1 si la recherche a réussi et 0 sinon.

Solution:

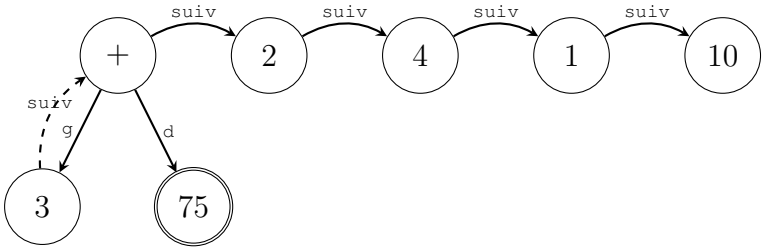
```
int lecompte_v0(Lm_Nd *jeu, int val)
   {
   Lm_Nd *op1, *op2;
   enum oper op;

   for (op1 = jeu; op1; op1 = op1->suiv)
      {
      int op1_v = op1->val;

      if (op1_v == val) return 1;
      for (op2 = op1->suiv; op2; op2 = op2->suiv)
         {
         int op2_v = op2->val;

         for(op = PLUS; op <= DIV; op++)
            {
            switch (op)
               {
               case PLUS :
                  op2->val = op1_v + op2_v; break;

               case MOINS :
                  op2->val = op1_v - op2_v; break;

               case MULT :
                  op2->val = op1_v * op2_v; break;

               case DIV :
                  if ((op1_v == 0) || (op2_v == 0)) continue;
                  if ((op1_v % op2_v) == 0) op2->val = op1_v / op2_v
                     ; break;
                  if ((op2_v % op1_v) == 0) op2->val = op2_v / op1_v
                     ;
               }
            if (lecompte_v0(op1->suiv, val)) return 1;
            }
         op2->val = op2_v;
         }
      }
   return 0;
   }
```

4 Représentation de la solution

Pour représenter la solution trouvée nous avons choisi d'utiliser un arbre d'expression. Cet arbre s'appuie sur la structure déjà utilisée à la différence que nous allons également utiliser des `Lm_Nd` de type opérateur. Pour l'exemple présenté notre objectif est de produire l'arbre d'expression suivant :

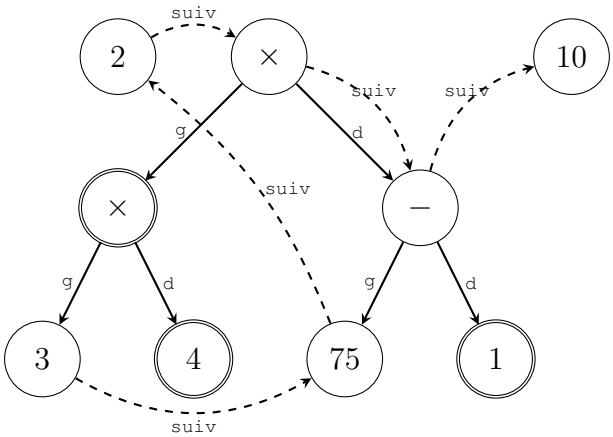


Pour construire cet arbre d'expression nous allons utiliser quasiment le même algorithme que pour les questions précédentes à la différence que au lieu de remplacer la seconde valeur par une des combinaisons possibles des deux valeurs sélectionnées nous allons la remplacer par l'arbre correspondant à l'expression. Voici une représentation de la structure de données après la première étape de l'algorithme (sélection des deux premières valeurs et de l'opérateur d'addition).



Pour accélérer l'algorithme le champ `val` des noeuds de type `OPERATEUR` sera mis à jour avec la valeur résultant de l'évaluation de l'expression correspondante. La valeur du second noeud de la liste a été modifiée (75 est devenu $+$) et un nouveau noeud a été ajouté (fils droit) avec la valeur 75. Afin d'éviter les fuites de mémoire nous avons fait le choix de préserver le chaînage initial de la liste et d'ajouter des nouveaux noeuds représentés avec un double cerclage car il corresponde à des variables locales de la fonction de recherche de la solution. Cette méthode nous évite d'avoir gérer la libération de la mémoire correspondant à des noeuds que nous ne voulons pas conserver.

À l'issue de la recherche nous obtenons la structure suivante :



Comme vous le voyez sur la figure, l'arbre construit utilise pour partie des noeuds correspondant à des

variables locales à la fonction de recherche, vous ne pouvez donc pas directement retourner un pointeur sur cet arbre comme résultat de la recherche. Il va donc vous être nécessaire d'effectuer une recopie de l'arbre.

Question 6 (6 points)

Écrivez une fonction qui réalise la copie d'un arbre fourni en arguments et retourne un pointeur sur ce nouvel arbre. À la valeur près des pointeurs l'arbre copie devra être rigoureusement identique à l'arbre source.

```
Lm_Nd *copier_arbre(Lm_Nd *src);
```

Solution:

```
Lm_Nd *copier_arbre(Lm_Nd *src)
   {
   if (src == NULL) return NULL;

   Lm_Nd *nv = malloc(sizeof(Lm_Nd));
   nv->type = src->type;
   nv->val = src->val;
   nv->op = src->op;
   nv->suiv = NULL;
   nv->g = copier_arbre(src->g);
   nv->d = copier_arbre(src->d);

   return nv;
   }
```

Question 7 (6 points)

En vous inspirant très fortement de ce que vous avez écrit pour la `v0` écrivez la fonction qui recherche une solution pour atteindre la valeur recherchée. Pour limiter la taille du code à écrire vous pourrez considérer que toutes les opérations ($+ - \times$) sont commutatives. Les soustractions peuvent produire des valeurs négatives.

```
Lm_Nd *lecompte_v1(Lm_Nd *jeu, int val);
```

Cette fonction retourne un pointeur sur l'arbre résultat ou `NULL` en cas de recherche infructueuse.

Solution:

```
Lm_Nd *lecompte_v1(Lm_Nd *jeu, int val)
   {
   Lm_Nd *op1, *op2, *res;
   enum oper op;

   for (op1 = jeu; op1; op1 = op1->suiv)
      {
      int op1_v = op1->val;

      if (op1_v == val) return copier_arbre(op1);
      for (op2 = op1->suiv; op2; op2 = op2->suiv)
         {
         int op2_v = op2->val;
         Lm_Nd cpop2 = *op2;
         op2->type = OPERATEUR;
         op2->g = op1;
         op2->d = &cpop2;
         for(op = PLUS; op <= DIV; op++)
            {
            op2->op = op;
            switch (op)
               {
               case PLUS :
                  op2->val = op1_v + op2_v; break;
               case MOINS :
                  op2->val = op1_v - op2_v; break;
               case MULT :
                  op2->val = op1_v * op2_v; break;
               case DIV :
                  if ((op1_v == 0) || (op2_v == 0)) continue;
                  if ((op1_v % op2_v) == 0) op2->val = op1_v / op2_v
                     ; break;
                  if ((op2_v % op1_v) == 0)
                     {
                     op2->g = op1;
                     op2->d = &cpop2;
                     op2->val = op2_v / op1_v;
                     }
               }
            if ((res = lecompte_v1(op1->suiv, val))) return res;}
         *op2 = cpop2;}
      }
   return NULL;}
```

Question 8 (6 points)

Maintenant que vous avez trouvé la solution il serait triste de ne pas l'afficher. Écrivez la fonction qui réalise cet affichage sous forme infixe en utilisant des parenthèses pour les opérateurs.

```
void afficher_arbre(Lm_Nd *pa);
```

Exemple pour l'expression habituelle :

```
((75 - 1) x (4 x 3))
```

Solution:

```
void afficher_arbre(Lm_Nd *pa)
   {
   char *oper = "+-*/";
   if (pa->type == OPERATEUR)
      {
      putchar('(');
      afficher_arbre(pa->g);
      printf("%c", oper[pa->op]);
      afficher_arbre(pa->d);
      putchar(')');
      }
   else
      printf("%d", pa->val);
   }
```

Question 9 (3 points)

Écrivez la fonction permettant de libérer la mémoire allouée pour un arbre.

```
void liberer_arbre(Lm_Nd *pa);
```

Solution:

```
void liberer_arbre(Lm_Nd *pa)
   {
   if (pa == NULL) return;
   liberer_arbre(pa->g);
   liberer_arbre(pa->d);
   free(pa);
   }
```

Question 10 (3 points)

Pour la suite de cet exercice nous allons devoir déterminer le nombre de valeurs présentes dans un arbre d'expression. Écrivez cette fonction retourne ce nombre.

```
int nb_valeurs(Lm_Nd *pa);
```

Solution:

```
int nb_valeurs(Lm_Nd *pa)
   {
   if (pa->type == OPERATEUR)
      return nb_valeurs(pa->g) + nb_valeurs(pa->d);
   return 1;
   }
```

5 Recherche de la solution approchée la meilleure

Il existe des configurations (jeu et valeur recherchée) pour lesquelles il n'existe pas de solution de solution exacte. Il faudra dans ce cas déterminer l'expression qui approche le plus la valeur recherchée. Il peut exister plusieurs solutions, dans ce cas nous considérerons que la meilleure est celle qui utilise le moins de cartes (valeurs).

Il est probable que dans votre code (pour la recherche d'une expression) vous ayez écrit la ligne suivante :

```
if (op1->val == val) return copier_arbre(op1);
```

Nous vous proposons de remplacer cette ligne par un appel à la fonction :

```
update_best(best, op1, val);
```

`best` étant un pointeur sur un pointeur sur un arbre représentant la meilleure solution trouvée jusqu'alors, `op1` un pointeur sur l'arbre candidat pour devenir la meilleure solution et `val` la valeur recherchée.

Question 11 (6 points)

Écrivez la fonction ;

```
void update_best(Lm_Nd **best, Lm_Nd *pa, int val);
```

Solution:

```
void update_best(Lm_Nd **best, Lm_Nd *pa, int val)
   {
   if (*best == NULL) *best = copier_arbre(pa);
   else
      {
      if ((pa->val == val) && (((*best)->val != val) || (nb_valeurs(
         pa) < nb_valeurs(*best))))
         {
         liberer_arbre(*best);
         *best = copier_arbre(pa);
         }
      else if (abs (pa->val - val) < abs((*best)->val - val))
         {
         liberer_arbre(*best);
         *best = copier_arbre(pa);
         }
      }
   }
```

Question 12 (3 points)

Nous nous intéressons maintenant à la fonction `v2` qui permet de trouver la meilleure solution.

```
void lecompte_v2(Lm_Nd *jeu, int val, Lm_Nd **best);
```

Quelles sont les modifications à apporter à votre `v1` pour passer à la `v2`.

Solution: L'appel à `update_best` :

```
   for (op1 = jeu; op1; op1 = op1->suiv)
      {
      int op1_v = op1->val;

      update_best(best, op1, val);

      for (op2 = op1->suiv; op2; op2 = op2->suiv)
```

La non propagation de la valeur de retour lors de l'appel récursif :

```
            lecompte_v2(op1->suiv, val, best);
```

Mémento de l'UE 2I001

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, ...);
```

La fonction `scanf` permet de saisir et analyser un texte saisi sur l'entrée standard (par défaut le clavier). Le texte saisi 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` 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`.