ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE
EIDGENÖSSISCHE TECHNISCHE HOCHSCHULE – LAUSANNE
POLITECNICO FEDERALE – LOSANNA
SWISS FEDERAL INSTITUTE OF TECHNOLOGY – LAUSANNE

Faculté Informatique et Communications
Cours de Programmation Orientée Système, sections IN et SC
Chappelier J.-C.

PROGRAMMATION ORIENTÉE SYSTÈME

Correction Examen

30 mai 2022

SUJET A

INSTRUCTIONS (à lire attentivement)

IMPORTANT ! Veuillez suivre les instructions suivantes à la lettre sous peine de voir votre examen annulé dans le cas contraire.

1. Vous disposez d'une heure quarante-cinq minutes pour faire cet examen (8h15 – 10h00).
2. Vous devez **écrire à l'encre noire ou bleu foncée**, pas de crayon ni d'autre couleur.
   N'utilisez **pas non plus de stylo effaçable** (perte de l'information à la chaleur).
3. Vous avez droit à toute documentation papier.
   En revanche, vous ne pouvez pas utiliser d'ordinateur personnel, ni de téléphone portable, ni aucun autre matériel électronique.
4. Répondez aux questions directement sur la donnée ; ne joignez aucune feuille supplémentaire ; **seul ce document sera corrigé**.
   Si nécessaire, il y a des pages supplémentaires en fin de copie.
5. Lisez attentivement et *complètement* les questions de façon à ne faire que ce qui vous est demandé. Si l'énoncé ne vous paraît pas clair, ou si vous avez un doute, demandez des précisions à l'un des assistants.
6. L'examen comporte quatre exercices indépendants, qui peuvent être traités dans n'importe quel ordre, mais qui ne rapportent pas la même chose (les points sont indiqués, le total est de 126 points) ; tous les exercices comptent pour la note finale :
   — question 1 : 33 points ;
   — question 2 : 25 points ;
   — question 3 : 29 points ;
   — question 4 : 39 points.

Question 1 – Petites questions [33 points]

1.1 Tableau [5 points]

Le code suivant est-il correct ? Si non, corriger toutes les erreurs (récrivez directement sur le code ou des *portions* de code à droite).

Note : la correction d'une erreur qui n'en est pas une sera pénalisée.

```
#include <stdio.h>

typedef double el_t;
#define EL_FMT "%lf"

typedef el_t Array[];

void display(Array a, size_t s)
{
  printf("{ ");
  if (s > 0) {
    printf(EL_FMT, a[0]);

    for (size_t i = 1; i <= s; ++i) {
      printf(", ");
      printf(EL_FMT, a[i]);
    }
  }
  printf(" }");
}

int main(void)
{
  Array tab = {
    1.2, 2.3, 3.4, 4.5, 5.6,
    6.7, 7.8, 8.9, 9.0, 0.1
  };

  display(tab, sizeof(tab));
  putchar('\n');

  return 0;
}
```

Il faut :

1. changer le `<=` en `<` dans la boucle `for`
2. passer la bonne taille à l'appel de `display()` :

```
display(tab, sizeof(tab) / sizeof(tab[0]));
```

1.2 Tailles [8 points]

Sur une architecture 64 bits [1], qu'affiche le code suivant ?

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

void pequal(const char* prefix, size_t v1, size_t v2)
{
  if (prefix != NULL) printf("%s ", prefix);

  if (v1 == v2) printf("VRAI (%zu)\n"     , v1    );
  else          printf("FAUX : %zu, %zu\n", v1, v2);
}

void f(const char* s, char t[])
{
  pequal("5)", sizeof(s), strlen(s));
  pequal("6)", sizeof(t), strlen(t));
  pequal("7)", sizeof(s), sizeof(t));
  pequal("8)", strlen(s), strlen(t));
}

int main(void)
{
#define MESSAGE "ABCD"
  const char* s1 = MESSAGE;
  char tab[]     = MESSAGE;

  pequal( "1)", sizeof(s1) , strlen(s1)  );
  pequal( "2)", sizeof(tab), strlen(tab) );
  pequal( "3)", sizeof(s1) , sizeof(tab) );
  pequal( "4)", strlen(s1) , strlen(tab) );

  f(s1, tab);

  return 0;
}
```

1) FAUX : 8, 4
2) FAUX : 5, 4
3) FAUX : 8, 5
4) VRAI (4)

5) FAUX : 8, 4
6) FAUX : 8, 4
7) VRAI (8)
8) VRAI (4)

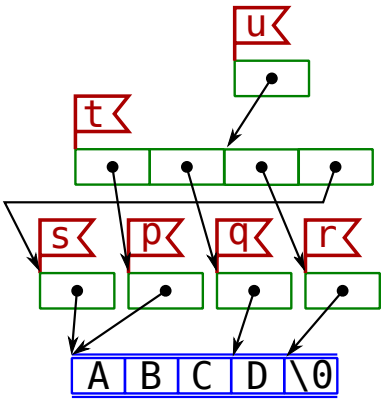
1. c.-à-d. `sizeof(void*) == 8`

1.3 Point...teurs [15 points]

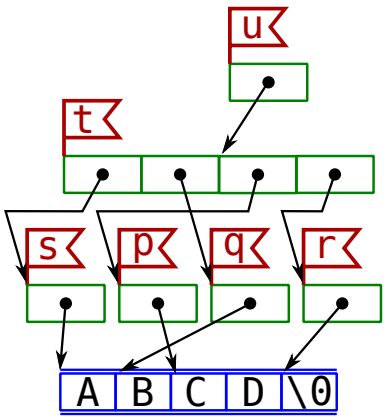
Considérez le code suivant :

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

int compare_pptr(const char*** ptr1, const char*** ptr2)
{
  return **ptr1 > **ptr2;
}

int main(void)
{
  const char* s = "ABCD";
  const char* p = s;
  const char* q = strrchr(s, 'D');
  const char* r = q + 1;
  const char** t[] = { &p, &q, &r, &s };
  const size_t nb_el = 4;
  const char*** const u = t + 2;

  while(--q >= ++p);

  qsort(t, nb_el, sizeof(NULL), compare_pptr);

  printf("%s\n", **u);

  return 0;
}
```

① **[7.5 points] Dessinez** une image illustrative de la mémoire au *début* de ce code (*avant* le `while`).



② **[5.5 points] Dessinez** une *autre* image, illustrative de la mémoire à la *fin* de ce code (au moment du `printf()`).



③ **[2 points]** Qu'affiche ce code ?

```
CD
```

1.4 Concaténation [5 points]

Le code suivant est-il correct ? Si non, corriger toutes les erreurs **en conservant un appel** à `strncat()` ou à `strcat()` (récrivez à droite le code qui doit être corrigé, si nécessaire).

Note : la correction d'une erreur qui n'en est pas une sera pénalisée.

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

int main(void)
{
  char* s1 = "Bonjour ";
  char* s2 = "tout le monde !";
  char* s3 = strncat(s1, s2,
                     strlen(s1) + strlen(s2));

  printf("%s\n", s3);

  return 0;
}
```

Il y a plusieurs choses :

- la plus grave : on ne peut pas modifier `s1` ; il faut donc, d'une façon ou d'une autre, avoir une chaîne modifiable allouée ;
- mineur : `const` à au moins `s2` ;
- mineur : pas besoin du « `n` » ici pour `strcat()` : on connaît déjà les tailles et on les maîtrise. Qui plus est, si on voulait vraiment utiliser le « `n` », l'argument à passer serait alors `strlen(s2)` et non pas la somme des deux !

Voici deux corrigés possibles, le premier étant meilleur (pas de « magic number ») :

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

int main(void)
{
  const char* s1 = "Bonjour ";
  const char* s2 = "tout le monde !";
  char* s3 = calloc(strlen(s1) + strlen(s2) + 1,
                    1);
  strcpy(s3, s1);

  printf("%s\n", strcat(s3, s2));
  free(s3);

  return 0;
}
```

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

int main(void)
{
  char s1[24] = "Bonjour ";
  const char* s2 = "tout le monde !";

  printf("%s\n", strcat(s1, s2));

  return 0;
}
```

suite au dos ☞

Question 2 – Saisie d'une réponse de taille variable [25 points]

Un des problèmes lors des interactions avec l'utilisateur est que l'on ne connaît pas *a priori* la longueur de sa réponse ; et nous n'avons en C aucun moyen simple de lire dynamiquement une telle réponse.
Le but de cet exercice est d'écrire une fonction `ask()` qui pose une question reçue en paramètre et retourne la chaîne de caractères de toute la réponse en entier (une ligne).

Par exemple, dans l'interaction suivante :

```
Que voulez-vous me dire ?
Ceci est une réponse assez longue mais pas totalement incongrue !
```

(où l'utilisateur a terminé sa réponse par un retour à la ligne, qui ne fait pas partie de la réponse), la fonction `ask()` retournera la chaîne de caractères
`"Ceci est une réponse assez longue mais pas totalement incongrue !"`.

L'idée, pour gérer dynamiquement la taille de la réponse lue, est d'adopter une stratégie de croissance exponentielle par doublement. Si l'on suppose, par exemple, commencer par lire 4 caractères, et que la réponse donnée est `"abcdefghijklmno"` (15 caractères), la fonction `ask()` se comportera comme si l'on avait fait la séquence suivante :

```
// s est vide
scanf("%4[^\n]", s); // s contient "abcd" (4 char)
// s est réallouée, s2 pointe sur sa fin
scanf("%8[^\n]", s2); // s contient "abcdefghijkl" (12 char)
// s est réallouée, s2 pointe sur sa fin
scanf("%16[^\n]", s2); // s contient "abcdefghijklmno" (15 char)
// s est réallouée à la vraie taille de la réponse
```

Concrètement, nous vous demandons de définir ci-contre une fonction `ask()` qui :

1. pose la question reçue en paramètre ;
2. alloue une chaîne vide (disons : « `answer` ») pouvant lire jusqu'à 4 caractères de l'entrée standard ;
3. initialise une chaîne « `format` » à la valeur `"%4[^\n]"` ;
4. tant qu'elle arrive à lire (avec `scanf()`) au format donné par `format` :
   — double l'*incrément* de taille allouée à `answer` (4, 8, 16 dans l'exemple ci-dessus) ;
   — met à jour le `format` (expliqué ci-dessous) pour pouvoir lire le bon nombre de *nouveaux* caractères (8, puis 16 dans l'exemple ci-dessus) ;
   — re-boucle (en 4.) pour lire le bon nombre de nouveaux caractères en les ajoutant en *fin* de `answer` ;
5. retourne la chaîne lue, mais réallouée à la vraie taille lue (supprime l'éventuelle allocation excédentaire).

Pour la mise à jour du `format` (qui peut être alloué statiquement, à 26 caractères [2]), on utilisera la fonction standard

```
sprintf(format, "..%..", ...);
```

qui s'utilise exactement comme `printf()`, mais écrit dans `format` (sans se préoccuper de son allocation)

2. Pour celles/ceux que ça intéresse, 26 c'est 1 (`'%'`) +20 (chiffres ; $\log_{10}(2^{64})$) +4 (`"[^\n]"`) +1 (`'\0'`).

au lieu d'écrire dant `stdout`. Par exemple :

`sprintf(format, "%d", i);`

écrit la valeur de `i` dans `format` (qui contiendrait donc p.ex. `"12"` si la valeur de `i` est 12).

```
#define FMT "%%%zu[^\n]"

char* ask(const char* question)
{
  printf("%s ? ", question);

  size_t bunch_size     = 4;          // initial bunch size
  size_t allocated_size = bunch_size; // initial size

  char* answer = calloc(allocated_size + 1, 1); // + 1 for final '\0'
  if (answer == NULL) return NULL;

  char format[26] = ""; // 26 = % + 20 digits (log_10(2**64)) + 4 ("[^\n]") + '\0'
  sprintf(format, FMT, bunch_size);

  char* next_bunch = answer;
  while (1 == scanf(format, next_bunch)) {
    bunch_size *= 2; // exponential growth
    sprintf(format, FMT, bunch_size);
    allocated_size += bunch_size;

    char* new_answer = realloc(answer, allocated_size + 1); // grow
    if (new_answer == NULL) return answer;

    answer = new_answer;
    next_bunch = answer + (allocated_size - bunch_size); // notice that answer might
                                                         // have changed; could also be:
    // next_bunch = answer + strlen(answer);
  }

  return realloc(answer, strlen(answer) + 1); // downsize
}
```

suite au dos ☞

Question 3 – Programmation C : aspirateur Web [29 points]

On s'intéresse ici à un programme pour télécharger des pages Web et récursivement les pages vers lesquelles elles ont des hyperliens (« aspirateur de pages »).

Dans cet exercice, vous allez commencer par créer des structures de données pour modéliser les pages du Web, puis implémenter les fonctions (expliquées plus loin) de recherche de pages référentes et d'aspiration à profondeur fixée.

3.1 Structures de données [6 points]

On s'intéresse ici à modéliser les objets `Web`, `Page` et `Link`.

Le `Web` est un ensemble de `Pages`. Chaque `Page` contient un `titre` (texte) et un ensemble de liens sortants vers d'autres pages (`Links`). Un `Link` est donc une référence à une autre `Page`.

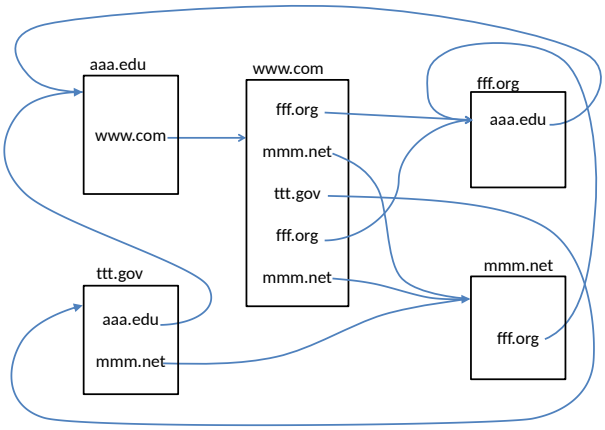
Il sera également utile d'ajouter un champ `visited` de type `unsigned int` pour marquer les pages déjà visitées lors des recherches effectuées (voir plus loin).

Nous vous demandons de définir ici les types C permettant de représenter les objets ci-dessus.

```
typedef struct Page_ Page;
typedef Page* Link;

struct Page_ {
  char* title;
  unsigned int visited;
  size_t nb_links;
  Link* links; // could also be a flexible array
};

typedef struct {
  size_t nb_pages;
  Page** content; /* or even Page* (= array of Pages) if storage
                   * of * pages directly here.
                   * And this could also be a flexible array.
                   * These four conceptions are accepted.        */
} Web;
```

3.2 Pages pointant vers une page [10 points]

Sur la base des structures de données proposées précédemment, écrivez la fonction `incoming_links()` qui prend en paramètre (un pointeur sur) un objet `Web` et (un pointeur sur) une page et affiche la liste de toutes les `Pages` du `Web` qui ont un `Link` vers la page donnée en paramètre.



Par exemple, avec le réseau ci-contre, l'appel de cette fonction sur la page `fff.org` donnerait : `www.com, mmm.net`
puisque ces deux pages pointent vers `fff.org`.
Notez bien que la page `www.com` n'est affichée qu'*une* seule fois (bien que contenant *deux* liens vers `fff.org`). Pour cela, utilisez le champ `visited` des `Pages` pour marquer les pages déjà affichées et ne plus les ré-afficher.

```
void reset_visited(Web* web) {
  for (size_t i = 0; i < web->nb_pages; ++i) {
    web->content[i]->visited = 0;
  }
}

//-------------------------------------------------
void incoming_links(Web* web, const Page* pageref)
{
  printf("Incoming links on %s :\n", pageref->title);
  reset_visited(web);
  for (size_t i = 0; i < web->nb_pages; ++i) {
    Page * const pageref2 = web->content[i];
    for (size_t j = 0; (pageref2->visited == 0) && (j < pageref2->nb_links); ++j) {
      if (pageref2->links[j] == pageref) {
        printf("%s, ", pageref2->title);
        ++(pageref2->visited);
      }
    }
  }
  putchar('\n');
}
```

3.3 Recherche à profondeur donnée [13 points]

Ecrivez maintenant une fonction `wget()` qui affiche toutes les pages qui sont référencées par une page donnée, et ceci récursivement à une profondeur donnée.

Cette fonction aura comme paramètres : (un pointeur sur) un objet `Web`, (un pointeur sur) la page de départ et la profondeur désirée.

Par exemple, sur le réseau jouet de la sous-question précédente, la fonction `wget()` appelée à profondeur 2 sur la page `aaa.edu` donnerait `www.com` (prof. 1), puis `fff.org`, `mmm.net` et `ttt.gov`.

Notez bien qu'aucune page ne doit être affichée plus d'une fois (c.-à-d. aucune répétition).

Indication : pensez à deux fonctions : une fonction principale (`wget()`), et une fonction auxiliaire récursive.

```
void wget_rec(const Web* web, const Page* pageref, unsigned int prof)
{
  if (prof) {
    for (size_t i = 0; i < pageref->nb_links; ++i) {
      if (pageref->links[i]->visited == 0) {
        printf("%s, ", pageref->links[i]->title);
        ++(pageref->links[i]->visited);
        wget_rec(web, pageref->links[i], prof - 1);
      }
    }
    putchar('\n');
  }
}

void wget(Web* web, const Page* pageref, unsigned int prof)
{
  printf("WGET @ prof %u on page %s :\n", prof, pageref->title);
  reset_visited(web);
  wget_rec(web, pageref, prof);
}
```

Question 4 – Analyse des arguments d'un logiciel [39 points]

La plupart des logiciels que vous pouvez exécuter sur votre ordinateur acceptent des arguments quand vous les évoquez sur la ligne de commande. Par exemple, lors d'une commande de compilation, vous spécifiez le nom du fichier à compiler et celui de l'exécutable :

```
gcc -o exemple exemple.c
```

Certains des ces arguments sont obligatoires (p.ex. le nom du fichier à compiler), d'autres optionnels. Par ailleurs, certains des ces arguments attendent un argument supplémentaire (p.ex. `-o` attend le nom du fichier exécutable) et d'autres non (p.ex. `--version` qui affiche la version du compilateur).

Le but de cet exercice est d'écrire *quelques* fonctions outils pour analyser les arguments donnés sur la ligne de commande.

4.1 Structures de données [13 points]

Commençons par définir les structures de données qui nous seront utiles.

4.1.1 Conversion générique [4 points]

Pour être parfaitement générique, il faut tout d'abord un moyen de convertir une chaîne de caractères en une valeur quelconque. Par exemple, si on a un logiciel avec une option « `--age 19` », il faudra convertir la chaîne `"19"` en l'entier (`int`) 19 ; mais si on a une option « `--prix 3.5` », il faudra convertir la chaîne `"3.5"` en `double` 3.5.

Ceci n'est bien sûr pas possible en une seule fonction, mais nous pouvons pour cela faire un modèle de fonction générique que l'on utilisera ensuite pour choisir la bonne fonction à appliquer.

Une telle fonction doit donc recevoir un « pointeur générique » (vers la variable qui recevra la valeur convertie) et une chaîne de caractères. On peut par exemple imaginer avoir deux fonctions `arg2int()` et `arg2double()` que l'on pourrait par exemple appeler comme ceci :

```
err = arg2int(&i, "19");
err = arg2double(&x, "3.5");
```

avec `i` de type `int` et `x` de type `double`.
Ces fonctions de conversion retournent un code d'erreur (de type `int`).

Définissez ici le type `ConvertFunc` comme pointeur vers une telle fonction de conversion générique :

```
typedef int (*ConvertFunc)(void* val, const char* arg);
```

4.1.2 Le type pour un argument [5 points]

Revenons maintenant aux arguments de la ligne de commande du logiciel. Chaque argument peut avoir deux formes :

1. la version courte avec un seul tiret (`-`) ; p.ex. `-h` pour afficher un message d'aide ;
2. la version longue avec deux tirets (p.ex. `--help`).

De plus, un argument peut être un simple « *flag* », sans argument supplémentaire (p.ex., `-h` ou `--version`), ou il peut nécessiter une valeur (p.ex., `-o <nom du fichier>` qui fournit à `gcc` le nom du fichier exécutable ou `--age` comme illustré dans la sous-question précédente). Si c'est le cas, il faut pouvoir indiquer, de façon générique, où stocker et comment convertir cette valeur. Cela se fait avec deux champs : un pointeur générique (vers la variable qui recevra la valeur convertie) et un `ConvertFunc` pour la fonction a utiliser (exemples ci-dessous).

Enfin, un argument de la ligne de commande peut, ou non, stoper l'exécution du logiciel. Par exemple des arguments comme `-h` ou `--version` stopent l'exécution, alors que `-o` ou `--age` ne le feraient pas.

Définissez ci-dessous un type `Argument` pouvant stocker :

- deux chaînes de caractères (une pour la version courte, une pour la version longue de l'argument),
- un « pointeur générique » pour recevoir une valeur ;
- une fonction de conversion (`ConvertFunc`) ;
- ainsi qu'un booléen (ou `char`) qui signifie si cet argument arrête le programme ou non.

Par exemple l'`Argument` pour `-h` contiendrait [3] :

```
"-h", "--help", NULL, NULL, true
```

et celui pour `--age` contiendrait :

```
"-a", "--age", &i, arg2int, false
```

Réponse :

```
typedef struct Argument_ {
    const char *short_arg;
    const char *long_arg;
    void* value;
    ConvertFunc convert;
    bool stop_exec;
} Argument;
```

4.1.3 La liste des arguments possibles [4 points]

Un logiciel doit connaître tous ses arguments possibles. Pour plus de flexibilité, nous souhaitons pouvoir indiquer cela de façon dynamique. Définissez ci-dessous un type `ArgumentList` permettant de stocker un ensemble d'arguments (p.ex. les `Argument` correspondant à `-h`, `-o`, `--version`, etc.). On ne connait pas *a priori* le nombre d'arguments d'un logiciel.
Vous êtes libres de choisir votre implémentation, mais expliquez là brièvement (simples commentaires).

Il s'agit typiquement d'un tableau dynamique, p.ex. :

```
typedef struct ArgumentList_ {
  Argument *args;
  size_t arg_num;       // used
  size_t arg_list_size; // allocated
} ArgumentList;
```

Le champ pour l'allocation « par pages » (`arg_list_size`) est optionnel.

On pourrait aussi faire une liste chaînée (mais ça nous semble moins pratique) ou un arbre binaire de recherche (ordonner la liste ; mais ça nous semble trop compliqué `;-)`).

3. Vous pouvez, bien sûr, adapter ces exemples à vos choix d'implémentation.

Le contenu de ce tableau peut être des `Argument`, comme ici, ou des `const Argument*` qui référenceraient des `Argument` existant par ailleurs (typiquement dans le `main()`).

4.2 Initialisation et libération d'une liste d'arguments [8 points]

4.2.1 Initialisation [5 points]

Créez une fonction `init_argument_list()` qui attend une liste d'arguments (`ArgumentList`), l'initialise et retourne `0` si la liste donnée a bien été initialisée, et `1` sinon.

Voici un exemple avec pré-allocation (modèle d'allocation « par pages ») :

```
int init_argument_list(ArgumentList *list) {
    list->arg_num = 0;
    list->arg_list_size = INCREASE_ARG_LIST_SIZE; // whatever
    list->args = calloc(list->arg_list_size, sizeof(Argument));
    if (list->args == NULL) {
        list->arg_list_size = 0;
        return 1;
    }
    return 0;
}
```

L'initialisation peut aussi être plus simple (p.ex. `NULL`) et l'augmentation de taille gérée lors de l'ajout (question 4.3).

4.2.2 Libération [3 points]

Créez une fonction `deinit_argument_list()` qui attend une liste d'arguments et qui ne retourne rien. Cette fonction doit s'occuper de libérer les (éventuelles) ressources de la liste d'argument donnée (mais ne la supprime pas elle-même).

```
void deinit_argument_list(ArgumentList *list) {
    free(list->args);
    list->args = NULL;
    list->arg_num = 0;
    list->arg_list_size = 0;
}
```

Note : celles/ceux chez qui cela fait sens, on peut simplement appeler `init_argument_list()` après le `free()`.

4.3 Ajout d'arguments à la liste des arguments acceptés [9 points]

On s'intéresse maintenant à créer une fonction `add_argument()` permettant d'ajouter des `Argument` à une `ArgumentList` (que l'on suppose avoir été initialisée).
Cette fonction doit recevoir une liste d'arguments à laquelle ajouter un argument, et l'argument qui sera ajouté. Elle retourne `0` si l'ajout a pu se faire correctement, et `1` sinon.

Définissez ici la fonction `add_argument()` :

```
int add_argument(ArgumentList *list, const Argument *arg) {
    if (list->arg_num >= list->arg_list_size) {
        // Need to increase list size

        const size_t new_size = list->arg_list_size + INCREASE_ARG_LIST_SIZE;
        // optional: check overflow (twice: + and *)
        Argument * const new_ptr = realloc(list->args, sizeof(*arg) * new_size);
        if (new_ptr == NULL) {
            return 1;
        }
        list->args          = new_ptr ;
        list->arg_list_size = new_size;
    }

    list->args[list->arg_num++] = *arg;
    return 0;
}
```

4.4 Traiter un argument [9 points]

Supposons maintenant que l'on ait trouvé (dans `argv`) un argument qui corresponde. Par exemple que `argv` pointe sur[4] :

```
"--age", "19", NULL
```

et que l'on ait trouvé que cela correspond à l'`Argument arg`.

On souhaite alors traiter l'argument en appelant une fonction `do_arg()` qui prend en paramètre `argv` (bien placé, comme ci-dessus) et l'`Argument` correspondant (par référence constante) et qui retourne un code d'erreur. Dans l'exemple précédent, on ferait l'appel

```
err = do_arg(&argv, &arg);
```

Cette fonction `do_arg()` doit :

- faire appel à `convert` de `arg` s'il n'est pas `NULL` ;
  cet appel doit se faire avec un `argv` valide, sinon `do_arg()` retourne `2` ;
- retourner `1` si l'argument doit stoper l'exécution et sinon retourner le résultat de l'appel à `convert` ou `0` s'il n'y a pas eu d'appel (ou `2` si l'appel ne peut être fait).

Après l'appel, `argv` pointe sur le *prochain* argument de la ligne de commande.

Par exemple :

- si `argv` pointe sur `"--age", "19", "--prix", "3.5", NULL`, `do_arg()` retournera le résultat de l'appel à `arg2int()` et `argv` pointera alors sur `"--prix", "3.5", NULL` ;
- si `argv` pointe sur `"--age", NULL`, `do_arg()` retournera `2` et `argv` pointera alors sur `NULL` ;
- si `argv` pointe sur `"--help", "--age", "19", NULL`, `do_arg()` retournera `1` car `"--help"` doit stoper le programme, et `argv` pointera alors sur `"--age", "19", NULL`.

Définissez ici la fonction `do_arg()` :

```
int do_arg(char*** argv, const Argument* arg)
{
  int retour = 0;
  if (arg->convert != NULL) {
    if (arg->value != NULL) {
      ++(*argv);
      if (**argv == NULL) return 2;
    }
    retour = arg->convert(arg->value, **argv);
    if (retour == 0) ++(*argv);
  }
  if (arg->stop_exec) retour = 1;
  return retour;
}
```

4. On rappelle que le tableau `argv` reçu au départ par `main()` se termine par le pointeur `NULL`.