C avancé

Piles

18 mars 2022

Version 10

Fabrice BOISSIER <fabrice.boissier@epita.fr>
Mark ANGOUSTURES <mark.angoustures@epita.fr>

Copyright

Table des matières

1 Consignes Générales

Les informations suivantes sont très importantes :

Le non-respect d'une des consignes suivantes entraînera des sanctions pouvant aller jusqu'à la multiplication de la note finale par 0.

Ces consignes sont claires, non-ambiguës, et ont un objectif précis. En outre, elles ne sont pas négociables.

N'hésitez pas à demander si vous ne comprenez pas une des règles.

Consigne Générale 0 : Vous devez lire le sujet.
Consigne Générale 1 : Vous devez respecter les consignes.
Consigne Générale 2 : Vous devez rendre le travail dans les délais prévus.

Consigne Générale 3 : Le travail doit être rendu dans le format décrit à la section Format de Rendu.

Consigne Générale 4 : Le travail rendu ne doit pas contenir de fichiers binaires, temporaires, ou d'erreurs (`*~`, `*.o`, `*.a`, `*.so`, `*#*`, `*core`, `*.log`, `*.exe`, binaires, ...).

Consigne Générale 5 : Dans l'ensemble de ce document, la casse (caractères majuscules et minuscules) est très importante. Vous devez strictement respecter les majuscules et minuscules imposées dans les messages et noms de fichiers du sujet.

Consigne Générale 6 : Dans l'ensemble de ce document, `login.x` correspond à votre login.

Consigne Générale 7 : Dans l'ensemble de ce document, `nom1-nom2` correspond à la combinaison des deux noms de votre binôme (par exemple pour Fabrice BOISSIER et Mark ANGOUSTURES, cela donnera `boissier-angoustures`).

Consigne Générale 8 : Dans l'ensemble de ce document, le caractère ␣ correspond à une espace (s'il vous est demandé d'afficher ␣␣␣, vous devez afficher trois espaces consécutives).

Consigne Générale 9 : Tout retard, même d'une seconde, entraîne la note non négociable de 0.

Consigne Générale 10 : La triche (échange de code, copie de code ou de texte, ...) entraîne **au mieux** la note non négociable de 0.

Consigne Générale 11 : En cas de problème avec le projet, vous devez contacter le plus tôt possible les responsables du sujet aux adresses mail indiquées.

Conseil : N'attendez pas la dernière minute pour commencer à travailler sur le sujet.

2 Format de Rendu

Responsable(s) du projet :	**Fabrice BOISSIER** `<fabrice.boissier@epita.fr>`
Balise(s) du projet :	**`[CAV][TP1+]`**
Nombre d'étudiant(s) par rendu :	1
Procédure de rendu :	Envoi par mail
Nom du répertoire :	`login.x-TP1`
Nom de l'archive :	`login.x-TP1.tar.bz2`
Date maximale de rendu :	14/04/2022 23h42
Durée du projet :	1 semaine
Architecture/OS :	Linux - Ubuntu (x86_64)
Langage(s) :	C
Compilateur/Interpréteur :	**`/usr/bin/gcc`**
Options du compilateur/interpréteur :	**`-W -Wall -Werror -std=c99 -pedantic`**

Les fichiers suivants sont requis :

`AUTHORS`	contient le(s) nom(s) et prénom(s) de(s) auteur(s).
`Makefile`	le Makefile principal.
`README`	contient la description du projet et des exercices, ainsi que la façon d'utiliser le projet.

Un fichier **`Makefile`** doit être présent à la racine du dossier, et doit obligatoirement proposer ces règles :

`all`	*[Première règle]* lance la règle `libmystack`.
`clean`	supprime tous les fichiers temporaires et ceux créés par le compilateur.
`dist`	crée une archive propre, valide, et répondant aux exigences de rendu.
`distclean`	lance la règle `clean`, puis supprime les binaires et bibliothèques.
`check`	lance le(s) script(s) de test.
`libmystack`	lance les règles `shared` et `static`
`shared`	compile l'ensemble du projet avec les options de compilations exigées et génère une bibliothèque dynamique.
`static`	compile l'ensemble du projet avec les options de compilations exigées et génère une bibliothèque statique.

Votre code sera testé automatiquement, vous devez donc scrupuleusement respecter les spécifications pour pouvoir obtenir des points en validant les exercices. Votre code sera testé en générant les exécutables ou bibliothèques avec les commandes suivantes :

```
./configure
make
```

Suite à cette étape de génération, les exécutables ou bibliothèques doivent être placés à ces endroits :

login.x-TP1/libmystack.a
login.x-TP1/libmystack.so

L'arborescence attendue pour le projet est la suivante :

login.x-TP1/
login.x-TP1/AUTHORS
login.x-TP1/README
login.x-TP1/Makefile
login.x-TP1/configure
login.x-TP1/check/
login.x-TP1/check/check.sh
login.x-TP1/src/
login.x-TP1/src/stack_array.c
login.x-TP1/src/stack_array.h
login.x-TP1/src/stack_linked_list.c
login.x-TP1/src/stack_linked_list.h
login.x-TP1/src/stack_static.c
login.x-TP1/src/stack_static.h

Vous ne serez jamais pénalisés pour la présence de makefiles ou de fichiers sources (code et/ou headers) dans les différents dossiers du projet tant que leur existence peut être justifiée (des makefiles vides ou jamais utilisés sont pénalisés).
Vous ne serez jamais pénalisés pour la présence de fichiers de différentes natures dans le dossier `check` tant que leur existence peut être justifiée (des fichiers de test jamais utilisés sont pénalisés).

3 Aide Mémoire

Le travail doit être rendu au format ***.tar.bz2***, c'est-à-dire une archive **bz2** compressée avec un outil adapté (voir **`man 1 tar`** et **`man 1 bz2`**).
Tout autre format d'archive (zip, rar, 7zip, gz, gzip, ...) ne sera pas pris en compte, et votre travail ne sera pas corrigé (entraînant la note de 0).

Pour générer une archive *tar* en y mettant les dossiers *folder1* et *folder2*, vous devez taper :

```
tar cvf MyTarball.tar folder1 folder2
```

Pour générer une archive *tar* et la compresser avec GZip, vous devez taper :

```
tar cvzf MyTarball.tar.gz folder1 folder2
```

Pour générer une archive *tar* et la compresser avec BZip2, vous devez taper :

```
tar cvjf MyTarball.tar.bz2 folder1 folder2
```

Pour lister le contenu d'une archive *tar*, vous devez taper :

```
tar tf MyTarball.tar.bz2
```

Pour extraire le contenu d'une archive *tar*, vous devez taper :

```
tar xvf MyTarball.tar.bz2
```

Pour générer des exécutables avec les symboles de debug, vous devez utiliser les flags **`-g -ggdb`** avec le compilateur. N'oubliez pas d'appliquer ces flags sur *l'ensemble* des fichiers sources transformés en fichiers objets, et d'éventuellement utiliser les bibliothèques compilées en mode debug.

```
gcc -g -ggdb -c file1.c file2.c
```

Pour produire des exécutables avec les symboles de debug, il est conseillé de fournir un script **`configure`** prenant en paramètre une option permettant d'ajouter ces flags aux **`CFLAGS`** habituels.

```
./configure
cat Makefile.rules
CFLAGS=-W -Wall -Werror -std=c99 -pedantic

./configure debug
cat Makefile.rules
CFLAGS=-W -Wall -Werror -std=c99 -pedantic -g -ggdb
```

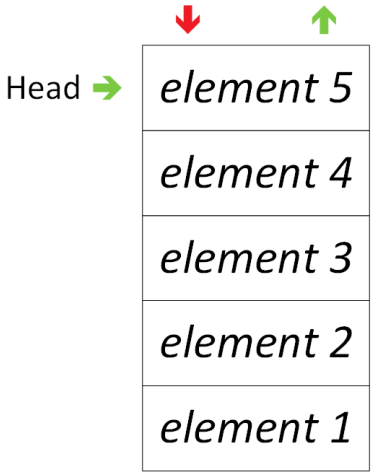
Dans ce sujet précis, vous ferez du code en C et des appels à des scripts shell qui afficheront les résultats dans le terminal (donc des flux de sortie qui pourront être redirigés vers un fichier texte).

4 Cours

Notions étudiées : Tableaux, Pointeurs, Piles

4.1 Rappel sur les piles

Les **piles**, ou **stacks** en anglais, sont des structures visant à stocker les données dans l'ordre d'arrivée, mais ne permettant leur récupération uniquement dans l'ordre inverse. Dans une pile, on ne peut accéder qu'à la dernière donnée stockée, celle se situant au *sommet* de la pile. Ces structures sont aussi appelées **LIFO** (*Last In First Out*).

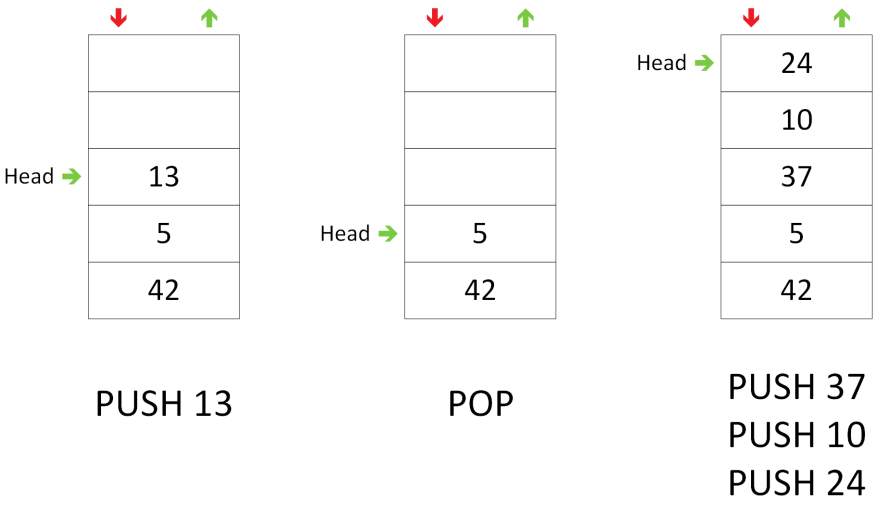


Deux opérations permettent d'utiliser une pile :

- **`PUSH`** : permettant d'*empiler* une donnée supplémentaire dans la pile
- **`POP`** : permettant de *dépiler* une donnée depuis la pile

On ajoute donc une donnée en l'empilant avec un **`PUSH`**, et il est possible de directement y accéder, car elle est au sommet de la pile. À l'inverse, pour accéder à une donnée tout au fond de la pile, il est nécessaire de dépiler autant d'éléments que nécessaire avec un **`POP`**.

Voici un exemple où l'on crée une pile, puis on empile successivement 42, 5, et 13, puis, on dépile une fois (pour récupérer 13), et enfin, on empile successivement 37, 10, 24.

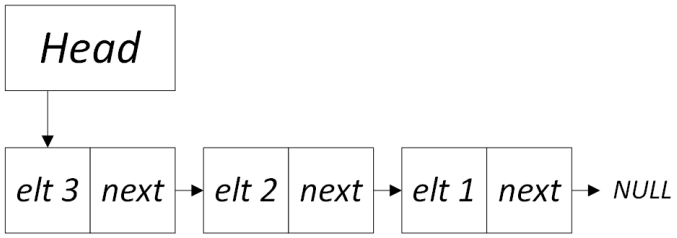


Les piles, et surtout la contrainte d'accès aux objets, sont couramment utilisées : un camion de livraison sera d'abord rempli avec les paquets à livrer en dernier/le camion sera rempli dans l'ordre inverse de livraison (on accède d'abord aux derniers éléments chargés).
En informatique, on utilisera les piles dans certains *parsers* (analyse grammaticale) pour connaître en premier l'opérateur à exécuter (opérateur binaire ? unaire ?) et dépiler par la suite le nombre exact de paramètres. La *pile d'appels* est également une convention fondamentale partagée par les processeurs et les systèmes d'exploitation permettant de passer des paramètres (et d'autres informations de contexte) aux fonctions appelées par les programmes, ou en cas d'interruption pour sauvegarder l'adresse de l'instruction qui était en cours d'exécution.

Afin d'implémenter une pile, il est donc nécessaire d'avoir un espace de stockage ordonné (un tableau numéroté ou une liste chaînée), et un indicateur de l'élément en haut de la pile. Nous allons maintenant voir comment implémenter une pile avec des listes chaînées et un tableau de taille fixe.

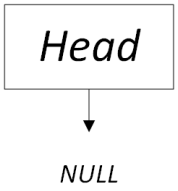
4.2 Piles : implémentation avec des listes chaînées

Une implémentation à l'aide d'une liste chaînée permet d'exploiter la mémoire et d'être donc beaucoup plus flexible en terme de nombre maximum d'éléments. Le schéma suivant illustre une pile sous forme de liste chaînée en mémoire :

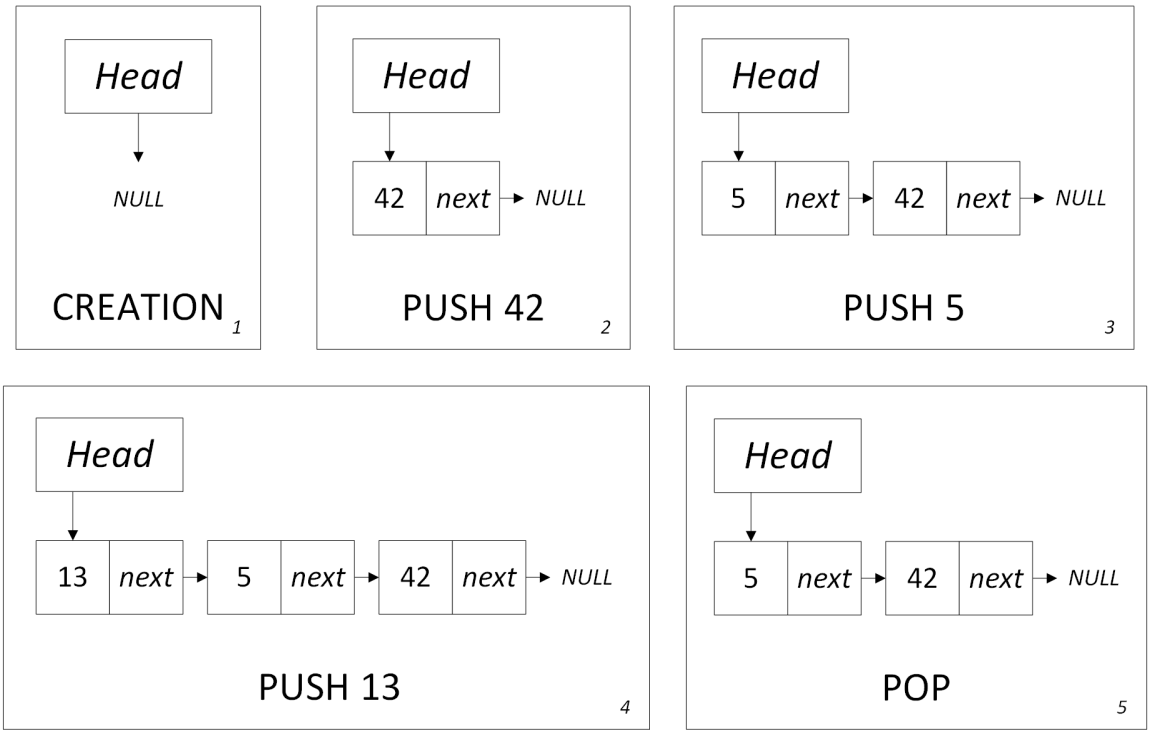


On y retrouve plusieurs fois la structure typique des listes chaînées (un élément et un pointeur vers l'élément suivant), ainsi qu'un pointeur indiquant le sommet de la pile (*head* dans notre cas).

L'unique cas particulier concerne une pile vide : le pointeur de sommet vaut dans ce cas **`NULL`**. Il s'agit également de l'état dans lequel se trouve une pile vidée ou nouvellement créée.



L'exemple suivant montre l'évolution d'une pile au fur et à mesure des ajouts (empiler / **`PUSH`**) et suppressions (dépiler / **`POP`**).

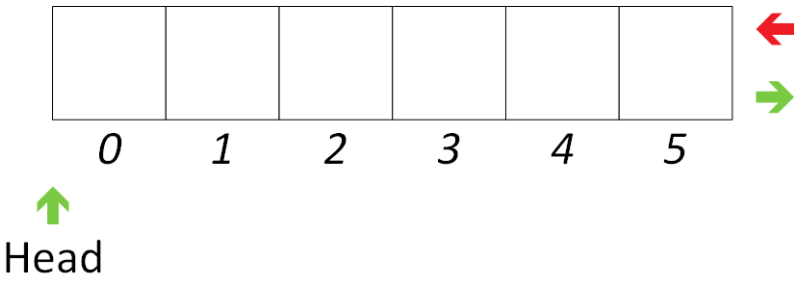


Les principales opérations se résument ainsi :

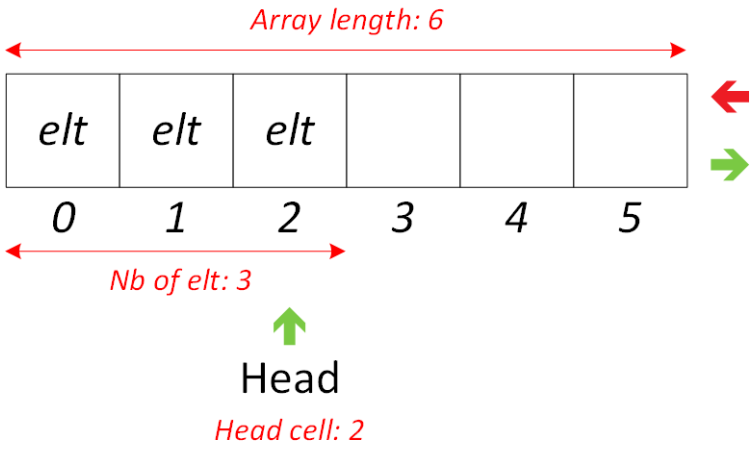
- Création : on alloue en mémoire la structure générale de la pile, et on fixe le sommet de la pile à **`NULL`**.
- Empiler : on alloue en mémoire un nouvel élément dont le pointeur *next* pointe vers l'actuel élément au sommet de la pile, puis, on met à jour le pointeur de sommet de la pile vers l'adresse de ce nouvel élément.
- Dépiler : si la pile est vide, on retourne une erreur, sinon, on récupère tout d'abord l'adresse de l'élément suivant celui au sommet, puis, on libère l'élément au sommet, puis, on met à jour le pointeur de sommet de la pile vers l'adresse de l'élément suivant.
- Vider : on dépile successivement tous les éléments jusqu'à obtenir un sommet à **`NULL`**.
- Sommet : on renvoie le contenu de l'élément au sommet de la pile.

4.3 Piles : implémentation avec un tableau de taille fixe

Une implémentation avec un tableau de taille fixe impose cette fois une limitation : la pile aura une taille maximale, et on peut refuser l'ajout d'un élément si la pile est déjà pleine. La structure diffère également du fait que le tableau est alloué une seule fois lors de sa création (voire même lors de la compilation dans le cas statique).
Le schéma suivant présente la structure générale :



On notera cette fois que plusieurs informations distinctes doivent être conservées : l'adresse du tableau, le numéro de case correspondant au sommet de la pile (*head* dans notre cas), la taille du tableau (le nombre maximum d'objets pouvant être stockés), le nombre d'éléments dans le tableau.
Le schéma suivant détaille certaines informations de façon plus explicite :

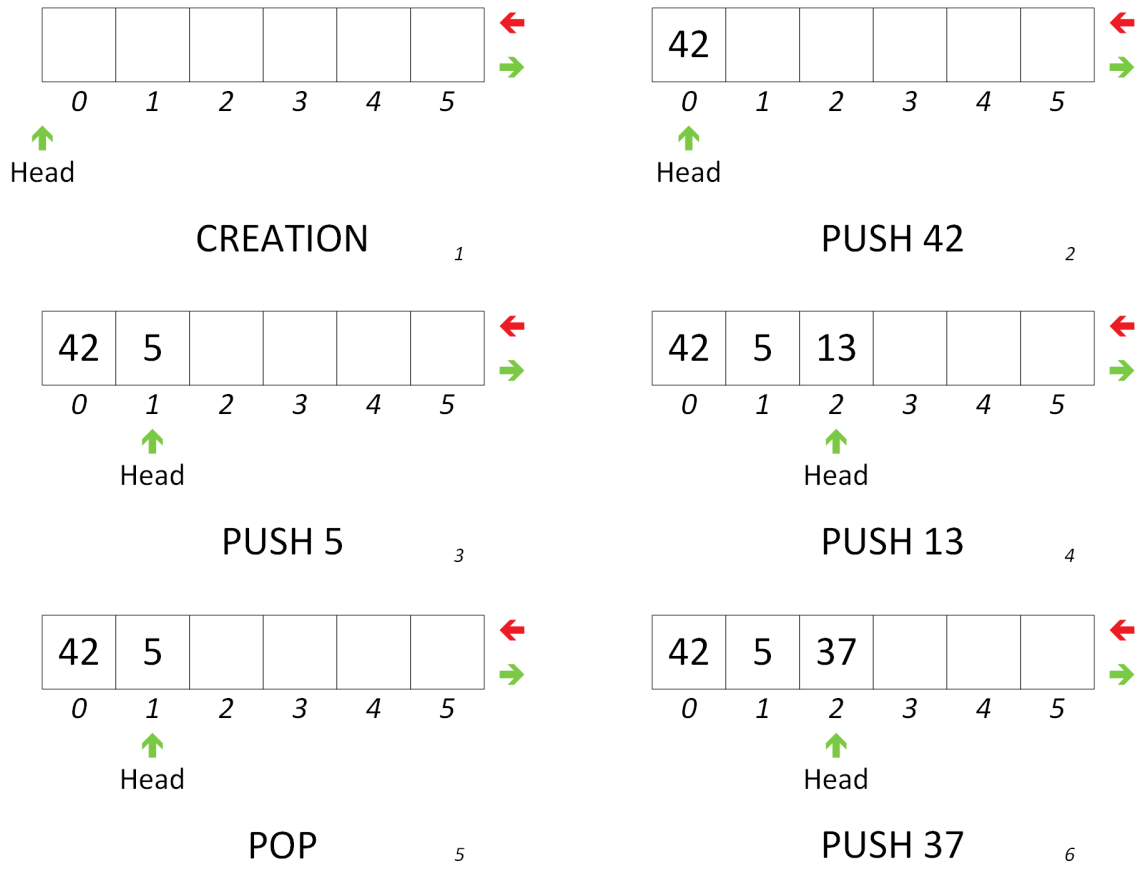


Dans le cas d'un tableau de taille fixe, le pointeur de sommet ne peut pas utiliser la valeur **`NULL`** comme indicateur de tableau vide, car cette valeur est égale à 0 (ce qui laisserait à penser que le sommet est effectivement à la case 0). Plusieurs solutions sont possibles pour indiquer le sommet de la pile et le cas vide :

- On enregistre dans la structure de la pile une variable servant à compter le nombre d'éléments présents (le sommet peut donc prendre n'importe quelle valeur tant que la pile est vide).
- On utilise un entier relatif pour indiquer le sommet, et -1 indique que la pile est vide (l'ajout d'un élément décalera le sommet à 0, c'est-à-dire la case où sera l'élément).

— On place le sommet de la pile sur la première case non utilisée, et l'accès au premier élément se fait donc en retirant 1 au pointeur de sommet (ainsi, un sommet à la case 0 indique que la pile est vide). Attention : dans ce cas précis, un tableau plein aura un sommet hors des cases du tableau (il ne faudra donc *jamais* le déréférencer s'il atteinte une telle valeur).

L'exemple suivant montre l'évolution d'une pile implémentée avec un tableau fixe au fur et à mesure des ajouts (empiler / **`PUSH`**) et suppressions (dépiler / **`POP`**).



Les principales opérations se résument ainsi :

— Création : on alloue en mémoire le tableau (sauf s'il est statique), et on fixe le sommet de la pile à la valeur prévue pour démarrer (-1, 0, ou toute autre valeur choisie) [éventuellement, on met à jour le nombre d'objets dans le tableau en le fixant à 0].

— Empiler : si le tableau est plein, on retourne une erreur, sinon, on ajoute un élément, et on décale le sommet de la pile [éventuellement, on met à jour le nombre d'objets dans le tableau].

— Dépiler : si la pile est vide, on retourne une erreur, sinon, on réduit la valeur du sommet de la pile [éventuellement, on met à jour le nombre d'objets dans le tableau].

— Vider : on fixe le sommet de la pile à la valeur prévue pour démarrer (-1, 0, ou toute autre valeur choisie) [éventuellement, on met à jour le nombre d'objets dans le tableau en le fixant à 0].

— Sommet : on renvoie le dernier élément ajouté (cela dépend de comment le sommet a été implémenté !).

5 Exercice 1 - Bibliothèques statique et dynamique

Nom du(es) fichier(s) :	**`Makefile`**
Répertoire :	**`login.x-TP1/`**
Droits sur le répertoire :	750
Droits sur le(s) fichier(s) :	640
Outils recommandés :	**`gcc(1)`**, **`ar(1)`**

Objectif : Le but de l'exercice est de faire fonctionner l'ensemble de votre projet avec un makefile simple, et de produire une bibliothèque statique ainsi qu'une dynamique.

Une bibliothèque est un ensemble de fonctions et procédures prêtes à être utilisées (ayant éventuellement des variables statiques, ou encore faisant appel à `malloc(3)`) par un utilisateur ne connaissant pas leur implémentation. L'utilisateur développe et écrit son propre programme (utilisant un `main` pour s'exécuter) et il peut éventuellement s'appuyer sur des bibliothèques (n'ayant pas de `main`) pour réutiliser des implémentations existantes.

Les exercices suivants vont vous conduire à produire des fonctions manipulant des structures, afin d'offrir un service à des utilisateurs : une pile et son interface pour l'exploiter (une `API`). Vous devez donc écrire le (ou les) *makefile(s)* de votre projet et un fichier `configure` (éventuellement vide) afin de produire une bibliothèque statique nommée **`libmystack.a`** et une bibliothèque dynamique nommée **`libmystack.so`**.

Le *makefile* que vous allez écrire rendra votre projet complet et autonome. Pour cela, vous devez faire en sorte que plusieurs *cibles* soient présentent dans le makefile (celles-ci sont rappelées dans le paragraphe suivant).

Plusieurs stratégies existent pour compiler avec des makefiles, l'une d'entre elle consiste à placer un makefile par dossier afin que le makefile principal appelle les suivants avec la bonne règle (par exemple : le makefile principal va appeler le makefile du dossier *src* pour compiler le projet, mais le makefile principal appellera celui du dossier *check* lorsque l'on demandera d'exécuter la suite de tests).
Pour ce premier contact avec les makefiles, vous pouvez vous contenter d'un seul makefile à la racine exécutant des lignes de compilation toutes prêtes sans aucune réécriture ou extension.

Afin de générer des bibliothèques statiques et dynamiques, vous devez compiler vos fichiers pour produire des fichiers *objets* (des fichiers **`.o`**), mais vous ne devez pas laisser l'étape d'*édition de liens* classique se faire. À la place, la cible *static* de votre makefile devra produire une bibliothèque statique nommée **`libmystack.a`**, et la cible *shared* devra produire une bibliothèque dynamique nommée **`libmystack.so`**.

Pour générer une bibliothèque statique (*static library* en anglais), on utilise la commande **`ar`** qui créée des *archives*. Pour produire la bibliothèque statique *libtest.a* à partir des fichiers *test1.c* et *file.c*, on utilisera ces commandes :

```
cc -c test1.c file.c
ar cr libtest.a test1.o file.o
```

Pour générer une bibliothèque dynamique (*shared library* en anglais), on utilise l'option **-shared** du compilateur. Pour produire la bibliothèque dynamique *libtest.so* à partir des fichiers *test1.c* et *file.c*, on utilisera ces commandes :

```
cc -c test1.c file.c
cc test1.o file.o -shared -o libtest.so
```

La première commande génère des fichiers objets, et la deuxième les réunit dans un seul fichier. Il arrivera sur certains systèmes qu'il soit nécessaire d'ajouter l'option **-fpic** ou **-fPIC** (*Position Independent Code*) pour générer des bibliothèques dynamiques. Gardez en tête que *toutes* les bibliothèques **doivent** être préfixées par un *lib* dans le nom des fichiers **.a** et **.so** générés. Mais, lorsque l'on utiliser une bibliothèque dans un projet, on doit ignorer ce préfixe lors de la compilation et l'édition de liens.

Pour utiliser une bibliothèque dynamique sur votre système, plusieurs méthodes existent selon le système où vous vous trouvez.

— L'une d'entre elle consiste à ajouter le dossier contenant la bibliothèque à la variable d'environnement **LD_LIBRARY_PATH** (comme pour la variable **PATH**). Selon votre système, il peut s'agir de la variable d'environnement **DYLD_LIBRARY_PATH**, **LIBPATH**, ou encore **SHLIB_PATH**. Pour ajouter le dossier courant comme dossier contenant des bibliothèques dynamiques en plus d'autres dossiers, vous pouvez taper :
export LD_LIBRARY_PATH=.:/usr/lib:/usr/local/lib

— Une autre méthode consiste à modifier le fichier **/etc/ls.so.conf** et/ou ajouter un fichier contenant le chemin vers votre bibliothèque dans **/etc/ld.so.conf.d/** puis à exécuter **/sbin/ldconfig** pour recharger les dossiers à utiliser.

— Enfin, vous pouvez demander les droits administrateur et installer votre bibliothèque dans **/usr/lib** ou **/usr/local/lib**.

Évidemment, à long terme, l'objectif est d'avoir une bibliothèque installée dans le répertoire **/usr/lib**. Mais, lors du développement, il est préférable d'utiliser la variable **LD_LIBRARY_PATH**. Pour vous assurer du fonctionnement de votre exécutable, vous pouvez utiliser **ldd(1)** avec le chemin complet vers un exécutable. **ldd** vous indiquera quelles bibliothèques sont requises, et où celui-ci les trouve. Si l'une d'entre elle n'est pas trouvée, **ldd** vous en informera :

```
ldd /usr/bin/ls
ldd my_main
```

Pour rappel voici les cibles demandées pour le **Makefile** principal à la racine de votre projet :

`all`	*[Première règle]* lance la règle `libmystack`.
`clean`	supprime tous les fichiers temporaires et ceux créés par le compilateur.
`dist`	crée une archive propre, valide, et répondant aux exigences de rendu.
`distclean`	lance la règle `clean`, puis supprime les binaires et bibliothèques.
`check`	lance le(s) script(s) de test.
`libmystack`	lance les règles `shared` et `static`
`shared`	compile l'ensemble du projet avec les options de compilations exigées et génère une bibliothèque dynamique.
`static`	compile l'ensemble du projet avec les options de compilations exigées et génère une bibliothèque statique.

6 Exercice 2 - Pile avec liste chaînée

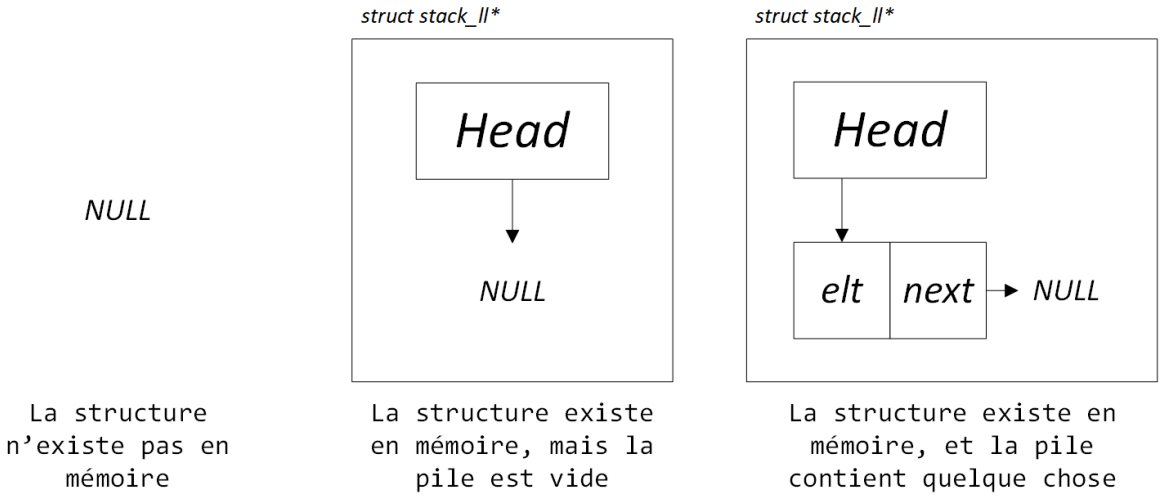
Nom du(es) fichier(s) :	**stack_linked_list.c**
Répertoire :	**login.x-TP1/src/**
Droits sur le répertoire :	750
Droits sur le(s) fichier(s) :	640
Fonctions autorisées :	**malloc(3)**, **free(3)**, **memcpy(3)**, **printf(3)**

Objectif : Le but de l'exercice est d'implémenter une pile en C à base de liste chaînée.

Les fonctions demandées dans cet exercice devront se trouver dans une bibliothèque nommée **libmystack**. Après un appel à la commande `make` à la racine du projet, il faut que votre chaîne de compilation produise à la racine de votre projet une version statique de la bibliothèque (qui se nommera **libmystack.a**) ainsi qu'une version dynamique de la bibliothèque (qui se nommera **libmystack.so**).

Vous devez écrire plusieurs fonctions permettant de créer, utiliser, vider, et libérer une pile. Un fichier **stack_linked_list.h** contenant toutes les fonctions exportables à implémenter vous est fourni en annexe. Vous devez déclarer une structure **stack_ll** et l'ajouter dans **stack_linked_list.h**. N'oubliez pas de déclarer également une structure qui contiendra les éléments de la liste chaînée. Pour les premières étapes, vous devrez implémenter une version simplifiée de la pile qui ne prend en charge que des entiers positifs.

Conceptuellement, les fonctions manipulant des piles de type **stack_ll*** devront pouvoir gérer ces 3 cas :



Vous devez implémenter les fonctions suivantes :

```
stack_ll *stack_ll_create(void);
void stack_ll_delete(stack_ll *stack);

int stack_ll_length(stack_ll *stack);

int stack_ll_push(int elt, stack_ll *stack);
int stack_ll_pop(stack_ll *stack);
int stack_ll_head(stack_ll *stack);

int stack_ll_clear(stack_ll *stack);
int stack_ll_is_empty(stack_ll *stack);

int stack_ll_search(int elt, stack_ll *stack);
stack_ll *stack_ll_reverse(stack_ll *stack);
void stack_ll_print(stack_ll *stack);
```

Liste des fonctions pour une pile avec liste chaînée

`stack_ll *stack_ll_create(void)`

Cette fonction crée une pile vide. En cas d'erreur (pas assez de mémoire), elle renvoie un pointeur **`NULL`**.

`void stack_ll_delete(stack_ll *stack)`

Cette fonction vide une pile de l'ensemble de ses éléments, et détruit la structure restante. Si le paramètre donné est **`NULL`**, la fonction ne fait rien.

`int stack_ll_length(stack_ll *stack)`

Cette fonction renvoie la longueur de la pile (c'est-à-dire le nombre d'éléments actuellement dans la pile). Si le paramètre donné est **`NULL`**, la fonction renvoie -1.

`int stack_ll_push(int elt, stack_ll *stack)`

Cette fonction empile un élément dans une pile, c'est-à-dire qu'elle ajoute un élément au sommet. En cas de succès, la fonction renvoie 0. Si la pile donnée en paramètre est **`NULL`**, la fonction renvoie -1. Si le nombre donné en paramètre est inférieur à 0, la fonction renvoie -4. S'il y a un problème de mémoire, la fonction renvoie -3.

`int stack_ll_pop(stack_ll *stack)`

Cette fonction dépile un élément d'une pile, c'est-à-dire qu'elle supprime l'élément au sommet. En cas de succès, la fonction renvoie 0. Si la pile donnée en paramètre est **`NULL`**, la fonction renvoie -1. Si la pile donnée en paramètre est vide, la fonction renvoie -2.

`int stack_ll_head(stack_ll *stack)`

Cette fonction renvoie l'élément au sommet de la pile. Si la pile donnée en paramètre est **`NULL`**, la fonction renvoie -1. Si la pile donnée en paramètre est vide, la fonction renvoie -2.

`int stack_ll_clear(stack_ll *stack)`

Cette fonction vide une pile de l'ensemble de ses éléments, sans détruire la structure de la pile. La fonction renvoie le nombre d'éléments supprimés de la mémoire. Si le paramètre donné est **`NULL`**, la fonction renvoie -1. Si la pile donnée en paramètre est vide, la fonction renvoie 0.

`int stack_ll_is_empty(stack_ll *stack)`

Cette fonction teste si une pile est vide ou non. Si la pile est vide, la fonction renvoie 1. Si la pile n'est pas vide, la fonction renvoie 0. Si la pile donnée en paramètre est **`NULL`**, la fonction renvoie -1.

`int stack_ll_search(int elt, stack_ll *stack)`

Cette fonction recherche un élément dans la pile et renvoie sa position dans la liste chaînée. La première position est celle où l'élément le plus ancien a été placé (c'est-à-dire le fond de la pile), cette position sera numérotée 0. Si l'élément n'est pas trouvé, la fonction renvoie -4. Si la pile donnée en paramètre est **`NULL`**, la fonction renvoie -1.

`stack_ll *stack_ll_reverse(stack_ll *stack)`

Cette fonction inverse la position de tous les éléments de la pile. Le premier élément devient le dernier, l'avant dernier devient le deuxième, etc. En cas de succès, la fonction renvoie le pointeur vers l'éventuelle nouvelle adresse en mémoire de la structure de la pile inversée. En cas de problème mémoire, on renvoie **`NULL`**, et l'ancienne pile doit rester à son ancienne adresse mémoire sans subir la moindre modification. Si la pile donnée en paramètre est **`NULL`**, la fonction renvoie **`NULL`**.

`void stack_ll_print(stack_ll *stack)`

Cette fonction affiche le contenu de la pile. Le format d'affichage attendu implique d'afficher un seul élément par ligne, suivi d'un retour à la ligne. L'élément en tête de pile sera affiché en premier. Si la pile donnée en paramètre est vide, seul un retour à la ligne est affiché. Si la pile donnée en paramètre est **`NULL`**, rien n'est affiché.

```
$ ./my_stack_linked_list
42
5
13

$
```

Exemple d'affichage du cas normal : pile contenant 42, 5, 13

```
$ ./my_stack_linked_list

$
```

Exemple d'affichage d'une pile vide

```
$ ./my_stack_linked_list
$
```

Exemple d'affichage d'un pointeur NULL

7 Exercice 3 - Pile avec tableau

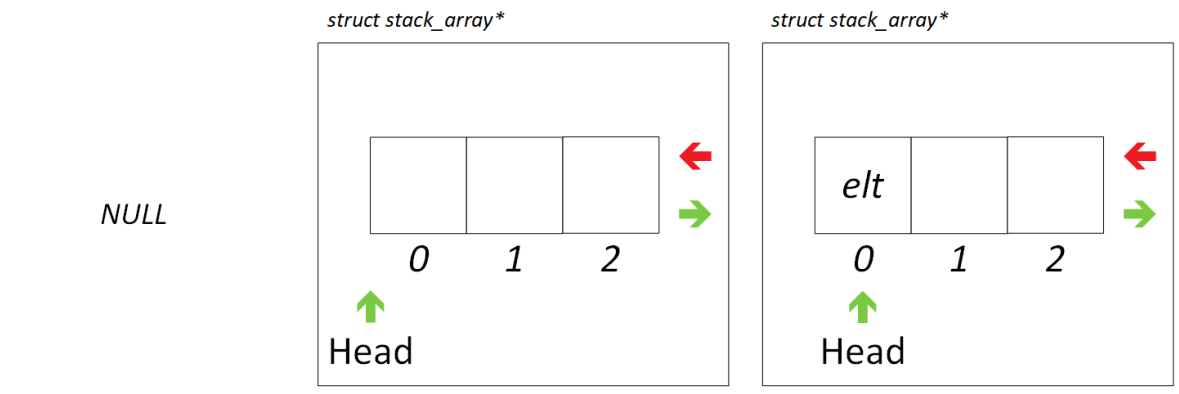
Nom du(es) fichier(s) :	**`stack_array.c`**
Répertoire :	**`login.x-TP1/src/`**
Droits sur le répertoire :	750
Droits sur le(s) fichier(s) :	640
Fonctions autorisées :	**`malloc(3)`**, **`free(3)`**, **`memcpy(3)`**, **`printf(3)`**

Objectif : Le but de l'exercice est d'implémenter une pile en C utilisant un tableau de taille fixe.

Les fonctions demandées dans cet exercice devront également se trouver dans la bibliothèque nommée **`libmystack`**.

Vous devez écrire plusieurs fonctions permettant de créer, utiliser, vider, et libérer une pile. Un fichier **`stack_array.h`** contenant toutes les fonctions exportables à implémenter vous est fourni en annexe. Vous devez déclarer une structure **`stack_array`** et l'ajouter dans **`stack_array.h`**. Pour les premières étapes, vous devrez implémenter une version simplifiée de la pile qui ne prend en charge que des entiers positifs.

Conceptuellement, les fonctions manipulant des piles de type **`stack_array*`** devront pouvoir gérer ces 3 cas :



Vous devez implémenter les fonctions suivantes :

```
stack_array *stack_array_create(int max_length);
void stack_array_delete(stack_array *stack);

int stack_array_length(stack_array *stack);
int stack_array_max_length(stack_array *stack);
```

```
int stack_array_push(int elt, stack_array *stack);
int stack_array_pop(stack_array *stack);
int stack_array_head(stack_array *stack);

int stack_array_clear(stack_array *stack);
int stack_array_is_empty(stack_array *stack);

int stack_array_search(int elt, stack_array *stack);
stack_array *stack_array_reverse(stack_array *stack);
void stack_array_print(stack_array *stack);
```

Liste des fonctions pour une pile avec liste chaînée

`stack_array *stack_array_create(int max_length)`

Cette fonction crée une pile vide. Étant donné qu'il s'agit d'une implémentation à base de tableau de taille fixe, la taille maximale du tableau est donnée en paramètre. En cas d'erreur (pas assez de mémoire), elle renvoie un pointeur **`NULL`**.

`void stack_array_delete(stack_array *stack)`

Cette fonction vide une pile de l'ensemble de ses éléments, et détruit la structure restante. Si le paramètre donné est **`NULL`**, la fonction ne fait rien.

`int stack_array_max_length(stack_array *stack)`

Cette fonction renvoie la longueur du tableau contenant la pile. Si le paramètre donné est **`NULL`**, la fonction renvoie -1.

`int stack_array_length(stack_array *stack)`

Cette fonction renvoie la longueur de la pile (c'est-à-dire le nombre d'éléments actuellement dans la pile). Si le paramètre donné est **`NULL`**, la fonction renvoie -1.

`int stack_array_push(int elt, stack_array *stack)`

Cette fonction empile un élément dans une pile, c'est-à-dire qu'elle ajoute un élément au sommet. En cas de succès, la fonction renvoie 0. Si la pile donnée en paramètre est **`NULL`**, la fonction renvoie -1. Si le nombre donné en paramètre est inférieur à 0, la fonction renvoie -4. Si le tableau est déjà plein, la fonction renvoie -3.

`int stack_array_pop(stack_array *stack)`

Cette fonction dépile un élément d'une pile, c'est-à-dire qu'elle supprime l'élément au sommet. En cas de succès, la fonction renvoie 0. Si la pile donnée en paramètre est **`NULL`**, la fonction renvoie -1. Si la pile donnée en paramètre est vide, la fonction renvoie -2.

`int stack_array_head(stack_array *stack)`

Cette fonction renvoie l'élément au sommet de la pile. Si la pile donnée en paramètre est **`NULL`**, la fonction renvoie -1. Si la pile donnée en paramètre est vide, la fonction renvoie -2.

`int stack_array_clear(stack_array *stack)`

Cette fonction vide une pile de l'ensemble de ses éléments, sans détruire la structure de la pile. La fonction renvoie le nombre d'éléments supprimés de la mémoire. Si le paramètre donné est **`NULL`**, la fonction renvoie -1. Si la pile donnée en paramètre est vide, la fonction renvoie 0.

`int stack_array_is_empty(stack_array *stack)`

Cette fonction teste si une pile est vide ou non. Si la pile est vide, la fonction renvoie 1. Si la pile n'est pas vide, la fonction renvoie 0. Si la pile donnée en paramètre est **`NULL`**, la fonction renvoie -1.

`int stack_array_search(int elt, stack_array *stack)`

Cette fonction recherche un élément dans la pile et renvoie sa position dans le tableau. La première position est celle où l'élément le plus ancien a été placé (c'est-à-dire le fond de la pile), cette position sera numérotée 0. Si l'élément n'est pas trouvé, la fonction renvoie -4. Si la pile donnée en paramètre est **`NULL`**, la fonction renvoie -1.

`stack_array *stack_array_reverse(stack_array *stack)`

Cette fonction inverse la position de tous les éléments de la pile. Le premier élément devient le dernier, l'avant dernier devient le deuxième, etc. En cas de succès, la fonction renvoie le pointeur vers l'éventuelle nouvelle adresse en mémoire de la structure de la pile inversée. En cas de problème mémoire, on renvoie **`NULL`**, et l'ancienne pile doit rester à son ancienne adresse mémoire sans subir la moindre modification. Si la pile donnée en paramètre est **`NULL`**, la fonction renvoie **`NULL`**.

`void stack_array_print(stack_array *stack)`

Cette fonction affiche le contenu de la pile. Le format d'affichage attendu implique d'afficher un seul élément par ligne, suivi d'un retour à la ligne. L'élément en tête de pile sera affiché en premier. Si la pile donnée en paramètre est vide, seul un retour à la ligne est affiché. Si la pile donnée en paramètre est **`NULL`**, rien n'est affiché.

```
$ ./my_stack_array
42
5
13

$
```

Exemple d'affichage du cas normal : pile contenant 42, 5, 13

```
$ ./my_stack_array

$
```

Exemple d'affichage d'une pile vide

```
$ ./my_stack_array
$
```

Exemple d'affichage d'un pointeur NULL

8 Exercice 4 - Suite de tests

Nom du(es) fichier(s) :	**`check.sh`**
Répertoire :	**`login.x-TP1/check/`**
Droits sur le répertoire :	750
Droits sur le(s) fichier(s) :	750
Outils recommandés :	**`diff(1)`**, **`find(1)`**, **`printf(3)`**

Objectif : Le but de l'exercice est de construire une suite de tests pour valider le fonctionnement de la pile.

Vous devez écrire plusieurs tests démontrant que vos implémentations des piles fonctionnent. Pour cela, vous devrez faire plusieurs programmes en C qui utilisent vos piles, et un script shell qui compare les résultats.
Dans les exercices précédents, vous n'avez pas écrit de fonction *main* pour la bonne raison qu'il s'agissait d'implémenter une structure (une pile) de différentes façons. Il est maintenant temps de réutiliser tous les tests que vous avez écrits lors du développement de vos piles ! Vous devez néanmoins penser aux cas un peu plus complexes que le cas général.

Un exemple avec 3 programmes C de test serait :
- Un programme qui utilise la pile implémentée avec des listes chaînées, imprime l'état de la pile à différents moments, et imprime certains retours de fonctions.
- Un programme qui utilise la pile avec tableau, imprime l'état de la pile à différents moments, et imprime certains retours de fonctions.
- Un programme témoin qui servira de vérité absolue sur le comportement attendu.
- Les trois programmes empilent et dépilent les mêmes valeurs dans le même ordre, afin que le même comportement soit visible d'un point de vue utilisateur. Seule l'implémentation sous-jacente varie.

Votre fichier **`main.c`** pourra contenir un scénario précis à suivre qui imprimera différentes valeurs dans le terminal, et vous pouvez éventuellement comparer cela à une sortie texte que vous avez manuellement préparée dans un fichier. Ainsi, le fichier préparé manuellement sera stocké en dur dans le dossier *check* et sera comparé avec un **`diff`** pour confirmer que tout se passe bien, ou au contraire que des problèmes existent lors de certains tests.

Pour utiliser une bibliothèque statique, lors du développement, vous devez disposer de la bibliothèque (le **`.a`** ou **`.so`**) et du fichier **`.h`** associé à l'interface (l'API) de votre bibliothèque, c'est-à-dire aux fonctions exportées.

Lors de la phase de *compilation*, vous devrez parfois ajouter un paramètre *INCLUDE* indiquant le dossier où trouver les *headers* de la bibliothèque (par exemple pour la *libxml2*, on ajoutera ce flag : **`-I/usr/include/libxml2`**). Ce flag **`-I`** permet d'ajouter un dossier dans lequel chercher les fichiers d'en-têtes propres aux bibliothèques (donc les **`.h`** entourés de **`< >`** dans vos *includes*). Vous pouvez cumuler plusieurs flags **`-I`** à la suite.

Lors de la phase d'*édition de liens*, vous devrez parfois ajouter un paramètre *LIBRARY* indiquant le nom de la bibliothèque à utiliser (par exemple pour la *libxml2*, on ajoutera

ce flag : **`-lxml2`**). Ce flag **`-l`** (littéralement : *tiret L minuscule*) permet d'indiquer à l'éditeur de lien qu'il doit utiliser une bibliothèque dynamique dont le nom est donné en paramètre (attention : le préfixe *lib* est supprimé des noms de bibliothèques). Attention, le placement de la bibliothèque par rapport aux fichiers objets (les fichiers **`.o`**) dans la ligne de commande a une importance.

Vous devrez également indiquer dans quel dossier trouver les bibliothèques grâce au flag **`-L`** (par exemple, pour chercher une bibliothèque dans le dossier courant, on ajoutera le flag : **`-L.`**). Vous pouvez cumuler plusieurs flags **`-L`** et **`-l`** à la suite.

Si vous disposez de la même bibliothèque avec une version statique et une dynamique, l'*éditeur de liens* choisira la version dynamique par défaut, mais vous pouvez forcer la statique grâce au flag **`-static`**.

La méthode idéale consiste à produire plusieurs scénarios numérotés et décrits, puis d'afficher quels tests ont réussi, et lesquels ont échoué. Pour cet exercice, vous n'êtes pas obligés de descendre à ce niveau de précision. Une simple petite suite de tests est suffisante, mais n'oubliez pas de vérifier autre chose que le cas général.
N'hésitez pas à vous appuyer sur le fait que votre projet produit des bibliothèques ! Vous pouvez développer un programme entier qui affiche une suite de tests avec un script shell très simple appelant ce programme, ou, vous pouvez utiliser un script shell plus complet qui fait de nombreux tests en s'appuyant sur un ou des programmes C très simples.

9 Exercice 5 - Pile avec tableau statique (bonus)

Nom du(es) fichier(s) :	**stack_static.c**
Répertoire :	**login.x-TP1/src/**
Droits sur le répertoire :	750
Droits sur le(s) fichier(s) :	640
Fonctions autorisées :	**write(2)**

Objectif : Le but de l'exercice est d'implémenter une pile en C utilisant un tableau statique et sans jamais modifier le tas du processus (c'est-à-dire sans utiliser `malloc(3)`, `mmap(2)`, ou tout autre appel système ou fonction modifiant le tas ou réservant des pages mémoire lors de l'exécution).

Les fonctions demandées dans cet exercice devront également se trouver dans la bibliothèque nommée **libmystack**.

Vous devez écrire plusieurs fonctions permettant de créer, utiliser, vider, et libérer une pile. Un fichier **stack_static.h** contenant toutes les fonctions exportables à implémenter vous est fourni en annexe. Vous devez déclarer une structure **stack_static** et l'ajouter dans **stack_static.h**. N'oubliez pas qu'un tableau statique est généré par le compilateur : vous devrez indiquer dans la constante de pré-compilation **STACK_STATIC_MAX_LEN** la taille maximum. La pile étant statique, il vous faudra déclarer une variable globale statique nommée **g_stack_static** qui pointera vers la structure (elle-même déclarée dans la variable globale nommée **g_s_stack_static**). Pour les premières étapes, vous devrez implémenter une version simplifiée de la pile qui ne prend en charge que des entiers positifs.
*Attention : étant donné qu'il s'agit d'une version statique, vous ne devez **JAMAIS** utiliser **malloc**, **free**, ou toute autre fonction ou appel système visant à réserver de la mémoire.*
Vous devez implémenter les fonctions suivantes :

```
void stack_static_create(void);
void stack_static_delete(void);

int stack_static_length(void);
int stack_static_max_length(void);

int stack_static_push(int elt);
int stack_static_pop(void);
int stack_static_head(void);

int stack_static_clear(void);
int stack_static_is_empty(void);

int stack_static_search(int elt);
```

```
void stack_static_reverse(void);
void stack_static_print(void);
```

Liste des fonctions pour une pile avec liste chaînée

`void stack_static_create(void)`

Cette fonction initialise les valeurs de la structure.

`void stack_static_delete(void)`

Cette fonction vide une pile de l'ensemble de ses éléments.

`int stack_static_max_length(void)`

Cette fonction renvoie la longueur du tableau contenant la pile.

`int stack_static_length(void)`

Cette fonction renvoie la longueur de la pile (c'est-à-dire le nombre d'éléments actuellement dans la pile).

`int stack_static_push(int elt)`

Cette fonction empile un élément dans une pile, c'est-à-dire qu'elle ajoute un élément au sommet. En cas de succès, la fonction renvoie 0. Si le nombre donné en paramètre est inférieur à 0, la fonction renvoie -4. Si le tableau est déjà plein, la fonction renvoie -3.

`int stack_static_pop(void)`

Cette fonction dépile un élément d'une pile, c'est-à-dire qu'elle supprime l'élément au sommet. En cas de succès, la fonction renvoie 0. Si la pile est vide, la fonction renvoie -2.

`int stack_static_head(void)`

Cette fonction renvoie l'élément au sommet de la pile. Si la pile est vide, la fonction renvoie -2.

`int stack_static_clear(void)`

Cette fonction vide une pile de l'ensemble de ses éléments, sans détruire la structure de la pile. La fonction renvoie le nombre d'éléments supprimés de la mémoire. Si la pile est vide, la fonction renvoie 0.

`int stack_static_is_empty(void)`

Cette fonction teste si une pile est vide ou non. Si la pile est vide, la fonction renvoie 1. Si la pile n'est pas vide, la fonction renvoie 0.

`int stack_static_search(int elt)`

Cette fonction recherche un élément dans la pile et renvoie sa position dans le tableau. La première position est celle où l'élément le plus ancien a été placé (c'est-à-dire le fond de la pile), cette position sera numérotée 0. Si l'élément n'est pas trouvé, la fonction renvoie -4.

`void stack_static_reverse(void)`

Cette fonction inverse la position de tous les éléments de la pile. Le premier élément devient le dernier, l'avant dernier devient le deuxième, etc. *Attention : vous ne devez pas utiliser de* ***`malloc`*** *ou de tableau temporaire pour effectuer cette fonction.*

`void stack_static_print(void)`

Cette fonction affiche le contenu de la pile. Le format d'affichage attendu implique d'afficher un seul élément par ligne, suivi d'un retour à la ligne. L'élément en tête de pile sera affiché en premier. Si la pile est vide, seul un retour à la ligne est affiché. *Attention, en version statique, vous devrez utiliser* `write(2)` *et non pas* `printf(3)`*.*

```
$ ./my_stack_static
42
5
13

$
```

Exemple d'affichage du cas normal : pile contenant 42, 5, 13

```
$ ./my_stack_static

$
```

Exemple d'affichage d'une pile vide