TP de SEPC
Révision de C

Ensimag 2A

Résumé

Ce TP a pour but de vous faire réviser le langage C. Vous devriez pouvoir réaliser chacun des TPs. N'hésitez pas à poser des questions pendant la séance si vous bloquez sur un point.

Préambule

Cette séance part du principe que vous avez suivi (et compris !) le parcours *Autoformation sur les concepts de base* du cours d'introduction au langage C de l'Ensimag (sur gitlab).

Si vous sentez avoir besoin de vous rafraichir la mémoire, ou si vous n'avez pas suivi ce cours et trouvez les exercices de cette feuille difficiles, utilisez cette séance pour (re)parcourir les fiches de cours du *Kit de démarrage* disponible à l'adresse qui suit (en particulier les fiches *Pointeurs*) :
`https://gitlab.ensimag.fr/formationc/prepa/prof/wikis/home`

1 Introduction

Vous devez récupérer les squelettes de code et les tests avec la commande :

```
1  git clone https://github.com/gmounie/ensimag-rappeldec.git
```

Cela va créer un répertoire `ensimag-rappeldec`.

Le code est à écrire dans le répertoire `ensimag-rappeldec/src`.

Pour compiler, vous devrez éditer le fichier `ensimag-rappeldec/CMakeList.txt` pour y insérer votre login (comme lors de votre futur examen) et utiliser les makefiles créés par cmake pour la compilation.

La séquence des commandes pour créer les makefiles, compiler et lancer les tests (après édition du `ensimag-rappeldec/CMakeList.txt`) est :

```
cd ensimag-rappeldec/build
cmake .. # Il y a bien deux points ! Le but est de compiler dans build
make
make test
make check
```

2 Les listes chainées

Dans le répertoire `src`, modifier le fichier `listechainee.c` pour écrire un module de gestion de listes simplement chainées. Ce module contient les fonctions suivantes à implémenter :

```
/* Affiche les éléments de la liste passée en paramètre sur la sortie
 * standard. */
void affichage_liste(struct elem *liste);

/* Crée une liste simplement chainée à partir des nb_elems éléments du
 * tableau valeurs. */
struct elem *creation_liste(long unsigned int *valeurs, size_t nb_elems);

/* Libère toute la mémoire associée à la liste passée en paramètre. */
void destruction_liste(struct elem *liste);

/* Inverse la liste simplement chainée passée en paramètre. Le
 * paramètre liste contient l'adresse du pointeur sur la tête de liste
 * à inverser. */
void inversion_liste(struct elem **liste);
```

2.1 Utilisez votre debugger !

Lorsque l'on programme, une grande partie du temps consiste à déboguer le programme, surtout si il est en C. Pour avoir votre diplôme, utiliser un « débogage printf » est probablement suffisant : tant que le programme n'est pas débogué, ajouter et enlever des printf et recompiler.

Il est difficile, sur des petits exemples académiques de vous convaincre que passer 15 minutes à apprendre à vous servir d'un débogueur, vous économisera des heures d'édition/compilation plus tard et vous permet d'explorer une exécution en cours avec plu de facilité et de souplesse.

D'expérience, le saut sera franchi pour tout le monde lorsque vous n'aurez plus le choix, devant un projet technique (similaire au « Projet Système » (PCSEA)), gros, mal instrumenté ou pas facilement recompilable. C'est dommage, car après, vous utiliserez votre débogueur quasi-systématiquement vous demandant comment vous faisiez avant.

Ensuite, vous allez lire la documentation de votre débogueur, et y découvrir les fonctions avancées que vous auriez rêvé de connaître avant.

Alors, autant partir de la fin, et vous faire manipuler un exemple avancé de GDB.

Un débogueur permet de lire, et changer, facilement, les valeurs des variables, de la mémoire, l'instruction courante ou d'explorer la pile d'appel avec les vraibles locales des fonctions.

Après l'écriture de `affichage_liste`, il est facile d'ajouter une fonction qui affiche les valeurs des têtes de listes et le contenu de la liste.

Exemple :

```
1  void debug_inversion(struct elem *h1, struct elem *h2) {
2      printf("head1= %p, head2= %p\n", h1, h2);
3      affichage_liste(h1);
4      affichage_liste(h2);
5  }
6
7  void inversion_liste(struct elem **liste) {
8     struct elem *one_list_head= *liste;
9     struct elem *another_list_head;
10 ...
11    Inversion loop is here, e.g. at line 90
12 ...
13 }
```

Puis lancez GDB sur le programme et définissez un breakpoint. Sur ce

breakpoint ajoutez une commande qui va appeller la fonction `debug_inversion` avec les bons arguments. À chaque fois que le breakpoint est atteint, la commande s'exécute.

```
$ gdb listechainee
(gdb) break listechainee.c:90
....
(gdb) command 1
Type commands for breakpoint(s) 1, one per line.
End with a line saying just "end".
>call debug_inversion(one_list_head, another_list_head)
>end
(gdb) run
...
(gdb) cont
...
(gdb) cont 10 # passe 9 fois le breakpoint (stop à la 10
↪  ième)
```

2.2 Rappel sur l'algorithme d'inversion de liste simplement chaînée

Contactez votre enseignant si vous avez des soucis d'ordre d'algorithmique !

Quelques explications rapides pour une inversion itérative. Le principe de base est remplir à l'envers une liste temporaire. Pour cela il suffit de supprimer le premier élément de la liste originale puis de l'insérer en tête dans la liste temporaire. Ces opérations de suppression du premier, puis son insertion en tête sont répétées tant que la première liste n'est pas vide.

À la fin il suffit de mettre à jour la tête de liste originale avec la tête de la liste temporaire.

Il est possible de faire ensemble ces deux opérations, ce qui permet d'éviter de faire quelques affectations de pointeurs inutiles.

2.2.1 Pour réfléchir un peu...

Pourquoi faut-il éviter de faire une inversion récursive (fonctionnelle) en C ?

3 Les opérateurs binaires

Dans le répertoire `ensimag-rappeldec/src`, compléter le fichier `binaires.c`. La fonction `unsigned char crand48()`, à chaque appel, elle devra appliquer à la variable globale X la fonction suivante :

$$X_{n+1} = (aX_n + c) mod(m)$$

avec $m = 2^{48}$, `a = 0x5DEECE66D` et `c = 0xB`.

La fonction retourne ensuite l'octet correspondant aux bits 32 à 39 (en commençant la numérotation à 0 pour les bits de poids faibles).

Vous devrez utiliser les opérateurs bits à bit comme `<<`, `>>`, `&`, `|`, `~`, `^` (Décalage à gauche, Décalage à droite, ET binaire, OU binaire, NOT binaire, XOR binaire).

4 Allocation et ramasse-miette

Un ramasse-miette (Garbage Collector) généraliste performant est difficile à écrire. C'est toujours un sujet de recherche actif. Java et Go sont deux exemples, différents, d'implantations.

La programmation d'un ramasse-miette simplifié pour un pool d'objets identiques, en nombre borné, et dont les références sont très bien encadrés, est beaucoup plus simple. Vous implantez trois fonctions qui initialisent, allouent et ramassent des éléments de listes chaînées. Ces fonctions seront utilisées par un programme de test fourni.

4.1 Allocation et ramasse-miette

Le but est d'écrire la fonction d'allocation et la fonction de ramassage d'éléments de liste chaînée :

```
1  struct elem {
2    int val;
3    struct elem *next;
4  };
```

Les éléments sont alloués au sein d'un bloc de mémoire pré-réservé. L'utilisation de chaque élément est notée dans un vecteur de booléens mis-à-jour par les fonctions. Ce vecteur de booléens permet à la fonction d'allocation de savoir si une zone du bloc correspondant à un élément est libre ou pas.

Les fonctions de manipulation du vecteur de booléens sont fournies (lire la valeur d'un booléen, l'écrire, mettre tout le vecteur à `false`).

4.2 Fonctions interdites

Vous ne devez pas utiliser les fonctions qui vous permettraient de gérer des allocations dynamiques pour les `struct elem` tel que `malloc()`, `free()`, `realloc()`, etc.

Vous pouvez bien sûr ajouter vos propres structures de données et les initialiser correctement. Suivant votre façon de coder d'autres fonctions peuvent être utiles.

4.3 Le travail à réaliser

Vous devez implanter dans le fichier `src/elempool.c` les deux fonctions

```
1  struct elem *alloc_elem();
2  void gc_elems(struct elem **heads, int nbheads);
```

Ces deux fonctions manipulent des adresses qui sont dans le bloc de mémoire `static unsigned char *memoire_elem_pool` de `src/elempool.c` alloué par la fonction `void init_elems()`.

Le bloc de mémoire alloué a une taille de 1000 `struct elem`. La fonction d'initialisation est appelée au début de chaque fonction de tests.

Pour tracer l'utilisation du bloc, des fonctions de manipulation d'un vecteur de 1000 booléens sont fournies. Chaque booléen peut être lu ou écrit individuellement. Il est également possible de les passer tous à `false (0)`.

```
1  bool bt1k_get(long unsigned int n); // obtenir la valeur
   ↪  du bit n
2  void bt1k_set(long unsigned int n, bool val); // mettre le
   ↪  bit n à val
3  void bt1k_reset(); // tous les bits à false
```

La fonction `struct elem *alloc_elem(void)` renvoie :

0 ou NULL : si il n'y a pas de portion du bloc mémoire disponible,

une adressse : celle d'une portion du bloc qui sera utilisée comme un `struct elem`.

Pour cela elle utilisera le vecteur de booléens pour trouver une zone libre. Dans cet exercice un simple parcours linéaire depuis le début du vecteur pour trouver la valeur que vous cherchez suffira.

La fonction `void gc_elems(struct elem **heads, int nbheads)` a pour but de trouver les portions du bloc qui sont et ne sont pas utilisées. Pour cela elle reçoit en argument un tableau contenant les adresses de 0, une ou plusieurs têtes de liste. Ces listes contiennent les éléments utilisés. **Tout élément qui n'est pas chaîné dans une des listes passées en paramètre, au moment de l'appel de `gc_elems()`, est donc libre.**

Le but de la fonction sera donc de mettre à `true` tous les booléens correspondant à un élément utilisé et de mettre à `false` les autres.

5 Hello World !

Dans le répertoire `src`, compléter le fichier C de nom `hello.c`.
Votre programme affichera :

$$\forall\ p \in world,\ \ hello\ p$$

6 Les nombres flottants

Dans le répertoire `src`, compléter le fichier C de nom `flottants.c`

6.1 question facile

Afficher la valeur de l'opération $0.1 + 0.2 - 0.3$ en effectuant le calcul avec les 3 tailles de nombres flottants (Il y a bien trois tailles !). Les constantes devront aussi être initialisées avec le bon type. Pour cela il faut parfois leur adjoindre le bon suffixe (`0.1f` ou `0.1L`).

Chacune de ces trois valeurs sera affichée sur une ligne en notation `[-]d.ddddddedd` (avec les puissances de 10). Les trois lignes seront à afficher dans l'ordre croissant de la taille en octet des 3 types flottants.

Indication : vous devriez créer des variables intermédiaires du bon type.

6.2 question moins facile

Idem mais sans créer de variables intermédiaires.

7 Les fonctions : les pointeurs de fonctions

Dans le répertoire `ensimag-rappeldec/src`, compléter le fichier `fqsort.c` pour trier le tableau de nombres complexes en fonction de l'argument complexe des nombres.

Vous utiliserez la fonction `qsort` de la bibliothèque standard.

8 Pour aller plus loin

Vous n'avez probablement pas fait tous les exercices du stage C. Rendez-vous sur Chamilo pour faire un des exercices qui vous intéressent.

Vous pouvez aussi faire l'allocateur mémoire, sujet réalisé par les apprentis 2A (sujet sur ensiwiki).