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 Chamilo).

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 parcours *Kit de démarrage* disponible à l'adresse qui suit (en particulier les fiches *Pointeurs*) : `http://chamilo.grenoble-inp.fr/courses/ENSIMAG3MMPLC/index.php`

1 Introduction

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

```
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(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 Rappel sur l'algorithme d'inversion de liste simplement chaînée

La base de l'inversion itérative est :

1. récupérer l'adresse du deuxième élément de la liste originale
2. au premier de la liste originale, lui affecter comme suivant le premier de la liste inversée
3. insérer le premier de la liste originale en tête de la liste inversée
4. prendre l'ex deuxième de la liste originale comme nouvelle tête de la liste originale et recommencer

2.1.1 Pour réfléchir un peu...

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

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 `«, », &, |, ~, ^`.

4 Allocation et ramasse-miette

L'objectif est la programmation d'un ramasse-miette simplifié (Garbage Collector) pour un pool d'objets identiques, éléments de listes simplement chaînées. Vous implantez deux fonctions qui allouent et ramassent ces éléments de liste chaînée. 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 :

```
struct elem {
  int val;
  struct elem *next;
};
```

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 Le travail à réaliser

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

```
struct elem *alloc_elem();
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)`.

```
bool bt1k_get(int n); // obtenir la valeur du bit n
void bt1k_set(int n, bool val); // mettre le bit n à val
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.

4.3 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.

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 !).

Chacune de ces trois valeurs sera affichée sur une ligne en notation `[-]d.dddedd` (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).