Université Gustave Eiffel — INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

L3 Info

TD Prog.C n°3

Traceur d'allocation
part.1 : fonctions clônes et environnement

Ce TP constitue la 1° partie de ce qui deviendra au final une bibliothhèque partagée (`libmtrack.so`) permettant de tracer les appels aux fonctions de gestion mémoire (`malloc`, `calloc`, `realloc`, `free`) dans un programme, sans intervention dans le code source, l'activation se faisant via des directives de compilation.

→ il faut avoir lu le document détaillant le projet complet (sur `eLearning`) pour bien comprendre l'objectif final.

Dans les 2 ou 3 TP consacrés à ce sujet rien n'est imposé quant aux structures de données, fonctions ou autre : il s'agit d'une description des fonctionalités attendues et de conseils pour vous guider, mais la mise en œuvre vous appartient.

Dans la mesure où il s'agit de créer des outils permettant de repérer et éventuellement surmonter des erreurs de programmation liés à l'utilisation dynamique de la mémoire (allocations, libérations), il faudra écrire des codes de tests ciblés *en essayant de couvrir un maximum de cas susceptibles de poser des problèmes.*

1°) Bases du traceur de mémoire

Dans cette première étape, il va s'agir de mettre en place les bases (structures de données et fonctions «clônes») de ce qui deviendra la bibliothèque de traçage de la mémoire utilisée par un programme.

L'idée est d'enregistrer un maximum d'informations sur tous les blocs mémoire alloués, déplacés, libérés et essayer de détecter les problèmes. Par exemple, avant de libérer un bloc (`free(b)`) il faudra vérifier que `b` est bien une adresse valide (non `NULL`, obtenue par une allocation dynamique et qui n'a pas encore été libérée). Et ci ce n'est pas le cas, il faudra notifier le problème et ne pas faire la libération (l'erreur est contournée).

Pour cela il va falloir remplacer tous les appels aux fonctions de gestions de mémoire (juste `malloc` et `free` pour l'instant) par des appels à des fonctions «clônes».

Ces clônes devront avoir exactement les mêmes prototypes (entrées/sorties) que les originaux.

Il ne s'agit pas de redéfinir les protocoles de gestion de mémoire des fonctions `malloc` et `free`, mais simplement de les surcharger : les clônes feront bien appel aux fonctions `malloc` et `free` de la `libC`.

La base de départ sera un programme de test tout simple (mais contenant des erreurs) tel que :

track_00.c

```c
#include <stdlib.h>
int main(void)
{
    char *a=malloc(1);
    char *b=malloc(1);
    free(a);
    b++;
    free(b);
    return 0;
}
```

xterm

```
$>gcc -Wall track_00.c -o track_00
$> ./track_00
free(): invalid pointer
Abandon (core dumped)
```

Ce programme va grossir au fil des versions (mise en place) puis s'alléger (passage en modulaire, usage de `Makefile(s)`) pour finalement revenir exactement à son état initial lorsque le programme sera suffisamment avancé.

Il sera donc fondamental de bien conserver toutes les étapes, de choisir des conventions de nommage claires et non-ambigües et de s'y tenir jusqu'au bout (cf. **annexe** en fin de document).

▸ Exercice 1. (structure(s) d'enregistrement)→ `track_01.c`

Les informations collectées par les fonctions clônes seront stockées dans des *cellules de données*, elles-mêmes rangées dans une «table» de taille indéterminée puisque dépendant du code à traiter.

Il faudra également maintenir divers compteurs pour produire, en sortie de programme, un rapport sur les éléments récoltés et analysés.

ⓐ **Cellule** : correspond à un bloc d'allocation (`malloc`).
dans un premier temps les seules données à enregistrer sont :

- l'adresse renvoyée (forcément `void*`).
- la taille du bloc alloué (attention c'est un type `size_t`[(1)]).
- un booléen, mis à `true` à l'allocation et qui passera à `false` à la libération.

important : une cellule crée ne sera jamais détruite (sauf bien sûr en sortie de programme) : c'est ce qui permettra en particulier de détecter les `'double free'` et le recyclage d'adresse.

ⓑ **Environnement** : c'est la table d'enregistrement des cellules. Elle devra contenir :

- un moyen quelconque de stocker les cellules (tableau, liste chaînée, table de hachage ...).
  ☞ à vous de choisir une méthode adaptée.
  ☞ dans un 1° temps, on pourra utiliser un tableau (dynamique – c'est mieux – ou statique) de taille "raisonnable" au départ et s'en contenter[(2)], la question de la croissance de la table pouvant être traitée bien plus tard, en finalisation.
- des compteurs d'appels pour chaque fonction clône (`malloc` et `free` pour l'instant).
  il peut être intéressant de disposer de 2 compteurs pour `free` : un pour les appels "réussis", un autre pour les appels qui échouent (`malloc` échoue rarement, et si ça arrive c'est que le problème est vraiment sérieux ☞ il vaut mieux arrêter le programme : `exit(1);`).
- des compteurs d'accumulation pour les quantités totales de mémoire allouée / libérée.

Note importante : puisque les clônes doivent avoir accès aux données d'environnement (table de cellules, compteurs) mais que celles-ci ne peuvent pas leur être passées en paramètres (mêmes prototypes que les originaux), il n'y a pas d'autre option que de déclarer l'environnement en **global**.
☞ pour limiter le recours à ces variables globales, on aura intérêt à tout regrouper dans une structure générale
☞ cet `Environnement` sera, du moins au début, la seule variable globale
☞ il faudra donc prévoir des fonctions (initialisation, bilan, nettoyage) pour gérer cette variable

ⓒ **Clônes** : ce sont les fonctions qui vont venir remplacer (ou plutôt *surcharger*) celles de la `libC`.

Elles feront bien sûr appel aux fonctions originale, mais en récoltant / vérifiant au passage les informations demandées pour mettre à jour la table.

Chaque appel au clône de `malloc` devra donc

- faire l'appel au vrai `malloc`.
- vérifier si l'adresse créée existe déjà dans la table,
- si c'est le cas (recyclage), faire la mise à jour
- sinon, créer une nouvelle cellule.

Et pour chaque appel au clône de `free` il faudra :

- vérifier si l'adresse créée existe déjà dans la table, et n'a pas encore été libérée.
- si c'est le cas, faire appel au vrai `free` et indiquer que la zone est libérée (booléen mis à `false`)
- sinon c'est une **erreur** (adresse invalide) ou un **warning** (`free(NULL)`)
  ☞ afficher un message et sortir de la fonction clône sans appeler le vrai `free` (on évite l'erreur).

(1) i.e. un entier `unsigned` dont on ne connait pas la taille

(2) concrètement, si la table grossit de manière "déraisonnable", c'est probablement le signe que le code analysé est très mal conçu : le traceur pourra alors afficher un message et se désactiver (arrêter de comptabiliser).

ⓓ **Code à analyser** : c'est juste les quelques lignes de la fonction `main` dans le code `track_00.c`.

La version 1 (`track_01.c`) doit donc intégrer tous les éléments précédents (types et fonctions).
Il faudra limiter au strict minimum les modifications dans la fonction `main` :

- bien sûr remplacer les appels à `malloc` et `free` par leur clône
- réaliser l'activation : en amont, avant ces appels, initialiser l'environnement
- réaliser la sortie : en aval, juste avant de quitter, afficher le bilan et vider l'environnement

Sur l'exemple initial ça donnerait (c'est largement adaptable) quelque chose comme :

version initiale

```
xterm
$> ./track_00
free(): invalid pointer
Abandon (core dumped)
```

version avec traceur

```
xterm
$> ./track_01
(01) malloc(1)->0x561f04baf2a0
(02) malloc(1)->0x561f04baf5d0
(01) free(0x561f04baf2a0)
(02) free(0x561f04baf5d1) - ERREUR : adresse illégale -> ignoré
----------------------
BILAN FINAL
total mémoire allouée : 2 octets
total mémoire libérée : 1 octet
ratio                 : 50%
<malloc>              : 2 appels
<free>                : 1 appel   correct
                      : 1 appels incorrect
----------------------
```

▸ **Exercice 2. (activation et arrêt automatique)→** `track_02.c`

Pour limiter les interventions dans le code source à traiter (ici, la fonction `main`), on peut déjà facilement rendre «invisibles» les parties amont (activation) et aval (sortie) de la version précédente.

L'intérêt sera aussi (plus tard...) que si le code à analyser ne fait aucun appel aux fonctions `malloc, free`, le traceur ne s'activera même pas.

ⓐ **activation** :
déclenchée au premier appel à une fonction clône.

- la création de l'environnement et tout ce qui va avec est activé, via une variable globale de type `flag` booléen, initialisée à `false` : le premier clône qui la rencontre, déclenche l'activation
  ☞ le `flag` de démarrage passe alors à `true` et les clônes suivants ignoreront cette partie.
- toutes les fonctions clônes doivent disposer de ce mécanisme puisqu'on ne sait pas laquelle sera la première (même le clône de `free` doit pouvoir activer le traceur, au cas où un codeur étourdi y fasse appel avant même d'avoir alloué quoi que ce soit).

ⓑ **sortie** :
il faut ici faire appel à la fonction standard `int atexit(void (*func)(void))` (définie dans `<stdlib.h>` – `$>man atexit`) où `func` est une fonction, définie par l'utilsateur, regroupant des instructions qui ne seront exécutées qu'à l'arrêt du programme, à la réception du signal `exit`.

- le traceur devra donc disposer d'une fonction de prototype `void f(void)` regroupant les appels de fin : affichage du bilan et nettoyage de l'environnement.
- l'appel à `atexit(f);` devra être judicieusement placé : il ne doit être exécuté qu'une seule fois, et uniquement si le traceur a été activé.

Avec cette adaptation, la modification du code à traiter (ici, fonction `main`) se limite désormais au remplacement des noms des fonctions `malloc` et `free` par les noms de leur clône.

L'exécution de `track_02` doit être strictement identique à celle de `track_01` .

Annexe : conventions de nommage

en `C` il n'y aucune règle concernant les noms donnés aux types, variables, fonctions ou autres. C'est le programmeur qui choisit.

Ici, étant donné l'usage que l'on vise au final – *patcher* un code existant – il est important de limiter au maximum les risques de conflits de nom. Il faut donc éviter de choisir des noms simples comme `FLAG` pour une variable boléenne ou `Cell, List` pour des types gérant des listes chaînées...

Il est judicieux de se choisir un préfixe (en minuscule[(3)]) adapté, encadré par des caractères `underscore` (`_truc_`) et de nommer tout (types, variables, fonctions, constantes, macros) sous la forme `_truc_fonction, _truc_type, _truc_const ...`

☞ les fonctions clônes seront `_truc_malloc` et `_truc_free` et une cellule de la table sera de type `_truc_cell` et la déclaration d'un *flag booléen* prendait la forme `_truc_bool _truc_flag=_truc_true;`

[(3)] les préfixes `_TRUC` étant réservés pour les standards et les compilateurs