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

L3 Info

TD Prog.C n°4

Traceur d'allocation
partie 2 : fonctionalités des clônes (patch)

Ce TP constitue la 2° partie du mini-projet «Traceur».
Les fonctions de patch mises en place dans la 1° partie ne font rien de spécial : elle indique qu'elles sont bien des clônes et affichent éventuellement une adresse : celle du bloc mémoire alloué ou libéré.

Dans cette seconde partie il s'agit de leur faire faire des choses un peu plus évoluées.

Note importante *: à ce stade, le* traceur *n'est encore qu'un simple module (pas encore une bibliothèque partagée).*

- *il apparaît donc encore comme une* **composante** *classique (cf.* `CM2 make/Makefile`*) du programme final.*
- *le code objet que contient le module traceur* `mtrack.o` *est intégré (pas seulement lié) à l'exécutable par l'étape finale d'édition de lien.*

☞ *en conséquence toute modification du code source* `src/mtrack.c` *implique une recompilation de celui-ci (mise à jour de* `mtrack.o`*) mais aussi une recompilation des codes cibles* `src/test*.c` → `test*.o` *(même si ceux-ci n'ont pas été modifiés), et donc une relance de l'édition de lien (intégration des mises à jour) pour recréer l'exécutable final.*

☞ *l'usage d'un* `Makefile` *adapté est donc vivement recommandé.*

☞ *conventions de nommage : avant de vous lancer, jetez un oeil sur le court annexe en fin de document.*

2°) Récolte d'informations

④ (Structure(s) d'enregistrement)

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 (nombre/ordre d'appel des fonctions à patcher).

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`), caractérisé par l'adresse de son 1° octet et sa taille en octets..
  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 indiquant si le bloc d'allocation a été libéré par un appel à `free`.

  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.

[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).

- des compteurs d'appels pour chaque fonction clône (`malloc, calloc, realloc`[3], `free`).
  il peut être intéressant de disposer de 2 compteurs pour `free` : un pour les appels "réussis", un autre pour les appels qui échouent (`malloc/calloc` échouent rarement, et si ça arrive c'est que le problème est vraiment sérieux et il vaut mieux arrêter le programme : `exit(1);`).
- des compteurs d'accumulation pour les quantités totales de mémoire allouée / libérée.
- en option, ou pour plus tard, divers dispositifs pour avoir un mode *bavard*, pour rediriger la sortie vers un fichier...

- **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 aux clônes de `malloc/calloc` 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
  - sinon c'est une **erreur** (adresse invalide) ou un **warning** (`free(NULL)`)
    ☞ afficher un message et sortir de la fonction clône sans appeler `free` (on évite l'erreur).

Sur l'exemple de référence (`src/test0.c`) ça donnerait (c'est largement adaptable) quelque chose comme :

version initiale

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

version avec traceur

xterm
```
$> ./test0
(01) malloc(1)->0x561f04baf2a0
(01) free(0x561f04baf5d1) - ERREUR : adresse illégale -> ignoré
-----------------------
BILAN FINAL
total mémoire allouée : 1 octets
total mémoire libérée : 0 octet
<malloc>              : 1 appels
<free>                : 0 appel   correct
                      : 1 appels incorrect
```

⑤ (activation automatique)

Le module traceur va donc agir en recherchant et/ou créeant des données dans la table d'enregistrement à chaque appel à une fonction clône : les données crées par un `malloc` seront utilisées par un `free` et alimentent les statistiques globales.

Cette table d'enregistrement a donc, à l'évidence, une portée **globale** mais **statique** au niveau du module traceur : elle doit être accessible (lecture/écriture) à tous les niveaux du module mais totalement invisible de l'extérieur.

En particulier, les fonctions clônes doivent avoir un accès complet a ces données (création/lecture de cellules, alimentation des compteurs...). Mais ces données ne peuvent pas être passées en paramètres des fonctions clônes puisqu'elles ont les mêmes prototypes que leurs originaux.

Et si le code à tester ne contient aucun appel aux fonctions *tracées* la table d'enregistrement et tout ce qui s'y rapporte n'a aucune raison d'exister.

[3] le cas de `realloc`, un peu plus complexe, peut être mis de côté dans un 1° temps

Il faut donc mettre en place des processus spécifiques pour gérer ces données :

- activation : le traceur est activé (créé, initialisé) uniquement si nécessaire, donc forcément au premier appel à une fonction clône quelconque. Il n'est bien sûr activé qu'une seule fois.

  ☞ cette activation déclenche la création et le paramétrage de la table (structure laissée à votre appréciation) et des divers compteurs.
- circulation : chaque fonction clône doit être en mesure de :
  - activer le système d'enregistrement si ce n'est pas encore fait
  - y accéder en lecture / écriture
  - si nécessaire désactiver le système (ex. table statique pleine ou en croissance «déraisonnable»).

Comme pour le `TP03 (iList.[h|c])`, la solution la plus simple et la plus naturelle consisterait à utiliser une variable **globale** contenant tout l'environnement du traceur (la table, les compteurs....).

Une solution préférable consiste à utiliser une fonction (où l'environnement est déclaré en variable locale statique) qui l'initialisera au premier passage et se contentera d'en renvoyer l'adresse aux appels suivants.

⑥(procédure d'arrêt : transfert en sortie d'exécution)

En fin d'exécution du programme appelant, le traceur devra, de lui-même (et si il a été activé), produire un rapport de synthèse, se désactiver et nettoyer (libérer) toute mémoire qu'il aurait consommé (table dynamique, par exemple).

il faut faire appel à la fonction standard `int atexit(void (*func)(void))`[(4)] 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 sortie* de prototype `void f(void)` regroupant tous 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.

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, `Cell, List` pour des types gérant des listes chaînées ou encore `f()` pour une fonction.

Il est judicieux de se choisir un préfixe (en minuscule[(5)]) 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;`

c'est un peu lourd, mais préfigure des pratiques qui seront souvent **imposées** par des langages de plus haut niveau (`Python, Java, C++`)

(4) (définie dans `<stdlib.h>` – `$>man atexit`)

(5) les préfixes `_TRUC` étant réservés pour les standards et les compilateurs