EPFL CS212 : ImgStore – Système de fichiers orienté images — 06 : refactoring 1 (`open`, `close`) et `delete`

E. Bugnion & J.-C. Chappelier EPFL

Rev. 2021.04.12 / 1

Table des matières

Projet de Programmation Système – CS212 – 2021

ImgStore – Système de fichier orienté images

06 : refactoring 1 (`open`, `close`) et `delete`

Introduction

Cette semaine, nous allons ajouter une nouvelle fonctionnalité à notre outil : la suppression d'images (`delete`) ; mais aussi mettre en œuvre vos tous nouveaux savoirs sur les pointeurs pour écrire du code plus « pro » (ce que l'on continuera la semaine prochaine). Nous allons pour cela devoir faire ce qui se passe assez souvent dans de vrais projets : réviser le code déjà écrit (« *code refactoring* »). Nous allons en particulier pouvoir modulariser un peu plus notre interpréteur de commande (`imgStoreMgr`) en ajoutant deux nouvelles fonctions : `do_open()` and `do_close()`.

Matériel fourni

Nous ne vous fournissions cette semaine que du matériel de test. Pour le reste, vous allez cette fois devoir créer par vous-même le fichier supplémentaire nécessaire. Bien évidemment, vous allez également ré-utiliser et réviser les fichiers de la semaine précédente.

Pour récupérer le matériel de test fourni, procédez comme la semaine passée. Afin de ne pas perturber votre travail en cours, ce matériel est à nouveau fourni dans le répertoire `provided/`, **lequel NE doit absolument PAS être modifié** (par vous). Copiez ce matériel fourni vers votre répertoire `done/` lorsque vous jugez nécessaire de l'utiliser.

Travail à faire

Votre travail de la semaine consiste en quatre modifications :

1. remplacer chaque utilisation de `struct` comme argument de fonction par un *pointeur* ou un « const *pointeur* » suivant les cas ; comme indiqué en cours, cela permet de faire des passages par référence lorsque c'est nécessaire, ou d'éviter des copies lors du passage par valeur ;
2. définir les fonctions `do_open()` et `do_close()` : `do_open()` doit ouvrir le fichier de base de données d'images, lire son « *header* », ainsi que le tableau des « métadonnées » ; la fonction `do_close()` ferme le fichier de base de donnée d'images ouvert ;
3. prototyper `do_delete()` dans le fichier `imgStore.h` et l'implémenter dans un nouveau fichier `imgst_delete.c` ; la fonction `do_delete()` doit « effacer » une image spécifiée (on va voir ci-dessous ce que cela signifie vraiment) ;

4. réviser le cœur de l'interpréteur de commande, `imgStoreMgr.c`, pour lui apporter les modifications précédentes.

1. Passage par référence/pointeurs constants

La signature de la fonction `do_list()` de la semaine 4 était :

```
do_list (const struct imgst_file imgst_file)
```

Comme vous l'avez vu en classe, le contenu entier de la structure `imgst_file` est alors copié lors de l'appel (passage par valeur). Lorsque la structure est grande, cela a un coût non négligeable. Plus grave, la pile (« *stack* ») est souvent d'une taille très limitée dans beaucoup d'environnements et l'utilisation abusive de la pile pour y copier des objets crée le risque d'un dépassement de pile (en anglais « *stack overflow* »).

Pour éviter cela, il suffit de passer comme argument l'adresse de la structure au lieu de la structure elle-même.

Modifiez toutes les fonctions qui ont des arguments de type « structure » pour éviter les passages par valeurs. Faites attention à définir des passages par référence lorsque cela est nécessaire et des passages par « const pointeur » sinon.

Pour rappel, la notation C « `X->Y` » est une version plus courte, et surtout plus fréquente, pour « `(*X).Y` ». Personne n'utilise cette seconde syntaxe, préférez donc la première.

2. `do_open()` et `do_close()`

L'objectif est ici de modulariser le code : séparer/expliciter les fonctions d'ouverture et de fermeture de la base d'images. Cela simplifiera l'organisation du code par la suite.

Vous devez écrire les prototypes (dans le fichier `imgStore.h`) et les définitions (dans le fichier `tools.c`) de `do_open()` et `do_close()`.

La fonction `do_open()` prend comme arguments :

— le nom de fichier de la base d'image (`const char *`) ;
— le mode d'ouverture du fichier (`const char *`, par exemple `"rb"`, `"rb+"`) ;
— la structure `imgst_file` dans lesquelles stocker les données lues (n'oubliez pas la section 1. précédente ; nous ne le rappellerons plus).

La fonction doit

1. ouvrir le fichier ;
2. lire le contenu du « *header* » ; et
3. lire le contenu des « métadonnées ».

La fonction doit retourner la valeur zéro si tout s'est correctement passé, et sinon un code d'erreur approprié en cas de problèmes. Comme dans les semaines

précédentes, vous devez traiter tous les cas d'erreurs possibles dans cette fonction et utiliser les définitions de `error.h` (voir les tests unitaires, plus bas).

La fonction `do_close()` prend un seul argument de type structure `imgst_file` et doit fonction doit fermer le fichier (qu'elle contient). Elle ne retourne pas de valeur. Là aussi, pensez à traiter le cas d'erreur possible : si le fichier (`FILE*`) est `NULL`. Cela doit être un réflexe qui va de soit lorsque vous écrivez du code, et notamment lorsque vous utilisez un pointeur. Nous ne le rappelerons plus dans la suite.

3. Prototyper et définir `do_delete()`

Le moment est venu d'implémenter la fonctionnalité qui permet d'effacer une image. L'idée est la suivante : on n'efface pas réellement le contenu de l'image, car ce serait trop coûteux (surtout en temps). En fait, la taille sur disque du fichier base d'images ne diminue jamais, même lorsqu'on demande d'« effacer » une image de la base.
Concrètement, on « efface » une image en

1. trouvant la réference de l'image avec le même nom dans les « métadonnées » ;
2. en invalidant la référence en écrivant la valeur zéro dans `is_valid` ;
3. ajustant les informations du « *header* ».

Les changements doivent être faits d'abord sur les « métadonnées » (mémoire, puis disque), puis sur le « *header* » en cas de succès.
Note : pour des raisons de compatibilité entre systèmes (« ___offsets__ »), il est préférable de récrire toute la « `struct` » sur disque, plutôt que simplement les champs modifiés.

En utilisant cette approche, il est facile de rajouter une nouvelle image : il suffit simplement de trouver la première entrée dans les « métadonnées » qui n'est pas valide et l'utiliser. À ce moment, la position dans la « métadonnées » sera réutilisée pour la nouvelle image, mais les photos elles-mêmes seront rajoutées en fin du fichier base d'images.

La fonction `do_delete()` prend les arguments suivants :

— un identifiant (chaîne de caractères, `const char *`) ;
— une structure `imgst_file`.

Pour écrire les changements sur disque, il faut d'abord se positionner au bon endroit dans le fichier en utilisant `fseek()` (voir le cours et `man fseek`) et `fwrite()`.

Il faut évidemment traiter correctement le cas où la référence dans la base d'image n'existe pas (et qu'il n'y a pas d'invalidation).

N'oubliez pas de mettre à jour le « *header* » si l'opération est un succès. Il faut également augmenter de 1 le numéro de version (`imgst_version`), ajuster le

nombre d'images valides stockées (`num_files`) et également écrire le « *header* » sur disque.

Comme toujours, il faut s'occuper des cas d'erreurs qui pourraient se produire dans votre fonction. `do_delete()` retourne zéro en cas de success et un code défini dans `error.h` en cas d'erreur (l'erreur `FILE_NOT_FOUND` est à utiliser lorsqu'aucune image valide correspondant à l'identifiant donné n'a pu être trouvée).

4. Adapter `imgStoreMgr.c`

La première chose à faire est de penser à ajouter la description de la commande « *delete* » dans l'aide :

```
delete <imgstore_filename> <imgID>: delete image imgID from imgStore.
```

Ensuite modifiez aux endroits pertinents l'ouverture, la lecture et la fermeture du fichier : remplacez-les par des appels à `do_open()` et `do_close()`.

Modifiez également tous les appels qui doivent l'être (pointeurs ; section 1. ci-dessus).

Dans la commande `do_create_cmd()`, maintenant que `myfile` est passé par référence à `do_create()`, nous allons pouvoir ajouter un affichage informatif lors de la création d'une base d'images : ajoutez un appel à `print_header()` après l'appel à `do_create()`, si ce dernier a réussi.

Complétez enfin le code de `do_delete_cmd()`. Si la `imgID` reçue est vide ou de longueur (`strlen(imgID)`) supérieure à `MAX_IMG_ID`, `do_delete_cmd()` doit retourner l'erreur `ERR_INVALID_IMGID` (définie dans `error.h`).

N'oubliez pas, enfin, d'adapter votre `Makefile`.

STYLE !

Note importante : écrire du code propre, lisible par tous, est aussi important. Il semble que ce ne soit pas le cas de tout le monde jusqu'ici `;-)`. Nous vous proposons donc à partir de maintenant de strictement p.ex. le style définit par `astyle -A8`.

`astyle` est un programme qui a justement pour but de reformater les codes sources pour suivre un standard (`man astyle` pour plus de détails).

Voir aussi la cible `style` dans le `Makefile` fourni au départ.

Tests

Tests « à la main »

Il nous semble préférable de commencer à tester votre code sur un cas simple, que vous maîtriser.

Utiliser pour cela **une copie** du fichier `test02.imgst_static` des semaines précédentes (nous insistons : **faites-en une copie !!**) pour voir son contenu, supprimer une, deux image(s). Vérifier à chaque fois en regardant le résultat avec `list`.

Faites également tous les tests de cas problèmatiques auxquels vous pouvez penser.

Tests fournis

Comme d'habitude, nous fournissons également, *à bien plaire*, *quelques* tests, ainsi que, comme toujours, le moyens de les lancer à partir de rien via `make feedback`. Nous rappellons encore que ce `make feedback` est normalement à utiliser pour une vérification *minimale* **finale** de votre travail, avant de rendre. Préférez auparavant faire des tests **locaux** directement sur votre machine, via `make check` (et y compris plus de tests que vous aurez vous-même ajoutés si nécessaire).

Tests unitaires

Comme nous avançons dans le projet, nous pensons qu'il est maintenant temps d'avoir également des tests unitaires. Nous vous en fournissons un exemple, `tests/unit-test-cmd_args.c`, qui teste, de façon assez exhaustive pour cette fois, les cas d'erreurs des arguments de quelques fonctions.

Nous vous conseillons fortement d'éditer ces fichiers pour y ajouter vos propres tests, voire d'en créer d'autres au fur et à mesure de vos développements. Cela devrait se faire assez simplement en ajoutant vos propres valeurs ou vos propres lignes de code aux tests déjà fournis, ou en copiant ce fichier et en vous en inspirant. Il n'est, à notre avis, pas nécessaire de tout comprendre, en tout cas dans un premier temps, à ce fichier.

Ceci dit, pour ceux qui veulent aller plus loin, les principales fonctions de test disponibles dans l'environnement que nous utilisons (Check) sont décrites là-bas : https://libcheck.github.io/check/doc/check_html/check_4.html#Convenience-Test-Functions. Par exemple, pour tester si deux `int` sont égaux, utilisez alors la « fonction » `ck_assert_int_eq` : `ck_assert_int_eq(a, b)`.

Nous avons également défini les « fonctions » suivantes dans `tests.h` :

- `ck_assert_err_none(int erreur)` : teste si l'erreur `erreur` est `ERR_NONE` (c.-à-d. correspond à un retour de fonction sans erreur ; voir `error.h`) ;
- `ck_assert_bad_param(int erreur)` : teste si l'erreur `erreur` est `ERR_BAD_PARAMETER` (c.-à-d. correspond à une erreur d'une fonction ayant reçu un mauvais paramètre ; voir `error.h`) ;
- `ck_assert_ptr_nonnull(void* pointeur)` : teste si le pointeur `pointeur` n'est pas `NULL` ;

— `ck_assert_ptr_null(void* pointeur)` : teste si le pointeur `pointeur` est `NULL`.

Pour faire l'édition de lien avec la bibliothèque Check, il faut ajouter les options `-lcheck -lm -lrt -pthread` à l'édition de liens ; par exemple :

```
gcc unit-test-machintruc.o machin.o truc.o error.o -lcheck -lm -lrt -pthread -o unit-test-ma
```

Sur certaines architectures, il faut aussi ajouter la bibliothèque `-lsubunit`. Le fichier `Makefile` fourni comprend déjà toutes ces bibliothèques. Pensez à le mettre à jour si nécessaire pour votre architecture. Il est peut-être aussi nécessaire d'installer cette bibliothèque sur votre machine (p.ex. sur Linux : `sudo apt install check`).

Pour lancer les tests unitaires : le plus simple est de les intégrer à `make check`, en les ajoutant aux `CHECK_TARGETS`. Par exemple :

```
CHECK_TARGETS += tests/unit-test-cmd_args
```

Après, bien sûr, comme n'importe quel programme C, il faut qu'il compile correctement et donc ajouter les règles correspondantes. Par exemple :

```
tests/unit-test-cmd_args.o: tests/unit-test-cmd_args.c tests/tests.h \
  error.h imgStore.h
tests/unit-test-cmd_args: tests/unit-test-cmd_args.o $(OBJS)
```

En ajoutant les 3 règles précédentes, `make check` devrait aussi lancer ces tests unitaires (après les « black-box » tests).

Enfin, nous rappellons enfin que ce n'est pas parce que 100% des tests fournis ici passent que vous aurez 100% des points. D'abord parce que ces tests ne sont peut-être pas exhaustifs (cela fait aussi partie du travail de programmeur que de penser aux tests), mais aussi et surtout (comme indiqué dans la page expliquant le barème du cours (ici en HTML et ici en PDF) ; à lire absolument si ce n'est pas déjà fait) parce que nous accordons une grande importance à la qualité de votre code, qui sera donc évalué par une relecture par des humains (et non pas à l'aveugle par une machine).

Rendu

Comme pour les semaines passées, vous n'avez pas à rendre de suite le travail de cette semaine de projet. Celui-ci ne sera à rendre qu'à la fin de la semaine 8 (délai : le dimanche 25 avril 23:59) en même tant que le travail des semaines 4 à 7.

Nous vous conseillons cependant toujours de travailler régulièrement et faire systématiquement ces commits réguliers, au moins hebdomadaires, lorsque votre travail est opérationnel. Cela vous aidera à sauvegarder votre travail et à mesurer votre progression.

Pour sauvegarder la partie correspondant à cette semaine, ajoutez les nouvelles versions des fichiers `tools.c`, `imgStore.h`, `imgStoreMgr.c`, `imgst_list.c`, `imgst_create.c` et `Makefile`, ainsi que le nouveau fichier `imgst_delete.c` et vos éventuels nouveaux tests dans le répertoire `done/` de votre GitHub, puis « pousser » le résultat vers GitHub (`commit` plus `push`).