# EPFL CS212 : ImgStore – Système de fichiers orienté images — 07 : refactoring 2 (allocation dynamique) et préparation de `read` et `insert`

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

## 07 : refactoring 2 (allocation dynamique) et préparation de `read` et `insert`

### Introduction

L'objectif de cette semaine est double (pensez à vous répartir le travail) :

1. incorporer les nouveaux concepts du langage C vus pendant le cours – notamment l'allocation dynamique de mémoire ;
2. préparer la mise en place des fonctions de manipulation d'images (`read` et `insert`) qui seront finalisées dans deux semaines.

## Matériel fourni

Il n'y a pas de nouveau fichier C cette semaine. Comme la semaine passée, vous construisez votre projet en vous basant sur les versions des semaines précédentes et créerez par vous-même les nouveaux fichiers nécessaires.

La seule chose que nous vous fournissons cette semaine est un petit test `07.test-dynamic.sh` pour quelques tests basiques sur les « *image store* » dynamiques.

## Travail à faire

L'objectif de la semaine consiste en les étapes suivantes, décrites plus en détail plus bas :

1. l'allocation dynamique de la partie « métadonnées » ;
2. la création d'images de tailles réduites (formats « *small* » et « *thumbnail* ») ;
3. adapter votre `Makefile`.

### 1. Allocation dynamique

Actuellement, le champ `metadata` de la structure de données `imgst_file` consiste en un tableau de taille fixe, alloué statiquement donc.

Cette définition a l'avantage de la simplicité de programmation, puisque que toute la partie `metadata` *a priori* utilisable est déjà stockée dans la même structure (et contiguë en mémoire). Malheureusement, cette simplicité a un coût énorme du point de vue de l'occupation mémoire et de la flexibilité : le nombre maximum d'images stockées dans un fichier de base de données d'images occupe déjà toute la place et est par ailleurs lié à une constante de compilation (`MAX_MAX_FILES`). Idéalement, le nombre maximum d'images devrait être laissé à la discrétion de l'utilisateur du programme, et non à celle de celui/celle qui l'a écrit.
Cette approche a un troisième défaut – rédhibitoire celui-là : le programme ne fonctionne correctement que si la valeur `MAX_MAX_FILES` ne change pas entre la création du fichier et son utilisation. Tout changement de cette valeur, suivi d'une recompilation de l'outil de commandes, rendrait impossible le décodage des images des bases précédentes !

L'objectif est donc d'offrir flexibilité à l'utilisateur et robustesse à l'usage. Pour ce faire, nous allons mettre en place l'utilisation **dynamique** de la partie `metadata`.

Les changements doivent avoir lieu a plusieurs endroits :

— tout d'abord, le champ `metadata` de la structure `imgst_file` doit devenir un pointeur ;
— ce pointeur doit ensuite être initialisé en allouant la mémoire de manière dynamique à chaque endroit qui initialise une structure de type `imgst_file`, à savoir dans `do_create()` et dans `do_open()` ;
— mettez enfin à jour toute partie du code devant l'être suite aux changements précédents.

Pour allouer la mémoire, utilisez la fonction `calloc()`. Si nécessaire, revoyez le cours ou regardez la « *man page* » pour plus d'explications et n'oubliez pas que la fonction peut retourner une erreur. En cas d'erreur, votre fonction doit retourner `ERR_OUT_OF_MEMORY` (voir `error.h` et `error.c`).

Une fois ces changements faits, la constante `MAX_MAX_FILES` change de rôle : elle ne définit plus le nombre maximum d'images de la table, et ne détermine plus non plus la quantité de mémoire requise pour pouvoir exécuter le programme. Il s'agit maintenant juste d'une valeur *maximale*, spécifiée en fonction des besoins extrêmes.
Changez sa valeur à 100000. Est-ce que votre programme tourne toujours ? (il devrait...)

### 2. Création et gestion d'images dérivées

Une des fonctions principale de `imgStore` est de gérer de manière transparente et efficace les différentes résolutions d'une même image (pour rappel : dans ce projet, nous aurons la résolution d'origine et les résolutions « *small* » et « *thumbnail* »).

Dans une première étape, cette semaine, vous devez implémenter une fonction `lazily_resize()`. Son nom suggère son emploi : en informatique, « *lazy* » correspond à une stratégie couramment utilisée qui consiste à différer le travail jusqu'au dernier moment, dans l'idée d'éliminer du travail inutile.
(**Note de la part des enseignants :** ne pas confondre « informatique » et « études en informatique » `;-)`).

Cette fonction a trois arguments :

— un entier correspondant à un code interne d'une des résolutions dérivées de l'image : `RES_THUMB` ou `RES_SMALL` (voir `imgStore.h`) ;
(note : si c'est `RES_ORIG` qui est passé, la fonction ne fait simplement rien et ne retourne pas d'erreur (`ERR_NONE`)) ;
— une structure `imgst_file` (celle avec laquelle on travaille) ;
— et un index, de type `size_t`, position/index de l'image à traiter.

Elle doit implémenter la logique suivante :

— vérifier la légitimité des arguments, et retourner si nécessaire une valeur d'erreur appropriée (voir `error.h` et `error.c`) ;
— si l'image demandée existe déjà dans la résolution correspondante, ne rien faire ;

— dans les autres cas, il faut d'abord créer une nouvelle variante de l'image spécifiée, dans la résolution spécifiée ;
— ensuite, copier le contenu de cette nouvelle image à la fin du fichier `imgStore` ;
— finalement, mettre à jour le contenu de la `metadata` en mémoire ainsi que sur disque.

Pour créer la nouvelle variante de l'image, vous utiliserez la bibliothèque `VIPS` introduite en semaine 1 (voir aussi la section suivante).
Nous vous conseillons par ailleurs d'avoir une approche modulaire (ceci devrait toujours être le cas !!)

Votre solution devra consister en :

— un nouveau fichier `image_content.c` qui implémente la fonction `lazily_resize()` ;
— un nouveau fichier `image_content.h` qui prototype `lazily_resize()` ;
— les changements nécessaires de votre `Makefile` (voir ci-dessous).

### 3. Bibliothèque VIPS et modification du Makefile

Un des objectifs de ce cours-projet, déjà commencé en semaines 1 et 2, est d'apprendre à incorporer dans votre solution des bibliothèques complexes (parfois très complexes) ; en l'occurrence ici la bibliothèque VIPS, pour comprimer les images.

Pour vous aider, veuillez regarder comme exemple l'utilisation de VIPS en semaine 2 (`thumbify.c`), ainsi que toute documentation sur Internet concernant cette bibliothèque.

Les images n'étant pas ici stockées de façon individuelle dans des fichiers séparés, vous ne pouvez pas utiliser `vips_image_new_from_file()`. Il faudra utiliser `fread()`, puis `vips_object_local_array()` et `vips_jpegload_buffer()` pour charger l'image originale en mémoire.
Pour utiliser `vips_object_local_array()`, il faudra avoir au préalable avoir alloué un `VipsObject` au moyen de `vips_image_new()` (équivalent le l'allocation effectuée par votre `vips_image_new_from_file()` en semaine 1). Si nécessaire (warning), pour transformer un `ptr` de type `VipsImage*` en `VipsObject*`, utilisez `VIPS_OBJECT( ptr )`.

De même, pour écrire l'image en nouvelle résolution sur le disque (en fin de fichier imgStore), vous ne pouvez pas utiliser `vips_image_write_to_file()`, mais utilisez plutôt `vips_jpegsave_buffer()` et `fwrite()`.

Nous insistons sur le fait que c'est une partie, *non négligeable*, de **votre** travail de cette semaine que de comprendre et adapter le code fourni en semaines 1 et 2.

## Tests

L'allocation de mémoire dynamique peut se tester en créant des fichiers de différentes tailles (en utilisant la fonction `do_create()` dans vos tests), ainsi qu'en lisant le fichier `test02.imgst_dynamic` fourni en semaine 4 (il est déjà dans `done/tests/data`).

Pour le moment, pour tester la fonction `lazily_resize()`, il vous faut modifier de manière temporaire votre programme principal `imgStoreMgr.c` (ou, certainement mieux, travailler sur un nouveau fichier C, de test, partant d'une copie de `imgStoreMgr.c`) afin de tester sa fonctionnalité (qui sera intégrée plus tard). Vous pouvez par exemple modifier la fonction `do_list_cmd()` pour y insérer les appels à `lazily_resize()` que vous voudrez tester, entre autant d'appels à `do_list()` que vous jugerez nécessaires (pensez alors à changer le mode de `do_open()` et, si vous ne travaillez pas sur une copie, le remettre à sa valeur correcte une fois vos tests terminés !). Testez sur une ou des **copie(s)** de `test02.imgst_dynamic` et assurez-vous à chaque fois que le fichier a crû du nombre attendu d'octets (vrai accroissement si les images n'existaient pas encore dans les résolutions demandées, et aucune modification si les résolutions avaient déjà été demandées une première fois).

Et comme toujours, nous fournissons également, *à bien plaire*, un `make feedback` ; lequel 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 aussi plus de tests que *vous* aurez vous-même ajoutés.

## Outils

Avec l'arrivée des pointeurs, et surtout cette semaine de l'allocation dynamique, il peut être utile de connaître d'autres outils que le débogueur pour trouver des erreurs liées à la mémoire. Nous avons pour cela écrit un autre document d'aide, spécifiquement sur les outils de débogage de mémoire. Nous vous recommandons cependant *d'également continuer* à utiliser un débogueur : les nouveaux outils présentés sont *complémentaires* de celui-ci.

## Rendu

Comme vous le savez, le travail de cette semaine constitue, avec le travail des semaines 4 à 6, le **premier rendu** du projet. Le délai pour le rendre (avec le travail des semaines 4 à 6) est fixé au **dimanche 25 avril 23:59** ; mais veillez à ne pas accumuler de retard et bien vous répartir le travail.

Pensez donc à faire des commits réguliers de votre travail et n'oubliez pas de mettre à jour votre fichier `time.csv`.

Pour effectuer ce rendu, le plus simple est de faire

```
make submit1
```

dans votre répertoire `done/`. Mais **attention** :
avant la soumission, vérifiez avoir bien ajouté (`git add`), validé (`git commit`) et transmis (`git push`) toutes vos dernières versions de tous vos fichiers sources `.c` et `.h`, ainsi que le `Makefile`. Merci par contre de **ne pas** ajouter les fichiers `.o`, ni les exécutables.

Avant de soumettre, veuillez également retirer (ou commenter) tous les appels à `printf()` superflus que vous auriez pu ajouter (p.ex. pour déboguer). Nous vous conseillons d'ailleurs d'utiliser plutôt le flux d'erreur `stderr` (`fprintf(stderr,`) pour vos messages supplémentaires, car nous ne testons pas son contenu.

Ce qui sera considéré comme rendu sera ce que l'on trouvera dans (la branche `master` de) votre dépôt à la date indiquée ci-dessus et marqué d'une étiquette (« *tag* ») `projet01_NB`. C'est ce que fait la commande

```
make submit1
```

(mais vous pouvez aussi faire cela différemment, par vous-même, comme expliqué plus bas).

La raison pour laquelle nous étiquetons (`git tag`) votre contenu est pour vous permettre de continuer à travailler et prendre de l'avance : ainsi si votre dépôt contient à la date de rendu une version en avance sur le rendu et qui n'est pas fonctionnelle, ce n'est pas grave, nous ne prendrons que la dernière version pour laquelle vous aurez fait

```
make submit1
```

Ne faites donc pas de « `make submit1` » sur une version qui ne compile pas !..
Mais vous pouvez faire plusieurs fois « `make submit1` » si vous vous appercevez d'une erreur. Nous ajouterons simplement une nouvelle étiquette `projet01_NB`, avec `NB` augmenté de 1.

Si vous préférez faire l'étiquettage vous-même, par exemple pour étiqueter un ancien commit ou parce que vous êtes sur une machine sur laquelle `submit.sh` ne fonctionne pas, vous pouvez aussi bien sûr le faire :

```
# pour être sûr d'où vous en êtes :
git status -suno

# aussi pour voir où vous en êtes, choisir votre commit à étiqueter :
git log --graph --oneline --all --decorate

# pour voir les tags que vous auriez déjà mis :
git tag -l


# POUR FAIRE LE RENDU LUI-MÊME :
# choisissez bien X (p.ex. 1) et Y (numéro de commit)
# p.ex. :
```

```
# git tag projet1_1 e8ec3e8

git tag projet1_X  Y
git push --tags
git push
```

Faites bien attention de faire ces trois dernières commandes (en particulier les *deux* `push`).
Allez ensuite vérifier sur GitHub que vous y avez bien un tag et qu'il correspond bien au commit que vous voulez.