# TD Outils 3 : débogage

## Programme « jouet » à corriger

1. Compilez le code présent dans le répertoire TD\_valgrind du dépôt git de l’UE

<https://forge.univ-lyon1.fr/Alexandre.Meyer/L2_ConceptionDevApp>

1. Exécutez-le
2. Regardez le code de main.cpp pour comprendre ce qui est calculé
3. Ce code a différents problèmes : lenteur, et deux types de mauvaises gestions de la mémoire.

**Objectif** : en utilisant valgrind qui va fournir des informations sur l'exécution du programme, vous devez résoudre les problèmes de lenteur et de mémoire : fuite de mémoire (memory leak), accès invalide dû à des débordements, etc.

### Mémoire perdu (memory leak) et accès mémoire non autorisé

Par défaut, valgrind permet de vérifier si les allocations mémoires se passent bien. En particulier, de savoir si vous ne débordez pas d'un tableau (par exemple : int t[10]; t[12]=14;), de savoir si tous les new correspondent à des delete dans le programme, etc.

$> valgrind --tool=memcheck --leak-check=full ./prog

==20689== Invalid write of size 4

==20689== at 0x804859D: main (main.cpp:23)

==20689== Address 0x4286FA0 is 0 bytes after a block of size 24 alloc'd

==20689== at 0x4022A55: operator new[](unsigned int)

==20689== by 0x8048541: main (main.cpp:18)

Ceci indique qu'il y a des problèmes d'écriture mémoire non autorisée dans main.cpp à la ligne 23, ce qui est sûrement dû à un problème de taille d'allocation ou d'indice de tableau. Pour cela valgrind vous indique la ligne d'allocation mémoire de la variable mise en cause (ici le tableau f1) à la ligne 18.

1. **Faites la correction nécessaire pour résoudre ce problème, ainsi que celles nécessaires pour résoudre d'autres problèmes d'allocation et d'accès.**

De plus, dans la rubrique “LEAK SUMMARY” à la fin vous trouverez que 24 octets sont perdus, car ce programme ne fait pas le bon nombre de delete.

1. Faites le bon nombre de libérations mémoire et vérifiez que votre programme est propre en relançant valgrind.
2. **Faites la même chose pour votre module Image afin de vérifier si vous n'avez pas de fuite mémoire**.

### Optimisation de code

Pour optimiser votre code, vous devez savoir où votre programme passe le plus de temps :

$> valgrind --tool=callgrind ./prog

$> callgrind\_annotate callgrind.out.20092

La 1re ligne a produit un fichier de statistique : callgrind.out.20092 (nom automatiquement généré, change à chaque exécution). La 2e ligne permet de visualiser les statistiques du fichier :

--------------------------------------------------------------------------

Ir file:function

--------------------------------------------------------------------------

2,028 Calcul.cpp:intAdd(int, int) [/home/.../prog]

570 Calcul.cpp:intMul(int, int) [/home/.../prog]

Ce programme passe 2028 fois dans intAdd et 570 fois dans intMul. C'est donc la fonction intAdd qu'il faut optimiser en priorité.

1. **Regardez et changez le code de intAdd pour avoir un programme plus efficace**. Relancer valgrind pour voir le gain de performance obtenue. Faire éventuellement d'autres modifications du code pour gagner encore plus. Pour vous rendre compte du gain apporté, vous pouvez refaire l'expérience avec une valeur MAX plus grande.



## Débogage de fonctions du module Image

Trois fonctions d'entrée/sortie sauver (sauver une image dans un fichier), ouvrir (ouvrir une image depuis un fichier) et afficherConsole (afficher les valeurs des pixels sur la console) sont données dans le fichier TD\_moduleImage/IOimage.cpp du projet git de l’UE.

Vous devez tout d'abord les incorporer dans votre code (i.e. les ajouter à la classe Image). **Des bugs ont été placés intentionnellement dans ces trois fonctions :-? !!** Vous devez les trouver et les corriger en utilisant GDB.

1. Créez un deuxième programme principal nommé mainExemple.cpp qui contient le code suivant.

**Attention, vous devez lancer l'exécutable à partir du dossier racine avec la commande bin/exemple.** Ceci est valable pour les trois exécutables du module Image.

mainExemple.cpp (produisant l'exécutable bin/exemple)

#include "Image.h"

int main() {

Pixel rouge (205, 9, 13);

Pixel jaune (242, 248, 22);

Pixel bleu (120, 193, 246);

Image image1 (64,48);

image1.effacer(bleu);

image1.dessinerRectangle(6, 8, 26, 21, rouge);

image1.setPix(16,14,jaune);

image1.setPix(46,33,jaune);

image1.sauver("./data/image1.ppm");

Image image2;

image2.ouvrir("./data/image1.ppm");

image2.dessinerRectangle(23, 18, 37, 28, rouge);

image2.dessinerRectangle(34, 26, 50, 36, jaune);

image2.sauver("./data/image2.ppm");

return 0;

}

1. Pour trouver les bugs des fonctions, vous pouvez tester votre programme avec une petite image (ex. de taille 3×2) et dérouler le code pas à pas en utilisant un débogueur et en affichant les contenus des images.

Si vous ne maîtrisez pas bien l’utilisation d’un débogueur, regardez dans la doc du projet git de l’UE.

<https://forge.univ-lyon1.fr/Alexandre.Meyer/L2_ConceptionDevApp/-/blob/master/doc/debug.md>

## Pour aller plus loin avec valgrind

* valgrind permet d'avoir bien d'autres informations. Tapez valgrind -h pour avoir une liste.
* [Site officiel](http://valgrind.org/)
* [Documentation](http://valgrind.org/docs/manual/manual.html)