

Etude de l'outil de gestion de test Valgrind

CHARPIN Etienne
ISIMA
Université Clermont Auvergne
Année Universitaire 2022 - 2023

Enseignant : David HILL
Date du rendu : 17 Avril 2023

ISIMA

1 Rue de la Chebarde - CS 60032 - 63178 Aubière
Tel : 04 73 40 50 00
Site web : isima.fr

Sommaire

1	Introduction	2
2	Intérêts de la gestion des tests	2
3	Valgrind	3
3.1	Présentation de Valgrind	3
3.1.1	Mise en place dans un tel projet	3
3.1.2	Affichage des résultats des tests	4
4	Conclusion sur l'outil	6

Table des figures

1	Logo de Valgrind	3
2	Implémentation du makefile	4
3	Résultat de l'exécution non optimisé	5
4	Résultat de l'exécution optimisé	6

1 Introduction

Lors de notre projet de deuxième année et pour le compte du cours d'Architecture Logicielle et Qualité nous avons l'opportunité d'étudier et d'approfondir un outil utile au développement de notre choix. L'intérêt d'un tel exercice est de pouvoir découvrir un outil méconnu ou bien de comprendre plus en profondeur un outil peu connu.

Dans un premier temps mon choix s'était porté sur l'outil de gestion de qualité de code Sonar Cube mais après plusieurs recherches concernant son implémentation, cela me semblait trop compliqué à mettre en place sur un projet développé en c++.

Mon second choix s'est alors porté sur l'outil de test qui m'était peu connu : Valgrind.

2 Intérêts de la gestion des tests

La gestion de tests et l'utilisation d'outils de profilage sont deux éléments clés pour garantir la qualité et la performance d'un programme en C++. La gestion de tests permet de vérifier que chaque fonctionnalité du programme fonctionne correctement, tandis que l'utilisation d'outils de profilage permet de mesurer et d'optimiser les performances du programme.

La gestion de tests consiste à écrire des tests pour chaque fonctionnalité du programme afin de s'assurer qu'elle fonctionne comme prévu. Les tests doivent être exécutés à chaque modification du code pour vérifier que les modifications n'ont pas introduit de nouveaux bogues. Un bon système de gestion de tests doit couvrir toutes les fonctionnalités du programme et être automatisé pour faciliter l'exécution régulière des tests.

L'utilisation d'outils de profilage est également importante pour garantir les performances du programme. Les outils de profilage permettent d'analyser le code et d'identifier les zones qui prennent le plus de temps d'exécution. Les développeurs peuvent alors optimiser ces zones pour améliorer les performances du programme. Les outils de profilage peuvent également aider à détecter les fuites de mémoire, les erreurs de segmentation et les autres problèmes qui peuvent nuire aux performances du programme.

Bien que ces 2 outils soient parfois considéré comme complémentaires, nous allons ici nous concentrer sur le profilage et l'optimisation de performance grâce à Valgrind.

3 Valgrind

3.1 Présentation de Valgrind



FIGURE 1 – Logo de Valgrind

Valgrind est un outil open-source de débogage et de profilage de programmes informatiques. Il est conçu pour aider les développeurs à détecter et à corriger les erreurs de mémoire, les fuites de mémoire, les problèmes de performance et les incidents liés à l'utilisation de la mémoire dynamique.

Valgrind fonctionne en interceptant les appels système relatifs à l'allocation de mémoire et en exécutant le code dans un environnement simulé. Cela permet à Valgrind de détecter les erreurs de mémoire telles que les lectures et écritures illégales, les accès à la mémoire libérée, les fuites de mémoire mais aussi les variables non initialisés pouvant causer des problèmes.

En plus de sa capacité à détecter les erreurs de mémoire, Valgrind peut également être utilisé pour profiler le code et identifier les ralentissements de performance. Il peut mesurer le temps d'exécution de chaque fonction, identifier les appels de fonction coûteux et détecter les blocages d'attente dans le code.

3.1.1 Mise en place dans un tel projet

Dans notre projet, j'ai décidé d'utiliser Valgrind pour profiler les problèmes liés à l'allocation de ressource en c++ mais aussi les petites erreurs pouvant être coûteuses ou parfois engendrer des erreurs d'exécution.

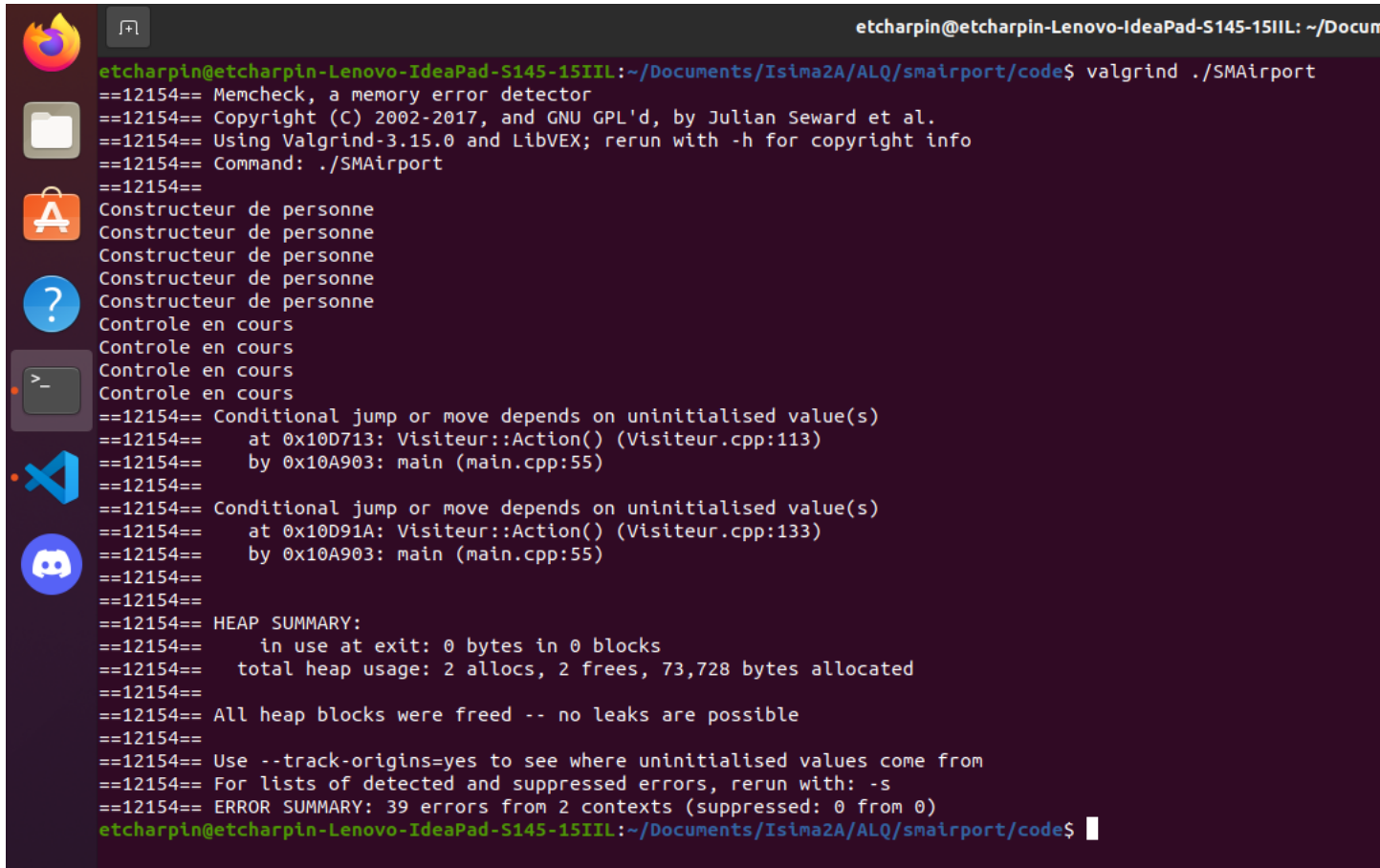
Afin de pouvoir préparer l'utilisation d'un tel outil nous devons modifier la chaîne de compilation dans le makefile afin d'ajouter l'option **-g** pour pouvoir profiler le code à exécuter. Ensuite, il nous reste juste à appeler Valgrind en exécutant dans notre terminal la commande : **valgrind ./nomdeprogramme**.

```
code > M makefile
1 SRC=main.cpp
2 SRC+=Aeroport.cpp ZoneBagage.cpp Boutique.cpp Personne.cpp Douanier.cpp Visiteur.cpp Position.cpp
3 EXE=SMAirport
4
5 CC=g++
6 CFLAGS=-Wall -Wextra -g
7 LDFLAGS=-lm
8
9 OBJ=$(addprefix build/,${SRC:.cpp=.o})
10 DEP=$(addprefix build/,${SRC:.cpp=.d})
11
12 all: ${OBJ}
13     $(CC) -o $(EXE) ^ $(LDFLAGS)
14
15 build/%.o: %.cpp
16     @mkdir -p build
17     $(CC) $(CFLAGS) -o $@ -c $<
18
19 clean:
20     rm -rf build core *.o
21     rm -rf $(EXE)
22
23 -include $(DEP)
```

FIGURE 2 – Implémentation du makefile

3.1.2 Affichage des résultats des tests

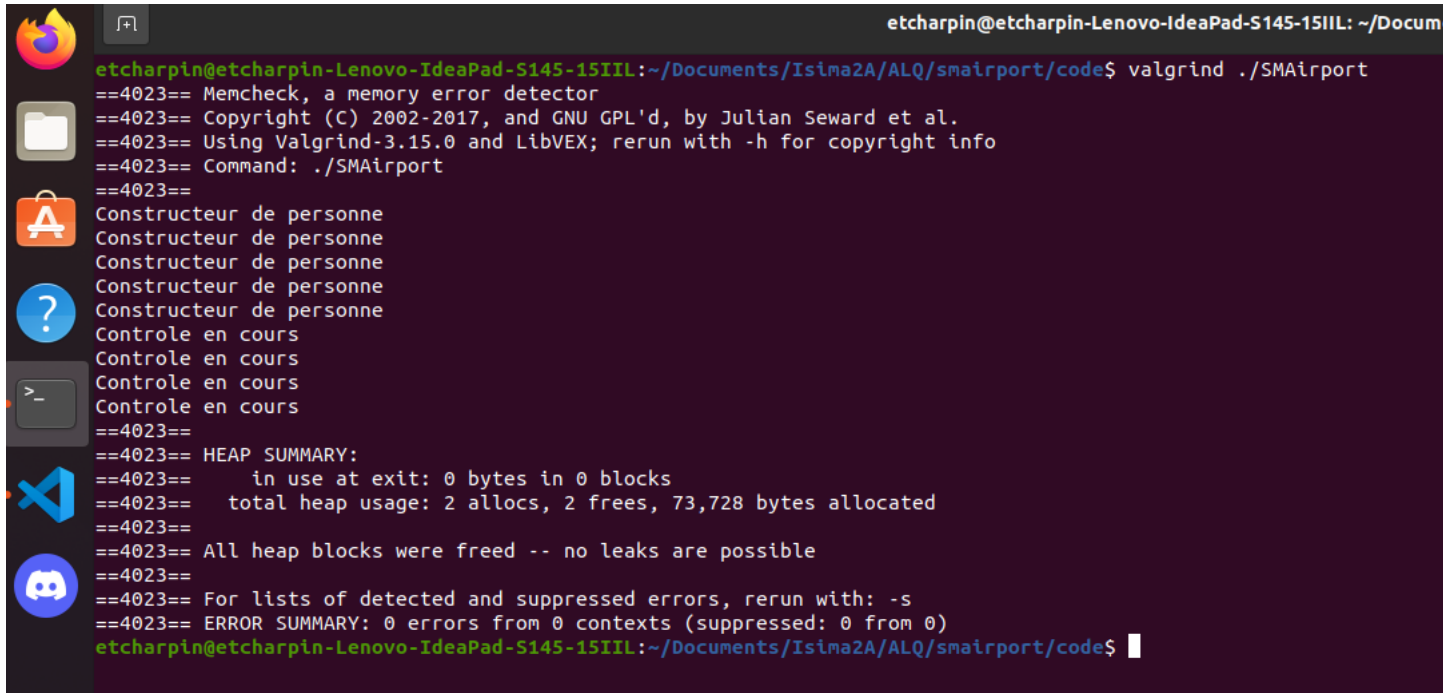
Une fois l'exécution du programme lancée avec Valgrind, nous pouvons étudier les résultats sur le terminal pendant le programme. Dans un premier temps nous voyons apparaître les erreurs dites de contexte de l'application. Ces erreurs affichées le sont une seule fois sur le terminal même si celle-ci sont récurrentes comme dans une boucle. Ces erreurs minimes se trouvent au niveau de la rapidité d'exécution et non d'une fuite mémoire pouvant altérer le fonctionnement de l'application. Comme nous le voyons sur l'image ci-dessous, le nombre total d'erreur est affiché en bas en fin de profilage. A coté de ce nombre d'erreurs, nous trouvons le nombre de fuites mémoires de notre programme. Lors de ma premier utilisation de Valgrind je ne possédais pas de problème de fuite mémoire. Sinon le nombre de blocs et d'octets perdus seraient affichés.

A terminal window with a dark background and light-colored text. The window title is 'etcharpin@etcharpin-Lenovo-IdeaPad-S145-15IIL: ~/Documents/Isima2A/ALQ/smairport/code\$'. The terminal shows the execution of 'valgrind ./SMAirport'. The output includes copyright information, a list of memory errors (e.g., 'Constructeur de personne', 'Contrôle en cours'), and a summary of heap usage. The summary indicates that all heap blocks were freed and no leaks are possible. The terminal also shows a sidebar with various application icons on the left.

```
etcharpin@etcharpin-Lenovo-IdeaPad-S145-15IIL:~/Documents/Isima2A/ALQ/smairport/code$ valgrind ./SMAirport
==12154== Memcheck, a memory error detector
==12154== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==12154== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==12154== Command: ./SMAirport
==12154==
Constructeur de personne
Constructeur de personne
Constructeur de personne
Constructeur de personne
Constructeur de personne
Contrôle en cours
Contrôle en cours
Contrôle en cours
Contrôle en cours
==12154== Conditional jump or move depends on uninitialised value(s)
==12154==    at 0x10D713: Visiteur::Action() (Visiteur.cpp:113)
==12154==    by 0x10A903: main (main.cpp:55)
==12154==
==12154== Conditional jump or move depends on uninitialised value(s)
==12154==    at 0x10D91A: Visiteur::Action() (Visiteur.cpp:133)
==12154==    by 0x10A903: main (main.cpp:55)
==12154==
==12154==
==12154== HEAP SUMMARY:
==12154==    in use at exit: 0 bytes in 0 blocks
==12154==    total heap usage: 2 allocs, 2 frees, 73,728 bytes allocated
==12154==
==12154== All heap blocks were freed -- no leaks are possible
==12154==
==12154== Use --track-origins=yes to see where uninitialised values come from
==12154== For lists of detected and suppressed errors, rerun with: -s
==12154== ERROR SUMMARY: 39 errors from 2 contexts (suppressed: 0 from 0)
etcharpin@etcharpin-Lenovo-IdeaPad-S145-15IIL:~/Documents/Isima2A/ALQ/smairport/code$
```

FIGURE 3 – Résultat de l'exécution non optimisé

Après l'étude de cette première observation de Valgrind, j'ai eu la possibilité de savoir grâce à Valgrind d'où venais le problème mais aussi des indications sur comment le résoudre. L'outil m'indiquant une possibilité de variable non initialisé pouvant ralentir l'exécution, je me suis donc empressé de régler le problème avant de relancer le programme avec Valgrind. Avec cette deuxième exécution, j'ai pu voir que je ne trouvais plus d'erreurs de contexte lors de l'exécution et que je ne possédais toujours pas de fuite mémoire.

A terminal window on a Linux system. The prompt is 'etcharpin@etcharpin-Lenovo-IdeaPad-S145-15IIL: ~/Documents/Isima2A/ALQ/smairport/code\$'. The command 'valgrind ./SMAirport' has been executed. The output shows Valgrind's startup messages, followed by five 'Constructeur de personne' calls and four 'Contrôle en cours' calls. Then, a 'HEAP SUMMARY' is printed, showing 0 bytes in use at exit, 2 allocations, 2 frees, and 73,728 bytes allocated. It also states 'All heap blocks were freed -- no leaks are possible'. Finally, an 'ERROR SUMMARY' shows 0 errors from 0 contexts. The prompt returns to 'etcharpin@etcharpin-Lenovo-IdeaPad-S145-15IIL: ~/Documents/Isima2A/ALQ/smairport/code\$'.

```
etcharpin@etcharpin-Lenovo-IdeaPad-S145-15IIL: ~/Documents/Isima2A/ALQ/smairport/code$ valgrind ./SMAirport
==4023== Memcheck, a memory error detector
==4023== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==4023== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==4023== Command: ./SMAirport
==4023==
Constructeur de personne
Constructeur de personne
Constructeur de personne
Constructeur de personne
Constructeur de personne
Contrôle en cours
Contrôle en cours
Contrôle en cours
Contrôle en cours
==4023==
==4023== HEAP SUMMARY:
==4023==    in use at exit: 0 bytes in 0 blocks
==4023==   total heap usage: 2 allocs, 2 frees, 73,728 bytes allocated
==4023==
==4023== All heap blocks were freed -- no leaks are possible
==4023==
==4023== For lists of detected and suppressed errors, rerun with: -s
==4023== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
etcharpin@etcharpin-Lenovo-IdeaPad-S145-15IIL: ~/Documents/Isima2A/ALQ/smairport/code$
```

FIGURE 4 – Résultat de l'exécution optimisé

4 Conclusion sur l'outil

Pour conclure sur l'outil utilisé lors de ce projet SMA d'architecture logicielle et qualité, j'ai décidé d'approfondir un outil plutôt que d'en apprendre un nouveau et cela me semblait utile comme connaissance afin de pouvoir comprendre le fonctionnement et la mise en œuvre d'un programme utilisant le profilage pour optimiser ses performances. Dans ce cas précis, Valgrind m'a permis de corriger mes erreurs de contexte relatives à la non initialisation de variables pouvant altérer le comportement de mon application. Ceci n'aurait pas été possible sans un outil de profilage qui cherche à optimiser chaque partie de l'exécution. Je pense que ce genre d'outil me sera utile pour tout projet future afin de rendre un code optimisé et sans fuites mémoires