UNIVERSITÉ DU QUÉBEC

MÉMOIRE

PRÉSENTÉ À

L'UNIVERSITÉ DU QUÉBEC À CHICOUTIMI

COMME PARTIELLE EXIGENCE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

HENI DHIAB

GESTION DE MEMOIRE DYNAMIQUE ET IMPLICITE PAR ASPECT POUR DES LOGICIELS C++

HIVER 2010

03016788

Sommaire

Le langage C++ est un langage destiné pour des projets complexes et de taille importante. En effet, il permet la programmation sous de multiples paradigmes, comme la programmation procédurale, la programmation orientée objet et la programmation générique. Sa facilité d'utilisation et son efficacité, soutenues avec une bonne flexibilité et une bonne portabilité, font de lui un langage performant et le 3ème utilisé dans le monde.

Toutefois, les dispositifs de sécurité sont soit absents, soit mal supportés par la programmation C++; malheureusement, la flexibilité et le contrôle explicite des données que C++ offre au programmeur représentent une importante source de vulnérabilité et d'erreurs critiques. Certes, les librairies C++ sont fournies aux programmeurs avec des fonctions pour la gestion des privilèges, la gestion des fichiers, ou encore la gestion du réseau, mais négligent les problèmes de sécurité. Notons que la *sécurité* est celle de la gestion de mémoire et celle des types tels que les débordements de mémoire tampon, les erreurs de format de chaîne et les mauvaises conversions de type.

Malgré la disponibilité de nombreux livres et documents sur lesquels un programmeur peut s'appuyer pour développer un code sécuritaire, des erreurs de mise en œuvre ayant des conséquences graves pour la sécurité existent toujours dans le code source. Plus le nombre de lignes augmente, plus un contrôle *manuel* des violations de la sécurité devient difficile. N'étant pas appuyée par une gestion implicite, la gestion de la mémoire en C++ s'est vue toujours comme un exercice difficile, voire parfois périlleux. Cette contrainte a toujours posé un défi considérable.

Ce travail de recherche vise à développer une méthode et un outil qui permettront une gestion et un débogage implicite de la mémoire C++. Nous nous sommes inspirés des limites de débogueurs de mémoire connues dans le domaine pour réviser certains choix de structuration et de conception de notre outil. La programmation par aspect, incarnant la séparation des préoccupations et formant un support méthodologique de modélisation et de programmation, a servi de support pour l'implémentation d'un outil générique et nous avons pu regrouper l'aspect mémoire en un seul aspect. Et en étant soutenus par la programmation par aspect, nous avons pu aussi contourner les limites du modèle de programmation par objet.

Nous avons opté pour une solution raffinée des solutions déjà proposées (*Purify* et *Memcheck&Valgrind*). Cette dernière inclut i) un aspect permettant la sécurisation des allocations et des désallocations des espaces mémoire C++, ii) un aspect de détection et de traitement des erreurs qui sont dues à une mauvaise gestion de mémoire C++, et iii) un aspect capable de calculer le besoin d'une application C++ en mémoire ainsi que son temps d'exécution. L'outil incarnant la solution proposée a été expérimenté sur une application existante et a été comparé avec l'outil *Purify* selon six critères bien identifiés (Chapitre 5).

Table des matières

Chapitre1 : Introduction......1
1.1. Énoncé de la problématique......5
1.2. Principe de solution......5
1.3.Contribution......6
1.4.Structure du mémoire......7
Chapitre 2: Gestion de la mémoire : un état de l'art......8
2.1.Gestion du cycle de vie des objets......8
2.1.1. Introduction......8
2.1.2. Gestion de mémoire implicite......10
2.1.3. Les erreurs dues à une mauvaise gestion de mémoire......132
2.1.3.1. Les erreurs d'accès (Access Errors)......*12*
2.1.3.2. Les fuites de mémoires (Memory Leaks)......*13*
2.1.3.3. Les dépassements de tampon (Buffer Overflow)......*15*
2.2.Les débogueurs de mémoire dynamiques......16
2.2.1. Introduction......16
2.2.2. Purify......17
2.2.2.1. Les erreurs d'accès de mémoire (Memory Access Errors)......*17*
2.2.2.2. Les erreurs de fuite de mémoire (Memory Leaks Errors)......*22*
2.2.2.3. Purify expérimental......*23*
2.2.2.4. Résultats......*23*
2.2.3. Memcheck......27
2.2.3.1. Valgrind......*27*
2.2.3.2. Valid-value (V) bits......*28*
2.2.3.3. Valid-adress (A) bits......*28*
2.2.3.4. Notes et limitations......*28*
2.3.Gestion de mémoire et C++......29
2.3.1. Gestion de la mémoire en C++......30
2.3.2. Problèmes actuels avec les compilateurs C++......30
2.4.Conclusion......311
Chapitre 3: Préoccupations et aspects......33
3.1.Introduction......33
3.2.Séparation des préoccupations......34
3.3.Programmation par aspect (POA)......37
3.3.1. Descriptions......37
3.3.2. Fonctionnement et Motivations......38
3.3.3. Intégration des aspects......40
3.3.4. Programmation par aspect avec AspectC++......41
3.4.Conclusion......44
3.5.Objectif de la recherche......45

Chapitre 4: Développement d'un outil de gestion implicite et de débogage de mémoire basé sur la programmation par aspect (AspectC++Debogger)....................47
4.1.Introduction 47
4.2.Gestion de mémoire par aspect....................47
4.2.1. AspectC++....................48
4.2.1.1. Les points de jointure (Join point) 49
4.2.1.2. Les coupes transverses (Pointcut)....................49
4.2.1.3. Composition de coupes transverses (Pointcut composition) 50
4.2.1.4. Les coupes transverses nommées (Named Pointcut)....................50
4.2.1.5. Les expressions de correspondance (Match expressions) 50
4.2.1.6. Les conseils (Advice) 51
4.2.2. Les éléments affectant la consommation de mémoire dynamique en C++...52
4.2.2.1. Les membres de la classe JointPoint....................53
4.2.2.2. Exemple de collection de données lors d'une allocation et d'une désallocation....................53
4.2.2.3. Collection des noms de classes 54
4.3.Outil de gestion et de débogage de mémoire par aspect 55
4.3.1. Aspect calculant le temps d'exécution de différentes applications 55
4.3.1.1. Mesure du temps d'exécution....................55
4.3.1.2. Détection des portions de codes à mesurer avec Aspect C++ 56
4.3.2. Aspect évaluant la mémoire utilisée lors de l'exécution d'une application......57
4.3.2.1. Détection des appels de l'opérateur new....................58
4.3.2.2. Changement du comportement de l'opérateur new en allouant deux espaces mémoires supplémentaires 58
4.3.2.3. Extraction de la taille de l'espace mémoire alloué....................59
4.3.3. Aspect pour sécuriser les allocations et les désallocations de mémoire...........60
4.3.3.1. Allocation sécurisée....................60
a.Extraction du pointeur retourné par l'opérateur new....................61
b.Extraction du nom de fichier et du numéro de ligne 61
c.Coordination du conteneur avec les allocations dynamiques atteintes dans l'application cliente....................62
d.Initialisation des informations accompagnant toute allocation....................63
e.Initialisation du Tampon _Overflow....................63
4.3.3.2. Validité d'un pointeur....................64
4.3.3.3. Désallocation sécurisée d'une zone mémoire 65
4.3.4. Aspect de détection des erreurs de fuites de mémoire....................66
4.4.Conclusion 67
Chapitre 5: Etude de cas et résultats expérimentaux....................68
5.1.Application à analyser et motivations 69
5.1.1. Choix de l'application et Motivations....................69
5.1.2. Le Jeu Asteroids....................69
5.2.AspectC++Debogger et l'intégration de ses principaux composants 71

5.2.1. Visual Studio 2005 et AspectC++Add-In....70
5.2.2. Integration du jeu Asteroids sous Visual Studio 2005....71
5.2.3. AspectC++ Debogger et manuel d'utilisation....71
5.3.Expérience et résultats....74
5.3.1. Les fuites de mémoire....74
5.3.2. Les erreurs d'accès....76
5.3.3. Les dépassements de tampon....77
5.4.Discussion....81
5.4.1. Habilité de détection et de traitement des différents types d'erreurs de gestion de mémoire C++....80
5.4.2. Coexistance d'une gestion implicite et explicite de la mémoire....80
5.4.3. Rlentissement de l'application cliente et diminution de ses performances....81
5.4.4. Généricité de l'outil....81
5.4.5. Stabilité d'AspectC++....81
5.4.5.1.Problèmes de compilateur....82
5.4.5.2.Templates C++ non reconnus....82
5.5.Conclusion....82
Chapitre 6: Conclusion....87
Bibliographie....87
Annexe....92

Liste de Figures

Figure 1-1 : Fréquence d'utilisation du langage C++ 2
Figure 2-1 : Disposition de la mémoire 11
Figure 2-2 : Cas de présence d'une fuite de mémoire 14
Figure 2-3 : A et B avant le dépassement de tampon 16
Figure 2-4 : A et B après le dépassement de tampon 16
Figure 2-5 : Purify étiquette les états de mémoire par couleur 18
Figure 2-6 : Erreur d'écriture hors de la limite d'un tableau 19
Figure 2-7 : Disposition des espaces mémoire 20
Figure 2-8 : Structure InfoPointeur et allocation de données avec deux espaces supplémentaires 21
Figure 2-9 : Accès légal à une zone mémoire inappropriée 21
Figure 2-10 : Capture d'écran du résumé des erreurs de *DebugTest* 24
Figure 2-11 : Capture d'écran pour les détails des erreurs de *DebugTest* 25
Figure 2-12 : Utilisation non-sécurisée d'un tableau de bit 27
Figure 2-13 : Memcheck ne détecte pas les erreurs d'accès hors de la limite d'un tableau 29
Figure 3-1 : Les exigences non-fonctionnelles traversent la modularisation fonctionnelle du... système 35
Figure 3-2 : Intégration du code des composantes et des aspects pour former le système final 39
Figure 1-1 : Aspect Tracing s'exécutant avant le démarrage de l'exécution de n'importe quelles fonctions du programme principale 41
Figure 4-1 : Exemple avant tissage de l'asp ect. 48
Figure 4-2 : Aspect *Malloccounter* 51
Figure 4-3 : Effet de l'aspect après tissage 52
Figure 4-4 : Modèle de gestion de mémoire *(AspectC++Debogger)* 55
Figure 4-5 : Structure du conteneur et les informations ajoutées pour chaque pointeur 58
Figure 4-6 : Changement du comportement de l'opérateur *new* 59
Figure 4-7 : Déclaration d'un aspect avec accès à une variable contextuelle 60
Figure 4-8 : Modèle d'allocation accompagnant la DAC 63
Figure 5-1 : Capture d'écran du jeu *Asteroids* 71
Figure 5-2 : Activer *AspectC++Add-In* sous le projet Asteroids 73
Figure 5-3 : Asteroids, Visual Studio 2005 et AspectC++Add-In 73
Figure 5-4 : Interface utilisateur de *AspectC++Debogger* 74
Figure 5-5 : Fuite de mémoire dans *AsteroidField.cpp* 76
Figure 5-6 : Purify signale une fausse alerte d'une fuite de mémoire dans *AsteroidGame.cpp* 77
Figure 5-7 : Cas d'une erreur d'un accès légal à une zone mémoire inappropriée 78
Figure 5-8 : Erreur de dépassement de tampon dans *d3dfile.cpp* 80
Figure 5-9 : Injection d'un test dans *d3dfile.cpp* 80

Liste de tableaux

Tableau 5-1 : Résultat de détection des fuites de mémoire 76
Tableau 5-2 : Résultat de détection des erreurs d'accès...78
Tableau 5-3 : Résultat de détection des dépassements de tampon...79
Tableau 5-4 : Tableau récapitulatif de l'étude de cas de Purify et d'AspectC++Debogger....84

À ma mère que Dieu lui accorde Sa miséricorde
À mon père, Mouna, ma sœur, Dali, mon frère, Maryouma, et mes nièces
À mes ami(e)s

Remerciements

Au nom de **Dieu** le Clément le Miséricordieux

Je remercie tout d'abord, **Dieu,** qui m'a donné le courage et la force de réaliser ce mémoire et qui a exaucé mes prières en me donnant la chance de pouvoir vous le remettre aujourd'hui.

Je dédie ce mémoire à ma chère et regrettée mère, Moufida Maalej, paix à son âme et que **Dieu** lui accorde Sa miséricorde et l'accepte dans ses Paradis.

Je remercie mon père, Monji Dhiab, un **Homme** qui a toujours été présent pour sa famille, bon, modeste et qui m'a toujours fais confiance. Les sacrifices que tu as fais à mon égard sont inestimables. Je prie **Dieu** pour que je puisse t'apporter main forte et être à la hauteur de tes espérances. Je t'aime papa! Je remercie Mouna, une femme au grand cœur et à qui je souhaite donner autant d'amour que possible. Que **Dieu** vous protège.

Je remercie mon frère, ami et coach, Mouhamed Ali El Ghazel (Dali). Une personne qui m'a aidé à trouver le droit chemin et qui a toujours été présente quand j'en avais besoin. J'espère être à la hauteur de tes espérances, et qu'un jour je puisse être là pour toi aussi.

Un gros câlin à ma douce et adorable sœur, Jihène Dhiab. Une sœur que j'admire et qui ne cesse de me donner le sourire. Tu es la sœur, la mère et la femme envers laquelle mon respect et mon amour sont inépuisables. Que **Dieu** te protège.

Je remercie mon grand frère, Atef Dhiab. Un frère que durant toute sa vie n'a pas arrêté de s'inquiéter et de prendre soin de moi! En effet, je suis chanceux de t'avoir comme frère. Tu es un exemple à suivre. J'espère être à la hauteur de tes espérances. Un grand merci à toi et à Maryouma, que **Dieu** vous protège.

Un grand merci à mes nièces, Aya, Lina et Elaa pour tout le bonheur et la joie qu'elles nous apportent. Que **Dieu** vous protège.

J'exprime ma sincère gratitude à mon directeur de recherche M Hamid Mcheick tant sur le plan du mémoire en lui-même que sur le plan du soutien moral afin que je puisse terminer ce travail. Un très grand merci et chapeau bas pour votre conscience professionnelle, votre générosité et votre disponibilité.

De même, je remercie aussi tout le corps professoral et administratif du département d'Informatique et de Mathématiques de l'Université du Québec à Chicoutimi pour tous les enrichissements que j'ai eus ainsi que leur aide inestimable.

Finalement, je remercie également mes amis qui m'ont aidé durant ce travail. Merci Bouhmid, merci Sabry, merci Ghazi et merci Achraf. Merci pour votre hospitalité! Merci à tous ceux qui m'ont aidé à finir mon travail de recherche.

Chapitre 1 : Introduction

À travers les dernières décennies, le domaine du génie logiciel s'est forgé une place importante au sein du monde moderne. Le logiciel est rendu l'une des principales clés de réussite de l'entreprise d'aujourd'hui. En effet, les logiciels voient leurs performances se multiplier, il s'agit d'une révolution. Au sein de cette croissance, le langage C++ s'est toujours approprié une place importante vu sa fréquence d'utilisation élevée. La plupart des logiciels, que ce soit libre (*open source*) ou non, sont écrits en C++ [ISO/IEC, 2003]. Cette panoplie de logiciels comprend les systèmes d'exploitation (Linux [Linux, 2009], FreeBSD [FreeBSD, 2008]), des pilotes de périphériques, des jeux vidéo, le tout complémenté par des serveurs internet et des bases de données (MySQL), etc.

Le C++ est considéré comme un langage performant, flexible et fortement portable [Tlili, 2009]. N'étant pas soutenu par dispositifs de sécurité, les services que offre le langage C++ deviennent donc une source de *vulnérabilité* et *d'erreurs* critiques telles que les erreurs de gestion de mémoire, les erreurs de format de chaîne et les mauvaises conversions des types. D'ailleurs, les librairies C/C++ sont fournies aux programmeurs avec des fonctions pour la gestion des privilèges, la gestion des fichiers, la gestion du réseau, etc. Ces fonctions sont conçues avec soin pour fournir la flexibilité et les caractéristiques de performance, mais *négligent les problèmes de sécurité* [Debbabi et Tlili, 2009]. Notons que la *sécurité* est celle de la gestion de mémoire et celle des types tels que les débordements de mémoire tampon, les erreurs de format de chaîne, et les mauvaises conversions de type.

Malgré la disponibilité de nombreux livres et documents sur lesquels un programmeur peut s'appuyer pour développer un code sécuritaire, des erreurs de mise en œuvre ayant des conséquences graves pour la sécurité existent toujours dans le code source [Debbabi et Tlili, 2009]. Les fonctions des bibliothèques C/C++ peuvent être considérées non sécurisées et par conséquent doivent être utilisées avec précaution. Plus le nombre de lignes augmente, plus un contrôle *manuel* des violations de la sécurité devient difficile. Même les programmeurs les plus qualifiés ont tendance à commettre des erreurs involontaires qui rendent le code vulnérable et potentiellement exploitable d'un point de vue sécurité. Par conséquent, les

techniques *automatisées* de détection des erreurs et de vulnérabilité sont devenues très utiles dans un diagnostic de sécurité et d'erreurs [Tlili, 2009]. Dans la littérature, il existe une gamme d'approches de détection des erreurs qui sont principalement classées sous l'analyse *dynamique* et l'analyse *statique* [IBM Rational Purify, 2008] [Sioud, 2006] [Aiken, 2007] [Tlili, 2009]. L'analyse dynamique surveille l'exécution du programme afin de trouver les erreurs une fois qu'elles se produisent. La précision est sa principale clé. Pour atteindre un haut degré d'efficacité d'une analyse dynamique, le programme cible devrait être exécuté avec des entrées suffisantes de test afin d'observer l'ensemble de ses comportements possibles. L'exploration de ces entrées (chemins) d'exécution nécessite la définition difficile d'un grand nombre de cas de test. D'autre part, l'analyse statique opère sur le code source sans l'exécution du programme afin de prédire les erreurs potentielles lors du temps d'exécution. Contrairement à son homologue dynamique, l'analyse statique n'introduit pas la notion du temps d'exécution. Dans ce mémoire, notre effort de recherche se concentre sur le développement d'une technique automatisée afin de détecter et ensuite résoudre les erreurs de gestion de mémoire au sein de C++ et ceci en se basant sur une analyse dynamique.

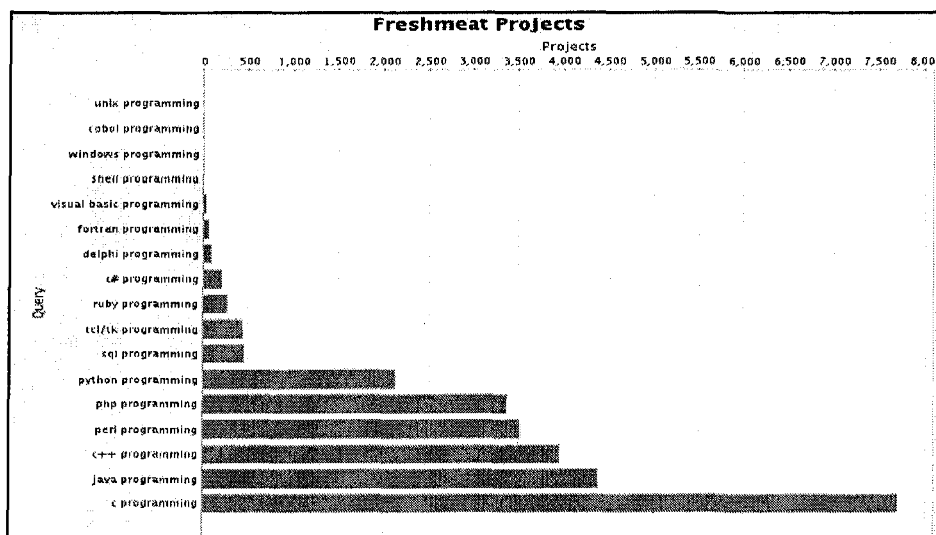


Figure 1-1: Fréquence d'utilisation du langage C++ [Freshmeat, 2007]

Les erreurs de gestion de mémoire sont principalement les fuites de mémoire (*Memory Leaks*), les dépassements de mémoire (*buffer Overflows*) et les erreurs d'accès aux zones mémoire (*Memory Access Errors*). Ces types d'erreurs sont essentiellement dus aux bogues reliés aux allocations et aux désallocations des espaces mémoire : dans un premier temps, la

mémoire est gérée par le programmeur lui-même en utilisant les fonctions d'allocations et de libérations qu'un langage de programmation peut fournir, on parle alors de gestion explicite de la mémoire. Par exemple, en C/C++, on fait appel aux fonctions *alloc()*, *malloc()*, les opérateurs *new*, *delete*, *free()* et les méthodes constructeurs et destructeurs pour les objets. Dans un deuxième temps et avec des approches différentes, la mémoire est gérée implicitement par les langages de programmation via le ramasse miette (*Garbage Collector* : *GC*), une diminution au niveau des erreurs de gestion de mémoire a été atteinte [Vassev et Paquet, 2006]. Et enfin, un outil d'analyse est apparu et connu sous le nom de débogueur de mémoire (*Memory Debugger*). En effet, même si certains programmes se voient écrits avec des langages munis d'un ramasse miette, l'utilisation d'un débogueur de mémoire s'est vue fortement conseillée afin de remédier efficacement aux erreurs déjà citées ci-dessus [Detlefs, 1994] [Boehm, 2002] [Tlili, 2009]. Une explication détaillée concernant le degré de gravité de ces erreurs et leurs manières d'affecter le programme durant son exécution sera présentée dans le Chapitre 2.

Le développement d'outils de débogage de mémoire est vite rendu une obsession. *Purify* [IBM Rational Purify, 2008] est un exemple parmi tant d'autres. C'est un programme utilisé par les développeurs de logiciels afin de détecter les erreurs d'accès de mémoire, en particulier, celles écrites en C/C++. *Purify* permet une analyse dynamique des applications. Lorsqu'un programme est relié à ce dernier, un code de vérification est automatiquement inséré à l'exécutable en analysant et en s'ajoutant au code objet [CiteSeer, 2008]. Si une erreur de mémoire se produit, le programme imprimera sur l'écran de sortie l'emplacement exact de l'erreur, l'adresse de mémoire en question, et d'autres informations importantes.

Aussi, on peut prendre l'exemple de *Valgrind* qui est un outil d'analyse dynamique. Il permet le débogage des zones mémoire allouées dynamiquement et dans certain cas, celles allouées d'une manière statique. Il permet aussi la détection des fuites de mémoire et l'analyse de performance (*profiling*). Il fut à l'origine conçu pour être une version libre de *Purify* sous GNU/Linux, sur architecture x86. Il a aujourd'hui évolué, en devenant un framework générique pour la création d'outils dynamiques comme des *checkers* et *profilers* [Valgrind, 2008]. *Valgrind* est un logiciel modulaire, parmi ces modules on trouve *Memcheck* qui permet de débusquer les failles dans un programme au niveau de l'utilisation de sa mémoire ; Il

effectue une série de tests afin de s'assurer de la bonne utilisation des pointeurs et de la conformité des accès et des libérations des zones mémoire.

Cet ensemble d'outils a permis à la gestion implicite de la mémoire de généraliser la plupart des langages orientés objet. En effet, Meyer [1988] a placé la gestion implicite de la mémoire en troisième position dans ses "sept commandements" de la programmation orientée objet. Selon les chercheurs [IBM Rational Purify, 2008] [Valgrind, 2008], les outils de débogages de mémoire facilitent énormément la tâche des programmeurs. Ces derniers n'ont plus à se soucier des erreurs de manipulation de mémoire et passer des heures de frustration à essayer de venir à bout des bogues. Ellis [1993] a établi que dans le cas des langages à gestion explicite seulement de la mémoire comme C++, les programmeurs passent plus des deux cinquième du temps accordé à l'élaboration du logiciel à déboguer les erreurs liées à la gestion de la mémoire. Certaines recherches effectuées au sein de boites renommées de développement logiciel [IBM Rational Purify, 2008] ont permis l'intégration des outils de débogage de mémoire au sein de C++. Ils ont utilisé des approches telles que le *Mark and Sweep* [Sioud, 2006], les *Red Zone* [CiteSeer, 2008] ou encore les compteurs de référence [Ellis, 1993]. Malheureusement, ces outils de débogages de mémoires connaissent certaines limites que des recherches essaient encore de surpasser, par exemple l'inhabilité à détecter les erreurs d'accès à travers des pointeurs [IBM Rational Purify, 2008]. Toute tentative d'intégration d'un débogueur de mémoire au sein du langage C++ rencontre certains problèmes et contraintes, le cas de *Purify* et *Valgrind* réunies peut être résumé comme suit :

- suite à la bonne flexibilité de manipulation des pointeurs qu'offre les langages C et C++, il est parfois impossible de détecter les erreurs d'accès à travers des pointeurs, par exemple *l'accès légal à une zone de mémoire inappropriée* ;
- pénalité de performance et *ralentissement de 20 à 30* fois de l'exécution d'un programme lors de son analyse ;
- absence d'une *coexistence transparente entre une gestion implicite et explicite* de la mémoire ;
- il est juste possible de détecter les fuites de mémoire pour seulement les zones mémoire retournées par la fonction *malloc* ;
- l'inaptitude de détecter les erreurs d'accès hors du bloc de mémoire dans le cas d'une *allocation statique*, par exemple les variables locales ou encore les données allouées sur une pile.

Néanmoins, les travaux chez *Pure Software* et *Valgrind-developers* restent optimistes puisqu'ils ont pu résoudre la plupart des problèmes rencontrés lors de l'intégration d'un outil de débogage de mémoire.

1.1. Énoncé de la problématique

Le langage C++ est un langage renommé pour sa puissance et son utilisation fréquente dans le domaine du génie logiciel. Néanmoins, l'absence d'une gestion implicite de sa mémoire se trouve être une grande faiblesse et remet en question sa robustesse. Vu que des chercheurs ont déjà résolu le problème de façon plus ou moins adéquate, l'objectif de la présente recherche est de raffiner ces solutions en tenant compte de leurs limites. Ce que nous proposons est comme suit :

- développer un concept théorique d'un gestionnaire de mémoire pour le langage C++ ;
- implémenter ce concept pour que le langage C++ puisse garder l'image d'un langage puissant, flexible et sécuritaire et ce, que ce soit à travers une gestion implicite, explicite, ou encore les deux de la mémoire ;
- Détecter et traiter les erreurs de gestion de mémoire qu'un programme C++ peut contenir. Il s'agit de permettre au programmeur de reconnaitre la nature de ces erreurs et leurs emplacements pour qu'il puisse procéder à d'éventuelles corrections.

1.2. Principe de solution

Essentiellement, les programmes ont besoin de partager des ressources (mémoire, imprimante, etc.), de traiter des erreurs, d'optimiser les performances, etc. Ces besoins peuvent être classifiés en deux parties : composantes fonctionnelles et l'aspect technique. La composition entre ces deux derniers dans un même programme crée des systèmes difficiles à développer, à comprendre et à maintenir [Kicsales et al., 1997] [Robillard et Murphy, 01]. Par conséquent, il faut distinguer entre une composante fonctionnelle et un aspect technique. Une composante fonctionnelle est une propriété qui peut être encapsulée dans une procédure généralisée telle qu'un objet, une méthode, une procédure ou une fonction. Cependant, un aspect technique est un module facilitant la gestion des propriétés d'objets. Il transverse plusieurs composantes fonctionnelles d'un système. En effet, i) la gestion de la mémoire, ii) le traitement des erreurs de mémoire, iii) la sécurité et enfin iv) l'optimisation des performances,

sont des aspects auxquels nous allons nous intéresser tout au long de ce mémoire. Dans le langage C++, certains de ces aspects se trouvent à être gérés par plusieurs classes peu réutilisables et ainsi couplées avec l'application, d'autres sont quasi absents. La *programmation par aspect* [Kicsales et al., 1997], incarnant la séparation des préoccupations [Tarr et al., 1999] et formant un support méthodologique de modélisation et de programmation, sera utilisée afin de nous permettre d'abstraire et de composer ces aspects (la POA[1] comme étant un support d'instrumentation et d'analyse dynamique).
Pour une solution flexible, générique de gestion de mémoire et de détection d'erreurs C++ et une intégration fiable des aspects de sécurité et d'optimisation de performances, nous proposons un outil complet de débogage de mémoire pour le C++ basé sur la programmation par aspect. L'évaluation de cet outil se portera sur chacun des points suivants :

- habilité de détection et de traitement des erreurs de gestion de mémoire C++ ;
- coexistence d'une gestion implicite et explicite de la mémoire ;
- ralentissement de l'application cliente et diminution de ses performances ;
- facilité et transparence d'utilisation ;
- généricité de l'outil ;
- stabilité de l'approche utilisée (POA-AspectC++™) pour l'implémentation de l'outil.

1.3. Contribution

Dans le cadre de ce travail, nous avons développé une méthode et un outil qu'on a appelé *AspectC++Debogger* qui gère la mémoire de toute application C++ en utilisant la programmation orientée aspect comme support d'instrumentation. Notre contribution peut être résumée par les points suivants :

1. un modèle et outil de gestion et de débogage de mémoire ;
2. un aspect pour sécuriser les allocations et les désallocations de la mémoire ;
3. un aspect pour le débogage de mémoire, détection et traitement des erreurs qui sont dues à une mauvaise gestion de la mémoire ;
4. un aspect capable de calculer le temps d'exécution de différentes applications ;
5. un aspect capable de calculer la mémoire utilisée tout au long de l'application ;

[1] *POA* : Programmation orientée aspect

6. une intégration des différents aspects avec une application existante et ceci, d'une façon transparente, une interface utilisateur conviviale serait la solution proposée.

Ces contributions seront détaillées et réalisées au Chapitres 4. Les résultats expérimentaux de notre outil seront présentés au Chapitre 5 à travers une étude de cas. Les critères de comparaisons de cette dernière sont ceux présentées en Section 1.2 de ce présent chapitre.

1.4. Structure du mémoire

Ce mémoire est divisé en six chapitres. Le présent chapitre présente une introduction à la problématique générale ainsi qu'un extrait des objectifs et solutions qu'on propose. Le Chapitre 2 présentera une revue pertinente de la littérature des différentes erreurs qui sont dues à une mauvaise gestion de mémoire, les techniques utilisées pour détecter et remédier à ces erreurs, et les différentes techniques employées au sein de C++ pour assurer la gestion de la mémoire. Le Chapitre 3 présentera la séparation des préoccupations et, plus particulièrement, la programmation par aspect. Le Chapitre 4 présentera une description détaillée de notre outil *AspectC++Debogger*. Le Chapitre 5 décrit une étude de cas d'*AspectC++Debogger* avec *Purify*. Enfin, la présente recherche se termine par une conclusion et quelques aspects de recherches futures seront proposés.

Chapitre 2 : Gestion de la mémoire : un état de l'art

2.1. Gestion du cycle de vie des objets

2.1.1. Introduction

L'exécution d'algorithmes d'une application informatique consomme essentiellement deux ressources : le temps et l'espace mémoire. En machine, l'espace en question, où toute création et gestion d'objets, de variables et de structures de données est effectuée, peut être la mémoire vive ou la mémoire de masse persistante. Les composantes d'une application (objets, variables, structures de données) pourront être modifiées lors de son déroulement et elles seront détruites une fois cette même application achevée. Ces étapes de création, de l'utilisation et de destruction d'objets forment le cycle de vie d'un objet [Bray, 1977]. Lors de l'appel d'un objet, celui-ci est initialisé en mémoire centrale et occupera un espace mémoire qui correspondra à sa taille. Une fois l'application finie, ce même objet, devenu inutile, devra être détruit afin que l'espace mémoire utilisé soit récupéré. On parle tout simplement de gestion de la mémoire. De cette manière, on fait en sorte que des objets ou des variables, qui ne sont plus d'aucune utilité, n'occupent pas un espace mémoire qui pourrait être utile pour d'autres objets ou variables [Metropolis *et al.*, 1980]. Le principe même de cycle de vie d'une variable ou d'un objet est fondé sur le fait que toute création finit par une destruction, sinon le cycle serait incomplet [Bray, 1977]. Le fait qu'un objet ne soit plus sous le contrôle d'une application et qu'il ne soit pas encore libéré représente un gaspillage au niveau de l'espace mémoire. Cette forme de gaspillage, appelée *fuite de mémoire* [Hirzel *et al.*, 2002], est une erreur que la plupart de langages de programmation, voire la totalité, ont essayé d'en venir à bout.

Dans le domaine de la gestion de la mémoire, nous distinguons deux tâches connexes: une étape d'*allocation* de la mémoire (création) et une autre de *désallocation* (destruction ou libération). La première se résume au fait d'allouer ou de réserver de la mémoire pour des objets, des variables ou encore des structures de données, et la seconde s'occupe de libérer les espaces mémoire déjà alloués, mais non utilisés. Différentes techniques, autant pour l'allocation que pour la désallocation, ont été élaborées [Salagnac, 2004]. On parlait au début

des années cinquante d'une gestion statique de la mémoire, pour ensuite passer à une gestion dynamique vers le début des années soixante [Metropolis *et al.*, 1980].

Le fait d'allouer statiquement de la mémoire pour un programme donné signifie avoir prévu l'espace mémoire nécessaire avant même l'exécution du programme, et ceci en spécifiant la quantité de mémoire nécessaire à partir du code source général. Lors de la compilation, cet espace mémoire sera réservé dans un fichier binaire. Au chargement du programme en mémoire (juste avant l'exécution) l'espace réservé devient donc accessible. Vu que l'allocation de la mémoire s'effectue avant l'exécution de l'application, un gain au niveau des performances est atteint (les coûts de l'allocation dynamique à l'exécution sont épargnés). La mémoire statique est immédiatement utilisable. Une gestion de mémoire statique peut être vue comme la gestion la plus sécuritaire dans le sens où la quantité de mémoire consommée est constante et complètement connue avant l'exécution. Pour les programmeurs dont les besoins peuvent varier de façon imprévisible, ce genre de gestion est très inflexible et insuffisant [Metropolis *et al.*, 1980]. Prenons l'exemple d'une application informatique qui a besoin de 50 entiers comme entrée initiale. Le programmeur alloue un tableau statique pouvant contenir 100 entiers, il prévoit le cas où l'application aurait à augmenter le nombre de ces entrées. Si un jour l'application prend de l'ampleur et nécessitera un nombre d'entrées beaucoup plus grand, une redéfinition des espaces alloués serait nécessaire ; Le cas où cette situation se retrouve partout dans un code source constitue une grave problématique. Par conséquent, il est important d'avoir une gestion dynamique de la mémoire pour éviter une redéfinition obligatoire des espaces mémoire d'un programme (limite de gestion statique de la mémoire).

Contrairement à l'allocation qui ne peut se faire qu'explicitement, la libération quant à elle peut se faire de deux manières : soit explicitement (manuelle), soit implicitement (automatique). Dans le cas d'une *désallocation explicite*, le programmeur se retrouve à avoir un contrôle direct sur la libération de la mémoire. Aucun espace mémoire ne pourra être libéré sans des instructions venant directement du programmeur. La plupart des langages de programmation offrent des fonctions afin de permettre cette désallocation explicite. On peut citer les fonctions *delete* et *free* dans le cas du langage de programmation C++. Un principal avantage de ce genre de désallocation est qu'il est plus facile au programmeur de comprendre exactement ce qui se passe. Mais cela peut nécessiter une quantité importante de code source qui fera partie significativement de n'importe quelle interface de module. Ceci rend les

bogues de gestion de mémoire plus fréquents. Plusieurs langages utilisent cette technique tels que C-C++, ADA.

En revanche, dans le cas d'une *désallocation implicite*, l'espace mémoire non utilisé est automatiquement récupéré. Ce type de gestion de mémoire est un service qui est considéré comme une partie ou une extension du langage de programmation. Les entités qui assurent la gestion de mémoire implicite, connues sous le nom de *Garbage Collector,* réalisent habituellement leurs tâches en libérant les blocs de mémoire non accessibles à partir des variables du programme (par exemple les variables qui ne peuvent être atteintes à partir des pointeurs) [Sioud, 2006]. Plusieurs langages utilisent cette technique, parmi lesquels on cite Java, Lisp, Eiffel, Modula III, etc.

Dans la section suivante, la gestion implicite via le *Garbage Collector* et ses limites seront décrites. L'ensemble d'erreurs engendrées suite à une mauvaise gestion de mémoire sera détaillé. Et enfin, les débogueurs de mémoire seront introduits comme la solution à ces problèmes et leurs rôles au niveau de la gestion de la mémoire seront expliqués.

2.1.2. Gestion implicite de la mémoire

Sureté et sécurité sont des termes que des langages de programmation orientés objet comme Java, Lisps, Eiffel et Modula III ont utilisé pour gagner leur popularité. En effet, Meyer [1988] a placé la gestion automatique de la mémoire en troisième position dans les "sept commandements" de la programmation orientée objet. Pour Meyer [1988], une gestion dynamique de la mémoire via une désallocation automatique est une méthode clé pour garantir une bonne performance des langages de programmation. À cet effet, une entité appelée *Garbage Collector* ou *ramasse-miette* est utilisée pour assurer une gestion automatique de la mémoire. Un *Garbage Collector* est un sous-système informatique responsable du recyclage de la mémoire préalablement allouée puis inutilisée. Le premier à avoir utilisé un *Garbage Collector* fut John McCarthy en 1959 afin de remédier aux problèmes dus à une gestion manuelle de la mémoire au sein du premier système Lisp [Sioud, 2006]. Le principe de base de la récupération automatique de la mémoire est simple : une fois les objets qui ne sont plus utilisés sont détectés (c'est-à-dire les objets qui ne sont pas référencés et que le programme en cours d'exécution ne pourra jamais atteindre), on récupère l'espace mémoire que ces derniers ont utilisé [Jones et Lins, 1996] [Abdullahi et Ringwood, 1998].

Il est possible de détecter à l'avance le moment où un objet ne sera plus utilisé, il est possible de le découvrir lors de l'exécution : un objet sur lequel un programme ne maintient plus de référence (objet devenu inaccessible) ne sera plus utilisé [Sioud, 2006]. Ainsi, un *Garbage Collector* construit le graphe d'objets atteignables qui est formé de tous les objets accessibles à partir de toute racine. Ces derniers ne seront plus utilisables et devraient rester en mémoire vu que leurs valeurs sont accessibles depuis les racines. Tous les objets qui ne seront pas accessibles par le programme seront alors collectés par le *Garbage Collector* puisqu'ils sont considérés comme des *miettes*.

Dans l'exemple présenté à la Figure 2-1, E et F ne sont pas accessibles ni directement, ni par l'intermédiaire d'autres objets et ils sont considérés comme des miettes.

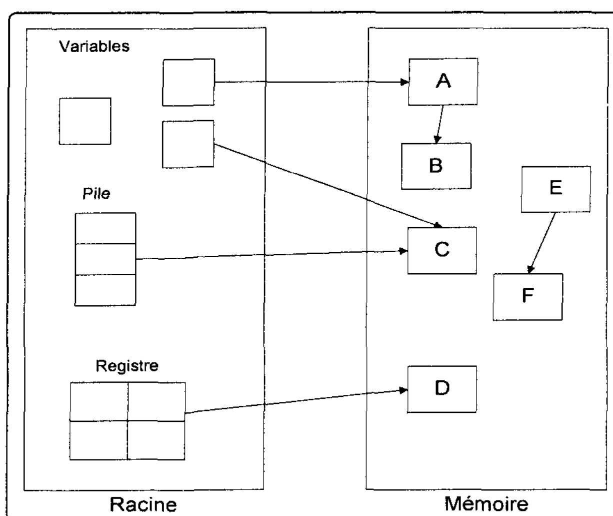


Figure 2-1 : Disposition de la mémoire

Toutes les techniques de *Garbage Collection* supposent que les miettes doivent rester stables en mémoire et qu'elles ne peuvent en aucun cas redevenir vivantes.

Un langage muni d'un *Garbage Collector* permet d'écrire des applications informatiques plus simples et plus sûres. Le fait que la mémoire soit gérée automatiquement par l'environnement d'exécution, le programmeur se libère de cette tâche. La gestion manuelle de mémoire est l'une des sources les plus courantes d'erreurs dont les trois principaux types sont i) l'accès à une zone mémoire non allouée ou qui a été libérée, ii) la libération d'une zone mémoire déjà libérée et ii) la non-libération de la mémoire inutilisée (*fuites de mémoires*).

L'utilisation d'outils et de méthodologies appropriés permet d'en réduire l'impact, tandis que l'utilisation d'un *Garbage Collector* permet de les éliminer presque complètement. En effet, les *Garbage Collector* n'arrivent pas à résoudre tous les problèmes de gestion de mémoires et ceci en tenant compte des problèmes d'allocations et de désallocation (par exemple les erreurs de dépassement de tampon *buffer overflow*). En effet, la fonction d'un *Garbage Collector* ne se limite pas à la libération seulement, mais aussi par son biais se fait l'opération d'allocation à travers une entité appelée *mutateur* [Abdullahi et Ringwood, 1998]. Cette simplification de tâche de gestion de mémoire à travers les *Garbage Collector* ne peut qu'apporter de l'aide aux programmeurs, mais peut aussi présenter quelques inconvénients, principalement au niveau des performances des programmes les utilisant. Des études montrent que dans certains cas l'implémentation d'un *Garbage Collector* augmente les performances d'un programme, dans d'autres cas le contraire se produit. Le choix de l'algorithme et des paramètres d'un *Garbage Collector* est important, car c'est de cette base qu'on peut altérer ou améliorer significativement les performances d'un programme ou encore le langage de programmation en question [Levanoni et Petrank, 2001].

Pour remédier aux failles citées ci-dessus, les spécialistes dans le domaine ont opté pour le développement d'un outil de programmation spécialisé dans la gestion de mémoire, que se soit au niveau de l'allocation ou encore de la désallocation : i) un outil qui serait capable de détecter les fuites de mémoires (*memory leaks*) et les erreurs de dépassement de tampon (*buffer overflow),* et ii) un outil qui est capable d'éviter les erreurs d'allocations et de fournir ce qu'on appelle une *allocation sécurisée*. En effet, c'est là qu'intervient l'outil de programmation connu sous le nom de débogueur de mémoire (*Memory Debugger*), dont le but n'est pas de remplacer un *Garbage Collector*, mais de le compléter.

Dans ce qui suit, nous décrirons l'utilité d'un débogueur de mémoire en élaborant une étude de deux outils qu'on considère les plus pertinents dans le domaine. Mais tout d'abord, on juge important de décrire les différentes erreurs dues à une mauvaise gestion de mémoire qu'un programme pourrait avoir à confronter. On évaluera le degré de gravité de ces erreurs ainsi que leurs manières d'affecter le programme durant son exécution.

2.1.3. Erreurs dues à une mauvaise gestion de la mémoire

2.1.3.1. Les erreurs d'accès (Access Errors)

Une simple erreur d'accès en mémoire telle que la lecture d'une zone mémoire non initialisée ou l'écriture sur une zone mémoire libérée, peut pousser un programme à agir d'une façon imprévisible ou encore causer son dysfonctionnement (*crash*). Dans un programme considéré comme non-trivial, il est presque impossible d'éliminer la totalité de ce type d'erreurs vu qu'elles produisent rarement et irrégulièrement des comportements observables [Tlili, 2009]. Même si les programmes sont examinés intensivement durant des périodes prolongées de temps, des erreurs peuvent échapper à une éventuelle détection [CiteSeer, 1990]. La combinaison de circonstances exigées (pour qu'une erreur se produise et pour que ses symptômes deviennent visibles) peut être pratiquement impossible à créer dans l'environnement de développement ou d'essai. Ceci pousse les programmeurs à passer beaucoup d'heures à la recherche de ce type d'erreurs sans écarter l'éventualité qu'un utilisateur final puisse être confronté à ces dernières en premier lieu. Dans le cas où une erreur d'accès de mémoire déclenche un symptôme observable, la dépister pourrait prendre des jours pour enfin réussir à l'éliminer. Ceci est dû au raccordement fréquent, retardé et coïncident entre la cause, généralement une corruption de mémoire, et le symptôme, typiquement un crash du à la lecture d'une donnée invalide.

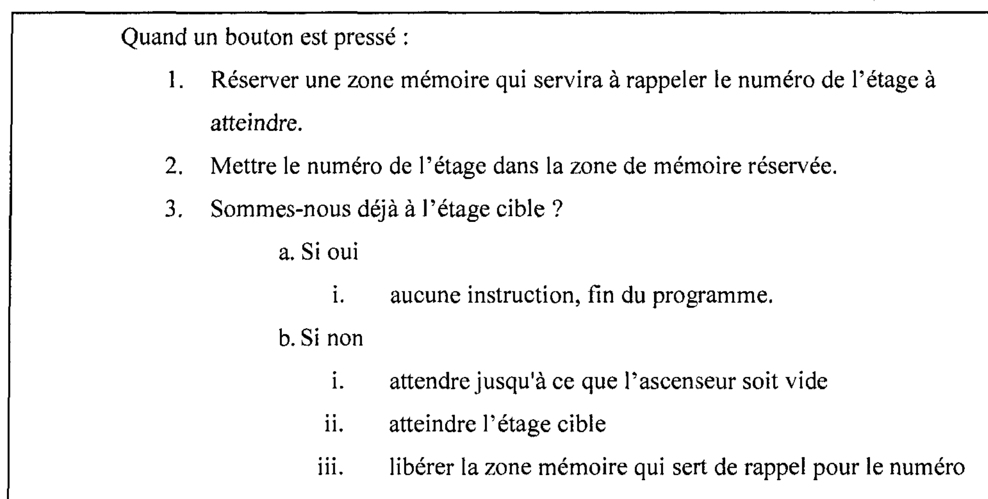
2.1.3.2. Les fuites de mémoire (Memory Leaks)

Les fuites de mémoire (*Memory Leaks*) sont les zones de mémoire allouées, mais non utilisées ou non-accessibles par le programme. La présence de ce type d'erreurs cause un ralentissement au niveau de l'exécution du programme [CiteSeer, 1990]. Ceci est principalement dû à une augmentation de pagination qui peut engendrer un manque de mémoire (*out of memory*). Les fuites de mémoire sont plus difficiles à détecter que les accès illégaux aux zones mémoire et se produisent suite à un "oubli" de libération, elles sont donc des erreurs d'inattention. Contrairement à des erreurs classiques de gestion de mémoire qui sont directement observables, les fuites de mémoire quant à elles, ne font que contribuer à la dégradation de l'exécution globale du programme. Même détectées, elles restent toujours difficiles à corriger [CiteSeer, 1990]. En effet, une libération prématurée d'un espace mémoire peut engendrer des erreurs d'accès qui engendreraient à leurs tours des problèmes aux

comportements irréguliers. C'est pour cette raison que la correction des erreurs dues à des fuites de mémoire peut parfois exiger de longues séries de tests et un suivi de protocoles d'administration de mémoire dynamique, le tout afin de ne pas nuire à un code qui est supposé être stable pendant quelques années. Les cas où la présence de fuite de mémoire est plus sérieuse incluent les cas où [CiteSeer, 2008] [Marguerie, 2009] :

- un programme est toujours fonctionnel et consomme de plus en plus de mémoire dans le temps, par exemple des tâches masquées sur des serveurs (*background tasks*), mais spécialement dans des systèmes embarqués qui sont en exécution pour plusieurs années ;
- une nouvelle allocation est souvent effectuée, par exemple la création de fenêtre d'un jeu vidéo ou d'une animation vidéo ;
- la mémoire est très limitée, par exemple dans les systèmes embarqués ou encore des dispositifs portatifs.

Sans pour autant avoir des connaissances en programmation, l'exemple de la Figure 2-2 écrit en pseudo-code, montre comment une fuite de mémoire pourrait survenir et les effets que cette dernière pourrait provoquer. Le programme fait partie d'un logiciel très simple conçu pour commander un ascenseur. Une exécution de ce programme est atteinte toutes les fois qu'une personne se trouvant à l'intérieure de l'ascenseur appuie sur un bouton pour atteindre un étage quelconque.



Quand un bouton est pressé :

1. Réserver une zone mémoire qui servira à rappeler le numéro de l'étage à atteindre.
2. Mettre le numéro de l'étage dans la zone de mémoire réservée.
3. Sommes-nous déjà à l'étage cible ?
   a. Si oui
      i. aucune instruction, fin du programme.
   b. Si non
      i. attendre jusqu'à ce que l'ascenseur soit vide
      ii. atteindre l'étage cible
      iii. libérer la zone mémoire qui sert de rappel pour le numéro

Figure 2-2 : Cas de présence d'une fuite de mémoire [Marguerie, 2009]

Une fuite de mémoire s'est glissée dans cette série d'instructions : supposons que l'utilisateur est à l'étage 2 et qu'il demande d'aller au même étage. Dans ce cas, les lignes 1, 2 et 3.a.i seront exécutées et la ligne 3.b.iii ne sera jamais exécutée vu que l'instruction 3.a.i met fin au programme. L'exécution du programme prend fin sans que l'espace mémoire réservé à la ligne 1 ne soit libéré (fuite de mémoire).

Ce cas d'erreur n'a pas un effet immédiat sur le fonctionnement du programme. Par contre, après une répétition excessive de fuite de mémoire, l'ascenseur manquera d'espace mémoire. Une telle circonstance pourrait ne jamais être découverte même avec des essais relativement complets. Les conséquences dans le cas où un dysfonctionnement de l'ascenseur se produit sont très désagréables, du moins l'ascenseur cesserait de répondre aux demandes de ses utilisateurs et ne pourra plus se déplacer d'un étage à un autre. Pire, le programme pourrait avoir besoin de mémoire pour ouvrir la porte de l'ascenseur, alors que quelqu'un pourrait être emprisonné à l'intérieure vu la non-disponibilité d'espace mémoire.

2.1.3.3. Les dépassements de tampon (Buffer Overflow)

Dans le domaine de la sécurité informatique et de la programmation, le *Buffer Overflow* représente un état anormal où un processus essaye de stocker des données au-delà des limites d'une zone mémoire réservée et de longueur constante. Ces données qu'on peut décrire comme supplémentaires peuvent se retrouver dans des zones mémoire adjacentes qui sont supposées accueillir d'autres variables et données. Une telle manipulation peut avoir comme conséquences :

- un comportement sporadique du programme (irrégulier) ;
- une exception d'accès de mémoire ;
- un arrêt du programme (crash) ;
- des résultats incorrects ;
- ou encore, suite à une provocation délibérée d'un utilisateur malveillant, une violation du système de sécurité.

Les dépassements de mémoire sont des sources de vulnérabilité et forment une base pour l'exploit des failles de sécurité informatique dans un système d'exploitation ou dans un logiciel. Les langages de programmation qui ne sont pas munis de protection contre les accès illégaux, la réécriture des données dans des zones mémoire inappropriées et l'écriture dans les

zones de limite de mémoire (comme C et C++), se voient fréquemment associés aux erreurs citées ci-dessus [Howard et Leblanc, 2002].

La Figure 2-3 et la Figure 2-4 illustrent un exemple simple de dépassement de tampon. Un programme a défini deux données élémentaires ayant deux zones mémoire adjacentes : A, un tampon *String* de 8 bits de longueur, et B, un tampon *int* de 2 bits de longueur. Initialement, A est vide, mais les bits sont mis à zéro, et b contient l'entier 3.

A								B	
0	0	0	0	0	0	0	0	0	3

Figure 2-3 : A et B avant le dépassement de tampon

Maintenant, le programme essaye de stocker la chaîne de caractères « excessive » dans le tampon A, suivie de la marque zéro pour invoquer la fin de cette dernière. Suite à une non-vérification de la longueur de la chaîne de caractères, une réécriture (*overwrites*) de la valeur de B se produit :

A								B	
'e'	'x'	'c'	'e'	's'	's'	'i'	'v'	'e'	0

Figure 2-4 : A et B après le dépassement de tampon

Bien que le programmeur n'ait aucune intention de changer le tampon B, la valeur de celui-ci fut bel et bien remplacée par un nombre formé de la chaîne de caractères du tampon A. Dans cet exemple et dans le cas d'un système *big-endian* utilisant la norme ASCII, "e" suivi d'un octet mis à zéro deviendrait le numéro 25856. Si B est supposé stocker la seule variable élémentaire et entière définie par le programme, l'écriture d'une plus longue chaîne de caractères qui dépasserait la fin du tampon A pourrait causer une erreur de segmentation qui mettrait fin au processus.

2.2. Les débogueurs de mémoire dynamiques

2.2.1. Introduction

Les erreurs d'accès et les fuites de mémoire sont des erreurs faciles à introduire dans un programme, mais difficiles à éliminer [Tlili, 2009]. N'ayant aucune facilité pour détecter les erreurs d'accès, un programmeur peut être poussé à récupérer les fuites de mémoire de façon brusque causant des comportements imprévisibles. Inversement, sans une répercussion des fuites de mémoire, les programmeurs peuvent gaspiller de la mémoire en réduisant au minimum les appels de libération dans le but d'éviter les erreurs d'accès lors d'une libération d'une zone mémoire. Par conséquent, les programmeurs accordent plus d'importance aux accès et fuites de mémoire afin d'améliorer la robustesse et la performance de leurs programmes [IBM Software, 2003].

En effet, quelques erreurs d'accès de mémoire sont détectables statiquement, par exemple l'affectation d'un pointeur à une variable de type *short*. Dans d'autres cas, les erreurs ne peuvent être détectées que lors de l'exécution du programme, par exemple l'écriture au-delà de la limite d'un tableau dynamique. Aussi, il existe des erreurs de « frappes » qui sont détectables seulement par le programmeur; par exemple, le stockage de l'âge d'une personne dans une zone mémoire prévue de contenir sa date de naissance. Les compilateurs et les outils d'analyse de code statique, *Lint* par exemple, n'ont pu cerner que les erreurs statiquement détectables. Tandis que des logiciels comme *Purify* et *Valgrind &Memcheck* se sont spécialisés dans la détection des erreurs qui se produisent en cours d'exécution.

Dans la section suivante, nous allons introduire deux outils de débogage de mémoire. Nous expliquerons leurs manières de détecter les erreurs d'accès aux zones mémoire et les erreurs de fuite de mémoire, et citerons enfin leurs limites.

2.2.2. Purify

2.2.2.1. Les erreurs d'accès de mémoire (Memory Access Errors)

a – Comment *Purify* trouve-t-il les erreurs d'accès de mémoire ?

Pour réaliser une détection presque complète des erreurs d'accès de mémoire, *Purify* cerne chaque accès de mémoire qu'un programme effectue, maintient et examine un code d'état de chaque octet de mémoire. Des accès inconsistants de mémoire génèreraient des messages

d'erreur imprimés sur écran [CiteSeer, 1990]. *Purify* garde la trace de différents états de zones mémoire :

- celles qui ne sont pas encore allouées ;
- celles qui sont allouées, mais pas encore initialisées ;
- celles qui sont allouées et initialisées ;
- celles qui ont été libérées, mais qui sont encore initialisées.

Pour maintenir à jour toutes ces informations, *Purify* utilise une table qui lui permet de dépister le statut de chaque octet de mémoire utilisé par le programme. Dans cette table, chaque octet de mémoire est représenté par deux bits. Le premier est mis en question si son octet correspondant a été alloué. Et le deuxième est mis en question dans le cas où la zone mémoire a été initialisée. *Purify* se sert de ces deux bits pour décrire quatre états différents de mémoire : rouge, jaune, vert et bleu [Purify User's Guide, 1999].

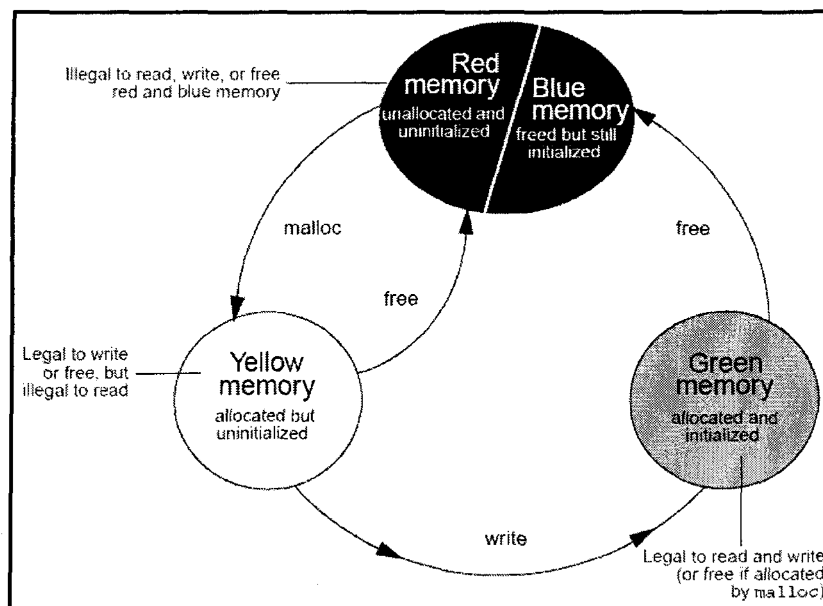


Figure 2-5 : Purify étiquette les états de mémoire par couleur [Purify User's Guide, 1999]

Chaque manipulation d'une zone mémoire donne suite à une vérification de son état de couleur. *Purify* vérifie l'état de la couleur du bloc de mémoire pour déterminer si l'opération effectuée est valide ou non. Si le programme accède à une zone mémoire d'une manière illégale, *Purify* signale une erreur.

Rouge : Au début, *Purify* étiquette les tas et les piles de mémoire en rouge. Ces zones mémoires sont non allouées et non initialisées. C'est à dire, soit qu'elles n'ont jamais été allouées ou encore qu'elles l'ont déjà été, mais libérées par la suite. Il est illégal de lire, d'écrire et de libérer une zone de mémoire coloriée en rouge (celle-ci n'est pas encore la propriété du programme, elle est non allouée et non initialisée).

Jaune : On attribue la couleur jaune à chaque zone mémoire retournée par la fonction *malloc()* ou encore l'opérateur *new()*. Une zone mémoire de couleur jaune signifie qu'elle a été allouée et qu'elle est en la possession du programme, mais elle reste toujours non initialisée. L'écriture sur une zone mémoire jaune est possible, de même que sa libération. Par contre, il est illégal de lire une zone mémoire étiquetée en jaune, car elle est encore non initialisée.

Vert : Si on effectue une opération d'écriture sur une zone mémoire jaune, *Purify* l'étiquette en vert. Cela veut dire que cette zone mémoire a été allouée et initialisée. Il est tout à fait légal de lire, d'écrire et de libérer une zone mémoire étiquetée en vert.

Bleu : Une fois qu'une zone mémoire est libérée, *Purify* l'étiquette en bleu. Une zone mémoire bleue est initialisée, mais pas pour autant accessible. Il est illégal de lire, d'écrire et de libérer une zone mémoire étiquetée en bleu.

b – Comment Purify contrôle-t-il les allocations de mémoire statiques ?

En plus de détecter les erreurs d'accès de mémoire dynamique, *Purify* détecte aussi les accès au-delà des limites des données des variables globales et statiques (les données allouées statiquement lors de la compilation). L'exemple ci-dessous, nous montre le cas ou des données sont traitées à l'aide d'un dispositif de contrôle statique :

```
int array[10];
main()
{
      array[10] = 1; //Erreur : array[10] non valide
}
```

Figure 2-6 : Erreur d'écriture hors de la limite d'un tableau

Dans l'exemple de la Figure 2-6, *Purify* rapporte une erreur d'écriture hors de la limite d'un tableau, plus précisément lors de l'affectation de la valeur 1 à *array[10]*, (ABW/*Array Bounds Write*). L'opération d'affectation est de 4 octets au-delà des limites du tableau. *Purify* insère des zones de garde rouges autour de chaque variable se trouvant dans le secteur des données statiques du programme. On se réfère à ces zones de garde comme des « *zones-rouges* » (*Red-Zones*). Si une tentative de lecture ou d'écriture dans une de ces *zones-rouges* est détectée, *Purify* rapporte une erreur de borne d'un tableau (*ABR /Array Bounds Read* ou *ABW /Array Bounds Write*). Ces erreurs vont être considérées dans la Section 4.4.3.2 du Chapitre 4.

c – Notes et limitations de Purify par rapport aux accès des zones mémoire

L'accès légal à une zone mémoire inappropriée constitue une limite importante pour *Purify*. Dû à une bonne flexibilité de la manipulation des pointeurs au sein des programmes écrits en C/C++, un pointeur pourrait accidentellement accéder à un bloc de mémoire légalement allouée, mais qui se trouve au-delà des limites du bloc de mémoire visé. Dans ce cas, *Purify* ne signale aucun accès illégal vu que cette zone mémoire est correctement allouée et initialisée [Purify Overview, 1999] [Purify User's Guide, 1999].

Pour une meilleure compréhension de la problématique, nous allons illustrer un simple exemple d'un *accès légal à une zone mémoire inappropriée*, et ensuite donner notre solution dans le Chapitre 4 :

Supposons que lors de chaque allocation dynamique pour une donnée quelconque, nous effectuons deux allocations supplémentaires : i) un espace mémoire de la taille d'une structure *InfoPointeur* afin de contenir certaines informations concernant l'allocation et i) un espace mémoire égal à 16 octets pour d'éventuels tests *d'overflow*. Nous obtenons donc trois espaces mémoire adjacents. La Figure 2-7 illustre l'adjacence des espaces mémoire alloués.

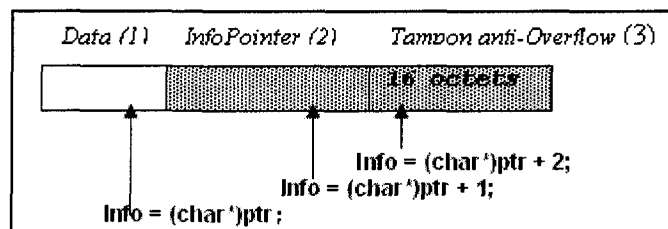


Figure 2-7 : Disposition des espaces mémoire

La Figure 2-8 nous montre plus de détails sur les instructions d'allocation :

```
typedef struct
{
      size_t taille;    //attribut taille
}InfoMPointeur;

void* Malloc (size-t size,...)
{
      ...
      void* ptr;

      //allocation des espaces mémoire supplémentaires
      ptr=malloc(size + sizeof(InfoPointer) + 16);
      ...
      return ptr;
}
```

Figure 2-8 : Structure InfoPointeur et allocation de données avec deux espaces supplémentaires

Une fois l'allocation effectuée, nous allons afficher l'attribut *taille* de la structure *InfoPointeur*. On devra pointer sur l'espace mémoire réservé à *InfoPonteur* (le deuxième espace mémoire) pour ensuite accéder à l'attribut *taille*. Dans la ligne 6 de la Figure 2.9, une erreur de frape s'est produite : une sommation de 2 a été effectuée au lieu de 1 et le pointeur *Info* se trouve pointé sur le tampon *overflow* au lieu de la structure *InfoPointer*. La Figure 2.7 montre plus de détails sur le principe de sommation et de pointage pour chacun des 3 espaces mémoire.

```
      void AfficherTailleAllocation ()
      {
            InfoPonter *Info;
            void* ptr = Malloc(...);
            ...
            //Pointer Info sur la zone mémoire à lire

L6          Info = (char*)ptr + 2;

            // Erreur : sommation de 2 au lieu de 1 et Info pointe sur le
            //tampon anti-overflow au lieu de InfoPointeur

L10         printf("Size data is : %lu", Info->taille);
            //Impossible d'accèder à la zone Info->taille
            ...
      }
```

Figure 2-9 : Accès légal à une zone mémoire inappropriée

Lors de l'affichage de l'attribut *Info->taille* (ligne 10 de la Figure 2.9), une erreur se produit. Le pointeur *Info* ne pointe pas sur la bonne adresse et ne peut atteindre l'attribut *taille*. *Purify* est incapable de détecter une telle erreur vu que l'adresse mémoire '*(char*)ptr +2*' est correctement allouée et initialisée, donc pour *Purify* cet accès est totalement légal. *Purify* contrôle les accès aux mémoires et les blocs de mémoire accédés, non pas l'arithmétique des pointeurs.

2.2.2.2. Les erreurs de fuite de mémoire (Memory Leaks Errors)

a – Comment Purify détecte-t-il les fuites de mémoire ?

Une fois qu'un programme finit son exécution, *Purify* fournit un résumé de ses fuites de mémoire (*MLS/Memory Leaked Summary*). Un *résumé de fuites de mémoire* indique la quantité de la mémoire inaccessible durant l'exécution du programme ainsi que l'emplacement de chacune d'entre elles.

Un *Garbage Collector* est formé de deux composantes : un détecteur de miettes (*Garbage detector*) et un récupérateur de miettes (*Garbage Reclaimer*). Pour atteindre certains objectifs d'un *Garbage Collector*, comme par exemple éliminer les fuites de mémoire sans pour autant courir des risques ou élever les coûts au niveau du temps d'exécution, *Purify* a apporté de nouvelles orientations et d'importants changements. En effet, à la place de fournir automatiquement un *Garbage Collector*, *Purify* fournit à la place un *Garbage Detector* accessible par un simple appel, et capable d'identifier les fuites de mémoire. Un *Garbage Detector* est un sous-système qui aide le développeur à trouver et ensuite éliminer les fuites de mémoire durant la phase de développement. La simple utilisation d'un *Garbage Detector* permet aux programmeurs de profiter de la technologie qu'offre un *Garbage Collector* sans pour autant subir les coûts qui lui sont associés durant le temps d'exécution.

L'utilisation de la technologie *Garbage Collector* lors de la phase de développement d'un programme nous permet de contourner les problèmes fondamentaux qu'un *Garbage Collector* dédié pour les langages C/C++ est supposé rencontrer. Par exemple, pouvoir distinguer les objets considérés comme des miettes (*Garbage*) de ceux qui ne le sont pas. En effet, le *Garbage Detector* que *Purify* utilise sépare le tas de miettes en trois catégories :

1. les zones mémoire dont on est presque sûre qu'elles sont des miettes, aucune référence de pointeur par le programme

2. les zones mémoire qui sont potentiellement des miettes, pas de référence de pointeur vers leur début
3. les zones mémoire qui ne sont probablement pas des miettes, ils ont des références de pointeur en leur début

Chaque bloc de mémoire est identifié par son appel de fonction d'allocation, et le développeur juge du fait s'il est nécessaire ou pas de le libérer. Durant le processus de correction des fuites de mémoire, le développeur peut libérer un bloc de mémoire prématurément, dans ce cas *Purify* signale le fait qu'une éventuelle et inadéquate libération de mémoire s'est produite.

b – Notes et limitations

Purify trouve seulement les fuites de mémoire qui ont été allouées à l'aide de la fonction *malloc()* et lui est impossible d'interagir avec des programmes utilisant autre fonction que cette dernière. Par défaut, *Purify* ne peut pas trouver les fuites dans un bloc de mémoire manipulé par des routines de gestion de mémoire que l'utilisateur à créer par lui-même. Par exemple, si un programmeur alloue un grand bloc de mémoire et le partage par la suite en plusieurs petits blocs de mémoire, ce dernier pourrait bien gérer les allocations et les libérations par lui-même, par contre *Purify* sera dans l'incapacité de détecter les fuites de mémoire dans toutes ces subdivisions.

2.2.2.3. Purify expérimental

Dans cette section, nous allons approcher la version Windows de *Purify* et voir comment procéder à son installation ainsi qu'à son utilisation. Aussi, nous avons décidé de capturer des données qualitatives associées au nombre de zones mémoire défectueuses trouvées dans une application test nommée *"DebugTest"*. Cette dernière manipulation nous assurera une installation correcte de *Purify* ainsi que son utilisation conforme.

2.2.2.4. Résultats

Ci-dessous se trouvent les résultats de l'exécution de notre plan expérimental :

a – Facilitée d'installation :

Une simple recherche sur *Google* nous a menés directement au site officiel d'IBM ainsi qu'un lien de téléchargement d'une version Windows de *Purify*. Sur un simple clic, la version 7.0.0 de *Purify* est téléchargée et ensuite installée. L'opération n'a demandé que quelques minutes de temps [Purify User's Guide, 1999] [Purify Overview, 1999].

b – Facilitée d'utilisation :

Une fois l'outil installé, nous avons trouvé son interface assez intuitive. La vérification des erreurs de mémoire d'un programme était très simple : *Purify* affiche sur sa fenêtre principale les erreurs rencontrées au fur et à mesure qu'il exécute le programme client. Il est important de noter que *Purify* n'exigeait pas de modifications du code source à analyser afin d'aboutir à un résultat, l'analyse était tout de suite disponible.
Il était relativement facile de comprendre les résultats. La Figure 2-10 nous procure plus d'explications. Un point pouvant être incompréhensible à première vue est que les erreurs détectées sont codées en des abréviations de 3 lettres, tel que ABR, IPW et UMR. Cependant, les abréviations sont toujours expliquées et la classe contenant l'erreur est fournie dans un fichier d'aide.

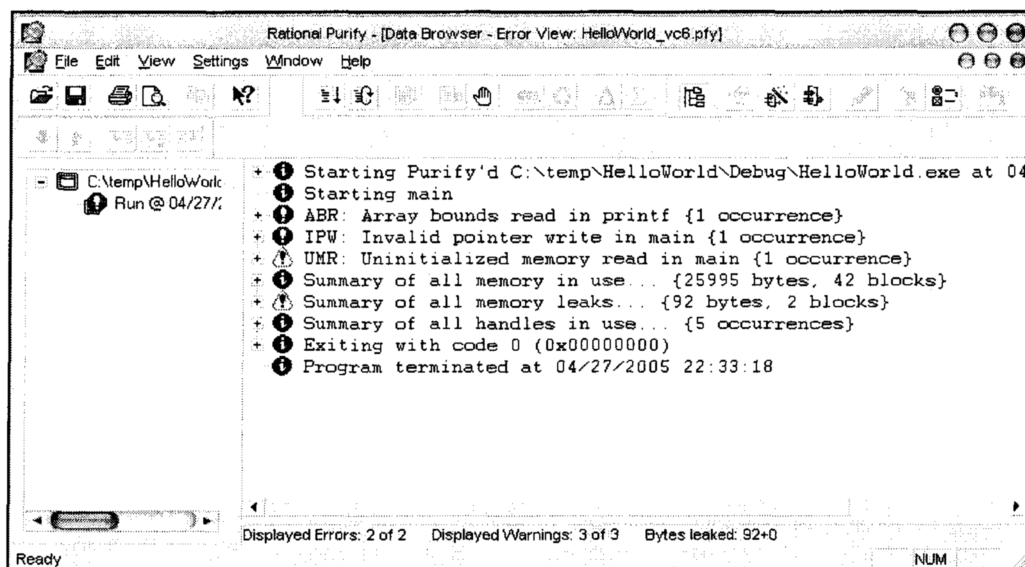


Figure 2-10 : Capture d'écran du résumé des erreurs de *DebugTest* [IBM Software, 2003]

Le résumé d'erreurs peut également être rallongé afin d'indiquer plus de détails au sujet de l'emplacement d'une erreur, voir la Figure 2-11. Si le code source est disponible, le chemin de pile est augmenté et l'endroit exact de la violation de mémoire peut être révélé.

Le principal problème rencontré durant l'utilisation de *Purify* est le bogue engendré lors de son intégration avec l'IDE utilisé tout le long de notre travail de recherche : *Microsoft Visual Studio 2005*. Une fenêtre a été affichée signalant l'erreur suivante :

Microsoft Development Environment cannot shut down because a modal dialog is active.Close the active dialog and try again.

Aucune fenêtre de dialogue n'était affichée et le seul recours était d'arrêter le processus à l'aide du gestionnaire de tâches de Windows.

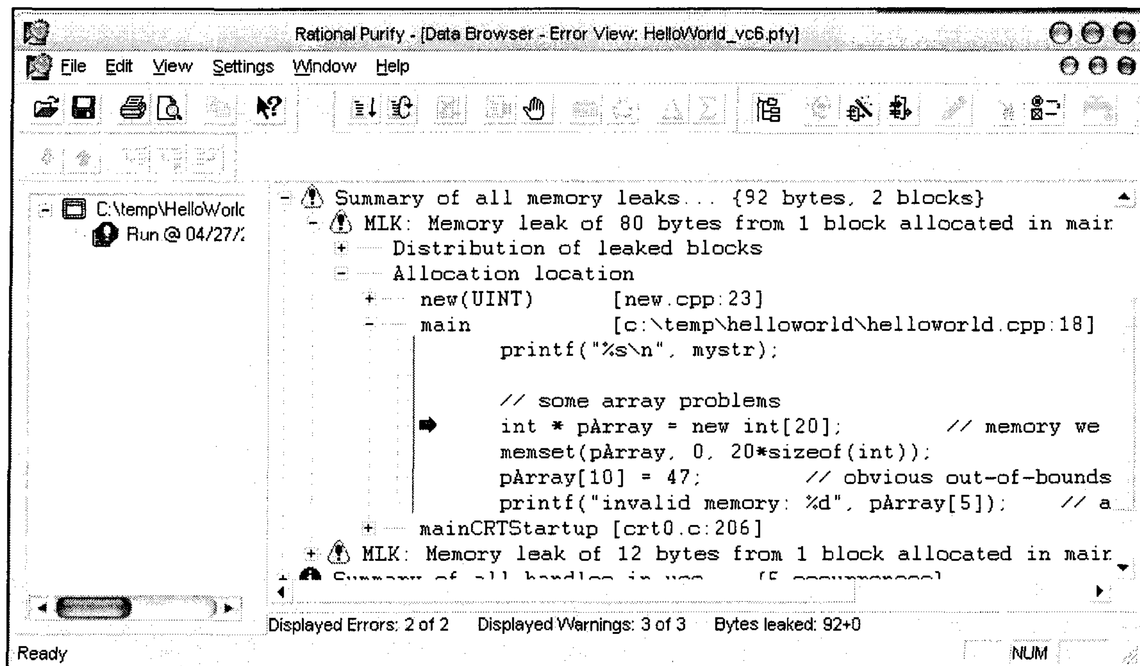


Figure 2-11 : Capture d'écran pour les détails des erreurs de *DebugTest* [IBM Software, 2003]

c – Les erreurs détectées dans *DebugTest* :

Afin de valider notre compréhension de la manière avec laquelle *Purify* est mis en place et comment interpréter ses résultats, nous avons injecté plusieurs erreurs de gestion de mémoire dans l'application test *" DebugTest»* [Hagen et Zhang, 2002].

Purify a été capable de détecter les six erreurs injectées. On peut se référer à la Figure 2-10 et à la Figure 2-11 pour les captures d'écran des résultats :

- 2 fuites de mémoire ;
- 1 erreur de lecture au-delà des limites d'un tableau ;
- 1 accès invalide à un pointer ;
- 1 erreur de lecture d'une zone de mémoire non initialisée ;
- 1 erreur de dépassement de tampon.

2.2.3. Memcheck

Memcheck est un vérificateur de mémoire à grande capacité. Il permet la vérification de toutes les instructions d'écriture et de lecture, et l'interception des appels de fonction et d'opérateur comme *malloc*, *new*, *free* et *delete*. Par conséquent, *Memcheck* est dans la possibilité de détecter les problèmes suivants :

- l'utilisation d'une zone mémoire non initialisée ;
- lecture/écriture d'une zone mémoire après qu'elle a été libérée ;
- lecture/écriture au-delà d'un bloc de mémoire retourné par la fonction *malloc ;*
- lecture/écriture d'une zone inadéquate dans une pile ;
- fuite de mémoire, lorsqu'un pointeur sur un bloc de mémoire retourné par la fonction *malloc* est perdu à jamais ;
- mauvaise utilisation de *malloc/new/ new []* et *free/delete/delete [].*

Memcheck utilise une instrumentation binaire et dynamique afin d'analyser un programme client, et ce, durant son temps d'exécution. L'exécution du code instrumenté est entre-calée avec l'exécution normale du programme sans pour autant déranger son comportement normal. Ce genre d'instrumentation procure un travail en extra dans le sens où elle aide à détecter des erreurs de gestion de mémoire. Le fait que *Memcheck* instrumente et analyse le code machine et non pas le code source ou encore le code objet lui permet d'avoir :

- une large applicabilité : fonctionne avec des programmes écrits avec différents langages de programmation ;
- couverture totale : toutes les parties du programme client sont exécutables sous *Memcheck*, incluant les bibliothèques liées dynamiquement ;
- facilité d'utilisation : *Memcheck* ne demande pas que le programme client soit recompilé ou redirigé.

Memcheck est réputé pour sa facilité d'utilisation. Considérons un programme *badprog.cpp*. Pour procéder à l'analyse du programme compilé *badprog*, l'utilisateur aurait à taper tout simplement : *valgrind --tool=memcheck badprog*

Un programme analysé sous *Memcheck* s'exécute à peu près 20 à 30 fois plus lentement que d'habitude. Ce ralentissement est dû d'une part aux vérifications effectuées par *Memcheck* et d'une autre part à l'exécution du framework qui l'abrite (Valgrind). *Memcheck* fut le premier outil à traquer les erreurs de gestion de mémoire au niveau du *bit*. Des outils semblables n'ont pu atteindre que l'octet comme niveau de granularité (Purify). *Memcheck* est capable de bien analyser ou de manipuler correctement un code traitant des définitions partielles d'octet, par exemple le cas où l'on utilise les deux idiomes communs aux langages C et C++ : les *tableaux de bits* ou encore le type *bitfield* dans les structures de données. Dans de telles circonstances et avec un outil d'analyse dont le niveau de granularité ne dépasse pas l'octet, on ne pas garantir des résultats précis.

Dans la Figure 2-12, on montre un programme utilisant un bit non initialisé dans un tableau de bit (*bit-array*). Dans ce cas, *Memcheck* rapporte l'erreur contrairement à *Purify* qui ne la détecte pas [Nethercote et Seward, 2005].

```
void set_bit ( int* arr, int n )
{
        arr[n/32] |= (1 << (n%32));
}
        int get_bit ( int* arr, int n )
{
        return 1 & (arr[n/32] >> (n%32));
}
int main ( void )
{
        int* arr = malloc(10 * sizeof(int));
        set_bit(arr, 177);
        printf ("%d\n", get_bit(arr, 178));//Erreur : bit 178
        return 0;                       // n'est pas accessible
}
```

Figure 2-12 : Utilisation non sécurisée d'un tableau de bit

2.2.3.1. Valgrind

Memcheck fait partie d'une suite d'outils de *Valgrind*, un logiciel gratuit (GPL) qui fonctionne seulement sur la plate-forme de x86/Linux, des travaux sont actuellement en cours pour lui permettre de communiquer avec d'autres plateformes telle que AMD64/Linux. *Memcheck* est implémenté comme un plugin à *Valgrind*, qui est à son tour un *framework* pour

la création d'outils utilisant une instrumentation binaire et dynamique [Nethercote and Seward, 2003]. En effet, *Valgrind* est chargé d'effectuer les tâches considérées comme les plus difficiles en insérant l'instrumentation au niveau du code machine, et ce, lors du temps d'exécution. Les outils que le *noyau Valgrind* (*Valgrind core*) supporte comme plugin sont écrits en C. La structure de base est comme suit :

Le noyau de Valgrind + Le plugin d'outil = L'outil Valgrind

2.2.3.2. Valid-value (V) bits

Dans cette section et celle qui s'en suit, nous allons nous intéresser à ce que *Memcheck* analyse et surtout à la façon dont il le fait.

Il est plus simple de penser à *Memcheck* mettant en application une unité centrale de traitement synthétique (*CPU)* qui serait identique à une vraie unité centrale de traitement, excepté un détail crucial : chaque *bit* (littéralement parlant) des données qui seront stockées et traitées par le vrai *CPU* aurait, dans le *CPU* synthétique, un bit *Vbit* associé indiquant si le bit qu'il accompagne aurait ou pas une valeur légitime. Par exemple, si un CPU charge dans la mémoire un mot dont la taille est égale à 4 octets, il charge également les 32 V-bits correspondant à partir d'un *bitmap*. La présence des V-bits ne veut pas dire que *Memcheck* va forcément effectuer une vérification ou encore signaler une erreur. Le contrôle se produit au besoin, plus précisément, dans l'un de ces trois endroits : i) quand une valeur est utilisée pour générer une adresse mémoire, ii) quand la décision du flux de contrôle doit être précise, et iii) quand un appel système est détecté.

2.2.3.3. Valid-address (A) bits

Comme décrit ci-dessus, chaque bit dans la mémoire ou dans le CPU détient un V-bit propre à lui. De plus, chaque bit dans la mémoire, et non pas dans le CPU, est associé à une *adresse-valide* (A)-bit. Cette dernière indique si le programme est dans la possibilité ou pas d'effectuer une lecture ou une écriture légitime sur cette zone. La tâche du A-bit se limite au droit d'accès et ne donne aucune indication sur la validité des données dans cette zone, ceci est bien entendu la tâche du V-bit. Ainsi, chaque fois qu'un programme client effectue une opération de lecture ou d'écriture sur une zone mémoire, *Memcheck* vérifie les A-bit associés à l'adresse mémoire en question. Si l'un de ces A-bit indique une adresse invalide ou inadmissible, une erreur est automatiquement émise.

2.2.3.4. Notes et limitations

En plus de la pénalité de performance, *Memcheck* se trouve incapable de détecter les erreurs de limite de données allouées d'une manière statique. La Figure 2-13 présente un code qui passerait sous l'analyse *Memcheck* de *Valgrind* sans aucun incident, et ce, en dépit des erreurs indiquées :

```
int Static[5];
int func(void)
{
      int Stack[5];
      Static[5] = 0;//Erreur - Static [0] à Static [4] existe,
                     //Static[5] est hors limite
      Stack [5] = 0;//Erreur - Stack[0] à  Stack[4] existe,
                     //Stack[5]est hors limite
      return 0;
}
```

Figure 2-13 : Memcheck ne détecte pas les erreurs d'accès hors de la limite d'un tableau

2.3. Gestion de la mémoire et C++

Le langage C++ fait partie des langages de programmation les plus utilisés, il est aussi bien connu pour sa facilité d'utilisation que pour son efficacité. Sinon, le langage C++ reste un langage réputé comme étant compliqué et illisible. Cette réputation est en partie justifiée [Ellis et Stroustrup 1990]. La complexité du langage est inévitable lorsqu'un programmeur cherche à atteindre un niveau supérieur de fonctionnalités. Par contre, pour ce qui concerne la clarté et la lisibilité du programme, tout dépend de la bonne volonté du programmeur.

Le langage C++ détient des caractéristiques qui ont fait de lui un langage idéal pour certains types de projet. Il est incontournable dans le développement des grands programmes. Les optimisations des compilateurs actuels en font également un langage de prédilection pour ceux qui recherchent les performances. Enfin, le langage C++ est idéal pour ceux qui doivent atteindre une portabilité des fichiers sources de leurs programmes (non pas des fichiers exécutables) [Stroustrup, 2004].

Après des années de travail, un comité réunissant l'ANSI et l'ISO standardisa C++ en 1998 (ISO/CEI 14882:1998). Quelques années après la sortie officielle du standard, le comité traita le rapport de problèmes et publia une version corrigée du standard C++ en 2003 [Kernighan et Ritchie, 1988]. Le langage C++ est un langage de programmation permettant la

programmation sous de multiples paradigmes comme, par exemple, la programmation procédurale, la programmation orientée objet et la programmation générique. La structure d'un programme écrit en C++ est évidemment similaire à la structure d'un programme écrit en C. On y retrouve la fonction principale *main()*, des fichiers d'en-tête et des fichiers sources, et une syntaxe très proche du C. La bibliothèque standard du C++ est en grande partie un sur-ensemble des fonctions disponibles dans la bibliothèque standard du C. Elle englobe la *Standard Template Library* (STL) qui met à la disposition du programmeur des outils puissants comme les collections (conteneurs) et les littérateurs. Il est connu que le langage C++ tient sa puissance du fait qu'il est la descendance directe du langage C. D'autres affirment que cela constitue son principal handicap [Jones et Lins, 1996]. Le langage C++, et comme tout langage de programmation, gère la création et la destruction de ces entités. Vu qu'il est une extension de l'ANSI-C, il offre donc une gestion de mémoire explicite avec les *delete*, *new*, *free*, et *malloc*.

2.3.1. Gestion de la mémoire en C++

Lors de la déclaration d'un objet, il est possible de définir les méthodes *constructeur* et *destructeur*. Le *constructeur* est appelé lors de l'instanciation de l'objet, et ce, en lui allouant un espace mémoire et en initialisant les membres de cet objet. Il peut cependant avoir d'autres fonctionnalités : acquérir des ressources comme des routines du système ou ouvrir des fichiers ou des bases de données. Aussi, un *destructeur* est appelé à la fin du programme ou encore lorsque la fonction *delete* est appelée. Le *destructeur* est utilisé pour l'action de *finalisation* lorsqu'elle est nécessaire, mais aussi pour désallouer l'espace mémoire consommé par l'objet. Typiquement, l'action de finalisation consiste à libérer les routines ou à fermer un fichier.

Comme son prédécesseur, pour la gestion du cycle de vie de ses variables, le langage C++ a utilisé les fonctions *new (malloc())* et *delete (free())*. Ces deux fonctions ont aussi intégré respectivement les constructeurs et les destructeurs des objets pour pouvoir leurs allouer et leurs libérer, respectivement, de l'espace mémoire. Ainsi, la création d'un objet passe automatiquement par l'allocation d'un espace mémoire qui lui sera requis. Le cycle de vie d'un objet se termine par la destruction de cet objet, et ce, en libérant l'espace mémoire qu'il occupait au moyen de la fonction *delete*.

2.3.2. Problèmes actuels avec les compilateurs C++

Puisque le langage C++ est une surcouche du langage C, il comporte alors tous les défauts de ce dernier. Par exemple, dans la programmation C, les fuites de mémoire et les débordements sont très courants en raison de l'utilisation des fonctionnalités telles que les types *char*/char[]*, les fonctions de manipulation de chaîne comme *strcpy/strcat/strncpy/strncat*, et les fonctions de manipulation de mémoire comme *malloc/realloc/strdup* [Vasudevan, 2000].

L'utilisation de *char* * et *strcpy* crée d'affreux problèmes dus au débordement (overflow), aux accès hors limite (*fence past errors*), aux altérations des emplacements en mémoire ou aux fuites de mémoire. Les problèmes de mémoire sont extrêmement difficiles à déboguer et très longs à corriger et à éliminer. Ils diminuent la productivité des programmeurs. Certains programmeurs se basent sur différentes méthodes destinées à résoudre les défauts de la gestion de mémoire en C++ pour augmenter cette productivité. Les bogues liés à la mémoire sont très durs à éliminer, et même les programmeurs expérimentés y passent plusieurs jours, semaines ou mois pour en venir à bout. La plupart du temps, les bogues de mémoire restent "tapis" dans le code durant plusieurs mois et peuvent causer des plantages inattendus. Par exemple, l'utilisation de *char* * en C++ coûte aux États-Unis et au Japon deux milliards de dollars chaque année en temps perdu en débogage et en arrêt des programmes [Vasudevan, 2000]. Si on utilise *char* * en C++, cela est très coûteux, en particulier si le programme fait plus de 50.000 lignes de code.

Le langage C++ n'a pas la capacité de gérer sa mémoire d'une façon autonome. Plusieurs tentatives d'incorporation d'outils pour la gestion implicite des entités et objets au sein du langage C++ ont connu l'échec à cause de circonstances compliquées où évolue le langage [Stroustrup, 2004]. Les chercheurs ont tous simplement opté pour l'utilisation d'outils externes comme le ramasse miette et le débogueur de mémoire.

2.4. Conclusion

Dans ce chapitre, nous avons survolé les principes de cycle de vie des objets, pour en arriver ensuite à la gestion de mémoire que ce soit d'une manière explicite ou encore implicite. En effet, les méthodes de gestion implicite de la mémoire, par exemple les *Garbage Collector*, se voient parmi les plus prometteuses. Mais nous avons vu aussi que sans l'aide

d'un débogueur de mémoire on ne peut garantir une gestion de mémoire complète d'une application C++. Les erreurs d'accès, de fuites de mémoire et de débordement peuvent être fatales pour un système ou encore nuire à son comportement. Leurs présences et leurs suppressions nécessitent beaucoup d'heures de travail sans pour autant garantir un bon résultat, elles sont donc très couteuses en temps et en argent. Dans la Section 2.2, nous avons vu que les débogueurs de mémoire ont évolué sous plusieurs formes dans les langages à gestion implicite de la mémoire, mais qu'ils ont été aussi dédiés pour des langages à gestion explicite de la mémoire comme le langage C++. Ce dernier est un langage reconnu pour sa puissance et sa robustesse et l'ajout d'un débogueur de mémoire ne devrait en aucun cas l'en diminuer. Différents débogueurs de mémoire spécifiques pour C++ ont été élaborés, mais, malheureusement, ne garantissent pas une gestion de mémoire sans erreurs. En effet, les points suivants sont toujours soulevés (ils ne sont pas résolus par les débuggeurs existants comme *Purify et Valgrind*) :

- accès incorrecte à une zone de mémoire légale (manipulation des pointeurs) ;
- ralentissement de l'application cliente ;
- détection des erreurs hors de limite dans le cas où les données sont allouées d'une manière statique ou dans une pile.

Les aspects énumérés à la fin du Chapitre 1 forment un cadre de solution pour ces points. Ces derniers seront résolus dans le Chapitre 4 en se basant sur la programmation par aspect qui nous permettra de développer un outil robuste de gestion de mémoire. La programmation par aspect est expliquée en détail dans le chapitre suivant.

Chapitre 3 : Préoccupations et aspects

Dans le chapitre précédent de ce mémoire, nous avons présenté différents axes de recherche en matière de gestion et de débogage de mémoire, et avons choisi de mettre au point notre propre outil. Nous présenterons dans ce Chapitre, l'approche qu'on utilisera pour l'implémentation de ce dernier.

3.1. Introduction

Les méthodes d'adaptation de logiciel sont employées dans la technologie de la programmation pour deux raisons : i) pour contrôler la complexité des systèmes que nous essayons d'établir, et ii) pour retarder, dans la mesure du possible, les différentes liaisons entités/préoccupations des systèmes en collaboration [Mcheick, 2006]. Ainsi, ils peuvent être développés, testés et maintenus séparément, pour ensuite atteindre un plus grand degré de réutilisation et de facilité de maintenance. Au fur et à mesure du temps, un certain nombre d'approches ont vu leur apparition dans le but de rendre modulaires des logiciels, compris la programmation orientée objet, la méthode OORAM [Reenskaugh et al., 1995], la programmation orientée sujet [Harrison et Ossher, 1993], la programmation orientée aspect qui sert de toile de fond pour ce travail [Kiczales et al., 1997], la programmation orientée vue [Mili et al., 1999], etc. La croissance des expériences dans l'utilisation des approches d'adaptation des logiciels a poussé les praticiens à se poser des questions aussi bien fondamentales (c'est quoi la relation aspect/préoccupation, quels types de préoccupation peuvent être séparables/composables) que techniques (comment utiliser une technique particulière pour résoudre un problème particulier). Ce chapitre décrit une approche existante qui pourrait répondre aux problèmes d'évolution et d'adaptation logicielle tels ceux cités dans le Chapitre 2. Ce chapitre s'articule sur les trois principaux points suivants :

- un outil conceptuel développé par les chercheurs et praticiens sera présenté : la séparation des préoccupations (*separation of concerns*) ;
- la programmation par aspect, ses motivations et son fonctionnement seront aussi présentés. AspectC++™, un outil implémentant la programmation par aspect, sera détaillé par la suite ;

- enfin, nous mettrons l'accent sur la problématique de notre recherche et citerons les objectifs à atteindre.

3.2. Séparation des préoccupations

Les systèmes informatiques répondent à un ensemble d'exigences du client, plus ou moins bien définies. L'implémentation de certaines de ces exigences dans les langages de programmation actuels n'est pas toujours bien circonscrite. On se retrouve donc avec des problématiques dont les implémentations se retrouvent un peut partout dans les modules du système. C'est ce qu'on appelle des préoccupations qui se recoupent (*crosscutting concerns*) [Baltus, 2000]. Pour faire face à ce genre de problématiques, chercheurs et praticiens du génie logiciel ont toujours utilisé un outil conceptuel qui est la séparation des préoccupations (*separation of concerns*).

La séparation des préoccupations est un concept présent depuis de nombreuses années dans l'ingénierie des logiciels [Mcheick, 2006]. Les différentes préoccupations des concepteurs apparaissent comme les motivations premières pour organiser et décomposer une application en un ensemble d'éléments compréhensibles et facilement manipulables. La séparation en préoccupations apparait dans les différentes étapes du cycle de vie du logiciel. Il s'agit de préoccupations d'ordre fonctionnel (séparations des fonctions de l'application), d'ordre technique (séparation des propriétés du logiciel système), ou encore liées aux rôles des acteurs du processus logiciel (séparation des actions de manipulation du logiciel). Par ces préoccupations, le logiciel n'est plus abordé dans sa globalité, mais en parties. Cette approche réduit la complexité de conception, de réalisation, mais aussi de maintenance d'un logiciel et en améliore la compréhension, la réutilisation et l'évolution [Sioud, 2006].

En ce qui nous concerne, nous devons essentiellement pouvoir distinguer les préoccupations fonctionnelles des préoccupations non fonctionnelles (techniques/ architecturales). Les préoccupations fonctionnelles décrivent le comportement du système et définissent les fonctions ou les services que le système doit remplir. Ce sont des préoccupations relatives au domaine d'application. Or, les préoccupations non fonctionnelles expriment les qualités ou contraintes imposées sur la manière de satisfaire les exigences fonctionnelles. Il s'agit principalement des préoccupations de performance, sécurité, sûreté, convivialité, concurrence, etc.

La Figure 3-1 montre un exemple d'une application composée de trois modules fonctionnels dans lesquelles on a dû y insérer l'implémentation d'exigences de performance, de journalisation (enregistrement d'informations dans un fichier journal) et de synchronisation.

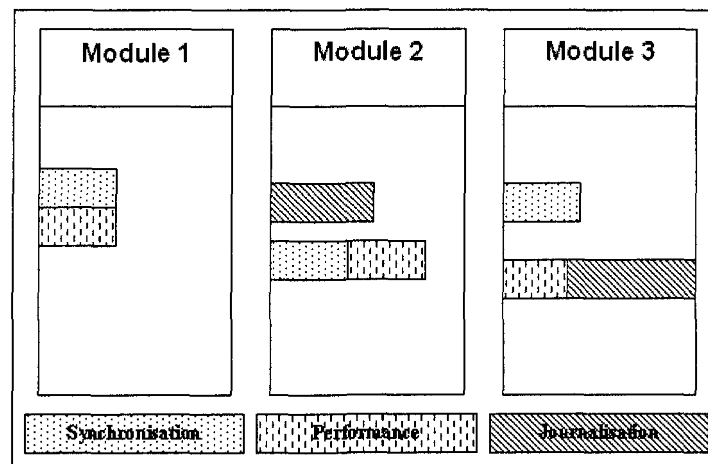


Figure 3-1 : Les exigences non fonctionnelles traversent la modularisation fonctionnelle du système

Les exigences non fonctionnelles qui traversent la modularisation fonctionnelle du système posent un problème capital qu'on peut identifier facilement sur la Figure 3-1. En effet, on peut remarquer la *dispersion ou l'enchevêtrement du code* traitant un aspect du système à travers différents modules (par exemple la synchronisation). Considérons un langage orienté objet, l'unité fonctionnelle est le package et si nous raffinons encore la découpe, cette unité est la classe. Il n'est pas rare de voir des méthodes traitant d'une exigence non fonctionnelle se disperser dans l'implémentation des différentes classes composant le cœur fonctionnel du système. Par exemple, dans un système de gestion de base de données, il se peut que la performance, la journalisation et la synchronisation concernent toutes les classes accédant à la base de données. On voit donc que ces aspects seront implémentés dans plusieurs modules sans être bien circonscrits. Il s'agit très souvent de portions de code très similaires à ajouter un peu partout dans les modules concernés. Ce genre d'obstacle entraine des conséquences négatives sur le développement de logiciels [Gati , 2004] :

A. *Traçage difficile*: les différentes préoccupations d'un logiciel deviennent difficilement identifiables dans l'implémentation. Il en résulte une correspondance assez obscure entre les exigences et leurs implémentations.

B. *Diminution de la productivité*: la prise en considération de plusieurs exigences au sein d'un même module empêche le programmeur de se focaliser uniquement sur son but premier. Le danger d'accorder trop ou pas assez d'importance aux aspects accessoires d'un module en découle directement.

C. *Diminution de la réutilisation du code*: dans les conditions actuelles, un module implémente de multiples exigences. D'autres systèmes nécessitant des fonctionnalités similaires pourraient ne pas pouvoir réutiliser le module tel quel, entraînant de nouveau une diminution de la productivité à moyen terme.

D. *Diminution de la qualité du code*: les programmeurs ne peuvent pas se concentrer sur toutes les contraintes à la fois. L'implémentation disparate de certaines préoccupations peut entraîner des effets de bords non désirés, i.e. des bugs.

E. *Maintenance et évolutivité du code difficile*: lorsque l'on veut faire évoluer le système, on doit modifier de nombreux modules. Modifier chaque sous-système pour répercuter les modifications souhaitées, peut conduire à des incohérences.

Vu que les méthodes de programmation traditionnelles sont incapables de fournir une modularisation car il est difficile d'en limiter la portée dans un module ou artéfact bien circonscrit, de nouveaux paradigmes communément regroupés sous le nom de développement de logiciel orienté aspect (ou AOSD pour Aspect-Oriented Software Development) apportent des solutions simples et élégantes à ces problèmes [Baltus, 2000]. En effet, la programmation par aspect, qui sera notre approche cible, [Kicsales et al., 1997] permet de modulariser des exigences non fonctionnelles telles que la sécurité, la persistance, la journalisation, le traçage, etc. La programmation par sujet [Harisson et Ossher, 1993] et ses descendantes [Tarr t al., 1999] ou la programmation par vues [Mili et al., 1999], quant à elles, permettent de modulariser des préoccupations touchant aux mêmes entités.

Dans la section suivante, nous allons introduire la notion de programmation par aspect autant que technique de séparation de préoccupations, montrer ses avantages et motivations, puis présenter l'outil AspectC++™ avec lequel nous allons pouvoir mettre en pratique notre débogueur de mémoire *AspectC++Debogger* qu'on présentera en détail dans le Chapitre 4.

3.3. Programmation par aspect (POA)

Pour les problèmes de recoupement et les conséquences négatives qui en résultent, la programmation par aspect se trouve être la solution élégante et la plus facile à implémenter. Celle-ci a été introduite par des chercheurs du XEROX PARC en 1997 [Kicsales *et al.*, 1997]. Cette nouvelle approche peut être considérée comme l'approche clé pour l'ingénierie des applications complexes, telles que les applications distribuées, les Entreprises Ressources Planning(ERP) et bien d'autres.

3.3.1. Descriptions

L'POA est une méthode de programmation dont le but est de séparer l'implémentation de toutes les exigences fonctionnelles ou non d'un logiciel. Le principe est donc de coder chaque problématique d'une façon indépendante et ensuite de définir leurs règles d'intégration afin de pouvoir les combiner en vue de former le système final. Contrairement à l'approche orientée objet, cette nouvelle technique permet aux programmeurs d'encapsuler des comportements qui affectaient de multiples classes dans des modules réutilisables. En d'autres termes, l'POA permet d'encapsuler dans un module les préoccupations qui se recoupent avec d'autres. Néanmoins, il est important de souligner que la programmation par aspect utilise la technologie objet dans un cadre qui en conserve les bénéfices tout en palliant à ses limitations.

Pour mieux comprendre ce principe, nous pouvons reprendre la description originale faite par George Kiczales et ses collègues [Kicsales et al., 1997]. Une préoccupation qui doit être implémentée est soit :

- *Une composante fonctionnelle*, si la préoccupation peut clairement être encapsulée dans une procédure généralisée telle qu'un objet, une méthode, une procédure ou une fonction. Par définition, les composantes sont donc des unités fonctionnelles d'un système. Ex. : les services imprimantes, accès aux bases de données, etc.
- *Un aspect*, si la préoccupation ne peut pas être clairement encapsulée dans une composante. Les aspects correspondent souvent aux exigences non fonctionnelles d'un système traversant plusieurs de ses composantes fonctionnelles et facilitant la gestion de leurs propriétés. Ex. : l'exception, la sécurité, la synchronisation, la performance, etc.

En utilisant ces termes, on entrevoit le but de l'POA qui se résume à aider le programmeur à séparer clairement les aspects et les composantes les uns des autres via des mécanismes qui permettent de les abstraire et de les composer pour obtenir un système final. On sent tout de suite la différence avec les langages orientés objet qui ne permettent que de séparer les composantes, rendant impossible l'abstraction claire des aspects. Avec l'POA, chaque problématique est implémentée de façon complètement indépendante et sans pour autant se soucier de l'existence des autres contraintes.

3.3.2. Fonctionnement et Motivations

Après avoir décrit la programmation orientée aspect et ses buts, nous allons nous intéresser à ce que cette nouvelle approche implique dans le processus de développement d'un logiciel. Comme le montre la Figure 3-2, l'implémentation d'une application orientée aspect peut se dérouler en 3 étapes :

1. La décomposition des éléments du système. Il s'agit d'identifier tous les composantes et aspects. On sépare toutes les préoccupations, qu'elles soient fonctionnelles ou non.

2. L'implémentation de chaque préoccupation. Chaque problématique sera codée séparément dans une composante ou un aspect. Dans un aspect, le programmeur définit aussi les règles d'intégration de l'aspect avec les composantes concernées.

3. L'intégration du système. Tout langage orienté aspect offre un mécanisme d'intégration appelé le *compilateur d'aspect* ou *weaver*. Le compilateur, à l'image d'un métier à tisser, va donc composer le système final en se basant sur des règles et des modules qui lui ont été attribués

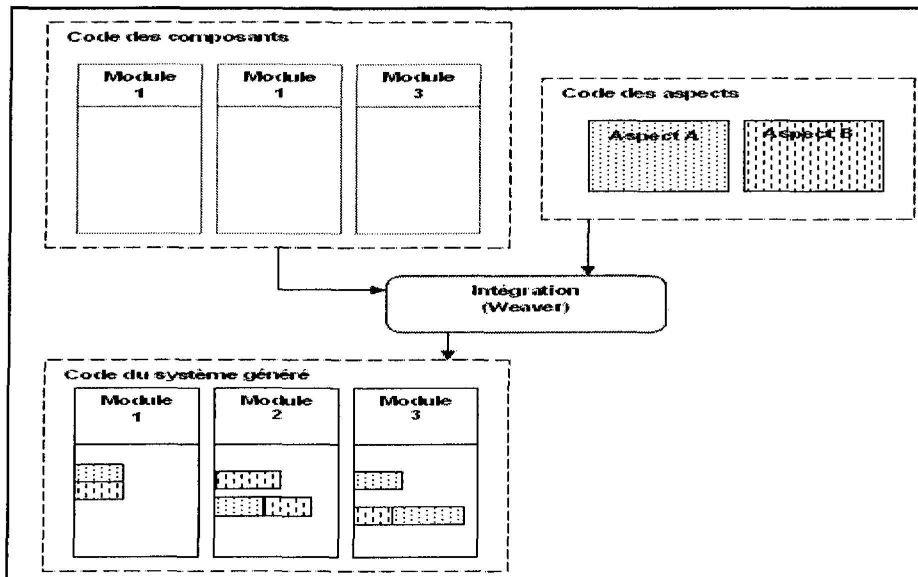


Figure 3-2 : Intégration du code des composantes et des aspects pour former le système final

L'implémentation séparée des préoccupations permet d'éviter la duplication du code. Lorsque chaque module s'adresse toujours à une seule problématique, il est beaucoup plus facile de comprendre le code et de procéder à des changements. Les systèmes orientés aspect sont beaucoup plus évolutifs pour deux autres raisons. Premièrement, vu que les modules ne sont pas au courant des problématiques qui se recoupent, il est plus facile d'ajouter de nouvelles fonctionnalités au système. Deuxièmement, si on veut plus tard rajouter une exigence non fonctionnelle comme la synchronisation, il suffit de créer un nouvel aspect qui s'en occupe, sans toucher au code des composantes existantes [Tarr et al., 1999].

Aussi avec l'POA, on trouve moins de difficulté sur le plan de la conception qu'avec une approche plus classique. Avec les langages traditionnels, le concepteur doit prévoir à l'avance les éléments susceptibles de changer dans l'architecture de son système. Il en résulte ce que certains appellent la *paralysie par l'analyse*, i.e. le fait qu'un développeur ne sache jamais quand s'arrêter dans la spécification du design de son système. La programmation orientée aspect diminue fortement ce risque puisqu'il est toujours possible de rajouter des aspects qui n'avaient pas nécessairement été prévus à l'avance. Et comme dernier avantage, en se basant sur une approche orientée aspect et en cas de bogue, un programmeur a tendance à trouver plus facilement la source du problème au niveau du code. Les développeurs passent moins de temps à essayer de comprendre la sémantique des instructions et gagnent donc un temps considérable.

3.3.3. Intégration des aspects

Pour construire une application à partir de différents aspects définis séparément les uns des autres, une étape d'intégration est forcement nécessaire. Cette étape est appelée *composition des aspects*, ou encore *tissage des aspects*. C'est là où le *compilateur d'aspect* ou *weaver* permet d'appliquer les aspects aux composantes et ceci au moyen d'un langage d'aspect procédural simple fournissant des opérations sur des méthodes et des classes développées dans les composantes fonctionnelles.

Techniquement, un aspect est une entité regroupant un ensemble de conseil (advice), de point de jonction (joinpoint), ou encore des coupes transverses (pointcut). L'intégration des composantes d'une application avec les aspects non fonctionnels se traduit par l'établissement de jonctions. Ces jonctions se situent au niveau d'un ensemble de points de jonction du flot d'exécution du noyau [Kicsales et al., 1997]. Cet ensemble de points porte le nom de *pointcut* et tout point de ce flot d'exécution constitue un point de jointure potentiel susceptible d'être utilisé pour l'intégration. Ainsi, une invocation de méthode, une boucle ou une affectation pourrait être des points de jointure potentiels. Enfin viens le rôle du *conseil* ou *greffon* qui est un fragment de code à insérer dans les points de jonction. Le *conseil* implante une préoccupation transverse qui regroupe un ou plusieurs domaines. Notons que la variété des points de jointure potentiels qui peuvent être désignés dans la configuration dépend fortement de la mise en œuvre de la programmation par aspect. Ces concepts sont implantés dans les langages de programmation tels que C++ (AspectC++™ [Spinczyk O. et al., 2002] , TACO [Vaysse, 2005]), Java (AspectJ [AspectJ, 2001] et [Lesiecki, 2002]), AspectWerkz [AspectWerkz, 2005] et JAC [JAC, 2006].
Selon la technique de tissage, les tisseurs peuvent être classés en deux catégories :

- *un tissage statique* par instrumentation du code source ou du pseudo-code machine intermédiaire (AspectJ [AspectJ, 2001] et AspectWerkz [AspectWerkz, 2005]);
- *un tissage dynamique* lors de l'exécution du logiciel (AspectC++™ [Spinczyk O. *et al.*, 2002], TACO [Vaysse, 2005] et JAC [JAC, 2006]).

Dans ce travail de recherche, nous utilisons seulement le langage C++ et bien évidemment le compilateur d'aspect AspectC++™. Tous nos exemples et applications sont écrits avec ce langage que nous approcherons de façon plus détaillée dans la section suivante.

3.3.4. Programmation par aspect avec AspectC++

AspectC++™ est une extension du langage de programmation C/C++. Il a pour objectif de traduire un code source d'AspectC++™ en un code source C++. Le projet AspectC++™ a débuté avec un prototype issu de recherches en 2001 qui a gagné en maturité au cours des années. De nos jours, le langage AspectC++™ est appliqué avec succès sur un certain nombre de projets à une échelle internationale aussi bien dans le domaine informatique qu'académique.

Afin de présenter AspectC++™, nous nous baserons sur un exemple simple d'aspect intitulé *tracing*. Cet exemple illustrera les éléments de base du langage comme les aspects, les coupes transverses, ainsi que les méthodes d'aspect. Notre exemple a pour but de modulariser l'implémentation des opérations de données de sortie qui sont utilisées afin de pister le flux de contrôle du programme. À chaque démarrage de l'exécution d'une fonction, l'aspect exposé ci-dessous édite son nom. On s'inspirera de l'exemple *tracing* afin de développer la Section 4.2.1 du Chapitre 4 qui présentera AspectC++™ dans le cadre pratique de notre travail de recherche.

```
#include <cstdio>
// Exemple de dépistage du flux de contrôles
aspect Tracing
{
      // édite le nom de la fonction avant
      // le début de son exécution
      advice execution ("% ...::%(...)") : before ()
      {std::printf ("in %s\n", JoinPoint::signature ());}
};
```

Figure 3-3 : Aspect Tracing s'exécutant avant le démarrage de l'exécution de n'importe quelles fonctions du programme principale

Même en n'ayant pas une grande maîtrise des éléments syntaxiques exposés ci-dessus, l'POA offre d'énormes avantages sensés être évidents après une lecture diagonale de l'exemple. Sans le recours à ce simple aspect, qui ne prend que quelques lignes de code, le développeur serait obligé d'étendre l'ensemble des fonctions du programme au moyen d'une déclaration supplémentaire *printf* afin d'obtenir le même résultat. Dans un projet important, un guide de style devrait indiquer cette technique sous forme d'exigences et l'ensemble des programmeurs devrait lire et observer cette règle générale. Ainsi, la solution proposée par l'POA permet de gagner un temps précieux, des efforts d'organisation, et garantit l'absence d'oubli de fonction. De la même façon, le code affecté par l'aspect est entièrement découplé

du code de dépistage, soit la déclaration *printf.* Il n'est même pas nécessaire d'inclure *<cstdio>*, dans la mesure où tout ce procédé est effectué à part par l'aspect.

L'exemple de notre aspect exposé ci-dessus, englobe la plupart des éléments du langage AspectC++™ chargés d'accomplir tous les avantages qu'un aspect est supposé offrir. Un aspect du langage AspectC++™, principalement conçu comme un module pour implémenter une préoccupation transverse, s'inspire fortement de la syntaxe d'une classe sous C++. Toutefois, en plus des fonctions membres et des éléments de données, un aspect peut définir une méthode d'aspect. Après le mot clé *advice*, une expression de coupe transverse définit l'emplacement où la méthode d'aspect est censée affecter le programme (autrement dit les points de jonction), alors que la partie qui suit définit la façon dont le programme est censé être affecté sur ces points. Il s'agit de la règle générale pour l'ensemble des différents types de méthode d'aspect sous AspectC++™. Dans le Chapitre 4, des exemples du même domaine et concernant notre travail de recherche seront détaillés.

L'expression de la coupe transverse donnée dans l'exemple de la Figure 3-3 est `execution ("% ... %(...)")`. C'est à dire, cette méthode d'aspect devrait affecter l'exécution de l'ensemble des fonctions qui satisfont l'expression `"% ... %(...)"`. Dans les « expressions de correspondance » le signe '*%*' et les caractères '*...*' sont utilisés en tant que caractères de remplacement. Le signe pourcentage '*%*' fait référence à n'importe quel type. Par exemple, `"% *"` correspond à tous types de pointeur ainsi qu'à n'importe quelle séquence de caractères dans les identifiants. Aussi, "Affiche_%" correspond à l'ensemble des classes dotées d'un nom commençant par Affiche_. Les points de suspension « ... » équivalent à n'importe quelle séquence de types ou à des espaces de nommage. Par exemple, `"float Racine (...)"` équivaut à n'importe quelle fonction générale chargée de retourner un réel et nommée Racine. Enfin, l'expression de correspondance `"% ... %(...)"` équivaut à n'importe quelle classe ou espace de nommage [Quick Reference, 2006] [Spinczyk et al., 2005].

Les expressions de correspondance représentent des ensembles d'entités de programme nommées comme les fonctions ou les classes. Ainsi, les expressions de correspondance sont toujours des expressions de coupes transverses primitives, chargées de décrire un ensemble de points de jonction dans la structure statique du programme et sont appelées *points de jonction statiques*. Toutefois, dans l'exemple de la Figure 3-3, nous visons une méthode d'aspect pour

décrire un évènement dans le flux dynamique de contrôle du programme, à savoir l'exécution des fonctions. La fonction de coupe transverse *execution ()* est par conséquent utilisée. Celle-ci a pour objectif de produire l'ensemble des points de jonction d'exécution des fonctions données en argument.

Mais dans le cas où on met l'accent sur les points de jonction dynamiques, AspectC++™ dispose de trois types de méthodes d'aspect, *before (), after (), et around ().* Ces méthodes implémentent une partie supplémentaire au comportement du programme. Dans notre exemple, ce comportement dynamique est invoqué au moyen de la déclaration *printf* à la suite de *before( ).* D'un point de vue syntaxique, cette technique est identique au corps d'une fonction et, de ce fait, il est d'ores et déjà possible de considérer *advice body* (ou corps de la méthode d'aspect) comme une fonction membre anonyme de l'aspect. À la place du *before(),* nous aurions pu faire appel à la méthode *after ()* ou encore les deux en même temps. Ainsi, le corps de la méthode d'aspect s'exécuterait tout juste après l'achèvement de l'exécution d'une quelconque fonction. Un corps de méthode d'aspect *around ()* est exécuté à la place du flux de contrôle, censé suivre normalement le point de jonction dynamique.

Aussi, il est possible de combiner les coupes transverses au moyen des opérateurs «*&&* » (intersection), « || » (union) et « ! » (inversion). Par exemple, l'expression `"%Affiche_(int,...)" || "float Select_(...)"` invoque n'importe quelle fonction générale appelée `Affiche_` prenant un entier comme premier paramètre et n'importe quelle fonction générale intitulée `Select_` chargée de retourner un réel. Avec la combinaison de fonctions de coupes transverses, un programmeur peut obtenir des expressions plutôt puissantes afin de décrire l'emplacement où la méthode d'aspect est censée affecter le programme. Par exemple, nous pourrions changer l'expression de la coupe transverse de l'aspect *tracing* comme suit :

```
advice call ("% ...::%(...)")
&& within ("Bataille") : before () {
Cout<<"calling "<<(JoinPoint::signature ());
}
```

La fonction `call()` produit l'ensemble des points de jonction des appels pour des fonctions données. Par opposition aux points de jonction d'exécution, les points de jonction d'appels prennent effet du côté de la routine appelante, c'est-à-dire avant, après ou en même temps que l'appel de la fonction concernée. La fonction de la coupe transverse `within()` se

contente de retourner l'ensemble des points de jonction dans les classes ou les fonctions indiquées. En introduisant la méthode d'appel à l'intersection de `call ("% ...::%(...)")` (soit n'importe quel appel de fonction) et de `within ("Bataille")` (n'importe quel point de jonction dans la classe Bataille), l'aspect ne pistera dès lors que les appels de fonction lancés à partir d'une méthode de la classe *Bataille*.

3.4. Conclusion

Le but avoué de la séparation des préoccupations est de séparer toutes les méthodes dans des aspects isolés, de les coder, de les maintenir ou les remplacer tout à fait indépendamment des autres aspects. La séparation a lieu différenciant les aspects fonctionnels et non fonctionnels. Les aspects non fonctionnels comme le *tracing*, ne pouvant pas être élaborés par des techniques de développement traditionnelles, sont implémentés par des nouvelles approches regroupées sous ce qu'on appelle *développement de logiciel par aspect* ou encore *Aspect Oriented Development* (AOSD). Parmi ces approches, nous pouvons citer la programmation orientée sujet (SOP), la programmation orientée vue (VOP), ou encore la programmation orientée aspect (POA). En ce qui concerne notre travail de recherche, nous nous baserons sur la programmation par aspect qui est un paradigme qui ne consiste pas à décrire le fonctionnement du programme, ou bien encore à décrire sa structure comme dans la programmation orientée objet. Il consiste plutôt à exposer des modifications à lui apporter. Pour mettre au point ces modifications, l'utilisation d'un *tisseur d'aspect* est nécessaire (*aspect weaver*). Ce dernier prend en entrée le code source du programme principal et les fonctionnalités non-fonctionnelles, codées bien évidement en langages aspect, pour donner comme sortie un seul code composé du code principal et des aspects. Dans notre travail, nous utiliserons l'outil AspectC++™ [AspectC++, 2006] qui est un tisseur d'aspect adapté pour le langage C++. Actuellement, il existe un autre tisseur adapté à C++, Taco [Vaysse, 2005], mais contrairement à AspectC++™ qui permet de modifier les méthodes ou les membres d'une classe donnée, Taco permet de modifier les structures de contrôle du langage cible [Vaysse, 2005]. En plus de cela, AspectC++™ ayant subit plus de tests, il serait plus stable que le tisseur Taco [Vaysse, 2005].

Après avoir présenté la notion de gestion et de débogage de mémoire dans le chapitre précédent, introduit la séparation des préoccupations et la programmation par aspect en ce

chapitre, nous établissons dans la prochaine section l'objectif de la recherche ainsi que les différentes étapes de la méthodologie proposée afin de développer notre outil de débogage de mémoire.

3.5. Objectifs de la recherche

La gestion de la mémoire est un élément indispensable dans le cycle de vie du développement d'un logiciel. Comme nous l'avons déjà vu dans le deuxième chapitre, un délaissement de la gestion de mémoire nuirait aux développeurs et à l'application même. Comme solution efficace, les chercheurs ont opté pour une gestion implicite de la mémoire, mais dans notre cas, qui est l'utilisation du langage C++, ce dernier n'offre pas de gestion implicite qui allègerait le travail du développeur et lui permettrait des manipulations sécurisées des zones mémoire. Pour résoudre ce conflit et permettre au C++ d'offrir une gestion implicite de sa mémoire, chercheurs et praticiens ont opté pour l'implémentation d'un débogueur de mémoire. Certain ont pu donner des résultats convaincants au niveau de la détection des erreurs d'accès et de fuites de mémoire [Valgrind Manual, 2008] [Purify User's Guide, 1999], mais aucun n'offre ce qu'on appelle une gestion de mémoire sécurisée, et ceci que se soit au niveau de l'allocation ou de la désallocation des espaces mémoire.

L'objectif primaire de cette recherche est de proposer un outil basé sur la programmation par aspect afin d'assurer pour les applications implémentées en langage C++, une gestion dynamique, implicite et sécurisée de leurs espaces mémoire. Assurer une continuité implicite d'une gestion de mémoire explicite offerte par le langage C++ est aussi notre priorité. Dans le cas du langage C++, l'opérateur *new* est responsable des allocations dynamiques de la mémoire. Avec *new* on crée un objet et l'espace mémoire est implicitement demandé. Une fois l'objet créé, *new* renvoie un pointeur du type de l'objet, et c'est à travers ce même pointeur que l'objet sera manipulé. Dans le langage C++, la notion de mémoire dynamique est très étroitement liée à celle de pointeur.

En effet, une bonne manipulation des pointeurs induit une gestion de la mémoire dynamique sans erreur. L'idée est d'utiliser la programmation par aspect pour mettre au point un gestionnaire de pointeurs. Le principe même de cette approche se résume dans le fait de lister tous les pointeurs alloués tous au long de l'exécution du programme, et d'ajouter à chaque pointeur une série d'informations indiquant les manipulations permises sur le pointeur

ou sur l'objet auquel il réfère. La Section 4.3.3 du Chapitre 4 nous donnera plus de détail sur cette approche. Lors de chaque opération d'ajout, de suppression, ou encore de modification de l'objet référencé, une série de vérifications est appelée pour s'assurer de la conformité de ces manipulations. Ainsi, l'adjonction des aspects se fera dans des endroits biens précis du programme :

- à chaque invocation de l'opérateur *new* : un objet est créé, un pointeur référençant ce même objet sera ajouté directement à la liste des pointeurs;
- à chaque manipulation d'un objet référencé par un pointeur : des séries de vérifications seront appelées, et des mises à jour seront affectées au niveau de l'information accompagnant le pointeur même;
- à chaque invocation de la fonction *delete* : l'objet et son pointeur seront supprimés. Mais avant toute manipulation de ce genre, des séries de vérifications sont appelées, avant et après la suppression.

AspectC++™ est utilisé afin i) d'élaborer les différents aspects qui gèrent la création des objets, ii) le listage de leur pointeur correspondant, iii) toute action associée à ces mêmes objets (création, modification, etc.) iv) et enfin leurs suppressions.

Le Chapitre 4 décrit les différentes étapes de l'élaboration de notre outil. Le Chapitre 5 inclut une comparaison de notre outil avec celui de *Purify* et ce, à travers une étude de cas.

Chapitre 4 : Développement d'un outil de gestion implicite et de débogage de mémoire basé sur la programmation par aspect (AspectC++Debogger)

4.1. Introduction

Nous avons vu dans le Chapitre 2, qu'en plus du fait que le langage C++ ne bénéficie pas d'un gestionnaire implicite de mémoire, les erreurs dues à une mauvaise gestion de mémoire sont assez fréquentes. Plusieurs outils conçus spécifiquement pour gérer la mémoire au sein d'une application C++ ont été mis au point, notamment chez la firme IBM [Purify User's Guide, 1999]. Cette firme a opté pour une instrumentation et une analyse au niveau du code objet, ou encore d'autres travaux de différents chercheurs et praticiens qui ont opté pour une instrumentation au niveau du code binaire [Valgrind Manual, 2008]. Dans ce chapitre, nous présenterons notre outil de gestion de mémoire pour le langage C++ qui utilisera une instrumentation par aspect.

4.2. Gestion de mémoire par aspect

À travers un outil de gestion de mémoire ou encore se qu'on appelle un *débogueur de mémoire*, on facilite la tâche de programmation et surtout la phase de débogage. En effet, la détection des erreurs telles que les fuites de mémoire et les erreurs d'accès sera plus facile, et le programmeur aura plus de facilité à retracer ces éventuelles erreurs qui sont dues à une mauvaise gestion de la mémoire et lui épargnera donc un temps précieux. Dans notre travail de recherche, on va plutôt se spécialiser dans tout ce qui concerne l'allocation dynamique de la mémoire au sein du langage C++, et proposons une technique de listage de pointeurs qui permettra une manipulation sécurisée et un suivi détaillé des espaces mémoire alloués dynamiquement. Cette technique sera implémentée par la programmation par aspect, et plus précisément avec l'outil AspectC++™ [AspectC++, 2006].

La gestion de la mémoire sera considérée comme une préoccupation transverse et sera regroupée en un seul aspect. Ainsi, les objets seront moins encombrés et ne contiendront que leurs fonctions essentielles et assureront alors une meilleure lisibilité, ainsi qu'une maintenance et une réutilisation plus faciles.

4.2.1. AspectC++

AspectC++™ est une extension orientée aspect du langage universel C++. C'est un préprocesseur pour un compilateur régulier de C++. Une fois le tissage d'aspect effectué (*weaving*), il produit en sortie du code standard C++ [Gati, 2004]. Afin de comprendre clairement les concepts fondamentaux du langage AspectC++™, on fournit un simple exemple qu'illustre la Figure 4-1. Dans cet exemple, on traite les allocations de mémoire utilisant la fonction *malloc()*, plus exactement, on essaye d'implémenter un aspect afin d'obtenir de l'information sur le nombre d'occurrences des allocations de *malloc()*.

Dans le programme de la Figure 4-1, on alloue via *malloc* un objet de type la structure *myCalc*, ensuite on fait appel à la fonction *doSomething()*, et enfin l'objet en question est libéré. Ce processus est effectué deux fois.

```
struct myCalc
{
public:
        float a,b;
        float result;
        char opera;
};
void doSomething()
{
        printf("Hello there\n");
}

int main ()
{
        myCalc* ab;
        (myCalc *) malloc(sizeof(myCalc));//allocate malloc-1
        doSomething();//appel de la fonction doSomething()
        free(ab);
        myCalc* bb;
        (myCalc *)malloc(sizeof(myCalc));//allocate malloc-2
        doSomething();//encore un appel de la fonction
        free(bb);       //doSomething()
}
```

Figure 4-1 : Exemple avant tissage de l'aspect.

Nous voudrions que le code d'aspect dépiste chaque allocation que *malloc* effectue, et incrémente à la fois un compteur afin d'obtenir le nombre de fois où on a fait appel à *malloc*. Avant que nous puissions écrire le code aspect, la compréhension de quelques concepts fondamentaux de la programmation orientée aspect est primordiale [Gati , 2004].

4.2.1.1. Les points de jointure (Join point)

Par définition, les points de jointure sont les emplacements dans le code source où les aspects devront être injectés. Les points de jointure peuvent référer à des classes, des structures, des unions, des objets ou des méthodes. Généralement parlant, un point de jointure est un motif (*pattern*) qui est identifié lors de l'exécution du tisseur d'aspect. Dans l'exemple de la Figure 4-1, *malloc(...)* est le motif que nous recherchons. La fonction retourne un pointeur de type *void* et a un argument de type *size_t*, ainsi notre point de jointure sera comme suit :

```
void* malloc (sizet );
```

4.2.1.2. Les coupes transverses (Pointcut)

Une coupe transverse est utilisée pour identifier un groupe de points de jointure et devrait être mis entre guillemets. Afin de définir des points de jointure plus spécifiques, les coupes transverses sont exprimées via les expressions de correspondance (Voir Section 3.3.4). Selon l'objectif du code d'aspect, les coupes transverses peuvent être :

1. un ensemble d'appels ou d'exécutions à partir d'un point de jointure
    - call (point de jointure)

   Par exemple : `call("void test()")` collecte l'ensemble des appels de la fonction *test*.
    - execution (point de jointure)

   Par exemple : `execution("void test()")` collecte l'ensemble des exécutions de la fonction *test*.
2. un ensemble des noms de classes qui sont seulement dérivées des classes où le point de jointure est spécifié
    - derived (point de jointure)

   Par exemple : `derived("maclasse")` collecte tous les noms de classes dérivées de *maclasse*.
3. un ensemble identifiant la portée d'un point de jointure
    - within (point de jointure)

   Par exemple : `call("void foo()")&& within(myClas)` collecte l'ensemble des fonctions *foo* qui sont dans les méthodes de la classe *myClas*.

Dans le cas de notre exemple, pour collecter la totalité des appels de *malloc*, nous utiliserons la coupe transverse suivante :

```
call("void* malloc(size_t)");
```

4.2.1.3. Composition de coupes transverses (pointcut composition)

Les opérateurs logiques peuvent être utilisés afin de combiner les coupes transverses, ils sont utilisés de la même façon comme dans le cas de C++. Supposons que nous voulons enrichir l'exemple de la Figure 4-1 et lui permettre de vérifier si l'espace mémoire déjà alloué par *malloc* a été changé ou non pas *realloc*. Nous utiliserons pour ce propos l'opérateur logique *ou* (||) pour combiner les deux coupes transverses de *malloc* et *realloc*.

```
call("void* malloc(size_t") ||
call("void  realloc(size_t, void* )");
```

Le résultat de la combinaison est une nouvelle coupe transverse qui trouvera aussi bien l'appel de *malloc* que celui de *realloc*.

4.2.1.4. Les coupes transverses nommées (Named Pointcut)

Une coupe transverse nommée peut être déclarée afin de faciliter la composition des coupes transverses. Dans la POA, ce type de coupe transverse joue un rôle important au niveau de la réutilisabilité. Ainsi, la coupe transverse caractérisant *malloc* et *realloc* peut être nommée *memory_allocation ()* et écrite comme suit :

```
Pointcut memory_allocation()=
call("void* malloc (size_t)")||
call("void  realloc (size_t, void* )");
```

Maintenant l'implémentation de *memory_allocation ()* peut être utilisée chaque fois qu'un appel de fonction de *malloc* ou de *realloc* devrait être dépisté.

4.2.1.5. Les expressions de correspondance (Match expressions)

Les expressions de correspondance sont utilisées pour exécuter des filtres spécifiques sur les points de jointure. Une expression de correspondance est très similaire à une méthode de signature de C++. Le symbole ("%") est utilisé comme un joker pour augmenter la flexibilité des filtres sur les points de jointure. Par exemple, les points de jointure suivants ont différentes significations :

```
void* %::malloc(size_t)          //rapporte la fonction malloc retrouvée
                                 //dans n'importe quelle classe

void* maClass::malloc(size_t)    //rapporte la fonction malloc retrouvée
                                 //seulement dans la classe maclasse
```

4.2.1.6. Les conseils (Advice)

Un conseil peut être vu comme une action activée par un aspect une fois un point de jointure correspondant est atteint dans le programme cible. L'activation du code du conseil peut avoir lieu avant, après ou avant et après que le code du point de jointure soit atteint. Le langage AspectC++™ permet de spécifier les conseils grâce aux mots clés *before ()* pour avant, *after ()* pour après et *around ()* qui permet de le faire avant et après. Voici un exemple de conseil avec *before ()* :

```
advice call("void* malloc (size_t)"): void before()
{
cout <<x++<<endl; // Incrémente x avant que la fonction
                  // malloc ne soit atteinte
}
```

Maintenant que nous avons fait le tour de tous les concepts fondamentaux que AspectC++™ peut détenir et pouvons compléter l'exemple de l'aspect de la Figure 4-1. La structure du code aspect est similaire à celle d'un code d'une classe en C++. Notre aspect sera nommé *malloc_counter* :

```
aspect Malloccounter
      {
            int x =0;
            public: //Liste des coupes transverses
            advice call("void* malloc (size_t)"): void before ()
            {
            cout<<x++<<endl;
            //incrémentation de x après avoir trouvé malloc
            }
      }; //end
```

Figure 4-2 : Aspect *Malloccounter*

Le concept est illustré par la Figure 4-3. Lors de chaque appel de la fonction *malloc*, le code aspect de la Figure 4-2 affecte le programme client et se chargera de comptabiliser les appels de *malloc*.

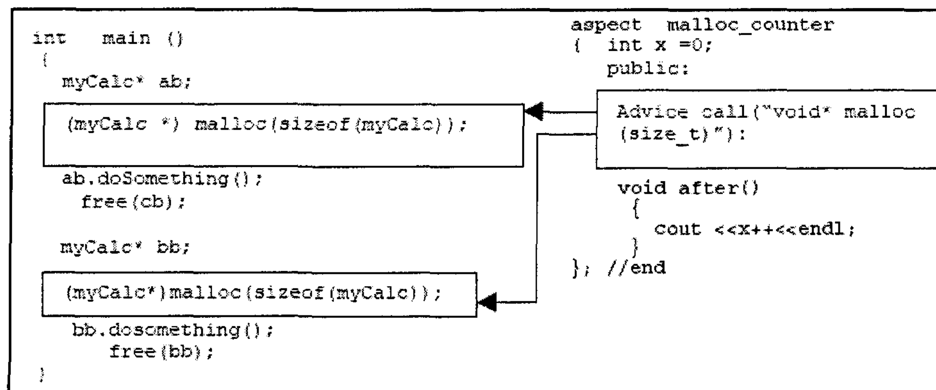


Figure 4-3 : Effet de l'aspect après tissage

Dans cet aspect, le point de jointure `call("void* malloc (size_t)")` se produit toutes les fois qu'un appel de *malloc* est atteint. Le conseil *after ()* contient la portion de code qui sera exécutée après chaque appel de la fonction *malloc* et suscite l'appel du *cout()* pour incrémenter la valeur de X.

4.2.2. Les éléments affectant la consommation de mémoire dynamique en C++

Un point clé pour acquérir des données avec AspectC++™ est la correspondance des motifs *(Pattern Matching)*, c.-à-d. retrouver des objets similaires dans différents codes sources. Les points de jointure sont utilisés pour cette fin. Par exemple, dans le but d'identifier les modèles de fonctions et d'opérateurs dédiés pour la mesure et la gestion de la mémoire dynamique en C++, nous considérons les fonctions d'allocation et de désallocation suivantes :

- **void* operator new**`(size_t size)`

Allocation d'un espace mémoire dont la taille est égale à celle de l'argument size.

- **void* operator new**`[] (size_t size)`

Allocation de tableau.

- **void* operator delete** (**void** * `ptr)`

Désallocation de l'espace pointé *ptr*. Si *ptr* est à NULL, rien ne se passe.

- **void* operator delete**`[]` (**void** * `ptr)`

Désallocation d'un bloc de mémoire d'un tableau.

Afin de traiter des données au niveau des points de jointure, AspectC++™ fournit un mécanisme de traçage au niveau du code source de l'application cliente. Plus exactement, la

classe *JoinPoint* [AspectC++, 2008] qui est une classe propre à AspectC++™ est responsable de ce mécanisme. Les fonctions de cette classe peuvent être employées pour déterminer le code à ajouter ou encore à détecter pour une gestion de la mémoire dynamique. Ci-dessous, on énumère les fonctions auxquelles nous avons eu recours pour l'extraction des données concernant toute gestion de mémoire dynamique.

4.2.2.1. Les membres de la classe JointPoint

- *const char* signature()*

Retourne le nom d'une fonction ou d'un attribut dans le cas d'une surcharge d'un opérateur.

- *Result* result()*

Retourne un pointeur vers l'emplacement mémoire désigné par la valeur résultat, 0 si la fonction ne retourne aucune valeur. Dans notre cas, on utilisera cette fonction pour passer en paramètre l'adresse de chaque espace mémoire que l'opérateur *new* allouera.

- *args(type pattern, ...)*

Retourne une liste des types filtrés pour des méthodes ou des attributs avec une expression de correspondance dans le cas d'un constructeur par copie.

- *void proceed ()*

Exécute les points de jointure dans un conseil *around ()*.

4.2.2.2. Exemple de collection de données lors d'une allocation et d'une désallocation

Les exemples suivants présentent un code aspect pour une allocation de mémoire à travers l'opérateur *new* et une désallocation à travers la fonction d*elete*. Ils utilisent des fonctions membres de la classe *JoinPoint* afin d'extraire la taille de l'espace mémoire, le pointeur alloué et le pointeur désalloué.

```
advice call("void* new(...)")&&args(size):around (size_t size)
{
      printf ("taille de la mémoire %ld: \n", size);
      tjp->proceed ();
      printf ("Pointeur alloué : %p\n", *tjp->result ());
}
```

- *args(size)*

Choisit tous les appels ou exécutions des points de jointure ayant *size* comme argument. Ainsi, on sépare les fonctions qui ont *size* comme argument de celles qui ne l'ont pas.

- *around (size_t size)*

Dans le corps du conseil *around*, la fonction *tjp->proceed* permet l'exécution du point de jointure. L'exécution du point de jointure *new(...)* est importante dans ce cas, car nous aurons besoin de sa valeur de retour, qui est bien évidement un pointeur portant l'adresse de la mémoire allouée *'*tjp->result ()'*.

Pour vérifier si la mémoire allouée par l'opérateur *new* est bel et bien libérée ou non, nous ajoutons le code ci-dessous lors de l'exécution de la fonction *delete*. On vérifie si le pointeur sur cette mémoire est à NULL ou non.

```
advice call("void delete(...)")&&args(ptr):
before(void* ptr)
{
        printf ("Pointer deallocated: %p\n", ptr);
}
```

- *args (ptr)*

Choisit tous les appels du point de jointure *delete(...)* avec un argument *ptr*.

- *before(void* ptr)*

La fonction *before* force le code du conseil à s'exécuter avant le point de jointure et à prendre en considération ce dernier seulement lorsque le type void * est inclus.

- *printf ("Pointer désalloué: %p\n", ptr);*

Imprime le pointeur de l'adresse désalloué et affiche NULL si rien ne se passe.

4.2.2.3. Collection des noms de classes

Nous voulons permettre à notre outil de débogage de mémoire de retracer les informations concernant le type de chaque objet alloué ou désalloué. Nous entendons dire par '*information sur le type d'objet*' les classes où chaque objet a été créé.

```
pointcut all_classes()=classes("%");
advice all_classes():void print()
{
        cout <<"Address of"<<thisJoinPoint->toString()
        <<"object:"<<(void*)this<<endl;
}
```

- *Classes ("%")*

Retourne toutes les classes.

- *thisJoinPoint->toString*

Retourne un texte lisible d'une chaine de caractères représentant l'objet de la classe.

4.3. Outil de gestion et de débogage de mémoire par aspect

Dans le cadre de notre mémoire, notre priorité est d'assurer une bonne gestion de la mémoire allouée d'une manière dynamique, c.-à-d. lors de l'exécution de l'application cliente. Mais cela n'empêche pas que plus loin dans nos recherches, on essayera de trouver des solutions aux problèmes que des débogueurs de mémoire comme *Purify* et *Memcheck* ont pu rencontrer lors de certaines manipulations de la mémoire.

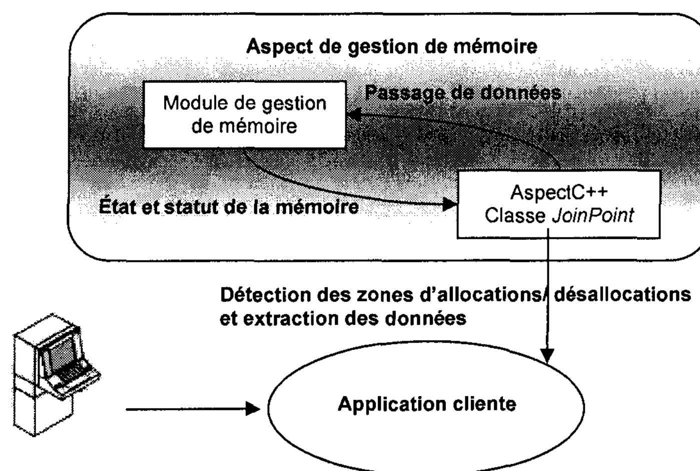


Figure 4-4 : Modèle de gestion de mémoire ***(AspectC++Debogger)***

L'idée même de notre approche se résume dans l'implémentation d'un module appart entière de gestion de mémoire, et une fois établi, son intégration au niveau de l'application cliente se fera à l'aide de la programmation par aspect.

4.3.1. Aspect calculant le temps d'exécution de différentes applications

4.3.1.1. Mesure du temps d'exécution

La mesure du temps d'exécution CPU d'une portion de code est parfois utile (ex. : lors de l'optimisation de l'efficacité d'une procédure), mais dans la plupart des cas, on n'a pas toujours les moyens de consacrer une machine exclusivement aux tests. Le langage C++ met à la disposition des programmeurs des procédures comme *clock()*, *GetTime()*, *QueryPerformanceCounter()* ou d'autres permettant le calcul du temps "réel" qui s'est écoulé entre deux points du code. Lorsqu'on ne peut pas dédier une machine aux tests, l'inconvénient

de ces méthodes est de prendre en compte le temps des autres applications qui tournent sur le système en concurrence avec celles qui sont testées. Ce cas de figure est par exemple rencontré sur une machine dont on ne possède pas les droits suffisants pour désactiver une défragmentation, une analyse antivirus automatique, etc. Le temps consommé par ces applications n'est alors pas décompté par les méthodes suscitées. Cela fausse totalement les mesures.

On propose dans notre aspect de gestion de mémoire, et plus précisément dans le module de gestion de mémoire, une classe *Chrono* fournissant un moyen simple pour mesurer le temps processeur d'un processus courant entre deux points d'exécutions. La fonction *GetProcessTimes* de l'API de Windows sera utilisée à cette fin. L'utilisation de cette procédure nous évite d'avoir recours à une gestion avancée d'un système multitâche pour contrôler la préemption de notre processus. Notre aspect dispose en outre d'une mémoire comparable à celle d'un chronomètre classique afin de pouvoir ajouter une mesure à d'autres.

4.3.1.2. Détection des portions de codes à mesurer avec Aspect C++

Notre objectif est la mesure du temps d'exécution de toute application cliente analysée par notre outil de débogage de mémoire. L'implémentation de la classe *Chrono* à elle toute seule est insuffisante, une instrumentation du code de l'application cliente est obligatoire afin de détecter le *début* et la *fin* d'exécution des programmes cibles. Plus en détail, notre aspect *AspectChrono* devra détecter toute exécution de la procédure *main()* au moyen de la coupe transverse suivante :

```
pointcut chrono() = execution ("% main(...)");
```

Ensuite déclencher le chronomètre juste avant le commencement de la procédure main(), on utilisera le conseil suivant :

```
advice chrono(): before ()
{
      printf ("detection de la fonction principale main");
      CalculTempsExection.Start();
}
```

On soulignera l'utilisation du mot clé *before* qui est responsable de l'exécution des portions du code du conseil avant l'exécution de la procédure *main()*. L'appel de la méthode *CalculTempsExection.Start()* déclenche le comptage du temps d'exécution.

En fin, on utilisera un deuxième conseil afin de détecter la fin de l'exécution de la procédure *main()*, on utilisera pour cette fin le mot clé *after* :

```
advice chrono(): after ()
{
        CalculTempsExection.Stop();
        printf("Temps consommé: %dms.\n", CalculTempsExection.m);
}
```

La méthode *CalculTempsExection.Stop()* arrêtera le chronométrage et un affichage du temps d'exécution de l'application cliente est atteint avec la méthode *printf.*

4.3.2. Aspect évaluant la mémoire utilisée lors de l'exécution d'une application

Une des premières fonctionnalités qu'un débogueur de mémoire devrait assurer est l'évaluation de la mémoire que l'application cliente a utilisé tout au long de son exécution. En effet, ce type d'information peut toujours être utile en permettant au programmeur de connaître les besoins de son application en espace mémoire. Dans le cas du langage C++, la notion de mémoire dynamique est très étroitement liée à celle de pointeur. Plus précisément, c'est à travers l'opérateur *new* qu'on alloue de l'espace mémoire dynamiquement. L'opérateur *new* crée un objet et l'espace mémoire est implicitement demandé. Une fois l'objet créé, l'instruction *new* renvoie un pointeur du type de l'objet (pointant bien évidemment sur ce dernier). Le fait de contrôler les pointeurs d'une application durant son exécution nous donne la possibilité d'évaluer l'espace mémoire utilisés et aussi d'assurer une bonne gestion de la mémoire dynamique.

Dans ce cas de figure, la taille de chaque adresse mémoire pointée sera comptabilisée et ainsi fournir la taille de la mémoire consommée durant tout le cycle de vie de l'application. Plus loin dans ce travail, il sera question de l'implémentation d'un aspect capable de fournir i) la taille de la mémoire utilisée en un instant précis du cycle de vie de l'application, ii) et la taille de la mémoire non libérée une fois le cycle de vie de l'application terminé.

Comme approche, on a choisi de lister tous les pointeurs dans une liste doublement chainée ; chaque élément de la liste correspondra donc à une allocation dynamique bien précise et donnera accès à un nombre d'informations à son sujet. À partir de ces informations, une série de tests pourra être établie afin de garantir une bonne gestion de la mémoire. On notera que

cette liste doublement chainée, qu'on nommera *conteneur,* nous sera utile tout le long de notre travail. La structure de la liste sera comme suit :

```
// Informations concernant toute allocation dynamique
typedef struct
{
        unsigned long nbr_signature; //signature
        bool est_alloue; //Pointeur alloué ou libéré
        size_t taille; //Taille allouée (sans informations de
                //débogage)
        const char* nomfich; //Nom du fichier source
        ulong numligne;     //Ligne du fichier
        struct PtrChaine *ptr_liste;//Pointeur dans la liste
        unsigned long somme_inf; // Somme des informations (checksum)
} InfoPointeur;
```

```
// Un pointeur chaîné
struct PtrChaine
{
        struct PtrChaine *precedent; // Elément suivant
        struct PtrChaine *suivant;  // Elément suivant
        ulong nbr_signature;        // signature
        InfoPointeur *ptr_info;     // Pointeur à libérer
};
```

Figure 4-5 : Structure du conteneur et les informations ajoutées pour chaque pointeur

4.3.2.1. Détection des appels de l'opérateur new

Afin de lister tous les pointeurs avec leurs informations correspondantes, notre aspect se doit de détecter toute allocation dynamique, c.-à-d. tout appel de l'opérateur *new.* On utilisera pour cette fin la coupe transverse suivante :

```
pointcut New( size_t S)  =("% ...::operator new(...)" ||
                           "% ...::operator new[](...)"
                           ! within("conteneur")&& args(S);
```

Cette coupe transverse cible la totalité des classes '`% ...:`' et détectera touts les opérateurs *new* quelque soit le nombre et le type d'arguments qu'ils prendront '`new (...)/new[] (...)`'.

4.3.2.2. Changement du comportement de l'opérateur new en allouant deux espaces mémoire supplémentaires

Une fois une allocation dynamique est atteinte, notre aspect alloue deux espaces mémoire supplémentaires : i) Un premier d'une taille égale à 16 octets, dont l'utilité sera expliquée plus loin dans ce chapitre, ii) et un deuxième de la taille d'une structure de donnée *InfoPointeur* afin de contenir les informations correspondantes à l'allocation détectée (voir Figure 4-5).
À ce niveau, AspectC++™ intervient encore une fois et nous permet de changer le comportement de l'opérateur *new* (qui été sensé réserver de l'espace mémoire pour stocker les données propres à l'application seulement) en lui permettant d'allouer des espaces mémoire supplémentaires.

```
        aspect MonAspect
        {
        public:
L4                      size_t mémoire_util = 0 ;
        public:
        //securise la gestion de mémoire afin de s'assurer qu'aucune autre
        //gestion n'est en place (eviter les conflits)

L7      advice classes("%"):slice class NEW
        {

        public:
        void* operator new(size_t size)
        {
                        return  malloc(size+16+sizeof(InfoPointeur));
        }
        ...
        }
```

Figure 4-6 : Changement du comportement de l'opérateur *new*

AspectC++™ permet d'introduire une tranche de code à l'aide des mots clés *slice class*. En effet, un *slice* est un fragment d'élément du langage C++ définissant une portée. Par exemple, on peut prolonger la structure statique d'un programme via un *slice* en fusionnant sa portée avec une ou plusieurs classes. À la ligne 7 de la Figure 4-5, on a fusionné le nouveau comportement de l'opérateur *new* avec toutes les classes de l'application cliente, et ce, en utilisant `classes("%")` dans le code du conseil.

4.3.2.3. Extraction de la taille de l'espace mémoire alloué

À ce stade, il ne manquerait plus que l'extraction de la taille de l'espace alloué dynamiquement, pour ensuite la sauvegarder dans le champ *InfoPointeur->taille*. En Section *4.3.2.1*, la coupe transverse *NEW* comprend un argument de type *size_t*. Ceci signifie qu'une valeur de type *size_t* est fournie chaque fois que le point de jointure décrit par la coupe transverse *NEW* est atteint. La fonction *args(S)* utilisée dans la coupe transverse délivre tous les points de jointure du programme client où un argument de type *size_t* est utilisé.

```
        advice New( S) : after (size_t S)
        {
                ...
                InfoPointeur-> taille = S;
L5              mémoire_util = mémoire_util + InfoPointeur-> taille
                ...
        }
```

Figure 4-7 : Déclaration d'un aspect avec accès à une variable contextuelle

La déclaration du conseil rapporté à la Figure 4-6 contient un paramètre formel de type *size_t*, ce type de déclaration lie l'exécution du conseil en question à celui du point de jointure décrit dans la coupe transverse *NEW* de la Section *4.3.2.1.* Ainsi le paramètre *size_t* que l'opérateur *new* a pris comme argument au sein de l'application cliente, devient utilisable dans le code du conseil autant qu'un paramètre d'une fonction ordinaire ; et une transmission de la taille de l'espace alloué au champ *InfoPointeur-> taille* devient possible.
Enfin, pour calculer l'espace mémoire qu'une application cliente consomme dynamiquement tout au long de son exécution, il suffirait de sommer la valeur *InfoPointeur-> taille* dans une variable de type *size_t* chaque fois qu'une allocation dynamique est atteinte. Cette variable serait un membre public de l'aspect (ligne 4 de la Figure 4-5) et l'instruction même de la sommation se trouvera au niveau du conseil *NEW* (ligne 5 de la Figure 4-6).

4.3.3. Aspect pour sécuriser les allocations et les désallocations de mémoire

4.3.3.1. Allocation sécurisée

La sécurisation d'un espace mémoire se résume à la collection des informations pertinentes concernant son allocation. Ainsi, lors de toute manipulation des données que

l'espace alloué contient, il nous est possible d'entamer une série de vérifications afin de valider l'accès à ce dernier.

En effet, à la suite de toute détection de l'opérateur *new*, on fait appel à la méthode *NewSecurise*. Cette méthode fait partie de notre module de gestion de mémoire, et sa principale tâche est de sécuriser toute allocation dynamique que l'application cliente effectue. Elle prendra comme argument i) un pointeur pointant sur l'adresse de la mémoire retournée par l'opérateur *new*, ii) une variable de type *size_t* retournant la taille de l'espace mémoire allouée, iii) un pointeur constant de type *char** pour retourner le nom du fichier où l'allocation s'est produite, iv) et enfin une variable positive et constante de type *long* pour retourner le numéro de ligne où l'allocation s'est produite. Le prototype de la méthode *NewSecurise* sera comme suit :

```
void* NewSecurise (void * PrincipalP,size_t size,const char*
                   nom_fich, const unsigned long num_ligne)
```

Les fonctionnalités qu'AspectC++™ nous procure se chargent d'extraire les données qu'on passera à *NewSecurise* comme arguments. Vu que l'extraction de la *taille* à déjà été décrite dans la Section 4.3.2.3, on se contentera d'expliquer l'extraction du premier et des deux derniers arguments :

a. Extraction du pointeur retourné par l'opérateur new

La méthode *result()* retourne un pointeur vers l'emplacement mémoire désigné par la valeur résultat, 0 si la fonction ne retourne aucune valeur. Dans notre cas, on utilisera cette méthode pour passer en paramètre l'adresse de chaque espace mémoire que l'opérateur *new* allouera. Cette méthode est accessible à travers l'objet *tjp* (ThisJoinPoint), elle est de type *joinpoint* et elle est toujours disponible même dans un code conseil.

b. Extraction du nom de fichier et du numéro de ligne

L'utilisation des variables `__FILE__` et `__LINE__` lors de la redéfinition du comportement de l'opérateur *new*, nous permet de détecter le nom du fichier et le numéro de la ligne où l'allocation de la mémoire s'est produite. Le tout est implémenté dans la portion d'un code *slice classe*. L'opérateur *new* prendra donc deux arguments en plus de l'argument *size* :

```
advice classes("%"):slice class NewFileLine
{
        public:
        void* operator new (size_t size, const char* nom_fich, const
                unsigned long num_ligne)throw (std::bad_alloc);
        #define new new (__FILE__, __LINE__)
};
```

NewSecurise fera appel à une première méthode afin de lier toute allocation dynamique à son élément correspondant du *conteneur* et une deuxième méthode qui se chargera d'initialiser les zones mémoires supplémentaires :

c. Coordination du conteneur avec les allocations dynamiques atteintes dans l'application cliente

La méthode *ConteneurPointeur_Ajoute* ajoute un élément à notre *conteneur* et le lie avec le pointeur retourné par l'opérateur *new.* Ainsi, toute *DAC*[1] aurait donc son élément correspondant dans le *conteneur. ConteneurPointeur_Ajoute* prendra comme argument un pointeur *InfPtr* pointant sur la zone *InfoPointeur*[2] qui accompagne la *DAC*, voir la Figure 4-7 ci-dessous.

[1] *DAC* : Donnée allouée pour l'application cliente

[2] *InfoPointeur* : Définition dans la Figure 4.4

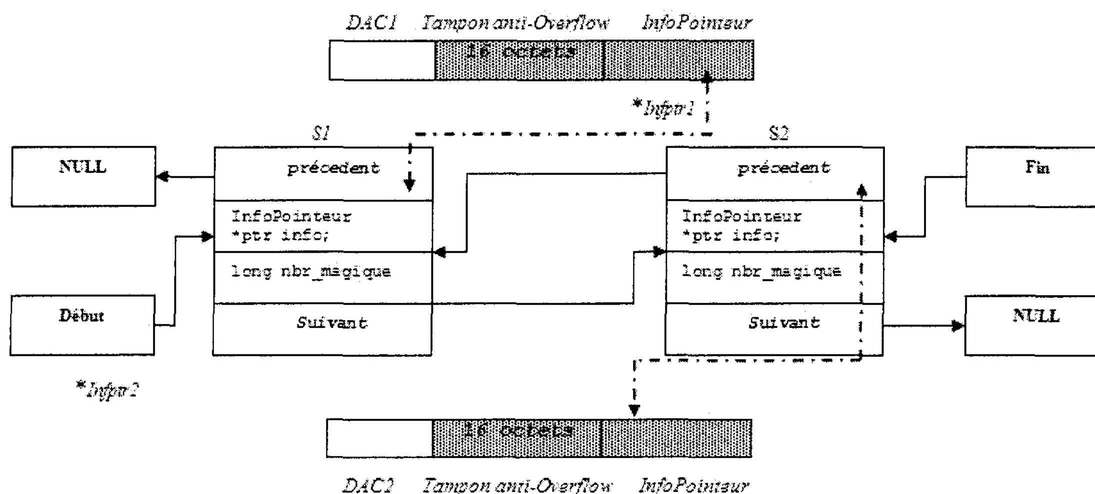


Figure 4-8: Modèle d'allocation accompagnant la DAC

La liaison s'effectue à travers deux opérations d'affectations :

- Chaque élément du conteneur aura son pointeur '*ptr_info*' pointé sur sa zone réciproque *InfoPointeur* allouée par la méthode *NewSecurise*.
- Et inversement, chaque élément *InfoPointeur* alloué avec la DAC pointera sur son élément réciproque du conteneur. A ce niveau, la déclaration de l'attribut **ptr_liste* de type *PtrChaine* dans la structure *InfoPointeur* est justifiée (Figure 4-4).

L'exemple de la Figure 4-7 sera complété par ces deux lignes de codes qui représenteront les affectations da la structure S1 à son élément correspondant, et inversement :

```
s1 -> ptr_info = InfPtr1; // 1ere afectation
InfPtr1-> ptr_liste = s1; // 2eme affectation
```

d. Initialisation des informations accompagnant toute allocation

La méthode **NewPreparePtr* est appelée. Elle se chargera d'initialiser les informations que chaque pointeur est sensé avoir et son prototype serait comme suit :

```
void *NewPreparePtr (InfoPointeur *info, size_t size,
        const char *nom_fich, unsigned long num_ligne)
```

Si on continue dans le même exemple de la Figure 4-7, les informations du pointeur *Infptr1* seront initialisées et prendront les valeurs suivantes :

```
Infptr1-> nbr_magique = MALLOC_NBR_MAGIQUE;
Infptr1-> est_alloue = true;
Infptr1-> taille = size;
Infptr1-> nom_fich = nom_fich;
Infptr1-> num_ligne = num_ligne;
Infptr1-> somme_info =
MallocCalculeSomme (Infptr1);
//calcule la somme des//information de Infptr1
```

e. Initialisation du Tampon _Overflow

Afin de détecter les erreurs d'*overflow*, on a opté pour une méthode très simple et déjà utilisée dans le domaine du génie logiciel : On alloue quelques octets en plus, on les initialise, et enfin on vérifie que cette zone n'a pas été modifiée. Pour l'initialisation, on écrit une chaîne simple, par exemple, on peut partir d'une valeur fixe puis on incrémente à chaque itération. La taille de cette zone mémoire ajoutée ne doit pas être trop importante, car chaque pointeur prendra cette taille en plus. Dans notre cas et comme l'indique la Figure 4-7, on choisit d'allouer 16 octets en plus. Pour mettre en pratique ce raisonnement, on fait appel à la méthode *New_PrepareOverflow* qui prend comme argument un pointeur de type *InfoPointeur.*

```
void Malloc_PrepareOverflow (InfoMalloc *info)
```

La méthode *New_PrepareOverflow* se charge de calculer la position du tampon *anti-Overflow* pour ensuite le remplir avec des valeurs fixées.

```
// Calcule la position du tampon anti-overflow
ptr = (char *)info +sizeof(InfoMalloc) +info -> taille;
// Remplit le tampon avec des valeurs fixées
for (i=0; i<TAILLE_TAMPON_OVERFLOW; i++)
{
    *ptr = rand;
    ptr++;
    rand++;
}
```

4.3.3.2. Validité d'un pointeur

Pour tester la validité d'un pointeur, on effectuera différentes vérifications dont certaines seront plus simples que d'autres. Par exemple, vérifier que si la taille de l'espace alloué n'est pas nulle, alors un message d'erreur doit être rapporté.

```
if (size == 0) ERREUR ("Taille du pointeur alloué est nulle!");
```

Sinon pour des vérifications plus complexes, on utilisera les informations stockées dans les espaces mémoire supplémentaires : avant toute manipulation d'un pointeur, que ce soit pour un accès aux données sur lesquelles il pointe, ou encore pour une suppression définitive de ces dernières, il faudrait être sûr qu'on pointe sur la bonne adresse. Les attributs déjà initialisés lors de l'*allocation sécurisée* seront nos indicateurs. La méthode *VerifieInfoPointeur* effectuera différents tests, elle prendra comme argument un pointeur de type *InfoPointeur* supposé pointer sur les informations des données à manipuler (par exemple, **Infptr1* dans le cas de la Figure 4-7). Nous énumérons ci-dessous les différents tests que la méthode *VerifieInfoPointeur* effectue :

1. Vérifier que l'attribut *nbr_magique* que chaque pointeur possède est bien égal à la variable globale et constate *NEW_NBR_MAGIQUE.* Ce genre de test nous assure le fait qu'on pointe sur une zone valide. Dans le cas contraire, un message d'erreur sera rapporté.

```
// Vérifie les nombres magiques
if (info -> nbr_magique != MALLOC_NBR_MAGIQUE)
                      ERREUR ("Pointeur invalide");
```

2. Additionner les données d'un pointeur constitue un indicateur important. En effet, chaque pointeur doit avoir une sommation unique vu qu'il ne peut y avoir deux pointeurs détenant un même statut, un même numéro de ligne, un même non de fichier et une même taille (le tout à la fois).

```
unsigned long MallocCalculeSomme (const InfoPointeur *info)
{
  // Calcule la somme des données d'un pointeur
return
((unsigned long) (info -> nbr_magique)
+( unsigned long) (info -> est_alloue)
+( unsigned long) (info -> taille)
```

```
+(unsigned long)(info -> nom_fich)
+( unsigned long)(info -> num_ligne));
}
// Vérifie la somme
if (info -> somme_info != MallocCalculeSomme (info))
ERREUR ("Somme incorrecte");
```

3. Vérifier qu'on n'a pas écrit en dehors du tampon *overflow*. Dans le cas contraire, un message d'erreur est rapporté indiquant la ligne et le nom du fichier où l'erreur s'est produite.

```
if (!TamponOverflowIntact(info))
{
ERREUR ("Erreur d'overflow (tampon %s:%u)",
      info -> nom_fich, info -> num_ligne);
}
```

Pour les erreurs d'*underflow*, la signature de nos informations et la "*checksum*" (somme de vérification) s'occupent de les détecter. Un gain au niveau de l'espace mémoire est suscité.

4.3.3.3. Désallocation sécurisée d'une zone mémoire

Pour une désallocation sécurisée, on fera appel à la méthode *DeleteSecurise* qui prendra en argument l'adresse des données à supprimer et le numéro de ligne et le nom de fichier où l'allocation s'est produite.

```
void DeleteSecurise (const void *ptr,
const char* nom_fich, const unsigned long num_ligne)
```

DeleteSecurise aura comme argument un pointeur de type void, c.-à-d. un pointeur nul pouvant rapporter n'importe quel type de donnée. La sécurisation de la désallocation est atteinte à travers la série d'instructions suivante :

1. on vérifie la validité du pointeur à l'aide de la méthode *VerifieInfoPointeur* déjà citée dans la Section 4.3.3.2 ;
2. on s'assure que les données n'ont pas été désallouées dans un temps antérieur (l'attribut *est_alloue* doit être à *true*) ;
3. on sort du *conteneur* l'élément correspondant aux données à supprimées ;
4. on met l'attribut *est_alloue* à *false* ;
5. et enfin on libère la mémoire à l'aide de la méthode *free*.

4.3.4. Aspect de détection des erreurs de fuites de mémoire

En listant toute allocation dynamique dans le conteneur, il nous est possible de voir quelle zone mémoire est encore allouée, ou peut être qu'on a oublié de désallouer. Lors du parcours de la liste doublement chainée, chaque élément rencontré peut être sujet d'une fuite de mémoire. Notre tâche est de fournir au programmeur tous les renseignements spécifiques à chaque allocation, et c'est à lui que revient la décision de la considérer comme étant une fuite de mémoire ou non. Un rapport de fin d'exécution est fourni au programmeur et comportera les détails suivants :

1. le nombre de pointeurs encore alloués;
2. la taille totale de la zone mémoire encore allouée dynamiquement;
3. le nom du fichier et le numéro de ligne de chaque allocation.

La méthode responsable de ces opérations est *ConteneurNew_Affiche*. Contrairement aux autres méthodes, elle ne prendra aucune adresse comme argument. Elle comportera deux variables locales *nbr_pointeur* et *taille_totale* pour stocker réciproquement le nombre de pointeurs et l'espace mémoire encore alloué. Vu que chaque élément de la liste est directement lié à une *DAC*, et ceci, à travers l'attribut *ptr_info* (voir Figure 4-7), il nous est possible de collecter lors du parcours de la liste toutes les informations nécessaires. En effet, une fois un pointeur atteint, i) on procède à la validation de son adresse, encore une fois à l'aide de la méthode *fInfoPointeur*, ii) on somme sa taille à la variable *taille_totale*, iii) et on incrémente la variable *nbr_pointeur*. Bien évidemment, il nous est possible de fournir une quantité d'informations concernant chaque pointeur atteint, incluant bien sur le nom du fichier et le numéro de ligne où l'allocation à été effectuée (voir Figure 4-5). Si le programmeur juge qu'un pointeur n'est plus nécessaire, il suffit de faire appel à la méthode *FreeSecurise* et il sera supprimé d'une manière sécurisée.

4.4. Conclusion

Au sein de ce quatrième chapitre, nous avons introduit les concepts fondamentaux d'AspectC++™ utilisés dans notre outil, présenté l'outil de débogage *AspectC++Debogger*, montré quelques exemples d'utilisation, pour ensuite détailler la réalisation de notre outil de débogage de mémoire pour C++.

En effet, nous avons implémenté une méthode de gestion et de contrôle des pointeurs afin d'assurer le débogage de la mémoire dynamique. On a choisi de lister les pointeurs dans une liste doublement chainée. Chaque élément de la liste correspond à un pointeur, et contient des informations au sujet de son allocation. En se basant sur ces informations, une série de tests est établie afin de valider tout accès ou traitement des données allouées dynamiquement.

Aussi nous avons détaillé dans ce chapitre, les différents aspects responsables de l'instrumentation du code de l'application cliente. Chaque aspect étant implémenté séparément, nous nous sommes servis d'un tisseur AspectC++™ pour son intégration et tissage au niveau de l'application cliente [AspectC++, 2008].

La totalité des fonctionnalités offertes et groupées constitue l'outil *AspectC++Debogger.* Le Chapitre 5 portera sur une évaluation de ce dernier. Une étude de cas comparant *AspectC++Debogger* avec l'outil *Purify* décrit dans le Chapitre 2 sera élaborée pour cette fin.

Chapitre 5 : Étude de cas et résultats expérimentaux

Dans ce chapitre, nous allons, à travers une étude de cas, comparer deux outils de débogage de mémoire pour le langage C++. Le premier outil, *AspectC++Debogger*, est celui proposé au Chapitre 4. Le deuxième outil, *Purify* (IBM), a été choisi, vu que nous l'avons approché expérimentalement au Chapitre 2 et que c'est aussi une référence dans le domaine. La plate-forme matérielle utilisée pour les expériences est un DellD810 avec Pentium M 1,86 GHz CPU et 2 Go de mémoire. Cette étude vise trois objectifs :

- Mesurer la capacité d'*AspectC++Debogger* à détecter les erreurs dues à une mauvaise gestion de la mémoire en C++.
- Démontrer qu'*AspectC++Debogger* peut être utilisé efficacement pour mettre à jour les vulnérabilités indésirables de gestion de mémoire au niveau du code source.
- Exploiter cette expérience afin d'améliorer l'implémentation des tisseurs d'AspectC++™.

Pour effectuer cette comparaison, nous partons de l'analyse d'une même application cliente à travers chacun des deux outils. Cette comparaison se portera sur les critères suivants :

1. habilité de détection et de traitement des différents types d'erreurs de gestion de mémoire C++ ;
2. coexistence d'une gestion implicite et explicite de la mémoire ;
3. ralentissement de l'application cliente et diminution de ses performances ;
4. facilité et transparence d'utilisation ;
5. généricité de l'outil ;
6. stabilité d'AspectC++™ dans le cas de l'outil *AspectC++Debogger.*

Dans ce chapitre, nous argumenterons dans une première section notre choix de l'application cliente autour de laquelle se déroulera cette étude de cas. Après, nous fournirons un manuel d'utilisation d'*AspectC++Debogger* (celui de *Purify* a été présenté dans le chapitre 2). Ensuite, nous détaillerons les résultats de l'étude de cas et clôturons le chapitre avec une conclusion.

5.1. Application à analyser et motivations

Dans cette section, nous argumentons notre choix de l'application analysée durant l'étude de cas. Ensuite, nous détaillerons la manière avec laquelle elle sera exécutée sous l'outil *AspectC++Debogger*. Notons que le manuel d'utilisation de ce dernier sera disponible à la fin de cette section.

5.1.1. Choix de l'application et Motivations

Le choix de l'application cliente est important, il nous permettra d'élargir nos chances afin d'effectuer une étude de cas *robuste* et qui aboutira à une évaluation *juste* de notre outil. L'application devra obéir à certaines normes que nous avons mises en place. En effet, i) il est indispensable que l'application cliente soit écrite avec le langage C++, ii) qu'elle utilise une *approche d'allocation dynamique* des espaces mémoire dédiés pour ses principales exécutions (le cas des allocations statiques n'est pas écarté), iii) et enfin qu'on puisse accéder au code source de cette application (connaître de prêt à quoi ressemble les erreurs détectées, procéder à des modifications si nécessaire et injecter d'éventuelles erreurs, ne peuvent que renforcer l'étude de cas).

Pour pouvoir réunir ces exigences, nous nous sommes intéressés aux logiciels *Open Source*. En effet, le principe même de ce type de logiciels est que l'utilisateur peut librement utiliser et modifier le code source. Certes, ce principe soulève un débat controversé sur la sécurité et la fiabilité au sein de ce type de logiciels, mais des statistiques révèlent qu'après deux décennies de commercialisation de logiciels fabriqués en grande série (Commercial-Off-The-Shelf/COTS), le marché connait une migration inexorable vers ces logiciels dits Libre et Open Source (FOSS) [Tlili, 2009]. À date, plusieurs logiciels libres sont largement disponibles et sont considérés aussi matures et sûrs que leurs équivalents COTS, et ceci à un faible coût ou gratuitement [Paulson et Succi, 2004] [Paulson et Succi, 2004].

Après avoir effectué des recherches dans des sites spécialisés dans le développement d'applications C++ [Deitel, 2010], nous avons trouvé une application Open Source qui obéit aux normes fixées. Nous la présentons dans la section suivante.

5.1.2. Le Jeu Asteroids

L'application *Asteroids* choisie, est un jeu 3D Open Source basé sur des vieux classiques d'arcade dans lesquels le joueur doit détruire des astéroïdes en les cassant en petits morceaux. *Asteroids* a été développé sous *Visual C++ 6.0* et avec *DirectX 8.1*. *DirectX* est une collection

de bibliothèques de *Microsoft* destinées à la programmation d'applications multimédias, plus particulièrement des jeux ou des programmes faisant intervenir de la vidéo sur les plates-formes *Microsoft* (Xbox, systèmes d'exploitation Windows). Le développement d'*Asteroids* s'est déroulé durant deux mois et le jeu est exécutable sur n'importe quel système d'exploitation *Windows* muni de *DirectX* et une *carte d'accélération 3D*. L'application comporte un total de 34 classes écrites en C++ et 68 allocations dynamiques d'espace mémoire (http://asteroids3d.sourceforge.net/).

Le code source complet d'*Asteroids* ainsi qu'une version complète du projet exécutable sur Visual C++ 6.0 sont téléchargeables à partir de sa page web [Asteroids, 2010]. En ce qui concerne *DirectX*, nous n'avons pas pu retrouver la version 8.1, nous avions eu recours à la dernière version *DirectX 9.1*. Le tout a été facilement téléchargeable pour ensuite être installé [DirectX, 2010].

Figure 5-1 : Capture d'écran du jeu *Asteroids*

5.2. AspectC++Debogger et l'intégration de ses principaux composants

Cette étape que nous avons appelée *intégration des principaux composants* fut une étape assez délicate. Elle se présente comme suit :

5.2.1. Visual Studio 2005 et AspectC++Add-In

Pour l'étape de conception et de tissage des aspects nous avons opté pour l'outil *AspectC++Add-In for Microsoft Visual C++*, ce dernier est connu pour sa facilité d'obtention ainsi que sa facilité d'utilisation [Sioud, 2006]. Comme son nom l'indique, cet outil de tissage d'aspect nécessite un *Microsoft Visual* comme environnement de travail, d'où notre choix pour *Microsoft Visual Studio 2005.*

AspectC++Add-In est un produit appartenant à un fournisseur de solutions et d'outils informatiques connu sous le nom de *pure*-systems [pure-systems, 2010]. Son intégration au sein de *Microsoft Visual Studio 2005* a nécessité une licence qu'on a obtenue gratuitement dans un premier temps. Ensuite, il a fallu négocier l'obtention de cette même licence avec un des membres de l'équipe de *pure-systems* se trouvant en Allemagne. Cette licence n'était plus disponible et devait être renouvelée chaque 30 jours. Une licence permanente nous a été fournie afin de pouvoir finir notre projet de mémoire.

5.2.2. Intégration du jeu Asteroids sous Visual Studio 2005

Afin de pouvoir exécuter le jeu *Asteroids* sous *Microsoft Visual Studio 2005*, une désignation des références des fichiers d'entête et des bibliothèques que *DirectX 9.1* fournit, est nécessaire:

- Désignation du répertoire qui contient les fichiers d'entête (*.h) *DirectX 9.1* dans la rubrique *MicrosoftVisualStudio2005/Tools/VC++Directories/Include Files.*
- Désignation du répertoire qui contient les bibliothèques (*.lib) *DirectX 9.1* dans la rubrique *MicrosoftVisualStudio2005/Tools/Options/VC++Directories/ Library File.*

5.2.3. AspectC++Debogger et son manuel d'utilisation

Enfin, pour instrumenter le code source d'*Asteroids* sous *Microsoft Visual Studio 2005* et avec les aspects développés (*AspectC++Debogger*) à l'aide d'*AspectC++ Add-In*, il faudra procéder comme suit (voir Figure 5-2 et Figure 5-3) :

- lancer le projet de travail *Asteroids* sous *Microsoft Visual Studio 2005*;
- configurer *AspectC++Add-In* sous le projet *Asteroids*. Deux répertoires s'ajouteront à la solution du projet : *Aspects actifs* et *Aspects inactifs*;

- ajouter les aspects décrits dans le Chapitre 4 au répertoire *aspects actifs*.

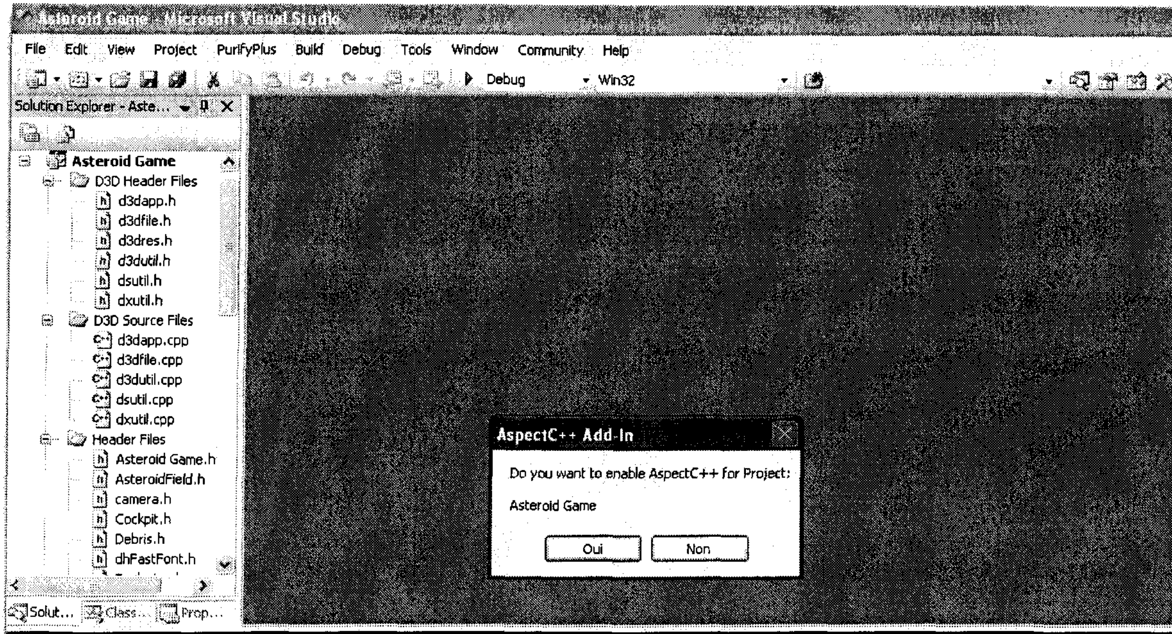


Figure 5-2 : Activer *AspectC++Add-In* sous le projet Asteroids

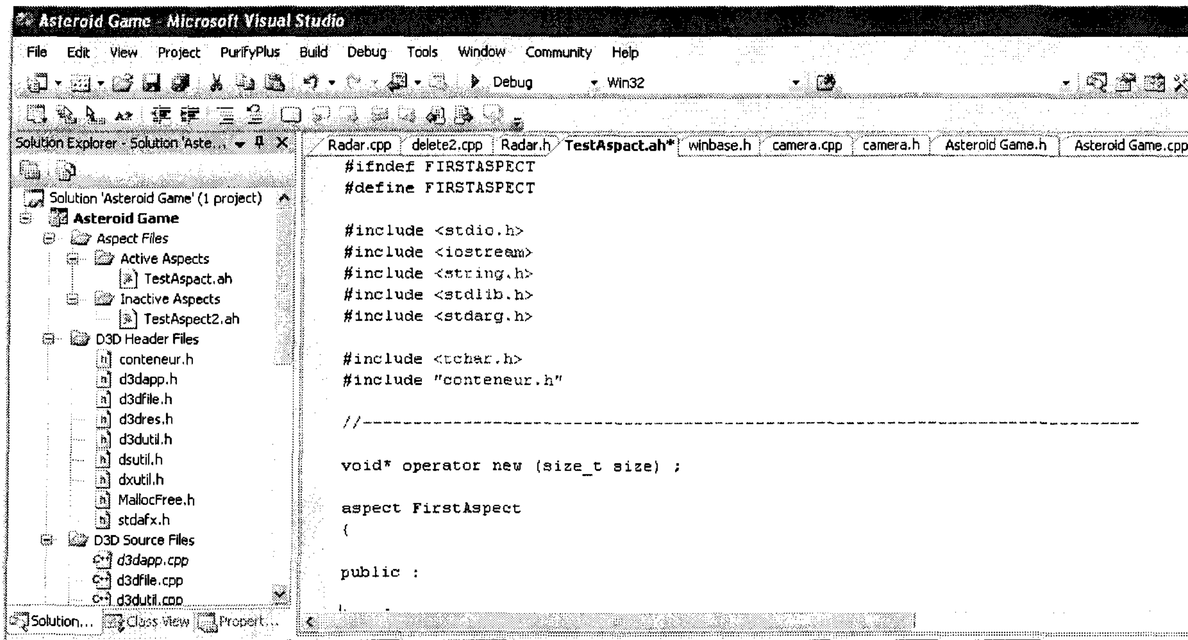


Figure 5-3: Asteroids, Visual Studio 2005 et AspectC++Add-In

Afin de rendre transparent et de faciliter aux programmeurs/utilisateurs l'analyse et le débogage de la mémoire, nous avons mis au point une interface derrière laquelle s'exécute cette étape *d'intégration des principaux composants* (voir figure 5-4).

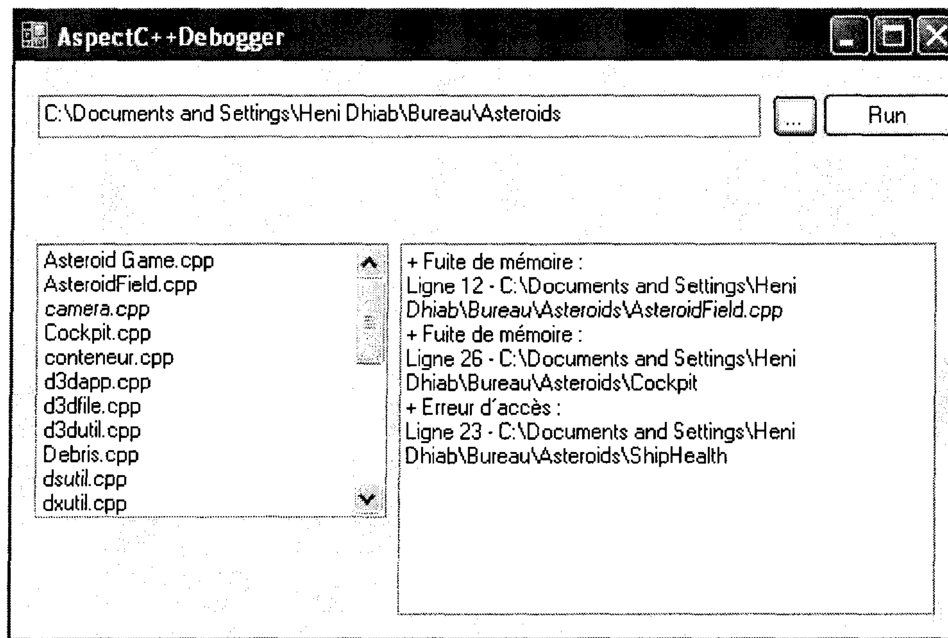


Figure 5-4 : Interface utilisateur de *AspectC++Debogger*

En effet, il suffit au programmeur de donner le chemin du répertoire contenant toutes les classes de l'application à analyser, puis de cliquer sur le bouton *Run. AspectC++Debogger* affichera dans un premier temps les classes ciblées et ensuite les résultats de l'analyse. La nature de toute erreur détectée, le nom du fichier et le numéro de ligne où elle s'est produite sont consultables sur écran.

À ce stade de l'expérience, nous mettons l'accent sur le quatrième critère de comparaison de cette étude de cas : *facilité et transparence d'utilisation* de l'outil. Avec une telle interface le programmeur n'a plus à se soucier d'aucun paramétrage, il suffit d'un seul clic et l'application cliente est exécutée puis analysée dans un même domaine (espace). On notera que le code source n'est en aucun cas modifié durant l'analyse. Cette interface est fortement inspirée de celle de *Purify*, la seule différence est que *Purify* n'affiche pas le détail des classes C++ analysées (voir Section 2.2.2.4 du Chapitre 2).

5.3. Expérience et résultats

Dans cette section nous mettons l'accent sur le premier critère de comparaison de cette étude de cas, l'*habilité de détection et de traitement des erreurs de gestion de mémoire*. Bien que nous jugions important la totalité des critères fixés afin de comparer les deux outils, ce critère reste tout de même le plus important. L'expérience consiste à exécuter l'application *Asteroids* sous *AspectC++Debogger* et *Purify* et nous nous efforcerons de couvrir les fuites de mémoire, les erreurs d'accès et enfin les dépassements de tampon. Notre choix de ce type d'erreurs est argumenté dans le Chapitre 2 de ce mémoire. Les résultats de l'expérience seront résumés dans des tableaux. La disposition de ces tableaux sera décrite dans le paragraphe suivant. Une série de benchmarks illustrant les résultats de notre expérience sont disponibles à l'annexe B.

La première colonne définit l'application analysée. À partir de là, le tableau est divisé en deux parties, chacune est réservée pour un outil. En ce qui concerne la répartition de ces deux principales parties du tableau, elle est identique. La première colonne contient le nombre d'erreurs reportées par chacun des outils. Après une inspection manuelle des erreurs signalées, nous classons ces dernières dans les trois colonnes suivantes :

- Err : Colonne pour les erreurs potentielles ;
- IP : Colonne pour les erreurs nécessitant une intervention du programmeur afin de les valider ;
- II : Colonne pour les erreurs qui sont indécidables à l'inspection manuelle (une série de tests s'impose).

Le temps de contrôle des programmes est donné dans la dernière colonne.

Nous avons jugé adéquat de tester *AspectC++Debogger* avec l'application *DebugTest* analysée par *Purify* dans la Section 2.2.2.3 du Chapitre 2. Le rapport d'erreurs étant déjà disponible, il fallait vérifier qu'*AspectC++Debogger* puisse détecter les mêmes erreurs à son tour. Le test a été réussi, la totalité des erreurs a été atteinte (pour plus de précision, se référer à la section mentionnée au début du paragraphe).

5.3.1. Les fuites de mémoire

Les emplacements mémoires qui sont affectés à des processus peuvent contenir des données spécifiques qui ne devraient pas être divulguées à des utilisateurs non autorisés.

Aussi, un processus devrait s'assurer que tout emplacement mémoire sera libéré et nettoyé une fois qu'il n'est plus d'aucune utilité. Le cas contraire pourrait entraîner des fuites de mémoire et ainsi réduire les performances et la disponibilité du système en fonctionnement. En effet, les systèmes d'exploitation limitent la taille des espaces mémoire et un processus devrait se défendre lui-même contre l'épuisement de ces ressources. Lorsque la limite est atteinte, le processus n'est plus en mesure de s'exécuter et est victime d'un échec de service.
Pour prévenir les fuites de mémoire, *AspectC++Debogger* fait état de chaque espace mémoire alloué dynamiquement et qui devrait être libéré avant la fin du processus. Même si un espace mémoire est dans le doute qu'il soit une fuite de mémoire, notre outil le signale de la sorte et c'est au programmeur que revient la décision finale de le supprimer ou de le laisser.

Tableau 5-1 : Résultat de détection des fuites de mémoire

	Purify					AspectC++Debogger				
	Erreurs reportées	Err	IP	II	Temps écoulé(s)	Erreurs reportées	Err	IP	II	Temps écoulé(s)
DebugTest	2	1	1	0	2,1	2	1	1	0	10
Asteroids	3	2	0	1	3,6	2	2	0	0	13,9

Notre outil rapporte des erreurs pour tous les chemins qui allouent des emplacements mémoire non libérés avant la sortie. Nous illustrons ci-dessous différents exemples.
L'exemple du code de la Figure 5-5 illustre une réelle fuite de mémoire dans la classe *AsteroidField.* Un pointeur de type *DWORD* est alloué et ne sera jamais libéré. *AspectC++Debogger* et *Purify* détectent cette erreur.

```
COLORREF* pcrArrayColor = NULL;
pcrArrayColor = new DWORD[dwWidth * dwHeightDest]; //Allocation dynamique d'un
                                                    //pointeur de type DWORD
pCursorBitmap->LockRect( &lr, NULL, 0 );
pBitmap = (DWORD*)lr.pBits;
if (pcrArrayColor != 0)
{
      fprintf ( stderr , "%s: %s: %s\n", progname , editor ,strerror ( errno ));
      exit (1); //Sortie du programme sans liberer le pointeur pcrArrayColor
}
else exit (0);
```

Figure 5-5 : Fuite de mémoire dans *AsteroidField.cpp*

Le code listé dans la Figure 5-6 est rapporté de la classe *AsteroidGame*. *Purify* signale une erreur qu'on a classifié de type (II). La fonction *DirectInput8Create ()* retourne un pointeur

sur un espace mémoire '*HRESULT*' alloué dynamiquement. En cas d'échec d'allocation, le programme se termine. **Cependant, *Purify* est insensible à ce type de raisonnement, il ne peut déterminer que la fonction *DirectInput8Create ()* prend fin lorsque *hr* est à NULL et rapporte une fausse alerte de fuite de mémoire. Dans le cas d'*AspectC++Debogger*, aucune fuite de mémoire n'est signalée. Les informations tenues concernant le pointeur *hr* et sur lesquelles notre outil se base pour connaitre son état lui évitent ce genre de fausse alerte.**

```
if( FAILED( hr = DirectInput8Create( GetModuleHandle(NULL),
DIRECTINPUT_VERSION,
IID_IDirectInput8, (VOID**)&m_pDI, NULL ) ) )
{
      printf ( stderr ," Process failed .\n");
      exit (2);        // sortie du programme avec hr == NULL
}                      // purify rapporte une fausse fuite de mémoire

// DirectInput8Create alloue un espace mémoire dynamiquement
HRESULT * WINAPI DirectInput8Create(HINSTANCE hinst,DWORD
dwVersion,REFIID riidltf, LPVOID *ppvOut, LPUNKNOWN punkOuter)
{
      HRESULT * hr = new  HRESULT;
      ...
      return hr
}
```

Figure 5-6: Purify signale une fausse alerte d'une fuite de mémoire dans *AsteroidGame.cpp*

5.3.2. Les erreurs d'accès

Les erreurs d'accès auquel un programme pourrait faire face sont diverses et souvent trompeuses. Après une première analyse de l'application *Asteroids*, nous avons trouvé quelques erreurs d'accès qui ont été vérifiées tant par notre outil que par *Purify*. Aucune de ces erreurs n'a signalé le cas d'un *accès légal à une zone mémoire inappropriée*. Vu que nous avons accès au code source de l'application *Asteroids*, nous nous sommes permis d'injecter le cas d'une erreur d'un accès légal à une zone mémoire inappropriée, voir la Figure 5-7.

Tableau 5-2 : Résultat de détection des erreurs d'accès

	Purify					Débogueur de mémoire				
	Erreurs reportées	Err	IP	II	Temps écoulé	Erreurs reportées	Err	IP	II	Temps écoulé
DebugTest	3	2	1	0	3	3	2	1	0	11
Asteroids	5	4	1	0	5	6	4	1	1	13,3

Le tableau dynamique *pMaterials* contient un nombre X d'astéroïdes, avec X> *numMaterials*. Parmi ces astéroïdes, *numMaterials* ont explosé. Tout astéroïde explosé est traité par la classe *explosion* et enregistré dans un fichier à l'aide de la méthode *Spawn* (voir la ligne 8 de la Figure 5-7). Si on essaye de traiter de la même façon l'astéroïde *numMaterials+1* du tableau *pMaterials,* une erreur d'accès légal à une zone mémoire inappropriée fait surface. En effet, la case *pMaterials[numMaterials+1]* existe, mais ne peut être traitée par la classe *explosion* vu que son astéroïde correspondant n'a pas encore explosé.

```
      D3DMATERIAL8*       m_pMaterials;
      //Allocation dynamique du tabelau m_pMaterials
      pMaterials = (D3DXMATERIAL*)pbufMaterials->GetBufferPointer();
      ...
      for( unsigned int i=0; i < numMaterials; i++ )
      {
            if((D3DXCreateTextureFromFile(m_pd3dDevice,
                  pMaterials[i]->explosion(),&ppTextures[i] )))
            //traitement des astéroides explosés par la classe explosion
L8          pMaterials[i]->explosion->Spawn(spaceship->getPosition(),
                                                      20,true);
      }
            ...
      //l'asteroide (numMaterials+1) est traité par la classe
      //explosion sans avoir explosé, on signale une erreur
L10   pMaterials[numMaterials+1]->explosion->
                  Spawn(spaceship->getPosition(),20,true) ;
```

Figure 5-7 : Cas d'une erreur d'un accès légal à une zone mémoire inappropriée

Le langage C++ permet une grande flexibilité lors de la manipulation des pointeurs. Ce type d'avantage constitue une grande limite pour l'outil *Purify* qui a été dans l'incapacité de détecter l'erreur injectée. Par contre, la structuration même d'*AspectC++Debogger* constitue une solution à cette même limite. En effet, grâce aux tests décrits à la Section 4.3.3.2 du Chapitre 4, chaque adresse mémoire est validée avant tout traitement des données qu'elle contient. Que se soit pour un accès, une suppression, une manipulation (le cas de l'exemple de

la Figure 5-7) ou encore une réallocation, l'utilisateur est sûre d'accéder *légalement* à une adresse mémoire qui reste *valide.*

5.3.3. Les dépassements de tampon

Les erreurs de dépassement de tampon sont assez simples à détecter. Dans le cas de l'application *Asteroids*, une seule erreur a été détectée, et ceci, tant par *AspectC++Debogger* que par *Purify*. Il faut dire que ce type d'erreur reste bien entretenu par les débogueurs de mémoire. Si on se réfère à la Section 2.2.2.1 du Chapitre 2, on constatera que pour éliminer les erreurs d'*underflow* et d'*overflow*, *Purify* insère deux zones de garde qu'il a nommé *zones rouges*. Dans le cas de notre outil, on insère un seul espace mémoire supplémentaire afin de détecter les erreurs d'*overflow*. Pour les erreurs d'*underflow*, la signature des informations collectées concernant l'allocation et la *checksum* (somme de vérification) décrites dans la Section 4.3.3.1 du Chapitre 4 s'occupent de les détecter. Un gain d'espace mémoire est suscité. Nous illustrons ci-dessous l'erreur détectée.

Tableau 5-3 : Résultat de détection des dépassements de tampon

	Purify					AspectC++Debogger				
	Erreurs reportées	Err	IP	II	Temps écoulé	Erreurs reportées	Err	IP	II	Temps écoulé
DebugTest	1	0	0	1	3,2	1	0	0	1	11
Asteroids	1	0	0	1	4,2	1	0	0	1	12,3

Dans la classe *d3dfile*, une des classes responsables de la configuration 3D du jeu *Asteroids*, un tableau *m_pTextures* est alloué dynamiquement. *m_pTextures* est un tableau de textures de type *_D3DMATERIAL8* qui contiennent des valeurs de paramétrage dimensionnel (voir la ligne 1 de la figure 5-8). Une erreur de dépassement de tampon est signalée au niveau de l'espace mémoire réservé pour le tableau *m_pTextures* (voir la ligne 10 de la figure 5-8). Ce dépassement est dû au nombre de textures qui est largement supérieur à la taille *m_dwNumMaterials* du tableau, avec *m_dwNumMaterials* = *0L*. N'étant pas expert en la programmation 3D, nous n'avons pas pu cerner à quoi le nombre de textures correspondait. Par contre, nous sommes allés voir à quoi correspondaient les valeurs des textures *_D3DMATERIAL8* du tableau *m_pTextures* et avons constaté que nombreuses d'entre elles étaient erronées.

```
      //structure d'une texture
L1    typedef struct _D3DMATERIAL8 {
          D3DCOLORVALUE    Diffuse;        /* Diffuse color RGBA */
          D3DCOLORVALUE    Ambient;        /* Ambient color RGB */
          D3DCOLORVALUE    Specular;       /* Specular 'shininess' */
          D3DCOLORVALUE    Emissive;       /* Emissive color RGB */
          float            Power;          /* Sharpness if specular
                                                 highlight */
      } D3DMATERIAL8;


      if( pMtrlBuffer && m_dwNumMaterials > 0 )
      {
            //Allocation de mémoire pour le tableau m_pTextures de
            //texture
            D3DXMATERIAL* d3dxMtrls = (D3DXMATERIAL*)pMtrlBuffer-
                                      >GetBufferPointer();
L10         m_pTextures  = new D3DMATERIAL8 [m_dwNumMaterials];
      }
      ...
      if( d3dxMtrls[i].pTextureFilename )
      {
            //Création d'une texture
            TCHAR strTexture[MAX_PATH];
            TCHAR strTextureTemp[MAX_PATH];

            DXUtil_ConvertAnsiStringToGeneric(strTextureTemp,
                                      d3dxMtrls[i].pTextureFilename);
            DXUtil_FindMediaFile( strTexture, strTextureTemp );
            //Affectation de la structure dans le tableau
            //m_pTextures
L21         D3DXCreateTextureFromFile( pd3dDevice, strTexture,
                                             &m_pTextures[i] );
      }
```

Figure 5-8 : Erreur de dépassement de tampon dans *d3dfile.cpp*

Afin de nous assurer que notre raisonnement est juste, nous nous sommes proposés pour résoudre cette erreur de dépassement de tampon et avons modifié le code source de la classe *d3dfile*.

```
      if( d3dxMtrls[i].pTextureFilename )
      {
      //Création d'une texture
            TCHAR strTexture[MAX_PATH];
            TCHAR strTextureTemp[MAX_PATH];
            DXUtil_ConvertAnsiStringToGeneric(strTextureTemp,
                                      d3dxMtrls[i].pTextureFilename);
            DXUtil_FindMediaFile( strTexture, strTextureTemp );
      //Affectaion de la structure dans le tableau m_pTextures
      //si l'affecation échoue la case i du tableau m_pTextures est
      //mise à NULL.
L11         if( FAILED( D3DXCreateTextureFromFile( pd3dDevice,
            strTexture,&m_pTextures[i])))
                  m_pTextures[i] = NULL;
      }
```

Figure 5-9 : Injection d'un test dans *d3dfile.cpp*

Nous avons remplacé l'instruction responsable de l'affection des textures au tableau *m_pTextures* (ligne 21 à la Figure 5-8) par l'instruction écrite à la ligne 11 de la Figure 5-9. En effet, un test a été ajouté de façon que si un problème quelconque se produit lors de cette affectation, la case '*i*' du tableau *m_pTextures* est remise à NULL. Ainsi, au lieu de contenir une valeur erronée, *m_pTextures[i]* peut être sujet d'une nouvelle affectation. Une nouvelle analyse nous a permis de constater que le nombre de textures ayant des valeurs erronées était rendu à zéro, et aucun dépassement de tampon au niveau du tableau *m_pTextures* n'a été signalé.

5.4. Discussion

Dans cette section, nous passons en revue les critères posés au début de ce chapitre et comparons les outils *AspectC++Debogger* et *Purify*. Chaque critère sera discuté dans un paragraphe pour ensuite résumer le tout dans un tableau comparatif (le critère '*facilité et transparence d'utilisation*' ne sera pas abordé vu que nous l'avons traité à la fin de la Section 5.2).

5.4.1. Habilité de détection et de traitement des différents types d'erreurs de gestion de mémoire C++

Nous avons vu que dans certains cas de fuite de mémoire, *AspectC++Debogger* était plus efficace que l'outil *Purify*. Aussi, nous avons confirmé que le cas d'un *accès légal à une zone mémoire inapproprié* présente une limite pour *Purify* et qu'*AspectC++Debogger* a pu en venir à bout. Les erreurs de dépassement de tampon sont couvertes par les deux outils, la seule différence, c'est que *AspectC++Debogger* suscite un gain d'espace mémoire dans sa manière à détecter ce type d'erreur. Pour les systèmes ayant des ressources limitées en espace mémoire, *AspectC++Debogger* pourrait être préférable.

5.4.2. Coexistence d'une gestion implicite et explicite de la mémoire

Même si notre objectif est de fournir une gestion implicite et sécurisée de la mémoire pour le langage C++, il reste tout de même important de conserver et de pouvoir utiliser les méthodes que le langage C++ fourni. Le modèle de conception de notre outil permet une coexistence entre une gestion implicite et explicite de la mémoire. On peut prendre le cas d'une éventuelle fuite de mémoire : notre outil donne la possibilité au programmeur de juger

de l'utilité ou l'inutilité d'un espace mémoire qui n'est pas encore libéré. Il lui suffit de mettre une variable boolienne à *true* et l'espace mémoire est libéré d'une manière sécurisée. D'un autre côté, le programmeur pourra en tout temps récupérer la mémoire explicitement en usant des différentes fonctions dont le langage C++ dispose *(delete/free)*. Si c'est le cas, notre outil assurera que des erreurs d'accès ne se produisent pas, par exemple rejeter une récupération qui a déjà été faite via notre outil. L'outil *Purify* se contente juste d'afficher les erreurs détectées, l'aspect d'une coexistence entre une gestion implicite et une gestion explicite de la mémoire n'est pas traité pas ce dernier.

5.4.3. Ralentissement de l'application cliente et diminution de performances

Les résultats de notre expérience appuient les constatations de Hursley. En effet, une instrumentation par aspect génère un ralentissement de 30% sur l'exécution de programmes usuels [IBM Hursley, 2002]. Même en ayant subi des améliorations, AspectC++™ reste tout de même une nouvelle technologie et nous n'avons pas bu atteindre un gain de temps lors de l'instrumentation de l'application cliente. Tout au contraire, l'outil *Purify* s'est avéré beaucoup plus rapide. Nous espérons qu'avec des versions plus récentes d'AspectC++™, les erreurs imprévisibles seront de moins en moins rencontrées, ce qui procurera plus de rapidité pour les tisseurs d'aspect.

5.4.4. Généricité de l'outil

Il est certes plus ardu de penser « *générique* » lors de la conception d'un logiciel, mais si cela est fait, la maintenabilité et l'évolution s'en trouvent grandement facilitées, laissant le champ libre à des évolutions futures. Dans le cas de notre outil, le terme *généricité* est de taille. La *programmation par aspect*, incarnant la séparation des préoccupations et formant un support méthodologique de modélisation et de programmation pour *AspectC++Debogger*, permet de séparer toutes les méthodes dans des aspects isolés, de les coder, de les maintenir, et de les remplacer tout à fait indépendamment des autres aspects. *AspectC++Debogger* peut être sujet d'une évolution qui ne le pénalisera point.

5.4.5. Stabilité d'AspectC++

5.4.5.1. Problèmes de compilateur

Lors de la liaison du tisseur d'AspectC++™ avec le compilateur C++ certains problèmes ont fait surface. Le premier, qui est forcement dû à l'immaturité d'AspectC++™, est l'incapacité du tisseur à reconnaître certains fichiers entête 3D de *DirectX* (*header files*). Même avec la solution mentionnée dans le README.Win32 [AspectC++, 2008], il reste encore des fichiers qu'on n'a pas pu charger.

5.4.5.2. Templates C++ non reconnus

L'installation d'AspectC++™ est venue avec un ensemble d'exemples. Ces derniers s'avèrent assez simples et sans aucune implémentation de classes *templates* C++. Lors du tissage de notre premier aspect avec des *templates* C++, le problème est survenu. En effet, certaines fonctions membres de la bibliothèque `iostream` (comme `cin`, `cout`, etc.) ne sont pas reconnues lors de l'utilisation des classes *templates* et nous ne pouvons donc pas retourner certaines données ou certains résultats directement par des flux. L'utilisation d'une autre version d'AspectC++™ nous a permis de résoudre certains problèmes comme la reconnaissance des entêtes `iostream`. Par contre, une autre contrainte est survenue qui est l'incapacité d'utiliser des `namespace`. Cependant, l'utilisation du `namespace` demeure un grand problème à résoudre et étions obliger de la supprimer. Nous avons changé `std::string(...)` par `string(...)` pour permettre au tisseur d'analyser le code C++.

AspectC++™ reste tout de même une nouvelle technologie et comme avec toutes les technologies non mûres, l'adoption répandue de l'POA est gênée par un manque d'appui d'outil, et d'éducation répandue.

5.5. Conclusion

Dans ce chapitre, nous avons présenté expérimentalement notre outil de débogage de mémoire, *AspectC++Debogger,* aussi nous l'avons comparé avec l'outil *Purify*. Notre expérience s'est avéré intéressante et avions pu connaitre les limites de notre outil et plus précisément celle de l'approche adoptée pour son implémentation. Bien qu'il ait pu résoudre certaines limites de *Purify*, *AspectC++Debogger* admet encore des faiblesses et a connu des

contraintes qu'il n'a pas pu surmonter. Ci-dessous, un tableau comparatif englobant les résultats de notre expérience.

En se référant toujours à la numérotation des critères énumérés au début de ce chapitre, la première colonne indiquera le numéro du critère à noter.

Tableau 5-4 : Tableau récapitulatif de l'étude de cas de Purify et d'AspectC++Debogger

Critère	*Purify*	*AspectC++Debogger*
1	+	++
2	+/-	+
3	++	--
4	+	+
5	so	++
6	so	--

++ : Bon

+ : Assez bon

+/- : Minimal

- : Mauvais

-- : Absent

So : Sans objet, le critère n'est pas tenu en compt

Conclusion

Le langage C++ est un langage performant, flexible et d'une forte portabilité. Il permet la programmation sous de multiples paradigmes, comme la programmation procédurale, la programmation orientée objet et la programmation générique. Ses caractéristiques en font un langage idéal pour des projets de taille importante. Vu sa facilité d'utilisation et son efficacité, le langage C++ est actuellement le 3éme langage le plus utilisé au monde [Tiobe Software, 2009] [Freshmeat, 2007]. Toutefois, les dispositifs de sécurité sont soit absents, soit mal supportés par la programmation C++; malheureusement, la flexibilité et le contrôle explicite des données que C++ offre au programmeur représentent une importante source de vulnérabilité et d'erreurs critiques. N'étant pas appuyée par une gestion implicite, la gestion de la mémoire en C++ s'est vue toujours comme un exercice difficile, voire parfois périlleux. Cette contrainte pose un défi considérable.

Le problème d'offrir une gestion automatique ou implicite au langage C++ a connu différentes tentatives de résolution. Dans un premier temps, les chercheurs ont opté pour l'implémentation d'un *Garbage Collector*. Malheureusement, cette solution n'a pas été efficace à plus d'un titre. Afin de fournir une solution plus robuste et plus complète, les chercheurs se sont concentrés sur le développement d'un outil de programmation spécialisé dans la gestion de mémoire : le *débogueur de mémoire*. Malgré les efforts fournis dans ce sens, les débogueurs de mémoire C++ les plus performants admettent encore des limites, et dans certains cas, ne peuvent assurer une bonne gestion de la mémoire

Dans ce travail de maîtrise, nous avons proposé un outil complet de gestion implicite de la mémoire pour le langage C++. La *programmation par aspect* a servi de support pour l'implémentation de cet outil et avons pu centraliser la gestion de la mémoire au moyen d'un seul aspect. Nous nous sommes inspirés des limites de débogueurs de mémoire comme *Purify* et *Valgrind* pour réviser certains choix de structuration et de conception de notre outil. Et étant soutenus par la *programmation par aspect*, nous avons pu contourner les limites du modèle de programmation par objet, par exemple, la *dispersion du code*.

Nous avons élaboré un cadre de développement (méthode et outil), pour permettre une analyse dynamique des applications C++. Dans ce cadre, nous avons privilégié la détection des erreurs de *gestion de mémoire dynamique* et avons essayé de répondre aux critères posés dans le premier chapitre. Ayant englobé le tout dans un outil qu'on a surnommé *AspectC++Debogger*, nous avons réalisé une étude de cas afin de comparer ce dernier avec l'outil *Purify*. Cette étude a eu comme objectifs de juger l'efficacité d'*AspectC++Debogger* à détecter et traiter les erreurs de gestion de mémoire, et d'améliorer l'implantation des tisseurs d'AspectC++™ dans le domaine du génie logiciel (cf. Chapitres 5). Nous avons constaté que notre outil été efficace dans la détection des fuites de mémoire, des erreurs d'accès, et des dépassements de tampon. Aussi, nous avons pu conserver la gestion explicite que le langage C++ offre tout en la sécurisant avec la gestion implicite qu'*AspectC++Debogger* procure, une coexistence entre les deux a été atteinte. Enfin, nous avons constaté que l'POA était une technique qui peut automatiser la plupart des tâches quand des nouvelles exigences sont ajoutées. L'avantage de l'POA est son aspect générique et sa capacité à instrumenter un code durant son exécution sans avoir à le modifier. Une interface utilisateur a été fournie afin de permettre un haut niveau de transparence et de facilité d'utilisation au programmeur. Mes contributions dans ce mémoire sont résumées comme suit :

- Nous avons développé un outil et une méthode pour la gestion et la sécurisation de la mémoire pour le langage C++.
- La principale limite de l'outil *Purify*, l'*accès légal à une zone de mémoire inappropriée*, a été résolue. Cette dernière est du à une bonne flexibilité des pointeurs qu'offre le langage C++. Nous avons implémenté une structure qui permet la gestion des pointeurs C++ et qui permet d'éviter ce type d'accès frauduleux (cf. Section 4.3.3.2).
- Une bonne coexistence entre une gestion implicite et explicite de la mémoire a été atteinte. Les deux types de gestion ne sont plus dissociés : le programmeur garde son rôle de décideur tout en étant contrôlé et sécurisé (cf. Section 5.4.2).
- Un avancement considérable est relevé pour les tisseurs d'AspectC++, nous avons réussi à implémenter un premier débogueur de mémoire basé sur la programmation par aspect. Nous proposons un outil de débogage de mémoire C++ adéquat, analysé, développé et testé.

AspectC++Debogger a répondu à tous les critères émis pour son évaluation sauf pour le troisième et le dernier critère (voir Chapitre 5 pour la liste des critères). En effet, une instrumentation par aspect diminue considérablement les performances de l'application analysée (cf. Section 5.4.3). Aussi, nous avons constaté une instabilité du tisseur AspectC++™. Ce dernier n'a pas pu reconnaitre un nombre important de fichier d'entête et de templates C++. Beaucoup d'erreurs de compilation ont été rencontrées (cf. Section 5.4.5). Un tableau récapitulant les avantages et les désavantages d'*AspectC++Debogger* est fourni à la fin du Chapitre 5.

Dans le cadre de ce travail, même si nous n'avons pas résolu toutes les limites que d'autres débogueurs de mémoire admettent, nous avons tout de même réussi à couvrir l'essentiel de nos objectifs. En effet certains points n'ont pas été résolus et pourraient être explorés à travers des recherches futures. Voici quelques-unes de nos suggestions :

- En effet, une instrumentation par aspect a généré un ralentissement considérable de l'application cliente. Même en ayant subi des améliorations, AspectC++™ reste instable et avons rencontré plusieurs cas d'erreur ralentissant les tisseurs d'aspect. Nous comptons sur le travail des chercheurs dans ce domaine, pour qu'AspectC++™ puisse être plus performant et consolider encore plus le langage C++.
- Les subdivisions d'espaces mémoire qu'un programmeur peu alloué sans l'aide de la fonction *malloc* n'est pas traité par notre outil, des recherches dans ce sens ne peuvent qu'élargir la flexibilité du langage C++ (cf. Section 2.2.2.2.b).
- Le cas d'une collaboration entre une gestion de mémoire statique et dynamique n'est pas à écarter. En effet, certains chercheurs pensent pouvoir bénéficier des données de sortie d'une analyse statique afin d'appuyer une éventuelle analyse dynamique (*dunno-point*). Est-il possible de développer et d'enrichir ce concept d'idée avec l'POA?

L'aspect générique que notre outil détient lui permet une extension et un développement de ces futures recherches; Notre travail de recherche est ouvert à toute possibilité d'amélioration.

Enfin, pour conclure et pour divertir le lecteur, nous avons ajouté en annexe des benchmarks résumant les résultats de notre étude de cas (Chapitre 5) et quelques exemples du code source d'*AspectC++Debogger*.

Bibliographie

[Abdullahi et Ringwood, 1998] S. E. Abdullahi and G. A. Ringwood. Garbage collecting the Internet: a survey of distributed garbage collection. *ACM Computing Surveys*, 30(3):330-373, September 1998.

[AspectJ, 2001] The AspectJ Organization, "Aspect-Oriented Programming for Java."http://www.aspectj.org , 2001.

[AspectC++, 2006] http://www.aspectc.org/.

[AspectWerkz, 2005] BEA Systems, http://www.aspectwerkz.codehaus.org/, Chicago, Illinois, USA, 2005.

[Asteroids, 2010] http://asteroids3d.sourceforge.net/

[Baecker, 1970] H. D. Baecker, Implementing the Algol-68 heap. *BIT*, 10(4):405-414, 1970.

[Bartlett, 1989] J. F. Bartlett, Mostly-Copying garbage collection picks up generations and C++. Technical note, DEC Western Research Laboratory, Palo Alto, CA, October 1989.

[Baltus, 2000] J. Baltus, ''La programmation Orientée Aspect et AspectJ : Présentation et Application dans un système Distribué,'' Facultés Universitaires Notre Dame de la Paix, Namur, Belgique.

[Boehm, 1993] H.J. Boeh, Space efficient conservative garbage collection. *Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation*, volume 28(6) of *ACM SIGPLAN Notices*, Albuquerque, NM, ACM Press, June 1993.

[Boehm, 2002] H. Boehm, "Bounding Space Usage of Conservative Garbage Collectors", *Proceeedings of the 2002 ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, , pp. 93-100 Jan. 2002.

[Bray, 1977] O. H. Bray, Data management requirements: The similarity of memory management, database systems, and message processing, Proceedings of the 3rd workshop on Computer architecture : Non-numeric processing, ACM Press New York, NY, USA, Pages: 68 – 76, 1977.

[CiteSeer, 1990] Scientific Literature Digital Library and Search Engine, Purify: Fast Detection of Memory Leaks and Access Errors, 1990.

[CiteSeer, 2008] Scientific Literature Digital Library and Search Engine, Purify: Fast Detection of Memory Leaks and Access Errors, 2008.

[Collins, 1960] G. E. Collins, A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655-657, December 1960.

[Cousot, 2000] Patrick Cousot, Interprétation abstraite. J. F. Technique et science informatiques. Vol. 19- no 1/2000.

[Debbabi et Tlili, 2009] Syrine Tlili et Mourad Debbabi, Type and Effect Discipline for Memory and a Type Safety of C Code, Faculty of Engineering and Computer

Science, Concordia university, 2009.

[Dailey, 1999] Aaron Dailey, Effective C++ memory allocation, EE Times-India, Janvier 1999.

[Deitel, 2010] http://www.deitel.com/ResourceCenters/Programming.

[Detlefs, 1992] D. L. Detlefs, Garbage collection and runtime typing as a C++ library. In *USENIX C++ Conference*, Portland, Oregon, USENIX Association. August 1992.

[Detlefs, 1994] David L. Detlefs, Concurrent garbage collection for C++. In Peter Lee, editor, *Topics in Advanced Language Implementation.* MIT Press, 1994.

[DirectX, 2010] http://www.microsoft.com/games/en-US/aboutGFW/pages/directx.aspx

[Edelson et Pohl, 1991] D. R. Edelson and I. Pohl, A copying collector for C++. In *Usenix C++ Conference Proceedings*, pages 85-102. USENIX Association, 1991.

[Ellis et Stroustrup, 1990] M. A. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[Ellis, 1993] J. R. Ellis, Put up or shut up. In Moss *et al. OOPSLA/ECOOP '93 Workshop on Garbage Collection in Object-Oriented Systems*, October 1993.

[FreeBSD, 2008] The FreeBSD Foundation. Freebsd the power to serve. http://www.freebsd.org/.

[Freshmeat, 2007] www.freshmeat.net, 2007.

[Ferreira, 1991] Paulo Ferreira. Garbage collection in C++. In *Workshop on Extensions to C++*, Lisbon, June 1991.

[Gati , 2004] Gati Michel, ''Analyzing Run-time Component Memory Consumption with Aspect-Oriented Techniques '', Information Technology European Advancement, Mars 2004.

[Harrison et Ossher, 1993] W. Harrison, and H. Ossher, ''Subject-oriented programming: a critique of pure objects,'' in Proc. of OOPSLA'93, pp. 411-428, 1993.

[Hagen et Zhang, 2002] Mike Hagen, Zhen Zhang, Evaluation of Rational Purify, Master of Software Engineering Program School of Computer Science Carnegie Mellon University

[Hirzel et al., 2002] M. Hirzel , J. Henkel , A. Diwan , M. Hind, Understanding the connectivity of heap objects, Proceedings of the third international symposium on Memory management, Berlin, Germany, June 20-21, 2002.

[IBM Hursley, 2002] Hursley, An Aspect Oriented Performance Analysis Environment, IBM Corporation, 2002.

[IBM Rational Purify, 2008] IBM Rational Purify, http://en.wikipedia.org/wiki/IBM_Rational_Purify#Overview, Avril 2008.

[IBM Software, 2003] Develop Fast, Reliable Code with IBM Rational PurifyPlus, 2003.

[ISO/IEC, 2003] ISO. ISO/IEC 14882:2003: Programming languages: C++. 2003. http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail CSNUMBER=38110.

[JAC, 2006] http://www.jac.objectweb.org/ , AOPSYS compagnie, Paris, France, 2006.

[Jones et Lins, 1996] R. E. Jones and R. Lins, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* Wiley, 1996.

[Kicsales et al., 1997] G. Kicsales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. Technical Report SPL97-008 P9710042, XEROX PARC, February 1997.

[Kernighan et Ritchie, 1988] B.W. Kernighan et D.M. Ritchie, The C Programming Language (Ansi C), Edition Prentice Hall, 1988.

[Levanoni et Petrank, 2001] Y. Levanoni et E. Petrank, An on-the-fly Reference Counting Garbage Collector for Java, Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications, Volume 36 Issue 11 October 2001.

[Loriant, 2007] N.Loriant,''Évolution dynamique des systèmes d'exploitation, une approche par la programmation par aspect'', l'École Nationale Supérieure des Techniques Industrielles et des Mines de Nantes, décembre 2007.

[Lesiecki, 2002] http://www128.ibm.com/developerworks/library/j- aspectj/, 2002.

[Linux, 2009] Linux Online Inc. Linux Online! http://www.linux.org/.

[Marguerie, 2009] Fabrice Marguerie, How to detect and avoid memory and resources leaks in .NET applications, September 2009.

[Meyer, 1988] Bertrand Meyer, *Object-oriented Software Construction.* Prentice-Hall, 1988.

[Metropolis *et al.*, 1980] N. Metropolis, J. Howlett, and Gian-Carlo Rota, editors. *A History of Computing in the Twentieth Century.* Academic Press, 1980.

[Meyer, 1988] Bertrand Meyer, *Object-oriented Software Construction.* Prentice-Hall, 1988.

[Mcheick, 2006] H. Mcheick, ''Distributions d'objets avec les techniques de développement orientées aspect,'' dans une thèse présentée à la Faculté des Etudes Supérieurs, 2006.

[Mili et al., 1999] H. Mili, J. Dargham, A. Mili, and R. Godin. "View Programming: Towards a Framework for Decentralized Development and Execution of OO Programs," Proc. of TOOLS USA '99, Aug. 1-5, 1999, Prentice-Hall, pp. 211-221, 1999.

[Nethercote et Seward, 2003] Nethercote and J.Seward, Valgrind: A program supervision framework. In Proceedings of RV'03, Boulder, Colorado, USA, July 2003.

[Nethercote et Seward, 2005] N. Nethercote and J.Seward, Using Valgrind to detect undefined errors with bit-precision, Proceedings of the USENIX Annual Technical

Conference 2005.

[Purify User's Guide, 1999] IBM Rational Purify, http://www.rational.com , 1999.

[Purify Overview, 1999] http://www.agsrhichome.bnl.gov/Controls/doc/purify/purify_info.html.

[Quick Reference, 2006] AspectC++ Quick Refernce, Version 1.11, 15 mars, 2006

[Reenskaugh et al., 1995] T. Reenskaugh et al., in Working with Objects, Prentice-Hall, 1995.

[Salagnac, 2004] Salagnac Guillaume, ''Gestion automatique de la mémoire dynamique pour des programmes Java temps-réel embarqués'', Verimag et Unité Mixte de Recherche, 2004.

[Spinczyk O. et al., 2002] Olaf Spinczyk, Andreas Gal et Wolfgang Schroder-Preikschat, AspectC++: An Aspect-Oriented Extension ro the C++ Programming Language", Australian Computer Society, Inc. Proceedings of the 40th International Conference on Technology.

[Sioud, 2006] A. Sioud, ''Gestion de cycle de vie des objets par aspects pour C++,'' dans un mémoire présenté à l'Université de Québec à Chicoutimi, juin 2006.

[Stroustrup, 2004] B. Stroustrup, The C++ Programming Language. Addison-Wesley Third Edition and Special Edition, 2004.

[Spinczyk et al., 2005] Olaf Spinczyk, Daniel Lohmann et Matthias Urban, ''AspectC++ - extension de la programmation orientée aspect pour C++ '', Software Developer's Journal 6/2005.

[Tarr *et al.*, 1999] P. Tarr, H. Ossher, W. Harrison, and S.M. Sutton, Jr., N Degrees of Separation: Multi-Dimensional Separation of Concerns. ICSE-21, pp. 107-119, 1999.

[Tiobe Software, 2009] http://www.tiobe.com/ , classement des langages les plus utilisés.

[Tlili, 2009] Syrine Tlili, Automatic Detection of Safety and Security Vulnerabilities in Open Source Software, A thesis in The Department of Electrical and Computer Engineering, Concordia University.

[Vassev et Paquet, 2006] Emil Vassev et Joey Paquet, Aaron Dailey, Aspects of Memory Management in Java and C++, Department of Computer Science and Software Engineering Concordia University.

[Valgrind, 2008] Valgrind home page, http://valgrind.org/, June 2008.

[Vasudevan, 2000] Alavoor vasudevan, Comment programmer en C++, v16.0, Août 2000.

[Valgrind Manual, 2008] Complete manual of Valgrind, http://www.valgrind.org/info/about.html.

[Vaysse, 2005] G.Vaysse, http://afpac.gforge.inria.fr/docs/taco/utilisateur.pdf, 2005.

[Paulson et Succi, 2004] James W. Paulson, Giancarlo Succi, and Armin Eberlein. An empirical study of open-source and closed-source software products. IEEE Trans. Softw. Eng., 2004

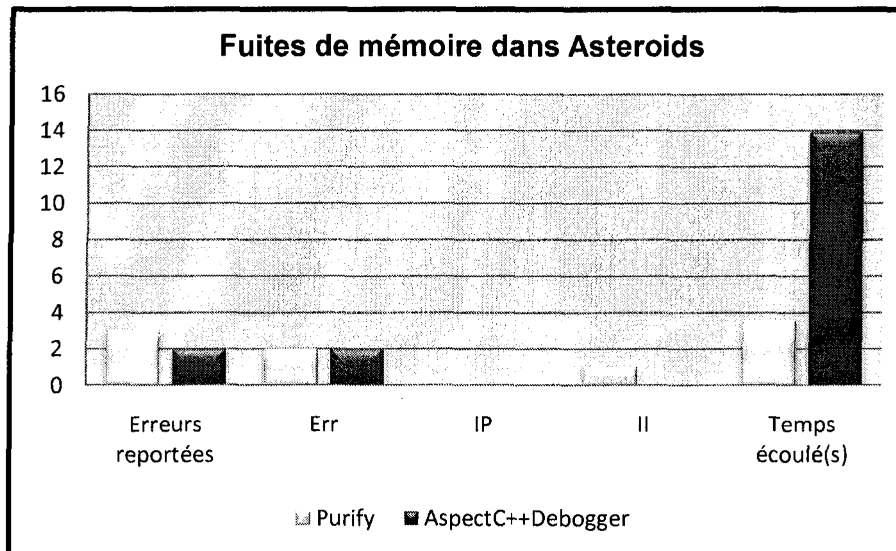
[pure-systems, 2010] http://www.pure-systems.com/

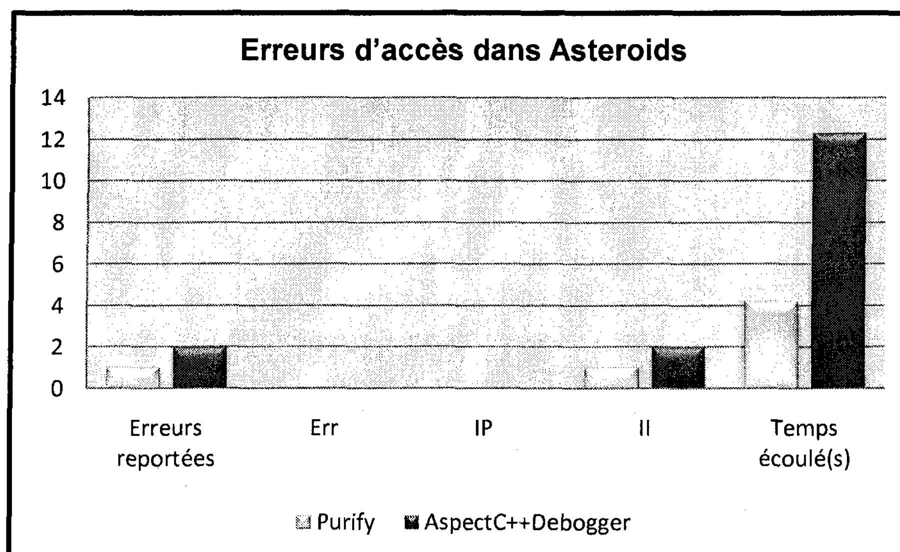
[Robillard et Murphy, 2001] Martin P. Robillard et Gail C. Murphy, "Analysing Concerns Using Class
Member Dependencies," ICSE 2001 Workshop on Advanced Separation of Concerns in Software Engineering, 2001.

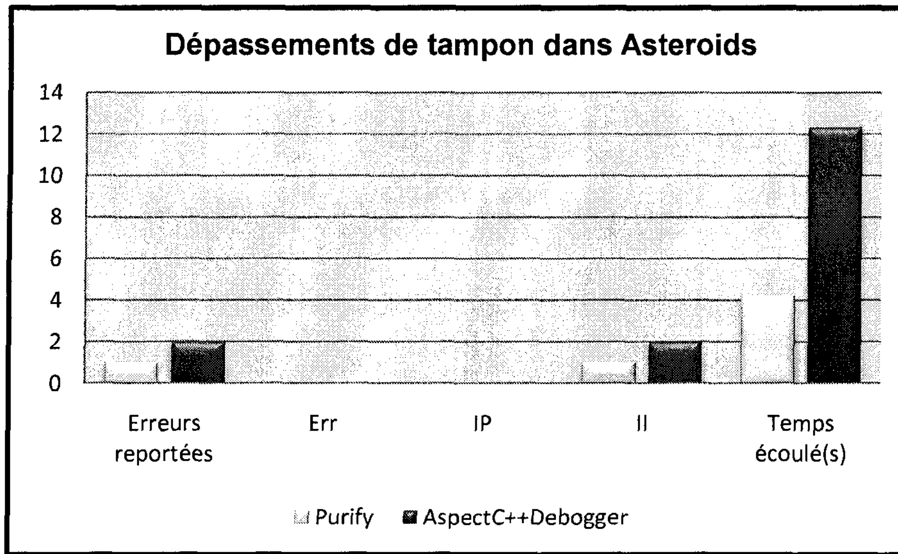
Annexe

A. Benchmarks des résultats expérimentaux

Cette section de l'annexe contient trois benchmarks comparant les résultats d'analyse de l'application *Asteroids* sous les deux outils *AspectC++Debogger* et *Purify*. Le Chapitre 5 utilise ces benchmarks pour compléter l'expérimentation.







B. Code source de l'application *AspectC++Debugger*

Cette section de l'annexe contient le code source des principaux modules de l'application *Asteroids*. Certaines parties du code ont été éliminées pour aider à la lecture.

Conteneur.h

```
#ifndef CONTENEUR_H
#define CONTENEUR_H
//----------------------------------------------------------------------

#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>

//----------------------------------------------------------------------
//---------------------------Définition des structures------------------
//----------------------------------------------------------------------

typedef struct InfoMalloc
  {
    unsigned long nbr_magique;       // Nombre magique
    bool est_alloue;                 // Pointeur alloué ou libéré ?
    size_t taille;           // Taille allouée (sans ces informations)
    int nom_fich;            // Nom du fichier source
    unsigned long num_ligne;            // Ligne du fichier
    struct PtrChaineMalloc *ptr_liste; // Pointeur dans la liste
    unsigned long somme_info;        // Somme des informations (checksum)
  } ;

//----------------------------------------------------------------------
```

```
// Conteneur : liste des pointeurs allouées
// Un pointeur chaîné
typedef struct PtrChaineMalloc
  {
    struct PtrChaineMalloc *precedent;     // Elément suivant
    struct PtrChaineMalloc *suivant;       // Elément suivant
    unsigned long nbr_magique;             // Nombre magique
    InfoMalloc *ptr_info;                  // Pointeur à libérer
  };

//-------------------------------------------------------------------------
typedef struct CM
  {
    struct PtrChaineMalloc *debut;   // Pointeur vers le premier élément
    struct PtrChaineMalloc *fin;     // Pointeur vers le dernier élément
 };
//-------------------------------------------------------------------------
//--------------------Définition de la classe Conteneur-----------------
//-------------------------------------------------------------------------

class Conteneur{
public :
       CM ConteneurMalloc;
       unsigned int  GlobalUsedMemory;
       unsigned int MemoryStillAllocated;
       unsigned int  PointerNumber;
public :

       Conteneur :: Conteneur( ){
             ConteneurMalloc.debut=NULL;
             ConteneurMalloc.fin  =NULL;
             GlobalUsedMemory=0;
             MemoryStillAllocated=0;
             PointerNumber=0;

       }
//-------------------------------------------------------------------------
//--- Tests anti-overflow (écriture en dehors de la zone allouée --------
//-------------------------------------------------------------------------

//Prépare la zone anti-overflow d'un pointeur (ptr pointe sur les infos
//de débogage InfoMalloc)
      void Malloc_PrepareOverflow (InfoMalloc *info);

//Vérifie la tampon anti-overflow : renvoie true s'il n'a pas été
//touché,false sinon (ptr pointe sur les informations de débogage
//InfoMalloc)
      bool Malloc_TamponOverflowIntact (const InfoMalloc *info);

//-------------------------------------------------------------------------
//--- Conteneur Malloc : liste des pointeurs alloués --------------------
//-------------------------------------------------------------------------


// Affiche un message d'erreur, puis quitte le programme
      void  ERREUR (char *message, ...);
// Calcule la somme de vérification (checksum) des données d'unpointeur
```

```
      unsigned long  MallocCalculeSomme (const InfoMalloc *info);

// Fonction qui écrit les informations d'un pointeur et modifie son
//adresse pour qu'il pointe sur les données
      InfoMalloc *MallocPreparePtr (InfoMalloc *info, size_t size,
                        const int nom_fich, unsigned long num_ligne);

// Ajoute un pointeur au conteneur malloc
      void ConteneurMalloc_Ajoute (InfoMalloc* info);

// Vérifie les informations sur un pointeur
      void VerifieInfoMalloc (InfoMalloc* info);

// Affiche le contenu du conteneur malloc (libère les pointeurs si
// libere_mem=true)
      void  ConteneurMalloc_Affiche (bool libere_mem);

// Sort un pointeur du conteneur malloc
      void ConteneurMalloc_Remplace (InfoMalloc *info, InfoMalloc
*nv_info);

// Sort un pointeur du conteneur malloc
      void ConteneurMalloc_Efface (struct PtrChaineMalloc* elem);

//free sécurisé : utilisé pour le conteneur
//void FreeSecurise (const void *ptr,const char* nom_fich, const
//unsigned long num_ligne);

// Calcule la taille réelle d'un bloc alloué
      size_t  MallocCalculeTaille (size_t size);
      void affiche_Info(InfoMalloc *info);
      void ConteneurMalloc_Ecrire ();
      int Return_taille();
      };


#endif
```

NewFree.h

```
#ifndef MALLOCFREE_H
#define MALLOCFREE_H

#include <stdlib.h>
#include "conteneur.h"

class Malloc{
public :

      static Conteneur C ;

public  :

      Malloc();

// Allocation de mémoire sécurisée (malloc) : test si malloc renvoie NULL
```

```
      void* MallocSecurise (void * PrincipalP, size_t size, const int
nom_fich, const unsigned long num_ligne);

//free sécurisé : utilisé pour le conteneur
      void FreeSecurise ( void *ptr,const int nom_fich, const unsigned
long num_ligne);

};
#endif
```

Chrono.h

```
//-----------------------------------------------------------------------
// Classe Chronomètre:permet de mesurer le temps CPU du processus courant
// Le temps considéré est le UserTime du processus, soit le temps pendant
// lequel le système exécute du code appartenant au processus uniquement.
//-----------------------------------------------------------------------
#ifndef chronoH
#define chronoH
#include <stdio.h>
#include <windows.h>
//-----------------------------------------------------------------------
class Chrono
{
// Handle du processus courant
  HANDLE ihP;
// Temps CPU lors du déclenchement
  unsigned int startTime;
public :
// mémoire du chrono
  unsigned int m;
// Constructeur
  Chrono() {reset();}
// mise à zéro de la mémoire
  void reset() ;
// Lance le chronomètre
  void iGo()
    {
    FILETIME ilpCreationTime;
    FILETIME ilpExitTime;
    FILETIME ilpKernelTime;
    FILETIME ilpUserTime;
    SYSTEMTIME stUser;
    ihP=GetCurrentProcess();

GetProcessTimes(ihP,&ilpCreationTime,&ilpExitTime,&ilpKernelTime,&ilpUser
Time);
    FileTimeToSystemTime(&ilpUserTime,&stUser);
    startTime=(stUser.wMilliseconds +1000*(stUser.wSecond
           +60*(stUser.wMinute+60*(stUser.wHour+24*(stUser.wDay-1)))));
  }
// Renvoie le temps CPU du processus, en millisecondes,
// écoulé depuis le lancement du chrono
  unsigned int iStop()
    {
    FILETIME ilpCreationTime;
    FILETIME ilpExitTime;
```

```
     FILETIME ilpKernelTime;
     FILETIME ilpUserTime;
     SYSTEMTIME stUser;

GetProcessTimes(ihP,&ilpCreationTime,&ilpExitTime,&ilpKernelTime,&ilpUser
Time);
     FileTimeToSystemTime(&ilpUserTime,&stUser);
     return stUser.wMilliseconds +1000*(stUser.wSecond
       +60*(stUser.wMinute+60*(stUser.wHour+24*(stUser.wDay-1))))
       -startTime;
     }
// Ajoute le temps CPU du processus, en millisecondes,
// écoulé depuis le lancement du chrono à la mémoire
   void iStopAndAdd()
     {
     m+=iStop();
     }
};
//-------------------------------------------------------------------------
#endif
```

NewSecurise.ah

```
#ifndef FIRSTASPECT
#define FIRSTASPECT

#include <stdio.h>
#include <iostream>
#include <string.h>
#include <stdlib.h>
#include <stdarg.h>

//securise la gestion de memoire afin de s'assurer qu'aucune autre
//gestion n'est en place (eviter les conflits) *globalement*
void* operator new (size_t size)
            {
                  printf ("\n****************************************");
                  printf ("%s \n ligne : %lu - ",__FILE__, __LINE__);
                  printf("\n changelment de l'opérateur new");
                  return malloc(size);
            }

void *operator new[] (size_t s)
            {
                  printf
("\n****************/////*********************");
                  printf ("%s \n ligne : %lu - ",__FILE__, __LINE__);
                  printf("\n changement de l'opérateur new[]..");
                  return malloc (s);
            }
//-------------------------------------------------------------------------
 aspect NewSecuriseAspect
      {
      void * ptr;
      int d;
      FILE *F3;

public :
```

```
      advice classes("%"):
//securise la gestion de memoire afin de s'assurer qu'aucune autre
//gestion n'est en place (eviter les conflits)
      slice class newdelete
      {
      public:
//Assure que loperateur new n'a pas ete redefinit sinon le garbage
//collection ne peu fonctionner
            void* operator new (size_t size)
            {
                  printf ("\n*****************************************");
                  printf ("%s \n ligne : %lu - ",__FILE__, __LINE__);
                  printf("\n changelment de l'opérateur new");
                  return malloc(size);
            }

            void *operator new[] (size_t s)
            {
                  printf   ("\n************/////*****************");
                  printf ("%s \n ligne : %lu - ",__FILE__, __LINE__);
                  printf("\n changement de l'opérateur new[]..");
                  return malloc (s);
            }

      };

  advice execution ("% ...::operator new(...)" ||
                              "% ...::operator new[](...)") : before () {
    printf ("executing heap operation \"%s\"\n", JoinPoint::signature
());
    printf ("  tjp->that() is %s (should be 0)\n", (tjp->that () ? "not
0" : "0"));
  }
};
#endif //NewSecurise aspect
```

ChronoAspect.ah

```
#ifndef TESTASPECT
#define TESTASPECT

#include <stdio.h>
#include <iostream>
#include <string.h>
#include <stdlib.h>
#include <stdarg.h>

#include "chrono.h"

//----------------------------------------------------------------------
----

aspect ChronoAspect
{
      Chrono h;

public :
```

```
        advice execution ("% main()") : before()
        //informations a la fin de l'application
        {
        //Demarrage du chrono
        h.iGo();
        }
        pointcut Mat()  = execution("% ...::%genererMatrice(...)")&&
                                                ! within("conteneur");
        advice Mat(): after ()
        {
        //detection de la fonction principale CHRONO MAT
        h.iStopAndAdd();
        printf("\n temps ecoulee =  %d\n", h.m);
        FILE * F,* E;
                    F=fopen("c:\ApplicationRunTime.txt","wt");
                    fprintf (F," %d ",h.m);
                    fclose(F);
        }
};
#endif //ChronoAspect
```