Avdelning för datavetenskap

Reza Javanbakhti
Jimmy Pesola

Statisk detektering av minneshanteringsfel i C/C++

Static detection of memory management errors in C/C++

Examensarbete (10p)
Dataingenjörsprogrammet

Datum:	2006-06-07
Handledare:	Robin Staxhammar
Examinator:	Stefan Lindskog
Löpnummer:	C2006:13

Karlstads universitet 651 88 Karlstad
Tfn 054-700 10 00 Fax 054-700 14 60
Information@kau.se www.kau.se

Statisk detektering av minneshanteringsfel i C/C++

Reza Javanbakhti

Jimmy Pesola

© 2006 Reza Javanbakhti, Jimmy Pesola och Karlstads universitet

Denna rapport är skriven som en del av det arbete som krävs för att erhålla en kandidatexamen i datavetenskap. Allt material i denna rapport, vilket inte är vårt eget, har blivit tydligt identifierat och inget material är inkluderat som tidigare använts för erhållande av annan examen.

Reza Javanbakhti

Jimmy Pesola

Godkänd, 2006-06-07

Handledare: Robin Staxhammar

Examinator: Stefan Lindskog

Sammanfattning

Det här examensarbetet är baserat på idéer ur ett uppdrag från företaget Saab Aerotech men är ett eget arbete.

Målet var att undersöka om det finns behov av ett verktyg som statiskt kan detektera dynamiska minneshanteringsproblem, som till exempel minnesläckage, i applikationer skrivna i C/C++. På grund av att minneshanteringsfel i C/C++ länge har varit ett känt problem undersökte vi detta och de befintliga lösningarna till det.

Vi fann två metoder till lösningar som de flesta verktyg använde sig av; statisk och dynamisk detektering. De flesta verktyg löste problemet genom att dynamiskt detektera minnesläckor och andra brister som till exempel buffer overflows. Ett verktyg löste dock problemet genom att statisk detektera minneshanteringsfel i källkoden för applikationerna. Eftersom alla befintliga lösningar har någon form av ineffektivitet så har vi undersökt möjligheten att utveckla ett mer effektivt verktyg. Vi har kommit fram till att denna möjlighet finns men det kräver enormt mycket tid och arbete att göra ett komplett verktyg som detekterar minneshanteringsfel statiskt.

Vår prototyp detekterar dynamiska minneshanteringsproblem i källkoden statiskt. Vi har använt oss av hjälpverktygen Flex och Bison för att utveckla vår prototyp av verktyget. Prototypen kan analysera källkod skriven i programspråken C och C++ och klarar att detektera minnesläckage, felaktiga avallokeringar av minne, dangling pointers, samt läsning från och skrivning till ogiltiga minnesområden. På grund av tidsbrist har vi i nuläget inte implementerat något stöd för klasser och objekt i prototypen.

Static detection of memory management errors in C/C++

Abstract

This bachelor's project is our own project, but it is based on ideas from an assignment from the Saab Aerotech company.

The goal was to investigate if there is a need for a tool that statically can detect dynamic memory management errors, such as memory leaks, in applications written in C/C++. Since the problem of memory management errors in the C/C++ languages has been known for a long time, we decided to investigate this and the existing solutions.

We found that most tools used two methods as solutions; static and dynamic detection. Most of these tools solve the problem by dynamically detecting memory leaks and other deficiencies such as buffer overflows. However, one of these tools used static detection of these deficiencies by scanning the source code of the applications. Since all the existing solutions have some kind of inefficiency, we have investigated the possibility to develop a more efficient tool. We concluded that this is possible but it will take a lot of time and effort to implement a complete tool that statically detects memory management errors.

Our prototype statically detects dynamic memory management problems in the source code. We have used the tools Flex and Bison to develop our prototype of a static detection tool. The prototype analyzes source code written in the programming languages C and C++ and is capable of detecting memory leaks, invalid deallocations of memory, dangling pointers and reading from and writing to invalid memory areas. Currently, due to lack of time, we have not implemented any support for classes and objects in the prototype.

Innehållsförteckning

1 Inledning 1
1.1 Bakgrund 1
1.2 Mål för projektet 2
2 Analys av projektet 3
2.1 Befintliga verktyg 3
2.1.1 Splint 3
2.1.2 Purify 4
2.1.3 MemWatch 4
2.1.4 Visual Leak Detector 5
2.1.5 YAMD 6
2.1.6 MemLeakCheck 7
2.1.7 CMemLeak 8
2.1.8 Sammanställning av de befintliga verktygen 9
2.2 Användargränssnitt 9
2.3 Sammanfattning 11
3 Kravspecifikation 13
3.1 Användbarhetskrav 13
3.1.1 Start av verktyget utan parametrar 13
3.1.2 Start av verktyget med en eller flera källkodsfiler som parametrar 13
3.1.3 Verktygets beteende efter att alla giltiga källkodsfiler har angetts 13
3.2 Funktionella krav 14
4 Konstruktionslösning 15
4.1 Konstruktionsproblem 15
4.2 Alternativa lösningar 18
4.3 Vår lösning 19
4.4 Antaganden i vår konstruktionslösning 20
4.5 Begränsningar 21

5 Implementation ... 23
5.1 Utvecklingsmiljö ... 23
5.2 Alternativa verktyg ... 23
5.3 Flex & Bison, Lex & Yacc ... 25
5.4 Generell uppbyggnad av prototypen ... 31
5.5 Detaljerad beskrivning av prototypen ... 31
5.6 Begränsningar i prototypen ... 37
5.7 Begränsningar i Flex och Bison ... 40
6 Tester ... 47
7 Resultat och rekommendationer ... 59
8 Summering av projektet ... 61
Referenser ... 63
Bilagor ... 65

Figurförteckning

Figur 4-1 Illustrering av olika möjliga exekveringsvägar.. 17

Figur 5-1 Illustrering av interaktionen mellan lexern och de tre parsrarna............. 42

Tabellförteckning

Tabell 2-1 Sammanställning av de befintliga verktygen. 9

1 Inledning

1.1 Bakgrund

Explicit dynamisk minnesallokering har varit en av orsakerna till buggar i språk så som C och C++. Garbage Collection (skräpsamling) togs fram som en lösning till komplexiteten av dynamisk minneshantering. Men att använda skräpsamlare innebär en nackdel vilken är dess overhead som kan påverka programmets prestanda [20]. Applikationer som använder skräpsamlare är mycket svåra att förutsäga exekveringshastigheten på vid en given tidpunkt under applikationens exekveringstid. Applikationer kan stanna upp vid en obestämd tidpunkt under exekvering för att frigöra allokerat minne som inte längre används. Därför kan det vara olämpligt att använda skräpsamlare i realtidsapplikationer. För applikationer med höga prestandakrav och som är tidskritiska i sin exekvering är det viktigt att välja ett effektivt sätt att hantera allokering och avallokering av minne dynamiskt. Därför skulle explicit minnesallokering kunna vara ett bra alternativ för sådana applikationer som till exempel realtidsapplikationer. Men användning av explicit minnesallokering medför också risker eftersom det är lätt för programmeraren att sköta allokering, referering och avallokering av minnet på ett felaktigt sätt. Vi har undersökt om det finns verktyg som detekterar minneshanteringsbuggar och de flesta av dem som vi hittade detekterar dessa buggar dynamiskt, det vill säga under körning. Vi hittade dock ett verktyg, Splint [3, 5] (Secure Programming Lint / SPecifications Lint), som statiskt detekterar dessa fel. Splint är en utvecklad version av LCLint [4] (Larch C Lint) och kan detektera bland annat minneshanteringsfel genom att analysera källkod. Splint fungerar enbart för språket C som sin föregångare LCLint. Vi vill därför försöka utveckla ett verktyg som statiskt detekterar minnesläckor för kod skriven i C/C++.

1.2 Mål för projektet

Vårt mål är att undersöka om det finns ett behov av ett verktyg som statiskt detekterar minneshanteringsfel i källkod skriven i C/C++. Om det skulle visa sig att det finns behov kommer vi att utveckla en prototyp av ett sådant verktyg. De minneshanteringsfel som prototypen ska detektera är minnesläckage, felaktiga avallokeringar av minne, dangling pointers, samt läsning från och skrivning till ogiltiga minnesområden. Vi väljer detta språk av anledningen att all dynamisk allokering och avallokering av minne måste göras explicit. Idén är att göra utvecklingen av applikationer säkrare i C/C++, det vill säga att applikationerna ska ha så få minneshanteringsbuggar som möjligt redan innan kompilering. Motiveringen är att språk som C/C++ fortfarande används i stor utsträckning för bland annat utveckling av applikationer som kräver förutsägbar och högsta möjliga prestanda. Dessa språk är prestandamässigt kraftfulla och har stöd för hårdvarunära programmering, vilket medför att det är lättare att programmera på en atomär nivå. Om dessa dynamiska minneshanteringsfel detekteras, lokaliseras och åtgärdas innan kompilering, resulterar det i att man slipper leta efter felen i testningsfasen.

2 Analys av projektet

Detta kapitel innehåller en analys av projektets delar, vilka omfattar en undersökning av befintliga verktyg och användargränssnitt. Befintliga verktyg måste undersökas för att se om det finns möjlighet att utveckla en prototyp av ett verktyg som mer effektivt kan upptäcka brister i källkod. Då handlar det framförallt om att testa deras förmåga att upptäcka minneshanteringsfel. En analys kommer att göras för vilket slags användargränssnitt som prototypen kan ha. Alternativen är antingen ett terminalfönstergränssnitt eller ett grafiskt användargränssnitt.

2.1 Befintliga verktyg

I detta avsnitt analyseras några av de befintliga verktygen som finns att köpa eller ladda ner gratis. De verktyg som analyserats är: Splint [3, 5] (jmf Lclint [4], Lint [13]), IBM Rational Purify [10], MemWatch 2.71 [12], Visual Leak Detector [14], YAMD (Yet Another Malloc Debugger) [2], MemLeakCheck [21] och CMemLeak [16]. De flesta av dessa verktyg är Open Source vilket innebär att all källkod till verktygen är öppen att läsa eller göra ändringar i.

2.1.1 Splint

Splint [3, 5] är ett verktyg som används för att hitta brister i källkod som kompilatorerna inte haft förmågan att upptäcka. Splint utvecklades från föregångaren Lint [13] som kunde hitta brister i källkod så som oanvända deklarationer, typinkonsistenser, oåtkomlig kod, användning före definition, sannolika oändliga loopar och villkorsstyrda fall som aldrig inträffar, returvärden som ignoreras vid funktionsanrop och exekveringsvägar utan retur. Lint kunde också upptäcka inkonsistens mellan programmoduler och kompatibilitet mellan olika C-kompilatorer. De flesta av dagens C-kompilatorer har inbyggd detektering

av de flesta fel som nämns här. Lint utvecklades för Bell Labs sjunde version av operativsystemet UNIX och Lint var en del av C-kompilatorn PCC (Portable C Compiler). Splint står för Secure Programming Lint och har utökats med detektering av dagens IT-säkerhetsinriktade sårbarheter. Splint är intressant genom dess förmåga att kunna detektera minnesläckor statiskt. Fördelen med Splint är att det kan upptäcka en mycket bred omfattning fel i källkoden. Den återger också var felet har inträffat genom att ange vilken rad och kolumn som felet inträffade i. Detektionsfall där användardefinierade datatyper används har lösts genom att använda annotationer. Dessa annotationer ger extra information om hur uttryck ska tolkas. En annan fördel är att verktyget utför statisk detektering, det vill säga en analys av källkod. Det innebär att applikationens prestanda inte påverkas på något sätt av analysen. Den största nackdelen med Splint är dess rapportering av resultat. Rapporten från Splint ser mycket rörig ut och har ingen bra struktur. Med tanke på omfattningen av antalet fel som kan detekteras borde rapporten ha en tydligare struktur.

2.1.2 Purify

Purify [10] är ett mycket omfattande verktyg för att hitta olika typer av fel som bland annat minnesläckor. Detekteringen sker under körning av kompilerad kod. Verktyget finns till både Linux (inklusive Solaris) och Windows. Det används för att testa applikationer främst skrivna i C++ och även i Java men då enbart för Solaris. Purify är en kommersiell programvara och kräver licens för att kunna användas. I det här fallet finns inte mer information om Purify på grund av att licens saknades och analysen kunde inte göras mer ingående.

2.1.3 MemWatch

MemWatch 2.71 [12] är ett Open Source-verktyg som kan detektera minnesläckage i C-applikationer. Det har även stöd för C++ men enligt dokumentationen rekommenderas

inte användning av verktyget till C++. MemWatch detekterar minnesläckor genom att verktygets källkod inkluderas i den egna applikationens källkod och sedan kompileras. Dessutom kan verktyget upptäcka skrivningar utanför tillåtna gränser i minnet, så kallade buffer overflows. När det kompilerade programmet har exekverat färdigt skrivs en rapport i form av en textfil i samma katalog som programmet exekverades i. MemWatch ersätter alla förekomster av minnesallokeringsfunktionerna malloc, calloc, realloc och free i koden med anrop till sina egna minnesallokeringsfunktioner. Även new och delete ersätts om detektering är aktiverad för språket C++. Alla Allokeringar registreras i en länkad lista vid analys av källkoden och avregistreras sedan från listan när ett anrop för avallokering påträffas. Fördelen är att det här är ett snabbt och enkelt sätt att kolla om applikationer har minnesläckor eftersom i princip bara en inkludering av header-filen till MemWatch kod behöver göras. En nackdel är att enligt dokumentationen som följer med MemWatch klarar det inte alltid att med 100 % säkerhet hitta alla minnesläckor. MemWatch klarar bara att detektera minnesläckor från allokeringar med malloc-funktionerna i C och new i C++ och har inte stöd för 64-bitars plattformar (64-bit Windows/Linux.) En annan nackdel är att detekteringen direkt försämrar applikationens prestanda på grund av att verktyget och källkoden integreras vid kompilering. Det kan finnas fall där det kan vara avgörande som till exempel i realtidsapplikationer som är tidskritiska i sin exekvering.

2.1.4 Visual Leak Detector

Visual Leak Detector [14] är ett annat verktyg som kan detektera minnesläckor i kompilerade applikationer. Verktyget har stöd för både C och C++ och fungerar som ett alternativ till den inbyggda minnesläckagedetektorn i Visual C++. Visual Leak Detector används genom att inkludera ett bibliotek vld.h (vld.lib) i den kod som ska testas. Liksom MemWatch [12] tar Visual Leak Detector bara hänsyn till minneshanteringsfunktionerna malloc, calloc, realloc och free, samt new och delete. Istället för att skriva en rapport i en textfil visar Visual Leak Detector en rapport i debugfönstret i Visual C++. Rapporten

består av vilken rad och i vilken fil som minnesläckan uppstod i. Rapporten visar också vad det läckta minnet innehåller i form av text och en hexadecimal representation. Det går dessutom att ställa in detaljnivån i rapporten. Visual Leak Detector har även stöd för 64-bitarsversionen av Windows men kräver omkompilering av källkoden till verktyget för att kunna utnyttja detta. Källkoden till detta verktyg kan laddas ned från verktygets hemsida [14] och det är fritt att göra ändringar i koden för att anpassa det till egna behov. Fördelen med Visual Leak Detector är att det är väldigt enkelt att använda. En nackdel är att verktyget bara fungerar i Visual C++ 6.0/.NET. En annan nackdel är att det inte kan detektera minnesläckage från till exempel COM-anrop (COM står för Component Object Model), vilka allokerar minne på ett annat sätt än med malloc och new. En ytterligare nackdel är att detekteringen påverkar direkt prestandan i applikationen, precis som för MemWatch, eftersom verktyget är integrerad i applikationen efter kompilering.

2.1.5 YAMD

YAMD (Yet Another Malloc Debugger) [2] är ett Open Source-verktyg som detekterar minnesläckor under exekvering liksom MemWatch [12] och Visual Leak Detector [14]. Förutom minnesläckor kan YAMD även detektera otillåten läsning och skrivning utanför det allokerade minnet och kan rapportera detta precis på den instruktion som utförde den felaktiga åtkomsten. Minnesläckor och felaktiga läsningar och skrivningar rapporteras med fullständig tillbakaspårning genom att visa alla funktionsanrop fram till minnesallokeringen där felet uppstod. YAMD klarar dessutom av att spåra minnesläckor på en låg nivå som till exempel indirekta allokeringar från funktioner i kompilerade standardbibliotek som string.h. YAMD kan detektera alla allokeringar och avallokeringar gjorda med malloc, calloc, realloc och free, samt new och delete. Rapporteringen av minnesläckor sker till en loggfil. YAMD är skrivet för GNU C/C++ [8] och fungerar bäst i Linux. Det finns ett visst stöd för DJGPP [1] men kravet är att det måste köras under rent DOS med CWSDPMI [17] (Charles W Sandmann DOS Protected Mode Interface) som DPMI-server. (DPMI står för DOS Protected Mode Interface.) Fördelen med YAMD

är att det kan detektera minnesläckor inte bara från det egna programmet utan även i standardbibliotek. En annan fördel är tillbakaspårningen av alla funktionsanrop från den punkt där minneshanteringsfelet uppstod, vilket ger en tydlig överblick av vad som kan vara orsaken till minneshanteringsfelet. En nackdel med YAMD är att det bara har fullt stöd för GNU C/C++ och fungerar inte med andra utvecklingsmiljöer. YAMD har också nackdelen att det påverkar applikationens prestanda liksom MemWatch och Visual Leak Detector eftersom YAMD integreras i applikationen efter att applikationen kompilerats.

2.1.6 MemLeakCheck

MemLeakCheck [21] är ett Open Source-verktyg som kan användas för att detektera minnesläckor under exekvering men också för att automatiskt frigöra minne som allokerats i applikationer. Verktyget stöder endast program skrivna i C och kan detektera minnesläckor som har orsakats av minneshanteringsfunktionerna malloc, calloc, realloc och free. Övriga minnesallokeringsfunktioner till exempel de som ingår i stränghanteringsfunktioner i standardbibliotek kan inte analyseras av detta verktyg. MemLeakCheck är inte begränsad till något operativsystem eller utvecklingsmiljö, vilket är en fördel. DLL-versionen av verktyget fungerar korrekt tillsammans med applikationer som använder trådar. MemLeakCheck fungerar också på ett mer unikt sätt än andra verktyg, vilket gör det snabbare när det gäller att hålla reda på allokeringarna. Principen är att allokera 16 byte extra minne för varje allokering. Diagnostisk information för varje allokering sparas i början av det allokerade minnesutrymmet. Eftersom denna information sparas i det allokerade minnet, istället för i en separat lista eller array, behövs det ingen sökning efter den sparade informationen, vilket gör att verktyget fungerar snabbare. Endast de pekare som används vid allokeringen av minne sparas i en array. Jämför detta med en länkad lista som innehåller diagnostisk information om referenser till varje minnesallokering. För varje pekare som får minne allokerat sparas en array i pekaren för att hålla reda på varje referens till samma allokerade minne. För den automatiska minneshanteringen gäller också följande; när omkring 1000 allokeringar har

gjorts komprimeras arrayen med alla pekare till allokeringarna för att undvika fragmentering och all information för varje minnesallokering uppdateras. När antalet referenser i arrayen för en viss minnesallokering blir noll frigörs minnet automatiskt. Den största fördelen är att användaren kan antingen reda ut minnesläckor i sin applikation eller slipper helt tänka på att avallokera minnet själv. Den automatiska avallokeringen av minne leder förstås till sämre prestanda vilket kan vara en nackdel i vissa fall men då finns alternativet att bara använda sig av detektering av minnesläckor istället. En nackdel är att verktyget endast klarar av applikationer skrivna i C och detekterar då endast minnesläckor orsakade av malloc, calloc, realloc och free. Verktyget MemLeakCheck fungerar bäst på Windows-plattformen men kan fungera även i andra operativsystem. I det fallet krävs omkompilering. En annan nackdel är att detekteringen påverkar applikationens prestanda eftersom verktyget integreras i applikationen efter kompilering.

2.1.7 CMemLeak

CMemLeak [16] är ett verktyg som enbart detekterar minnesläckor under exekveringstid och klarar detektering av minnesallokeringsfunktionerna malloc, calloc, realloc, free och strdup. Det klarar att detektera skrivning utanför det allokerade minnet, skrivning till redan avallokerat minne, avallokering av redan avallokerat minne och minne som inte har frigjorts, det vill säga minnesläckage. Verktyget är avsett för applikationer skrivna i C och fungerar i de flesta utvecklingsmiljöer och är plattformsoberoende. Verktyget används liksom MemWatch [12] genom att källkoden inkluderas i applikationens källkod och kompileras tillsammans. När applikationen exekveras genereras en rapport av minneshanteringsfelen. Rapporten sparas till filen CMemLeak.txt efter exekvering. CMemLeak klarar inte korruption av dess egna datastrukturer, det vill säga om applikationen skulle skriva över minne som CMemLeak använder. Det skulle i så fall med största sannolikhet leda till en krasch av applikationen. Problemet kan lösas genom att i verktyget ställa in antalet förväntade allokeringar innan krasch. Då kan applikationen bryta sin exekvering och därmed tillåta CMemLeak att skriva en rapport innan

applikationen kraschar. Fördelar med CMemLeak är att det är lätt att använda och att det fungerar på de flesta plattformar. Nackdelarna är att det endast har stöd för applikationer skrivna i C och är integrerat i applikationen efter kompilering vilket kommer att påverka applikationens prestanda under exekvering.

2.1.8 Sammanställning av de befintliga verktygen

Nedan följer en sammanställning av de befintliga verktygen:

Verktyg	Statisk detektering	Dynamisk detektering	Program-språksstöd	Minnes-läckage	Avallokering av / skrivning till oallokerat minne	Buffer overflows / Array bounds checking	Användar-vänlighet (1 Dålig - 5 Bra)	Operativ-system	Integreras i applikation
Splint	Ja	Nej	C	Ja	Ja	Ja	2	Win32, UNIX	Nej
Rational Purify	Nej	Ja	C, C++, Java (Solaris)	Ja	Ja	Ja	ingen uppgift	Win32, UNIX, Solaris	Ja
MemWatch	Nej	Ja	C, C++	Ja	Nej	Ja	3	Win32, UNIX	Ja
Visual Leak Detector	Nej	Ja	C, C++	Ja	Nej	Nej	4	Win32	Ja
YAMD	Nej	Ja	C, C++	Ja	Ja	Ja	4	UNIX, MS-DOS	Ja
MemLeakCheck	Nej	Ja	C	Ja	Nej	Nej	3	Alla	Ja
CMemLeak	Nej	Ja	C	Ja	Ja	Ja	3	Alla	Ja

Tabell 2-1 Sammanställning av de befintliga verktygen.

2.2 Användargränssnitt

De flesta program och applikationer har ett användargränssnitt som användaren använder för att kommunicera med applikationen. Det finns olika typer av användargränssnitt som till exempel grafiska användargränssnitt. Här följer de antaganden som vi har gjort för användargränssnittet i vår prototyp av verktyget:

- Ett GUI (Graphical User Interface) är inte aktuellt. Detta på grund av att tiden inte räcker till. Därför har vi valt att göra verktyget konsollbaserat, det vill säga det ska köras via ett kommandoradsgränssnitt, CLI (Command Line Interface). Det är

dock möjligt att koppla ett GUI till verktyget i efterhand.

- Verktygets källkod är inte portabelt mellan olika operativsystemplattformar. Detta på grund av att verktyget till viss del kommer att utvecklas i Flex [7] och Bison [6], se kapitel 5.1.

De val vi gjort för det konsollbaserade gränssnittet är följande:

- Filerna med källkoden som ska testas anges som parametrar till verktygets gränssnitt. Vi antar att verktygets namn är mls (memory leak scanner) och filerna som ska testas heter main.c och functions.c. Då anges parametern main.c och functions.c till mls. Vilken ordning som filerna anges i spelar ingen roll. Detta skrivs enligt följande:

```
./mls main.c functions.c
```

- Rapporteringen från verktyget kan ske till fil eller direkt ut i konsollfönstret. För att rapportera ut direkt i konsollfönstret görs inga tillägg på kommandoraden men vid rapportering till fil ska namnet på rapportfilen anges efter en flagga, som betecknas -f, och användaren ska ange namnet på rapportfilen. Om filen redan existerar så skrivs den över, annars skapas den. Om flaggan -f används men ingen rapportfil specifieras, kommer verktyget att skapa en fil med namnet mls_logfile.txt och skriva rapporten till den. Formatet på den fil som rapporten skrivs i är i UNIX textformat. Om rapportfilen heter till exempel report.txt, då kan rapportering till filen report.txt görs på följande sätt:

```
./mls main.c functions.c -f report.txt
```

2.3 Sammanfattning

Av den undersökning som vi har gjort av de befintliga verktygen har vi kommit fram till att det finns ett behov av att utveckla en prototyp av ett verktyg eftersom det bara finns ett enda verktyg som detekterar minneshanteringsfel statiskt. Resten av de befintliga verktygen som analyserats använder dynamisk detektering av minneshanteringsfel. Verktyg som använder dynamisk detektering påverkar exekveringstiden av applikationen som analyseras, eftersom de integreras i applikationen. Dessutom detekteras inte alla minneshanteringsfel på en gång ifall applikationen skulle krascha. Några verktyg fungerar endast under ett visst operativsystem och Visual Leak Detector [14] krävde Visual C++ för att det skulle fungera. En annan aspekt som gäller för de verktyg som kompileras in i testapplikationen är minneskorruption. I detta fall finns det en risk att testapplikationen orsakar minneskorruption genom att skriva över gränserna för det minne som applikationen själv allokerat och eventuellt förstöra innehåll i minne som detekteringsverktyget använder.

Splint [3, 5] är det verktyg som verkar vara att föredra av dem när det gäller alla dessa aspekter. Eftersom detekteringen är helt statisk finns det ingen risk för minneskorruption från testapplikationen då den aldrig behöver exekveras tillsammans med Splint. Problem med krascher av testapplikationen undviks och alla fel kan detekteras i endast en körning av verktyget. Källkoden till testapplikationen behöver inte modifieras på något sätt förutom i de fall där annotationer behövs för att detektera eventuella brister. Dessutom kan Splint detektera fel i dynamisk minneshantering inklusive allt det som det lite äldre Lint [13] kan detektera. Problemet är att det inte är ett lättanvänt program och kräver en hel del erfarenhet och instudering för att man ska bemästra alla dess funktioner som till exempel hur det går att filtrera bort detaljer från rapporten så att endast minnesläckorna visas från detekteringen. Rapporteringen från Splint har ingen bra struktur, vilket skulle kunna förbättras.

3 Kravspecifikation

För att få med all funktionalitet i verktyget som vi tänkt oss tar vi fram en kravspecifikation. I denna specifikation beskriver vi vilka funktioner som vårt verktyg bör ha för att fungera på ett korrekt sätt. Ett exempel på ett krav i vår specifikation är att verktyget måste kunna läsa in en eller flera filer. Vi beskriver alla nödvändiga krav för projektet i vår specifikation. Sedan under projektets gång kan nya krav och ändringar uppkomma. Därför kan kravspecifikationen komma att ändras under projektets gång.

3.1 Användbarhetskrav

3.1.1 Start av verktyget utan parametrar

Vid start av verktyget utan parametrar ska ett meddelande visas som specificerar hur verktyget ska användas. Verktyget ska då tala om att det måste startas genom att ange alla filer, där de funktionsdefinitioner finns, som ska testas med verktyget.

3.1.2 Start av verktyget med en eller flera källkodsfiler som parametrar

Verktyget måste kontrollera att parametrarna är giltiga sökvägar till källkodsfiler. I exakt en av filerna måste funktionen `main()` vara definierad. Om någon av sökvägarna är ogiltig eller om någon av filerna inte finns, skrivs ett felmeddelande ut.

3.1.3 Verktygets beteende efter att alla giltiga källkodsfiler har angetts

Verktyget ska söka igenom källkodsfilerna efter minnesläckage, felaktiga avallokeringar av minne, läsning från och skrivning till ogiltiga minnesområden, samt dangling pointers. Sedan skriver verktyget ut en rapport om alla eventuella minneshanteringsfel i

konsollfönstret eller i en loggfil som användaren ska namnge. Om användaren anger flaggan som talar om för verktyget att felrapporteringen ska skrivas till en loggfil men inte anger något namn för loggfilen, kommer en loggfil med namnet mls_logfile.txt att skapas automatiskt.

3.2 Funktionella krav

Verktyget ska kunna detektera och rapportera minnesläckage, felaktig avallokering av dynamiskt minne, dangling pointers samt läsning från och skrivning till ogiltiga minnesområden i källkod skriven i C/C++. Det ska också kunna rapportera vilken fil och vid vilket radnummer i källkoden som felet inträffade i. Verktyget ska kunna ge varningar i de fall det inte går att med säkerhet avgöra om ett fel kommer att inträffa, det vill säga när det i koden finns olika exekveringsvägar som till exempel if-satser och switch-satser. Verktyget måste detektera dessa fel i alla filer som angivits som parametrar till verktyget och även filer som inkluderas i källkoden dock inte standardbiblioteken. De detekterade felen måste loggas i en fil samtidigt som de skrivs ut för användaren. Verktyget ska även ge en påminnelse om att källkoden som ska analyseras måste ha kontrollerats syntaktiskt.

4 Konstruktionslösning

I detta kapitel beskriver vi problemen med att detektera minnesläckage, felaktiga avallokeringar av minne, dangling pointers, samt läsning från och skrivning till ogiltiga minnesområden i källkod. Vi undersöker olika lösningar på dessa problem samt fördelar och nackdelar med dem. Därefter kommer vi att göra en utvärdering av lösningarna och använda den lösning som passar bäst till prototypen av vårt verktyg. Vi kommer även att nämna de antaganden som gäller för lösningen samt motivera varför vi gör dessa antaganden. Även konstruktionsbegränsningar som sätts av tidsbegränsningen för projektet kommer att anges här.

4.1 Konstruktionsproblem

För att hitta minneshanteringsfel i källkod skriven i C/C++ krävs det en ingående analys av källkoden. För att få en fullständigt säker och pålitlig analys krävs det att varje minneshanteringsfel måste detekteras och rapporteras. Dessutom finns det en risk vid dynamisk minneshantering att minnet kan avallokeras felaktigt som att avallokera minne som redan är avallokerat eller som aldrig har allokerats.

En statisk analys av källkoden medför ett stort problem; att det uppkommer mer än en exekveringsväg. I detta fall måste alla exekveringsvägar analyseras då det inte går via den statiska analysen att avgöra vilken exekveringsväg som ska väljas. Alla exekveringsvägar kan representeras som ett träd där varje barn är en möjlig exekveringsväg. I den statiska analysen måste varje barn till roten i trädet analyseras efter dynamiska minneshanteringsfel.

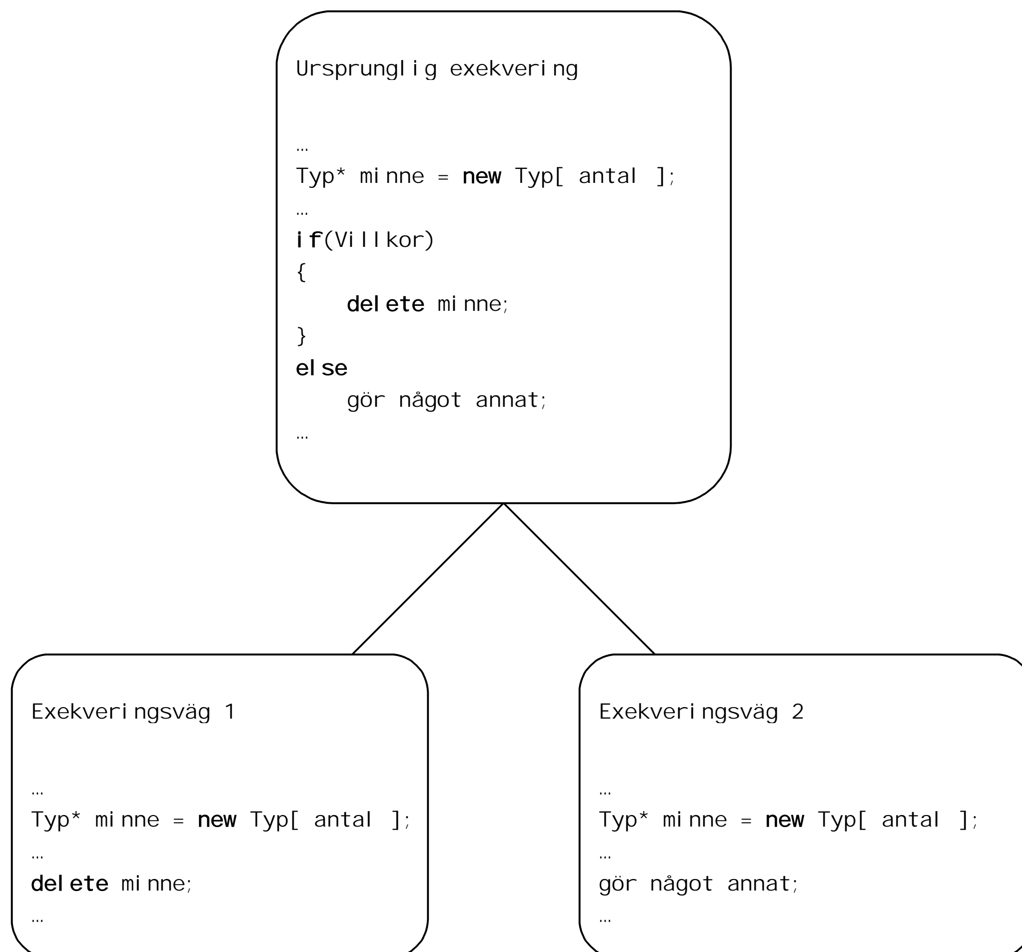
Eftersom minne kan allokeras vid en punkt i källkoden och avallokeras vid en annan kan det hända att allokering eller avallokering som finns i villkorstyrda satser inte kommer att utföras. Att möjliggöra utvärdering av villkor krävs att variabler från källkoden och deras

värden sparas och hålls uppdaterade. Villkor kan vara beroende av inläsning från någon användarinmatning som till exempel systemklockan eller en tangentbordsinmatning, vilket gör utvärdering av villkor ännu svårare. Om villkoren inte kan utvärderas, går det inte att avgöra med säkerhet vilken exekveringsväg som kommer att väljas. Därför går det inte heller att med säkerhet avgöra om minne allokeras eller avallokeras. Det finns dock en möjlighet att göra en bedömning av en risk till minnesläcka eller felaktig avallokering inom villkorsstyrda exekveringsvägar. Det kan göras genom att analysera varje exekveringsväg för att kontrollera om det finns kod för dynamisk allokering eller avallokering av minne i någon av dem. Om minne allokeras dynamiskt i en gemensam exekveringsväg men avallokeras i någon villkorsstyrd exekveringsväg finns det en risk för en minnesläcka. Detsamma gäller minne som allokeras dynamiskt i en villkorsstyrd exekveringsväg men som avallokeras i en gemensam exekveringsväg. Det förekommer också fall där minne allokeras dynamiskt i en villkorsstyrd exekveringsväg men avallokeras i en annan villkorsstyrd exekveringsväg. I de fallen finns det risk för både minnesläckor och felaktiga avallokeringar. I följande exempel på dynamisk minneshantering finns ett villkor som styr om minnet som har allokerats ska avallokeras:

```
...
Typ* minne = new Typ[ antal ];
…
if(Villkor)
{
    delete minne;
}
else
{
    gör något annat;
}
...
```

I detta fall skapas två möjliga exekveringsvägar vilka måste analyseras. Den första av de två exekveringsvägarna är om villkoret i if-satsen uppfylls. Då kommer koden som finns i

if-satsen att exekveras men kod som finns i else-satsen kommer däremot inte att exekveras. Den andra exekveringsvägen är om villkoret inte uppfylls. I detta fall kommer endast koden i else-satsen exekveras. Följden blir att det finns risk att det allokerade minnet inte avallokeras på grund av att kod i if-satsen aldrig kommer att exekveras. Därför innebär det också att koden som ska exekveras ser olika ut beroende på villkoret. Följande figur visar de två olika exekveringsvägarna för ovanstående exempel:



Figur 4-1 Illustrering av olika möjliga exekveringsvägar.

Ett annat problem är att få verktyget att läsa källkoden och sedan kunna tolka vad som står i den. Detta problem kan lösas genom att först läsa in koden, dela upp koden i de

minsta syntaktiska beståndsdelarna, så kallade tokens, och sedan låta en parser kontrollera i vilken ordning dessa tokens står i källkoden. Därmed skulle det gå att avgöra när till exempel pekarvariabler deklareras i källkoden och när de används.

En mer djupgående analys är att i analysen av källkoden även hålla reda på värden i variabler. Med hjälp av variabler kan villkorsstyrda val av exekveringsväg och antal iterationer i loopar avgöras. Även storlekar och gränser till en array kan spåras med hjälp av värden i variabler. I vissa fall där variabeln får värde via någon funktion som läser från maskinvara, som till exempel tangentbordet, går det inte att avgöra vilket värde variabeln innehåller. I dessa fall skulle källkoden kunna testas genom att variabeln antar värden inom ett visst intervall. Om variabeln till exempel har typen long, kan variabeln anta värden mellan det högsta möjliga respektive det minsta möjliga talet en variabel av typen long kan anta.

4.2 Alternativa lösningar

Det finns några alternativ till lösningar på problemet med att kunna statiskt detektera minneshanteringsfel i källkod skriven i C/C++. Det viktigaste är att kunna känna igen syntaxen i C/C++ och uttrycken för dynamisk minneshantering. Därför måste en parser skapas som kan läsa av och analysera källkoden. Det finns olika sätt att skapa en parser. En parser kan skrivas för hand eller den kan genereras med hjälp av ett verktyg genom att skapa regler och definiera tokens för att känna igen syntaxen. Det är möjligt att spara tid genom att använda ett verktyg som genererar en parser men ofta kräver ett sådant verktyg en hel del instudering. Källkoden kan sökas igenom efter dynamiska minnesallokeringar på olika sätt. En mycket enkel analys i teorin är att till exempel söka efter nyckelordet new i hela källkoden och spara alla dessa som allokeringar. Sedan kan allokeringarna tas bort en efter en när nyckelordet delete påträffas. Fördelen är att det är en relativt enkel lösning därför att det kräver bara detektering av två nyckelord. Däremot har denna typ av analys många stora nackdelar. Den tar inte hänsyn till villkorliga satser som if-satser och

case-satser där det bara finns en risk att det uppstår en minnesläcka om det finns en allokering eller avallokering inuti en sådan sats. Dessutom kan dessa nyckelord finnas i kommentarer och i stränguttryck, vilket kan leda till falska detekteringar. Detta problem kan lösas enkelt genom att ignorera kommentarer och stränguttryck eftersom dessa har ett visst syntaktiskt utseende som lätt kan detekteras. Ett annat sätt är att göra en fullständig analys och gå igenom alla syntaktiska beståndsdelar under parsningen. Detta är naturligtvis en mer avancerad lösning och kräver mer tid att implementera men möjliggör en betydligt säkrare analys. De stora fördelarna med detta är att både kommentarer och stränguttryck parsas och tolkas som kommentarer respektive stränguttryck, vilket eliminerar risken för att göra falska detekteringar i dessa fall. Dessutom förenklar detta analysen av flera exekveringsvägar som gäller för if-satser och case-satser så att risk för en eventuell minnesläcka kan bedömas.

4.3 Vår lösning

Vi valde att använda UNIX-verktygen Flex [7] och Bison [6] för att skapa en lexer respektive parser för att kunna skanna källkod. Fördelarna för oss med att använda verktygen Flex och Bison är att vi redan hade kunskaper om dessa verktyg för att kunna utveckla prototypen utan att behöva lära oss verktygen först och därmed spara tid. Dessutom är det ett smidigare alternativ att använda verktyg för att generera en parser än att skriva en parser för hand, vilket antagligen skulle ta mycket längre tid.

Vi använde Flex för att skapa en skanner som känner igen tokens från C/C++. Bison använde vi för att skapa en parser som tolkar källkod i C/C++ syntaktiskt och göra det möjligt att detektera dynamiska allokeringar och avallokeringar av minne samt deklarationer och tilldelningar av pekare. Nackdelarna med att använda verktygen Flex och Bison är själva begränsningarna i dessa verktyg. En begränsning är kommunikationen mellan Flex och Bison som sköts via globala variabler. Denna begränsning medför även att vi är tvungna att använda oss av globala variabler för att

kunna kommunicera mellan verktygen, vilket leder till försvårad läsbarhet i källkoden och ökad risk för sidoeffekter. Med sidoeffekter menar vi att värdet i en variabel ändras i delar av programmet där variabeln inte ska vara i scope.

4.4 Antaganden i vår konstruktionslösning

Vid utveckling av prototypen av verktyget måste den viktigaste funktionaliteten prioriteras. För att uppnå detta har vi gjort några antaganden om verktygets funktionalitet. Nedan följer de antaganden som vi gjort i vår lösning:

1. Alla källkodsfiler som ska ingå i parsningen måste redan vara syntaktiskt kontrollerade. Anledningen till detta är att verktygets uppgift inte är att göra en syntaktisk korrektion av källkoden utan enbart en analys för att hitta minnesläckor och andra fel som är relaterat till minneshantering.
2. Alla källkodsfiler förutom header-filer som ska ingå i parsningen måste anges som parametrar till verktyget. Anledningen till detta är att användaren själv ska kunna avgöra vilka filer som ska ingå i parsningen.
3. Verktyget kommer inte att ta hand om pekararitmetik på grund av att man inte har tillgång till minnesadressen som pekaren refererar till. En sådan lösning skulle kräva en simulering av riktig minnesallokering.
4. Verktyget kommer inte att parsa standardbibliotek som har inkluderats i filer varken explicit eller implicit. Anledningen till detta är att mycket källkod som finns definierad i standardbiblioteken ligger oåtkomlig i redan kompilerade filer.
5. Verktyget har inte stöd för parsning av goto-satser.
6. I de fall där verktyget inte kan säkerställa minneshanteringsfel men att det ändå finns en risk för det kommer verktyget att rapportera detta. Anledningen till det är att i vissa situationer går det inte att avgöra med säkerhet vilken exekveringsväg som väljs. Då berörs främst dynamisk allokering och avallokering av minne och

kopiering av referenser i samband med dynamisk minnesallokering i några av dessa exekveringsvägar, se figur 4.1.1.

7. Verktyget ska kunna detektera minneshanteringsfel orsakade av minnesallokeringsfunktionerna new, malloc, calloc och realloc samt minnesavallokeringsfunktionerna delete och free.

4.5 Begränsningar

En begränsning som vi stötte på var att vi inte hade tillräcklig kunskap om hur man genererar lexern och parsern i C++-kod med hjälp av Flex [7] och Bison [6]. Vi försökte få dessa verktyg att fungera med C++-kod men vi lyckades inte. Därför var vi tvungna att generera lexern och parsern i C-kod istället. Detta ledde till att det blev svårare att implementera prototypen eftersom detta ledde till att vi blev tvungna att skriva egna länkade listor istället för att kunna använda oss av färdiga listor i standardbiblioteken (Standard Template Library) som finns för C++.

5 Implementation

I detta kapitel beskriver vi hur prototypen av vårt verktyg är implementerad. Vi beskriver vilken miljö vi har implementerat prototypen i och även hur vi med hjälp av hjälpverktygen Flex [7] och Bison [6] har gjort det. Vi beskriver även de alternativa verktyg som fanns tillgängliga för implementation av verktyget. De datastrukturer som vi har implementerat och de funktioner som vi har använt i prototypen beskrivs i detalj. Även teoretiska lösningar som vi inte har hunnit implementera i prototypen beskrivs i det sista avsnittet.

5.1 Utvecklingsmiljö

Den utvecklingsmiljö som vi har valt till att utveckla prototypen i är UNIX-miljö. Vi ansåg att denna miljö var den mest lämpliga med tanke på vår erfarenhet av att arbeta i den. En annan anledning till vårt val var att miljön tillhandahöll hjälpverktygen Flex [7] och Bison [6], vilka vi använt för att skapa en parser som kan parsa källkod skriven i C/C++. Flex och Bison är vidareutvecklade från verktygen Lex [18] och Yacc [19], vilka var standardverktyg i de flesta UNIX-varianter. Kompilatorn som vi använt för att kompilera vårt verktyg är GNU C [8] i UNIX eftersom det var lämpligast att använda det i interaktion med Flex och Bison. Anledningen är att vi har tidigare erfarenhet av arbeta med GNU C tillsammans med Flex och Bison i UNIX-miljö.

5.2 Alternativa verktyg

Det finns alternativa verktyg för generering av en parser. Några av dessa är Bison [6], JavaCC [11] (Java Compiler Compiler), SableCC [9] och ANTLR [15].

Bison är en vidareutvecklad variant av det äldre verktyget Yacc [19] och finns till plattformarna GNU/Linux, UNIX och Win32. Det är ett verktyg som genererar en parser för de flesta möjliga typer av syntax som till exempel källkod och skript i något

programspråk. Bison skapar en parser genom att generera källkod till den antingen i C eller C++. Därför krävs det en C eller C++-kompilator för att kompilera parsern. Fördelarna med detta verktyg är att det inte är alltför komplicerat att använda och det finns dokumentation och gott om exempel att hitta på webben. Nackdelarna är att det fungerar bara ihop med C/C++-kompilatorer och att dokumentationen inte är lika omfattande som till exempel för verktyget Flex [7], vilket används för att generera en lexer. Dessutom ingår inte någon lexer i parsern utan den måste skrivas på egen hand eller använda ett verktyg som genererar en lexer.

JavaCC är ett nyare verktyg som fungerar på alla operativsystem som har stöd för Java och kan generera parsrar för de flesta möjliga syntaxer. Parsern som genereras av JavaCC är källkod i Java. Fördelarna med verktyget är att det har även ett IDE (Integrated Development Environment) och stöder tokens med en utökad Unicode-teckenuppsättning. JavaCC har också en inbyggd generering av lexer till parsern. Nackdelarna med JavaCC är att dokumentationen är svår att tyda och felrapporteringen från verktyget är aningen bristfällig, vilket leder till svårigheter att hitta fel.

SableCC är också ett verktyg skrivet för Java och fungerar därför på alla operativsystem med stöd för Java. Detta verktyg har ett objektorienterat gränssnitt för att generera abstrakta syntaxträd. För varje nod i detta syntaxträd kan man skapa regler. För att köra parsern anropas en metod för att traversera syntaxträdet. Parsern skapas främst i Java-källkod men kan också skapas i källkod i C, C++, C#, OCAML eller Python. SableCC har också inbyggd generering av lexer till parsern. Fördelar med SableCC är att det kan generera källkod för parsern i alternativa språk och att det har inbyggd generering av lexer. Nackdelar är att det är komlicerat jämfört med till exempel JavaCC och har en sämre felmeddelanderapportering än JavaCC.

5.3 Flex & Bison, Lex & Yacc

Flex [7] är ett verktyg som används för att skapa en så kallad lexer genom att generera kod till den. En lexer är en slags textskanner som läser in text och delar upp den i tokens. Texten kan komma från en fil, en sträng eller genom direkt inmatning från tangentbordet. Den skanner som vår prototyp använder läser in från både filer och strängar. I Flex kan man skapa regler, så kallade rules i Flex, för varje token med hjälp av reguljära uttryck för att kunna identifiera dem, ta ut dem från textströmmen och vidta åtgärder, så kallade actions, baserat på varje enskild token. I Flex kan man med hjälp av reguljära uttryck matcha alla de tokens som har definierats, till exempel i källkodsfiler i C/C++. Nedan följer ett exempel på definiering av tokens i Flex-filen för skanning av ett uttryck för dynamisk allokering av minne i C++:

```
TOKEN_TYPE (bool)|(char)|(unsigned\ char)|(signed\
char)|(wchar_t)|(int)|(unsigned\ int)|(signed\ int)|(short\
int)|(short)|(unsigned\ short\ int)|(signed\ short\ int)|(long\
int)|(long)|(unsigned\ long\ int)|(signed\ long\
int)|(float)|(double)|(long\ double)|(signed)|(unsigned)|(signed
short)|(unsigned short)|(signed long)|(unsigned long)
TOKEN_STAR          \*
TOKEN_NAME          [a-zA-Z_][a-zA-Z_0-9]*
TOKEN_ASSIGN        \=
TOKEN_NEW           new
TOKEN_LEFT_SQBR     \[
TOKEN_RIGHT_SQBR    \]
TOKEN_DIGIT         [0-9]+
TOKEN_SEMICOLON     ;
TOKEN_WHITESPACE    [ \t]+
TOKEN_NEWLINE       [\n\r]
```

När textskannern matchar ett token i textströmmen rapporteras den till parsern. Som fortsättning på exemplet ovan visas hur dessa tokens sedan returneras till parsern på följande sätt:

```
{TOKEN_TYPE}           {  yylval = strdup(yytext);
                          return TOKEN_TYPE; }

{TOKEN_STAR}           {  yylval = strdup(yytext);
                          return TOKEN_STAR; }

{TOKEN_NAME}           {  yylval = strdup(yytext);
                          return TOKEN_NAME; }

{TOKEN_ASSIGN}         {  yylval = strdup(yytext);
                          return TOKEN_ASSIGN; }

{TOKEN_NEW}            {  yylval = strdup(yytext);
                          return TOKEN_NEW; }

{TOKEN_LEFT_SQBR}      {  yylval = strdup(yytext);
                          return TOKEN_LEFT_SQBR; }

{TOKEN_RIGHT_SQBR}     {  yylval = strdup(yytext);
                          return TOKEN_RIGHT_SQBR; }

{TOKEN_DIGIT}          {  yylval = strdup(yytext);
                          return TOKEN_DIGIT; }

{TOKEN_SEMICOLON}      {  yylval = strdup(yytext);
                          return TOKEN_SEMICOLON; }

{TOKEN_WHITESPACE}     {}
```

```
{TOKEN_NEWLINE}          {}
```

Lexern arbetar genom att matcha ett definierat token som till exempel **TOKEN_TYPE** och sedan utför den åtgärder. I detta fall då **TOKEN_TYPE** matchas skickas det semantiska värdet för tokenet till parsern genom att kopiera det från en strängvariabel i lexern som heter yytext till en variabel i parsern som heter yylval. För att parsern ska veta vilket token som har matchats returnerar lexern det matchade tokenet till parsern via en return-sats. I de fall då det token som matchas inte har någon syntaktisk betydelse, till exempel **TOKEN_WHITESPACE** och **TOKEN_NEWLINE**, utför lexern inte några åtgärder, det vill säga att det matchade tokenet ignoreras och returneras inte till parsern.

Bison är ett verktyg som används för att skapa en parser genom att generera kod till den. Parserns uppgift är att kontrollera att de tokens som rapporteras från lexern stämmer överens enligt grammatiska regler som är definierade i parsern. I Bison måste alla tokens deklareras så att parsern känner igen de tokens som den tar emot från lexern. Token-deklarationerna i Bison sker på följande sätt:

```
%token TOKEN_TYPE TOKEN_STAR TOKEN_NAME TOKEN_ASSIGN TOKEN_NEW
TOKEN_LEFT_SQBR TOKEN_RIGHT_SQBR TOKEN_DIGIT TOKEN_SEMICOLON
```

Observera att **TOKEN_WHITESPACE** och **TOKEN_NEWLINE** kastas bort i lexern. Dessa tokens i vårt exempel har ingen betydelse för parsern. Eftersom de inte har någon grammatisk betydelse i C/C++ behöver de inte deklareras i Bison-filen. För att parsern ska kunna göra en semantisk analys av källkoden måste syntaxen i källkoden vara korrekt. Parsern har regler som förväntar tokens i en korrekt syntaktisk ordning. En regel i parsern kan motsvara en rad i källkoden där till exempel en minnesallokering görs. Nedan visas ett exempel på hur en rad i källkod, där en minnesallokering görs, motsvaras av en regel. Betrakta följande rad av C++ källkod:

```
int* a = new int[10];
```

Här allokeras totalt fyrtio byte minne dynamiskt (i den arkitektur som vi implementerat prototypen i tar en int upp fyra byte) och pekaren a sätts till att peka på det allokerade minnet. För att kunna analysera denna rad måste parsern ha en regel som motsvarar de tokens som raden består av. Regeln börjar med ett namn på själva regeln följt av ett kolon och ett tabulatortecken. Sedan följer de tokens i den ordning som ska motsvara den raden med källkod som ovan. Efter alla tokens skrivs C-kod för att spara information om allokeringen av minnet. Regeln avslutas sedan med ett semikolon. Denna regel kan se ut som nedan:

```
MEMORY_ALLOCATION: TOKEN_TYPE TOKEN_STAR TOKEN_NAME TOKEN_ASSIGN
TOKEN_NEW TOKEN_TYPE TOKEN_LEFT_SQBR TOKEN_DIGIT TOKEN_
RIGHT_SQBR TOKEN_SEMICOLON
{
  _mem_alloc_size = get_size($6, atol($8));
  strcpy(_value, "UNDEFINED VALUE");

  strcpy(_type_name, $1);
  strcat(_type_name, $2);

  a_add_item_by_details($3, _mem_alloc_size, _value , yylineno,
  _ID_number, _current_file_name, _alloc_list_head);

  r_add_item_by_details(_type_name, $3, yylineno, _ID_number,
      False, _current_file_name, _ref_list_head);
  _ID_number++;
}
;
```

I regeln `MEMORY_ALLOCATION` tas först reda på vilken datatyp som minnet ska allokeras för. Detta görs för att få reda på hur många bytes av minne som behöver allokeras enligt följande rad:

```
_mem_alloc_size = get_size($6, atol($8));
```

För detta ändamål finns det en funktion `get_size()` som beräknar det totala antalet byte, vilket kommer att tilldelas till variabeln `_mem_alloc_size`. De aktuella parametrarna som skickas in i funktionen är **`$6`** som motsvarar strängen i det sjätte tokenet i regeln, det vill säga `int` i detta exempel och **`$8`** som motsvarar det åttonde tokenet i regeln, det vill säga talet 10 i detta exempel.
Nästa instruktion skapar ett simulerat odefinierat värde för det allokerade minnet:

```
strcpy (_value, "UNDEFINED VALUE");
```

Variabeln `_value` håller reda på värdet som tilldelas till det minne som pekaren pekar på , till exempel:

```
int* a = new int(10);
```

I detta fall får `_value` värdet 10. I föregående kodexempel tilldelas det inte något värde till det allokerade minnet och därför sätts `_value` till `UNDEFINED VALUE`.
I de två följande instruktionerna hämtas informationen om datatypen som det allokerade minnet har:

```
strcpy (_type_name, $1);
strcat (_type_name, $2);
```

Variabeln `_type_name` håller reda på datatypen som pekaren har. **`$1`** får värdet i det första tokenet det vill säga **`TOKEN_TYPE`** och **`$2`** får värdet i det andra tokenet det vill säga **`TOKEN_STAR`**. Här kommer variabeln `_type_name` att få värdet `int*`. I nästa instruktion läggs den information som samlats hittills i ett nytt element i listan som innehåller alla dynamiska minnesallokeringar:

```
a_add_item_by_details ($3, _mem_alloc_size, _value , yylineno,
      _ID_number, _current_file_name, _alloc_list_head);
```

Funktionen `a_add_item_by_details()` lägger till information om pekarens namn vars värde fås av **`$3`**, antal byte allokerat minne, värdet i det allokerade minnet, radnumret i källkoden där allokeringen sker (`yylineno`), ett unikt ID-nummer för identifiering av denna minnesallokering och namnet för den fil där denna minnesallokering sker. `yylineno` är en extern variabel som skapas av Flex för att hålla reda på vilken rad i källkoden som lexern för tillfället skannar. `_alloc_list_head` är den länkade listan som informationen om minnesallokeringen kommer att sparas i.
Om alla de tokens som rapporteras från lexern motsvarar de i regeln och är i rätt ordning, kan parsern spara information om minnesallokeringen. Den semantiska analysen består i prototypens fall bland annat av att kontrollera alla fall där minne allokeras dynamiskt i den källkod som textskannern söker igenom. De regler som vi skapat för parsern med hjälp av Bison analyserar källkod skriven i C/C++. Parsern sparar viktig information om källkoden som variabler och deras värden, pekare och vad de refererar till, funktioner, deras parametrar, returtyp och returvärde, dynamisk minnesallokering och storleken av alla allokerade minnen.

Språket C/C++ stöder modulär programmering vilket innebär att det går att inkludera källkod från andra filer. Därför måste lexern kunna byta textström för att kunna läsa i de andra källkodsfilerna. I Flex finns det en inbyggd bufferthantering som möjliggör hopp i skanningen mellan olika filer. Dessutom finns möjligheten att kunna skanna från strängar. Vi har utnyttjat dessa möjligheter i vår prototyp som även öppnar och skannar filer som har inkluderats i källkoden. Dessa filer behöver dock inte anges explicit till prototypen. Alla funktionskroppar som finns i källkoden sparas som strängar i en funktionslista. Därför kan prototypen leta rätt på en funktion och skanna igenom den då ett funktionsanrop till den funktionen påträffas i källkoden och därefter återuppta skanningen i källkodsfilen efter funktionsanropet. För att veta vilken ordning alla tokens ska vara i när vi skannar dem i källkod har vi använt oss av verktyget Bison för att skapa

en parser med grammatik för C/C++. De grammatiska reglerna som är definierade i parsern kontrollerar följden av de tokens som textskannern skickar till parsern och på så sätt kan parsern känna igen vad som står i källkoden.

5.4 Generell uppbyggnad av prototypen

För att lyckas med att detektera minnesläckor, felaktiga dynamiska avallokeringar av minne och dangling pointers, parsar prototypen deklarationer av pekare, tilldelningar av dynamiska minnesallokeringar till pekare och borttagning av dynamiskt allokerat minne. Dessutom uppdateras informationen om dynamiskt allokerade minnen och alla pekare under hela parsningen. För en fullständig detektering krävs att prototypen håller reda på alla variabler, konstanter och deras datatyper samt funktioner, deras parametrar, returvärden och returtyper. Den metod som vi använt oss av i prototypen är en analys av bland annat pekarvariabler. Om en pekare deklareras och eventuellt tilldelas en adress till dynamiskt allokerat minne, måste den övervakas efter alla eventuella förändringar som kan påverka den tills den går ur scope. I detta fall kommer pekaren att tilldelas det ID-nummer som det dynamiskt allokerade minnet har. Detta görs på grund av att prototypen ska kunna övervaka alla dynamiskt allokerade minnesområden och pekare som pekar på dessa områden under parsningen. Det är viktigt att varje tilldelning undersöks noggrant, till exempel om en pekare kopieras till en annan pekare, så kallad aliasing. Eftersom flera pekare kan referera till samma allokerade minnesområde är det nödvändigt att kunna spåra alla pekare under deras livstid. Parsern måste även kunna spåra alla funktioner som finns med i källkoden eftersom dessa kan returnera pekare till dynamiskt allokerade minnesområden.

5.5 Detaljerad beskrivning av prototypen

De datastrukturer som vi använder i prototypen för att spara all information är ett antal dubbellänkade cirkulära listor. Anledningen till att vi väljer länkade listor istället för

arrayer är att vi i förväg inte vet hur många element som kommer att sparas i datastrukturerna. Informationen som sparas i elementen i respektive lista delas upp i följande kategorier:

- Lokala variabler
- Globala och statiska variabler
- Dynamiska minnesallokeringar
- Lokala pekare och aliasvariabler
- Globala pekare och aliasvariabler
- Funktioner
- Parametrar till funktioner

Varje kategori sparas i en separat lista. Lokala variabler lagras med typ, namn, värde, ID-nummer och filnamn. Under prototypens analys av källkod sparas lokala variabler som deklarerats i en funktion och tas sedan bort där analysen av funktionen avslutas. Varje lokal variabel sparas som element i en lista som en struktur enligt följande:

```
struct VariableItem
{
	char type [MAX_STRING_LENGTH];
	char name [MAX_STRING_LENGTH];
	char value [MAX_STRING_LENGTH];
	unsigned long ID_number;
	char file_name [MAX_STRING_LENGTH];
};
typedef struct VariableItem VItem;
```

Filnamnet sparas för att enkelt få veta i vilken källkodsfil som den lokala variabeln deklarerades i. ID-numret används för att skilja mellan alla variabler, pekare och referenser samt att kunna koppla pekare till dynamiskt allokerat minne och referenser till variabler.

Globala variabler sparas på samma sätt som lokala variabler fast i en separat lista. De sparas var och en som ett element enligt följande struktur:

```
struct GVariableItem
{
  char type [MAX_STRING_LENGTH];
  char name [MAX_STRING_LENGTH];
  char value [MAX_STRING_LENGTH];
  unsigned long ID_number;
  char file_name [MAX_STRING_LENGTH];
};
typedef struct GVariableItem GVItem;
```

Dynamiska minnesallokeringar sparas som element i en egen lista med namn, antal byte minne som allokerats, data som minnet innehåller, radnumret där allokeringen sker och ett unikt ID-nummer. Pekare kan ha samma ID-nummer som allokeringen och det tolkas som att pekaren refererar till allokeringen. Strukturen som den dynamiska minnesallokeringen har ser ut som nedan:

```
struct AllocItem
{
  char name [MAX_STRING_LENGTH];
  unsigned long number_of_bytes;
  char value [MAX_STRING_LENGTH];
  unsigned long line_number;
  unsigned long ID_number;
  char file_name [MAX_STRING_LENGTH];
};
typedef struct AllocItem AItem;
```

Namnet som sparas i elementet används enbart i debug-syfte. Antal byte kan fås genom att analysera variablernas värden i de fall där det är möjligt.

Alla pekare och referenser lagras i en separat lista med typ, namn, radnummer, ID_nummer, en flagga som indikerar om en pekare refererar till null eller inte och filnamn. Varje pekare eller referens sparas i en lista som en struktur enligt följande:

```
struct ReferenceItem
{
  char type [MAX_STRING_LENGTH];
  char name [MAX_STRING_LENGTH];
  unsigned long line_number;
  unsigned long ID_number;
  int refers_to_null;
  char file_name [MAX_STRING_LENGTH];
};
typedef struct ReferenceItem RItem;
```

ID-numret för varje minnesallokering är unikt och pekaren som det allokerade minnet tilldelas till får samma ID-nummer. På så sätt kan minnesallokeringen spåras till pekare som har samma ID-nummer. Prototypen tar bort informationen om de dynamiska minnesallokeringarna från listan när nyckelordet delete påträffas genom att hitta pekaren med det namn som har angivits efter detta nyckelord. Pekaren innehåller ID-numret som det dynamiskt allokerade minnet har. Därefter söks minnesallokeringen upp i listan med hjälp av ID-numret i pekaren och minnesallokeringen tas bort. Lokala pekare och referensvariabler tas bort från listan när funktionen som de har deklarerats i är färdiganalyserad. ID-numret kan ändras beroende på vad pekaren eller referensvariabeln refererar till. Om till exempel en pekare refererar till en variabel, får pekaren samma ID-nummer som den variabeln. Globala pekare och referenser lagras i en separat lista med typ, namn, radnummer, ID-nummer, en refers_to_null-flagga och filnamn i en struktur enligt nedan:

```
struct GReferenceItem
{
```

```
  char type [MAX_STRING_LENGTH];
  char name [MAX_STRING_LENGTH];
  unsigned long line_number;
  unsigned long ID_number;
  int refers_to_null;
  char file_name [MAX_STRING_LENGTH];
};
typedef struct GReferenceItem GRItem;
```

Eftersom globala och statiska pekare respektive referenser existerar under hela exekveringstiden efter att de deklarerats måste denna lista existera under hela analysen av källkoden och alla element i listan finnas kvar efter de skapats.

Varje funktion i källkoden sparas i en separat lista med returtyp, namn, en formell parameterlista, returvärde, ett retur-ID-nummer och ett eget ID-nummer. Filnamn sparas för att få kännedom om vilken fil som funktionen definierades i. Strängen contents är innehållet i funktionskroppen, det vill säga funktionskoden. Variabeln `func_body_line_number` innehåller radnumret i filen där källkoden i funktionens kropp börjar. En illustrering av funktionselementet visas nedan:

```
struct FuncItem
{
  char return_type [MAX_STRING_LENGTH];
  char name [MAX_STRING_LENGTH];
  struct PNode* param_head;
  char return_value [MAX_STRING_LENGTH];
  unsigned long return_ID_number;
  unsigned long ID_number;
  char file_name [MAX_STRING_LENGTH];
  char contents [FUNC_BUFFER_MAX_LENGTH];
  int func_body_line_number;
};
```

```c
typedef struct FuncItem FItem;
```

Retur-ID-numret används då en funktion returnerar en pekare eller ett alias och returvärdet används då funktionen returnerar ett värde. Returtypen avgör om det är ett värde, ett alias eller en pekare som returneras. Namnet på funktionen används för att söka upp funktionen i funktionslistan när den anropas från någon punkt i källkoden. Parameterlistan innehåller en eller flera parametrar vilka sparas med typ, namn, och ID-nummer. Parameternummer används för att ange ordningen på parametrarna i parameterlistan för varje funktion. ID-numret är unikt för all data som sparas i alla listorna bortsett från pekare och referenser som kan ha samma ID-nummer i de fall då de refererar till en parameter. Filnamnet anger vilken fil som parametern är deklarerad i. Ett parameterelement i parameterlistan har följande struktur:

```c
struct ParamItem
{
  char type [MAX_STRING_LENGTH];
  char name [MAX_STRING_LENGTH];
  unsigned long param_number;
  unsigned long ID_number;
  char file_name [MAX_STRING_LENGTH];
};
typedef struct ParamItem PItem;
```

För varje funktionselement i funktionslistan finns det en parameterlista. Parameterlistan kan vara tom eller ha ett obegränsat antal parameterelement. Efter att funktionslistan deklarerats finns den kvar under hela analysen av källkoden.

5.6 Begränsningar i prototypen

På grund av tidsbrist finns det i nuläget inget stöd för följande funktionalitet:

- Funktionspekare
- Objekt av egendefinierade datatyper (strukturer och klasser)

Funktionspekare har inte implementerats på grund av tidsbrist men en teoretisk lösning är att låta funktionspekare lagras på samma sätt som andra pekare som till exempel i prototypens referenslistor tillsammans med övriga typer av pekare. När en funktionspekare sätts att peka på en funktion kommer funktionspekaren att tilldelas samma ID-nummer som funktionen har. På så sätt kan man övervaka anrop till funktioner via funktionspekare i källkoden under parsningen.

Stöd för klasser och objekt har inte heller implementerats på grund av tidsbrist. Därför beskriver vi endast en teoretisk lösning på hantering av klasser och objekt i analysen. Alla klasser sparas i en lista i form av element. Strukturen skulle då kunna ha följande utseende:

```
struct CNode;

struct ClassItem
{
  char name [MAX_STRING_LENGTH];
  struct CFNode* method_list_head;
  struct CSMethodNode* static_method_list_head;
  struct CMNode* member_list_head;
  struct CSMemberNode* static_member_list_head;
  struct CNode* super_class_list_head;
  struct CNode* sub_class_list_head;
  unsigned long line_number;
```

```
  unsigned long ID_number;
  char file_name [MAX_STRING_LENGTH];
};
typedef struct ClassItem CItem;

struct CNode
{
  CItem              item;
  struct CNode*      next;
  struct CNode*      prev;
};
```

Den information om varje klass som sparas i listan är namnet på klassen, en lista på alla metoder som klassen innehåller, två listor för alla datamedlemmar som variabler, pekare och referensvariabler, en lista som sparar all information om alla klasser (superklasser) som den aktuella klassen ärver från och även en lista som innehåller information om klassens alla subklasser. På samma sätt som funktioner sparas radnumret där klassen definieras. Klassen får ett unikt ID-nummer och ett filnamn där klassen är definierad. Elementstrukturen som listan använder för metoder ser ut som nedan:

```
struct ClassFuncItem
{
  char return_type [MAX_STRING_LENGTH];
  char name [MAX_STRING_LENGTH];
  struct PNode* param_head;
  char return_value [MAX_STRING_LENGTH];
  unsigned long return_ID_number;
  unsigned short isVirtual;
  unsigned short accessSpecifier;
  unsigned long ID_number;
  char file_name [MAX_STRING_LENGTH];
  char contents [FUNC_BUFFER_MAX_LENGTH];
```

```
  int func_body_line_number;
};
typedef struct ClassFuncItem CFItem;
```

Strukturen ser ut som för en vanlig funktion men innehåller en flagga som anger om metoden är virtuell (virtual) och en annan flagga som anger om metoden är public, private eller protected. Arv mellan klasser simuleras genom att kopiera över de publika (public) och skyddade (protected) metoder och datamedlemmar som basklassen har till subklassen. Med hjälp av flaggan `isVirtual` och en analys av både statisk och dynamisk datatyp för instansen av klassen implementeras stöd för polymorfism. Här följer ett exempel på polymorfism:

```
class Fruit
{
  public:
     virtual void print()
     { std::cout << ”Det här är en frukt.” << std::endl; }
  };

class Apple : public Fruit
{
  public:
    virtual void print()
    { std::cout << ”Det här är ett äpple.” << std::endl; }
};

Fruit* fruit;
fruit = new Apple();
fruit->print();
delete fruit;
```

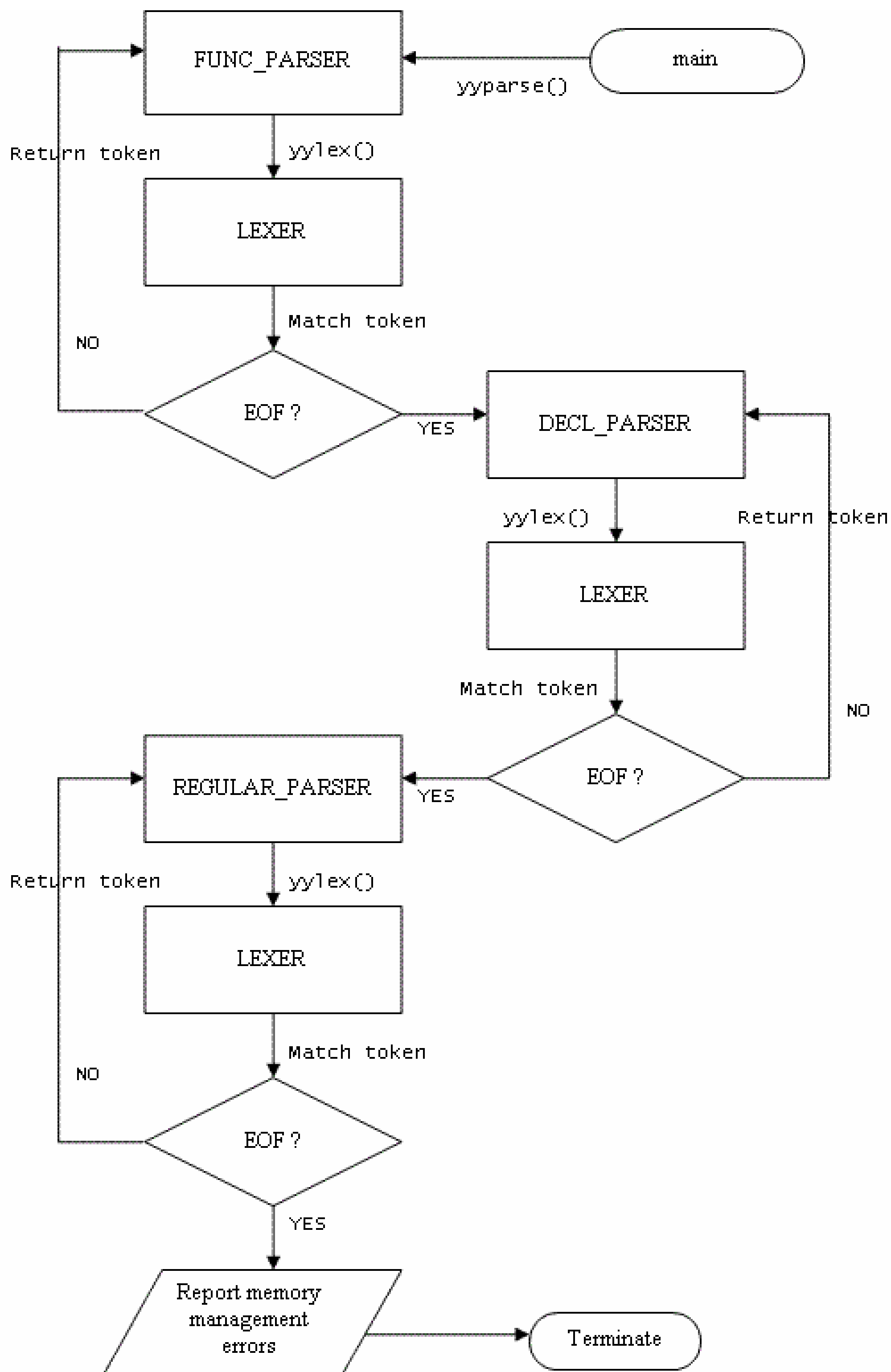
Genom att analysera den statiska datatypen för instansen som är av klassen `Fruit` och den dynamiska datatypen för den som är av klassen `Apple` och samtidigt undersöka flaggan `isVirtual` för metoden print i basklassen kan vi avgöra vilken av metoderna `print()` som ska analyseras efteråt. En analys av en instansiering av ett objekt och ett objekts anrop till dess metoder kan ske genom att först analysera objektets typ genom att leta upp den i klasslistan och sedan spara objektet i variabellistan eller referenslistan beroende på hur det har instansierats. Om objektet har instansierats som ett lokalt objekt i en metod, sparas det i den lokala variabellistan och om det har instansierats genom en dynamisk minnesallokering, sparas objektet i allokeringslistan och namnet samt ID-numret till det objektet sparas i pekaren som pekar på objektet. Eftersom statiska datamedlemmar (klassvariabler) och statiska metoder (klassmetoder) delas av alla instanser av en klass sparas de i separata listor i strukturen. Detta på grund av att dessa medlemmar inte tillhör enskilda objekt.

5.7 Begränsningar i Flex och Bison

Användning av fler listor istället för en symboltabell tycker vi har resulterat till att det ger en klarare överblick över olika beståndsdelar i prototypen samt till en förbättring när det gäller effektivitet på grund av att hela konstruktionen bygger på tillägg, sökning och borttagning av element i listor. Bortsett från vissa komplikationer som begränsningar i Flex [7] och Bison [6] har implementationen av prototypen med hjälp av Flex och Bison varit framgångsrik.

En av begränsningarna var att det inte gick att rapportera radnumret där ett fel inträffade inuti en funktion eftersom prototypen lagrade funktioner i buffertar för senare analys av kod. En annan begränsning var att lexern inte kunde hoppa till olika ställen i källkoden. Vissa situationer krävde detta som till exempel när prototypen parsade ett funktionsanrop och därefter försökte analysera källkoden i funktionskroppen. Som lösning till detta problem blev vi tvungna till att skapa tre olika parsrar `FUNC_PARSER`, `DECL_PARSER`

och `REGULAR_PARSER`. Funktionsparsern börjar parsa källkoden i första tillståndet. Den skannar endast funktioner och lägger till dem i funktionslistan till en senare analys då ett funktionsanrop påträffas av den reguljära parsern som endast analyserar kod i funktionskroppar. All kod som funktionsparsern skannar i funktionskroppen lagras i listelementet för funktionen. Resterande kod som inte ingår i funktioner kommer att ignoreras av funktionsparsern. Denna parser tar även hand om inkluderingar av andra filer med källkod genom att lägga den nuvarande tokenströmmen på en stack. Efter att funktionsparsern har lagt till alla funktioner från den inkluderade filen i funktionslistan byts tillståndet i lexern och deklarationsparsern sätts igång. Deklarationsparsern skannar källkoden från början igen och lägger till globala variabler och globala pekare i olika listor för senare analys av källkoden. När deklarationsparsern når slutet av en inkluderad fil, byter lexern till den gamla tokenströmmen som hade lagts på stacken, lexerns tillstånd byts till ursprungstillståndet och funktionsparsern fortsätter att skanna i den gamla tokenströmmen. Nedan visas en översiktsbild över hur de olika parsrarna anropas:



Figur 5-1 Illustrering av interaktionen mellan lexern och de tre parsrarna.

Deklarationsparsern klarar även att parsa nästlade filinkluderingar med hjälp av stacken. I det tredje tillståndet parsas källkoden av den reguljära parsern som börjar med funktionen main genom att söka efter den i funktionslista. Vid funktionsanrop kommer den aktuella tokenströmmen att läggas på stacken och börja parsa källkoden som finns i den anropade funktionen. När parsningen av den anropade funktionen är färdig byts tokenströmmen till den föregående tokenströmmen som lades på stacken och parsningen fortsätter där den senast avbröts. Nedan följer ett exempel på hur vi har löst detta genom att använda Flex till att spara och skifta mellan olika tokenströmmar:

```
TOKEN_F_INCLUDE  #include

{TOKEN_F_INCLUDE}
{
  BEGIN(include_state);
}
```

När ett include-token matchas i källkoden som skannas går parsern in i ett tillstånd som heter include_state. Då kommer endast de tokens som är definierade för tillståndet include_state att matchas. I Flex anges tillstånden med syntaxen <tillståndsnamn>.

```
<include_state>[ \t]+\"          {}
```

I denna åtgärd matchas ett citationstecken efter nyckelordet include. Nästa token som måste matchas är filnamnet i include-satsen:

```
<include_state>[a-zA-Z0-9_\-\.]+
{
  if(include_stack_ptr >= MAX_INCLUDE_DEPTH)
  {
    fprintf(stderr, "Error: Too many nested file includes.\n");
    yyterminate();
```

```
  }
  include_stack[include_stack_ptr++] = YY_CURRENT_BUFFER;
  yyin = fopen(yytext, "r");
  if(!yyin)
  {
    fprintf(stderr, "Cannot find the file \'%s\'.\n", yytext);
    yyterminate();
  }
  yy_switch_to_buffer(yy_create_buffer(yyin, YY_BUF_SIZE));
  yylineno = 0;
}
```

I ovanstående åtgärd matchas filnamnet från en include-sats under tillståndet include_state. Koden i åtgärden kollar först att stacken inte är full och sedan läggs den nuvarande tokenströmmen på den stacken. Efteråt öppnas den fil som ska inkluderas och lexern byter från nuvarande tokenström till den nya inkluderade filens tokenström istället.

```
<include_state>\"\n
{
  BEGIN(INITIAL);
}
```

I denna åtgärd matchas det avslutande citationstecknet och radslutstecknet. Parserns tillstånd återställs till det tidigare tillståndet `FUNC_PARSER` och den inkluderade filen parsas.

```
<INITIAL><<EOF>>
{
  if(--include_stack_ptr < 0)
  {
    BEGIN(regular_scanner);
    yy_delete_buffer(YY_CURRENT_BUFFER);
```

```
    yyin = fopen(_current_file_name, "r");
    if(!yyin)
    {
      fprintf(stderr, "Cannot find the file \'%s\'.\n", yytext);
      yyterminate();
    }
    yy_switch_to_buffer(yy_create_buffer(yyin, YY_BUF_SIZE));
    yylineno = 0;
    return TOKEN_F_EOF;
  }
  else
  {
    yy_delete_buffer(YY_CURRENT_BUFFER);
    yy_switch_to_buffer(include_stack[include_stack_ptr]);
    yylineno = 0;
    return TOKEN_F_EOF;
  }
}
```

När parsern `FUNC_PARSER` når slutet av filen kollar lexern om det finns någon tokenström (från någon inkluderad fil) kvar på stacken. Om det finns någon tokenström kar på stacken, parsas de kvarvarande tokenströmmarna av `FUNC_PARSER` respektive `DECL_PARSER` och tas bort en efter en från stacken när de har parsats färdigt. Radnummervariabeln `yylineno` nollställs och lexern returnerar då ett filslutstoken till parsern för att indikera slutet på filen.

Lösningen med olika parsrar och buffring av funktionskroppar ledde till ett annat problem. Problemet var att radnumren inte stämde på grund av buffring av vissa delar av källkod. Detta löste vi genom att låta lexern istället för parsern skriva all källkod som finns i funktioner till bufferten och även låta den räkna upp alla radavslutstecken för att hålla reda på det korrekta radnumret och sedan spara radnumret där funktionen börjar sin

kropp tillsammans med resten av informationen om funktionen som ett element i funktionslistan.

6 Tester

Vi kommer att testa vår prototyp i några olika testfall som testar prototypens förmåga att detektera minneshanteringsfel. Testfallen prövar också stödet för de befintliga programkonstruktioner som prototypen har stöd för.

Testfall 1

Detta är ett enkelt testfall som testar prototypens förmåga att detektera dangling pointers och minnesläckage från minneshanteringsfunktionerna malloc, calloc, realloc och free. Det förväntade resultatet från analysen är en minnesläcka på 40 byte, två dangling pointers (`ptr1` och `ptr3`), minneskorruption, läsning från ogiltigt minnesområde och otillåten avallokering för `ptr2`.

```
#include <stdlib.h>

int main()
{

  int a = 10;
  int* ptr1;
  int* ptr2;
  char* ptr3;
  char ett_tecken;

  ptr1 = ( int* ) malloc ( sizeof ( int ) );
  ptr1 = ( int* ) realloc ( ptr1, sizeof ( int ) * 10 );

  ptr2 = ptr1;
  ptr2 = ( int* ) realloc ( ptr2, sizeof ( int ) * 20 );

  ptr3 = ( char* ) calloc ( 40, sizeof ( ett_tecken ) );
```

```
  free(ptr2);
  free(ptr2);

  *ptr2 = 8;

  a = *ptr2;

  ptr2 = NULL;

  return 0;
}
```

Analysen från prototypen gav följande resultat:

Scanning the file 'testfall1.c' for memory errors...

Attempted an incorrect free in 'testfall1.c' at line 21, 'ptr2' does not point to a dynamically allocated memory.
Memory corruption occured for pointer 'ptr2' in file 'testfall1.c' at line number 23.
Attempted to read from invalid memory area for pointer 'ptr2' in file 'testfall1.c' at line 25.

Memory leak summary:
Leaked 40 bytes of memory with value 'UNDEFINED VALUE' allocated in file 'testfall1.c' at line number 18.

Dangling pointers summary:
Dangling pointer 'ptr1' with type 'int*' found in file 'testfall1.c' at line number 12.
Dangling pointer 'ptr3' with type 'char*' found in file 'testfall1.c' at line number 18.

Testfall 2

Detta testfall testar prototypens förmåga att detektera dangling pointers och minnesläckage från minneshanteringsfunktionerna `new` och `delete`. Det förväntade resultatet är en minnesläcka på 40 byte och en dangling pointer (`ptr1`), minneskorruption, läsning från ogiltigt minnesområde och otillåten avallokering för `ptr2`.

```
#include <stdlib.h>

int* allocateMemory (int size);

int main ()
{
   int* ptr1;
   int* ptr2;
   int size = 10;
   int value = 40;

   ptr1 = allocateMemory(size);
   ptr2  = allocateMemory(size * 2);

   delete ptr2;

   *ptr2 = value;

   delete ptr2;

   ptr2 = NULL;

   value = *ptr2;

  return 0;
```

```
}

int* allocateMemory (int size)
{
  return new int [size];
}
```

Analysen från prototypen gav följande resultat:

Scanning the file 'testfall2.c' for memory errors...

Memory corruption occured for pointer 'ptr2' in file 'testfall2.c' at line number 17.
Attempted an incorrect delete in 'testfall2.c' at line 19, 'ptr2' does not point to a dynamically allocated memory.
Attempted to read from invalid memory area for pointer 'ptr2' in file 'testfall2.c' at line 23.

Memory leak summary:
Leaked 40 bytes of memory with value '' allocated in file 'testfall2.c' at line number 30.

Dangling pointers summary:
Dangling pointer 'ptr1' with type 'int*' found in file 'testfall2.c' at line number 12.

Testfall 3

Detta testfall testar prototypens förmåga att detektera minnesläckage och dangling pointers från olika funktioner och en villkorsstyrd sats. Det förväntade resultatet är en minnesläcka på 8 byte, en dangling pointer (`ptr1`), otillåten avallokering för `ptr1` och minneskorruption för `ptr2`. Eftersom prototypen rapporterar minnesläckor som finns i if-satser enbart som risker till minnesläcka kommer den endast att ge varningar om att en minnesläcka kan uppstå i dessa fall.

```
#include <stdlib.h>

double* allocateMemory (int size);

int main ()
{
   double* ptr1;
   double* ptr2;
   int size = 10;
   int value = 100;

   ptr1 = allocateMemory(0);

   delete ptr1;

   ptr2 = allocateMemory(size);

   delete ptr1;
   delete ptr2;

   ptr2 = NULL;
   *ptr2 = value;

  return 0;
}

double* allocateMemory (int size)
{
  if(size == 0)
  {
    return new double();
  }
  else if(size > 0)
```

```
  {
    return new double [size];
  }
}
```

Analysen från prototypen gav följande resultat:

Scanning the file 'testfall3.c' for memory errors...

Warning: memory was dynamically allocated in an if-clause at line 31.
Warning: memory was dynamically allocated in an if-clause at line 35.
Attempted an incorrect delete in 'testfall3.c' at line 14, 'ptr1' does not point to a dynamically allocated memory.
Warning: memory was dynamically allocated in an if-clause at line 31.
Warning: memory was dynamically released in an if-clause at line 26.
Warning: memory was dynamically allocated in an if-clause at line 35.
Attempted an incorrect delete in 'testfall3.c' at line 18, 'ptr1' does not point to a dynamically allocated memory.
Memory corruption occured for pointer 'ptr2' in file 'testfall3.c' at line number 22.

Memory leak summary:
Leaked 8 bytes of memory with value 'UNDEFINED VALUE' allocated in file 'testfall3.c' at line number 31.
Leaked 8 bytes of memory with value 'UNDEFINED VALUE' allocated in file 'testfall3.c' at line number 31.
Leaked 80 bytes of memory with value 'UNDEFINED VALUE' allocated in file 'testfall3.c' at line number 35.

Dangling pointers summary:
Dangling pointer 'ptr1' with type 'double*' found in file 'testfall3.c' at line number 12.

Testfall 4

Det sista testfallet testar prototypens stöd av programkonstruktioner i C/C++ och prototypens förmåga att analysera och rapportera källkod korrekt. Källkoden i testfall 4 har tagits från en stack som från början var tänkt att användas i prototypen. Inga ändringar har gjorts i koden för att anpassa den efter prototypen på något sätt. Det förväntade resultatet är några dangling pointers men inga övriga minneshanteringsfel rapporteras.

bufferStack.h

```
#ifndef _BUFFER_STACK_H_
#define _BUFFER_STACK_H_

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

#define True              1
#define False             0

struct BNode {
  char* buffer;
  struct BNode* next;
};

static struct BNode* b_create_node (struct BNode* n, const char*
buffer);
struct BNode* b_destroy_stack (struct BNode* head);
struct BNode* b_push (const char* buffer, struct BNode* head);
struct BNode* b_pop (struct BNode* head);
char* b_top (struct BNode* head);
```

```
int b_is_empty (struct BNode* head);
void b_print_stack (struct BNode* head);

#endif
```

bufferStack.c

```
#include "bufferStack.h"

static struct BNode* b_create_node (struct BNode* n, const char*
buffer)
{
  struct BNode* new_node = malloc(sizeof(struct BNode));

  new_node->next = n;
  new_node->buffer =
(char*)malloc(sizeof(char)*(strlen(buffer)+1));
  strcpy(new_node->buffer, buffer);

  return new_node;
}

struct BNode* b_destroy_stack (struct BNode* head)
{
  struct BNode* temp;

  while(head != 0)
  {
    temp = head;
    free(temp->buffer);
    head = head->next;
    free(temp);
```

```
  }
  return 0;
}

struct BNode* b_push (const char* buffer, struct BNode* head)
{
  struct BNode* new_node = b_create_node(0, buffer);
  struct BNode* temp = head;

  if(b_is_empty(head) == False)
  {
    while(temp->next != 0)
      temp = temp->next;

    temp->next = new_node;
  }
  else
    head = new_node;

  return head;
}

struct BNode* b_pop (struct BNode* head)
{
  struct BNode* temp = head;

  if(b_is_empty(head) == False)
  {
    if(temp->next != 0)
    {
      while(temp->next->next != 0)
        temp = temp->next;
```

```
      free(temp->next->buffer);
      free(temp->next);
      temp->next = 0;
    }
    else
    {
      free(temp->buffer);
      free(temp);
      temp = 0;
    }
    return temp;
  }
}

char* b_top (struct BNode* head)
{
  struct BNode* temp = head;

  if(b_is_empty(head) == False)
  {
    while(temp->next != 0)
      temp = temp->next;

    return temp->buffer;
  }
  else
    return "";
}

int b_is_empty (struct BNode* head)
{
  if(head == 0)
    return True;
```

```
  else
    return False;
}

void b_print_stack (struct BNode* head)
{
  struct BNode* temp = head;

  if(b_is_empty(head) == False)
  {
    printf("Stack: \n");
    while(temp != 0)
    {
      printf("%s\n", temp->buffer);
      temp = temp->next;
    }
    printf("\n");
  }
  else
  {
    printf("Stack is empty. \n");
  }
}

#include “bufferStack.h”

int main()
{
  struct BNode* stack_head;
  char str[] = “Testfall 1”;
  char str[] = “Testfall 2”;
  char str[] = “Testfall 3”;
  char str[] = “Testfall 4”;
```

```
  stack_head = b_push(str1, stack_head);
  stack_head = b_push(str2, stack_head);
  stack_head = b_push(str3, stack_head);
  stack_head = b_push(str4, stack_head);

  stack_head = b_pop(stack_head);

  b_destroy_stack(stack_head);
  stack_head = 0;

  return 0;
}
```

Tyvärr hann vi inte få ett komplett resultat från det sista testfallet. Prototypen klarade inte att parsa filen helt. Det är på grund av att vissa programkonstruktioner stöds inte och prototypen måste kunna parsa igenom hela källkoden för att kunna få en rapportering av minneshanteringsfel. Därför kunde vi inte ha med någon rapport här.

7 Resultat och rekommendationer

Målet var att skapa en prototyp av ett verktyg som kunde detektera minneshanteringsfel så som minnesläckage, felaktiga avallokeringar av minne, dangling pointers, samt läsning från och skrivning till ogiltiga minnesområden i källkod skriven i C/C++. Detta för att göra utvecklingen av applikationer säkrare i C/C++, det vill säga att applikationerna ska ha så få minneshanteringsfel som möjligt redan innan kompilering.

Under projektets gång har vi mött många motgångar på grund av bland annat tidsbrist och en del implementationsproblem. Projektet blev mycket större än vad vi väntade oss. Trots att vi har arbetat med projektet mer än den avsatta tiden för examensarbetet och har skrivit drygt 7000 rader kod har vi ändå inte hunnit klart med att implementera fullständigt stöd för språken C och C++ i prototypen. På grund av tidsbristen var vi tvungna att göra avkall på vissa programkonstruktioner som stöd för klasser och objekt i C++, komplett stöd för typomvandling i både C och C++, stöd för switch/case- och for-satser och några preprocessordirektiv. Men vi känner att vi har lyckats med att skapa en bra grund för vidareutveckling av prototypen då all kod för kommunikation mellan lexern och de olika parsrarna är färdig. Vi har varit tvungna att skriva om delar av prototypen flera gånger på grund av vissa begränsningar som finns i kommunikationen mellan verktygen Flex [7] och Bison [6], vilka vi upptäckte under projektets gång. Det svåraste med att implementera prototypen har varit att kunna få den att göra en korrekt semantisk tolkning av källkoden. Det var också svårt att hitta de syntaktiska fall i grammatiken som C och C++ har, vilka krävdes för en korrekt parsning. Vid analys av källkod som finns i standardbibliotek finns ofta bara funktionsdeklarationer och klassdeklarationer tillgängliga medan definitionerna till dessa finns redan i färdigkompilerade biblioteksfiler som till exempel lib-filer. I detta fall begränsas analysen till deklarationen och då kan bara parametrar och returvärdet analyseras. Problem kan även uppstå när värdet till en variabel bestäms under exekvering och denna variabel innehåller då ett värde som anger till exempel hur många minnesallokeringar som ska göras i en applikation. I det fallet går

det inte att statiskt ta reda på antal minnesallokeringar och därför inte heller hur många avallokeringar som behöver göras. På grund av detta är det omöjligt att detektera minnesläckor och dangling pointers statiskt i dessa fall.

I efterhand känns implementationen av de sju listorna som vi använder i prototypen för att lagra informationen om lokala variabler, globala variabler, lokala pekare, globala pekare, funktioner, funktionsparametrar och minnesallokeringar något krånglig att hantera. En symboltabell för dessa hade varit enklare att implementera och vi skulle bara ha haft en enda datatyp att hålla reda på. En sådan tabell kan lagra all information som behövs för den semantiska analysen.
För att kunna göra en fullständig analys av källkod måste prototypen kompletteras med de språkkonstruktioner som vi har varit tvungna att göra avkall på. Om vi hade mer tid skulle vi implementera stöd för dessa språkkonstruktioner.

8 Summering av projektet

Trots att det har varit ett väldigt krävande projekt har det varit väldigt roligt att arbeta med ämnet minneshantering. Projektet har varit givande och lärorikt i det avseendet att vi har lärt oss mycket om minneshantering och verktygen Flex [7] och Bison [6]. Vi känner också att uppsatsen har gett oss bättre kunskaper i att uttrycka oss på ett vetenskapligt sätt i skrift. Vi tycker att ämnet är väldigt intressant och vår ambition är att vidareutveckla prototypen genom att lägga till stöd för resterande funktionalitet i språken C och C++.

Uppgiften att utföra statisk detektering av minnesläckage som fungerar acceptabelt är mycket komplicerad att lösa. Eftersom det finns fall där risken att minnesläckage inträffar inte är 100% som till exempel där exekveringsvägen inte kan fastställas kan endast en varning för minnesläckage rapporteras. Detta kan vara förvirrande då man kan tro att det är något fel i källkoden som analyserats fast i själva verket är källkoden helt korrekt. Detta problem uppstår i if-satser och switch-satser. Om analysen istället är dynamisk, finns det ett annat problem, vilket är konstruktion av testfall som täcker alla exekveringsvägar i källkoden.

Dynamisk detektering av minneshanteringsfel fungerar bättre än statisk detektering när det gäller detektering av minneshanteringsfel med 100% säkerhet i villkorsstyrda satser. Däremot kan ett verktyg som utför dynamisk detektering utsättas för korruption av de applikationer som testas. Minnet där verktyget lagrar informationen om minneshanteringsfelen är åtkomlig i många fall av den testade applikationen och detta innebär att det minnet kan skrivas över eller raderas av applikationen. Detta problem existerar inte under statisk detektering. Vissa applikationer kan vara svårt att testa efter minneshanteringsfel som till exempel om valet av exekveringsväg styrs av läsningar från hårdvara.

Referenser

[1] DJGPP
http://www.delorie.com/djgpp/, 2006-05-15

[2] Nate Eldredge – Yet Another Malloc Debugger
http://www.cs.hmc.edu/~nate/yamd/, 2006-05-15

[3] David Evans (Secure Programming Group, project leader and the primary developer of Splint)
http://www.splint.org/, 2006-05-12

[4] David Evans – Static Detection of Dynamic Memory Errors (LCLint)
http://www.cs.virginia.edu/~evans/pubs/pldi96.pdf, 2006-05-12

[5] David Evans, David Larochelle – Improving Security Using Extensible Lightweight Static Analysis
http://www.cs.virginia.edu/~evans/pubs/ieeesoftware-revised.pdf, 2006-05-12

[6] Free Software Foundation – Bison
http://www.gnu.org/software/bison/bison.html, 2006-05-15

[7] Free Software Foundation – Flex
http://www.gnu.org/software/flex/flex.html, 2006-05-15

[8] Free Software Foundation – GNU C
http://gcc.gnu.org/, 2006-05-20

[9] Etienne M. Gagnon, Ben Menking, Mariusz Nowostawski, Komivi Kevin Agbakpem, Kis Gergely – SableCC
http://sablecc.org/, 2006-05-15

[10] IBM – IBM Rational Purify
http://www.pts.com/purify.cfm?det=y, 2006-05-12

[11] JavaCC
https://javacc.dev.java.net/, 2006-05-15

[12] Johan Lindh – MEMWATCH
http://www.linkdata.se/sourcecode.html, 2006-05-12

[13] Lint

http://linux.about.com/cs/linux101/g/lint.htm, 2006-05-12

[14] Dan Moulding - Visual Leak Detector

http://www.codeproject.com/tools/visualleakdetector.asp, 2006-05-15

[15] Terence Parr – ANTLR

http://www.antlr.org/, 2006-05-15

[16] Cheah Ui Poh – CMemLeak

http://www.codeguru.com/Cpp/misc/misc/memory/article.php/c3745/, 2006-05-15

[17] Charles W Sandmann – CWSDPMI (Charles W Sandmann DOS Protected Mode Interface)

http://clio.rice.edu/cwsdpmi/, 2006-05-20

[18] The Open Group – Lex

http://www.opengroup.org/onlinepubs/007908799/xcu/lex.html, 2006-05-20

[19] The Open Group – Yacc (Yet Another Compiler Compiler)

http://www.opengroup.org/pubs/online/7908799/xcu/yacc.html, 2006-05-20

[20] Bill Venners – Java's garbage-collected heap

http://www.javaworld.com/javaworld/jw-08-1996/jw-08-gc.html, 2006-05-12

[21] Robert Walker – MemLeakCheck

http://www.robertinventor.com/memleakcheck/index.htm, 2006-05-15

Bilagor

B.1 – Användarbeskrivning

Verktyget består av en exekverbar fil som heter mls. För att starta verktyget ska man skriva verktygets namn i ett konsolfönster och ange de filer som ska analyseras som parametrar.

Syntax:

```
./mls källkodsfil_1.c källkodsfil_2.c ... källkodsfil_n.c
```

Verktyget kommer att rapportera de fel som har detekterats i konsolfönstret.
För att rapportera till fil ska man ange flaggan –f och det önskade namnet på rapportfilen som rapporten ska skrivas till, till exempel:

```
./mls main.c functions.c –f rapportfil.txt
```

Ett exempel på en rapport kan se ut så här:

```
Memory errors reported:
Invalid delete in file 'accounts.c' at line number 56
Memory corruption occurs in file 'accounts.c' at line number 59

Memory leaks reported:
Leaked 80 bytes of memory in file 'accounts.c' at line number 120

Dangling pointers reported:
Dangling pointer 'ptr' found in file 'accounts.c' at line number
120
```

Observera att verktyget för närvarande inte kan göra en komplett detektering av minneshanteringsfel i C- och C++-kod eftersom prototypen inte har stöd för vissa konstruktioner som static-deklarationer, klasser, objekt, strukturer, egna typer med typedef och funktionspekare.

B.2 - Installation

För att kompilera prototypen finns det en färdig make-fil som innehåller alla filer som ska finnas med vid kompilering. Användaren behöver bara skriva make för att kompilera prototypen från källkoden.

Prototypen kan installeras genom att kopiera den exekverbara filen mls till den värddator som ska köra prototypen. Eventuella sökvägar som behöver ställas in måste användaren se till själv.

B.3 – Källkod från prototypen

```
F_ASSIGN_LIST:
    TOKEN_F_NAME TOKEN_F_ASSIGN F_ASSIGN_LIST
  | TOKEN_F_STAR TOKEN_F_NAME TOKEN_F_ASSIGN F_ASSIGN_LIST
  | TOKEN_F_NAME TOKEN_F_ASSIGN F_EXPR TOKEN_F_SEMICOLON
  | TOKEN_F_STAR TOKEN_F_NAME TOKEN_F_ASSIGN TOKEN_F_NAME
TOKEN_F_SEMICOLON
  | TOKEN_F_STAR TOKEN_F_NAME TOKEN_F_ASSIGN TOKEN_F_STAR
TOKEN_F_NAME TOKEN_F_SEMICOLON
  | TOKEN_F_NAME TOKEN_F_ASSIGN TOKEN_F_STAR TOKEN_F_NAME
TOKEN_F_SEMICOLON
  | TOKEN_F_NAME TOKEN_F_ASSIGN TOKEN_F_AMPERSAND TOKEN_F_NAME
TOKEN_F_SEMICOLON
  | TOKEN_F_STAR TOKEN_F_NAME TOKEN_F_ASSIGN TOKEN_F_DIGIT
TOKEN_F_SEMICOLON
```

```
  | TOKEN_F_NAME TOKEN_F_ASSIGN F_FUNC_CALL_STAT
  | TOKEN_F_STAR TOKEN_F_NAME TOKEN_F_ASSIGN F_FUNC_CALL_STAT
  ;

D_ASSIGN_LIST:
    TOKEN_D_NAME TOKEN_D_ASSIGN D_ASSIGN_LIST
  | TOKEN_D_STAR TOKEN_D_NAME TOKEN_D_ASSIGN D_ASSIGN_LIST
  | TOKEN_D_NAME TOKEN_D_ASSIGN D_EXPR TOKEN_D_SEMICOLON
  | TOKEN_D_STAR TOKEN_D_NAME TOKEN_D_ASSIGN D_EXPR
TOKEN_D_SEMICOLON
  | TOKEN_D_STAR TOKEN_D_NAME TOKEN_D_ASSIGN TOKEN_D_STAR
TOKEN_D_NAME TOKEN_D_SEMICOLON
  | TOKEN_D_NAME TOKEN_D_ASSIGN TOKEN_D_STAR TOKEN_D_NAME
TOKEN_D_SEMICOLON
  | TOKEN_D_NAME TOKEN_D_ASSIGN TOKEN_D_AMPERSAND TOKEN_D_NAME
TOKEN_D_SEMICOLON
  | TOKEN_D_NAME TOKEN_D_ASSIGN D_FUNC_CALL_STAT
  | TOKEN_D_STAR TOKEN_D_NAME TOKEN_D_ASSIGN D_FUNC_CALL_STAT
  ;

R_ASSIGN_LIST:
  TOKEN_R_NAME TOKEN_R_ASSIGN R_ASSIGN_LIST
  {
    RItem _gotten_l_RItem;
    RItem _gotten_r_RItem;
    VItem _gotten_l_VItem;
    VItem _gotten_r_VItem;
    AItem _gotten_r_AItem;
    if(r_item_exists_by_name($1, _ref_list_head) == True)
    {
      _gotten_l_RItem = r_get_item_by_name($1, _ref_list_head);
      if(v_item_exists_by_ID($3, _var_list_head) == True &&
gotten_l_RItem.refers_to_null == False)
```

```
      {
        _gotten_r_VItem = v_get_item_by_ID($3, _var_list_head);
        r_modify_item($1, yylineno, _gotten_r_VItem.ID_number,
      False, _ref_list_head);
       $$ = _gotten_r_VItem.ID_number;
      }
      else if(a_item_exists_by_ID($3, _alloc_list_head) == True
&& _gotten_l_RItem.refers_to_null == False)
      {
        _gotten_r_AItem = a_get_item_by_ID($3, _alloc_list_head);
        r_modify_item($1, yylineno, _gotten_r_AItem.ID_number,
False, _ref_list_head);
        $$ = _gotten_r_AItem.ID_number;
      }
      else
      {
        printf("Memory corruption occured for pointer \'%s\' in
file \'%s\' at line number %d.\n", _gotten_l_RItem.name,
_gotten_l_RItem.file_name, yylineno);
      }
   }
   else if(v_item_exists_by_name($1, _var_list_head))
   {
      _gotten_l_VItem = v_get_item_by_name($1, _var_list_head);
      if(v_item_exists_by_ID($3, _var_list_head) == True)
      {
        _gotten_r_VItem = v_get_item_by_ID($3, _var_list_head);
        v_modify_item($1, _gotten_r_VItem.value, _var_list_head);
       $$ = _gotten_r_VItem.ID_number;
      }
      else if(a_item_exists_by_ID($3, _alloc_list_head) == True)
      {
        _gotten_r_AItem = a_get_item_by_ID($3, _alloc_list_head);
```

```
        v_modify_item($1, _gotten_r_AItem.value, _var_list_head);
        $$ = _gotten_r_AItem.ID_number;
      }
    }
  }

| TOKEN_R_STAR TOKEN_R_NAME TOKEN_R_ASSIGN R_ASSIGN_LIST
  {
    RItem _gotten_RItem;
    VItem _gotten_l_VItem;
    VItem _gotten_r_VItem;
    AItem _gotten_l_AItem;
    AItem _gotten_r_AItem;

    // Om LHS finns i referenslistan...
    if(r_item_exists_by_name($2, _ref_list_head) == True)
    {
      _gotten_RItem = r_get_item_by_name($2, _ref_list_head);

      // Om LHS refererar till en variabel...
      if(v_item_exists_by_ID(_gotten_RItem.ID_number,
_var_list_head) == True && _gotten_RItem.refers_to_null == False)
      {
        _gotten_l_VItem =
    v_get_item_by_ID(_gotten_RItem.ID_number, _var_list_head);

        // Om RHS finns i variabellistan...
        if(v_item_exists_by_ID($4, _var_list_head) == True)
        {
          _gotten_r_VItem = v_get_item_by_ID($4, _var_list_head);
          v_modify_item(_gotten_l_VItem.name,
    _gotten_r_VItem.value, _var_list_head);
          $$ = _gotten_l_VItem.ID_number;
```

```
        }
        // Om RHS finns i allokeringslistan...
        else if(a_item_exists_by_ID($4, _alloc_list_head) == True)
        {
          _gotten_r_AItem = a_get_item_by_ID($4,
      _alloc_list_head);
          a_modify_item_by_ID(_gotten_l_VItem.ID_number,
      _gotten_r_AItem.value, _gotten_r_AItem.number_of_bytes,
      _alloc_list_head);
         $$ = _gotten_l_VItem.ID_number;
        }
        else
        {
          printf("Attempted to read from invalid memory area for
      pointer \'%s\' in file \'%s\' at line %d.\n",
      _gotten_RItem.name, _gotten_RItem.file_name, yylineno);
        }
      }
      // Om LHS refererar till en allokering...
      else if(a_item_exists_by_ID(_gotten_RItem.ID_number,
  _alloc_list_head) == True && _gotten_RItem.refers_to_null ==
  False)
      {
        _gotten_l_AItem = a_get_item_by_ID(_gotten_RItem.ID_number,
  _alloc_list_head);

       // Om RHS finns i variabellistan...
       if(v_item_exists_by_ID($4, _var_list_head) == True)
       {
         _gotten_r_VItem = v_get_item_by_ID($4, _var_list_head);
         a_modify_item_by_ID(_gotten_l_AItem.ID_number,
      _gotten_r_VItem.value, _gotten_l_AItem.number_of_bytes,
      _alloc_list_head);
```

```
        $$ = _gotten_l_AItem.ID_number;
      }
      // Om RHS finns i allokeringslistan...
      else if(a_item_exists_by_ID($4, _alloc_list_head) == True)
      {
        _gotten_r_AItem = a_get_item_by_ID($4, _alloc_list_head);
       a_modify_item_by_ID(_gotten_l_AItem.ID_number,
     _gotten_r_AItem.value, _gotten_l_AItem.number_of_bytes,
     _alloc_list_head);
        $$ = _gotten_l_AItem.ID_number;
      }
      else
      {
        printf("Attempted to read from invalid memory area for
     pointer \'%s\' in file \'%s\' at line %d.\n",
     _gotten_RItem.name, _gotten_RItem.file_name, yylineno);
      }
    }
    else
    {
      printf("Memory corruption occured for pointer \'%s\' in file
\'%s\' at line number %d.\n", _gotten_RItem.name,
_gotten_RItem.file_name, yylineno);
    }
  }
}

| TOKEN_R_NAME TOKEN_R_ASSIGN R_EXPR TOKEN_R_SEMICOLON
  {
    RItem _gotten_l_RItem;
    RItem _gotten_r_RItem;
    VItem _gotten_l_VItem;
    VItem _gotten_r_VItem;
```

```
    if(r_item_exists_by_name($1, _ref_list_head) == True)
    {
      _gotten_l_RItem = r_get_item_by_name($1, _ref_list_head);
      if(r_item_exists_by_ID($3, _ref_list_head) == True)
      {
        _gotten_r_RItem = r_get_item_by_ID($3, _ref_list_head);
        r_modify_item(_gotten_l_RItem.name, yylineno,
_gotten_r_RItem.ID_number, _gotten_r_RItem.refers_to_null,
_ref_list_head);
      }
      else
      {
        r_modify_item(_gotten_l_RItem.name, yylineno,
_gotten_l_RItem.ID_number, True, _ref_list_head);
        $$ = _gotten_l_RItem.ID_number;
      }
    }
    else if(v_item_exists_by_name($1, _var_list_head) == True)
    {
      _gotten_l_VItem = v_get_item_by_name($1, _var_list_head);
      if(v_item_exists_by_name($3, _var_list_head) == True)
      {
        _gotten_r_VItem = v_get_item_by_name($3, _var_list_head);
        v_modify_item(_gotten_l_VItem.name, _gotten_r_VItem.value,
_var_list_head);
      }
      else
      {
        v_modify_item(_gotten_l_VItem.name, $3, _var_list_head);
      }
      $$ = _gotten_l_VItem.ID_number;
    }
```

```
 }

| TOKEN_R_STAR TOKEN_R_NAME TOKEN_R_ASSIGN R_EXPR
TOKEN_R_SEMICOLON
{
  RItem _gotten_RItem;
  VItem _gotten_l_VItem;
  VItem _gotten_r_VItem;
  AItem _gotten_l_AItem;

  if(r_item_exists_by_name($2, _ref_list_head) == True)
  {
    _gotten_RItem = r_get_item_by_name($2, _ref_list_head);

    if(v_item_exists_by_ID(_gotten_RItem.ID_number,
_var_list_head) == True && _gotten_RItem.refers_to_null == False)
    {
      _gotten_l_VItem = v_get_item_by_ID(_gotten_RItem.ID_number,
_var_list_head);
      if(v_item_exists_by_name($4, _var_list_head) == True)
      {
        _gotten_r_VItem = v_get_item_by_name($4, _var_list_head);
        v_modify_item(_gotten_l_VItem.name,
_gotten_r_VItem.value, _var_list_head);
      }
      else
      {
        v_modify_item(_gotten_l_VItem.name, $4, _var_list_head);
      }
      $$ = _gotten_l_VItem.ID_number;
    }
```

```
   else if(a_item_exists_by_ID(_gotten_RItem.ID_number,
_alloc_list_head) == True && _gotten_RItem.refers_to_null ==
False)
   {
     _gotten_l_AItem = a_get_item_by_ID(_gotten_RItem.ID_number,
_alloc_list_head);
      if(v_item_exists_by_name($4, _var_list_head) == True)
      {
        _gotten_r_VItem = v_get_item_by_name($4, _var_list_head);
        a_modify_item_by_ID(_gotten_l_AItem.ID_number,
_gotten_r_VItem.value, _gotten_l_AItem.number_of_bytes,
_alloc_list_head);
      }
      else
      {
       a_modify_item_by_ID(_gotten_l_AItem.ID_number, $4,
_gotten_l_AItem.number_of_bytes, _alloc_list_head);
      }
      $$ = _gotten_l_AItem.ID_number;
   }
   else
   {
      printf("Memory corruption occured for pointer \'%s\' in
file \'%s\' at line number %d.\n", _gotten_RItem.name,
_gotten_RItem.file_name, yylineno);
   }
 }
}

| TOKEN_R_STAR TOKEN_R_NAME TOKEN_R_ASSIGN TOKEN_R_STAR
TOKEN_R_NAME TOKEN_R_SEMICOLON
 {
   RItem _gotten_l_RItem;
```

```
   RItem _gotten_r_RItem;
   VItem _gotten_l_VItem;
   VItem _gotten_r_VItem;
   AItem _gotten_l_AItem;
   AItem _gotten_r_AItem;

   // Om LHS finns i referenslistan...
   if(r_item_exists_by_name($2, _ref_list_head) == True)
   {
      _gotten_l_RItem = r_get_item_by_name($2, _ref_list_head);
     // Om LHS refererar till en variabel...
     if(v_item_exists_by_ID(_gotten_l_RItem.ID_number,
_var_list_head) == True && _gotten_l_RItem.refers_to_null ==
False)
     {
       _gotten_l_VItem =
v_get_item_by_ID(_gotten_l_RItem.ID_number, _var_list_head);
       // Om RHS finns i referenslistan...
      if(r_item_exists_by_name($5, _ref_list_head) == True)
      {
        _gotten_r_RItem = r_get_item_by_name($5, _ref_list_head);

        // Om RHS refererar till en variabel...
        if(v_item_exists_by_ID(_gotten_r_RItem.ID_number,
_var_list_head) == True && _gotten_r_RItem.refers_to_null ==
False)
        {
          _gotten_r_VItem =
v_get_item_by_ID(_gotten_r_RItem.ID_number, _var_list_head);
          v_modify_item(_gotten_l_VItem.name,
_gotten_r_VItem.value, _var_list_head);
          $$ = _gotten_l_VItem.ID_number;
        }
```

```
        // Om RHS refererar till en allokering...
        else if(a_item_exists_by_ID(_gotten_r_RItem.ID_number,
_alloc_list_head) == True && _gotten_r_RItem.refers_to_null ==
False)
        {
          _gotten_r_AItem =
a_get_item_by_ID(_gotten_r_RItem.ID_number, _alloc_list_head);
          a_modify_item_by_ID(_gotten_l_VItem.ID_number,
_gotten_r_AItem.value, _gotten_l_AItem.number_of_bytes,
_alloc_list_head);
          $$ = _gotten_l_VItem.ID_number;
        }
        else
        {
          printf("Attempted to read from invalid memory area for
pointer \'%s\' in file \'%s\' at line %d.\n",
_gotten_l_RItem.name, _gotten_l_RItem.file_name, yylineno);
        }
      }
    }
    // Om LHS refererar till en allokering...
    else if(a_item_exists_by_ID(_gotten_l_RItem.ID_number,
_alloc_list_head) == True && _gotten_l_RItem.refers_to_null ==
False)
    {
      _gotten_l_AItem =
a_get_item_by_ID(_gotten_l_RItem.ID_number, _alloc_list_head);

     // Om RHS finns i referenslistan...
     if(r_item_exists_by_name($5, _ref_list_head) == True)
     {
       _gotten_r_RItem = r_get_item_by_name($5, _ref_list_head);
```

```
     // Om RHS refererar till en variabel...

     if(v_item_exists_by_ID(_gotten_r_RItem.ID_number,
_var_list_head) == True && _gotten_r_RItem.refers_to_null ==
False)
     {
       _gotten_r_VItem =
v_get_item_by_ID(_gotten_r_RItem.ID_number, _var_list_head);
       a_modify_item_by_ID(_gotten_l_AItem.ID_number,
_gotten_r_VItem.value, _gotten_l_AItem.number_of_bytes,
_alloc_list_head);
      $$ = _gotten_l_AItem.ID_number;
     }
     // Om RHS refererar till en allokering...
     else if(a_item_exists_by_ID(_gotten_r_RItem.ID_number,
_alloc_list_head) == True && _gotten_r_RItem.refers_to_null ==
False)
     {
       _gotten_r_AItem =
a_get_item_by_ID(_gotten_r_RItem.ID_number, _alloc_list_head);
       a_modify_item_by_ID(_gotten_l_AItem.ID_number,
_gotten_r_AItem.value, _gotten_l_AItem.number_of_bytes,
_alloc_list_head);
      $$ = _gotten_l_AItem.ID_number;
     }
     else
     {
       printf("Attempted to read from invalid memory area for
pointer \'%s\' in file \'%s\' at line %d.\n",
_gotten_l_RItem.name, _gotten_l_RItem.file_name, yylineno);
     }
    }
  }
```

```
  else
  {
    printf("Memory corruption occured for pointer \'%s\' in file
\'%s\' at line number %d.\n", _gotten_l_RItem.name,
_gotten_l_RItem.file_name, yylineno);
  }
 }
}

| TOKEN_R_NAME TOKEN_R_ASSIGN TOKEN_R_STAR TOKEN_R_NAME
TOKEN_R_SEMICOLON
 {
   VItem _gotten_l_VItem;
   VItem _gotten_r_VItem;
   RItem _gotten_r_RItem;
   AItem _gotten_r_AItem;

   // Om LHS finns i variabellistan... (vi antar har att det ar
en variabel)
   if(v_item_exists_by_name($1, _var_list_head) == True)
   {
     _gotten_l_VItem = v_get_item_by_name($1, _var_list_head);

     // Om RHS finns i referenslistan...
     if(r_item_exists_by_name($4, _ref_list_head) == True)
      {
        _gotten_r_RItem = r_get_item_by_name($4, _ref_list_head);

        // Om RHS refererar till en variabel...

       if(v_item_exists_by_ID(_gotten_r_RItem.ID_number,
_var_list_head) == True && _gotten_r_RItem.refers_to_null ==
False)
```

```
        {
          _gotten_r_VItem =
v_get_item_by_ID(_gotten_r_RItem.ID_number, _var_list_head);
          v_modify_item(_gotten_l_VItem.name,
_gotten_r_VItem.value, _var_list_head);
          $$ = _gotten_l_VItem.ID_number;
        }
        // Om RHS refererar till en allokering...
        else if(a_item_exists_by_ID(_gotten_r_RItem.ID_number,
_alloc_list_head) == True && _gotten_r_RItem.refers_to_null ==
False)
        {
          _gotten_r_AItem =
a_get_item_by_ID(_gotten_r_RItem.ID_number, _alloc_list_head);
          v_modify_item(_gotten_l_VItem.name,
_gotten_r_AItem.value, _var_list_head);
          $$ = _gotten_l_VItem.ID_number;
        }
        else
        {
          printf("Attempted to read from invalid memory area for
pointer \'%s\' in file \'%s\' at line %d.\n",
_gotten_r_RItem.name, _gotten_r_RItem.file_name, yylineno);
        }
      }
    }
}

| TOKEN_R_NAME TOKEN_R_ASSIGN TOKEN_R_AMPERSAND TOKEN_R_NAME
TOKEN_R_SEMICOLON
{
  RItem _gotten_l_RItem;
  VItem _gotten_r_VItem;
```

```
  if(r_item_exists_by_name($1, _ref_list_head) == True)
  {
    _gotten_l_RItem = r_get_item_by_name($1, _ref_list_head);
    if(v_item_exists_by_name($4, _var_list_head) == True)
    {
      _gotten_r_VItem = v_get_item_by_name($4, _var_list_head);
      r_modify_item(_gotten_l_RItem.name, yylineno,
_gotten_r_VItem.ID_number, False, _ref_list_head);
      $$ = _gotten_l_RItem.ID_number;
    }
  }
}

| TOKEN_R_NAME TOKEN_R_ASSIGN R_FUNC_CALL_STAT
{
  RItem _gotten_RItem;
  VItem _gotten_VItem;
  GRItem _gotten_GRItem;
  GVItem _gotten_GVItem;

  if(v_item_exists_by_name($1, _var_list_head) == True)
  {
    _gotten_VItem = v_get_item_by_name($1, _var_list_head);
    v_modify_item(_gotten_VItem.name, _return_value,
_var_list_head);
 }
 else if(r_item_exists_by_name($1, _ref_list_head) == True)
 {
   _gotten_RItem = r_get_item_by_name($1, _ref_list_head);
   r_modify_item(_gotten_RItem.name,
_saved_line_number[_saved_line_number_ptr-1], _return_ID, False,
_ref_list_head);
 }
```

```
 else if(gv_item_exists_by_name($1, _g_var_list_head) == True)
 {
   _gotten_GVItem = gv_get_item_by_name($1, _g_var_list_head);
   gv_modify_item(_gotten_GVItem.name, _return_value,
_g_var_list_head);
 }
 else if(gr_item_exists_by_name($1, _g_ref_list_head) == True)
 {
   _gotten_GRItem = gr_get_item_by_name($1, _g_ref_list_head);
   gr_modify_item(_gotten_GRItem.name,
_saved_line_number[_saved_line_number_ptr-1], _return_ID, False,
_g_ref_list_head);
 }
}

| TOKEN_R_STAR TOKEN_R_NAME TOKEN_R_ASSIGN R_FUNC_CALL_STAT
{
  RItem _gotten_RItem;
  AItem _gotten_AItem;
  GVItem _gotten_GVItem;
  GRItem _gotten_GRItem;

  if(r_item_exists_by_name($1, _ref_list_head) == True)
  {
     _gotten_RItem = r_get_item_by_name($1, _ref_list_head);
     if(a_item_exists_by_ID(_gotten_RItem.ID_number,
_alloc_list_head) == True)
     {
       _gotten_AItem = a_get_item_by_ID(_gotten_RItem.ID_number,
_alloc_list_head);
       a_modify_item_by_ID(_gotten_AItem.ID_number,
_return_value, _gotten_AItem.number_of_bytes, _alloc_list_head);
     }
```

```
      else if(gv_item_exists_by_ID(_gotten_RItem.ID_number,
_g_var_list_head) == True)
      {
        _gotten_GVItem =
gv_get_item_by_ID(_gotten_RItem.ID_number, _g_var_list_head);
        gv_modify_item(_gotten_GVItem.name, _return_value,
_g_var_list_head);
      }
    }
    else if(gr_item_exists_by_name($1, _g_ref_list_head) == True)
    {
       _gotten_GRItem = gr_get_item_by_name($1,
_g_ref_list_head);
       if(a_item_exists_by_ID(_gotten_GRItem.ID_number,
_alloc_list_head) == True)
       {
         _gotten_AItem =
a_get_item_by_ID(_gotten_GRItem.ID_number, _alloc_list_head);
         a_modify_item_by_ID(_gotten_AItem.ID_number,
_return_value, _gotten_AItem.number_of_bytes, _alloc_list_head);
       }
       else if(gv_item_exists_by_ID(_gotten_GRItem.ID_number,
_g_var_list_head) == True)
       {
        _gotten_GVItem =
gv_get_item_by_ID(_gotten_GRItem.ID_number, _g_var_list_head);
        gv_modify_item(_gotten_GVItem.name, _return_value,
_g_var_list_head);
      }
    }
  }
;
```