An Implicit Garbage Collection Model for C++ Programing Language

A Disseration submitted

By Anubhav Arun Gupta

to

Department of

Computer Science & Engineering

in partial fulfilment of the Requirement for the

Award of the Degree of

Master of Technology

in

Computer Systems

Under the Guidance of

Er. Pushpendra Kumar Pateriya

(May, 2015)

INDIA'S LARGEST UNIVERSITY

LOVELY PROFESSIONAL UNIVERSITY

Transforming Education, Transforming India

School of: COMPUTER SCIENCE AND ENGINEERING

DISSERTATION TOPIC APPROVAL PERFORMA

Name of the Student: Anubhav Arun Gupta	Registration No: 11000527
Batch: 2010-2015	Roll No. RK2005A09
Session: 2014-2015	Parent Section: K2005
Details of Supervisor:	Designation: Assistant Professor
Name: Pushpendra Kumar Pateriya	Qualification: M.Tech
U.ID: 14623	Research Experience: 4 yrs

SPECIALIZATION AREA: System Programming (**pick from list of provided specialization areas by DAA)**

PROPOSED TOPICS

1. ✓ **An Implicit Garbage Collection Model for C++**
2. **Automated Attendance Marking via Image Processing**
3. **Bulk Data Storage Over Paper Using QR Code**

[illegible] 14623.

Signature of Supervisor

PAC Remarks:

Topic No. 1 is approved. Research paper from the work ~~topic~~ is also expected

APPROVAL OF PAC CHAIRPERSON: 11011 [illegible] 18/9/14 **Signature:** **Date:** 18/9/14

*Supervisor should finally encircle one topic out of three proposed topics and put up for approval before Project Approval Committee (PAC)

*Original copy of this format after PAC approval will be retained by the student and must be attached in the Project/Dissertation final report.

*One copy to be submitted to Supervisor.

CERTIFICATE

This is to certify that *Anubhav Arun Gupta* has completed ***Master of Technology*** Dissertation titled "*An Implicit Garbage Collection Model for C++ Programing Language*" under my guidance and supervision. To the best of my knowledge, the present work is the result of his original investigation and study. No part of the Dissertation has ever been submitted for any other degree or diploma.
The dissertation is fit for the submission and the partial fulfilment of the conditions for the award of ***Master of Technology*** Computer Science & Engineering

(Er. Pushpendra Kumar Pateriya)
Computer Science & Engineering Dept.
L.P.U., Phagwara

May, 2015

Abstract

Dynamic Memory Management concept enables memory allocation and deallocation at runtime. Beneficial as it is, but it's not perfect. A naughty anomaly called Memory Leak is observed whenever a program follows Dynamic Memory Management, and that is the topic of concern for this study. The Memory Leak is relevant because its not a logical or syntactical error rather it is a program control error. It grows as the program code grows i.e greater the number of line of code more are the chances of having memory leaks. It's also relevant because few line of code can become cause of memory leak occupying as much as 70-80% of physical memory. The worst part about it is, the operating system doesn't have any jurisdiction over it. The operating system can terminate the process itself but it cannot analyse the memory consumption of process. This study proposes a Garbage Collection Model to provide solution for Memory Leaks. The methodology is divided into three phases where, Source Code Instrumentation is carried out in Phase-1, Modification of compiler and Creation of Garbage Collector is carried out in Phase-2 and finally a driver to use outcomes of above two approaches is created in Phase-3. The result of the devised methodology on the test cases dealing with fixed scenario, have unanimously pointed towards desired conclusions. That is they were designed to leak over 500MiBi of memory and after execution of Garbage Collector the memory was deallocated till 8.0 MiBi. Though, the complete elimination of memory leak cannot be assured by any method till date. But the results have shown that the proposed methodology is quite efficient and effective.

Acknowledgment

I would like to take this opportunity to express my deep sense of gratitude to all who helped me directly or indirectly during this thesis work.

Firstly, I would like to thank my supervisor, **Er. Pushpendra Kumar Pateriya**, for being a great mentor and the best adviser I could ever have. His advise, encouragement and critics are source of innovative ideas, inspiration and causes behind the successful completion of this dissertation. The confidence shown on me by him was the biggest source of inspiration for me. It has been a privilege working with him.

I am highly obliged to all the faculty members of computer science and engineering department for their support and encouragement.

I would like to express my sincere appreciation and gratitude towards my friends for their encouragement, consistent support and invaluable suggestions at the time I needed the most.

I am grateful to my family for their love, support and prayers.

-Anubhav Arun

Declaration

I hereby declare that the dissertation entitled, "*An Implicit Garbage Collection Model for C++ Programing Language*", submitted for the ***Master of Technology*** degree is entirely my original work and all ideas and references have been duly acknowledged. It does not contain any work for the award of any other degreee or diploma.

May, 2015

Anubhav Arun Gupta
Regn No. 11000527

Contents

List of Tables

List of Figures

List of Algorithms

List of Program Codes

Chapter 1
Introduction

Dynamic Memory Management is a concept by which an object or data element of a program is allocated or deallocated over main memory dynamically, during the execution of the program. *Dynamic Memory Management* concept is explored mutually by Operating System, Compiler and Programming Languages domain. Dynamic Memory Management have helped the tech community by reducing Program executable size dramatically. But on the flip side it also have introduce a necessary evil called *Memory Leak*, which in fact is the problem this thesis report will focus on.
Before going into depths of *Memory Leak* and its effects, lets understand what are the components of Dynamic Memory Management and how it's implemented on C++ programming language.

1.1 What is Dynamic Memory Allocation ?

Dynamic Memory Allocation is a subset of Dynamic Memory Management in which an object is allocated over main memory during the runtime of an executable program. If the term Dynamic Memory Management is split and expressed as Dynamic + Memory Allocation, then its easy to understand that memory is assigned to an object during runtime i.e. Dynamically. During dynamic memory allocation, an executing program requests the underlying system (operating system or firmware kernel) to provide a block of main memory for an object. The object or data element is allocated over the Heap Storage of a Process Control Block.

1.1.1 How Dynamic Memory Management is carried out in C++?

In C++ language the dynamic memory allocation is carried out by operators `new` and `new[]`, though function `malloc()`,`calloc()`,`realloc()` are also provided for backward compatibility with C language. But they are not recommended to carry out dynamic memory allocation of class objects because memory allocation functions will not call constructor of the class and hence, the data members are not initialized. In C++ `new` is a language construct that dynamically allocates memory from free store and initializes the memory using the constructor.
If successful `new` attempts to allocate and initialize enough memory for the new data over heap storage and returns the address to the newly allocated and initialize memory.
However, if `new` cannot allocate memory in Heap Storage then it will throw an exception of type `std::bad_alloc`.

Allocation via `new` operator

Syntax::

```
    pointer_variable = new type_name;
```

where, `pointer_variable` is the pointer of type `type_name`. `type_name` can be either primitive data type or user-defined. If `type_name` is a class name then the `default constructor` is called to construct the object.

To initialize a new variable during dynamic allocation.
Syntax::

```
    pointer_variable = new type_name(initializer);
```

where, `initializer` is the initial value assigned to the new variable, or if `type_name` is of class type then `initializer` is the argument(s) to a constructor.

To create an array dynamically
Syntax::

```
    pointer_variable = new type_name [size];
```

where, `size` is the size of the array (one-dimensional array).

▮ *Deallocation via `delete` operator*

In C++ programming language, the `delete` operator deallocates the memory allocated by `new` operator and returns it back to heap. During deallocation the destructor of class type object is called.

This makes clear what are the components of *Dynamic Memory Management* and how its implemented in C++ Programing Language. Lets understands some basics of *Memory Leak*.

1.2 Memory Leaks

Memory Leak is an anomaly exhibited when access to an dynamically allocated object is lost during runtime of the program. More technically, a leak occurs when due to inappropriate memory management a pointer is not deallocate at the appropriate time[7](C++ insight at Appendix A1).

There are two types of memory leak.

1. **Lost Memory ::** In this case program overwrites or loses the pointer p and all other pointers derived from it.

2. **Forgotten Memory ::** In this case allocated memory area m remains reachable but is not deallocated or accessed in the rest of the execution.

Memory leaks are relevant for several reasons, some of them are defined below[7]

1. **Memory leaks are difficult to detect**
   Memory leaks do not immidiatly produce an easily visible symptoms like a crash or the output of the wrong value.

2. **Leaks have the potential to impact not only the application that leaks memory**
   Since, the amount of memory is limited. Thus, as the memory usage of leakin program increases, less memory is available to other running applications. Consequently, performance of every running application can be impacted that leaks memory.

3. **Leaks are common, even in mature applications**
   For example, in the first half of 2009, over 100 leaks in the Firefox web-browser were reported

The best solution to fix this is *Garbage Collection* (explained at Appendix A1.1) [5][16]. Through, this study a probable Research Methodology would be proposed to create a Garbage Collector which would have not been possible until the release of C++11 Standards[1](briefly explained at Appendix A1.2).

Chapter 2
Review of Literature

A limited amount of work have been done to check memory leak and memory corruption problem faced in C++ programming language, same is the situation with Garbage Collection. Apart from debugger, not many tools are available which thoroughly focuses on factors leading to memory leak and memory corruption. The current chapter briefly discusses the research paper published in IEEE Journals or presented in IEEE conferences from 1998 to 2013, which are relevant to the defined problem set.

2.1 Available Solutions to the Dynamic De-allocation problem

The current approaches to fix the issues related to dynamic deallocation can be classified into *implicit approach* and *explicit approach*. The following approaches doesn't remove or change any pre-defined language feature, rather they are more less like an add-on during compilation or a tool used for assistance.

2.1.1 Implicit Approach

The *implicit* approach defines a automatic method, to deal with Dynamic De-allocation issues. This approach can be executed in several different ways, some one of them will be explored in brief one by one. The implicit method are less of a tool rather than methods which are executed as a part of executable program file [15].

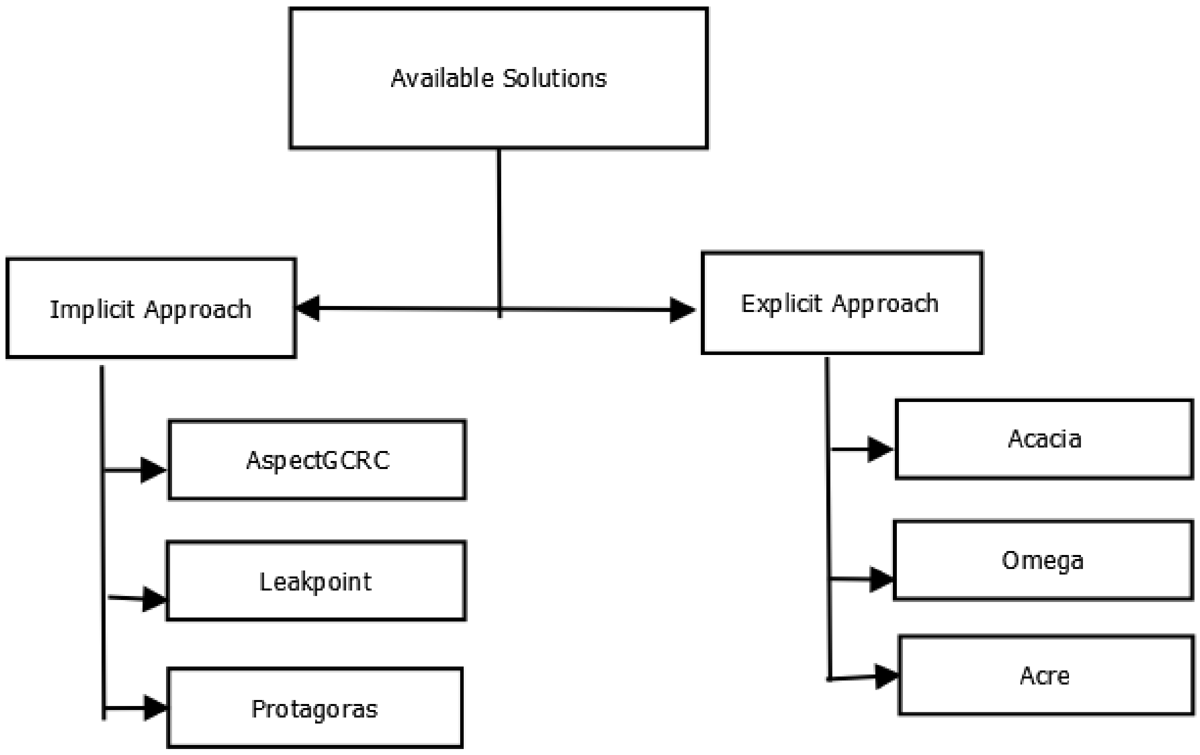


Figure 2.1: Available Solutions

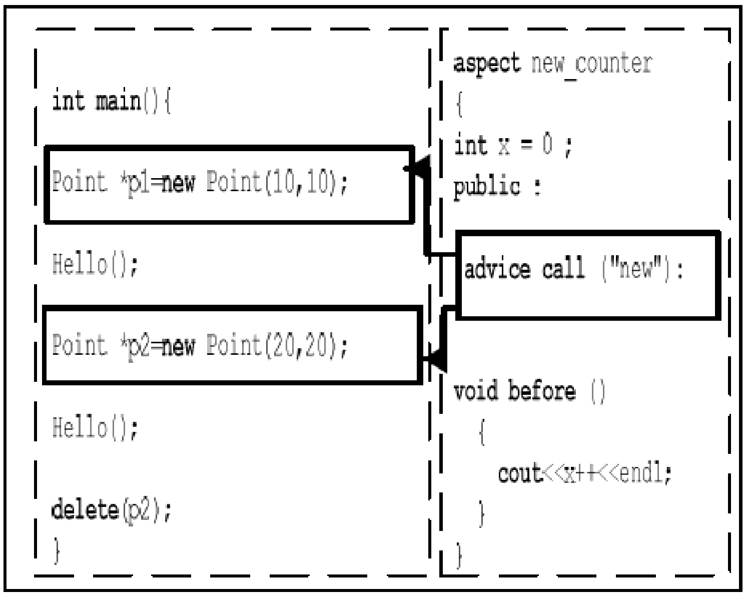
- *Aspect Garbage Collector : Weaving the code during compilation using AspectC++[13]*

This approach uses Aspect Oriented Programing(AOP), to make some thing similar to garbage collector. *Mcheick and Sioud*[13] made two prototypes, first being **ASPECTGCRC** (**A**spect **G**arbage **C**ollector **R**eference **C**ounter) based on the *reference counter* algorithm of garbage collection and second being **ASPECTGCMS** (**A**spect **G**arbage **C**ollector **M**ark and **S**weep) based on the *mark and sweep* algorithm of garbage collection.

```
class Point
{
  int x; int y;
  ...
  Point(int NewX=0, int NewY=0)
  {
     X=NewX; Y=NewY;
  }
```

```
    void showInfo()
    {
        std::cout<<X<<'\n'<< Y;
    }
};

int main()
{
    Point *p1 = new Point(10,10);
    p1->showInfo();
    delete(p1);
    Point *p2 = new Point(200,200);
    p2->showInfo();
}
```

Program Code 2.1: Example of Point Class before weaving [13]



```
int main(){
Point *p1=new Point(10,10);
Hello();
Point *p2=new Point(20,20);
Hello();
delete(p2);
}
```

```
aspect new_counter
{
int x = 0 ;
public :
advice call ("new"):
void before ()
  {
    cout<<x++<<endl;
  }
}
```

Figure 2.2: Example of Point Class after weaving [13]

Reference Counter Algorithm

The object reference counter consists of maintaining a counter of the number of objects referring to a shared object. This counter is incremented every time a new reference to the shared object is created. Also, this counter is decremented each time a reference to the shared object is destroyed.

The author's implementation of AspectGCRC prototype is done in two phases.

In the first phase, the prototype scans the application and retrieve all application class definitions including user created classes. In the second phase, retrieved classes are scanned for object creation. For each class in the prototype maintains in an object table.

A reference counter of the number of objects created is maintained in the object table. The reference counter is incremented when a static instantiation, a reference assignment or a constructor call occurs. And when a call to destructor, delete or free is encountered the reference counter is decremented. Finally, the prototype scans the object table for reference counters having a zero value and deletes the object associated with this reference counters.

Mark and Sweep Algorithm

Mark and Sweep technique is carried out in two phases. In the first phase, all the reachable objects from the roots are marked with a flag. Traditionally, in this garbage collection algorithm, garbage collector traverses a graph of objects using registers and classes as roots, but the author have traversed the code using classes as roots. Depth first search approach is used to traverse the graph in order to mark the reached objects. In the second phase, the slide compact method is used and the object is copied in a free space allocation.

▮ *Linking Code using source code instrumentation[12]*

To demonstrate this approach *Lee,Chang and Hasancite*[12] have created a tool called *Mtrace++*. This tool produces records of allocation and deallocation information. Mtrace++ indentifies origination of allocated memories and life-spans of objects.

Mtrace++ is coded in C++ and inserts begining and ending mark for each dynamic memory allocation scenario. *Mtrace++* is a two phase tool: execution phase and analyzing phase. During the execution phase, it modifies original source code and links with instrumented objects into an executing application and collects data as the application executes. During the analyzing phase, it creates tables which contain concise allocation information. After it finishes the analyzing phase it produces another format of data file which illustrates the allocaiton tree and depth of the path.

The author have explained three approaches to trace the dynamic memory. The first one is rewriting the compiler to handle the trace, second approach is *object code insertion* which is done directly on the executable or the linking stage and the third approach is source code instrumentation. It includes parsing, analyzing, and converting the original source code into a new modified source code.

The *Mtrace++* takes advantage of source code instrumentation. It inserts two marks at the beginning and the ending of the body of the target member function. The target member functions are constructor, copy constructor, type conversion, assignment operator= overloading and user-defined member function. Each mark contains information of a member function and a class name.

Execution Phase

During execution phase, between each *beginning* and *ending* mark of invocation scenario, dynamic memory invocation may be occurred. To generate data, three additional object files are needed: *malloc.o, filemake.o, optrnewg.o.* The allocation size and address information comes from malloc.o. This malloc.o is a modified, inside the *malloc malloc(), calloc(), realloc() and free() functions are modified to write result into the data file.* The filemake module is used to create data file instead of standard output.

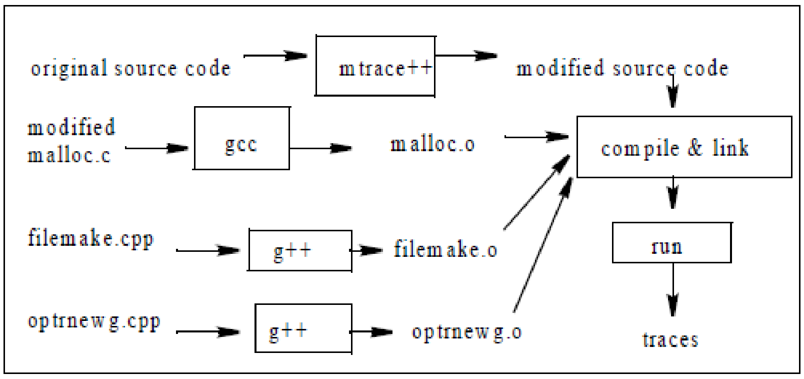


Figure 2.3: Experimental setup of Mtrace++ [12]

The operators *new* is overloaded both globally and locally. The overloaded global new operator is used to distinguish between *new()* and direct call of *malloc()*. To collect the invocation of member data, we need to overload the operator *new* locally.

Analysis Phase(Stat)

For the second phase of the tracing, the analyzing tool for the *Mtrace++*, called *Stat* creates table which contain the summarized allocation information, the table is created for every invocation case. C++ programs allocate a significant number of dynamic objects and furthermore allocate them at a higher rate. Thus, many of the objects can be reused. To reuse objects, a programmer needs to know whether the objects with the same size are allocated prolifically. To investigate this, *Stat* provides a table of object distribution. This table explicitly describes objects sizes, frequencies and originations in C++ programs. With the object distribution table, programmer can get the information of the relationship between size and frequency.

■ *Compile Plug-in*

Sean Et Al. have presented a technique similar to that of *mtrace++*. They have used an approach of source code instrumentation along with object code insertion, to insert patch code in an existing program[6]. The patch code is inserted at specific location during transition of source code into object code. They have developed *Protagoras* a new plugin architecture for GNU compiler collection which allows modification of internal representation of the program under compilation carried out by GCC compiler.

■ *LEAKPOINT: Pinpointing the cause of Memory Leak[7]*

Clause and Orso have explained that memory leaks are type of unintented memory consumption which can adversly affect the correctness and performance of a program. Clause Et Orso have used real leaks that they found in `300.twolf` application during their evaluation. `300.twolf` is a computer-aided-design program that calculates the routing and placement of transistors for microchip design.
The following code snippet shows an example of leak in application while it was executing its test-input sets, which are provided with the application

```
delHtab(){
  int i;
  HASHPTR hptr, zapptr;
  for(i=0; i<3001 ; i++){
      hptr=hashtab[i];
      if(hptr!=(HASHPTR)NULL){
          zapptr=hptr;
          while(hptr->hnext != (HASHPTR)NULL){
              hptr=hptr->hnext;
              free(zapptr);
              zapptr=hptr;
          }
          free(hptr);
27         }
28     }
29     free(hashtab);
30     return;
31 }
```

Program Code 2.2: delHtab Function in 300.twolf[7]

```
34 addhash(char hname[]){
35     int i;
36     HASHPTR hptr;
37     unsigned int hsum=0;
38     for(i=0; i<strlen(hname) ; ++i){
39             sum+=(unsigned int) hname[i];
40         }
41         hsum %= 3001;
42         if((hptr=hashtab[hsum])==(HASHPTR) NULL){
43             hptr=hashtab[hsum]=(HASHPTR) malloc(sizeof(HASHBOX));
44             hptr->hnext=(HASHPTR) NULL;
45             hptr->hnum=++netctr;
46             hptr->hname=(char *)malloc((strlen(hname) + 1) * sizeof(char));
47             sprintf(hptr->hname,"%s",hname);
48             return(1);
49         }
50         else
51         {
52         ...
53         }
54 }
```

Program Code 2.3: addHash Function in 300.twolf[7]

Clause Et Orso claims that most existing leak detection tools provides information for location where leaked memory was allocated i.e. in at line 45 in function `addhash`. While the memory leak is actually occurs at line 26 in function `delHtab`.
The above algorithms provides a good example for types of leak to be expected in practice while dealing with real programs. It illustrates following facts[7].

1. It is a real leak that occurs in a released, commosnly used application.

2. It is caused by common programming error i.e. forgetting to deallocate a component of an object before deallocating the object itself.

3. It occurence doesn't noticeably impact the application; even though it leak memory. `300.twolf` runs to completion and produces the correct output.

4. The leaky memory allocation site and the location where the error may be fixed are far apart in the code, they are in two seperate functions.

Clause Et Orso, have provided examples of memory leak using popularly used subjects like *gcc 3.0, lighttpd 1.4.19, transmission 1.20* where *gcc 3.0* is a popular Compiler, *lighttpd 1.4.19* is a webserver and *tansmission 1.20* is Bittorent client (Explained in Appendix A2.1).

2.1.2 Explicit Approach

The *explicit* approach defines a method to check the deallocation using tools, which may or may not run along the execution, they can be used as a testing tool as well as can be used during development of application.

- *Acacia, an reverse engineering tool based on data model made using software repository[18]*

Chen,Gansner and Koutsofios have made a tool Acacia, which is basically a tool build using reverse engineering approach. They have described a method, which uses a Data Model made using software repository (here from software repository they mean library) of C++, to perform reachability analysis[18]. Thus, detecting deadcode.
They have explained that Software repository is needed in building reverse engineering tools and to re-engineer the software code. A data model is needed to buils a Software repository. A data model is the backbone of software repositoy, the data model design directly affects the effectiveness of software repository ability to support various analysis tools. So, by using this approach a data model is build for C++ software repository which supports reachability analysis and dead code detection at declaration level. The data model is designed in a manner such that it catches all the necessary dependency needed for reachability analysis.

∎ *ACRE: An Automated Aspect Creator for Testing C++ Applications[9]*

Etienne Duclos Et Al have presented ACRE(Automated aspeCt cREator) a tool purely based on AspectC++ technique defined in implicit approach defined is Section 2.1.1 . But instead of using aspect-oriented programming(AOP) as an Garbage Collection tool they have used AOP as an testers tool [9]. In conjugation with domain-specific language ACRE allows tester to add or remove test aspect easily.

Chapter 3
Scope of The Study

This study would be applicable to any C++ Compiler and as a result every source code compiled from it. But the major impact would be observed in C++ application running over firmware. This is so because unlike operating system which run over PC or Server, firmware run over small devices like routers. And small devices have very small amount of memory, which if abused impacts other operation related to it.
For example, if memory leak is exhibited over routers then it will effect the number of buffer it can allocate which would in turn impact negatively over total network performance. Appendix A2.3[3] provides an example of memory leak in Android Operating System. Which occurs because in Android for each application an instance Dalvik machine is created and if their is a leak in that instance of Dalvik it propagates to the whole system. Like Java Virtual Machine, Dalvik is written in C and C++. Thus, there is no cure to fix memory leak on run-time. In this, kind of scenario a Garbage Collector for C++ application would help a lot.

Chapter 4
Objective Of The Study

The main objective of this study is *to propose and implement a Garbage Collection Model* to deal with *Memory Leak* issue in C++ Programming Language.
The Memory Leak problem can be broken down broadly into two sub problem.

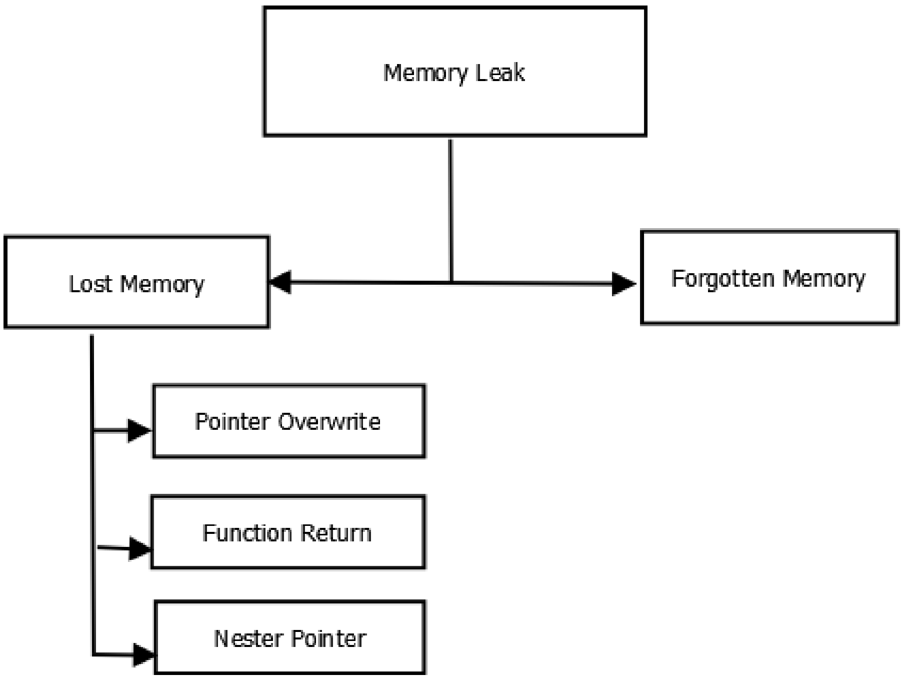


Figure 4.1: Memory Leak breakdown

1. **Lost Memory**
   Which can be further broken down as (sample code are provided in Appendix A2.2)

   (a) Lost Memory due to *pointer overwrite.* Illustrated in Figure 4.2

   (b) Lost Memory due to *function return.* Illustrated in Figure 4.3

   (c) Lost Memory due to *inner pointer*. Illustrated in Figure 4.4

2. **Forgotten Memory** : Its a NP-hard problem, thus, the solution for it is not in scope of this study.

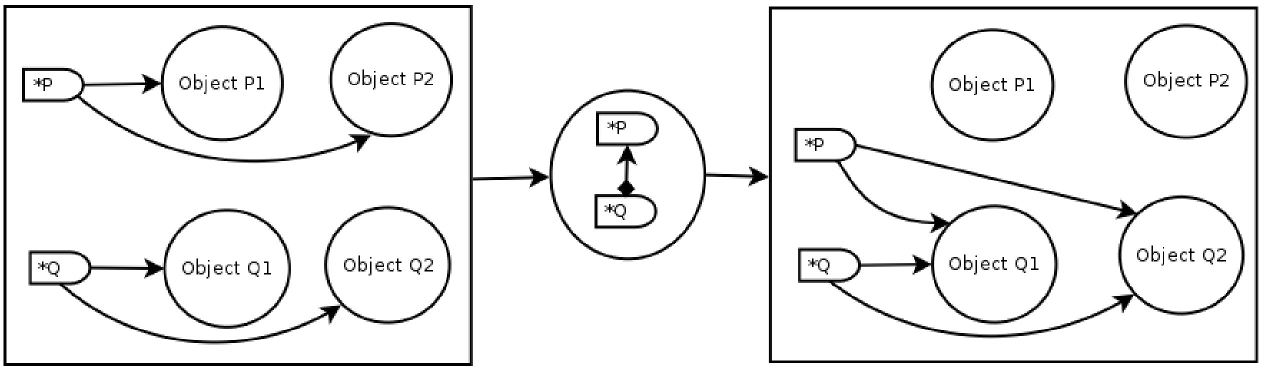


Figure 4.2: Lost Memory Due to Overwrite

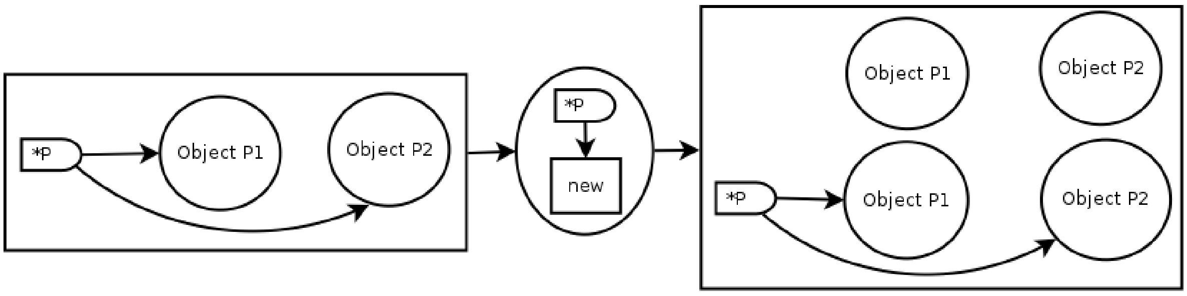


Figure 4.3: Lost Memory Due to function scope

The Garbage Collection Model can be broken down into following sub objectives.

1. A method to grab an object address
2. A method to create and maintain Object Maintenance Table
3. A method to update Object Maintenance Table whenever their is a change in object reference state
4. A method to create Garbage Collector Thread
   (a) A method to choose appropriate garbage collection algorithm

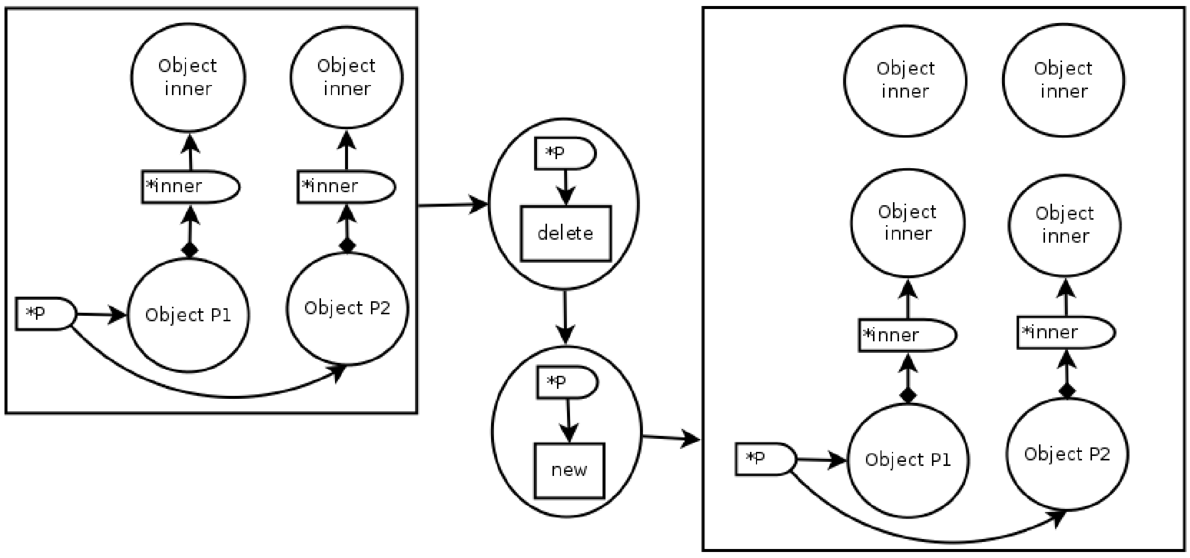


Figure 4.4: Lost Memory Due to Inner Pointer

Chapter 5
Research Methodology

The latest version of C++ i.e C++11 have provided a generic method of creating and managing threads[1], by providing a standardised memory model for multithreaded abstract machine. Thus, by taking inspiration from Java, a method is devised such that a light weight garbage collection thread is attached to application during compilation of source code. The garbage collector thread is automatically generated. Which is achieved by using existing implicit solutions techniques viz. `*Source Code instrumentation*` and `*Rewriting/Modifying the compiler*` defined in *mtrace++*[12].

The methodology follows three phases subsequently to achieve desired garbage collector. The first phase of methodology i.e. **Phase-1** employs *Source Code Instrumentation by using* ***gcc compiler plugin***. This phases involves static analysis of source code. The second phase i.e. **Phase-2** *modifies the source of GCC compiler to solicit in* ***creation of implicit garbage collection thread***. This phase is also responsible for Creation of Garbage Collection logic whose class interface is added to the GCC compiler's Standard C++ library source code. This phase creates logic for dynamic analysis of source code. The third phase i.e **Phase-3** involves creation of custom driver, ***which first compiles source code with GCC plugin*** created in Phase-1 then compiles the modified source without GCC plugin but ***with GarbageCollection Library. Thus, enabling GarbageCollection thread to run along with application.*** It involves dynamic analysis of memory heap.

The following procedure is an abstract of devised methodology to fix *Memory Leak* due to *Lost Memory*.

1. **Phase-1**

   (a) The source code instrumentation is carried out by analysing **GCC A**bstract

Syntax **T**ree (**AST**) of the *input Source Code*.

(b) This is achieved by creating a **Parser plugin for GCC compiler**. Such that it terminates the compilation process as soon as complete AST is formed.

(c) The obtained AST is parsed for pointer declarations along with their line number[19].

(d) A copy of orignal source is made for carrying out source code instrumentation.

(e) The `GarbageCollection::gc.signal_New_Pointer` method is appended after the pointer declaration line such that *the address of pointer is given as the input to the method.*

(f) Then, this modified source code is compiled without plugin in Phase-3.

2. **Phase-2**

(a) The source code of `new` header file is defined at location `./gcc-4.9.2/libstdc++-v3/libsupc++`

(b) The `_GC_Table` class (Figure 5.5) is made inside `new` header file such that it assist the Garbage Collection by intimating creation of new object. Which it keeps in a queue.

(c) As standard library insures that their wouldn't be any memory leaks in its own source code. Thus, all the source file in `./gcc-4.9.2/libstdc++` having call to *new* operator is encapsulated with `_GC_STATE_STOP` flag to intimate Garbage Collector of dynamic allocation carried out by standard library.

(d) Which provides following advantages

   i. It prevent Garbage Collector to accidentally deleting any object being used by standard library.
   ii. Keeps Object Maintenance Table light.
   iii. Enables Garbage Collector to use STL container classes

(e) A `GarbageCollection` class (Figure 5.6) is created whose interface is added to the system header `iostream`

(f) The `GarbageCollection` class the class responsible for creation of Garbage Collection Thread. And it contains main garbage collection logic.

3. **Phase-3**

    (a) A program/driver is built such that it compiles the *input Source Code* and associates GarbageCollection thread in the final executable of *input Source Code.*

    (b) The program/driver follows following steps

        i. The *input Source Code* is compiled using *g++* with **Parser plugin** built in Phase-1

        ii. The modified code obtained is then compiled again with **modified g++** compiler [6].

        iii. The `GarbageCollection` class's library, whose interface was added to iostream header, is compiled along with modified source.

    (c) Thus, we obtain an executable having Garbage Collection Thread associated with it.

    (d) When the executable is executed the GarbageCollection class executes two thread i.e. `gc_Object_Tracker` and `gc_Thread`.

    (e) The `gc_Object_Tracker` is responsible to update **Object_Maintenance_Table** whenever an object is created using a `new` operator.

    (f) The `gc_Thread` is responsible to call `gc_Worker` method after a fixed interval time.

    (g) The `gc_Worker` is responsible to employ garbage collection logic.

5.1 Phase-1

During *Phase-1* of implementation, a plugin for GCC Compiler is created. Which when compiled with any source code parses the code for Pointer declarations[19] and then encapsulate them in `signal_New_Pointer` method of `GarbageCollection` Class formed in Phase-2 (Refer Section 5.2) .
Parsing of *input source code* is carried out using **GCC A**bstract **S**yntax **T**ree(**AST**), which is formed during compilation of input source code. The Parser plugin interrupt the compilation process using `gate-callback` as soon as the **AST** is created. The obtained **AST** is traversed, in order to get the pointer location and name. This Parsing process is implemented as Parser class which is illustrated in the Figure 5.1.

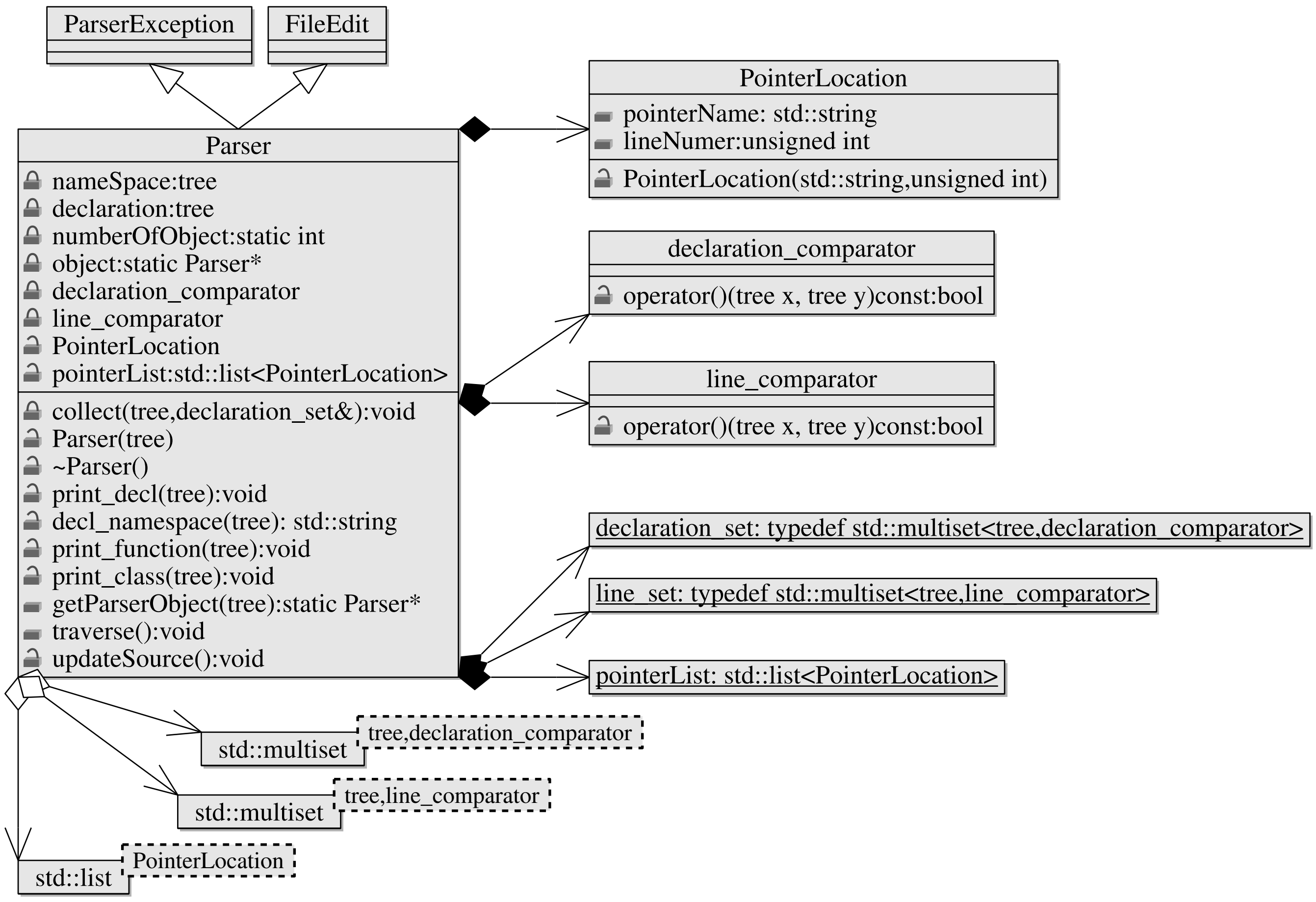

ParserException
FileEdit
Parser
nameSpace:tree
declaration:tree
numberOfObject:static int
object:static Parser*
declaration_comparator
line_comparator
PointerLocation
pointerList:std::list<PointerLocation>
collect(tree,declaration_set&):void
Parser(tree)
~Parser()
print_decl(tree):void
decl_namespace(tree): std::string
print_function(tree):void
print_class(tree):void
getParserObject(tree):static Parser*
traverse():void
updateSource():void
PointerLocation
pointerName: std::string
lineNumer:unsigned int
PointerLocation(std::string,unsigned int)
declaration_comparator
operator()(tree x, tree y)const:bool
line_comparator
operator()(tree x, tree y)const:bool
declaration_set: typedef std::multiset<tree,declaration_comparator>
line_set: typedef std::multiset<tree,line_comparator>
pointerList: std::list<PointerLocation>
std::multiset
tree,declaration_comparator
std::multiset
tree,line_comparator
std::list
PointerLocation

Figure 5.1: Parser Class

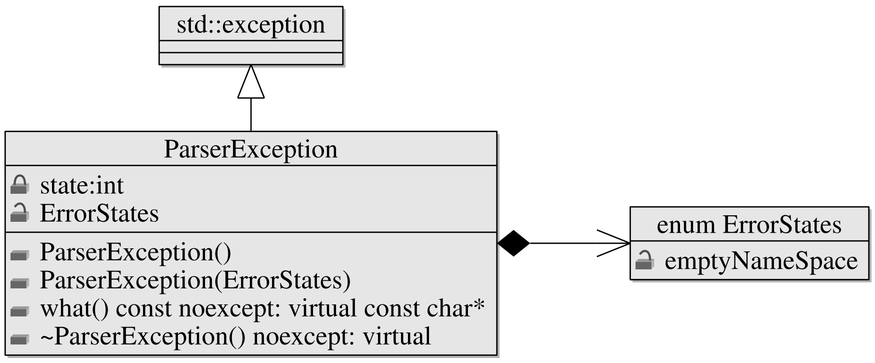

std::exception
ParserException
state:int
ErrorStates
ParserException()
ParserException(ErrorStates)
what() const noexcept: virtual const char*
~ParserException() noexcept: virtual
enum ErrorStates
emptyNameSpace

Figure 5.2: ParserException Class

The `Parser` Class is implemented as singleton class i.e. only one instance of class is allowed. This design pattern is chosen to make sure the complete operation is carried out by a unique object. The `Parser` Class inherits an custom Exception Class viz.

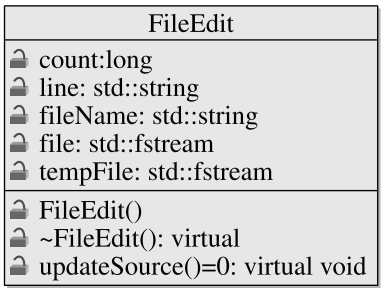


Figure 5.3: FileEdit Class

`ParserException` Class (defined in UML 5.2) which is present for future extension of Parser Class. It handles/notify logical error related to Parsing the *input Source Code*. The `Parser` Class also inherits an Abstract Class viz. `FileEdit` (defined in UML 5.3) which acts an interface for file related operation. Its abstract method `updateSource` is implemented in `Parser` Class. Its responsible to carry out updation of `input Source Code` on the basis of information collected during parsing. The Parsing of *input Source Code* starts with `traverse` method of `Parser` Class. It is described in Algorithm 1.

The `traverse` algorithm uses `collect` algorithm as described in Algorithm 2, to get global declaration (which can be either function, record or any other primitive declaration) as well as namespaces in the order of their appearances.

`Collect` Algorithm collects all the declaration and saves them in a `set` object of `declaration_set` which is `multiset` defined in `Parser` Class. Refer UML 5.1.
For each value of `set`, `print_decl` Algorithm (Algorithm 3) is called by `traverse` Algorithm.

The `print_decl` algorithm checks whether the given declaration is of `type` type. If it is then `print_class` or `print_function` methods are called depending upon their `type_code tc`.

Algorithm 1 TRAVERSE

```
\* This method is responsible to traverse whole source for global declarations,function declarations and class declarations *\
function TRAVERSE
    COLLECT(nameSpace,set)
    for all Items $i$ in set $set$ do
        PRINT_DECL(i)
    end for
end function
```

Algorithm 2 COLLECT

```
\* This method is responsible to find and arrange all the namespaces and declarations in a source code in the order they appear *\
function COLLECT(tree ns, declaration_set& set)
    $level \leftarrow \mathrm{NAMESPACE}(ns)$
    for all Declaration $decl$ in list $level$ do
        if $decl \neq built-in$ then
            SET.INSERT(decl)
        end if
    end for
    for all Namespaces $decl$ in list $level$ do
        if $decl \neq built-in$ then
            COLLECT(decl,set)
        end if
    end for
end function
```

Algorithm 3 PRINT_DECL

```
    \* This method is responsible for collecting all pointer declaration in source code
  in set $line_set$ *\
  function PRINT_DECL(tree decl)
      Define $type$ of type $tree$
      $type \leftarrow$ TREE_TYPE($decl$)
      Define $dc$ and $tc$ of type $tree\_code$
      $dc \leftarrow$ TREE_CODE($decl$)
      if $type = true$ then
          $tc \leftarrow$ TREE_CODE($type$)
          if $dc = TYPE\_DECL$ AND $tc = RECORD\_TYPE$ then
                \* If DECL_ARTIFICIAL is true this is a class declaration. Otherwise
  this is a typedef *\
              if $decl = DECL_ARTIFICIAL$ then
                  PRINT_CLASS(type)
                  return
              end if
          end if
          if $tc = FUNCTION\_TYPE$ then
              PRINT_FUNCTION(decl)
              return
          end if
      end if
  end function
```

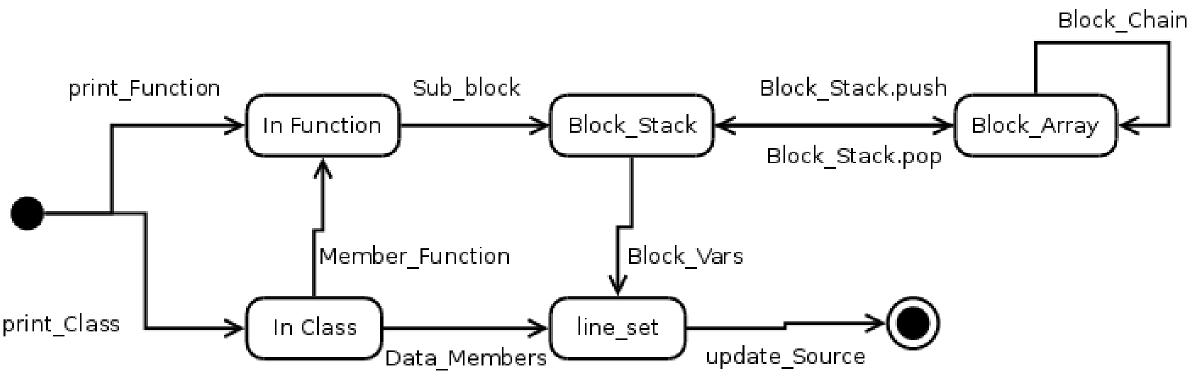


Figure 5.4: State Diagram Illustrating the Parser generated after Phase-1

The State Diagram illustrated in Figure 5.4. Describes the states followed during `print_Class` and `print_function` methods. As well as it provides the mechanism followed by the parser after execution of `traverse` method.

5.2 Phase-2

To implement implicit garbage collection and to provide a method to grab an object. There is a need to edit `new` operator code. To do so a helper class called `_GC_TABLE` is to be created in `new` header file (located at `./gcc-4.9.2/libstdc++-v3/libsupc++`). Figure 5.5 illustrates UML of `_GC_TABLE` class.

The `_GC_TABLE` class is a singleton class. It is designed as singleton to make its object viz. `gc_Table` globally accessible and to deny existence of more than one object of the class. The `_GC_TABLE` class is responsible to maintain a queue of addresses of new object. Whenever, a successful call to *new* is made and if that call is not made by system library i.e. `_GC_STATE_STATUS` flag is set as `_GC_STATE_RUN`. Then, the address of new object is pushed into local queue maintained by `_GC_TABLE` and `_GC_LIST_STATE` flag is set to `_GC_LIST_FILL`.

Algorithm 4 illustrates the modification which are to be carried out in new_op.cc and new_opnt.cc.

The Garbage Collector is made in accordance with Garbage Collection Class illustrated in Figure 5.6, such that an `object_Maintenance_Table` is made and maintained by `GarbageColletion` Class. The Algorithms 7,10,8 and 13 describes creation

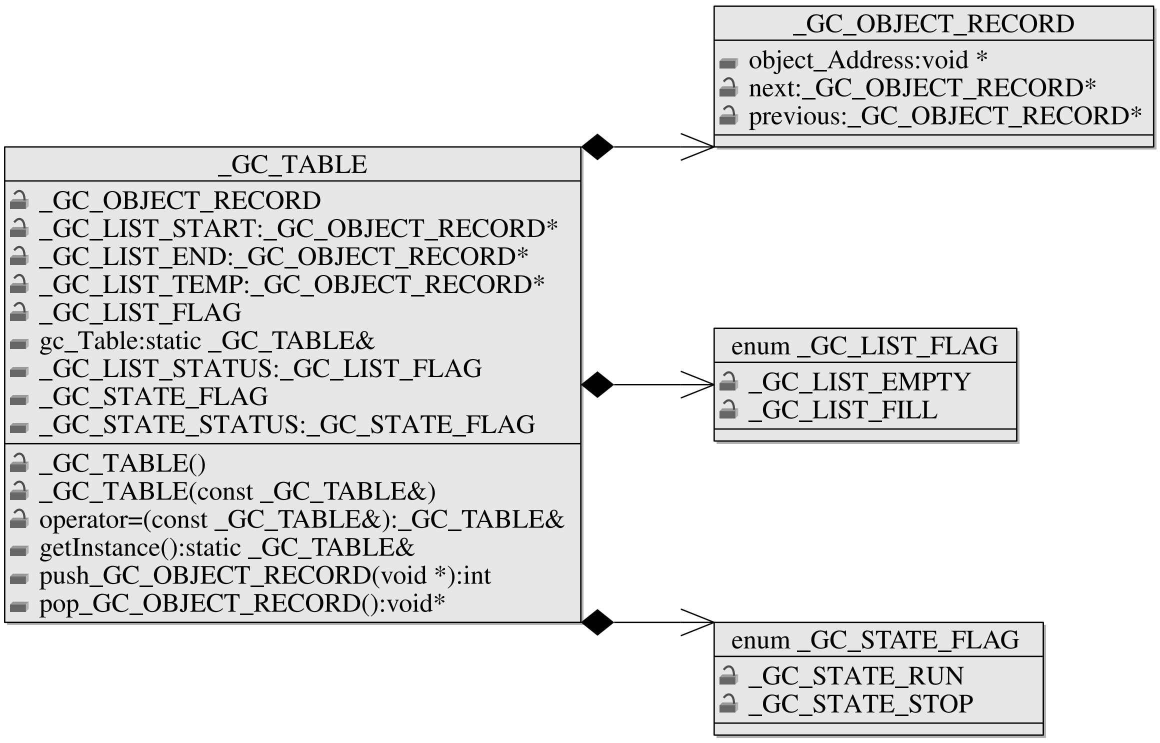

_GC_OBJECT_RECORD
object_Address:void *
next:_GC_OBJECT_RECORD*
previous:_GC_OBJECT_RECORD*
_GC_TABLE
_GC_OBJECT_RECORD
_GC_LIST_START:_GC_OBJECT_RECORD*
_GC_LIST_END:_GC_OBJECT_RECORD*
_GC_LIST_TEMP:_GC_OBJECT_RECORD*
_GC_LIST_FLAG
gc_Table:static _GC_TABLE&
_GC_LIST_STATUS:_GC_LIST_FLAG
_GC_STATE_FLAG
_GC_STATE_STATUS:_GC_STATE_FLAG
_GC_TABLE()
_GC_TABLE(const _GC_TABLE&)
operator=(const _GC_TABLE&):_GC_TABLE&
getInstance():static _GC_TABLE&
push_GC_OBJECT_RECORD(void *):int
pop_GC_OBJECT_RECORD():void*
enum _GC_LIST_FLAG
_GC_LIST_EMPTY
_GC_LIST_FILL
enum _GC_STATE_FLAG
_GC_STATE_RUN
_GC_STATE_STOP

Figure 5.5: _GC_TABLE Class

Algorithm 4 NEW_OP.CC AND NEW_OPNT.CC

```
\* Modifed new_op.cc and new_opnt.cc such that new object address is intimated to GarbageCollection Class. For simplicity, the static object gc_Table of class _GC_TABLE is used as gc_Table instead of using _GC_TABLE :: gc_Table *\
function OPERATOR NEW($\_SIZE\_STD\_$)
    .
    .
    .
    if $gc\_Table.\_GC\_STATE\_STATUS = gc\_Table.\_GC\_STATE\_RUN$ then
        $gc\_Table.push\_GC\_OBJECT\_RECORD(p)$
    else
        $gc\_Table.\_GC\_STATE\_STATUS \leftarrow gc\_Table.\_GC\_STATE\_RUN$
    end if
    return address
end function
```

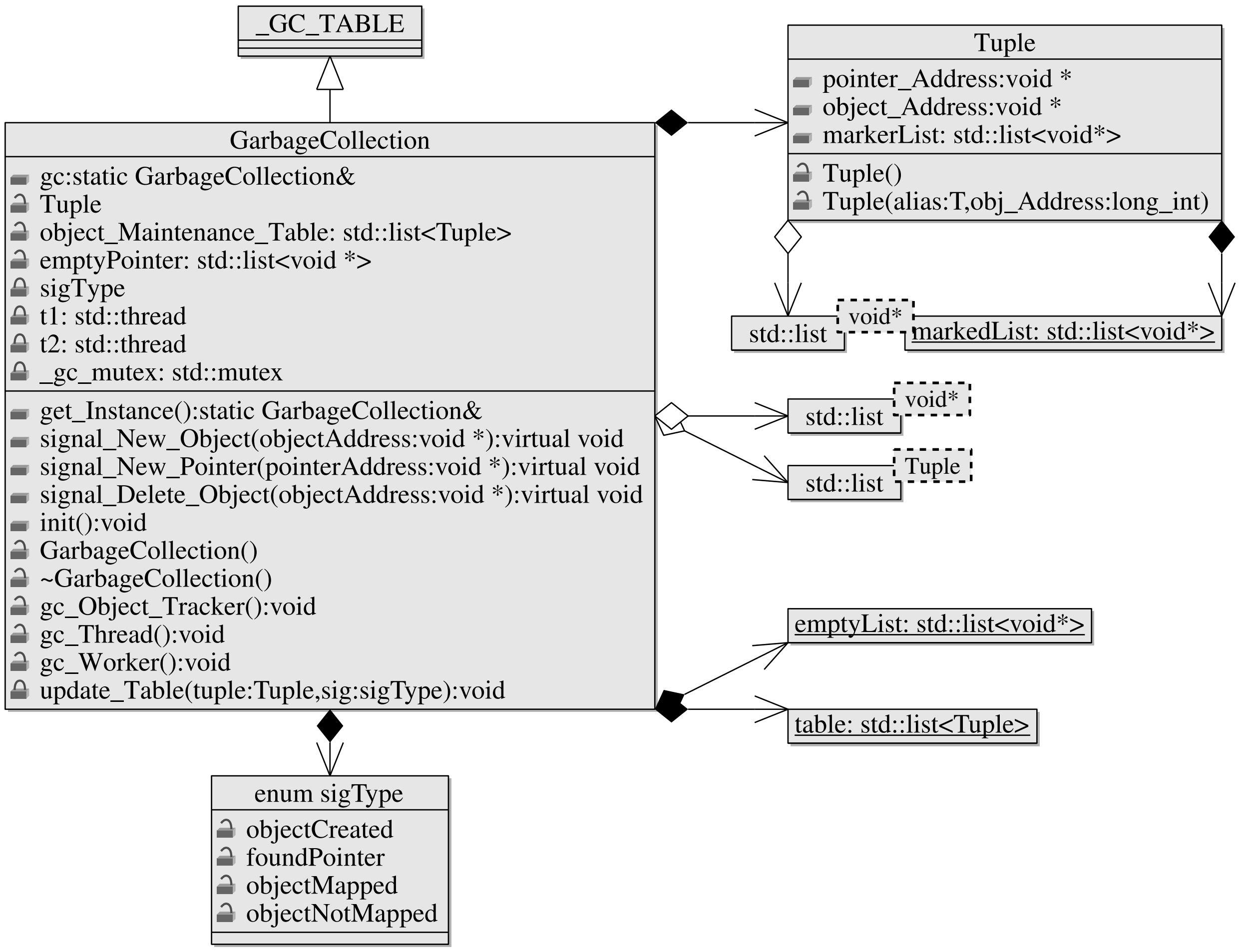

_GC_TABLE
GarbageCollection
gc:static GarbageCollection&
Tuple
object_Maintenance_Table: std::list<Tuple>
emptyPointer: std::list<void *>
sigType
t1: std::thread
t2: std::thread
_gc_mutex: std::mutex
get_Instance():static GarbageCollection&
signal_New_Object(objectAddress:void *):virtual void
signal_New_Pointer(pointerAddress:void *):virtual void
signal_Delete_Object(objectAddress:void *):virtual void
init():void
GarbageCollection()
~GarbageCollection()
gc_Object_Tracker():void
gc_Thread():void
gc_Worker():void
update_Table(tuple:Tuple,sig:sigType):void
Tuple
pointer_Address:void *
object_Address:void *
markerList: std::list<void*>
Tuple()
Tuple(alias:T,obj_Address:long_int)
std::list
void*
markedList: std::list<void*>
std::list
void*
std::list
Tuple
emptyList: std::list<void*>
table: std::list<Tuple>
enum sigType
objectCreated
foundPointer
objectMapped
objectNotMapped

Figure 5.6: Garbage Collection Class

and maintenance of `object_Maintenance_Table` by GarbageCollector in C++11 environment. While garbage collection algorithm is described in Algorithm 12.

The UML Class diagram in Figure 5.6, illustrates the the `GarbageCollection` class. Where, `Tuple` and `object_Maintenance_Table` are its attributes and `get_Instance`, `signal_New_Object()`, `signal_New_Pointer()`, `update_Table()`, `gc_Object_Tracker()`, `gc_Thread()` and `gc_Worker()` are its methods. The `GarbageCollection` class is a *singleton class.*

`Tuple` is a class defined in the `GarbageCollection` class. It have two attributes `pointer_Address` which is of `void *` type and `object_Address` of `void *` type as well. The attribute `pointer_Address` would be initialized with base address of pointer and `object_Address` would store the object's address. The `object_Address` would hold the address of object pointed by the value of `pointer_Address`. The objects of this class serve as a records in Object Maintenance table.

The `object_Maintenance_Table` is the data structure of `list<>` type where each node is of `Tuple` type. The `object_Maintenance_Table` as the name suggest would store total number of pointers as an alias and the address of the objects pointed by them.
Since, `GarbageCollection` class is a *singleton class* thus `get_Instance` is used to return `GarbageCollection` class object.

Algorithm 5 getInstance

```
    \* This method is responsible to return a static refernce of GarbageCollection
  class, this play a key role in making GarbageCollection class singleton *\
  function getInstance
      Define static GarbageCollection object
      return object
  end function
```

Algorithm 6 init

```
    \* This method is called as soon as main method starts. It is responsible to
  instantiate gc_Thread thread and gc_Object_Tracker thread. *\
  function init
      Instantiate thread gc_Object_Tracker
      Instantiate thread gc_Thread
  end function
```

The method `signal_New_Object` is defined in algorithm 8. It is called before return statement of `new` operator. It intimates garbage collector that a new object have been allocated and garbage collector in turn updated is Object Maintenance Table by mapping object address with corresponding pointer Address.
The `update_Table` method is defined in Algorithm 13. It takes `Tuple` and `sig-Type` variable as input. As its name suggest it maps the `pointer_Address` with `object_Address`. The Figure 5.7 illustrates the State Diagram followed to update `object_Maintenance_Table`.
The methods `gc_Thread` and `gc_Worker` are methods meant for creation of garbage collection thread. The thread `gc_Thread` is initiated by `init()` method, it contains a timer which initiate worker method `gc_Worker` whenever timer time-out. The worker method contains custom garbage collection algorithm. The Figure 5.8 illustrate a flowchart of `gc_Worker` algorithm.

Algorithm 7 GC_OBJECT_TRACKER

```
    \* This method checks the $\_GC\_LIST\_STATUS$ flag status, which if set triggers
  $signal\_New\_Object$ method. It is set if an object is created by $new$ operator. *\
  function GC_OBJECT_TRACKER
      while 1 do
          if $gc\_Table.\_GC\_LIST\_STATUS = gc\_Table.\_GC\_LIST\_FLAG :: \_GC\_LIST\_FILL$ then
              $object \leftarrow gc\_Table.pop\_GC\_OBJECT\_RECORD()$
              if $object \neq 0$ then
                  SIGNAL_NEW_OBJECT(object)
              end if
          end if
      end while
  end function
```

Algorithm 8 SIGNAL_NEW_OBJECT

```
    \* This method is responsible to encapsulate the new $objectAddress$ into $Tuple$
  object and then call $update\_Table$ method *\
  function SIGNAL_NEW_OBJECT($objectAddress$)
      UPDATE_TABLE(Tuple(nullptr,objectAddress),objectCreated)
  end function
```

Algorithm 9 SIGNAL_DELETE_OBJECT

```
    \* This method is responsible to encapsulate the deleted $objectAddress$ into
  $Tuple$ object and then call $update\_Table$ method *\
  function SIGNAL_DELETE_OBJECT($objectAddress$)
      UPDATE_TABLE(Tuple(nullptr,objectAddress),objectDeleted)
  end function
```

Algorithm 10 SIGNAL_NEW_POINTER

```
    \* This method is responsible to extract value out of $pointerAddress$ to en-
  capsulate the new $pointerAddress$ and $pointer\_Value$ into $Tuple$ object and then
  instantiate a thread of $update\_Table$ method *\
  function SIGNAL_NEW_POINTER($pointerAddress$)
      $*pointer\_value \leftarrow (unsignedlong * \&)(*((unsignedlong*)(pointerAddress)))$
      Instantiate thread $update\_Table(Tuple(pointerAddress, pointer\_Value), foundPointer)$
  end function
```

Algorithm 11 GC_THREAD

```
    \* This algorithm is a thread. This method is responsible to call $gc\_Worker$ method after a fixed interval of time. *\
  function GC_THREAD
      while 1 do
          Sleep Thread for $Time$ seconds
          GC_WORKER( )
      end while
  end function
```

Algorithm 12 GC_WORKER

```
    \* This method employs core garbage collection logic which is responsible to free unused memory *\
  function GC_WORKER
      Define $mutex\_lock$
      Define list $check$ of type $void*$
      Define $enum\ sig$ of type $sigType$
      $sig \leftarrow objectNotMapped$
      Define $pointer\_Value$ of type $void*$

      for all Values $i$ in list $object\_Maintenance\_Table$ do
          $pointer\_Value \leftarrow (unsignedlong*\&)(*((unsignedlong*)((*i).pointer\_Address)))$
          if $(*i).markedList.empty() = true$ then
              if $(*i).object\_Address \neq nullptr$ && $pointer\_Value \neq (*i).object\_Addresss$ then
                  $check.push\_back((*i).object\_Address)$s
                  $(*i).object\_Address \leftarrow nullptr$
              end if
          else
              CHECK.INSERT( check.end(),(*i).markedList.begin(),(*i).markedList.end())
              (*I).MARKEDLIST.CLEAR( )
          end if
      end for
      for all Values $checkIndex$ in list $check$ do
          for all Values $i$ in list $object\_Maintenance\_Table$ do
              $pointer\_Value \leftarrow (unsignedlong{*}\&)(*((unsignedlong*)((*i).pointer\_Address)))$
              if $pointer\_Value = (*checkIndex)$ then
                  $sig \leftarrow foundPointer$
                  break
              end if
          end for
          if $sig \neq foundPointer$ then
              $delete(*checkIndex)$
          else
              $sig \leftarrow objectNotMapped$
          end if
      end for

      for all Values $emptyIndex$ in list $emptyPointer$ do
          for all Values $i$ in list $object\_Maintenance\_Table$ do
              $pointer\_Value \leftarrow (unsignedlong{*}\&)(*((unsignedlong*)((*i).pointer\_Address)))$
              if $pointer\_Value = (*checkIndex)$ then
                  $sig \leftarrow foundPointer$
                  break
              end if
          end for
          if $sig \neq foundPointer$ then
              delete $(*emptyIndex)$
              $emptyIndex \leftarrow emptyPointer.erase(emptyIndex)$
          else
              $emptyIndex \leftarrow emptyIndex + 1$
              $sig \leftarrow objectNotMapped$
          end if
      end for
      CHECK.CLEAR( )
      $mutex\_unlock$
  end function
```

Algorithm 13 UPDATE_TABLE

```
\* This method update the $objectMaintenanceTable$ by mapping the object Address in the table with corresponding pointer Address. *\
function UPDATE_TABLE($Tuple\ record$, $sigType\ sig$)
    Define $mutex\_lock$
    Define $enum\ objectStatus$ of type $sigType$
    $objectStatus \leftarrow objectNotMapped$
    if $sig = foundPointer$ then
        for all Values $i$ in list $object\_Maintenance\_Table$ do
            if $(*i).pointer\_Address = record.pointer\_Address$ then
                return
            end if
        end for
        for all Values $i$ in list $emptyPointer$ do
            if $(*i) = record.object\_Address$ then
                $i \leftarrow emptyPointer.erase(i)$
                TABLE.PUSH_BACK(record)
                $objectStatus \leftarrow objectMapped$
            else
                $i = i + 1$
            end if
        end for
        if $objectStatus = objectNotMapped$ then
            $record.object\_Address \leftarrow nullptr$
            TABLE.PUSH_BACK(record)
        else
            $objectStatus \leftarrow objectNotMapped$
        end if
    **else if** $sig = objectCreated$ **then**
        Define $pointer\_Value$ of type $void*$
        **for all** Values $i$ in list $object\_Maintenance\_Table$ **do**
            $pointer\_Value = (unsignedlong*\&)(*((unsignedlong*)((*i).pointer\_Address)))$
            **if** $pointer\_Value = record.object\_Address$ **then**
                $objectStatus \leftarrow objectMapped$
                **if** $(*i).object\_Address \neq nullptr$ **then**
                    (*i).markedList.push_back((*i).object_Address)
                    $(*i).object\_Address \leftarrow record.object\_Address$
                **else**
                    $(*i).object\_Address \leftarrow record.object\_Address$
                **end if**
            **end if**
        **end for**
        **if** $objectStatus = objectNotMapped$ **then**
            emptyPointer.push_back(record.object_Address)
        **else**
            $objectStatus \leftarrow objectNotMapped$
        **end if**
    **end if**
    mutex.unlock( )
**end function**
```

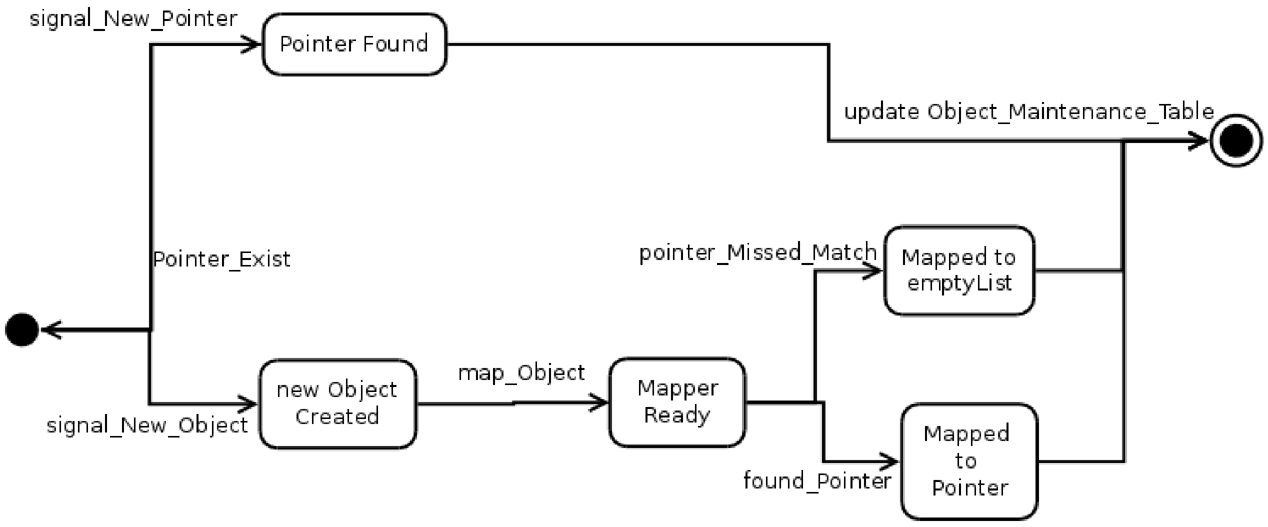


Figure 5.7: State Diagram to Illustrate Update Table Routine

5.3 Phase-3

The **Phase-3** is responsible to create a driver viz. `GC` which contains plugin and carries out parsing followed by compilation with modified GCC Compiler. It functionality is illustrated in Figure 5.9.

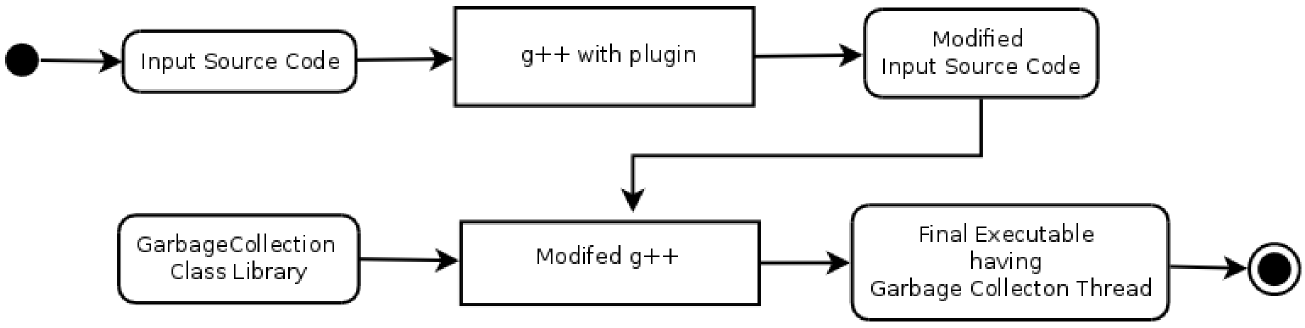

Input Source Code
g++ with plugin
Modified Input Source Code
GarbageCollection Class Library
Modifed g++
Final Executable having Garbage Collecton Thread

Figure 5.9: Figure Illustrating functioning of GC driver

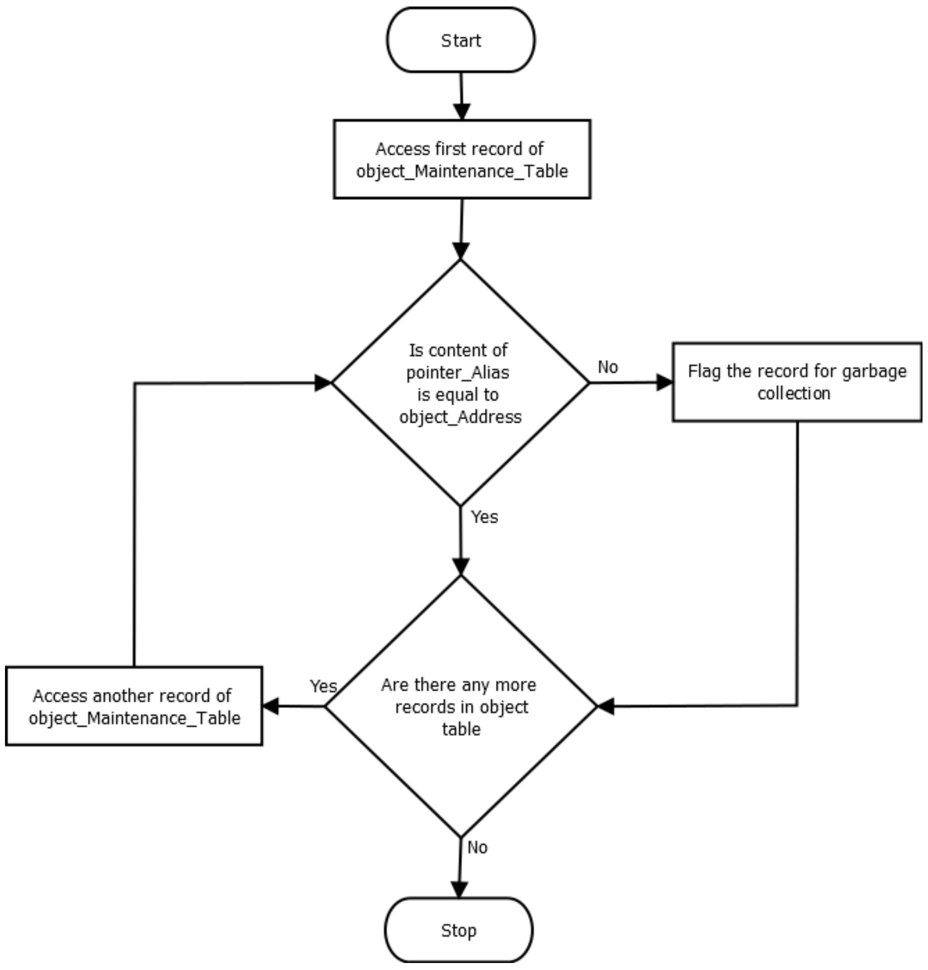

Start
Access first record of object_Maintenance_Table
Is content of pointer_Alias is equal to object_Address
No
Flag the record for garbage collection
Yes
Are there any more records in object table
Yes
Access another record of object_Maintenance_Table
No
Stop

Figure 5.8: Flow Chart Illustrating the functionality of gc_Worker

Chapter 6
Results And Discussion

In a C++ environment,at an average it takes more than 40% programmer's time to debug errors related to memory[13]. Which in worse cases can lead to 80% [8][10][11].

The implementation seems successful because when the driver obtained after **Phase-3** compiles the Test Cases mentioned in Appendix A2.2 i.e. Program Code 8.8,8.9 and 8.10, following outcomes were produced. The Test Cases are designed to leak memory higher than 500MiBi which is approximately 13% of total system memory (4.0 GB).

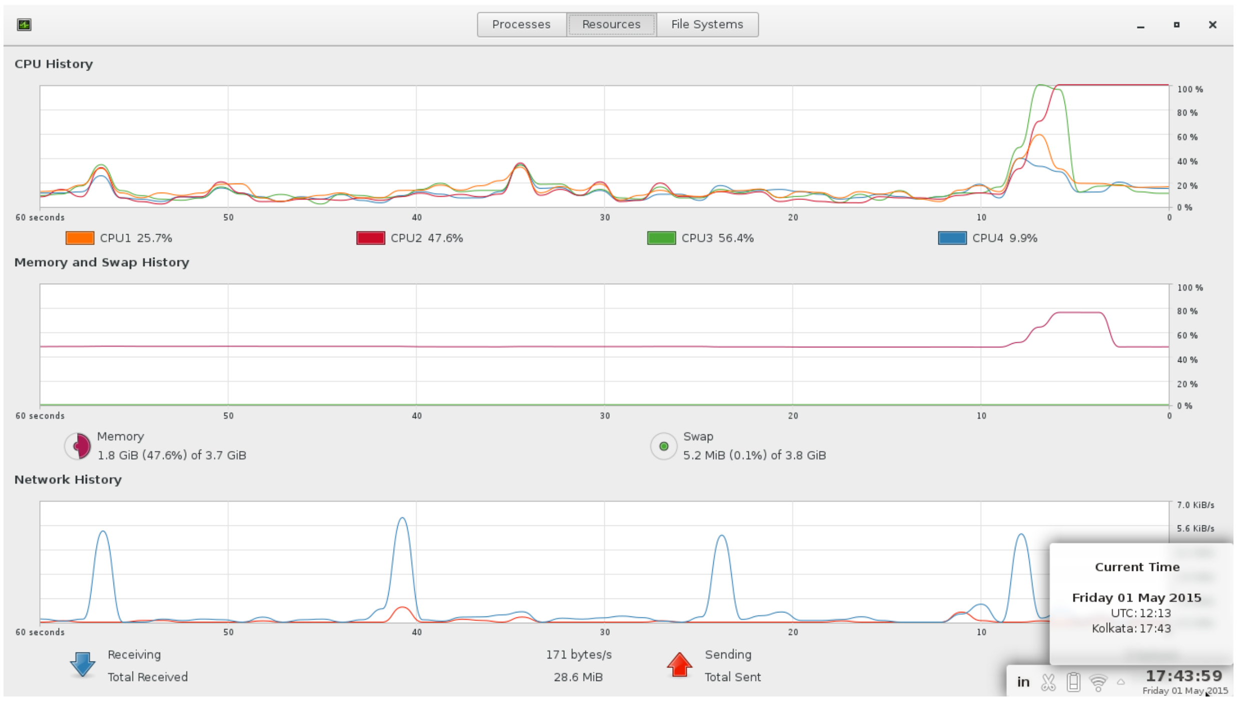


Figure 6.1: GC on Leak due to function scope Prog 8.9

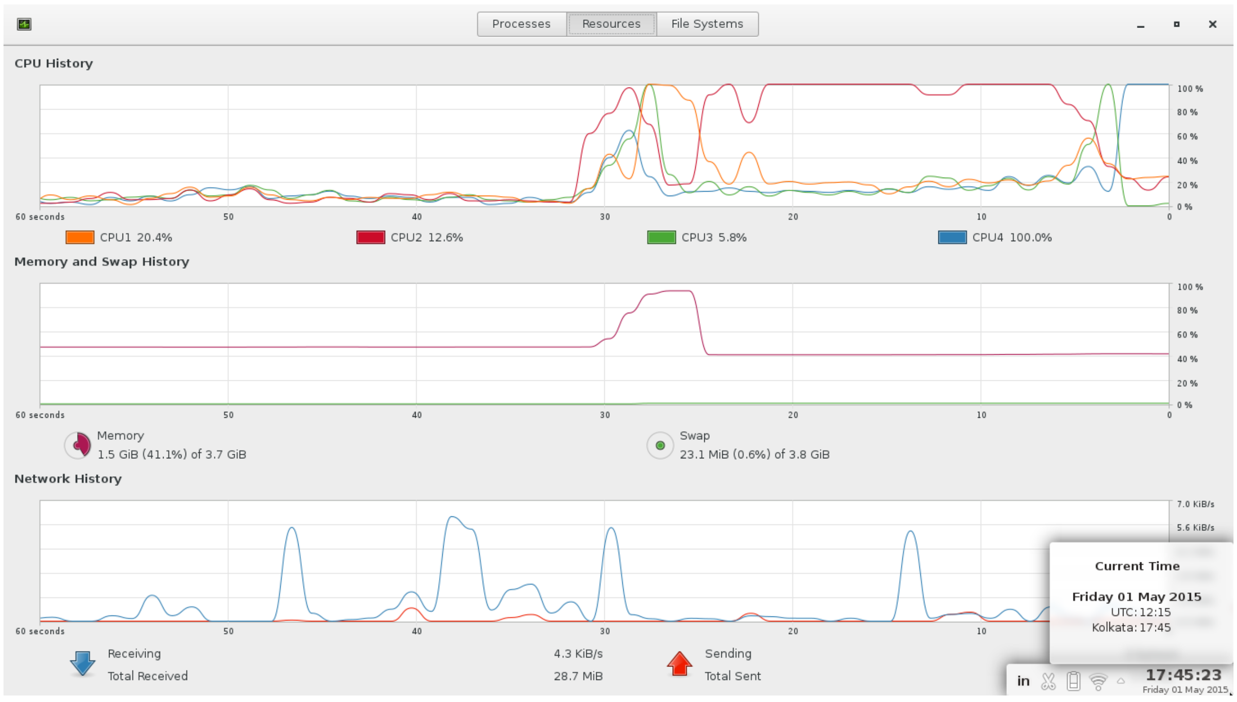


Figure 6.2: GC on Leak due to pointer overwrite Prog 8.8

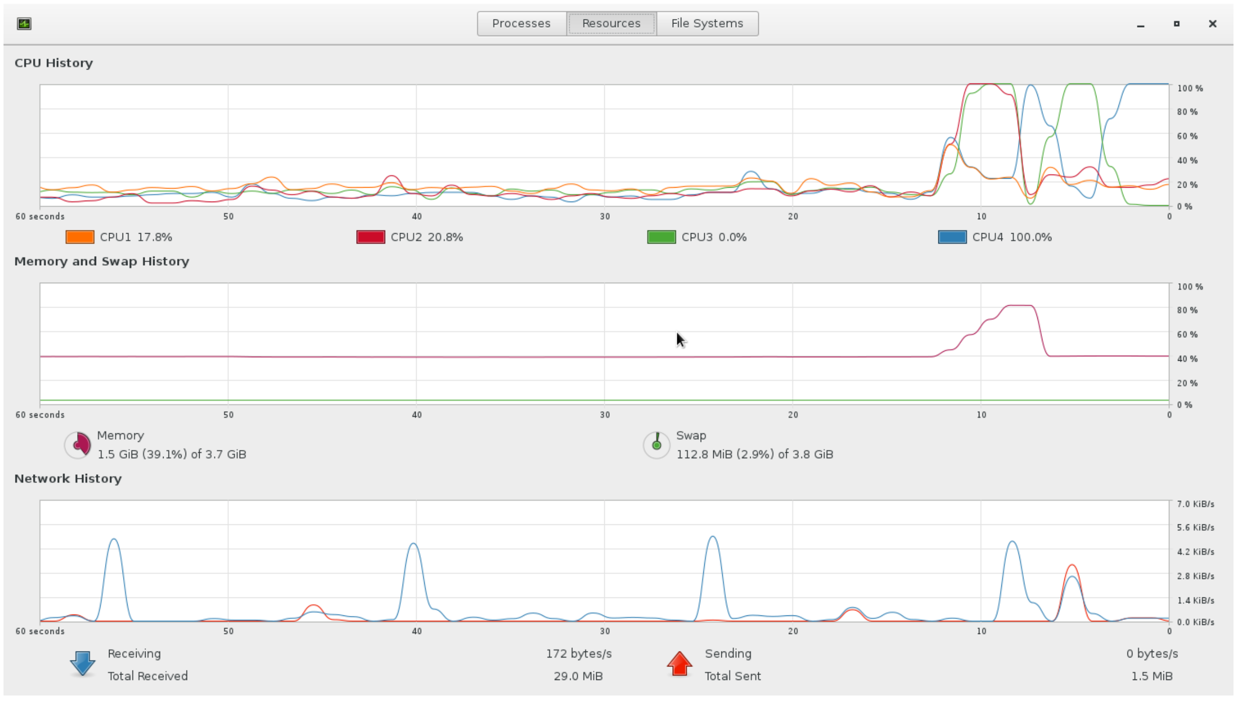


Figure 6.3: GC on Leak due to nester pointer Prog 8.10

Notice the graph being plot at **Memory and Swap history** section of illustrations. The graphs illustrates overall the memory consumption by the system. The rise in graph denotes high memory consumption by the system which is due to execution of the Test Cases. One can notice the fall in graph at **Memory and Swap history** after few seconds of execution. Which is due to the execution of Garbage Collection thread. As mentioned before Garbage Collection thread deallocates leaked memory and returns it back to the system. Thus, bringing down the graph, to an optimum state.

This behaviour is not observed when the same Test Cases are compiled with standard GCC C++ Compiler and executed after compilation (As illustrated by the Figure 6.4 and 6.5). The illustration in Figure 6.6 and 6.7 shows that the Program Code 8.8 keeps leaking around 500MiBi of memory for around 60 min. The Appendix A2.2 provides memory leak illustration for the remaining cases.
Thus, if a table is to be piloted for memory consumed from initial state to x state of Test Case execution. The loss due to memory leak as well as advantage and success of the Garbage Collection Model is clearly observed. The Table 1 illustrates the leak memory being claimed within the first minute of program execution while Table 2 illustrate the memory being continued to leak till x State.

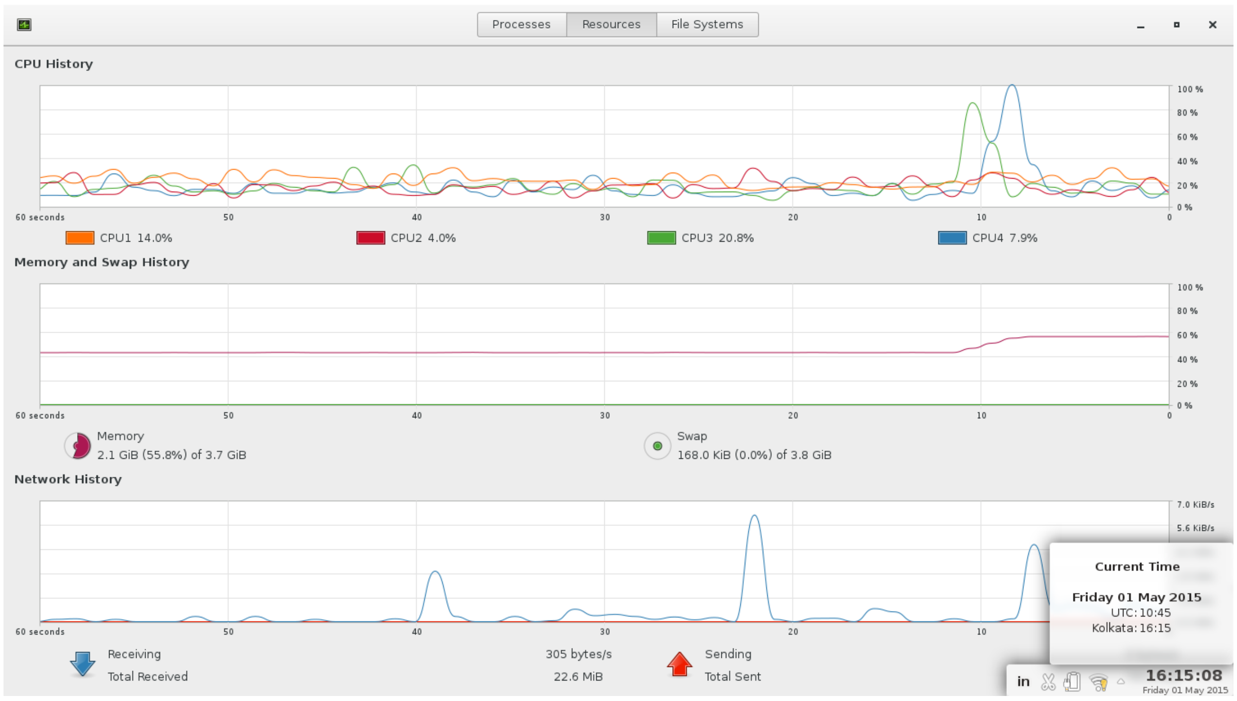


Figure 6.4: Leak due to nester pointer Prog 8.10

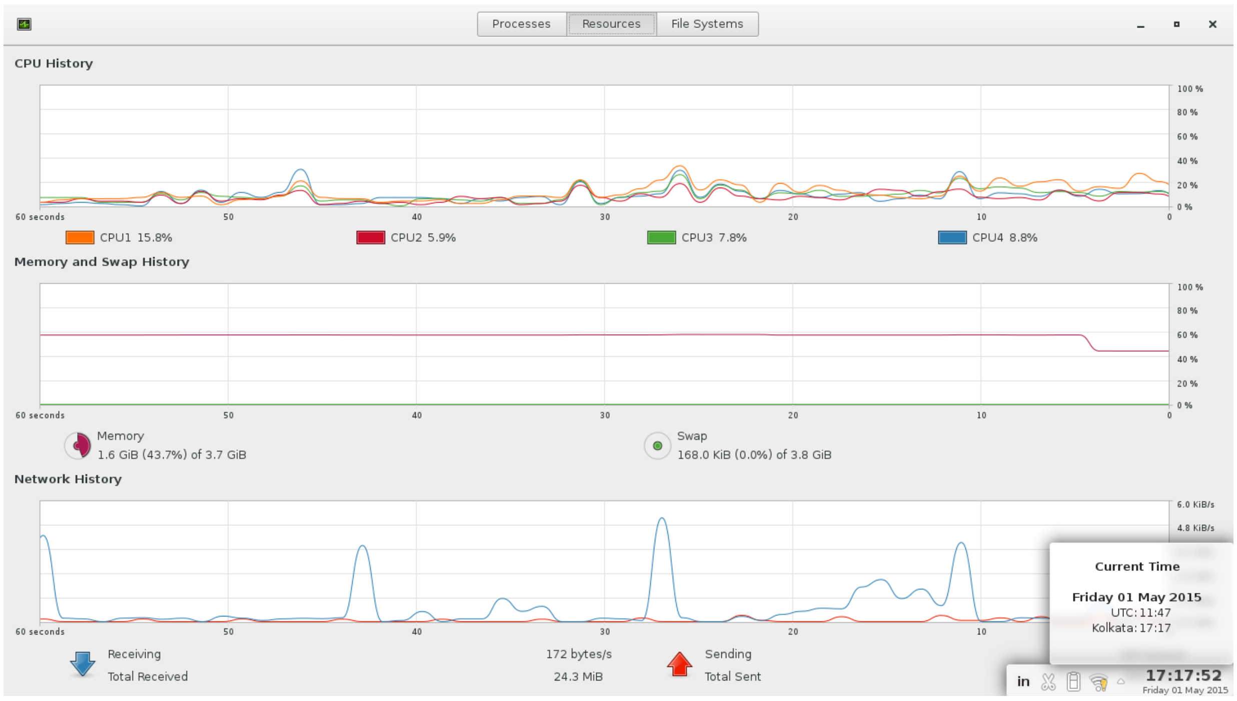
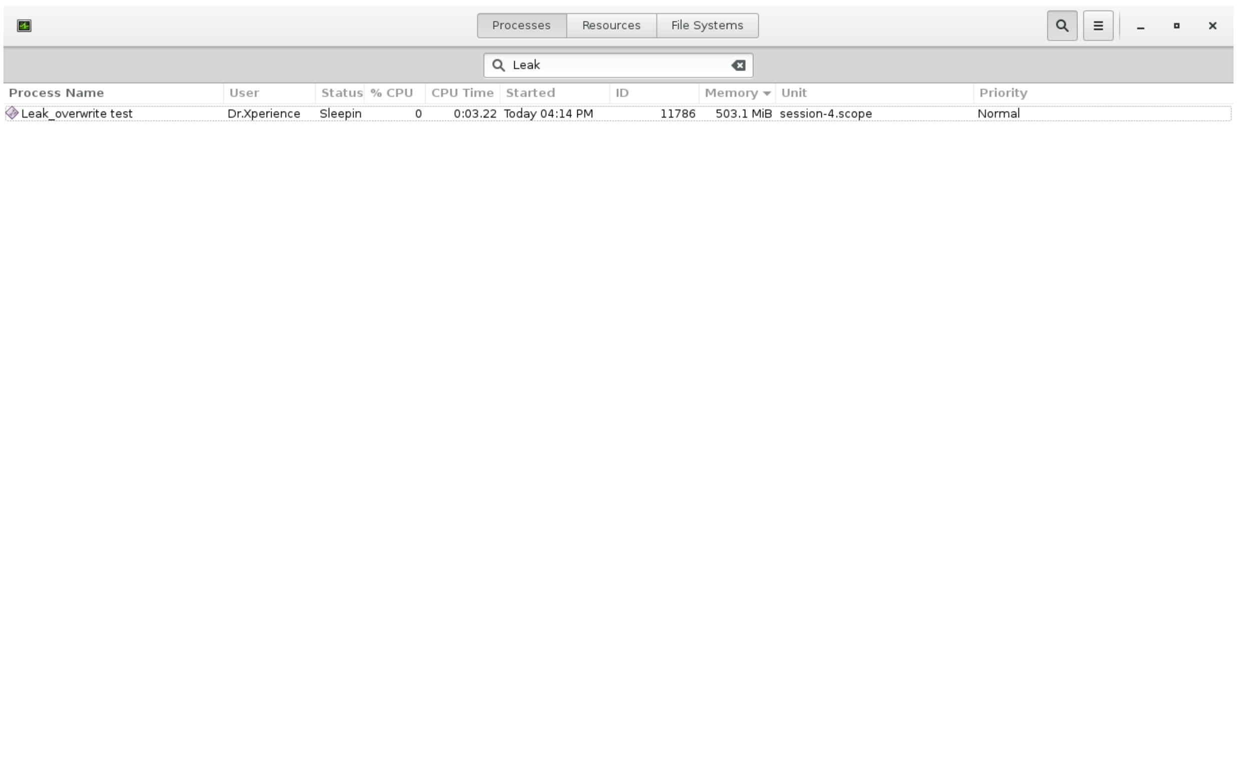
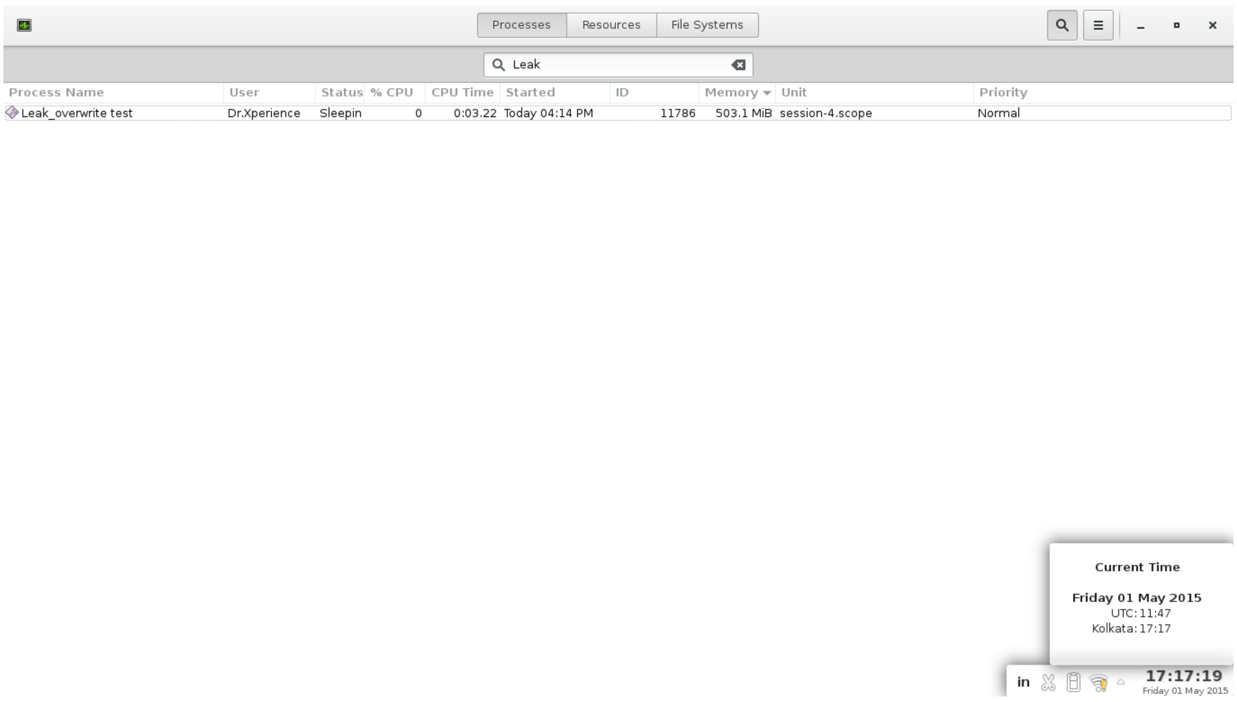


Figure 6.5: Leak due to nester pointer Prog 8.10



Process Name	User	Status	% CPU	CPU Time	Started	ID	Memory	Unit	Priority
Leak_overwrite test	Dr.Xperience	Sleepin	0	0:03.22	Today 04:14 PM	11786	503.1 MiB	session-4.scope	Normal

Figure 6.6: Leak due to nester pointer Prog 8.10



Process Name	User	Status	% CPU	CPU Time	Started	ID	Memory	Unit	Priority
Leak_overwrite test	Dr.Xperience	Sleepin	0	0:03.22	Today 04:14 PM	11786	503.1 MiB	session-4.scope	Normal

Figure 6.7: Leak due to nester pointer Prog 8.10

Table 1: Memory Consumption of Sample Code with GC

Sample Code	**Inital State**		x **State**	
	Memory	**CPU Time**	**Memory**	**CPU Time**
Sample 1 (Prog 8.8)	1.1 GiBi	05.00th sec	4.0 MiBi	08.74th sec
Sample 2 (Prog 8.9)	2.0 GiBi	07.32th sec	7.9 MiBi	10.44th sec
Sample 3 (Prog 8.10)	1.4 GiBi	06.10th sec	4.0 MiBi	09.50th sec

Table 2: Memory Consumption of Sample Code without GC

Sample Code	**Inital State**		x **State**	
	Memory	**CPU Time**	**Memory**	**CPU Time**
Sample 1 (Prog 8.8)	1.1 GiBi	05.00th sec	1.1 GiBi	61st min
Sample 2 (Prog 8.9)	2.0 GiBi	07.32th sec	2.0 GiBi	61st min
Sample 3 (Prog 8.10)	1.4 GiBi	06.10th sec	1.4 GiBi	61st min

Chapter 7
Conclusion And Future Scope

This Study briefly explained the concept of *Dynamic Memory Management* and its side-effect *Memory Leak* and its consequence. Severe negative impact of *Memory Leak* could be observed with practical examples provided in literature survey(Section 2.1.1 and in Appendix A2.2). The related work over the same provided some techniques to deal with it which were either *implict* or *explicit* in nature.
The Implicit Approach solutions were designed such that they change the source code as minimal as possible, or rather apart from source code a new intermediate file is generated[12]. But by doing so, they devoid programmer from fully utilizing the features of Garbage Collector(GC). Plus, this also limited the possibility to extend features of garbage collector. In our methodology we proposed a GarbageCollection Model, which contains Object Maintenance Table and custom Garbage Collection Algorithm. Our model achieved its goal in three Phases, where Phase-1 dealt with static analysis, Phase-2 dealt with Dynamic Analysis and Phase-3 made a driver to execute the outcomes of Phase-1 and Phase-2. Here by, attaching a GC Thread to executable which invoke GC worker thread manually after a fixed interval of time. The result obtained claimed the success of the Garbage Collection Model. Thus, concluding the objective of this study being achieved. By eliminating or reducing the overhead of memory management by programmer to a certain desirable extend.

7.1 Future Scope

The study only provides a Model and prototyped implementation of Model. Their are still certain remaining open for Future Analysis. The Model depends upon static analysis of source code thus, limiting garbage collection to work just for sources not for libraries the source includes. The reverse engineering of the object file obtained after assembling could be a key solution to this limitation and could be used in future to extend this model. Currently, the model doesn't enforces low level mutex over `new` and `delete` operators [17], this could be carried out in future as well. Lastly, the current model give undefined behaviour if source code uses multi-threaded environment. Thus, the above given limitation could be cured in by extending the current model in future.

7.2 Timeline

The Figure 7.1 and 7.2 illustrates the timeline followed throughout the study.

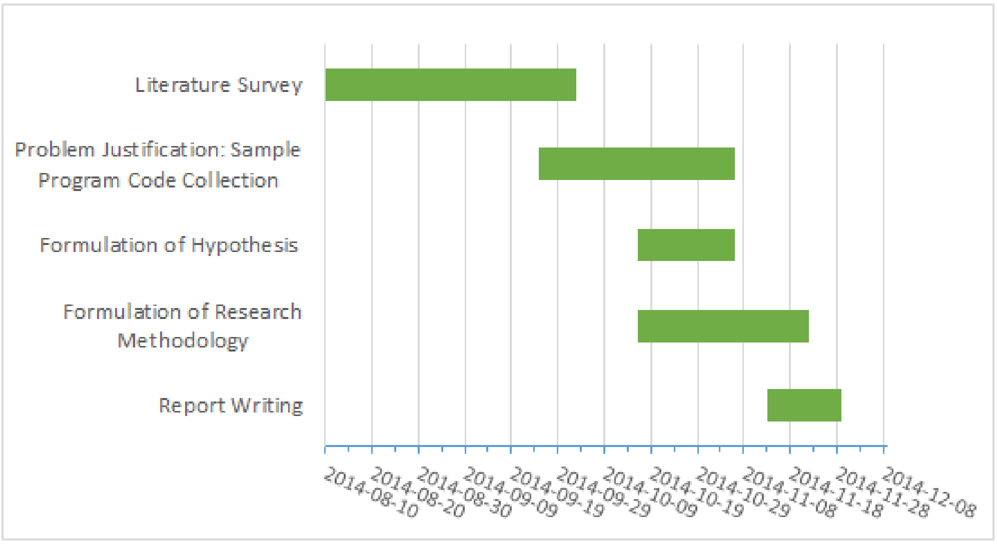


Figure 7.1: Timeline(Gantt Chart Part-1)

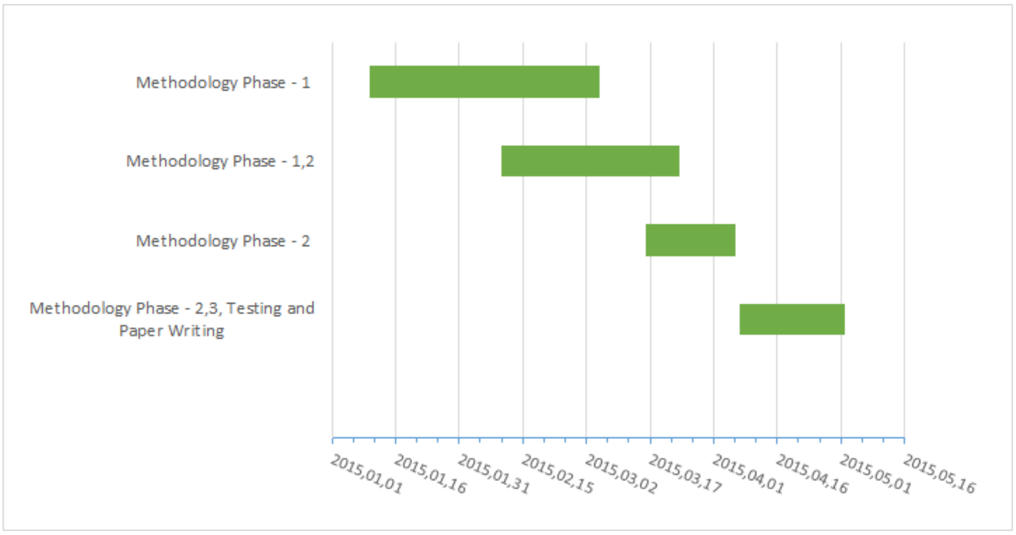


Figure 7.2: Timeline(Gantt Chart Part-2)

References

[1] The standard on the programing language c++, 2011.

[2] Alfred V. Aho, Lam, Sethi, and Jeffrey D. Ullman. *Compilers Principles,Techniques and Tools*. Pearson Education, London, 2nd edition edition, 2007.

[3] Cyanogenmod Android. Camera2 application package(src/com/android/camera/piecontroller.java. `https://github.com/CyanogenMod/android_packages_apps_Camera2/commit/0b9bae7b23b2d95ab7d8c62e591198080bb8c437?diff=split`.

[4] Daniil Berezun and Dmitri Boulytchev. Precise garbage collection for c. In *CEE-SECR '14 Proceedings of the 10th Central and Eastern European Software Engineering Conference in Russia*, number 15, 2014, 2014. ACM.

[5] Hans-J. Boehm and Mike Spertus. Garbage collection in the next c++ standard. In *ISMM '09 Proceedings of the 2009 international symposium on Memory management*, pages 39–48, Dublin, Ireland, March 2009. ACM.

[6] Sean Callanan, Radu Grosu, Xiaowan Huang, Scott A. Smolka, and Erez Zadok. Compiler-assisted software verification using plug-ins. In *20th International Parallel and Distributed Processing Symposium, 2006. IPDPS 2006.*, Rhodes Island, USA, April 2006. IEEE.

[7] James Clause and Alessandro Orso. Leakpoint: Pinpointing the cause of memory leak. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, volume 1, pages 515–524, Cape Town, South Africa, May 2010. ACM.

[8] Ziying Dai and Xiaoguang Mao. Light-weight resource leak testing based on finalisers. *IET Software*, pages 308–316.

[9] Etienne Duclos, Sebastien Le Digabel, Yann-Gael, and Bram Adams. Acre: An automated aspect creator for testing c++ applications. In *17th European Conference on Software Maintenance and Reengineering (CSMR), 2013*, pages 121–130, Genova, March 2013. IEEE.

[10] Guangyan Huang, Guangmei Zhang, Xiaowei Li, and Yunzhan Gong. A state machine for detecting c/c++ memory faults. In *14th Asian Test Symposium, 2005. Proceedings.*, pages 82–87, Calcutta, USA, Dec 2005. IEEE.

[11] Ashish Kundu and Elisa Bertino. A new class of buffer overflow attacks. In *31st International Conference on Distributed Computing Systems (ICDCS), 2011*, pages 730–739, Minneapolis, MN, June 2011. IEEE.

[12] Woo Hyong Lee, J. Morris Chang, and Yusuf Hasan. A dynamic memory measuring tool for c++ programs. In *Proceedings. 3rd IEEE Symposium on Application-Specific Systems and Software Engineering Technology, 2000.*, pages 155–159, Richardson, TX, March 2000. IEEE.

[13] Hamid Mcheick and Aymen Sioud. Comparison of garbage collector prototypes for c++ applications. In *IEEE/ACS International Conference on Computer Systems and Applications, 2009. AICCSA 2009.*, pages 668–674, Rabat, May 2009. IEEE.

[14] Jon Rafkind, Adam Wick, John Regehr, and Matthew Flatt. Precise garbage collection for c. In *ISMM '09 Proceedings of the 2009 international symposium on Memory management*, pages 39–48, Dublin, Ireland, March 2009. ACM.

[15] Yulei Sui, Ding Ye, and Jingling Xue. Detecting memory leaks statically with full-sparse value-flow analysis. *IEEE Transactions on Software Engineering*, pages 107–122.

[16] L. Veiga and P. Ferreira. Complete distributed garbage collection: an experience with rotor. *IEE Proceedings - Software*, pages 283–290.

[17] Farn Wang, Karsten Schmidt, Fang Yu Geng-Dian Huang, and Bow-Yaw Wang. Bdd-based safety-analysis of concurrent software with pointer data structures using graph automorphism symmetry reduction. *IEEE Transactions on Software Engineering*, pages 403–417.

[18] Yih-Farn, Emden R. Gansner, and Eleftherios Koutsofios. A c++ data model supporting reachability analysis and dead code detection. *IEEE Transactions on Software Engineering*, pages 682–694.

[19] Jianwen Zhu and Silvian Calman. Context sensitive symbolic pointer analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 516–531.

Chapter 8
Publications

Communicated Paper in Thomson Reuters indexed International Journal

Anubhav Arun Gupta and Pushpendra Kumar Pateriya, An Implicit Garbage Collection Model for C++ Programming Language, *ACM Transaction on Programming Languages and Systems, Current Issue: Volume 37 Issue 2, April 2015.* (*Communicated*)

Appendix

Appendix A1

Appendix A1.1

What is Garbage Collecion?

Data that cannot be referenced is generally known as *garbage.* Garbage collection is a form of *automatic memory management.* Garbage collector can be a program, method or thread which attempts to reclaim objects that are no longer in use by the program or simply deallocates unreachable data.
Garbage collector was initially implemented in Lisp programming language in 1958[2]. Since, then it is provided by languages like Java, Perl, ML, Modula-3, Prolog, Smalltalk etc. The main design goal of garbage collection is to automatically reclaim the chunks of storage holding objects that can no longer be accessed by a program[14],[4].

Appendix A1.2

What is new in ISO C++11 standards?

There is lot new offering by C++11 standards, like initialization of data member during definition, extending exception handling feature, some new member functions added into existing classes of standard library and introduction some new classes. But the most definitive or the one which would interest us the most is *thread* library. Now on every compiler supporting C++11 will be able to create and manage thread in standardized manner, allowing programmer to write generic multi-threaded code without relying on third party solutions (here, by generic we mean portable).

Appendix A1.3

Insight on Allocation Vs. Deallocation in C++

Allocation mechanism in C++ language, which is carried out by the *new* operator is not much prone to bugs, as an exception of type *std::bad_alloc* is thrown by the application and can be caught and rectified by the programmer[**?**]
Whereas, while carrying out deallocation using *delete* operator, there is no mechanism to verify that operating system carried out delete operation without any fault. Though, language standards[1] enforces execution shouldn't be continued until object is deleted properly.
Thus, if delete is not called before the object gets out of scope then the memory pointed by the object's reference pointer is totally at the mercy of underlying system.

Appendix A2

Appendix A2.1

Memory Leak Examples Provided by Clause Et Orso [7]

Clause Et Orso, have provided examples of memory leak using popularly used subjects like *gcc 3.0, lighttpd 1.4.19, transmission 1.20* where *gcc 3.0* is a popular Compiler, *lighttpd 1.4.19* is a webserver and *tansmission 1.20* is Bittorent client.

```
static struct spelling *spelling_base;

static void push_string(char *string){
...
 spelling_base=xmalloc(spelling_size * sizeof(struct spelling));


 ...
 }

 void finish_init(){
 ...
 //free(spelling_base);
 constructor_decl = p->decl;
 ...
 spelling_base = p->spelling_base;
```

```
  ...
}
```

Program Code 8.1: Relevant code for the error in gcc [7]

In Program Code 8.1 memory allocated at line 4540 in function `push_string` is leaked when *gcc*'s type verifier switches from an inner-context to an outer-context. The leak occurs at line 5187, where `spelling_base` is overwritten bt the memory area it points to is not deallocated. The commented code in the Algorithm 1.3 shows the code that was addded by the developers to fix this error: a call to `free` was added at line 5179 to deallocate the memory area before *spelling_base* is overwritten.
lighttpd version 1.4.19, contains two memory management errors. The first error in *lighttpd* causes a memory leak if the option *url.rewrite-repeat* is set in web server's configuration file.

```
URIHANDLER_FUNC(mod_rewrite_uri_handler){
...
428 // if(con->plugin_ctx[p->id] == NULL) {
429      hctx = handler_ctx_init();

430      con->plugin_ctx[p->id] = hctx;
431 // } else {
432 // hctx = con->plugin_ctx[p->id];
433 // }
...
}
```

Program Code 8.2: Relevant code for the first error in lighttpd[7]

In Program Code 8.2 memory allocated at line 429 in `mod_rewrite_uri_handler` is leaked if this section of code is executed twice. The leak occurs at line 430. In the first execution, the only pointed to the allocated memory area is stored in the plugin context array at line 430. During the second execution, this pointer is overwritten and, because the area of memeory it points to is not deallocated, a leak occurs.The commented code in the Algorithm 1.4 shows the code that was added by the developers to fix this memory management error: the code now checks whether memory was already allocated.
The second error in *lighttpd* causes a leak when the web server parses a request with duplicate http header.

```
    int http_request_parse(server *srv, connection *con){

    ...
774 if(NULL == (ds = (data_string*)array_get_unused_element(con->request.
        header, TYPE_STRING))){

775 ds = data_string_init();
    }
    ...
812 else if (cmp > 0 && 0 ==(cmp = buffer_caseless_compare(CONST_BUF_LEN(
        ds->key),CONST_STR_LEN("Content-Length")))){

814 char *err;
815 unsigned long int r;
816 size_t j;
817 if(con_length_set){
818     con->http_status = 400;
819     con->keep_alive = 0;
820     if(srv->srvconf.log_request_header_on_error){
821         log_error_write(srv,__FILE__,__LINE__,"s","duplicate...");
822         long_error_write(src,__FILE__,__LINE__,"Sb","request-header:\n",
                con->request.request);
823   }
824   //array_insert_unique(con->request.headers,(data_unset*)ds);

825   return 0;
    }
      ...
    }
    ...
    }
```

Program Code 8.3: Relevant code for the second error in lighttpd[7]

In Program Code 8.3 memory allocated at line 775 in `http_request_parse` is leaked because the function returns wihtout deallocating it. The leak occurs at line 825. Since, *lighttpd* is concerned with performance, it maintains a list of allocated request headers that it reuses to save the overhead of memory allocation. The inserted call fixes the error by adding the allocated memory area to the pool of request headers.

In *transmission* version 1.20 the leak occurs when the corresponding torrent file is stopped.

```
    static void invokeRequest(void *vreq){
    ...
718 hash = tr_new(uint8_t,SHA_DIGEST_LENGHT);
719 memcpy(hash,req->torrent_hash, SHA_DIGEST_LENGTH);
720 tr_webRun(rq->session,req->url,req->done_func,hash);

721 freeRequest(req);
722 }
```

Program Code 8.4: Relevant code for the error in transmission invokeRequest function[7]

In Program Code 8.4 memory allocated at line 718 in `invokeRequest` is leaked.

```
    void tr_webRun(tr_session *session, ..., void *done_fucn_user_data){
169 struct tr_web_task *task;
    ...
174 task->done_func_user_data=done_func_user_data;
    ...
177 tr_runInEventThread(session,addTask,task);
    ...
175 }
```

Program Code 8.5: Relevant code for the error in transmission tr_webRun function[7]

The leak occurs at line 82 in Program Code 8.5 i.e. `processCompleteTasks`

```
    static void processCompletedTasks(tr_web *web){
    ...
77  task->done_func(web->session, ... task->done_func_user_data);
    ...
80  evbuffer_free(task->response);
81  tv_free(task->url);
82  tr_free(task);
    ...
83  }
```

Program Code 8.6: Relevant code for the error in transmission processCompletedTasks function[7]

This error is fixed by inserting a call to a deallocation function at line in `onStoppedResponse`, which is called at line 77 in `processCompletedTasks`.

```
static void onStoppedResponse(tr_session *session, ... void *
    torrent_hash){
dbgmsg(NULL, "got a response ... message");
// tr_free(torrent_hash);
onReqDone(session);
}
```

Program Code 8.7: Relevant code for the error in transmission: onStoppedResponse function[7]

The above algorithms of `transmission` application is a good example of how an memory allocated in one function/Class can be leaked at completed different function during its execution. This also raises a question, what could a programmer do if the memory is leaked in some private library being used in current code.

This example illustrates how memory leak can propagate from one function to another.

1. Here, memory is allocated to `hash` at line 718 of `invokeRequest` is passed to function `tr_webRun` as the formal parameter.

2. In function `tr_webRun` the actual parameter is a generic pointer `done_fucn_user_data` storing same address as `hash`.

3. The address stored in `done_func_user_data` is assigned to `task->done_func_user_data` where `task` is a pointer to structure `tr_web_task`.

4. The `task` pointer is passed to running Thread. On thread completion `processCompletedTasks` function is called which in-turn calls `onStoppedResponse` function.

5. The memory is leaked when `task` is freed at line 82 of `processCompletedTasks` function without freeing object pointed by `task->done_func_user_data`.

Appendix A2.2

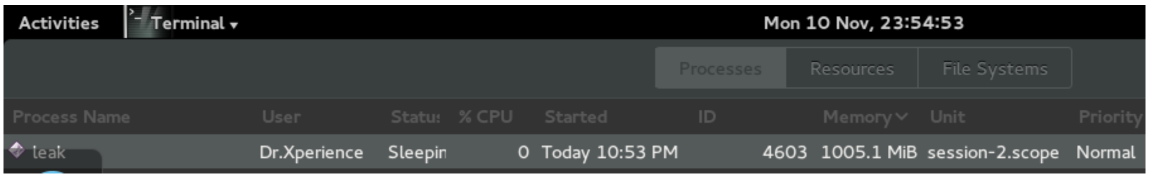
Sample Code to create Memory leak Situations

The following programs provides a working example of memory leak.

```
#include<iostream>
using std::cin;
using std::cout;
using std::endl;

int main()
{
    const long SIZE=1024*1024;
    char * leak= new char[SIZE], *test;

    for(long i=0; i<SIZE; ++i)
    {
        leak[i]='a';
    }

    for(int i=0; i<1000;++i)
    {
        test = new char[SIZE];
        for(long i=0; i<SIZE; ++i)
        {
            test[i]='b';
        }
//      delete leak;
        leak=test;
/*
        .
        .
        Do something  with test
        .
        .
*/
    }

//  delete test;
    delete leak;
    cout<<endl<<"Okay!!";
    cin.get();
34     return 0;
35 }
```

Program Code 8.8: Memory Leak- overwriting the pointer without deallocation

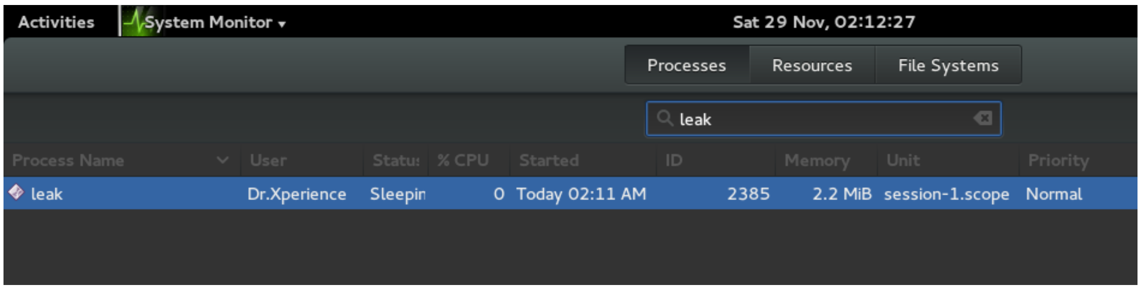
In Program Code 8.8 memory allocated at line 8 is leaked. The leak occurs at line 21, where `leak` is overwritten but the memory area it points to is not deallocated. To understand it more clearly, suppose `leak` points to an object *_obj* stored in memory location *_m*. If programmer have forgotten to write `delete leak;` statement at line 20 of Program Code 8.8, then, the object *_obj* pointed by pointer `leak` will not be removed from memory. Instead the address stored in `leak` will be replaced by address stored in `test`. And there will be no way to access that object *_obj* and the object is lost until the end of program execution. Hence, we have encountered *memory leak*.



```
Activities   Terminal ▾                         Mon 10 Nov, 23:54:53
                                   Processes   Resources   File Systems
Process Name   User          Status   % CPU  Started            ID    Memory      Unit             Priority
leak           Dr.Xperience  Sleepin      0  Today 10:53 PM    4603  1005.1 MiB  session-2.scope  Normal
```

Figure 8.1: Memory Occupied during Memory Leak

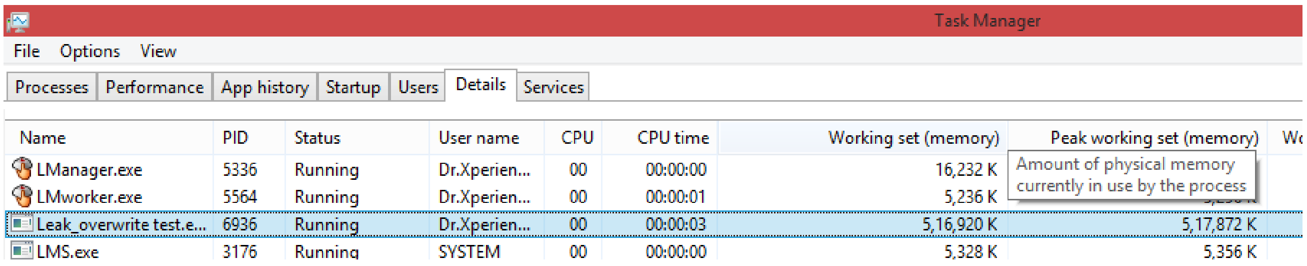
Figure 8.1 illustrate that over 1000 MiB of memory is occupied by the executable of Program Code 8.8 until the process was terminated by user. The program was under execution for around 60 min. But if `delete leak;` statement at line 20 is called, then there is no memory leak and executable of Program Code 8.8 occupies only 2.2 MiB of memory, illustrated in Figure 8.2.



```
Activities   System Monitor ▾                   Sat 29 Nov, 02:12:27
                                   Processes   Resources   File Systems
                                   leak
Process Name   User          Status   % CPU  Started            ID    Memory    Unit             Priority
leak           Dr.Xperience  Sleepin      0  Today 02:11 AM    2385  2.2 MiB   session-1.scope  Normal
```

Figure 8.2: Memory Occupied without Memory Leak

Similar case over Microsoft Windows Operating System. Here for loop is called till 500 so, around 500 MiB of memory is consumed. Illustrated in Figure 8.3.


Task Manager
File Options View
Processes Performance App history Startup Users Details Services

Name	PID	Status	User name	CPU	CPU time	Working set (memory)	Peak working set (memory)	Wo
LManager.exe	5336	Running	Dr.Xperien...	00	00:00:00	16,232 K	Amount of physical memory currently in use by the process	
LMworker.exe	5564	Running	Dr.Xperien...	00	00:00:01	5,236 K		
Leak_overwrite test.e...	6936	Running	Dr.Xperien...	00	00:00:03	5,16,920 K	5,17,872 K	
LMS.exe	3176	Running	SYSTEM	00	00:00:00	5,328 K	5,356 K	

Figure 8.3: Memory Occupied during Memory Leak (Windows Case)

```
#include<iostream>
using std::cin;
using std::cout;
using std::endl;

void leakyFunction()
{
    const long SIZE=1024*1024;
    char * leak= new char[SIZE];

    for(long i=0; i<SIZE; ++i)
    {
        leak[i]='a';
    }

// delete leak;
}
int main()
{

    for(int i=0; i<100;++i)
    {
        leakyFunction();
    }

    cout<<endl<<"Okay!!";
    cin.get();
    return 0;
```

Program Code 8.9: Memory Leak- function return without object deallocation

In Program Code 8.9 memory allocation at line 8 is leaked. The leak occurs at line 13,where `leak` is not deallocated before the termination of function `leakyFunction`

```
1  #include<iostream>
2  using std::cin;
3  using std::cout;
4  using std::endl;

5  const long SIZE=1024*1024;

6  struct Leak
7  {
8     char * leak;
9  };
10 // Main Function
11 int main()
12 {

13    for(int i=0; i<500;++i)
14    {
15       Leak *leak = new Leak;
16       leak->leak= new char[SIZE];

17       for(long i=0; i<SIZE; ++i)
18       {
19          leak->leak[i]='a';
20       }
21 /*
22       .
23       .
24       Do something  with leak
25       .
26       .
27 */
28 //    delete leak->leak;
29       delete leak;
30    }




31    cout<<endl<<"Okay!!";
32    cin.get();
```

```
    return 0;
}
```

Program Code 8.10: Memory Leak- Not freeing the intermediate pointers

Program Code 8.10 illustrates the memory leak due to non de-allocation of pointer present inside the structure

Appendix A2.3

Cyanogenmod Memory Leak [3]

Source: android_packages_apps_Camera2 /src/com/android/camera/PieController.java

`Change-Id: Iaca466f91fb7dc6273b03c2f148439fb1795b1d6`

```
public void initialize(PreferenceGroup group) {
mRenderer.clearItems();
mPreferenceMap.clear();
mListPreferenceMap.clear();
setPreferenceGroup(group);
// Clear Overrides Map when init , to prevent memory leak.
// mPreferences.clear();
// mOverrides.clear();
}
```

Program Code 8.11: Memory leak when suspend/resume camera