TOWARDS AN INTERACTIVE DEBUGGING TOOL FOR C++

BASED ON PROGRAM SLICING

By

RAJESHWAR RAMAKA

Bachelor of Science

in Mechanical Engineering

Jawaharlal Nehru Technological University

Hyderabad, India

1993

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of the
requirements for the degree of
MASTER OF SCIENCE
December 1995

TOWARDS AN INTERACTIVE DEBUGGING TOOL FOR C++

BASED ON PROGRAM SLICING

Thesis Approved

Mansur Samadzadeh

Thesis Advisor

H. Lu

Jacques E. LaFrance

Thomas C. Collins

Dean of the Graduate College

PREFACE

The purpose of this thesis work was to develop an interactive debugging tool for programs written in a subset of C++, based on the program slicing method under the DYNIX/ptx operating system. The topics that were covered as background and context in this work consisted of a review of debugging approaches, an introduction to program slicing and its types: static slicing and dynamic slicing, and a brief review of the different approaches used in implementing dynamic slicing.

The programming part of this thesis work consisted of the design and implementation of a program slicing tool, called cppslicer, for debugging programs written in a subset of C++. The cppslicer software tool is an interactive debugging tool that can be used to help debug simple C++ programs. The cppslicer tool can be used to debug programs with or without classes, operator overloading, and pointers to int, char, float, and classes. The cppslicer program is written in C++. It has about 4900 lines of code distributed around four classes.

ACKNOWLEDGEMENTS

I would like to express special appreciation to my advisor Dr. Mansur H. Samadzadeh. He provided essential guidance and inspiration throughout my thesis work. Dr. Samadzadeh continued to spend endless hours reviewing my work and offering suggestions for further refinements.

I would like to thank my other committee members, Drs. Huizhu Lu and Jacques LaFrance. Their time and effort are greatly appreciated.

Finally, I would like to express my sincerest thanks to my family for their continued support. They helped me throughout my MS program. I couldn't have done it without their continued love and support.

TABLE OF CONTENTS

CHAPTER PAGE

LIST OF FIGURES

FIGURE PAGE

LIST OF TABLES

TABLE PAGE

CHAPTER 1

INTRODUCTION

1.1 Introduction

The probability of a program to be compiled initially without errors is bleak. Such errors include errors in either design or in coding [Borland90]. Debugging techniques are used to identify and fix errors. Debugging techniques attempt to localize the cause of errors in a program and correct them [Brown and Sampson73]. It is generally difficult to find errors merely by observing the afflicted program's behavior. As the size of a program increases, the cost associated with debugging generally increases. The debugging process becomes more difficult especially when programs written by other people are involved [Korel and Laski88]. A number of methods, tools, and approaches have been developed to debug programs. Debugging approaches include file printing utilities, module testing packages, and built-in language facilities. Program Slicing is another debugging approach.

Program slicing, as an approach to debugging, is based on the assumption that it is easier to locate errors in programs of smaller size rather than in the original source program of larger size. Program slicing focuses on the statements that are associated with one or more variables of interest defined as criterion variables [Samadzadeh and Wichaipanitch93]. The program statements that are not related to the criterion variable are omitted. Program slicing is based on data and control flow analysis. It is applied to programs after they are written. Hence program slicing is primarily useful for the maintenance rather than the

design of software [Nanja90]. Using a slicing method, one can obtain a new smaller program (or a program of the same size, in the worst case) that preserves part of the original program's behavior for a particular output or variable [Weiser84]. Program slicing can be categorized into static slicing and dynamic slicing depending upon the slicing algorithm and approach. Static slicing is a method of computing program slices directly from the original source program [Weiser84]. Dynamic slicing, is a method used to compute program slices from the executable part of the original source program [Korel and Laski88] [Agrawal and Horgan90] [Samadzadeh and Wichaipanitch93].

1.2 Purpose of Study

The main purpose of this thesis was to implement a program slicing algorithm in generating a program slice. An interactive debugging tool called cppslicer was developed for debugging a subset of C++ programs running in a UNIX environment. The cppslicer program was designed and developed based on established slicing techniques, and it can run as a utility program on UNIX systems.

The cppslicer program was implemented using object-oriented techniques and approaches [Budd91] in the C++ programming language. It was designed so that it can help debug programs involving straight-line code; control statements such as if, for, while, do, and switch; and classes together with their member functions, both in public and private parts. It can also handle expressions manipulating simple pointers to int, char, and classes (the implicit "this" pointer). The cppslicer program can handle programs that involve operator overloading. Due to time constraints, structures, unions, and user-defined variables

were not included in the scope of this thesis. The cppslicer program cannot handle programs that contain functional overloading, friend functions, and inheritance.

1.3 Organization of Report

The rest of this thesis report is organized as follows. Chapter II discusses different debugging methods and tools that are present on UNIX and some other systems. It also introduces static slicing, dynamic slicing, the different approaches used in implementing dynamic slicing, and types of slices. The chapter concludes with a review of quasi-static slicing. Chapter III outlines the design aspects of the cppslicer program, including the data structures and the different algorithms that were used in designing and developing the cppslicer program. The chapter concludes by discussing a prototype evaluation of the cppslicer program followed by advantages and limitations of the cppslicer program. Chapter IV contains the summary of the thesis report and some of the possible future enhancements that can be done to the cppslicer program.

CHAPTER II

LITERATURE REVIEW

2.1 Debugging

2.1.1 Definitions

To facilitate better understanding of debugging, it is appropriate to define the concepts of error, bug, fault, and defect [Nanja90].

Error. An error is a syntactic discrepancy that can result in faults in software. Errors occur inevitably while writing programs. Sources of errors can be briefly summarized [Wichaipanitch92] as follows.

1. Error in specifying the problem definition. This results in solving a wrong problem.
2. Error due to a wrong algorithm. This error occurs due to choosing a wrong algorithm for a given problem.
3. Semantic errors due to lack of proper knowledge of how a command or a programming construct works.
4. Errors resulting from incorrect programming of an algorithm.
5. Syntactic errors in a program that occur due to lack of sufficient knowledge about or proficiency in programming language concepts.
6. Data errors resulting from failure to predict the ranges of various data items correctly.

Fault. A discrepancy in software, which can impair its ability to function as desired, is

referred to as a fault. Faults can lead to the generation of incorrect output values for a given input. For instance, faults may occur when input variables are not initialized.

Defect. A discrepancy between the code and the corresponding documentation, which may result in severe consequences in the process of installation, modification, maintenance, and testing is known as a defect.

Bug, Debugging, and Debugger. A bug in a computer program is an error that is due to either syntax or logical errors. Debugging attempts to locate such errors (without introducing new errors) and correct them. A debugger is a software tool that gives a user control over program execution status. A user can observe and control the execution of a program and fix the bugs by comparing it with the specified intention. It should be noted that as the sizes of programs increase, the bugs associated with them also increase. As the number of bugs increases, the cost associated with debugging also increases. It is a well known fact that almost fifty percent of the cost involved in software development is associated with debugging and correcting the errors in the program during the testing phase [Tassel74]. Reducing the occurrence of errors in programs is one of the ways to decrease the cost associated with debugging.

Testing. Testing is the process of attempting to verify the correctness of a program in its execution. Testing differs from debugging in that testing is used to test the correctness of a program whereas debugging is used to localize the cause of errors and to correct them [Brown and Sampson73]. The process of debugging and correcting errors of the program can be considered part of the testing step.

2.1.2 Debugging Steps

Debugging can be broadly classified into three steps [Borland90].

1. Identifying the bug: It is important in the debugging process to identify the bug in the testing process by studying the code. This process becomes more complicated when the size of the program increases. If the bug cannot be identified, the scope or set of statements must be narrowed down and the code needs to be studied again.

2. Identifying the cause of the bug: Once a bug is identified, the second and harder part is to identify the cause of the bug. The search for the cause is generally in that part of the program where the bug exists, rather than the whole program.

3. Fixing the problem: Once a bug and its cause are determined, necessary actions must be taken to rectify the problem.

The program is then compiled again and tested for other bugs. If any new or residual bugs still exist, then the above debugging process may be repeated until no more errors are practically detectable in the program.

2.1.3 Debugging Approaches

Debugging is not an exact science; it is called an art in the sense that it is difficult to learn and to teach about debugging. Most programmers are trained in programming, but very rarely are they trained for debugging. It is as true today as ever that it is a difficult process to find bugs and to correct programs. Some of the common debugging approaches are briefly described below [Borland90].

1. Bottom-Up Approach: Concentrate on debugging a program's lowest-level functions

(which do not call other functions) first. Then work upward towards the main part. In this way one obtains a foundation of reliable functions that can be used to step over when they are called in other parts of the code.

2. Look for Classes of Bugs: When a bug is identified look for bugs of similar kind in the same part of the program.

3. I/O-Based Approach: This approach comprises of five steps which are summarized as follows [Borland90].

a. Feed the program some input and trace the code. Watch expressions to check the values of output. Correct the bugs if found.

b. Feed the program with other sets of data that will access the parts of the program that are not accessed from the preceding step.

c. Test every statement in the program. Be alert for statements or expressions that must be tested in more than one way.

d. Concentrate on boundary conditions, which can make a program escape from a loop.

e. When a modification is made to a program, retest the affected parts thoroughly.

If a program is complex, keep a record of the tests performed on the program in the earlier steps. This record will help in all tests whose results could possibly be affected by the change. Once the above iteration is done, test the entire program for correct behavior. Test its response to every type of error it could possibly encounter, within the practical limits of time and effort.

4. Incremental Approach: To localize the cause of errors, adopt incremental testing. This process is feasible only when a programmer is conversant with various programming constructs. Moreover, the programmer should be able to understand the program that is to

be tested reasonably well. These things lay emphasis on the skills of the programmer involved.

5. Logical Approach: Use logical reasoning in determining the cause of errors. This process is done manually and becomes more difficult in dealing with large and complex programs.

6. Trace-Based Approach: Perform a program trace to determine when the program started performing incorrectly. This process becomes more difficult when dealing with large and complex programs. This approach depends upon the programmer's skills and knowledge acquired from experience, because experience will be of great help in identifying the elements of the program that are to be traced and in interpreting the trace information generated.

Most of the experienced programmers would use a subset, if not all of the aforesaid approaches or switch among them, while debugging programs that are unfamiliar [Nanja90].

2.1.4 Debugging Tools

Historically speaking, when debugging techniques were introduced, programmers needed to understand all aspects of a source program and localize the part of the program that did not function as expected. This period is known as "without-tool" generation [Nanja90]. Later on, several debugging tools were developed. In the first generation, debugging tools were based on specific machine architectures. Such tools are used to provide memory dumps and absolute instruction traces, and are called low-level debuggers. In the second generation, tools were designed and developed to provide the memory location address for a variable while debugging. In the third generation, the debugging tools

were capable of some deductions regarding the presence of errors in programs. Examples for low-level debuggers include UNIX adb and DOS-Debug. Examples for high-level debuggers include symbolic debuggers, knowledge-based debuggers, data-base debuggers, and slicing-based debuggers.

2.1.4.1 Symbolic Debuggers. Symbolic debuggers provide information based on the programming language that is used to write the programs which are to be debugged. Contents of the variables can be examined without mentioning the actual addresses of the variables. This type of debuggers provides various options such as tracing the variable, setting watch/break points, and line by line execution.

The main advantage of symbolic debuggers, when compared with low-level debuggers, is that there is no need to know the specific machine architectures. Examples of this type of debuggers can be found on VAX and UNIX systems. The symbolic debugger on VAX is called VAX-DEBUG and can be used to debug programs that are written in assembly language, FORTRAN, BLISS, Basic, Cobol, Pascal, and Pl/I [Nanja90]. The symbolic debugger on UNIX is called sdb which supports FORTRAN, C, and C++.

2.1.4.2 Symbolic Debugger on Sequent Symmetry. A symbolic debugger present on the Sequent Symmetry is pdbx . It can be used for source-level debugging and execution of both conventional and parallel programs. At present, this tool can be used to debug Pascal, Fortran, C, and C++ programs. This tool can be invoked by the command pdbx or dbx. When invoked by the command dbx, it can debug only conventional one-process, one-

program applications.

To debug using pdbx, a program should be compiled using option -g on the command line. This produces a file called execfile which contains the symbol table that includes the names of all the source files translated by the compiler. This makes available all the source files for perusal while using the debugger. It is perhaps worth noting that if the program is compiled with the -g option, the executable code generated is saved into an "a.out" file. To change this option, one needs to compile the source code with the option -o to redirect the executable code into another file.

2.1.4.3 Special Options for C++. To compile a source C++ program and to generate an execfile, one needs to compile the program by typing **CC -g <filename>**. To debug C++ programs, one needs to invoke dbx or pdbx using the -D option on the command line. As the on-line manual information on C++ indicates, this option is used to translate cfront-mangled names to the original C++ versions after reading the program symbols. This makes possible the usage of C++ identifier names instead of the mangled names generated by cfront.

The following example illustrates some of the features offered by pdbx. The program in the example is written in C++. It computes the sum, average, and product of the first n numbers.

```
#include<stdio.h>
#include<iostream.h>
#include<string.h>
class TestClass{
  private:
    int sum, fact;              // sum, fact, and average are used to
    float average;              // to store sum, product and average of
  public:                       // the first n numbers.
    TestClass();                // Constructor
    void Calculate(int);
    void PrintVal();            // Print the result
};
// Constructor which initialize the values without being called from
// program.
TestClass::TestClass(){
  sum = 0;
  average = 0;
  fact = 0;
}
// Function used to calculate sum, product, and average of the first n
// numbers
void
TestClass::Calculate(int num){
  int sub = 1;
  while(sub <= num){
    sum = sum + sub;
    sub++;
  }
  int count = 1;
  fact = 1;
  while(count <=  num){
    fact = fact * count;
    count++;
  }
  average = (float) sum / num;
}
// Print the values of variables sum, fact, and average
void
TestClass::PrintVal()
{
  printf("Sum Is %d\n", sum);
  printf("Fact Is %d\n", fact);
  printf("Average Is %f\n", average);
}
main()
{
  int number;
  TestClass t1;
  cout << "Enter the Number" << endl;
  cin >> number;
  t1.Calculate(number);
  t1.PrintVal();
}
```

Figure 1. A program to compute the sum, average, and product of the first n numbers

The usage and some of the features of pdbx on the Sequent S/81 computer are shown below. The % symbol denotes the UNIX Bourne shell prompt from where pdbx can be invoked by typing dbx or pdbx with the -D option.

```
% dbx -D                                  /*Invoking dbx */
dbx version 2.0.1        (00050)
Type 'help' for help.
enter object file name (default is `a.out'):
                                          /*Calling default
                                          executable code */
reading symbolic information ...
(dbx) l 30,39                             /*Listing the lines */
  30                    sub++;
   31           }
   32           int count = 1;
   33           fact = 1;
   34           while(count <=  num) {
   35                   fact = fact * count;
   36                   count++;
   37           }
   38           average = (float) sum / num;
   39   }
(dbx) stop in Calculate                   /*Setting a break point */
[1] stop in Calculate
(dbx) r                                   /*Starting execution */
Enter the Number                          /*Asking for input */
2                                         /*Input from user */
  %2  Stopped at breakpoint 1 in Calculate at line 25
  25           {
(dbx) n                                   /*Stepping to next line */
  %2  Stopped after next in Calculate at line 26
   26            int sub = 1;
(dbx) n
  %2  Stopped after next in Calculate at line 27
   27            while(sub <= num)
(dbx) n
  %2  Stopped after next in Calculate at line 29
   29                      sum = sum + sub;
(dbx) n
  %2  Stopped after next in Calculate at line 30
   30                      sub++;
(dbx) p num                               /*Printing the contents of
                                               variable num */
2                                         /*Value displayed on
                                               to the screen */
(dbx) n
  %2  Stopped after next in Calculate at line 31
```

```
   31            }
(dbx) p sub
2
(dbx) c                                  /*Continue the program */
Sum Is 3
Fact Is 2
Average Is 1.500000
  %2  Stopped after next in main at line 58
   58    }
(dbx) quit                               /*Quit dbx */
```

Using pdbx or dbx, one can locate the cause of errors, if any, by typing the command *where* at the dbx prompt. Some other features of pdbx are listed below.

1. The contents of the variables, structures, and pointers can be examined without mentioning their actual address.

2. The program resident in dbx can be displayed and break/watch points can be inserted during debugging.

2.1.5 Debugging Tools on UNIX

Debugging tools that are typically available on the UNIX systems, other than the symbolic debuggers, are LINT, ADB, PADB, and CTRACE

ADB is a low-level debugger that is used to debug at assembly level, i.e., it allows a programmer to analyze the execution of a program in terms of machine instructions. It also allows a programmer to look into core files and memory dumps.

CTRACE, a debugging tool for C programs, enables a programmer to follow the execution of a program step by step. It inserts statements to print each executable statement and the variables that are referenced or modified, and writes the output to the standard output which can be saved in another file. When the file is compiled and run, it will list each statement to be executed followed by the variables referenced or modified in that statement,

followed by any output from the statement.

2.1.6 Slicing-Based Debuggers

Slicing-based debugging tools produce a slice of a program depending on the variable(s) of interest. A slice can be either executable or not depending on the slicing criterion and the slicing method utilized. A slice is a set of program statements that directly or indirectly contribute to the values assumed by a set of variables at some program point [Weiser84] [Venkatesh91]. These debuggers are high-level debuggers that can be used to locate the cause of errors.

2.1.7 Other Types of Debuggers

Other types of debuggers [Nanja90] include the following. Database debuggers in which debugging is a process of performing queries and updates on a database that contains a program and the execution states of that program. An example of a database debugger is OMEGA. Knowledge based debuggers store the debugging programming knowledge in their knowledge banks. Examples of knowledge based debuggers include Laura, PUDSY, and Proust [Seviora87].

2.2 Program Slicing

Program slicing is a source-to-source transformation that can be used in construction, testing, analysis, and debugging of programs [Weiser84] [Venkatesh91]. Program slicing was introduced by Mark Weiser [Weiser81]. Program slicing is used to

localize errors in programs. Slicing is concerned with the variables of interest that are called criterion variables. The statements involving other variables are omitted. In general, one obtains a new program of smaller size that still retains all aspects of the original program's behavior with respect to the criterion variable [Samadzadeh and Wichaipanitch93]. Program slicing decomposes a large program into relatively smaller programs that are called slices [Weiser81] [Weiser82] [Weiser84].

Operationally, a slice of a program represents a subset of the program's behavior over all possible inputs. The implication of the definition of a slice is that one could execute a slice of a program to obtain the values of the criterion variable [Venkatesh91]. Moreover there can be different slices satisfying the definition for a given program point and a criterion variable. There is at least one slice for a given slicing criterion, the program itself. A statement-minimal slice is defined as a slice with the least number of statements. Finding a statement-minimal slice is reducible to the halting problem which is unsolvable, but one can find approximate slices using data and control flow [Weiser84].

The advantages of slices and slicing methods are based on four facts, as stated below [Weiser84].

1) Slices can be found automatically by methods used to decompose programs by analyzing their data flow and control flow.

2) A slice is normally smaller than the original program.

3) Slices can be executed independently of one another. A slice is itself an executable program whose behavior must be identical to a specified subset of the original program's behavior.

4) Each slice produces exactly one projection of the original program's behavior.

The problems with slices are listed below.

1) They may prove expensive to find for some programs.

2) Producing slices for some large complex programs may be difficult. There may not be any significant slices for a program.

3) Their total independence may cause additional complexity in each slice that could be cleaned up if some simple dependence can be found.

4) Selection of slicing variables may create problems.

In general, it is easy to find significant slices for large classes of programs. Program slicing can be categorized into static slicing and dynamic slicing depending upon the slicing criterion.

2.2.1 Static Slicing

Static slicing is defined on the basis of all computations of a program. Static slicing produces a program segment that consists of those statements that may possibly be executed if the program is sliced according to the desired criterion [Weiser84]. Static slicing is the method of computing slices directly from the original source program. A slice obtained by a static slicing criterion is called a static slice. Generally, it is easy to find a static slice for a program as compared to obtaining a dynamic slice.

In general, a slicing criterion of a program P is a tuple <i, v>, where i is a statement in P and v is a subset of the variables in P [Weiser84]. According to Weiser, "a slice can be defined behaviorally as any subset of a program which preserves a specified projection of its behavior" [Weiser84].

Given a program P, a node *n* in its flow graph, and a variable *var,* the static slice of P with respect to *var* at node *n* can be constructed by finding:

1) All reaching definitions of *var* at node *n*.

2) All reachable nodes in the program from each reaching definition obtained.

If SRD(*var*, *n*, F) represents the set of reaching definitions of a variable *var* at node *n* in flow graph F, then the static slice will be the union of all reachable nodes. More precisely,

$$\text{Staticslice}(P, var, n) = \bigcup_{x \in \text{SRD}(var, n, F)} \text{Reachablenodes}(x, D)$$

where reaching definition of a variable *var* at node *n* in flow graph F is be a set of nodes in F at which *var* is assigned a value and control can flow from that node to node n without any redefinition of *var* along the control flow path, and ReachableNodes(x, D) is a set of vertices in the program dependence graph D that can be reached from x by following one or more edges in D [Agrawal, et al.91].

SRD for the program in Figure 2 with respect to variable total is SRD(total, 15, F) = {6, 15} and static slice with respect to variable total is shown in Figure 3.

One advantage of static slicing over dynamic slicing is that it is easier and faster to identify a static slice [Samadzadeh and Wichaipanitch93]. The reason for this is that the computations for generating a static slice are done directly from the original source program. The disadvantages of static slicing are the following

1) Static slicing yields program slices of generally larger size than those obtained using dynamic slicing.

2) Static slicing cannot treat array elements and fields in the dynamic records as individual variables [Korel and Laski90].

3) Static program slices tend to be large and imprecise when the programs to be debugged

```
#include<stdio.h>                                                    S1
#include<string.h>                                                  S2
main()                                                              S3
{
      int number;                                                   S4
      int fact;                                                     S5
      int total;                                                    S6
      int variable;                                                 S7
      variable = 1;                                                 S8
      fact = 1;                                                     S9
      if( number < 0)                                               S10
      {                                                             S11
            printf("error\n");                                      S12
            number = 0;                                             S13
      }
      while(number < 10){                                           S14
            total = total + 1;                                      S15
            number++;                                               S16
            fact = fact * variable;                                 S17
            variable++;                                             S18
      }
printf("total is %d\t factorial is %d\n",total, fact);              S19
}
```

Figure 2. A sample program to compute the sum and product of the first ten numbers

```
#include<stdio.h>                                                    S1
#include<string.h>                                                  S2
main()                                                              S3
{
      int number;                                                   S4
      int total;                                                    S6
      if( number < 0)                                               S10
      {                                                             S11
            printf("error\n");                                      S12
            number = 0;                                             S13
      }
      while(number < 10){                                           S14
            total = total + 1;                                      S15
            number++;                                               S16
      }
}
```

Figure 3. A static slice of the sample program shown in Figure 2

involve pointers and composite variables such as arrays, records, and unions. Dynamic

slicing overcomes these shortcomings.

2.2.2 Dynamic Slicing

Static slicing was extended to dynamic slicing by Korel and Laski [Korel and Laski88]. According to Korel, a dynamic slice is a sub program that computes the values of the criterion variables in a specific execution of a program [Venkatesh91]. In contrast to Korel and Laski's approach, Agrawal [Agrawal, et al.91] defined a dynamic slice as a collection of statements that affect the values of the criterion variable in a specific execution of a program. The slice may not be executable by itself. To clarify the two different definitions, consider the example shown in Figure 2 that calculates the sum and product of the first ten natural numbers.

Korel and Laski's dynamic slice with respect to the variable total at S19 contains statement S16 because the statements which effect the control statements should be included in the slice. Figure 4 shows Korel and Laski's dynamic slice

```
#include<stdio.h>                                   S1
#include<string.h>                                  S2
main()                                              S3
{
      int number;                                   S4
      int total;                                    S6
      while(number < 10)                            S14
      {
            total = total + 1;                      S15
            number++;                               S16
      }
}
```

Figure 4. A dynamic slice of the sample program shown in Figure 2 based on Korel and Laski's definition of dynamic slicing

of the program in Figure 2.

According to Agrawal and Horgan's definition of a dynamic slice, the dynamic slice with respect to variable total at S19 includes all of the statements that directly affect the criterion variable. All other variables, even the variables that are involved in the control flow of the original program are omitted. The program in Figure 5 shows the output generated based on Agrawal and Horgan's definition of dynamic slice of the program in Figure 2.

```
#include<stdio.h>                                   S1
#include<string.h>                                 S2
main()                                             S3
{
      int number;                                  S4
      int total;                                   S6
      while(number < 10)                           S10
      {
            total = total + 1;                     S14
      }
}
```

Figure 5. A dynamic slice of the sample program shown in Figure 2 based on Agrawal and Horgan's definition of dynamic slicing

Compared to Agrawal and Horgan's dynamic slice, Korel and Laski's dynamic slice is larger in size. But Korel and Laski's slices are definitely executable and never end in infinite loops.

Dynamic slicing consists of two activities [Venkatesh95]: the first activity is to obtain the trace about the execution of the program for a given input, the record activity is to construct slices for variables present in the program.

The execution trace for a program can be obtained using source code or object code,

which are called as source-level instrumentation or object-level instrumentation, respectively [Venkatesh95].

To obtain an execution trace, Agrawal and Horgan used source-level instrumentation over object-level instrumentation because of the following assumptions [Venkatesh95]: ease of portability to different platforms and simplicity of implementation.

2.2.2.1 Dynamic Slicing Procedures. To facilitate better understanding of dynamic program slicing as proposed by Agrawal and Horgan, and by Korel and Laski, it is necessary that the following definitions be presented [Korel and Laski90] [Agrawal, et al.91] [Wichaipanitch92] .

Let the flow graph of the program P be a digraph (V, R, S, L) and C be the slicing criterion, where V represents a set of vertices, R represents a binary relation on program P which is referred to as the set of arcs, $S \in V$ is a unique entry node, and $L \in V$ is a unique exit node.

A vertex in V consists of one instruction, such as input/output statements, assignment statements, and control instructions (e.g., if-then-else or while), which are called test instructions.

An *arc* corresponds to a possible transfer of control flow from one instruction to another instruction.

A path from s to some vertex $n \in V$ is called a *sequence*. If there is input data that causes the path to be transferred during execution, then the path is feasible.

A *trajectory* is a feasible path that has been executed for some input. A trajectory

with respect to an instruction and its position is represented as an ordered pair (an instruction, its position in the trajectory) so as to distinguish among multiple occurrences of the same instruction in a trajectory. If trajectory T for a program P is represented by (k, P) for some instruction k in program P, then the pair can be replaced by k^p and will be referred to as an action. An action k^p is a test action if k is a test instruction.

If T represents the trajectory of a program on input x, then the dynamic slicing criterion of program P executed on x can be defined as C = (x, I^q , V) , where I^q is an action and v is a subset of variables in P.

If *hist* denotes the execution history of a program P on a test-case test and on a variable *var,* then the dynamic slice of P with respect to *hist* and *var* is the set of all statements in the *hist* whose execution has some effect on the value of *var* as observed at the end of the execution of the program. Unlike static slicing, dynamic slicing is defined with respect to the end of the execution history, i.e., if a dynamic slice with respect to some intermediate point in the execution is to be determined, then we should consider the partial execution history up to that point [Agrawal, et al.91].

A dynamic slice, as defined by Agrawal and Horgan, for a variable *var* can be determined as follows.

1) Find all the nodes that correspond to the last definition of variable var in the execution history.

2) Find all nodes in the graph reachable from that node. Then set

DynamicSlice(*hist, var*) = $\bigcup_{X \in DRD(var, hist)}$ Reachablenodes(X, DynamicDep(hist))

where DRD(*var, hist*) denotes the last occurrence of the node in hist that assigns value to

var.

The Steps involved in obtaining a dynamic slice based on Korel and Laski's definition for dynamic slicing for a program P can be summarized as follows [Korel and Laski88] [Wichaipanitch92].

1) Find a trajectory of the program P.

2) For each action k^p in the trajectory, compute $U(k^p)$, the set of variables that are used in k, and $D(k^p)$, and set of variables that are defined in k^p.

3) Compute the Definition-Use relation, a relation in which one action assigns a value to an item of data and the other action uses that value.

4) Compute the Test-Control relation, capturing the effect between test actions and actions that have been chosen to execute by these test actions.

5) Compute the slicing set S_c and the action set A_c. The Slicing set S_c can be computed using the following definition.

Let C = (x, I^q , V) be a slicing criterion and T be a trajectory on input x. To find the slicing set S_c, we have to find a set A^0 of all actions that have direct influence on V at q and on action I^q. A^0 is defined as follows:

$$A^0 = LD(q, V) \bigcup LT(I^q)$$

where LD(q, V) is the set of last definitions of variables in V at the execution position q, and $LT(I^q)$ is the set of test actions that have Test-Control influence on action I^q.

The slicing set S_c can be found by an iterative method, as the limit of the sequence $S^0, S^1, ..., S^n$, $0 \le n < q$, defined as follows:

$$S^0 = A^0,$$

$$S^{i+1} = S^i \bigcup A^{i+1},$$

where $A^{i+1} = \{ X^p \in M(T) : 1 \le p < q,$

(1) $X^p \notin S^i$, and

(2) there exists $Y^t \in S^i$, $t < q$, X^p **Z** Y^p }

where **Z** = DU $\bigcup$ TC $\bigcup$ IR, and M(T) represents a set of actions in the trajectory T.

Finally, we can get the slice from the following definition:

$$S_c = S_k \bigcup \{ I^q \}$$

where S^k is the limit of the sequence { S^i}.

As stated in the previous section, it is generally difficult and more complex to compute a dynamic slice than a static slice. Moreover, using dynamic slicing to produce slices, one needs to produce an executable code from the original source code. The process of finding slices becomes difficult when programs that are written in C++ are the inputs because of classes, operator overloading, functional overloading, polymorphism, and other features.

2.2.3 Quasi-Static Slicing

Quasi-Static slicing, which falls between static slicing and dynamic slicing, arises from applications in which values of some inputs are fixed while the behavior of the program varies with respect to other variables [Venkatesh91].

A quasi-static slice with respect to variable total at S19 in Figure 2 is obtained by including the statements which check the value of the variable number at line S10. Figure 6 shows the final slice using the definition of quasi-static slicing.

```
#include<stdio.h>                                                    S1
#include<string.h>                                                  S2
main()                                                              S3
{
      int number;                                                   S4
      int fact;                                                     S5
      int total;                                                    S6
      if( number < 0)                                               S10
      {                                                             S11
            printf("error\n");                                      S12
            number = 0;                                             S13
      }
      while(number < 10)                                            S14
      {
            total = total + 1;                                      S15
            number++;                                               S16
      }
printf("total is %d\t factorial is %d\n", total, fact);             S19
}
```

Figure 6. A quasi-static slice of the sample program shown in Figure 2 based on the quasi-static slicing method

2.3 Types of Slices

Slices can be broadly classified into forward slices and backward slices based on the type of computation involved. A backward slice consists of statements or expressions that affect the value of a variable of interest [Venkatesh91] [Venkatesh95]. In computing this type of slices, first the execution trace of a program is recorded and then the dynamic dependence relations are determined backwards.

A forward slice contains statements in a program whose computation is affected by the value of a variable of interest [Venkatesh91] [Venkatesh95]. These slices are computed during the execution of a program without recording the execution trace.

Slices can also be divided into four kinds based on the extent to which the closures on the dependencies are performed [Venkatesh95].

1. Data Dependence: In case of backward slicing, this type of slice contains the previous definition site of a variable of interest and is called Defsite. For forward slicing, it is a set of statements where the current definition for variable of interest is used and is called as Refsite.

2. Data Closure: This type of slice is obtained by performing closure just over data dependencies, starting from a variable of interest. This type is used for debugging, understanding of programs, for maintenance.

3. Data and Control Closure: This type of slice is obtained by performing closure over both data and control dependencies, starting from a variable of interest. This type of slice has applications in testing. The dynamic slice as defined by Agrawal and Horgan [Agrawal, et al.91] falls under this type.

4. Executable: This slice is obtained by performing closure over both data and control dependencies, and including some additional statements that make the slice an executable sub program that depicts the original program's behavior. This type of slice in backward slicing corresponds to Korel and Laski's dynamic slice.

CHAPTER III

C++ PROGRAM SLICER (cppslicer)

3.1 Introduction

Based on a review of the open literature, there seems to be no slicing-based debugging tool currently available for the C++ programming language. By applying static and dynamic slicing techniques, a slicing environment called C++ program slicer (cppslicer) was developed for programs written in a subset of C++. It is developed as a utility program of the UNIX system. The cppslicer tool is written using C++ and the object-oriented methods. It can also be used to slice programs that are written in ANSI C in a more efficient manner. The rest of this chapter discusses the design and implementation issues of the cppslicer tool.

3.2 C++ Program Slicer Overview

Using the cppslicer program, a user can locate the cause of errors in a program. It can be compared with the tools developed by Nanja [Nanja90] and Wichaipanitch [Wichaipanitch92], whose slicers can be used to locate errors for programs that are written using a subset of C. The cppslicer tool was designed and developed in a way to afford ease and convenience to the user. Menus are provided to allow a user to select any one of the number of functions (load, slice, cload help, man, etc.) supported by the cppslicer program.

The cppslicer program was designed and developed using object-oriented methods.

One important aspect of cppslicer is data abstraction. Data abstraction is not violated in any of the classes. Code reusability is one of the other aspects that was taken into consideration while designing and developing the cppslicer program.

3.3 Software Issues

3.3.1 Definitions

The following definitions [Weiser84] [Nanja90] are useful for applying slicing to programs written in a subset of C++.

DEF(n). DEF(n) is the set of variables whose values are changed at statement n [Weiser84]. For example, if n is an assignment statement, the values that appear on the left side of the expression are included in DEF(n). Other variables that can be included are variable declarations and read statements. If the statement does not contain any of these types, then DEF(n) is empty.

REF(n). REF(n) is the set of variables whose values are used at statement n [Weiser84]. Thus, if n is an assignment statement, REF(n) is the set of values that appear on the right side of the assignment. Other variables that can be included in REF(n) are variables that are used in printing and variables that are used to test the test-control relations. If the statement does not contain any of these types, then REF(n) is empty.

Slicing Criteria. A slicing criterion of a program P is a tuple <i, V>, where i is a specific statement in P and V is a subset of the variables in P [Weiser84].

Active Set. An active set is a set of variables at statement K of a program P, whose values just before execution of the statement K might influence the slicing criterion variables V

just before execution of the slicing criterion statement n .

3.3.2 Slicing Criterion

Based on the review of the open literature, there seems to be no algorithms for slicing C++ programs, hence a new method involving both static and dynamic slicing, which is different from quasi-static slicing [Venkatesh91], was developed.

The new algorithm, analogous to Weiser's algorithm, depends on the backward approach. It differs from a static slice in that the slices generated are executable irrespective of the criterion variable. The static slicing method can be used to generate slices of programs that may or may not be executable.

The new algorithm differs from dynamic slicing too. A dynamic slice of a program can be computed from the executable part of the source code [Korel and Laski88]. That is, an executable trace is developed from the original source code. This trace can be compiled into a dynamic dependence graph that links each instance of the use of a variable to its definition. Each occurrence of a definition is linked to the corresponding occurrences of the variable's uses (if any) in its defining expression, also to the variable's uses (if any) in expressions that are control dependent [Venkatesh95]. The new algorithm uses the original source program itself to produce slices that are executable.

This proposed method deviates also from quasi-static slicing [Venkatesh91]. Quasi-static slicing is dependent on the variables that affect the overall performance of the slices. Such variables are not considered in developing the new algorithm. On the other hand, the proposed method is similar to quasi-static slicing in the aspect that it uses both the static and

the dynamic slicing methods.

```
type RefDef is record of
      variable          :     StringClass;
      linenumber        :     int;
      PartsType         :     int;
end of record

type RefClass is record of
      PartsTytpe        :     int;
      Beginline         :     int;
      EndLine           :     int;
      rettype           :     RETTYPE;
      ClassName         :     StringClass;
end of record

type node is record of
      linenumber        :     1..n statements;
      INCLUDE           :     int;
      metoo             :     int;
      lineinformation   :     StringClass;
      RefHeader         :     array[1..n] of RefDef;
      DefHeader         :     array[1..n] of RefDef;
end of record
```

Figure 7. Slicing data structures

3.3.3 Data Structures

The data structures and algorithms that were designed and used in implementing the slicing algorithm are shown in Figure 7. The figure shows part of the data structures that were used in the implementation of cppslicer. Classes that were designed and implemented for cppslicer are discussed in the section on implementation issues (Section 3.4).

RefClass is used to store information regarding the criterion class. Similar to the criterion variable, criterion class is a class of interest with respect to which slicing is performed. For the cppslicer program, there can be at most one class of interest. The

information includes parts type (public or private), member functions, their beginning and terminating lines, and the type of value that is to be returned from the member functions (void, int, ..., etc.)

RefHeader and DefHeader are used to store the variable that are referenced and defined, which were defined as REF(n) and DEF(n) [Weiser84]. The additional information includes variable type (global, local, private, or public) and their line numbers.

In node type, INCLUDE is used to store a set of line numbers that are used for computing the scope of influence of each line.

3.3.4 Slicing Algorithms

For finding out the scope of influence of control statements and functions, the INCLUDE set is used. The INCLUDE set for each statement is initially set to NULL. If a line number X contains a control statement or function, then X will contain the starting and ending line numbers in it. Due to the complexity of the C++ language, and in order for the cppslicer program to be able to handle classes and their member functions, a set of rules that are simpler than the *metoo* set [Samadzadeh and Wichaipanitch93] were made as shown in Figure 8. Based on the rules in Figure 8, Figure 10 shows the INCLUDE set for the program shown in Figure 9, which calculates the sum and product of the first n numbers.

```
1. For any line number consisting of such instructions as control, the
include set will contain
      1.1 Its line number.
      1.2 Line number representing the beginning of the scope of
          influence.
      1.3 Line number representing the end of the scope of the
          influence.
2. If a control statement appears with one statement, it will be marked
with metoo set.
3. In case of member functions of the criterion class, the include set
will contain
      3.1 Member function's line number in the class.
      3.2 Starting line where the actual member function is defined.
      3.3 Ending line where the actual member function is defined.
4. If any other statements appear, their metoo and INCLUDE are set to NULL
```

Figure 8. Rules for computing the INCLUDE set

```
 1  #include<stdio.h>
 2  #include<iostream.h>
 3  #include<string.h>
 4
 5  main()
 6  {
 7    int sub = 1;
 8    int sum = 0;
 9    int fact = 1;
10    int num = 0;
11    cout << "Enter the Number" << endl;
12    cin >> num;
13    if(num < 0)
14    cerr << "error in input " << endl;
15    while(sub <= num)
16    {
17      sum = sum + sub;
18      sub++;
19    }
20    int count = 1;
21    while(count <=  num)
22    {
23      fact = fact * count;
24      count++;
25    }
26    cout << "sum is " << sum << "\nProduct is " << fact << endl;
27  }
```

Figure 9. A sample program to compute the sum and product of the first n numbers

Line Number	INCLUDE		Metoo
	Begin	End	
1	0	0	0
2	0	0	0
3	0	0	0
4	0	0	0
5	0	0	0
6	0	0	0
7	0	0	0
8	0	0	0
9	0	0	0
10	0	0	0
11	0	0	0
12	0	0	0
13	0	0	14
14	0	0	0
15	16	19	0
16	0	0	0
17	0	0	0
18	0	0	0
18	0	0	0
20	0	0	0
21	0	0	0
22	0	0	0
23	0	0	0
24	0	0	0
25	0	0	0

Figure 10. Results of INCLUDE sets that are computed for the sample program shown in Figure 9

The algorithm that was designed and implemented to compute the INCLUDE set is shown in Figure 11.

```
function ComputeIncludeSet( var begin : array[1..n] of node; var end
                            :array[1..n] of node; linenumber : int)
var
info : StringClass;
begin
1       info := 0;
2       while( begin != end)
        begin
        2.1     info = begin[linenumber] -> LineInformation;
        2.2     check for member function or check for control
                statement;
        2.3     if success store the starting line in INCLUDE and
                find for end statement;
        2.4     linenumber = linenumber + 1;
        end;
end;
```

Figure 11. Algorithm to compute the INCLUDE set

The slicing algorithm for cppslicer is based on Weiser's backward approach. The set of rules that are followed in finding a slice for a given program are shown in Figure 12.

```
1. Get the criterion line.
2. Get the corresponding member function from RefClass.
3. Using Weiser's backward approach.
       3.1 Compute straight line slicing based on the straight line
           slicing algorithm.
       3.2 Compute test control statements, the control statements that
          influence the statements of interest.
       3.3 Include Member function's (if any) that fall in between
           statements of interest.
4. Mark all other statements (to avoid their presence in the slice)
```

Figure 12. Rules for producing slices

As mentioned earlier, information regarding each member function is stored in a data structure called RefClass in the form of a linked list. The criterion line (the line at which slice to be performed) should be decided before starting to slice a program. Depending on the line number, the member function can be selected by searching the linked

list.

For each statement, compute *active set* (the criterion variables and a set of variables that influence the criterion variables), *def* (the set of variables that are defined), and *ref* (the variables that are referenced). A statement is marked for a slice, if the intersection of the active set and def is not empty. The same statement is checked for control statements. If there exists any control statements, then check for the presence of statements that are marked to include in the slice in its scope of influence. If found, then include the control statement in the slice. Finally, the statement is searched for any member functions that belong to the criterion class (class of interest). If found, the statement is included in the slice. The slicing algorithm is shown in Figure 13. Figure 14 shows the complete slicing algorithm that was designed and implemented for program slicing.

3.4 Implementation Issues

In designing and implementing program slicing for programs written in a subset of C++, the programming language C++ and object-oriented methods [Budd91] were employed.

The cppslicer program consists of four classes: StringClass, LoaderClass, SliceClass, and ProcessorClass. LoaderClass is used to load the program to be sliced, and compute the referenced and defined variables for each statement. SliceClass is basically designed and implemented to slice a program based on the class of interest, the variable of interest, and the line of interest. It sets slice marks to TRUE for the statements that influence the variable of interest and prints the final output onto the screen or saves into

```
function Slice(var begin : array[1..n] of node; var end :array[1..n]
      of node; Activeset : StringClass; line : int)
var
      refs: array[1..n] of RefDef;
      defs: array[1..n] of RefDef;
      beginline : int;

begin
1. Beginline is the starting line of the member function. It will be
   obtained from RefClass.
2. while( line >= beginline)
   begin
      2.1   ref := begin[line] -> RefHeader;
      2.2   def:= begin[line] -> DefHeader;
      2.3   if( ActiveSet ∩ def != NULL)
            begin
                  ActiveSet = ActiveSet U ref;
                  begin[line] -> SLICE = TRUE;
            end;
      2.4   Check for the presence of member functions.
      2.5   if(control statement)
            begin
                  -Check each statement for slice mark within
                   the scope of influence of control statement,
                   if marked, include the control loop in the slice,
                  -else proceed to the next statement
            end;
      2.6   line := line -1;
      end;
end;
```

Figure 13. Slicing algorithm for member functions

```
Procedure SliceProg( filename : StringClass; n : integer; RefClass :
      array[1..n] of RefClass)
var
      beginline : integer;
      endline   : integer;
      status    : integer;
      sliceline : integer;
      var       : StringClass;
      activeset : StringClass;

begin
1     if(bad filename) then
       - show error and return;
2     status := LaodSouceProgram(filename)
3     if(status == error) then
       - show error and return;
4     Get the class line number;
5     Get public and private members of class;
6     Insert class members into RefClass;
7     while(RefClass != NULL)
      begin
      7.1   Set all the statement's slice marks to false;
      7.2   Find all control statements in the program and mark
            their starting and ending line;
      7.3   BeginSlice := RefClass -> beginline;
      7.4   EndLine  := RefClass -> endline;
      7.5   ComputeRefDef(BeginLine, EndLine);
      7.6   RefClass := RefClass -> next;
      end;
8     sliceline := Get the slice line;
9     activeset := Load the active variables ;
10    BeginSlicingProgram(sliceline, activeset);
11    Print the slice;
end;
```

Figure 14. Complete slicing algorithm

a file. While designing the cppslicer tool, functional overloading and operator overloading techniques were adopted. The classes of the cppslicer program have "has a relation" thus avoiding the usage of inheritance [Budd91]. Overall, the program consists of 4500 lines of undocumented C++ code.

Linked lists were used to store the program information. The process of searching and inserting was made easy by keeping the addresses of statements such as the starting and ending line of each member function and the line number of member functions defined in a class.

The cppslicer program consists of a main program, five sub programs, and five header programs. The sub programs are Loader.C, Slice.C, FlowControl.C, Proc.C, and String.C. The header programs are defs.h, Loader.h, slice.h, Proc.h, and string.h.

3.5 Slicing Based Metrics

Program slicing can be used in two different areas [Weiser81]. First, program slicing is used for the debugging and testing of programs. Second, program slicing can be used to obtain slicing based metrics which provide useful information about the structure of a program. Weiser proposed five slicing based program metrics.

1. Coverage: Coverage compares the lengths of slices to the length of the entire program. Coverage might be expressed as the ratio of mean slice length to program length. A low coverage value indicates a program which has several distinct conceptual purposes.

2. Overlap: Overlap is a measure of how many statements in a slice are found only in that slice. This could be computed as the mean of the ratios of non-unique to unique statements in each slice. A high overlap might indicate very interdependent code.

3. Clustering: Clustering reveals the degree to which slices are reflected in the original code layout. It could be expressed as the mean of the ratio of statements formerly

adjacent to total statements in each slice. A low cluster value indicates that the slices are intertwined like spaghetti, while a high cluster value indicates that the slices are physically reflected in the code by statement groupings.

4. Parallelism: Parallelism is the number of slices that have few statements in common. Parallelism could be computed as the number of slices which have a pair-wise overlap less than a certain threshold. A high degree of parallelism would suggest that assigning a processor to execute each slice in parallel could give a significant program speed up.

5. Tightness: Tightness measures the number of statements that are in every slice, expressed as a ratio over the total program length. The presence of relatively high tightness might indicate that all the slices in a subroutine really belonged together because they all shared certain activities.

The cppslicer program is a tool developed using the program slicing method involving both static slicing and dynamic slicing approaches for simple programs written using a subset of C++. The size of the slice obtained depends on three factors: (1) The class name with respect to which the slice is marked , (2) The selected variables (private members of a class), and (3) The line number of the selected variables. C-Debug [Wichaipanitch92], and C-Sdicer [Nanja90] were designed and developed to handle simple programs written using a subset of C, and they adopted dynamic slicing and static slicing methods, respectively. Due to the difference in slicing criteria and programming languages used, the results that are obtained for slicing based metrics under cppslicer cannot be compared with the results that are obtained using C-Debug and C-Sdicer debuggers.

Some slicing based metrics were computed for six programs (three written in C++ and three in C) taken from different published sources. The values generated are given in TABLE I. The programs in this computation are listed in Appendix A. In TABLE I, P1, P2, P3, P4, P5, P6 represent the programs used in the computation of slicing based metrics.

TABLE I
SLICING-BASED METRICS

	C++			C		
Metric	P1	P2	P3	P4	P5	P6
Size(# of Lines)	60	111	52	45	67	62
Output Var's	3	2	5	3	10	1
Coverage	0.83	0.90	0.81	0.82	0.72	0.70

3.6 Evaluation

The cppslicer tool was evaluated for its effectiveness and utility as a debugging tool. A group of computer science students at Oklahoma State University were asked to test and evaluate the cppslicer tool using their own programs. The following are the results and the conclusions based on the responses submitted.

1. The students involved found that the cppslicer tool is of use in finding out the cause and location of error in their programs.
2. The graduate students that were involved felt that the cppslicer tool should be enhanced to handle programs containing multiple classes, functional overloading, and inheritance.
3. None of the subjects were aware of program slicing and its application to the debugging C++ programs.

3.7 Advantages and Limitations of cppslicer

The cppslicer program can be used to generate slices for programs written using a subset of C++ based upon a set of user-specified classes and variables that are present in the program. The cppslicer program is equipped to handle user-defined classes, its member

program. The cppslicer program is equipped to handle user-defined classes, its member functions, "this", an explicit pointer to a class, operator overloading within a class, and calling member functions within member functions.

The cppslicer program consumes considerably more time to compute slices than does static slicing. The tool is not equipped to slice programs with respect to user-defined variables, variables within structures, or unions. The cppslicer program cannot handle classes involving inheritance, functional overloading, or classes containing inline and friend functions.

CHAPTER IV

SUMMARY AND FUTURE WORK

4.1 Summary

In Chapter I, a brief overview of debugging was presented. It also introduced the concept of program slicing. The first chapter concluded by providing the purpose of the study and outlining the organization of this report.

In Chapter II, the different debugging approaches and a number of tools, which are available on UNIX systems, were presented. Chapter II described different program slicing methods: static slicing, dynamic slicing, and quasi-static slicing. Under dynamic slicing, the existing methods by Agrawal and Horgan [Agrawal and Horgan90], Korel and Laski [Korel and Laski90] and Samadzadeh and Wichaipanitch [Samadzadeh and Wichaipanitch93] were introduced. Chapter II ends by briefly discussing the advantages and disadvantages of each method.

Chapter III provided information about the debugging tool called cppslicer, which was designed and developed for this study. The steps involved in the design approach, the algorithms used, and the advantages and limitations were discussed in that chapter.

The cppslicer program is an interactive debugging tool designed and developed for novice programmers to debug their programs written in C++. Using cppslicer, a user can locate the cause of errors. It is designed to handle the basic C++ commands. It is developed as a utility program for UNIX. In designing and implementing the cppslicer program, the

expensive to use C++ when compared to other object-oriented programming languages (OOPs) such as Smalltalk or Simula, programming in C++ does not require a graphics environment, and since C++ is a strongly-typed language, programs do not incur run-time overhead from type checking or garbage collection.

As a rudimentary C++ slicer, cppslicer can generate a slice of a source program which preserves part of the original program's behavior for a specific input. The cppslicer tool was designed and developed to handle programs written in a subset of C++ involving data structures such as classes, member functions of classes involving operator overloading, pointers to classes (the implicit "this" pointer), straight-line codes, control statements, function calls, and expressions containing simple pointers to char, int, or float. The cppslicer program cannot handle structures, unions, user-defined variable, functional overloading, constructors, destructors, inheritance, or const and inline functions.

4.2 Future Work

The cppslicer program can be enhanced in future to handle expression involving pointers to structures and unions. It can also be enhanced to handle inheritance, constructors, destructors, and functional overloading.

Other significant improvements that can be made to cppslicer program include the slicing of the function members that are called from within other member functions. At present the cppslicer program includes all the statements of the member functions which are called from other member functions. It can be further enhanced to slice member functions based on private variables and the arguments that are passed to the member functions.

based on private variables and the arguments that are passed to the member functions.

REFERENCES

[Agrawal and Horgan90] Hiralal Agrawal and Joseph R. Horgan "Dynamic Program Slicing", *Proceedings of ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, pp. 1-19, White Plains, NY, June 1990.

[Agrawal, et al.91] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Stafford "Dynamic Slicing in the Presence of Unconstrained Pointers", *Technical Report SERC-TR-93-P,* Software Engineering Research Center, Department of Computer Science, Purdue University W. Lafayette, IN. Also published in the *Proceedings of the Fourth ACM/IEEE-CS Symposium on Testing, Analysis, and Verification (TAV 4)*, pp. 1-19, October 1991.

[Borland90] *Turbo C -- UPDATE*, Borland International, Inc., CA, 1990

[Brown and Sampson73] A. R. Brown and W. A. Sampson, *Program Debugging: The Prevention and Cure of Program Errors,* MacDonald/American Elsevier, NY, 1973.

[Budd91] Timothy Budd, *An Introduction to Object-Oriented Programming*, Addison-Wesley Publishing Company, Inc., Reading, MA, 1991.

[Korel and Laski88] Bogdan Korel and Janusz Laski, "Dynamic Program Slicing", *Information Processing Letters,* Vol. 29, No.3, pp. 155-163, October 1988.

[Korel and Laski90] Bogdan Korel and Janusz Laski, "Dynamic Slicing of Computer Programs", *Journal of Systems and Software*, Vol. 13, pp. 187-195, 1990.

[Nanja90] Sekaran Nanja, "An Interactive Debugging Tool for C Based on Program Slicing and Dicing, " MS Thesis, Computer Science Department, Oklahoma State University, Stillwater, OK, May 1990.

[Samadzadeh and Wichaipanitch93] M. Samadzadeh and W. Wichaipanitch, "An Interactive Debugging Tool for C Based on Dynamic Slicing", *Proceedings of the 1993 ACM Computer Science Conference*, Indianapolis, IN, pp. 30-37, February 1993.

[Seviora87] R. E. Seviora, "Knowledge-Based Program Debugging Systems", *IEEE Software*, Vol. 4, 3, pp. 20-32, May 1987.

[Tassel74] Dennie V. Tassel, *Program Style, Design, Efficiency, Debugging, and Testing*, Prentice-Hall, Inc., NJ, 1974.

[Venkatesh91] G. A. Venkatesh, "The Semantic Approach to Program Slicing",

Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation, Toronto, Ontario, Canada, pp. 107-119, June 1991.

[Venkatesh95] G. A. Venkatesh, "Experimental Results from Dynamic Slicing of C Programs", *ACM Transactions on Programming Languages and Systems*, Vol. 17, No. 2, pp. 197-216, March 1995.

[Weiser81] M. Weiser, "Program Slicing", *Proceedings of the Fifth International Conference on Software Engineering*, San Diego, CA, pp. 439-449, March 1981.

[Weiser82] M. Weiser, "Programmers Use Slices When Debugging", *Communications of the ACM*, Vol. 25, No. 7, pp. 446-454, July 1982.

[Weiser84] M. Weiser, "Program Slicing", *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 4, pp. 352-357, July 1984.

[Wichaipanitch92] W. Wichaipanitch, "An Interactive Debugging Tool for C Based on Dynamic Slicing and Dicing", MS Thesis, Computer Science Department, Oklahoma State University, Stillwater, OK, December 1992.

APPENDICES

APPENDIX - A

USER MANUAL FOR CPPSLICER

The cppslicer program is a debugging tool based on program slicing techniques. It runs under the UNIX environment (DYNIX/ptx) on the Sequent S/81 machine. The cppslicer program can be used to locate the cause of errors in programs written in a subset of C++. The basic information about the tool can be obtained by typing *info* at the cppslicer prompt. A menu listing all the commands can be obtained by typing help at the prompt. The menu is shown below.

```
-----------------------------------
|          Welcome to             |
|    C++  Program Slicer.         |
|     Type "info" to get          |
| information about the tool.     |
----- ---------------------------
```

Command	Usage	Purpose
cload	cload <filename>	Used to load programs written in C or in C++ without classes
edit	edit <filename>	Edit a program in cppslicer
help	help	Display list of commands that can be used in cppslicer
load	load <filename><classname>	Used to load a program into cppslicer written in C++ containing classes
man	man <command>	Give help on the usage of a command
q(uit)	q or quit or exit	Used to exit from the tool
save	save <filename>	Used to save the slice into a file
slice	slice<line no><var>	Used to slice a program residing in cppslicer
type	type	Display program resident in cppslicer

The help menu displayed when cppslicer is started

The cppslicer debugging tool can be invoked at the UNIX prompt by typing its pathname. Then the cppslicer program is invoked and a help menu appears on the screen. To find information about a command, type *help* and the command name as shown below.

cppslicer> help slice

The response provides help about the usage of that command. To generate a slice for a particular program, one needs to load the program into cppslicer tool. This can be done by typing *load <filename><classname>*, where the filename denotes the program name and the classname is the class of interest with which the slice marks will be made. After loading the program into cppslicer, other commands that can be used are slice and save. The list of commands that are available in cppslicer are shown below.

cload	Another way of loading a program into cppslicer. It is used to load programs that are written in C.
echo	Used to display the list of commands that are available in cppslicer. It echoes the list of commands whenever a user types an invalid command. This command cannot be executed by users.
edit or VI	Used to edit a program in the cppslicer environment using the vi editor.
help	Provides a list of the commands that are available in cppslicer, their usage, and their purpose.
man	Provides information about specific commands; explaining a purpose of the command and its usage.
save	Saves the resulting output into a file. The usage of the command is *save* followed by a file name into which the output should be directed.
slice	Used to generate a slice for a program that is resident in cppslicer. The usage of the command is *slice <linenumber><criterion variables>*. The linenumber denotes the line with respect to which a slice is to be generated, and the criterion variables are those with respect to which a slice can be obtained.
type	Displays the program resident in the cppslicer with line numbers on left side. This helps to locate the variables of interest and their line numbers.

! or /	Invokes the UNIX shell. Any of the system commands can be executed by typing / or ! followed by the command.

APPENDIX B

GLOSSARY AND TRADEMARK INFORMATION

GLOSSARY

Action:	A instruction k at position p in the trajectory is represented as (k, p). The pair is replaced by k and referred to as an action.
Amortized Time:	The average time per operation over a worst-case scenario of operations.
$D(X^p)$:	The set of variables that are defined in action X .
Execution History:	Execution history at any node denotes the partial program execution until that node.
Flow Graph:	The Flow graph of program P is a tuple < V, A>, where V is the set of vertices that represent simple statements such as read and write and conditional expressions such as if-then-else and while-do, and A is the set of directed edges between pairs of vertices in V.
Last Definition:	The last definition of variable v at node t is the action that last assigned a value to v when it was reached on trajectory T.
M(T):	A set of actions in a given trajectory T.
Program Dependence Graph:	The Program dependence graph of a program is obtained by the union of data and control dependence graphs, where the data dependence graph contains a set of edges that reflect data dependencies among nodes that are in the flow graph of the program, and the control dependence graph contains a set of edges that reflect control dependencies among the nodes that are in the flow graph of a program.
Reaching Definition:	If F represents a flow graph, n a node in the flow graph, and var a variable in the flow graph, then the set of all reaching definitions of var at n in F will be the set of all nodes in F at which var is assigned a value and control can flow from that node to node n without any redefinitions of var along the control flow path.

Slicing Criterion:	A slicing criteria can be expressed as the set of values of some set of variables at a statement .
Slicing Set:	A set of actions that have influence on the variable of interest.
$TC(X^p)$:	Test-Control relation, a binary relation on M(T) capturing the effect between test actions and actions that have been chosen to execute by these test actions. X denotes a action.
Test Action:	An action X is a test action if X is a test instruction.
Test Case:	A test case consists of a specific set of input values for a program.
Trajectory:	A trajectory is a feasible path that has been executed for some input.
$U(X^p)$:	The set of variables that are used in action X^p.

TRADEMARK INFORMATION

Sequent S/81: Sequent S/81 is a registered trademark of the Sequent Computer Systems, Inc.

UNIX: UNIX is a registered trademark of AT&T.

VAX: VAX is a registered trademark of Digital Equipment Corporation.

APPENDIX C

SAMPLE PROGRAMS USED FOR COMPUTING THE SLICING-BASED METRICS

The six sample programs shown below are used for the computation of the slicing-based metrics.

```
//////////////////////////////////////////////////////////////////////
//                          TEST PROGRAM 1
// A program to compute sum, average, and product of the first n numbers.
//////////////////////////////////////////////////////////////////////
#include<stdio.h>
#include<iostream.h>
#include<string.h>


class TestClass{
  private:
    int sum, fact;
    float average;
  public:
    TestClass(); // Contrsturctor
    void Calculate(int);
    void PrintVal();
};

// This is a constructor which will be called automatically without being
// called from the original source program. It initializes the values of
// sum, fact, and averages to 0.

TestClass::TestClass()
{
  sum = 0;
  average = 0;
  fact = 0;
}

// This member function calculates the sum, average, and product of the
// first n numbers denoted by variable num.

void
TestClass::Calculate(int num)
{
  int sub = 1;
  cout << "Enter the Number" << endl;
  cin >> num;
  while(sub <= num) {
    sum = sum + sub;
    sub++;
  }
  int count = 1;
  fact = 1;
  while(count <=  num)
  {
    fact = fact * count;
    PrintVal();
```

```
    count++;
  }
  average = (float) sum / num;
  printf("sum is %d\n", sum);
}

// This function prints the final result onto the standard output

void
TestClass::PrintVal()
{
  printf("Sum Is %d\n", sum);
  printf("Fact Is %d\n", fact);
  printf("Average Is %f\n", average);
}

main()
{
  int number;
  TestClass t1;
  printf("enter the number : ");
  scanf("%d", &number);
  t1.Calculate(number);
  t1.PrintVal();
}


//////////////////////////////////////////////////////////////////////////
//                        TEST PROGRAM 2
// Copyright (c) 1991 AT&T Bell Laboratories, All Rights Reserved
// Published in ``A C++ Primer''  by Stanley Lippman, Addison-Wesley.
// This program is directly downloaded. Hence the program is not
// documeneted and is presented as is.
// This program performs basic operations on strings like string copy,
// concatination, etc.
//////////////////////////////////////////////////////////////////////////

#include <iostream.h>
#include <string.h>
#include <assert.h>
#include <iomanip.h>

class String {
private:
    int len;
    char *str;
public:
    String(char*);
    String(const String&);
    String();
    String& operator=(const String&);
    int getLen();
    String& operator+=(const String&);
    operator <(String& s);
    operator >(String& s);
    friend String operator+(String&,String&);
};

String::String()
{
      len = 0;
```

```
        str = 0;
}

String::String( char *s )
{
        if ( !s ) {
                len = 0;
                str = 0;
        }
        else {
                len = strlen( s );
                str = new char[ len + 1 ];
                strcpy( str, s );
        }
}
String::String( const String& s )
{
        len = s.len;
        if ( (str = s.str) == 0)
                return;

        str = new char[ len + 1 ];
        strcpy( str, s.str );
}
String& String::operator=( const String& s )
{
    if (this == &s) return *this;
    delete str;
    len = s.len;
    if ( (str = s.str) == 0 )
        return *this;
    str = new char[ len + 1 ];
    strcpy( str, s.str );
    return *this;
}
int
String::getLen()
{
        return len;
}

String&
String::operator+=( const String &s )
{
    len = len + s.len;
    if ( len <= 0 )
                    return *this;
    char *p = new char[len+1];
    strcpy(p,str);
    strcat(p,s.str);
    delete str;
    str=p;
    return *this;
}
String
operator+(const String &s1, const String &s2 )
{
    String result = s1;
    result += s2;
    return result;
}
int String::operator<(String& s) {
```

```
      if (s.len == 0) return 0;
      if (len == 0) return 1;
      return(strcmp(str,s.str)<0) ? 1: 0;
}
int String::operator>(String& s) {
      if (len == 0) return 0;
      if (s.len == 0) return 1;
      return(strcmp(str,s.str)>0) ? 1: 0;
}
main()
{
      String S1("Ramaka");
      String S2("Hello");
      String S3;
      S3 = S1 + S2;
      cout >> S3.getlen()>> endl;
}
```

```
////////////////////////////////////////////////////////////////
//                          TEST PROGRAM 3
// This program performs the operations of a clock. It contains
// a clock class. This program is published in ``C++ for C Programmers``
// by Ira Pohl, The Benjamin/Cummings Publisihing Company, Inc., CA,
// 1993
////////////////////////////////////////////////////////////////
#include<iostream.h>
#include<string.h>
class clock{
private:
      int tot_secs, secs, mins, hours, days;
public:
      clock(int);                    // Constructor
      void print();                  // Function to print private parts
      void tick();
      void reset(const clock& c);
};

// This constructor initializes all the variables based on variable i.
// It then calculates the corresponding values of secs, mins, hours,
// and day.
clock::clock(int i)
{
      tot_secs = i;
      secs = tot_secs % 60;
      mins = (tot_secs / 60) % 60;
      hours = (tot_secs / 3600) % 60;
      days = tot_secs / 86400;
}

// This function is to print the final result onto the screen.

void clock::print()
{
      cout << days << "d : " << hours << "h:" << mins << "m:" << secs <<
"s:"<<endl;
}
// This function increments the total seconds by one sec and updates the
```

```
// other values.

void clock::tick()
{
      clock temp = clock(++tot_secs);
      secs = temp.secs;
      mins = temp.mins;
      hours = temp.secs;
      days = temp.days;
}


// This function is used to reset one clock variable with another clock
// variable. It is like a copy constructor.

void clock::reset(const clock& c)
{
      tot_secs = c.tot_secs;
      secs = c.secs;
      mins = c.hours;
      hours = c.hours;
      days = c.days;
}
main()
{
      int i;
      clock c1(150);
      for(i=0; i < 20; i++)
          c1.tick();
      c1.print();
}

////////////////////////////////////////////////////////////////
//                          TEST PROGRAM 4
// This program inputs and echoes back integers, beginning a new
// output line at each point where a comma appears in the input,the
// total of all the integers on that line is displayed. The input must
// itself consists of only one line. Any characters other than digits
// and commas are ignored, except as delimiters for the numbers. The new
// line is used to detect the end of the line.
// This program is scanned as it is from Nanaj's MS thesis [Nanja90].
////////////////////////////////////////////////////////////////
#include <stdio.h>
main()
{
      char character, last_char;
      int line_total, next_line, current_number;
      line_total = 0;
      next_line = 2;
      current_number = 0;
      last_char = 0;
      printf("Type a line of integers, with a comma everywhere\n");
      printf("the line is to split. Any other characters\n");
      printf("are ignored: \n\n");
      scanf("%c", &character);
      printf("Line 1> ");
      while (character != '\n')
      {
            if(character == ',')
            {
                  if(last_char >= '0' && last_char <= '9')
                  {
```

```
                        line_total += current_number;
                        current_number = 0;
                }
            printf(" < total: %d\nLine %d> ", line_total, next_line);
            line_total = 0;
    next_line++;
            }
            else
            {
                if(character >= '0' && character <= '9')
                current_number = (current_number * 10) + (character -
'0');
                else
                {
                        if(last_char >= '0' && last_char <= '9')
                        {
                                line_total = line_total + current_number;
                                current_number = 0;
                        }
                }
            }
            last_char = character;
            scanf("%c",&character);
      }
      printf("< total: %d\n", line_total);
}


//////////////////////////////////////////////////////////////////
//                        TEST PROGRAM 5
//      This program generates multiple coin toss samples. This program
//  is scanned from Nanja's MS thesis [Nanja90].
//////////////////////////////////////////////////////////////////

#include <stdio.h>
#define MAX_RAND 2000
#define MODULUS 327681
#define SEMI_MOD (MODULUS %2)
main()
{
      int index,start,nr_trials,nr_iter;
      int head,tail,h_lead,t_lead,iter,curr_seed;
      int mult1,mult2,incr1,incr2;
      float ratio, lead_sum, side_sum;
      float d_vals[MAX_RAND];

      head = tail = h_lead = t_lead = 0;
      printf("\n Starting seed?");
      scanf("%d",&curr_seed);
      printf("\n Sample size?");
      scanf("%d",&nr_trials);
      printf("\n Number of samples to generate?");
      scanf("%d",&nr_iter);
      printf("\n First multiplier?");
      scanf("%d",&mult1);
      printf("\n First increment?");
      scanf("%d",&incr1);
      printf("\n Second multiplier?");
      scanf("%d",&mult2);
      printf("\n Second increment?");
      scanf("%d",&incr2);
      printf("Starting seed = %d\n\n",curr_seed);
```

```
      printf("generating random values......\n");
      for(iter = 0; iter < nr_iter; iter++)
      {
      head = 0;
      tail = 0;
      h_lead = 0;
      t_lead = 0;
      for( index = 0; index < nr_trials; index++)
            {
                  if(curr_seed >= SEMI_MOD)
       start = (multl * curr_seed + incrl) % MODULUS;
                  else
       start = (mult2 * curr_seed + incr2) % MODULUS;
      if(start)
       head++;
      else
                        tail++;
                  if(head > tail)
                        h_lead++;
                  else if(tail > head)
                        t_lead++;
            }
            printf("%3d heads; %3d tails;",head,tail);
            printf("H leads = %3d; T leads = %3d", h_lead, t_lead);
            if(h_lead > t_lead)
            {
                  ratio = (float) h_lead / (h_lead + t_lead);
            }
            else
            {
                  ratio = (float) t_lead / (h_lead + t_lead);
            }
            d_vals[iter] = ratio;
            lead_sum = lead_sum + ratio;
            if(head > tail)
            {
                  side_sum = side_sum + (float) head / nr_trials;
            }
            else
            {
                  side_sum = side_sum + (float) tail / nr_trials;
            }
            printf("ratio = %.41f\n",ratio);
      }
      printf("\n DONE \n");
      printf("side_sum == %.41f; mean side lead == %.41f\n",
side_sum,side_sum / nr_iter);
      printf("lead_sum == %.41f; mean lead == %.41f\n",
lead_sum,lead_sum / nr_iter);
}


////////////////////////////////////////////////////////////////
//                      TEST PROGRAM 6
//      This program computes correlation coefficients. This program
//  is scanned from Nanja's MS thesis [Nanja90].
////////////////////////////////////////////////////////////////


#include <stdio.h>
#define MAX_VALS 50
#define MAX_STR 100
```

```
main()
{
	float c_vals[MAX_VALS];
	float d_vals[MAX_VALS];
	float sum1,sum2,var1,var2;
	float coeff,co_vari,numer,denom;
	int index ,n1,n2;
	char *null_str = "";
	char info[MAX_STR];

	printf(" Enter values for group 1.\n");
	printf("?");
	gets(info);
	index = 0;
	while( strcmp(info, null_str) != 0)
	{
		c_vals[index] = atoi(info);
		++index;
		printf("?");
		gets(info);
	}
	n1 = index; printf(" Enter values for group 2.\n");
	printf("?");
	gets(info);
	index = 0;
	while( strcmp(info, null_str) != 0)
	{
		d_vals[index] = atoi(info);
		++index;
		printf("?");
		gets(info);
	}
	n2 = index;
	if (n1 = n2)
	{
		sum1 = 0.0;
		for(index = 0; index < n1; index++)
			sum1 = sum1+ c_vals[index];
		sum2 = 0.0;
		for(index = 0; index < n1; index++)
			sum2 = sum2 + c_vals[index];
		co_vari = 0.0;
		for(index = 0; index < n1; index++)
			co_vari = co_vari + (c_vals[index] * d_vals[index]);
		numer = co_vari - (sum1 * sum2);
		var1 = 0.0;
		for (index = 0; index < n1; index++)
			var1 = var1 + (c_vals[index] * c_vals[index]);
		for(index - 0; index < n1; index++)
			var2 = var2 + (d_vals[index] * d_vals[index]);
		denom = (var1 - sum1 * sum2) * (var2 - sum2 * sum2);
		denom = sqrt(denom);
		if(denom != 0)
			coeff = numer / denom;
		printf("r == %7.311f\n",coeff);
	}
	else
		printf("Arrays must be the same size.\n");
}
```

APPENDIX D

CPPSLICER SOURCE CODE LISTING

The cppslicer program is a slicing-based debugging tool for a programs written in a subset of C++ that runs under UNIX (DYNIX/ptx) on the Sequent S/81 machine. This tool can be used to debug programs containing simple classes. It cannot handle inheritance, functional overloading, or friend and inline functions.

```
////////////////////////////////////////////////////////////
//                              cppslicer
// The source code consists of five programs, Loader.C,
// Slice.C, FlowControl.C, Proc.C, and String.C. It consists of
// four classes, LoaderClass, SliceClass, ProcessorClass, and
// stringClass. LoaderClass is used to load program, set the
// slice marks, store the public  and  private parts (if any)
// into a linked list.  SliceClass is used to slice the
// program with respect to the  given variable and line
// number. ProcessorClass handles all the input and output issues.
////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////
//
//                          Loader.h
////////////////////////////////////////////////////////////////////
#include<iostream.h>
#include<string.h>
#include"string.h"
#include"defs.h"

// Structure that can store the referenced and defined values in each
// statement of the program.

struct RefDef{
       StringClass variable;      // stores the information of each line
       int linenumber;            // stores the statement line number
       int PartType;              // if statement is a declaration, store its
                                  // type
       struct RefDef *next;
};

// Structure that stores the information of a class of interest. It stores
// parts type (public or private) and their information.

struct RefClass{
       int PartsType;             // part type, i.e., public or private
       int BeginLine;             // start line of the member function
       int EndLine;               // end line of the definition (function)
       RETTYPE rettype;           // type of value that is returned from that
                                  // member function.
       StringClass Member;
       StringClass ClassName;
       RefClass *next;
};

// This structure stores the information of each line of the program loaded into
// cppslicer program.

struct node {
       int LineNumber;                  // Statements line number
       StringClass LineInformation;     // Statement information
```

```
        int Position;                   // Position of the variable
        int SLICE;                      // Flag to identify the slice mark
        int INCLUDE;                    // Flag to include the statement
        BOOLEAN metoo;
        struct node *left;
        struct node *right;
        struct RefDef *RefHeader;
        struct RefDef *DefHeader;
};

// Loader Class is used to load the program into the cppslicer program.
// It computes referenced and defined variables for each statement and
// stores them in Rfdf structure in the form of linked  list.

class LoaderClass{
        private:
          node *Header;
          RefDef *Rfdf;             // Linked list to store referenced and defined
                                    // variables of each statement
          RefClass *SliceClass;     // linked list to store public and private
                                    // parts of a class of interest
          StringClass Buffer;
          StringClass ClassName;
          int LineNumber,Size;
          RETTYPE returntype;
          int Position;
        public:
          int LoadSourceprogram(char *);
          void InsertIntoList();
          void ComputeRefDefForClass(char *);
          void InsertPublicPartsIntoSliceClass(char *);
          int GetClassLineNumber(const StringClass& );
          void GetPublicAndPrivateParts(void );
          int GetVariableType(char *);
          void LoadIntoSliceClass(const int& , const int& , char* );
          void LoadIntoSliceClass(const int&,char*);
          void InsertPrivatePartsIntoSliceClass(char *);
          void PlaceIntoSliceClass(int , char *);
          char* CheckPresence(char*, char*);
          char* CheckPresence(const StringClass& , const StringClass& );
          node* GetHeader(void);
          RefClass* GetRefClass(void);
          char* GetClassName(void);
          int GetPosition(void);
          void MarkAllSlices();
          void PrintProgram();
          void PrintSliceClass();
          void EndLoader();
          void LoadMainParts();
          char* RemoveTabsAndBraces(char* );
          char* RemoveBackSpaces(char *);
          char* RemoveFrontSpaces(char *);
          void RemoveReturnTypes(StringClass& );
          LoaderClass& operator=(LoaderClass&);
          };
////////////////////////////////////////////////////////////////////////////
//                                                                        //
//                            slice.h                                     //
//                                                                        //
////////////////////////////////////////////////////////////////////////////
#include<iostream.h>
#include<string.h>
#include"Loader.h"

// Slice class is designed and developed to slice the program residing
// in the cppslicer program. It computes slices based on the line
// and class of interest. It first checks for the presence of the class
// and line number in the command line. If not found, it will give an error
// message and quits the process of slicing. Once the class and line of interest
```

```
// are identified, it slices the program from the starting line to the line
// of interest. If the statement is qualified to be present in the slice,
// slice mark is set to TRUE. The same set of statements are checked for control
// statements. If the statements present within the control loops are marked
// with slice marks, then the start and end statements of control loop are also
// included in the final slice. Also, the statements which influences the
// control loop are included to make the program executable.
class SliceClass{
      private:
        LoaderClass L1;
        RefDef *RefHeader;
        RefDef *DefHeader;
        RefDef *IoRefHeader;
        RefDef *ActiveVar;
        BOOLEAN RefCounter, DefCounter, IoCounter, WHILE;
      public:
        int GetSliceCriteria();
        int ComputeRefDefForClasses(const StringClass& );
        int ComputeRefDef(int, int);
        int CheckForControlStatement(StringClass );
        void IgnoreSpaces(char *, int& );
        void IgnoreBraces(char *, int& );
        void IgnoreBraces(char*, int&, const char);
        void IgnoreOperands(char*, int& );
        void IgnoreOnlyArthOPerators(char*, int& );
        void IgnoreDigitsOperators(char*, int&);
        int CheckForFor(StringClass );
        int CheckForIOStream(const StringClass&);
        void SetReserveVarIntoRef();
        void MarkAllSlices(int, int);
        void MarkAllSlices(void);
        int CheckForComments(const StringClass& );
        int CheckForComments(const StringClass&, int& );
        int CheckForBlankSpace(const StringClass&);
        int GetBeginLineForClassMemeber(const StringClass&);
        int GetEndLineForClassMemeber(const StringClass&, int);
        int InsertIntoRefDefList(const int& , const int&, char *);
        int MarkSliceForCriterionLine(char *, int);
        void CheckControlStatements(node*, node*);
        int SliceControlLoop(int);
        void SliceControlLoop(node*, node*);
        void PerformSliceOnOriginalProgram();
        IOTYPE GetIoType(const StringClass& );
        BOOLEAN IsAValidDeclaration(const StringClass& );
        void CheckForIoOPerands(const StringClass& );
        void InsertIntoIoList(IOTYPE , const StringClass& );
        void InsertIntoRefList(StringClass& );
        void DeleteRefFromRefList(RefDef*);
        void InsertIoRefIntoDefList(RefDef*);
        void InsertIntoHeaderAllRef(node * , RefDef* );
        void InsertIntoHeaderAllDef(node*, RefDef* );
        void InsertIntoHeader(RefDef*, RefDef* );
        void DeleteRefDefIoRef( RefDef *, const int& );
        int BeginSlicingProgram(int, const StringClass& );
        BOOLEAN CheckForActiveSet(node* );
        void VariableRefered(node* );
        int CheckPresenceOfActiveVars( RefDef*, const StringClass& );
        void InsertActiveVars(const StringClass& );
        void LoadActiveVars(const StringClass& );
        int GetBeginLine(int);
        BOOLEAN IsAControlStatement(const StringClass& );
        void MarkControlFlow(const int&, int, node**);
        void CheckOpenBracket(node* , int& );
        void CheckCloseBracket(node* , int& );
        void DeleteRefDefIoRef(RefDef*);
        void PrintSlice();
        void CopyLoader(LoaderClass&);
        void InsertControlVar(const StringClass& , int );
        void PrintVar( RefDef *);
```

```
        BOOLEAN IsAValidMember( StringClass );
        void MarkControlStatForRefDef(int , int );
        void PrintSlice(int );
        void ControlStatWithInControlState(node *);
        void PrintFinal();
        void DeleteActiveVars();
        void PostSliceMarks(node* , node *);
        void PrintFinalSlice(char *);
        void IncludeMemberFunction(node*);
        int CheckForNew(StringClass );
        void CheckInsideForNew(char*);
        void PrintProgram();
        void Display();
        int ComputeRefDefForMain();
        void Set();
        void PrintActiveVars();
};
////////////////////////////////////////////////////////////////////////
//
//                          Proc.h
//
////////////////////////////////////////////////////////////////////////
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include<ctype.h>
#include"slice.h"

// This Class is used to start the process, load the program, slice
// the program, print the resulting slice, and other functions depending
// on the users interest.

class ProcessorClass{
private:
      BOOLEAN status;                  // A flag to notify that a program
                                       // is loaded into cppslicer.
      SliceClass slicer;
public:
      void StartProcess();
      void LoadProg(char *);
      void SliceProg(char *);
      void PrintProg(char *);
      void DisplayProg();
      void ManCommands(char *);
      void TypeProg();
      void HelpMenu();
      void Echo();
      void CloadProgram(char *);
      void InfoTool();
      void EditProgram(char *);
      void SystemCommand(char *);
};
////////////////////////////////////////////////////////////////////////
//
//                          defs.h
//
////////////////////////////////////////////////////////////////////////
const int MAXLINE=80;
const int INT=1;
const int CHAR=2;
const int FLOAT=3;
const int ERROR=-1;
const int PRIVATE=0;
const int PUBLIC =1;
const int BODY =2;
const int PRESENT=1;
const int NOTPRESENT=0;
const int IOREF=2;
const int REFERED=1;
```

```
const int DEFERED=0;
const char SPACE=' ';
const char TAB='\t';
const char OPENBRACE='{';
const char CLOSEBRACE='}';
const char OPENBRACKET='(';
const char CLOSEBRACKET=')';
const char UNDERSCORE='_';
const char SQOPENBRACKET='[';
const char SQCLOSEBRACKET=']';
const char ENDOFLINE='\0';
const char CARRRETURN='\n';
const char PLUS='+';
const char SUBTRACT='-';
const char DIVIDE='/';
const char MOD='%';
const char MULTIPLY='*';
const char COLON=';';
const char COTE='\'';
const char COMMA=',';
const char PERIOD='.';
const char EQUALTO='=';
const int FOUND= 1;
const int NOTFOUND= -1;
const int SUCCESS=1;
const int FAILURE=1000;


typedef enum{TRUE, FALSE}BOOLEAN;
typedef enum{StartState, LastState, ControlState,SixthState, StrState,
SwitchState, VariableState, AirthematicState, IntermediateState,
LoopingState, PreFinalState, CheckingState }StateType;
typedef enum{ SCANF, GETS, GETCHAR, COUT, CIN, PRINTF, SWITCH, VOIDTYPE} IOTYPE;
typedef enum{ INTIGER, INTS, CHARECTER, CHARS, VOID, FLOATING, FLOATS,
FRIEND, CLASSTYPE} RETTYPE;

////////////////////////////////////////////////////////////////// //
//                    string.h                                     //
//////////////////////////////////////////////////////////////////

#include<iostream.h>
#include<string.h>

class StringClass{
      private:
        int len;
        char* str;
      public:
        StringClass();                          // default constructor
        StringClass(char*);                     // conversion
                                                // constructor
        StringClass(const StringClass&);        // conversion
                                                // constructor
        char* GetStr(void) const;               // returns the string
        void PutStr(char*);
        void operator=(const StringClass&);     //assigns values to
                                                //given constructor
        StringClass& operator=(const char*);    //assigns values to
                                                //given constructor
        int operator==(const StringClass& );
        int operator!=(const StringClass& );
        char* operator&=(const StringClass&);
        char* operator&&(const StringClass&);
        StringClass& operator+=(const StringClass& );
        friend char* operator<<(const StringClass&, int );
        friend char* operator>>(const StringClass&, int );
        friend char* operator+(const StringClass&, const
StringClass& );
        int operator>(const StringClass&);
```

```
        int operator<(const StringClass& String);
        void print();
};


//////////////////////////////////////////////////////////////////////
//
//                          Loader.C
//
//////////////////////////////////////////////////////////////////////
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include<ctype.h>
#include"Loader.h"

//////////////////////////////////////////////////////////////////////
//     Member Function : LoadSourceProgram
//     Purpose         : This function is used to load the program to be sliced
//                       into cppslicer. It scans each statement of the program
//                       and passes it to the InsertIntoList function.
//////////////////////////////////////////////////////////////////////

int
LoaderClass::LoadSourceprogram(char *Name)
{
       int linenumber;
       char buffer[MAXLINE];
       FILE *fp;
       //printf("Loading the Source Program into Slices\n");
       fp = fopen(Name, "r");
       if(fp == NULL)
       {
         printf("unable to open the file\n");
         return FALSE;
       }
       linenumber = 0;
       Header = NULL;
       do {
         memset(buffer, 0, MAXLINE);
         fgets(buffer, MAXLINE, fp);
         linenumber++;
         Buffer = buffer;
         LineNumber = linenumber;
         InsertIntoList();
       } while(!(feof(fp)));
       return TRUE;
}

//////////////////////////////////////////////////////////////////////
//     Member Function : InsertIntoList
//     Purpose         : The purpose of this function is to insert each line of
//                       the sample program into linked list.
//////////////////////////////////////////////////////////////////////

void
LoaderClass::InsertIntoList()
{
       node *sub;
       node *temp = new node;
       temp -> LineNumber = LineNumber;
       temp -> LineInformation = Buffer;
       temp -> left = NULL;
       temp -> right = NULL;
       temp -> DefHeader = NULL;
       temp -> RefHeader = NULL;
       temp -> INCLUDE = 0;
       temp -> metoo = TRUE;
       if(Header == NULL)
```

```
        {
          Header = temp;
          Header -> left = NULL;
          SliceClass = NULL;
          Rfdf = NULL;
        }
        else if( Header -> left == NULL)
        {
          Header -> left = temp;
          temp -> right = Header;
          temp -> left = NULL;
        }
        else
        {
          sub = Header;
          while(sub -> left != NULL)
          {
               sub = sub -> left;
          }
          sub -> left = temp;
          temp -> right = sub;
          temp -> left = NULL;
        }
}

////////////////////////////////////////////////////////////////////////
//      Member Function : InsertPublicPartsIntoSliceClass
//      Purpose         : Once the program is inserted into linked list,
//                        the list is searched for the class of interest.
//                        Then the public and private parts of the
//                        class of interest are inserted into refclass
////////////////////////////////////////////////////////////////////////
void
LoaderClass::InsertPublicPartsIntoSliceClass(char *info)
{
        char *temp;
        temp = strtok(info, ";");
        PlaceIntoSliceClass(PUBLIC, temp);
}

// Name represents class of interest. This function is used to get the line
// number of class of interest.

int
LoaderClass::GetClassLineNumber(const StringClass& Name)
{
        int len;
        node *temp;
        char *name, *tem;

        name = Name.GetStr();
        len = strlen(name);
        if(Header == NULL)
        {
          printf("file is not loaded into the slicer\n");
          printf("load and slice the program\n");
          //exit(0);
          return FAILURE;
        }
          temp = Header;
          while((temp != NULL)&&
          (((tem = strstr(temp -> LineInformation.GetStr(), name)) == NULL)||
          ((tem = strstr(temp -> LineInformation.GetStr(), "class")) == NULL)))
                      temp = temp -> left;
          if(tem != NULL)
          {
                      Position = temp -> LineNumber;
                      ClassName = name;
                      //strcpy(ClassName, name);
```

```
                return Position;
        }
        else
        {
             //printf("Searched %s\n", temp -> LineInformation.GetStr());
             //getchar();
             printf("not a valid class name: try again\n");
             tem = NULL;
             return FAILURE;
             //exit(0);
        }
}

////////////////////////////////////////////////////////////////////////
//     Member Function :GetPublicAndPrivateParts
//     Purpose         :This function is used to get the public and
//                      private parts of class of interest. The information is
//                      stored in refclass. It also stores the start and
//                      end line of each member function defined in the class of
//                      interest. This information is very useful while slicing
//                      member functions defined in public and private parts of
//                      the class of interest
////////////////////////////////////////////////////////////////////////


void
LoaderClass::GetPublicAndPrivateParts(void )
{
       node *temp;
       char *tem;
       char subinfo[MAXLINE];
       char *pinfo;
       char *poin;
       char info[MAXLINE];
       char tem1[80];
       int position, len, type;
       int count = 0;
       position = Position;
       temp = Header;
       while(temp -> LineNumber != position)
         temp = temp -> left;
       while((tem = strstr(temp -> LineInformation.GetStr(), "private:")) ==
NULL)
         temp = temp -> left;
       temp = temp -> left;
       while((tem = strstr(temp -> LineInformation.GetStr(), "public:")) ==
NULL)
       {
         memset(subinfo, 0, MAXLINE);
         memset(info, 0, MAXLINE);
         memset(tem1, 0, MAXLINE);
         strcpy(info, temp -> LineInformation.GetStr());
         type = GetVariableType(info);
         pinfo = RemoveFrontSpaces(info);
         strcpy(info, pinfo);
         sscanf(info, "%[^  ]%[^;]", tem1, subinfo);
         len = strlen(subinfo);
         subinfo[len] = ';';
         switch(type)
         {
         case INT: // For Int
                   //printf("Entered into Int Statement\n");
                   LoadIntoSliceClass(PRIVATE,INT,subinfo);
                   break;
         case CHAR: // For Char
                   LoadIntoSliceClass(PRIVATE,CHAR,subinfo);
                   break;
         case FLOAT: // For Float
                   LoadIntoSliceClass(PRIVATE,FLOAT,subinfo);
```

```
                break;
        case ERROR:
                printf("error: invalid type encountered\n");
                exit(0);
        };
        temp = temp -> left;
        type = 0;
      }
      tem = CheckPresence(temp -> LineInformation, "public");
      if(temp == NULL)
      {
        printf("error in the program loaded: no public parts\n");
        exit(0);
      }
      else
      {
        //printf("Into Public PArts\n");
        // At Presnt I am checking classes with no inline Functions.
        count = 0;
        temp = temp -> left;
        strcpy(info, temp -> LineInformation.GetStr());
        while(((tem = CheckPresence(info, "}")) == NULL) || (count != 0))
        {
             if((poin = strstr(temp -> LineInformation.GetStr(), "{")) != NULL)
                count++;
             else if((poin = strstr(temp -> LineInformation.GetStr(), "}")) !=
NULL)
                count--;
             tem = CheckPresence(temp -> LineInformation, "//");
             if( tem != NULL)
                strcpy(info, strtok(info , "//"));
             tem = strstr(temp -> LineInformation.GetStr(), "/*");
             if( tem != NULL)
                strcpy(info, (temp -> LineInformation && "/*"));
             pinfo= RemoveTabsAndBraces(info);
             if( strlen(pinfo) > 0)
             {
                strcpy(info, pinfo);
                pinfo = NULL;
                LoadIntoSliceClass(PUBLIC,info);
             }
             temp = temp -> left;
             memset(info, 0, MAXLINE);
             memset(subinfo, 0, MAXLINE);
             memset(info, 0, MAXLINE);
             pinfo = NULL;
             strcpy(info, temp -> LineInformation.GetStr());
        }
             LoadMainParts();
      }
}

////////////////////////////////////////////////////////////////////////
//     Member Function :GetVariableType
//     Purpose         :This function is used to find out the type of
//                      variable that is returned from the memeber
//                      function.
////////////////////////////////////////////////////////////////////////

int
LoaderClass::GetVariableType(char *info)
{
      char *temp;
      if((temp = strstr(info, "int")) != NULL)
        return INT;
      else if((temp = strstr(info, "char")) != NULL)
        return CHAR;
      else if((temp = strstr(info, "float")) != NULL)
        return FLOAT;
```

```
                                        temp = temp +1 ;
                }
          }
        }
}

////////////////////////////////////////////////////////////////////////
//      Member Function :GetHeader
//      Purpose         :This function is used to get the starting
//                       statement of the program residing in the cppslicer.
////////////////////////////////////////////////////////////////////////

node*
LoaderClass::GetHeader(void)
{
        if(Header == NULL)
          return NULL;
        else
          return Header;
}

////////////////////////////////////////////////////////////////////////
//      Member Function : GetRefClass
//      Purpose         : To get the starting member present in the
//                        refclass list.
////////////////////////////////////////////////////////////////////////

RefClass*
LoaderClass::GetRefClass(void)
{
        if(SliceClass == NULL)
          return NULL;
        else
          return SliceClass;
}

// This function is used to get the name of class of interest.

char*
LoaderClass::GetClassName()
{
        return ClassName.GetStr();
}

int
LoaderClass::GetPosition(void)
{
        return Position;
}

// This function is used to print the program resident in cppslicer program onto
// the screen.

void
LoaderClass::PrintProgram()
{
        int number;
        int num = 0;
        node *temp;
        temp = Header;
        number = 0;
        while( temp -> left != NULL)
          temp = temp -> left;
        number = temp -> LineNumber;
        temp = Header;
        num = 1;
        printf("-----------------------------------------------\n");
        printf("page %d of %d\n", num, ((number/20) + 1));
        while(temp != NULL)
```

```
        {
          printf("%d  %s", temp -> LineNumber, temp -> LineInformation.GetStr());
          temp = temp -> left;
          if((temp -> LineNumber % 20 ) == 0)
          {
               printf("-----------------------------------------------\n");
               printf("enter a key to continue ...");
               getchar();
               num++;
               printf("-----------------------------------------------\n");
               printf("page %d of %d\n", num, ((number/20) + 1));
          }
        }
        printf("\n-----------------------------------------------\n");
}

////////////////////////////////////////////////////////////////////////
//      Member Function :PrintSliceClass
//      Purpose         :Used to print the contents of the refclass.
////////////////////////////////////////////////////////////////////////

void
LoaderClass::PrintSliceClass(void)
{
        RefClass *temp;
        temp = SliceClass;
        while( temp != NULL)
        {
          printf("%d  ", temp -> PartsType);
          if( temp -> PartsType == PUBLIC)
               printf("%d  ", temp -> rettype);
          printf("%s  ", temp -> ClassName.GetStr());
          printf("%s\n", temp -> Member.GetStr());
          temp = temp -> next;
        }
}

////////////////////////////////////////////////////////////////////////
//      Member Function : EndLoader
//      Purpose         : Psedo destructor to end the loader. This function is
//                        called when the cppslicer has to be loaded with another
//                        program (that has to be sliced.)
////////////////////////////////////////////////////////////////////////

void
LoaderClass::EndLoader()
{
        printf("called \n");
        if( Header != NULL)
        {
          while( Header -> left != NULL)
          {
               node* temp = Header;
               Header = Header -> left;
               Header -> right = NULL;
               temp -> left = NULL;
               temp -> right = NULL;
               delete temp;
          }
        }
        if( SliceClass != NULL)
        {
          while( SliceClass != NULL)
          {
               RefClass *temslice = SliceClass;
               SliceClass = SliceClass -> next;
               temslice -> next = NULL;
               delete temslice;
          }
```

```
        }
}

////////////////////////////////////////////////////////////////////////
//      Member Function :RemoveTabsAndBraces
//      Purpose         : While parsing the member function definition,
//                        the spaces before and after the statement are
//                        removed. This function is called to remove spaces,
//                        tabs, and braces (if any) before and after the member
//                        function that is to be stored into refclass
////////////////////////////////////////////////////////////////////////

char*
LoaderClass::RemoveTabsAndBraces(char *info)
{
        int len = strlen(info);
        int count = 0;
        int counter = 0;
        StringClass temp;
        temp = RemoveFrontSpaces(info);
        temp = temp && "(";
        temp = RemoveBackSpaces(temp.GetStr());
        RemoveReturnTypes(temp);
        temp = RemoveFrontSpaces(temp.GetStr());
        return (temp.GetStr());
}

////////////////////////////////////////////////////////////////////////
//      Member Function :RemoveFrontSpaces
//      Purpose         :This function is used to remove the spaces and
//                       tabs infront of the statement. It is called from
//                       RemoveTabsAndBraces function.
////////////////////////////////////////////////////////////////////////

char *
LoaderClass::RemoveFrontSpaces(char *info)
{
        StringClass temp;
        int counter = 0;
        int count = 0;
        while(!(isalnum(info[counter])))
        {
          if((info[counter] == TAB) || (info[counter] == SPACE))
          {
                counter++;
                count++;
          }
          else
                break;
        }
        StringClass str = info;
        if( count > 0)
          temp = str << count;
        else
          temp = str.GetStr();
        return (temp.GetStr());
}

////////////////////////////////////////////////////////////////////////
//      Member Function :RemoveBackSpaces
//      Purpose         :This function is used to remove the spaces and
//                       tabs after the statement. It is called from
//                       RemoveTabsAndBraces function
////////////////////////////////////////////////////////////////////////

char *
LoaderClass::RemoveBackSpaces(char *info)
{
        int len = strlen(info);
```

```
        int counter = len;
        int count = 0;
        while((!isalnum(info[counter])))
        {
          if((info[counter] == TAB) || (info[counter] == SPACE))
          {
               counter--;
               count++;
          }
          else
               break;
        }
        StringClass str = info;
        if( count > 0)
          str = str >> count;
        return (str.GetStr());
}

////////////////////////////////////////////////////////////////////////
//      Member Function : RemoveReturnTypes
//      Purpose         : Once the spaces in front and back of the statement to
//                        be inserted into refclass are removed, the prototype
//                        containing the return type is also removed. After this
//                        the statement contains only the member function name
//                        (prototype). This is used in locating the function's
//                        body
////////////////////////////////////////////////////////////////////////

void
LoaderClass::RemoveReturnTypes(StringClass& info)
{
        returntype = INTIGER;
        int len = 0;
        char* pinfo = info.GetStr();
        if((info &= "friend") != NULL)
        {
          info = info << 6;
          returntype = FRIEND;
        }
        else if((info &= "int*") != NULL)
        {
          info = info << 4;
          returntype = INTS;
          return;
        }
        else if((info &= "char*")!= NULL)
        {
          info = info << 5;
          returntype = CHARS;
        }
        else if((info &= "float*")!= NULL)
        {
          info = info << 6;
          returntype = FLOATS;
        }
        else if((info &= "void")!= NULL)
        {
          info = info << 4;
          returntype = VOID;
        }
        else if(((info &= "char") != NULL))
        {
          info = info << 4;
          returntype = CHARECTER;
          return;
        }
        else if(( info &= "int") != NULL)
        {
          info = info << 3;
```

```
        returntype = INTIGER;
        return;
      }
      else if((info &= "float") != NULL)
      {
        info = info << 5;
        returntype = FLOATING;
        return;
      }
      else if((info &= ClassName.GetStr()) != NULL)
      {
        len = 0;
        len = strlen(ClassName.GetStr());
        if((info &= "& ") != NULL)
             info = info << len + 1;
        else
             info = info << len;

        returntype = CLASSTYPE;
        return;
      }
      else
      {
        int counter = 0;
        int count = 0;
        count = strlen(info.GetStr());
        len = 0;
        while(len < count)
        {
             if((pinfo[counter] != TAB) || (pinfo[counter] != SPACE))
             {
                    counter++;
                    len++;
             }
             else if((isalnum(pinfo[counter])))
                    break;
        }
        if( counter > 0)
             info = info << counter;
        else
             info = pinfo;
      }
}

// A copy of loaderclass can be made by making use of operator overloading. This
// function is of great use when a copy of loaderclass instinct is to me made
// which can be modified and those changes will not reflect in the original
// member

LoaderClass&
LoaderClass::operator=(LoaderClass& L)
{
      Header = L.GetHeader();
      SliceClass = L.SliceClass;
      Buffer = L.Buffer;
      ClassName = L.ClassName;
      LineNumber = L.LineNumber;
      Size  = L.Size;
      return (*this);
}

//////////////////////////////////////////////////////////////////////
//     Member Function : LoadMainParts
//     Purpose         : It is used to load the main program and its
//                       functions.
//////////////////////////////////////////////////////////////////////

void
LoaderClass::LoadMainParts()
```

```
{
        node *ptr = Header;
        RefClass *sub;
        char *tem;

        while((strstr(ptr -> LineInformation.GetStr(), "main()") == NULL)&&(ptr
!= NULL))
                ptr = ptr -> left;
        if(ptr == NULL)
        {
          printf("warning: no main program\n");
          return;
        }
          //printf("main linenumber is %d\n", ptr -> LineNumber);
          char name[80];
          memset(name, 80, NULL);
          strcpy(name, "main()");
          //LoadIntoSliceClass(PUBLIC,name);
          PlaceIntoSliceClass(PUBLIC, name);
          sub = SliceClass;
          while(sub -> Member != "main()")
                sub = sub -> next;
          sub -> BeginLine = ptr -> LineNumber;
          //printf("LineInformation is %s\n", ptr -> LineInformation.GetStr());
          int linenumber = 0;
          int i = -1;
          while((ptr != NULL) || (linenumber != 0))
          {
                tem = strstr(ptr -> LineInformation.GetStr(), "{");
                if( tem != NULL)
                        i++;
                tem = strstr(ptr -> LineInformation.GetStr(), "}");
                if(( tem != NULL) && ( i== 0))
                {
                        linenumber = ptr -> LineNumber;
                        //printf("Reached End Of Line: %d\n", linenumber);
                        break;
                }
                else if(( tem != NULL) && ( i > 0))
                        i--;
                ptr = ptr -> left;
          }

          if(ptr == NULL)
          {
                printf("warning: main part of the program is missing\n");
                printf("Check Source Code\n");
                return;
          }
          if(linenumber != 0)
                sub -> EndLine = linenumber;
        //printf("begin and end lines are %d and %d\n", sub -> BeginLine, sub ->
EndLine);
}

/////////////////////////// End of Loader.C ///////////////////////


//////////////////////////////////////////////////////////////////////////
//
//                                      Slice.C
//
//////////////////////////////////////////////////////////////////////////
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include<ctype.h>
#include"slice.h"
```

```
////////////////////////////////////////////////////////////////////////
//     Member Function : ComputeRefDefForClasses
//     Purpose         : This function performs all the necessary
//                       operations for slicing the program. It first
//                       parses each member function and then computes
//                       the ref and def by calling different member
//                       functions. It then calls a member
//                       function to slice the program based on the
//                       class of interest, criterion variable, and its line
//                       number.
////////////////////////////////////////////////////////////////////////

int
SliceClass::ComputeRefDefForClasses(const  StringClass& name)
{
       node *temp, *nodeclass;
       StringClass string ;
       char *tem;
       int endNumber;
       int beginnumber;
       int status = 0;
       temp = L1.GetHeader();
       RefClass *refclass;
       RefClass *temrefclass;
       refclass = L1.GetRefClass();
       RefHeader = NULL;
       DefHeader = NULL;
       IoRefHeader = NULL;
       if(refclass == NULL)
       {
         printf("there is no class \n");
         exit(0);
       }
       else
       {
         //printf("RefClass Is %s\n", refclass -> ClassName.GetStr());
         if(refclass -> ClassName == name)
         {
              temrefclass = refclass;
              while(temrefclass != NULL)
              {
                    if((temrefclass -> PartsType == PUBLIC)&&(temrefclass
->Member != "main()"))
                    {
                          if(temrefclass -> rettype != FRIEND)
                          {
                                string = temrefclass -> ClassName;
                                string = string + "::";
                                string = string + temrefclass -> Member;
                                if( temp != NULL)
                                      nodeclass = temp;
                                else
                                {
                                      printf("error in getting the header\n");
                                      exit(0);
                                }
                                endNumber = 0;
                                while((nodeclass != NULL)&&( endNumber == 0))
                                {
                                      tem =
strstr(nodeclass->LineInformation.GetStr(),string.GetStr());
                                      if(tem == NULL)
                                            nodeclass = nodeclass -> left;
                                      else
                                      {
         beginnumber = nodeclass -> LineNumber;
         endNumber = GetEndLineForClassMemeber(string, beginnumber);
         temrefclass -> BeginLine = beginnumber;
         temrefclass -> EndLine = endNumber;
```

```
        MarkAllSlices(beginnumber, endNumber);
        status = ComputeRefDef(beginnumber, endNumber);
        if( status == -1)
        {
             printf("cannot handle reference to pointers\n");
             return -1;
        }

MarkControlStatForRefDef(beginnumber, endNumber);
             break;
                                        }
                                  }
                            }
                      }
                      else if(temrefclass -> Member == "main()")
                      {
                            beginnumber = temrefclass -> BeginLine;
                            endNumber = temrefclass -> EndLine;
                            MarkAllSlices(beginnumber, endNumber);
                            status = ComputeRefDef(beginnumber, endNumber);
                            if( status == -1)
                            {
                                  printf("cannot handle reference to
pointers\n");
                                  return -1;
                            }
                            MarkControlStatForRefDef(beginnumber, endNumber);
                      }
                      temrefclass = temrefclass -> next;
             }
        }
        else
        {
             printf("error in choosing the slice class\n");
             printf("error in ComputeRefDefForSlcieClass\n");
             exit(0);
        }
        RefHeader = NULL;
        DefHeader = NULL;
        IoRefHeader = NULL;
      }

      // Here U have to call MarkAllSlices ComputeRefDEf,
SetReservedVarIntoRef,
      // ControlFlowCheck.
}

//////////////////////////////////////////////////////////////////////
//     Member Function : MarkAllSlices
//     Purpose         : The purpose of this function is to mark all the
//                       statement's slice marks to false. After setting slice
//                       marks to false, the program is sliced and the slice
//                       marks for statements of interest are set to true. Then
//                       it will be very easy to display the statements of
//                       interest onto the screen.
//////////////////////////////////////////////////////////////////////

void
SliceClass::MarkAllSlices(int start, int end)
{
      node *temp;
      temp = L1.GetHeader();
      while((temp != NULL)&&(temp -> LineNumber != start))
        temp = temp -> left;
      if( temp == NULL)
      {
        printf("warning: errror in setting initial slice marks \n");
        return;
      }
```

```
        while(temp -> LineNumber != end)
        {
          temp -> SLICE = FALSE;
          temp -> metoo = TRUE;
          temp = temp -> left;
        }
}

// This function is called to set ref, def for the variables defined in each
// statement of each member function of class of interest. The basic algorithm
// for parsing is adopted from Nanja's work. This function after computing the
// ref and def for each line inserts them into the linked list containing the
// statement itself.

int
SliceClass::ComputeRefDef(int begin, int end)
{
        int i =0;
        int index = 0;
        int count;
        int state;
        int NoHeader;
        node *temhead;
        StringClass pname;
        RefDef *calptr;
        char Information[80];
        char *buffer;
        char variable[MAXLINE];
        char secvariable[MAXLINE];
        char *test;
        char thirdvariable[MAXLINE];
        int seccount;
        int CommentFlag;
        StateType presentstate,prevstate;
        //printf("Entered Into Compute Ref Def Function\n");
        temhead = L1.GetHeader();
        while(( temhead -> LineNumber != begin) && ( temhead != NULL))
          temhead = temhead -> left;
        if(temhead == NULL)
        {
          printf("error in getting the line of the member
function\n");
          exit(0);
        }
        temhead = temhead -> left;
        while( temhead -> LineNumber != end)
        {
          presentstate = StartState;
          prevstate = StartState;
          CommentFlag = 0;
          RefCounter = FALSE;
          DefCounter = FALSE;
          IoCounter = FALSE;

          strcpy(Information ,temhead -> LineInformation.GetStr());
          buffer = Information;
          //buffer = temhead -> LineInformation.GetStr();
          CheckForComments(buffer, CommentFlag);
          if( strstr(buffer, "//") != NULL)
          {
               buffer = strtok(buffer, "//");
               //printf("after strtok: %s\n", buffer);
          }
          if(strstr(buffer, "/*") != NULL)
          {
               if( strstr(buffer, "*/") != NULL)
               {
                      buffer = strtok(buffer, "/*");
                      //printf("State Reached :%s\n", buffer);
```

```
                        break;
                }
                else
                {
                        while((strstr( temhead ->
LineInformation.GetStr(), "*/") == NULL))
                                temhead = temhead -> left;
                        buffer = temhead -> LineInformation.GetStr();
                }
        }
        NoHeader = FALSE;
        while((presentstate != LastState) && ( CommentFlag ==
      NOTPRESENT))
        {
                // This Case Check For the valid Command Name. It then
        // decides the the value of the PresentState till it
      // gets to final state.
                switch(presentstate)
                {
                        case StartState:
                                i = 0;
                        while((buffer[i] == SPACE ) || (buffer[i] ==
                TAB) || (buffer[i] == OPENBRACE))
                                        i++;
                        if((buffer[i] == ENDOFLINE) || (buffer[i] ==
                CARRRETURN)|| (buffer[i] == CLOSEBRACE))
                                {
                                        presentstate = LastState;
                                        NoHeader = TRUE;
                                        break;
                                }
                                prevstate = presentstate;
                                state = NOTPRESENT;
                                state = CheckForControlStatement(buffer);
                                if( state == PRESENT)
                                {
                                        presentstate = LastState;
                                        break;
                                }
                                state = NOTPRESENT;
                                state = CheckForFor(buffer);
                                if( state == PRESENT)
                                {
                                        presentstate = ControlState;
                                        break;
                                }

                                state = NOTPRESENT;
                                state = CheckForIOStream(buffer);
                                if( state == PRESENT)
                                {
                                        presentstate = SixthState;
                                        break;
                                }
                        else if(( strstr(buffer, "str") != NULL))
                                {
                        // This State has been Determined in the comming Code.
                                        presentstate = StrState;
                                        break;
                                }
                              else if(( strstr(buffer, "switch") != NULL))
                                {
                                        presentstate = SwitchState;
                                        break;
                                }
                                else if((isalnum(buffer[i]))|| (buffer[i]
                                == UNDERSCORE))
                                {
                                        presentstate = VariableState;
```

```
                        break;
                }
                else if((buffer[i] == PLUS) || (buffer[i]
                == SUBTRACT))
                {
                        presentstate = AirthematicState;
                        break;
                }
                else if((buffer[i]== SUBTRACT) &&
        (buffer[i+1] == '>'))
                {
                        printf("line %d: reference to
        pointer encountered\n", temhead ->
        LineNumber);
                        return -1;
                }
                else
                {
                        presentstate = LastState;
                        NoHeader = TRUE;
                        break;
                }
// Checks The Variable Presnnt in the line and stores in the ref list
        case VariableState:
                        index = 0;
                        memset(variable, NULL, MAXLINE);
                        while((buffer[i] == SPACE) || (buffer[i]
        == OPENBRACKET)|| (buffer[i] ==
CLOSEBRACKET)|| (buffer[i] == TAB) ||
                                (buffer[i] == COMMA))
                                i++;
                        while((isalnum(buffer[i])) || (buffer[i]
        == UNDERSCORE))
                                variable[index++] = buffer[i++];
                        variable[index] = ENDOFLINE;
                        IgnoreSpaces(buffer, i);
                        if(buffer[i] == PERIOD)
                                i++;
                        if(buffer[i] == SQOPENBRACKET)
                        {
                                i++;
                                IgnoreSpaces(buffer, i);
                                index = 0;
                                while(buffer[i] != SQCLOSEBRACKET)
                                {
                                        IgnoreSpaces(buffer, i);
                                        if((isalpha(buffer[i])) ||
                                                (buffer[i] ==
                                        UNDERSCORE))
                                        {
                                        while((isalpha(buffer[i]))
                                        ||(buffer[i] == UNDERSCORE))
                                        secvariable[index++] =
                        buffer[i++];
                                        secvariable[index] =
                                ENDOFLINE;
                                        InsertIntoRefDefList(REFERED,
                                        temhead -> LineNumber,
                        secvariable);
                                        }
                                        else
                                IgnoreDigitsOperators(buffer, i);
                                }
                        }
                        if( strcmp(variable ,"if")  == NULL)
                        {
                                presentstate = LastState;
                                break;
                        }
```

```
                        if((prevstate == AirthematicState))
                        {
                  if((buffer[i] == PLUS) && (buffer[i] ==
  buffer[i+1])|| (buffer[i] == SUBTRACT) &&
                  (buffer[i] == buffer[i+1]))
                              {
                                    InsertIntoRefDefList(REFERED,
                              temhead -> LineNumber, variable);
                                    InsertIntoRefDefList(DEFERED,
                              temhead -> LineNumber, variable);
                                    presentstate = LastState;
                                    break;
                              }
                        }
                        InsertIntoRefDefList(DEFERED, temhead ->
                  LineNumber, variable);
                        IgnoreBraces(buffer, i);
                  if(((buffer[i] == PLUS) && (buffer[i + 1] ==
            buffer[i]))|| ((buffer[i] == SUBTRACT) &&
  (buffer[i+1] == buffer[i])))
                        {
                        InsertIntoRefDefList(REFERED, temhead ->
                        LineNumber, variable);
                              presentstate = LastState;
                              break;
                        }
                        prevstate = presentstate;
                        if(buffer[i] == COLON)
                        {
                              presentstate = LastState;
                              break;
                        }
                        else if( buffer[i] == EQUALTO)
                        {
                              presentstate = IntermediateState;
                              break;
                        }
                  else if((buffer[i]== SUBTRACT) && (buffer[i+1]
                  == '>'))
                        {
                  printf("line %d: reference to pointer
  encountered\n", temhead -> LineNumber);
                              return -1;
                        }
                  else if((buffer[i] == PLUS) || (buffer[i] ==
            SUBTRACT)|| (buffer[i] == MULTIPLY) ||
(buffer[i] == DIVIDE))
                        {
                              i++;
                              if( buffer[i] == EQUALTO)
                                    presentstate = LoopingState;
                              break;
                        }
                        break;

  // This case is basically to determine the state of buffer

                  case IntermediateState:                  // Case 2
                        RefDef *front, *back;
                        if( prevstate == LoopingState)
                        {
                              if( RefHeader != NULL)
                              {
                                    front = RefHeader;
                                    back = front;
                                    while(front != NULL)
                                    {
                                          back = front;
                                          front = front -> next;
```

```
                                        }
                                        char *x = back -> variable.GetStr();
                                                InsertIntoRefDefList(REFERED,
                                                        back -> linenumber, x );
                                        }
                                }
                                IgnoreBraces(buffer, i);
                                IgnoreOperands(buffer, i);
                                IgnoreSpaces(buffer, i);
                                if(buffer[i] == COTE)
                                        i++;
                                if( buffer[i] == EQUALTO)
                                        i++;
                                IgnoreBraces(buffer, i, OPENBRACKET);
                                prevstate = presentstate;
                                if((buffer[i] == CARRRETURN) || (
                        buffer[i] == PERIOD)||
                                        (buffer[i] == COLON))
                                {
                                        presentstate = LastState;
                                        break;
                                }
                                else if((isdigit(buffer[i])) || (buffer[i]
                                == PERIOD))
                                {
                                        presentstate = PreFinalState;
                                        break;
                                }
                                else if((isalpha(buffer[i]))||(buffer[i]
                                == UNDERSCORE))
                                        presentstate = CheckingState;
// Next state
                                else if(buffer[i] == OPENBRACE)
                                        presentstate = LastState;
                                break;

                        case CheckingState:
// Case Number 3
                                index = 0;
                                memset(variable, NULL, MAXLINE);
                              while((isalpha(buffer[i])) || (buffer[i] ==
                      UNDERSCORE))variable[index++] =
buffer[i++];
                                variable[index] = ENDOFLINE;
                                IgnoreSpaces(buffer, i);
                                if(buffer[i] == SQOPENBRACKET)
                                {
                                        while(buffer[i] != SQCLOSEBRACKET)
                                        {
                                                i++;
                                                IgnoreSpaces(buffer, i);
                                                index = 0;
                                                if((isalpha(buffer[i])) ||
                                        (buffer[i]==UNDERSCORE))
                                                {
                                        while((isalpha(buffer[i]))||
                                                (buffer[i]==UNDERSCORE))
                                                        secvariable[index++] =
                                                        buffer[i++];
                                                        secvariable[index] =
                                                        ENDOFLINE;
                                                InsertIntoRefDefList(REFERED,
                                                        temhead -> LineNumber,
                                                secvariable);
                                                }
                                                else
                                        IgnoreDigitsOperators(buffer, i);
                                                }
                                }
```

```
                IgnoreBraces(buffer, i);
                prevstate = presentstate;
                if(buffer[i] == EQUALTO)
                {
                InsertIntoRefDefList(DEFERED, temhead ->
                LineNumber, variable);
                        presentstate = IntermediateState;
                        break;
                }
                InsertIntoRefDefList(REFERED, temhead ->
        LineNumber, variable);
                if((buffer[i] == COLON) || (buffer[i] ==
        OPENBRACKET))
                        presentstate = LastState;
        else if((buffer[i] == SUBTRACT) && (buffer[i+1]
        == '>'))
                {
        printf("line %d: reference to pointer
encountered\n", temhead -> LineNumber);
                        return -1;
                }
                else
                        presentstate = IntermediateState;
                break;

        case LoopingState:          // Case Number 4
                //printf("Entered Into Looping State\n");
                prevstate = presentstate;
                if( buffer[i] == EQUALTO)
                        presentstate = IntermediateState;
                break;

        case AirthematicState:      // Case Number 5
        //printf("Entered Into Airthematic State\n");
                ++i;
                prevstate = presentstate;
                if((buffer[i] == PLUS) || (buffer[i] ==
        SUBTRACT))
                presentstate = VariableState;
                break;

        case SixthState:                    // Case Number 6
                //printf("Entered Into Sixth State\n");
                memset(variable, 0, MAXLINE);
                i = 0;
                index =0;
                IgnoreSpaces(buffer, i);
                IOTYPE type;
                type = GetIoType(buffer);
                switch(type)
                {
                        case GETCHAR:
                        while((isalpha(buffer[i]))||
                                (buffer[i] == UNDERSCORE))
                                        variable[index++] =
                                buffer[i++];
                                variable[index] = ENDOFLINE;
                                InsertIntoRefDefList(DEFERED,
                                        temhead -> LineNumber,
                                variable);
                                presentstate = LastState;
                                break;
                        case GETS:
                                while((isalpha(buffer[i])))
                                        i++;
                                IgnoreSpaces(buffer, i);
                                if( buffer[i] == OPENBRACKET)
                                        i++;
                                while((isalpha(buffer[i])) ||
```

```
                        (buffer[i] == UNDERSCORE))
                                variable[index++] =
                        buffer[i++];
                        variable[index] = ENDOFLINE;
                        InsertIntoRefDefList(DEFERED,
                                temhead -> LineNumber,
                        variable);
                        presentstate = LastState;
                        break;
                case SCANF:
                case PRINTF:
                char *point = strchr(buffer, '"');
                        if( point != NULL)
                        {
                                ++point;
                        point = strchr(point, '"');
                                ++point;
                                strcpy(buffer, point);
                        }
                        index = 0;
                if( buffer[index] == CLOSEBRACKET)
                        {
                        presentstate = LastState;
                                break;
                        }
                        int length = strlen(buffer);
                        while(index < length)
                        {
                if(((!(isalpha(buffer[index])))))
                        {
                if((buffer[index] != UNDERSCORE))
                                buffer[index] = SPACE;
                        }
                        index++;
                        }
                        i = 0;
                        length = strlen(buffer);
                        while( i < length)
                        {
                                IgnoreSpaces(buffer, i);
                                index = 0;
                        while((isalnum(buffer[i])) ||
                        (buffer[i] == UNDERSCORE))
                                variable[index++] =
                        buffer[i++];
                        variable[index] = ENDOFLINE;
                        if((i < length) &&
(IsAValidDeclaration(variable)))
                                {
                                        if( type == SCANF)
                        InsertIntoRefDefList(DEFERED,
                temhead -> LineNumber, variable);
                                        else
                InsertIntoRefDefList(REFERED,
                temhead -> LineNumber, variable);
                                }
                        }
                        presentstate = LastState;
                        break;
// added  later when the code is working  fine
        case CIN:
                InsertIntoIoList(CIN, buffer);
                presentstate = LastState;
                break;
        case COUT:
                InsertIntoIoList(COUT, buffer);
                presentstate = LastState;
                break;
        case SWITCH:
```

```
                                InsertIntoIoList(SWITCH, buffer);
                                break;
                        case VOIDTYPE:
                                printf("cannot produce slice for the
                        folloing line\n");
                                printf("%s\n", buffer);
                                break;
                        };
                        break;

                case StrState       :             // Case Number 7
                        //printf("Entered Into Strstr State\n");
                        i = 0;
                        memset(variable, NULL, MAXLINE);
                        count = 0;
                        pname = buffer;
                        if( (pname &= "strlen") != NULL)
                        {
                                while((!(isalnum(buffer[i]))) &&
                                        (buffer[i] != UNDERSCORE))
                                        i++;
                                while((buffer[i] == UNDERSCORE) ||
                        (isalnum(buffer[i])))
                                variable[count++] = buffer[i++];
                                variable[count] = ENDOFLINE;
                                InsertIntoRefDefList(DEFERED,
                                temhead -> LineNumber, variable);
                                test = strstr(buffer, "strlen");
                                test = strchr(test, OPENBRACKET);
                                count = 0;
                                strcpy(secvariable, test);
                                IgnoreSpaces(secvariable, count);
                                if(secvariable[count] ==
        OPENBRACKET)
                                        count++;
                                seccount = 0;
                                while((secvariable[count] ==
                UNDERSCORE)
||((isalnum(secvariable[count]))))
                                        thirdvariable[seccount++] =
                                secvariable[count++];
                                thirdvariable[seccount] = ENDOFLINE;
                        InsertIntoRefDefList(REFERED, temhead
                ->LineNumber, thirdvariable);
                                presentstate = LastState;
                                break;
                        }
                        else if( (pname &= "strcpy") != NULL)
                        {
                                test = strstr(buffer, "strcpy");
                                test = strchr(test, OPENBRACKET);
                                count = 0;
                                memset(secvariable, NULL, MAXLINE);
                                strcpy(secvariable, test);
                                IgnoreSpaces(secvariable, count);
                        if(secvariable[count] == OPENBRACKET)
                                        count++;
                                seccount = 0;
                                IgnoreSpaces(secvariable, count);
                        while((secvariable[count] == UNDERSCORE)
                        ||((isalnum(secvariable[count]))))
                                        thirdvariable[seccount++] =
                                secvariable[count++];
                                thirdvariable[seccount] = ENDOFLINE;
                        InsertIntoRefDefList(DEFERED, temhead
                ->LineNumber, thirdvariable);
                                presentstate = LastState;
                                break;
                        }
```

```
                while((buffer[i] != UNDERSCORE) &&
        ((!(isalpha(buffer[i])))))
                        i++;
                IgnoreSpaces(buffer, i);
                memset(variable, NULL, MAXLINE);
                count = 0;
                while((buffer[i] == UNDERSCORE) ||
        ((isalpha(buffer[i]))))
                        variable[count++] = buffer[i++];
                variable[count] = ENDOFLINE;
                if( (pname &= "strcmp") != NULL)
                {
                InsertIntoRefDefList(REFERED, temhead ->
                LineNumber, variable);
                }
                else
                {
                InsertIntoRefDefList(DEFERED, temhead ->
                LineNumber,  variable);
                }
                if( (pname &= "strcat") != NULL)
                {
                InsertIntoRefDefList(REFERED, temhead ->
                LineNumber,  variable);
                }
                while((buffer[i] != UNDERSCORE) &&
                        ((!(isalpha(buffer[i])))))
                        i++;
                IgnoreSpaces(buffer, i);
                count = 0;
                while((buffer[i] == UNDERSCORE) ||
((isalpha(buffer[i]))))
                        variable[count++] = buffer[i++];
                variable[count] = ENDOFLINE;
                InsertIntoRefDefList(REFERED, temhead ->
        LineNumber, variable);
                presentstate = LastState;
                break;

        case ControlState   :             // Case Number 8
                //printf("Entered Into Control State\n");
                // This performs the insertion of all I/o
        // variables into special list.
                CheckForIoOPerands(buffer);
        InsertControlVar(buffer, temhead -> LineNumber);
                //InsertIntoRefList(buffer);
                presentstate = LastState;
                break;

        case PreFinalState:        // Case Number 9
                //printf("Entered Into Prefinal State\n");
        while((isdigit(buffer[i])) || (buffer[i] ==
PERIOD))
                        i++;
                IgnoreBraces(buffer, i, OPENBRACKET);
                prevstate = presentstate;
                if( buffer[i] == COLON)
                {
                        presentstate = LastState;
                        break;
                }
                else
                        presentstate = IntermediateState;
                break;

        case SwitchState:                 // Case Number 10
                //printf("Entered Into Switch State\n");
                InsertIntoIoList(SWITCH, buffer);
                presentstate = LastState;
```

```
                        break;
                case LastState:     // Will Not reach This State
                        //printf("Error: Reached LastState Which is to
Quit\n");
                        break;
                };
        }
        if(IoCounter == TRUE)
        {
                calptr = IoRefHeader;
                //PrintVar(calptr);
                while(calptr != NULL)
                {
                        DeleteRefFromRefList(calptr);
                        calptr = calptr -> next;
                }
                calptr = IoRefHeader;
                while(calptr != NULL)
                {
                        InsertIoRefIntoDefList(calptr);
                        calptr = calptr -> next;
                }
        }
        if( RefCounter == TRUE)
        {
                calptr = RefHeader;
                printf("ref: ");
                PrintVar(calptr);
                while( calptr != NULL)
                {
                        InsertIntoHeaderAllRef(temhead, calptr);
                        calptr = calptr -> next;
                }
        }
        if( DefCounter == TRUE)
        {
                calptr = DefHeader;
                printf("def: ");
                PrintVar(calptr);
                while( calptr != NULL)
                {
                        InsertIntoHeaderAllDef(temhead, calptr);
                        calptr = calptr -> next;
                }
        }
        if( RefCounter == TRUE)
        {
                DeleteRefDefIoRef(RefHeader);
                RefHeader = NULL;
        }
        if( DefCounter == TRUE)
        {
                DeleteRefDefIoRef(DefHeader);
                DefHeader = NULL;
        }
        if( IoCounter == TRUE)
        {
                DeleteRefDefIoRef(IoRefHeader);
                IoRefHeader = NULL;
        }
        temhead = temhead -> left;
      }
      return SUCCESS;
}

//////////////////////////////////////////////////////////////////////
//      Member Function : IgnoreBraces
//      Purpose         : This function is used while parsing the statements.
//                        when ever a space, tab brackets are encountered, they
```

```
//                        are discarded. This function is called from
//                        ComputeRefDef function.
////////////////////////////////////////////////////////////////////////

void
SliceClass::IgnoreBraces( char* buffer, int& i, const char tem)
{
       if( tem == OPENBRACKET)
       {
         while(( buffer[i] == SPACE) || ( buffer[i] == TAB)
              || (buffer[i] == OPENBRACKET)|| (buffer[i] ==
         CLOSEBRACKET))
              i++;
         return;
       }
       return;
       // If u get any other specific Conditions Add them here
}

// This function is used to ignore braces

void
SliceClass::IgnoreBraces(char *buffer, int& i)
{
       while((buffer[i] == SPACE) || (buffer[i] == TAB) ||
         (buffer[i] == OPENBRACKET) || (buffer[i] == SQOPENBRACKET)
         || (buffer[i] == CLOSEBRACKET) || (buffer[i] ==
SQCLOSEBRACKET))
              i++;
       return;
}

// This function is used to ignore spaces and tabs present in a statement of
// the program to be sliced. This function is called from ComputeRefDef
// function.

void
SliceClass::IgnoreSpaces(char *buffer, int& i)
{
       while((buffer[i] == SPACE) || (buffer[i] == TAB))
         i++;
       return;
}

// This function is used to ingore the operators. This function is called from
// ComputeRefDef function.

void
SliceClass::IgnoreDigitsOperators(char *pbuffer,int& i)
{
       char buffer[MAXLINE];
       strcpy(buffer, pbuffer);
       while((buffer[i] == PLUS) || (buffer[i] == SUBTRACT) ||
         (isdigit(buffer[i])) || (buffer[i] == DIVIDE))
              i++;
}

// This function is used to ignore operands. This function is called from
// ComputeRefDef function.

void
SliceClass::IgnoreOperands(char *buffer, int& i)
{
       while((buffer[i] == PLUS) || (buffer[i] == SUBTRACT)
         || (buffer[i] == MULTIPLY) || (buffer[i] == DIVIDE) || (buffer[i] ==
MOD))
              i++;
}
```

```
// This function is used to ignore arithematic operators present in statements
// of the program to be sliced

void
SliceClass::IgnoreOnlyArthOPerators(char *pbuffer, int& k)
{
       char spbuffer[MAXLINE];
       //printf("I is %d\n", k);
       memset(spbuffer, NULL, MAXLINE);
       strcpy(spbuffer, pbuffer);
       while((spbuffer[k] == PLUS) || (spbuffer[k] == SPACE) ||
         (spbuffer[k] == TAB) || (spbuffer[k] == SUBTRACT) ||
         (spbuffer[k] == MULTIPLY) || (spbuffer[k] == DIVIDE) ||
(isdigit(spbuffer[k])))
                    k++;
       //printf("I is %d\n", k);
}


//////////////////////////////////////////////////////////////////////////
//     Member Function : CheckForControlStatement
//     Purpose         : This function is used to check for the control
//                       statements such as while, else, if, etc.. if
//                       found, it returns TRUE else FALSE. This function is
//                       used while parsing the statements of the program to be
//                       sliced
//////////////////////////////////////////////////////////////////////////

int
SliceClass::CheckForControlStatement(StringClass temp)
{
       //StringClass temp = ptemp;
         if((( temp &= "else") != NULL) && ((temp &= "if") == NULL))
              return PRESENT;
         else if(((( temp &= "do") != NULL) && (strlen(temp.GetStr()) == 3)) ||
              ((temp &= "exit(") != NULL))
              return PRESENT;
         else if((temp &= "case") != NULL)
              return PRESENT;
         else
              return NOTPRESENT;
}

// Checks for the presence expressions containing if, while, for. If the
// statement contains if, for, while, then it returns PRESENT else NOTPRESENT.
// This function is called from ComputeRefDef function to identify weather a
// statement is control statement or not.

int
SliceClass::CheckForFor(StringClass buffer)
{
       //StringClass buffer = pbuffer;
       if((( buffer &= "if") != NULL) || (( buffer &= "while") != NULL)
       || ((buffer &= "for") != NULL))
              return PRESENT;
       else
         return NOTPRESENT;
}

//////////////////////////////////////////////////////////////////////////
//     Member Function : CheckForIOStreams
//     Purpose         : This function is to check for the iostream
//                       commands cin and cout. It also check for the
//                       regular stdio commands, scanf, gets and other
//                       input/output functions.
//////////////////////////////////////////////////////////////////////////

int
SliceClass::CheckForIOStream(const StringClass& pbuffer)
```

```
{
        StringClass buffer = pbuffer;
        if((( buffer &= "cout") != NULL) || ((buffer &= "scanf") != NULL) ||
        ((buffer &= "cin") != NULL) || ((buffer &= "printf") != NULL) ||
        ((buffer &= "gets") != NULL) || ((buffer &= "getchar") != NULL))
          return PRESENT;
        else
          return NOTPRESENT;

}

void
SliceClass::SetReserveVarIntoRef()
{
        // This Will Call CheckForReserveVar();
}

// This function is used to check for the comments within the statements.

int
SliceClass::CheckForComments(const StringClass& string, int& flag)
{
        char *info, *temp;
        info = string.GetStr();
        temp = strstr(info, "/*");
        if( temp == NULL)
        {
          flag = NOTPRESENT;
          return PRESENT;
        }
        else if( temp != NULL)
        {
          flag = PRESENT;
          return PRESENT;
        }
        temp = strstr(info, "//");
        if( temp != NULL)
        {
          flag = PRESENT;
          return PRESENT;
        }
        else
        {
          flag = NOTPRESENT;
          return PRESENT;
        }
}

int
SliceClass::CheckForComments(const StringClass& string)
{
        int i=0;
        char *info = string.GetStr();
        while(info[i] != NULL)
        {
          if((info[i] == '/') && (info[i+1] == '/'))
          {
               return PRESENT;
          }
          i++;
        }
        return NOTPRESENT;
}

int
SliceClass::CheckForBlankSpace(const StringClass& string)
{
        int i=0;
        int status=NOTPRESENT;
```

```
        char *info = string.GetStr();
        //int len = string.Getlen();
        int len = strlen(info);
        while( info[i] != len)
        {
          if( info[i] == ' ')
          {
               status = PRESENT;
               break;
          }
          else
               i++;
        }
        if((i == len) &&(status == NOTPRESENT))
          return NOTPRESENT;
        else
          return PRESENT;
}

////////////////////////////////////////////////////////////////////////
// Member Function : GetBeginLineForClassMember
// Purpose         : This function is used to find out the starting line of the
//                   class of interest. Once the starting line of class of
//                   interest is known, the public and private parts can be
//                   obtained which will be stored in the refclass.
////////////////////////////////////////////////////////////////////////

int
SliceClass::GetBeginLineForClassMemeber(const StringClass& name)
{
        node *temp = L1.GetHeader();
        StringClass string;
        char *tem;
        int linenumber = 0;
        while(temp != NULL)
        {
          string = temp -> LineInformation;
          tem = L1.CheckPresence(string.GetStr(), name.GetStr());
          if( tem == NULL)
               temp = temp -> left;
          else
          {
               linenumber = temp -> LineNumber;
               break;
          }

        }
        if( linenumber == 0)
        {
          printf("no memeber function defined for the memeber defined in
class\n");
          exit(0);
        }
        else
          return linenumber;
}

////////////////////////////////////////////////////////////////////////
// Member Function : GetEndLineForClassMember
// Purpose         : This function is called with member function name and begin
//                   line number of the member function. It checks the refclass
//                   for the member function, if found compares the begin number
//                   and returns the end line number.
////////////////////////////////////////////////////////////////////////

int
SliceClass::GetEndLineForClassMemeber(const StringClass& string,int beginnumber)
{
        int i=0;
```

```
        node *temp = L1.GetHeader();
        if( temp == NULL)
        {
          printf("error in getting header\n");
          exit(0);
        }
        char *tem;
        StringClass stri = string.GetStr();
        int linenumber;
        //printf("LineNumber is %d\n", beginnumber);
        while((temp -> LineNumber != beginnumber) && ( temp != NULL))
          temp = temp -> left;
        temp = temp -> left;
        if( temp == NULL)
        {
          printf("error in the program\n");
          printf("check source code of tool \n");
          exit(0);
        }
        else
        {
          linenumber = 0;
          i = -1;
          while((temp != NULL) || (linenumber != 0))
          {
                tem = L1.CheckPresence(temp -> LineInformation, "{");
                if( tem != NULL)
                        i++;
                tem = L1.CheckPresence(temp -> LineInformation, "}");
                if(( tem != NULL) && ( i== 0))
                {
                        linenumber = temp -> LineNumber;
                        //printf("Reached End Of Line: %d\n",
        linenumber);
                        break;
                }
                else if(( tem != NULL) && ( i > 0))
                        i--;
          temp = temp -> left;
          }

          if(temp == NULL)
          {
                printf("error in the code of tool or program\n");
                printf("check source code\n");
                exit(0);
          }
          if(linenumber != 0)
                return linenumber;
        }
}

////////////////////////////////////////////////////////////////////////
//      Member Function : InsertIntoRefDefList
//      Purpose         : If a variable is referenced or defined in the
//                        statement,then the variable is passed to this
//                        function from ComputeRefDef function. If the
//                        variable is already existing in the list, it is
//                        ignored else it is inserted into RefDef list.
////////////////////////////////////////////////////////////////////////

int
SliceClass::InsertIntoRefDefList(const int& ref, const int& linenumber, char
*info)
{
        if((IsAValidDeclaration(info)) && (IsAValidMember(info)))
        {
        RefDef *temrefdef;
        RefDef *temp = new RefDef;
```

```
temp -> variable = info;
temp -> PartType = ref;
temp -> linenumber = linenumber;
temp -> next = NULL;
if( ref == REFERED)
{
  RefCounter = TRUE;
  if( RefHeader == NULL)
  {
       RefHeader = temp;
       return SUCCESS;
  }
  else
  {
       temrefdef = RefHeader;
       while((temrefdef -> next != NULL) &&
              (temrefdef -> variable != temp -> variable))
              temrefdef = temrefdef -> next;
       if( temrefdef -> variable == temp -> variable)
              return FAILURE;
       else
       {
              temrefdef -> next = temp;
              return SUCCESS;
       }
  }
}
else if( ref == DEFERED)
{
  DefCounter = TRUE;
  if( DefHeader == NULL)
  {
       DefHeader = temp;
       return SUCCESS;
  }
  else
  {
       temrefdef = DefHeader;
       while((temrefdef -> next != NULL) &&
              (temrefdef -> variable != temp -> variable))
              temrefdef = temrefdef -> next;
       if( temrefdef -> variable == temp -> variable)
              return FAILURE;
       else
       {
              temrefdef -> next = temp;
                     return SUCCESS;
       }
  }
}
else if( ref == IOREF)
{
  IoCounter = TRUE;
  if( IoRefHeader == NULL)
  {
       IoRefHeader = temp;
       return SUCCESS;
  }
  else
  {
       temrefdef = IoRefHeader;
       while((temrefdef -> next != NULL) &&
              (temrefdef -> variable != temp -> variable))
              temrefdef = temrefdef -> next;
       if( temrefdef -> variable == temp -> variable)
              return FAILURE;
       else
       {
              temrefdef -> next = temp;
```

```
                        return SUCCESS;
                }
          }
        }
        }
}

// The name denotes the class of interest. The criterion line is checked for
// variable(s) of interest. If it contains them, then slice mark is set true

int
SliceClass::MarkSliceForCriterionLine(char *name, int linenumber)
{
        node *temp = L1.GetHeader();
        while((temp -> LineNumber != linenumber) && (temp != NULL))
          temp = temp -> left;
        if( temp == NULL)
        {
          printf("error in the source code\n");
          printf("check the source code\n");
          printf("at the line \n%d\t%s\n", linenumber, name);
          exit(0);
        }
        else
          temp -> SLICE = TRUE;
          return SUCCESS;
}

////////////////////////////////////////////////////////////////////////
//      Member Function : GetIoType
//      Purpose         : This function checks for presence of any input/
//                        output functions present in the statement. If
//                        found, it returns the type of function.
////////////////////////////////////////////////////////////////////////

IOTYPE
SliceClass::GetIoType(const StringClass& pbuffer)
{
        StringClass buffer = pbuffer.GetStr();
          if((buffer &= "getchar") != NULL)
                return GETCHAR;
          else if((buffer &= "gets") != NULL)
                return GETS;
          else if((buffer &= "scanf") != NULL)
                return SCANF;
          else if((buffer &= "printf") != NULL)
                return PRINTF;
          else if((buffer &= "cin") != NULL)
                return CIN;
          else if((buffer &= "cout") != NULL)
                return COUT;
          else if((buffer &= "fscanf") != NULL)
                return SCANF;
          else
                return VOIDTYPE;
}

// Here, variable can be a referenced, defined, or a reserved word. This
// function is used to check the variable. If the variable is a reserved word,
// then it returns PRESENT else NOTPRESENT.

BOOLEAN
SliceClass::IsAValidDeclaration(const StringClass& variable)
{
        StringClass buffer = variable.GetStr();
        int len = strlen(buffer.GetStr());
        if((((buffer &= "int") != NULL) && (len == 3)) ||
        (((buffer &= "char") != NULL)&& (len == 4)) ||
        (((buffer &= "float") != NULL) && (len == 4)) ||
```

```
        (((buffer &= "double") != NULL) && (len == 6)) ||
        (((buffer &= "friend") != NULL) && (len == 6)) ||
        (((buffer &= "return") != NULL) && (len == 6)) ||
        (((buffer &= "struct") != NULL) && (len == 5))||
        ((buffer &= "char *") != NULL) ||
        (((buffer &= "typedef") != NULL) && (len == 7)))
          return TRUE;
        else if(((buffer &= "int *") != NULL) || ((buffer &= "while") != NULL)
          || ((buffer &= "for") != NULL) || ((buffer &= "FILE") != NULL)
          || ((buffer &= "if") != NULL) || ((buffer &= "switch") != NULL)
          || ((buffer &= "exit") != NULL) ||((buffer &= "class") != NULL)
          || ((buffer &= "strcpy") != NULL) || ((buffer &= "strcmp") != NULL)||
         ((buffer &= "strlen") != NULL) || ((buffer &= "strcat") != NULL))
              return TRUE;
        else if(((buffer &= "strtok") != NULL) || ((buffer &= "NULL")!= NULL)
        || ((buffer &= "strstr") != NULL) || ((buffer &= "else") != NULL) ||
        ((buffer &= "break") != NULL) || ((buffer &= "feof") != NULL) ||
        ((buffer &= "this") && (len == 4)))
          return TRUE;
        else if(((buffer &= "private:")!= NULL) || ((buffer &= "public:")!= NULL)
        || ((buffer &= "private") != NULL) || ((buffer &= "public") != NULL) ||
        ((buffer &= "isalnum") != NULL) || (((buffer &= "new") != NULL) && (len
== 3)))
          return TRUE;
        char *var = buffer.GetStr();
        int length = strlen(var);
        len = 0;
        int i = 0;
        while(i < length)
        {
          if((isdigit(var[i])))
               len++;
               //return TRUE;
          i++;
        }
        if( len == length)
          return TRUE;
        return FALSE;
}

// This function checks for the presence of input output operands.

void
SliceClass::CheckForIoOPerands(const StringClass& buffer)
{
        char *string;
        IOTYPE type;
        type = GetIoType(buffer);
        string = buffer.GetStr();
        if( type != VOIDTYPE)
          InsertIntoIoList(type, buffer);
}

// If the variable referenced is in IO statement, then the variable is inserted
// into iolist.

void
SliceClass::InsertIntoIoList(IOTYPE type, const StringClass& pstring)
{
        int i=0;
        int count = 0;
        char *var;
        char *firststring, *sec, fin[MAXLINE];
        char variable[MAXLINE];
        StringClass string;
        string = pstring;
        if( type == SCANF)
        {
          //firststring = string &= OPENBRACKET;
```

```
  firststring = string && "(";
  if((var = strchr(firststring , '"')) != NULL)
  {
       var++;
       sec = strchr(var, '"');
  }
  while(var != sec)
  {
       *var = SPACE ;
       var++;
  }
  var++;
  i = 0;
  while((!isalnum(*var)) && (*var == UNDERSCORE)
       && (*var != CLOSEBRACKET))
       var++;
  while(*var != CLOSEBRACKET)
  {
       while((*var == SPACE ) || (*var == COMMA))
              var++;
       if( *var == '&')
              var++;
       count = 0;
      while((isalnum(*var)) || (*var == UNDERSCORE))
       {
              fin[count++] = *var;
              var++;
       }
       fin[count] = '\0';
       // THis Is the part I donot have access to the temhead.
       // For Compiling I use count
       InsertIntoRefDefList(IOREF, count, fin);
       //InsertIntoRefDefList(IOREF, temp -> LineNumber, fin);
  }
}
else if( type == GETS)
{
  firststring = string && "(";
  var = firststring;
  while(*var != ',')
  {
       while((*var == ' ') || (*var == '('))
       {
              var++;
              //cout << "Statement Reached" << endl;
       }
      while((isalnum(*var)) || (*var == '_'))
       {
              fin[count++] = *var;
              var++;
       }
       fin[count] = ENDOFLINE;
       if( strlen(fin) > 0)
              InsertIntoRefDefList(IOREF, count, fin);
  }
}
else if( type == GETCHAR)
{
  firststring = string && "(";
  i = 0;
  IgnoreSpaces(firststring, i);
  while((isalnum(firststring[i])) || (firststring[i] ==
UNDERSCORE))
       variable[count++] = firststring[i++];
  variable[count] = ENDOFLINE;
  // I need to get the temp from the calling member funtion
  //InsertIntoRefDefList(IOREF, temp -> LineNumber, variable);
  InsertIntoRefDefList(IOREF, count, variable);
}
```

```
        else if( type == SWITCH)
        {
          firststring = string &= "(";
          //printf("Got %s\n", firststring);
          //getchar();
          i = 0;
          IgnoreSpaces(firststring, i);
          if( firststring[i] == OPENBRACKET)
                i++;
          while(firststring[i] != CLOSEBRACKET)
          {
                count = 0;
                while((isalnum(firststring[i])) || (firststring[i] ==
                '_'))
                      variable[count++] = firststring[i++];
                variable[count] = ENDOFLINE;
                InsertIntoRefDefList(REFERED, count, variable);
                if( i > 250)
                {
                      printf("error in the parsing line\n");
                      exit(0);
                }
          }
        }
        else if( type == COUT)
        {
          memset(fin, NULL, MAXLINE);
          firststring = pstring.GetStr();
          firststring = strchr(firststring, '<<');
          strcpy(fin, firststring);
          count = 0;
          while( fin[count] != ';')
          {
                while((fin[count] == SPACE) || (fin[count] == TAB) ||
          (fin[count] == '<')||(fin[count] ==
PERIOD)||(fin[count] == OPENBRACKET) ||
(fin[count] == CLOSEBRACKET))
                      count++;
                if( fin[count] == '"')
                {
                      while( fin[count] != '<')
                      {
                            if( fin[count] == ';')
                                  return;
                            count++;
                      }
                }
                while((fin[count] == SPACE) || (fin[count] == TAB) ||
                (fin[count] == '<'))
                      count++;
                i = 0;
                memset(variable, NULL, MAXLINE);
                while((fin[count] == UNDERSCORE) ||
(isalnum(fin[count])))
                      variable[i++] = fin[count++];
                variable[i] = ENDOFLINE;
                //printf("Inserting Into Ref : %s\n", variable);
                if( strlen(variable) > 0)
                      InsertIntoRefDefList(REFERED, count, variable);
          }
        }
        else if( type == CIN)
        {
          memset(fin, NULL, MAXLINE);
          int count = 0;
          firststring = pstring.GetStr();
          firststring = strchr(firststring, '>>');
          strcpy(fin, firststring);
          memset(variable, NULL, MAXLINE);
```

```
        while(fin[count] != ';')
        {
             while((fin[count] == SPACE) || (fin[count] == TAB) ||
             (fin[count] == '>'))
                  count++;
             i = 0;
             while((fin[count] == UNDERSCORE) ||
(isalnum(fin[count])))
                  variable[i++] = fin[count++];
             variable[i] = ENDOFLINE;
             if( strlen(variable) > 0)
                  InsertIntoRefDefList(DEFERED, count, variable);
        }
      }
}

////////////////////////////////////////////////////////////////////////
// Member Function : InsertIntoRefList
// Purpose         : Once the variable is checked for reserved word, it
//                   is inserted into the reflist.
////////////////////////////////////////////////////////////////////////

void
SliceClass::InsertIntoRefList(StringClass& buffer)
{
      char variable[MAXLINE];
      char *string = buffer.GetStr();
      int len = strlen(string);
      int i = 0;
      int count = 0;
      while( i < len)
      {
        if((!(isalnum(string[i]))) &&(string[i] != UNDERSCORE))
             string[i] = SPACE;
             i++;
      }
      i = 0;
      while( i < len)
      {
        while((!(isalnum(string[i]))) &&(string[i] != UNDERSCORE))
             i++;
        count = 0;
        while((isalnum(string[i])) || (string[i] == UNDERSCORE))
             variable[count++] = string[i++];
        variable[count] = ENDOFLINE;
        if(( i < len) && (IsAValidDeclaration(variable)))
        {
             InsertIntoRefDefList(REFERED, count, variable);
        }
      }

}

////////////////////////////////////////////////////////////////////////
//   Member Function : DeleteRefFromRefList
//   Purpose         : After inserting the ref list into the respective
//                     node containing the line information, the reflist
//                     is deleted so that the same ref list can be used for
//                     remaining statements.
////////////////////////////////////////////////////////////////////////

void
SliceClass::DeleteRefFromRefList(RefDef* calptr)
{
      RefDef *temp;
      RefDef *front;
      temp = RefHeader;
      front = temp;
      if( RefHeader -> variable == calptr -> variable)
```

```
        {
          temp = RefHeader;
          RefHeader = RefHeader -> next;
          delete temp;
          return;
        }
        while( temp != NULL)
        {
          if( temp -> variable == calptr -> variable)
          {
               front = temp -> next;
               delete temp;
          }
          temp = temp -> next;
        }
}

////////////////////////////////////////////////////////////////////////
//    Member Function : InsertIoRefIntoDefList
//    Purpose         : This function inserts the variable into the Def
//                      list.
////////////////////////////////////////////////////////////////////////

void
SliceClass::InsertIoRefIntoDefList(RefDef *calptr)
{
        int flag = -1;
        RefDef *temp;
        temp = DefHeader;
        while( temp -> next!= NULL)
        {
          if( temp -> variable == calptr -> variable)
          {
               flag = PRESENT;
               break;
          }
          temp = temp -> next;
        }
        if( flag != PRESENT)
        {
          RefDef *tem = new RefDef;
          tem -> variable = calptr -> variable;
          tem -> linenumber = calptr -> linenumber;
          temp ->PartType = calptr -> PartType;
          return;
        }
}

////////////////////////////////////////////////////////////////////////
//     Member Function : DeleteRefDefIoRef
//     Purpose         : This function after parsing each statement, and
//                       inserting the variables into the reflist of the
//                       statement itself, deletes the global lists, reflist,
//                       deflist, and iolist. This makes the global lists
//                       pointing to null which can be used to store ref, def
//                       variables for remaining statements.
////////////////////////////////////////////////////////////////////////

void
SliceClass::DeleteRefDefIoRef(RefDef* temp)
{
        RefDef *front;
        while(temp != NULL)
        {
          front = temp;
          temp = temp -> next;
          delete front;
        }
}
```

```
// It inserts the first value refered into ref list.

void
SliceClass::InsertIntoHeaderAllRef(node *temhead, RefDef *calptr)
{
       RefDef *temp;
       if( temhead -> RefHeader == NULL)
       {
         RefDef *node = new RefDef;
         node -> variable = calptr -> variable;
         node -> linenumber = calptr -> linenumber;
         node -> PartType = calptr -> PartType;
         node -> next = NULL;
         temhead -> RefHeader = node;
       }
       else
       {
         temp = temhead -> RefHeader;
         while((temp -> next  != NULL) && ( temp -> variable != calptr ->
variable))
              temp = temp -> next;
         if((temp -> next  == NULL) && ( temp -> variable != calptr ->
variable))
         {
              RefDef *node = new RefDef;
              node -> variable = calptr -> variable;
              node -> linenumber = calptr -> linenumber;
              node -> PartType = calptr -> PartType;
              node -> next = NULL;
              temp -> next = node;
         }
       }
}

// This function is used to insert the variable into Deflist.

void
SliceClass::InsertIntoHeaderAllDef(node *temhead, RefDef *calptr)
{
       RefDef *temp;
       if( temhead -> DefHeader == NULL)
       {
         RefDef *node = new RefDef;
         node -> variable = calptr -> variable;
         node -> linenumber = calptr -> linenumber;
         node -> PartType = calptr -> PartType;
         node -> next = NULL;
         temhead -> DefHeader = node;
       }
       else
       {
         temp = temhead -> DefHeader;
         while((temp -> next  != NULL) && ( temp -> variable != calptr ->
variable))
              temp = temp -> next;
         if((temp -> next  == NULL) && ( temp -> variable != calptr ->
variable))
         {
              RefDef *node = new RefDef;
              node -> variable = calptr -> variable;
              node -> linenumber = calptr -> linenumber;
              node -> PartType = calptr -> PartType;
              node -> next = NULL;
              temp -> next = node;
         }
       }
}

// This is a pseudo copy constructor used to make a copy of the loader class.
```

```
void
SliceClass::CopyLoader(LoaderClass& Loader)
{
       L1 =  Loader;
}

void
SliceClass::InsertControlVar(const StringClass& buffer, int linenumber)
{
       char *string;
       char variable[80];
       int index;
       int i;
       int length;
       length = strlen(buffer.GetStr());
       index = 0;
       string = buffer.GetStr();
       while(index < length)
       {
         if((!isalnum(string[index])) && (string[index] != UNDERSCORE))
              string[index] = SPACE;
         index++;
       }
       //printf("Buffer in ControlStatement is %s\n", string);
       index = 0;
       while(index < length)
       {
         IgnoreSpaces(string, index);
         i = 0;
         while((isalnum(string[index]))|| (string[index] ==
       UNDERSCORE))
              variable[i++] = string[index++];
         variable[i] = ENDOFLINE;
         if((IsAValidDeclaration(variable))&& (strlen(variable) > 0))
              InsertIntoRefDefList(REFERED, linenumber, variable);
       }
}

// This function is used to print the variable present in the RefDef list.

void
SliceClass::PrintVar( RefDef *temp)
{
       int line = 0;
       line = temp -> linenumber;
       while( temp != NULL)
       {
         printf("%s  ", temp -> variable.GetStr());
         temp = temp -> next;
       }
       printf(" at line:%d\n", line);
}

////////////////////////////////////////////////////////////////////////
//     Member Function : PrintFinal
//     Purpose         : This function is to print the program resident
//                       in the loader.
////////////////////////////////////////////////////////////////////////

void
SliceClass::PrintFinal()
{
       node *ptr = L1.GetHeader();
       while( ptr != NULL)
       {
         if( ptr -> RefHeader != NULL)
              PrintVar(ptr -> RefHeader);
         else if( ptr -> DefHeader != NULL)
              PrintVar(ptr -> DefHeader);
```

```
        ptr = ptr -> left;
      }
}

// This function is used to check for the presence of reserve worr new.
// if found parses the statement by itself and finds out the ref and def
// variables that are present in the statement.

int
SliceClass::CheckForNew(StringClass  var)
{
      if((var &= "new") != NULL)
        return PRESENT;
      else
        return NOTPRESENT;
}

void
SliceClass::CheckInsideForNew(char *var)
{
      if( strstr(var, "=") != NULL)
      {
      char variable[MAXLINE];
      char sec[MAXLINE];
      char third[MAXLINE];
      int count, index;
      memset(variable, NULL, MAXLINE);
      memset(sec, NULL, MAXLINE);
      memset(third, NULL, MAXLINE);
      strcpy(variable, var);
      strcpy(sec, strchr(variable, 'new '));
      printf("sec is %s\n", sec);
      count = 0;
      index = 0;
      while((isalnum(sec[count])) || (sec[count] == UNDERSCORE))
        third[index++] = sec[count++];
      if( strcmp(third, "new") == 0)
      {
        count = 0;
        while( variable[count] != EQUALTO)
        {
            memset(sec, NULL, MAXLINE);
            index = 0;
            IgnoreSpaces(variable, count);
            while((isalnum(variable[count])) || (variable[count]
            ==UNDERSCORE))
            sec[index++] = variable[count++];
            sec[index] = ENDOFLINE;
            InsertIntoRefDefList(DEFERED, count, sec);
        }
        while(variable[count] != COLON)
        {
            IgnoreSpaces(variable, count);
            memset(sec, NULL, MAXLINE);
            index = 0;
            while((isalnum(variable[count])) || (variable[count]
            ==UNDERSCORE))
            sec[index++] = variable[count++];
            sec[index] = ENDOFLINE;
            InsertIntoRefDefList(REFERED, count, sec);
            IgnoreSpaces(variable, count);
            while(variable[count] != SQCLOSEBRACKET)
            {
                  IgnoreSpaces(variable, count);
                  if((isalpha(variable[count])) ||
                        (variable[count] == UNDERSCORE))
                  {
                        while((isalpha(variable[count]))
                              ||(variable[count] == UNDERSCORE))
```

```
                                        sec[index++] = variable[count++];
                                sec[index] = ENDOFLINE;
                                InsertIntoRefDefList(REFERED, count, sec);
                        }
                        else
                                IgnoreDigitsOperators(variable, count);
                        }
          }
        }
}
}

// This function is used to print the program present in the laoderclass.
void
SliceClass::PrintProgram()
{
        L1.PrintProgram();
}

////////////////////////////////////////////////////////////////////////
//      Member Function : ComputeRefDefForMain
//      Purpose         : This function is used to parse the programs
//                        written in C or in C++ without classes. It
//                        computes the ref and def values and inserts
//                        them into the reflist and deflist.
////////////////////////////////////////////////////////////////////////

int
SliceClass::ComputeRefDefForMain()
{
        RefClass *refclass;
        RefHeader = NULL;
        DefHeader = NULL;
        IoRefHeader = NULL;
        int beginnumber = 0;
        int endNumber = 0;
        refclass = L1.GetRefClass();
        if( refclass == NULL)
        {
          printf("error in computing refdef for main part of program\n");
          return FAILURE;
        }
        if( refclass -> Member == "main()")
        {
          beginnumber = refclass -> BeginLine;
          endNumber = refclass -> EndLine;
          MarkAllSlices(beginnumber, endNumber);
          ComputeRefDef(beginnumber, endNumber);
          MarkControlStatForRefDef(beginnumber, endNumber);
          RefHeader = NULL;
          DefHeader = NULL;
          IoRefHeader = NULL;
          return SUCCESS;
        }
        else
        {
          printf("error while inserting into refclass\n");
          return FAILURE;
        }
}

////////////////////////////// End of Slice.C ////////////////////////


////////////////////////////////////////////////////////////////////////
//
//                              FlowControl.C.
//
////////////////////////////////////////////////////////////////////////
```

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include<ctype.h>
#include"slice.h"


////////////////////////////////////////////////////////////////////////
//      Member Function : CheckForControlStatements
//      Purpose         : To include the control statements if the
//                        statements within the control loop are marked
//                        for slice.
////////////////////////////////////////////////////////////////////////

void
SliceClass::CheckControlStatements(node* start, node* end)
{
       int i = 0;
       int SLICE = FALSE;
       int line = start -> LineNumber;
       int currentline = 0;
       RefDef *ref, *def;
       StringClass variable;
       StringClass secvariable;
       char *buffer;
       node *ptr = start;
       //printf("info is %s\n", ptr -> LineInformation.GetStr());
       while((ptr -> LineNumber != end -> LineNumber) && (ptr != NULL))
       {
         if( ptr -> SLICE == TRUE)
         {
              SLICE = TRUE;
              currentline = ptr -> LineNumber;
              break;
         }
         ptr = ptr -> left;
       }
       if(SLICE == TRUE)
       {
         //printf("Statement Reached In Slice\n");
         end -> SLICE = TRUE;
         start -> SLICE = TRUE;
         //printf("Including %d %s\n", start -> LineNumber, start ->
LineInformation.GetStr());
         //printf("Including %d %s\n", end -> LineNumber, end ->
LineInformation.GetStr());
         ptr = start;
         variable = ptr -> LineInformation;
         //printf("Variable Is %s", variable.GetStr());
         while( ptr -> LineNumber != currentline)
              ptr = ptr -> left;
         def = ptr -> DefHeader;
         variable = " ";
         //printf("Variable Is %s", variable.GetStr());
         while( def != NULL)
         {
              variable += def -> variable;
              variable += " ";
              def = def -> next;
         }
         ref = ptr -> RefHeader;
         while( ref != NULL)
         {
              variable += ref -> variable;
              variable += " ";
              ref = ref -> next;
```

```
        }
        ref = start -> RefHeader;
        while( ref != NULL)
        {
             variable += ref -> variable;
             variable += " ";
             ref = ref -> next;
        }
        int flag = 0;
        if( end -> LineNumber < currentline)
        {
             BeginSlicingProgram(end -> LineNumber -1, variable);
             flag = 1;
        }
        if((WHILE == TRUE) && ( flag == 0))
        {
             LoadActiveVars(variable);
             SliceControlLoop(start, end);
        }
      }
}

////////////////////////////////////////////////////////////////////////
//     Member Function : SliceControlLoop
//     Purpose         : It checks for the presence of control
//                       statement. If found any, it will find the
//                       starting and ending line.
////////////////////////////////////////////////////////////////////////

int
SliceClass::SliceControlLoop(int currentline)
{
      int start;
      int flag = 0;
      int beginline;
      node *ptr = L1.GetHeader();
      beginline = GetBeginLine(currentline);
      if( beginline == ERROR)
      {
        printf("error in choosing the line number , try again\n");
        return FAILURE;
      }

      // This Begin Number is the Starting LIne Of the Memeber Function
      // In Which the criteion Line Exixts.

      while(( ptr -> LineNumber != beginline) && ( ptr != NULL))
        ptr = ptr -> left;
      if((ptr == NULL) && ( ptr -> LineNumber != beginline))
      {
        printf("error in locating the begin line\n");
        printf("reached slice contol loop\n");
        exit(0);
      }
      while(ptr -> LineNumber <= currentline)
      {
        //printf("Getting Into The Control Flow Check For the Control Loop\n");
        //printf("Info : %s\n", ptr -> LineInformation.GetStr());
        flag = 0;
        WHILE = FALSE;
        if(IsAControlStatement(ptr -> LineInformation) == TRUE)
        {
             if((ptr -> LineInformation &= "{") != NULL)
                    start = ptr -> LineNumber;
             else
             {
                    ptr = ptr -> left;
                    while(((ptr -> LineInformation  &= "{") == NULL)
                    && ( ptr != NULL))
```

```
                        ptr = ptr -> left;
                    if( ptr == NULL)
                    {
              printf("error in getting started in control loop\n");
                        printf("exiting\n");
                        exit(0);
                    }
                    else
                        start = ptr -> LineNumber;
              }
                    flag = 1;
                    //printf( "Reached : %d\n", ptr -> LineNumber);
                    MarkControlFlow(start, currentline, &ptr);
          }
          else if( flag != 1)
               ptr = ptr -> left;
        }
        return SUCCESS;
}


////////////////////////////////////////////////////////////////////
//      Member Function : InsertIntoHeader
//      Purpose         : After calculating the ref and def of the
//                        statement, the values are inserted into the
//                        list containing the information about the line
////////////////////////////////////////////////////////////////////

void
SliceClass::InsertIntoHeader(RefDef* header, RefDef* calptr)
{
        RefDef *tem;
        RefDef* theader = new RefDef;
        theader -> variable  = calptr -> variable;
        theader -> linenumber  = calptr -> linenumber;
        theader -> PartType  = calptr -> PartType;
        theader -> next = NULL;
        if( header == NULL)
        {
          header = theader;
          header -> next = NULL;
          return;
        }
        else
        {
          int flag = NOTPRESENT;
          tem = header;
          while((tem -> next != NULL) && (tem -> linenumber == calptr ->
linenumber))
          {
               if( tem -> variable == calptr -> variable)
               {
                    flag = PRESENT;
                    break;
               }
               else
                    tem = tem -> next;
          }
          if ( flag == NOTPRESENT)
          {
               tem -> next = theader;
               return;
          }
        }
}
// After Computing the RefDef for a particular Member Function Call This Func

////////////////////////////////////////////////////////////////////
//      Member Function : BeginSlicingProgram
```

```
//     Purpose          : To slice the proggram from the starting line to
//                        the line of interest.
////////////////////////////////////////////////////////////////////////
int
SliceClass::BeginSlicingProgram(int linenumber, const StringClass& string)
{
       node *ptr;
       int currentline;
       BOOLEAN reftype = FALSE;
       LoadActiveVars(string); // This is the way of adding into active list.
       ptr = L1.GetHeader();
       currentline = GetBeginLine(linenumber);
       if( currentline  == ERROR)
       {
         printf("error in choosing the line number , try again\n");
         return FAILURE;
       }
       while(( ptr -> LineNumber != linenumber) && ( ptr != NULL))
         ptr = ptr -> left;
       if( ptr -> LineNumber == linenumber)
       {
         while( ptr ->LineNumber > currentline)
         {
              reftype = CheckForActiveSet(ptr);
              if( reftype == TRUE)
              {
              // This Function Inserts All the Ref Variable Into
       // Active List
                    VariableRefered(ptr);
                    ptr -> SLICE = TRUE;
              }
              IncludeMemberFunction(ptr);
         ptr = ptr -> right; // Going Back till Reaching First Line
                             // of the Prog
         }
         return SUCCESS;
       }
       else
       {
         printf("cannot locate the criterion line\n");
         printf("exiting the process\n");
         exit(0);
       }
}

// This function checks for active variables in deflist If any
// match is found, then the variable is removed them from the active list. This
// process makes slicing fast as it avoids checking for same variable repeatedly
// in the deflist and trying to include the statement into the final slice.

BOOLEAN
SliceClass::CheckForActiveSet(node* ptr)
{
       int count = 0;
       int returntype = NOTFOUND;
       RefDef *tem = ptr -> DefHeader;
       while( tem != NULL)
       {
         returntype = CheckPresenceOfActiveVars(ActiveVar, tem -> variable);
         if( returntype == FOUND)
              count++;
         tem = tem -> next;
       }
       if( count > 0)
         return TRUE;
       else
         return FALSE;
}
```

```
////////////////////////////////////////////////////////////////////////
//      Member Function : VariableRefered
//      Purpose         : To include variables refered in the program into
//                        ActiveSet
////////////////////////////////////////////////////////////////////////
void
SliceClass::VariableRefered(node* ptr)
{
       RefDef* temp = RefHeader;
       temp = ptr -> RefHeader;
       while( temp != NULL)
       {
         InsertActiveVars(temp -> variable);
         temp = temp -> next;
       }
       temp = ptr -> DefHeader;
       while( temp != NULL)
       {
         InsertActiveVars(temp -> variable);
         temp = temp -> next;
       }
}

////////////////////////////////////////////////////////////////////////
//  Member Function : CheckPresenceOfActiveVars
//  Purpose         : To  check for the presence of any variable that is
//                    either referenced or defined in the previous statement.
//                    if found, and that variable is refering another variable,
//                    then the variable which is being referenced is also
//                    included in the active set.
////////////////////////////////////////////////////////////////////////

int
SliceClass::CheckPresenceOfActiveVars( RefDef *ptr, const StringClass& info)
{
       RefDef *tem;
       tem = ptr;
       if( tem -> next == NULL)
       {
         if( tem -> variable == info)
         {
              //delete tem;
              return FOUND;
         }
         return NOTFOUND;
       }
       tem = ptr;
       //ptr = ptr -> next;
       while( ptr != NULL)
       {
         if( ptr -> variable == info)
         {
              //tem -> next = ptr -> next;
              return FOUND;
         }
         else
         {
              tem = ptr;
              ptr = ptr -> next;
         }
       }
       if( ptr == NULL)
         return NOTFOUND;
}

// This function is used to insert the active variables into the linked list
// pointed by ActiveVar.

void
```

```
SliceClass::InsertActiveVars(const StringClass& variable)
{
       //printf("Inserting the Variable %s", variable.GetStr());
       if( ActiveVar == NULL)
       {
         RefDef *set = new RefDef;
         set -> variable = variable ;
         set -> linenumber = 0;
         set -> PartType = PUBLIC;
         set -> next = NULL;
         ActiveVar  = set;
         ActiveVar -> next = NULL;
         return;
       }
       RefDef* tem = ActiveVar;
       while(( tem -> next != NULL)&&(tem -> variable != variable))
              tem = tem -> next;
       if(( tem -> next == NULL)&&(tem -> variable != variable))
       {
         RefDef *set = new RefDef;
         set -> variable = variable ;
         set -> linenumber = 0;
         set -> PartType = PUBLIC;
         set -> next = NULL;
         tem -> next = set;
         tem = tem -> next;
         tem -> next = NULL;
         return;
       }
}

// This function is used to check for presence of control statements.
// If found, it will return TRUE otherwise FALSE.

BOOLEAN
SliceClass::IsAControlStatement(const StringClass& info)
{
       StringClass LineInformation = info.GetStr();
       if((LineInformation &= "while") != NULL)
       {
         WHILE = TRUE;
         return TRUE;
       }
       else if((( LineInformation &= "if") != NULL)||((LineInformation &=
"else") != NULL)
         || ((LineInformation &= "while") != NULL)||
              ((LineInformation &= "for") != NULL) ||
              ((LineInformation &= "switch") != NULL) ||
              ((LineInformation &= "do") != NULL))
                     return TRUE;
       else
         return FALSE;

}
// If a control statement is identified, its scope of influence is determined.
// If the statement contains only one statement, then statement is marked with
// metoo set.

void
SliceClass::MarkControlFlow(const int& start, int currentline, node **ptr)
{
       node *s, *e;
       int count = 0;
       int end = 0;
       // Makesure start and ptr are at the same point
       while( (*ptr) -> LineNumber != start)
         (*ptr) = (*ptr) -> right; // Traversing Back
       s = (*ptr);
       (*ptr) = (*ptr) -> left;
```

```
        while((((*ptr) -> LineInformation &= "}") == NULL) || ( count != 0))
        {
          CheckCloseBracket(*ptr, count);
          if( IsAControlStatement((*ptr) -> LineInformation) == TRUE)
          {
               //printf("LineNumber Is %d\n", (*ptr) -> LineNumber);
               ControlStatWithInControlState(*ptr);
          }
          CheckOpenBracket(*ptr, count); // To avoid writing the code again
          (*ptr) = (*ptr) -> left;
        }
        if((((*ptr) -> LineInformation &= "}") != NULL) && ( count == 0))
          end = (*ptr) -> LineNumber;
          e = (*ptr);
        CheckControlStatements(s, e);
}

////////////////////////////////////////////////////////////////////////
//      Member Function : CheckCloseBracket
//      Purpose         : To  find the end line of the control loop. This helps
//                        in locating the range of control statements
////////////////////////////////////////////////////////////////////////

void
SliceClass::CheckCloseBracket(node* ptr, int& count)
{
        if(( ptr -> LineInformation &= "}") != NULL)
        {
          if( count == 0)
               count = 0;
          else
               count--;
        }
}

////////////////////////////////////////////////////////////////////////
//      Member Function : CheckCloseBracket
//      Purpose         : To  find the start line of the control loop.
////////////////////////////////////////////////////////////////////////

void
SliceClass::CheckOpenBracket(node* ptr, int& count)
{
        if( ptr -> LineInformation &= "{")
          count++;
}

////////////////////////////////////////////////////////////////////////
//      Member Function : LoadActiveVars
//      Purpose         : This function is used to insert the active vars into
//                        the active set. The active set initially contains the
//                        criterion variables. Once the slicing operation is
//                        started, all other variables that influence the
//                        criterion variables are also included in active set.
////////////////////////////////////////////////////////////////////////

void
SliceClass::LoadActiveVars(const StringClass& string)
{
        int len = 0;
        int count = 0;
        char var[MAXLINE];
        int i = 0;
        char *temp  = string.GetStr();
        //printf(" String Is %s\n", temp);
        len = strlen(temp);
        while( i < len)
        {
          count = 0;
```

```
        while((temp[i] != ' ') && ((isalnum(temp[i]) || (temp[i] ==
UNDERSCORE))))
             var[count++] = temp[i++];
        //cout << var << endl;
             var[count] = ENDOFLINE;
        if(strlen(var) >0)
             InsertActiveVars(var);
        memset(var, NULL, 80);
        i++;
      }
}

////////////////////////////////////////////////////////////////////
//Member Function :  IsAValidMember
//Purpose         :  Checks for the member function defined in the class
//                   of interest. If the defintion and function donot
//                   coincide, it will give an error. This function is used to
//                   eliminate the posibility of existence of functional
//                   overloading and inheritance.
////////////////////////////////////////////////////////////////////

BOOLEAN
SliceClass::IsAValidMember( StringClass buf)
{
      int len1=0;
      int len2=0;
      RefClass *ref = L1.GetRefClass();
      RefClass *temp;
      if( ref != NULL)
      {
        temp = ref;
        while(temp != NULL)
        {
             len1 = strlen(temp -> Member.GetStr());
             len2 = strlen(buf.GetStr());
             if(((( temp -> Member &= buf) != NULL)&&(len1 ==
        len2))&&(temp -> PartsType == PUBLIC))
                  return TRUE;
             else
                  temp = temp -> next;
        }
        return FALSE;
      }
}

////////////////////////////////////////////////////////////////////
//  Member Function : GetBeginLine
//  Purpose         : This function is used to get the starting line of
//                    member function that contains the variable of
//                    interest. It checks the refclass which contains
//                    information about each memeber function of the class
//                    of interest, their starting and ending line nuember.
////////////////////////////////////////////////////////////////////

int
SliceClass::GetBeginLine(int number)
{
      RefClass *ref = L1.GetRefClass();
      while(ref -> PartsType == PRIVATE)
        ref = ref -> next;
        //printf("Begin Line Is %d\n", number);
      while(ref != NULL)
      {
        if((ref -> BeginLine < number) && ( ref -> EndLine > number))
             return ref -> BeginLine;
        else
             ref = ref -> next;
      }
      if( ref == NULL)
```

```
        {
          printf("error in selecting the class and line number\n");
          //exit(0);
          return ERROR;
        }
}

////////////////////////////////////////////////////////////////////////
//    Member Function : MarkControlStatForRefDef
//    Purpose         : It checks for presence of any control statements
//                      if found, marks its scope of influence.
////////////////////////////////////////////////////////////////////////

void
SliceClass::MarkControlStatForRefDef(int beginline, int endline)
{
       node *ptr = L1.GetHeader();
       while((ptr -> LineNumber != beginline)&&(ptr != NULL))
         ptr = ptr -> left;
       if( ptr != NULL)
       {
         while( ptr -> LineNumber != endline)
         {
              if(IsAControlStatement(ptr -> LineInformation) ==
         TRUE)
              {
                     StringClass string = ptr ->
       LineInformation.GetStr();
                     if((string &= "{") == NULL)
                     {
                            node *temp = ptr -> left;
                            while(strlen(temp ->
                            LineInformation.GetStr())==0)
                                   temp = temp -> left;
                            StringClass str1 =  temp ->
                            LineInformation.GetStr();
                            if((str1 &= "{") != NULL)
                            {
                                   ptr -> INCLUDE = temp -> LineNumber;
                                   temp -> RefHeader = ptr ->
                            RefHeader;
                                   temp ->  DefHeader = ptr ->
                            DefHeader;
                            }
                            else
                            {
                                   ptr -> INCLUDE = temp -> LineNumber;
                                   ptr -> RefHeader = temp ->
                            RefHeader;
                                   ptr ->  DefHeader = temp ->
                            DefHeader;
                            }
                     }
              }
              ptr = ptr -> left;
         }
       }
}

////////////////////////////////////////////////////////////////////////
//     Member Function : PrintSlice
//     Purpose         : It prints out the final slice
////////////////////////////////////////////////////////////////////////

void
SliceClass::PrintSlice(int linenumber)
{
       RefClass *ref = L1.GetRefClass();
       node *start, *end, *sub;
```

```
        while( ref -> PartsType == PRIVATE)
          ref = ref -> next;
        RefClass *tem = ref;
        while(ref != NULL)
        {
          if((ref -> BeginLine < linenumber) && ( ref -> EndLine >
        linenumber))
                break;
          else
                ref = ref -> next;
        }
        node *ptr = L1.GetHeader();
        node *sec;
        //printf("Start State Is %d\n", ref -> BeginLine);
        while( ptr -> LineNumber != ref -> BeginLine)
          ptr = ptr -> left;
        start = ptr;
        sub = start;
        while( sub -> LineNumber != ref -> EndLine)
          sub = sub -> left;
        end = sub;
        //printf("Start is %d and end is %d\n", start -> LineNumber, end
        -> LineNumber);
        printf("%d %s", ptr -> LineNumber, ptr ->
LineInformation.GetStr());
        ptr -> SLICE = TRUE;
        ptr = ptr -> left;
        printf("%d %s", ptr -> LineNumber, ptr ->
LineInformation.GetStr());
        ptr -> SLICE = TRUE;
        while( ptr -> LineNumber != ref -> EndLine)
        {
          if(( ptr -> SLICE == TRUE)&&(ptr -> INCLUDE == 0))
          {
                printf("%d  %s", ptr -> LineNumber, ptr ->
        LineInformation.GetStr());
          }
          else if(ptr -> INCLUDE != 0)
          {
                sec = ptr;
                ptr = ptr -> left;
                if(( ptr -> SLICE == TRUE) && (ptr -> LineNumber <=
          linenumber))
                {
                        printf("%d  %s", sec -> LineNumber, sec ->
          LineInformation.GetStr());
                        sec -> SLICE = TRUE;
                        printf("%d  %s", ptr -> LineNumber, ptr ->
                LineInformation.GetStr());
                }
          }
          ptr = ptr -> left;
        }
        printf("%d %s", ptr -> LineNumber, ptr ->
LineInformation.GetStr());
        ptr -> SLICE = TRUE;
        PostSliceMarks(start, end);
}

//////////////////////////////////////////////////////////////////////
// Member Function : ControlStatWithInControlState
// Purpose         : It checks for presence of control statements within
//                   the control statements and finds their scope of
//                   influence. This function helps in slicing the control loops
//                   within the control loops and including the control loops if
//                   the statements with its influence are marked for slice.
//////////////////////////////////////////////////////////////////////

void
```

```
SliceClass::ControlStatWithInControlState(node *ptr)
{
       node *start;
       int flag = FALSE;
       if(( ptr -> LineInformation &= "{") != NULL)
       {
         start = ptr;
         flag = TRUE;
       }
       else if((ptr -> left -> LineInformation &= "{") != NULL)
       {
         start = ptr -> left;
         ptr = ptr -> left;
         flag = TRUE;
       }
       if( flag == FALSE)
       {
         ptr -> INCLUDE = TRUE;
         return;
       }
       ptr = ptr -> left;
       int count = 0;
       while(((ptr -> LineInformation &= "}") == NULL) || (count != 0))
       {
         if( IsAControlStatement(ptr -> LineInformation) == TRUE)
              ControlStatWithInControlState(ptr);
         CheckCloseBracket(ptr , count);
         CheckOpenBracket(ptr, count); // To avoid writing the code
                                       // again
         ptr = ptr -> left;
       }
       node *end = ptr;
       CheckControlStatements(start, end);
}

//////////////////////////////////////////////////////////////////////////
//     Member Function : DeleteActiveVars
//     Purpose         : It deletes the active variables from the list after
//                       performing the slice based on the variables of
//                       interest. This process is performed when the slicing
//                       based on the desired criterion is being performed and
//                       another request based on different criterion is being
//                       made.
//////////////////////////////////////////////////////////////////////////
void
SliceClass::DeleteActiveVars()
{
       RefDef *ptr;
       if( ActiveVar != NULL)
       {
         while( ActiveVar != NULL)
         {
              ptr = ActiveVar;
              ActiveVar = ActiveVar -> next;
              delete ptr;
         }
         delete ActiveVar;
       }
}

// This function is to slice the while and other necessary control loops

void
SliceClass::SliceControlLoop(node *start, node *end)
{
       int currentline = start -> LineNumber;
       node *ptr = end;
       BOOLEAN reftype = FALSE;
       while( ptr ->LineNumber > currentline)
```

```
        {
          reftype = CheckForActiveSet(ptr);
          if( reftype == TRUE)
          {
               VariableRefered(ptr);
               ptr -> SLICE = TRUE;
          }
               ptr = ptr -> right; // going back till reaching first
                              //  line of the program
        }
}

// This function is used to mark all statements slice marks to false.

void
SliceClass::MarkAllSlices(void)
{
       node *ptr = L1.GetHeader();
       while(ptr != NULL)
       {
         ptr -> SLICE = FALSE;
         ptr -> metoo = TRUE;
         ptr = ptr -> left;
       }
}

////////////////////////////////////////////////////////////////////////
//  Member Function : PostSliceMarks
//  Purpose         : This function is used to set the post slice marks
//                    to exclude the statements that are not influenced by the
//                    variable of interest, and active variables.
////////////////////////////////////////////////////////////////////////

void
SliceClass::PostSliceMarks(node* start, node *end)
{
       while(start -> LineNumber != end -> LineNumber)
       {
         if( start -> SLICE == FALSE)
              start -> metoo = FALSE;
         start = start -> left;
       }
}

////////////////////////////////////////////////////////////////////////
//     Member Function : PrintFinalSlice
//     Purpose         : This function is used to print the part of the
//                      program which is sliced based on the variable and
//                     line of interest.
////////////////////////////////////////////////////////////////////////

void
SliceClass::PrintFinalSlice(char *name)
{
       node *ptr = L1.GetHeader();
       FILE *fp;
       fp = fopen(name, "w+");
       while(ptr != NULL)
       {
         if(( ptr -> SLICE == TRUE) || (ptr -> metoo == TRUE))
              fprintf(fp, "%s", ptr -> LineInformation.GetStr());
         ptr = ptr -> left;
       }
}

////////////////////////////////////////////////////////////////////////
//  Member Function : Display
//  Purpose         : It is used to display the program on to the scree.
////////////////////////////////////////////////////////////////////////
```

```
void
SliceClass::Display()
{
       node *ptr = L1.GetHeader();
       int count;
       count = 0;
       while(ptr != NULL)
       {
         if(( ptr -> SLICE == TRUE) || (ptr -> metoo == TRUE))
              count++;
         ptr = ptr -> left;
       }
       ptr = L1.GetHeader();
       int page = (count / 20) + 1;
       int num = 1;
       int var=1;
       printf("-----------------------------------\n");
       printf("page %d of %d\n", num, page);
       while(ptr != NULL)
       {
         if(( ptr -> SLICE == TRUE) || (ptr -> metoo == TRUE))
         {
              printf("%d  %s", ptr -> LineNumber, ptr ->
LineInformation.GetStr());
              var++;
         }
         if(var == 20)
         {
              num++;
              printf("-----------------------------------\n");
              printf("press enter to continue ...");
              getchar();
              printf("\n");
              printf("-----------------------------------\n");
              printf("page %d of %d\n", num, page);
              var = 1;
         }
         ptr = ptr -> left;
       }
       printf("-----------------------------------\n");
}

////////////////////////////////////////////////////////////////////////
// Member Function : IncludeMemberFunction
// Purpose         : This function is used to check for presence of memeber
//                   function in statements which fall between begin line and
//                   line of interest. If found, the slice mark for that
//                   statement is set to TRUE
////////////////////////////////////////////////////////////////////////

void
SliceClass::IncludeMemberFunction(node* ptr)
{
         RefClass* ref = L1.GetRefClass();
         if( ref == NULL)
              return;
         else
         {
              while( ref -> PartsType == PRIVATE)
                     ref = ref -> next;
              while( ref != NULL)
              {
                     if(strstr( ptr -> LineInformation.GetStr(), ref ->
                     Member.GetStr()) != NULL)
                     {
                            ptr -> SLICE = TRUE;
                            return;
                     }
                     else
```

```
                        ref  = ref -> next;
                }
        }
}

// This function set all metoo and slice marks .

void
SliceClass::Set()
{
       node *ptr = L1.GetHeader();
       while(ptr != NULL)
       {
         ptr -> SLICE = FALSE;
         ptr -> metoo = TRUE;
         ptr = ptr -> left;
       }
}

////////////////////////////////////////////////////////////////////////
// Member Function : PrintActiveVars
// Purpose         : This function is used to print the active vars that
//                   are inserted into active set.
////////////////////////////////////////////////////////////////////////

void
SliceClass::PrintActiveVars()
{
       if( ActiveVar != NULL)
       {
         printf("active var:  ");
         RefDef *set = ActiveVar;
         while( set != NULL)
         {
              printf("\t%s", set -> variable.GetStr());
              set = set -> next;
         }
         printf("\n");
       }
}

////////////////////// End of FlowControl.C ////////////////////////

////////////////////////////////////////////
//            String.C
////////////////////////////////////////////

#include<iostream.h>
#include<string.h>
#include<stdlib.h>
#include "string.h"

StringClass::StringClass()
{
       len = 0;
       str = new char;
}

StringClass::StringClass(char* s)
{
       len = strlen(s);
       str = new char[len+1];
       strcpy (str,s);
}

StringClass::StringClass(const StringClass& St)
{
       len = St.len;
       delete str;
```

```
        str = new char[len+1];
        strcpy(str,St.str);

}

char*
StringClass::GetStr(void) const
{
        return str;
}

// Copies contents of one string class to the given class

char*
StringClass::operator&=(const StringClass& string)
{
         char *temp = strstr(str,string.str);
         if( temp == NULL)
         {
          //cout << " Not Preset" << endl;
          return ((char *) NULL);
         }
         else
          return temp;
}

StringClass&
StringClass::operator=(const char * tem)        //assigns values to given
construct
{
        len = strlen(tem);
        str = new char[len + 1];
        strcpy(str,tem);
        return *this;
}

void
StringClass::operator=(const StringClass& String)
{
        if (this == &String)
          return;
        delete str;;
        len = String.len;
        str = new char[len + 1];
        strcpy(str,String.str);
        //cout << " Operator = is overloaded"<<endl;
}

// returns value 1 if two string classes have same string

int
StringClass::operator==(const StringClass& String)
{
        if (strcmp(str,String.str) == 0)
          return 1;
        else return 0;
}

// Checks weather two strings are identical. If identical, returns 0, else
// returns 1.

int
StringClass::operator!=(const StringClass& string )
{
        if( strcmp(str, string.str) == 0)
          return 0;
        else
          return 1;
}
```

```
// Performs concatination of two strings and returns resultant string

StringClass&
StringClass::operator+=(const StringClass& String1)
{
       char temp[80];
       strcpy(temp,str);
       delete str;
       len = len + String1.len ;
       str = new char[len+1];
       strcpy(str,temp);
       strcat( str, String1.str);
       return (*this);
}

// String concatination of string belonging to two different classes and
// returns the resultant string

char*
operator+(const StringClass& String1, const StringClass& String2)
{
       StringClass tem;
       tem.len = String1.len + String2.len;
       delete tem.str;
       tem.str = new char[tem.len + 1];
       strcpy (tem.str,String1.str);
       strcat( tem.str, String2.str);
       return (tem.str);
}

// returns 1 if string in the given class is greater than string of other class

int
StringClass::operator>(const StringClass& String)
{
       if (strcmp (str,String.str) > 0)
         return 1;
       else return 0;

}

// returns 1 if string in the given class is less than string of other class

int
StringClass::operator<(const StringClass& String)
{
       if (strcmp (str,String.str) < 0)
         return 1;
       else return 0;
}

// Prints string of the given class
void
StringClass::print(void)
{
       cout << "\nlength\t"<< len<< "\tString\t"<< str<<endl;
}

char*
StringClass::operator&&(const StringClass& string)
{
       char *var;
       var = strtok(str, string.GetStr());
       return var;
}

void
StringClass::PutStr(char *tem)
{
```

```
        delete str;
        len = strlen(tem);
        str = new char[len+1];
        strcpy(str, tem);
        return;
}

char*
operator<<(const StringClass& string, int i)
{
        char temp1[80];
        memset(temp1, 0, 80);
        strcpy(temp1, string.str);
        char fin[80];
        memset(fin, 0, 80);
        int len = strlen(temp1);
        int j = 0;
        for(j = i; j < len; j++)
          fin[j-i] = temp1[j];
        return fin;
}

char*
operator>>(const StringClass& string, int i)
{
        char temp1[80];
        strcpy(temp1, string.str);
        int len = strlen(temp1);
        char fin[80];
        strncpy(fin, temp1, len - i);
        return fin;
}

////////////////////////////////////////////////////////////////////////
//
//                                Proc.C
//
////////////////////////////////////////////////////////////////////////

#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include<ctype.h>
#include"Proc.h"

////////////////////////////////////////////////////////////////////////
//      Member Function : StartProcess
//      Purpose         : This function is used to start the process. It
//                        takes the input from the user and performs the
//                        necessary operation.
////////////////////////////////////////////////////////////////////////

void
ProcessorClass::StartProcess()
{
        char purpose[80];
        status = FALSE;
        printf("\t -----------------------------\n");
        printf("\t |        Welcome to          |\n");
        printf("\t |     C++ Program Slicer     |\n");
        printf("\t |   type \"info\" to get       |\n");
        printf("\t |   Information about tool   |\n");
        printf("\t -----------------------------\n");
        printf("\n");
              //HelpMenu();
        do{
          printf("cppslicer>");
          gets(purpose);
          if((strncmp(purpose, "load", 4) == 0) || (strncmp(purpose, "LOAD", 4)
```

```
== 0))
            LoadProg(purpose);
        else if((strncmp(purpose, "cload", 5) == 0))
            CloadProgram(purpose);
        else if((strncmp(purpose, "slice", 5) == 0) ||
            (strncmp(purpose, "SLICE", 5) == 0))
            SliceProg(purpose);
        else if((strncmp(purpose, "save", 4) == 0) ||
            (strncmp(purpose, "save", 5)== 0))
            PrintProg(purpose);
        else if((strncmp(purpose, "cls", 3) == 0))
            system("tput clear");
        else if((strncmp(purpose, "VI", 2) == 0) ||
            (strncmp(purpose, "edit", 4) == 0))
            EditProgram(purpose);
        else if((strncmp(purpose, "quit", 4) == 0) ||
            (strncmp(purpose, "exit", 4) == 0) ||(strcmp(purpose, "q")==0))
            exit(0);
        else if((strncmp(purpose, "d", 1) == 0) ||
            (strncmp(purpose, "display", 7) == 0))
            DisplayProg();
        else if(strncmp(purpose, "type", 4) == 0)
            TypeProg();
        else if(strncmp(purpose, "man", 3) == 0)
            ManCommands(purpose);
        else if( strcmp(purpose, "help") == 0)
            HelpMenu();
        else if( strcmp(purpose, "info") == 0)
            InfoTool();
        else if((strncmp(purpose, "/", 1) == 0)||(strncmp(purpose, "!", 1)
==0))
            SystemCommand(purpose);
        else if((strncmp(purpose, "friends", 7) == 0) ||(strncmp(purpose,
"scan", 4) == 0)
            ||(strncmp(purpose, "inc", 3) == 0) || (strncmp(purpose,"show", 4)
== 0))
                system(purpose);
        else
        {
            printf("invalid command\n");
            Echo();
        }
      } while(1);
}

//////////////////////////////////////////////////////////////////////////
//  Member Function : LoadProg
//  Purpose         : This function is used to load the program into the
//                    cppslicer. It first locates the class of interest,
//                    finds its members, then performs the ref def
//                    operation.
//////////////////////////////////////////////////////////////////////////

void
ProcessorClass::LoadProg(char *Name)
{
      LoaderClass Loader;
      int state = 0;
      char first[20] , sec[20], third[20] ;
      memset(first, NULL, 20);
      memset(sec, NULL, 20);
      memset(third, NULL, 20);
      sscanf(Name, "%s%s%s", first, sec, third);
      if((strlen(sec) <=0) || (strlen(third) <= 0))
      {
         printf("Usage: load <filename> <class>\n");
         return;
      }
      state = TRUE;
```

```
        state = Loader.LoadSourceprogram(sec);
        if( state == FALSE)
        {
          printf("try again\n");
          return;
        }
        state = Loader.GetClassLineNumber(third);
        if( state == FAILURE)
          return;
        //Loader.PrintProgram();
  //printf("Class Line Number is %d\n", state);
        Loader.GetPublicAndPrivateParts();
        slicer.CopyLoader(Loader);
        //Loader.PrintSliceClass();
        state = slicer.ComputeRefDefForClasses(third);
        if( state == -1)
        {
          printf("try again\n");
          return;
        }
        //slicer.PrintFinal();
        status = TRUE;
}

////////////////////////////////////////////////////////////////////////
// Member Function : SliceProg
// Purpose         : This function is used to slice program resident in the
//                   cppslicer. Before slicing, the referenced and defined
//                   variables of each statement are identified. Based on them
//                   and active variables, slice mark for each statement is set
//                   to TRUE or FALSE. After setting the slice marks for the
//                   statements from begin to line of interest, the same
//                   statements are checked for control loops and member
//                   functions. Control statements are included if they contain
//                   statement(s) that are marked with slice marks.
////////////////////////////////////////////////////////////////////////

void
ProcessorClass::SliceProg(char *name)
{
        int retval = 0;
        if( status == FALSE)
        {
          printf("load program before this option\n");
          return;
        }
        int state = 0;
        char first[20], sec[20], Name[80];
        //StringClass sams;
        memset(first, NULL, 20);
        memset(sec, NULL, 20);
        memset(Name, NULL, 80);
        sscanf(name, "%s%s%s", first, sec, Name);
        if((strlen(sec) <=0) || (strlen(Name) <= 0))
        {
           printf("Usage: slice <line number> <variable>\n");
           return;
        }
        state = atoi(sec);
        //sams = Name;
        slicer.Set();
        slicer.MarkAllSlices();
        slicer.LoadActiveVars(Name);
        retval = slicer.BeginSlicingProgram(state, Name);
        if( retval == FAILURE)
        {
          printf("error in slicing : try again\n");
          return;
        }
```

```
        retval = slicer.SliceControlLoop(state);
        if( retval == FAILURE)
        {
          printf("error in slicing : try again\n");
          return;
        }
        slicer.PrintSlice(state);
        //slicer.PrintFinalSlice("out");
        name = NULL;
}

////////////////////////////////////////////////////////////////////////
//    Member Function : PrintProg
//    Purpose         : This function is used to save the resulting slice
//                      into a file provided by the user.
////////////////////////////////////////////////////////////////////////

void
ProcessorClass::PrintProg(char *name)
{
        if( status == FALSE)
        {
          printf("program is not loaded try after loading and sling the
program\n");
          return;
        }
        char first[20];
        char sec[20];
        memset(first, NULL, 20);
        memset(sec, NULL, 20);
        sscanf(name, "%s%s", first, sec);
        if(strlen(sec) <= 0)
        {
          printf("Usage:<save><filename>\n");
          return;
        }
        slicer.PrintFinalSlice(sec);
}

////////////////////////////////////////////////////////////////////////
// Member Function : Display Program
// Purpose         : This function is used to display the program onto
//                   the screen by calling cppslicer's memeber function.
////////////////////////////////////////////////////////////////////////

void
ProcessorClass::DisplayProg()
{
        if( status == FALSE)
        {
          printf("program is not loaded try after loading and sling the
program\n");
          return;
        }
        slicer.Display();
}

////////////////////////////////////////////////////////////////////////
//  Member Function : ManCommands
//  Purpose         : It displays the list of man commands and the usage
//                    of commands that are available in cppslicer program.
////////////////////////////////////////////////////////////////////////

void
ProcessorClass::ManCommands(char *name)
{
        char first[20];
        char sec[20];
        memset(first, NULL, 20);
```

```
        memset(sec, NULL, 20);
        sscanf(name, "%s%s", first, sec);
        if(strlen(sec) <= 0)
        {
          printf("Usage:<man><command>\n");
          return;
        }
        if((strcmp(sec, "load") == 0) || (strcmp(sec, "l") == 0))
        {
          printf("-------------------------------------------------\n");
          printf("load:\n");
          printf("Usage: load <filename><classname>\n");
          printf("loads the program into the slicer. If the arguments are\n");
          printf("not specified as requested, the program will not be
loaded.\n");
          printf("One needs to load a program before slicing the program.\n");
          printf("Process of loading also performs parsing of member\n");
          printf("functions of the classes if present. It also marks initial\n");
          printf("slice which will be the whole program\n");
          printf("-------------------------------------------------\n");
        }
        else if(strcmp(sec, "cload") == 0)
        {

printf("-----------------------------------------------------------------\n");
          printf("cload:\n");
          printf("-----\n");
          printf("Usage: cload <filename>\n");
          printf("cload is another way of loading the program into slicer. If
the\n");
          printf("loading program does not contain any classes or is a C program
then it can \n");
          printf("be loaded by using this command. In this way, it decreases many
searches\n");
          printf("for the variables of interest and setting slice marks for\n");
          printf("unnecessary statements thereby increasing running efficiency of
the tool\n");

printf("-----------------------------------------------------------------\n");
        }
        else if((strcmp(sec, "slice") == 0) || (strcmp(sec, "s") == 0))
        {
          printf("-------------------------------------------------\n");
          printf("slice:\n");
          printf("-----\n");
          printf("Usage: slice <linenumber><variable>\n");
          printf("Where linenumber is the line at which you want to slice\n");
          printf("the program and variable is the criterion variable with\n");
          printf("respect to which the program should be sliced.\n");
          printf("Right now the slicer can slice the programs involving
simple\n");
          printf("statements, control loops, classes, member functions. To
perform\n");
          printf("slice, one need to have at least one class and one member
function\n");
          printf(" when loaded with cload option.\n");
          printf("-------------------------------------------------\n");
        }
        else if(strcmp(sec, "type") == 0)
        {
          printf("-------------------------------------------------\n");
          printf("type:\n");
          printf("----\n");
          printf("Usage: type <noarguments>\n");
          printf("Prints or displays the program present in the slicer\n");
          printf("on screen with line numbers on the left side\n");
          printf("It displays the current page of the program and number of pages
it occupies\n");
          printf("-------------------------------------------------\n");
```

```
        }
        else if((strcmp(sec, "quit") == 0) || (strcmp(sec, "exit") == 0))
        {
          printf("-------------------------------------------------\n");
          printf("quit or exit or q\n");
          printf("-----------------\n");
          printf("Usage: q(uit) < noarguments>\n");
          printf("or exit <noargs>\n");
          printf("exits the tool and gets back to the unix prompt\n");
          printf("-------------------------------------------------\n");
        }
        else if((strcmp(sec, "edit") == 0)||(strcmp(sec, "VI") == 0))
        {
          printf("-------------------------------------------------\n");
          printf("edit or VI\n");
          printf("----------\n");
          printf("Usage: edit <filename> or VI <filename>\n");
          printf("Helps in editing the file in vi editor.\n");
          printf("-------------------------------------------------\n");
        }
        else if(strcmp(sec, "save") == 0)
        {
          printf("-------------------------------------------------\n");
          printf("save\n");
          printf("-----\n");
          printf("Usage: save <filename>\n");
          printf("Save command basically helps to save the resulting slice\n");
          printf("into the specified filename. If this command is called before
doing\n");
          printf("the slice, save will include whole program which is called as
largest\n");
          printf("possible slice for a program\n");
          printf("-------------------------------------------------\n");
        }
        else if( strcmp(sec, "man") == 0)
        {
          printf("-------------------------------------------------\n");
          printf("man\n");
          printf("---\n");
          printf("Usage: man <command>\n");
          printf("Man displays information about the command mentioned at\n");
          printf("the command prompt. It takes only one command at a time.\n");
          printf("list of commands that are available are:\n");
          printf(" 1. cload     2. edit     3. help\n");
          printf(" 4. load      5. man      6. quit\n");
          printf(" 7. save      8. slice    9. type\n");
          printf("10. ! or /\n");
          printf("-------------------------------------------------\n");
        }
        else if(strcmp(sec, "help") == 0)
        {
          printf("-------------------------------------------------\n");
          printf("help\n");
          printf("----\n");
          printf("Usage: help \n");
          printf("Gives information about list of commands and their purpose\n");
          printf("help menu looks as\n");
          HelpMenu();
          printf("-------------------------------------------------\n");
        }
        else if( strcmp(sec, "echo") == 0)
        {
          printf("-------------------------------------------------\n");
          printf("echo\n");
          printf("----\n");
          printf("echos list of commands that are available in the tool\n");
          printf("this command is invoked when invalid command is typed at
command line\n");
          printf("-------------------------------------------------\n");
```

```
        }
        else if((strcmp(sec, "!") == 0) ||(strcmp(sec, "/") ==0))
        {
          printf("-------------------------------------------------\n");
          printf("! or /\n");
          printf("------\n");
          printf("Usage: ! <system command> or / <system command>\n");
          printf("Used to invoke any system command. Any system command can
be\n");
          printf("invoked by typing ! or / before the command\n");
          printf("-------------------------------------------------\n");
        }
        else
        {
          printf("No manual entry for %s\n", sec);
        }
}

//////////////////////////////////////////////////////////////////////
//      Member Function : TypeProg
//      Purpose         : It displays the original program with line
//                        numbers on left side. It displays page by page.
//////////////////////////////////////////////////////////////////////

void
ProcessorClass::TypeProg()
{
        if( status == FALSE)
        {
          printf("load program before this option\n");
          return;
        }
        slicer.PrintProgram();
}

//////////////////////////////////////////////////////////////////////
//    Member Function : HelpMenu
//    Purpose         : Displays the help menu on to screen when help is
//                      typed at the command prompt. It displays the commands,
//                      their usage, and purpose.
//////////////////////////////////////////////////////////////////////

void
ProcessorClass::HelpMenu()
{
printf("*************************************************************************
***\n");
printf("*  Command    Usage                          Purpose
*\n");
printf("*************************************************************************
***\n");
printf("*  cload    cload <filename>            Used to load programs written in
*\n");
printf("*                                       C or in C++ without classes
*\n");
printf("*  edit     edit <filename>             Edit a program in slicer
*\n");
printf("*  help     help                        Brings a menu giving
*\n");
printf("*  load     load <filename><classname>  Used to load program into slicer
*\n");
printf("*  man      man <command>               Gives help on usage of command
*\n");
printf("*  q(uit)   q or quit or exit           Used to exit from the tool
*\n");
printf("*  save     save <filename>             Used to save output of
*\n");
printf("*                                       the slice into the file
*\n");
```

```
printf("*  slice    slice<line no><var>         Used to slice a program
*\n");
printf("*                                       residing in slicer
*\n");
printf("*  type     type                        Displays program resident in
*\n");
printf("*                                       the cppslicer.
*\n");
printf("*  ! or /   ! <systemcommand>           Used to invoke system commands
*\n");
printf("*************************************************************************
***\n");
}

//////////////////////////////////////////////////////////////////////
//     Member Function : Echo
//     Purpose         : It echoes the list of commands whenever user
//                       issues a wrong command.
//////////////////////////////////////////////////////////////////////

void
ProcessorClass::Echo()
{
       printf("valid commands are :\n");
       printf("cload <filename>\n");
       printf("edit <filename>\n");
       printf("help\n");
       printf("load <filename><classname>\n");
       printf("man <command>\n");
       printf("q or quit or exit\n");
       printf("save <filename>\n");
       printf("slice <lineno><variable>\n");
       printf("type\n");
       printf("for more information on each command type man <command>\n");
}

//////////////////////////////////////////////////////////////////////
// Member Function : CloadProgram
// Purpose         : This function is used to load programs written in C
//                   or in C++ without classes.
//////////////////////////////////////////////////////////////////////

void
ProcessorClass::CloadProgram(char *Name)
{
       LoaderClass Loader;
       int state = 0;
       char first[20] , sec[20], third[20];
       memset(first, NULL, 20);
       memset(sec, NULL, 20);
       sscanf(Name, "%s%s", first, sec, third);
       if((strlen(sec) <=0) && (strlen(third) == 0))
       {
          printf("Usage: load <filename> <class>\n");
          return;
       }
       state = TRUE;
       state = Loader.LoadSourceprogram(sec);
       if( state == FALSE)
       {
         printf("try again\n");
         return;
       }
       Loader.LoadMainParts();
       slicer.CopyLoader(Loader);
       state = slicer.ComputeRefDefForMain();
       if( state == FAILURE)
       {
         printf("error in loading the program\n");
```

```
        printf("try again\n");
        return;
      }
      status = TRUE;
}

//////////////////////////////////////////////////////////////////////
// Member Function : InfoTool
// Purpose         : It displays the information regarding the tool when
//                   user issues a info command.
//////////////////////////////////////////////////////////////////////

void
ProcessorClass::InfoTool()
{
printf("-------------------------------------------------------------------\n");
printf("Program slicing is the process of producing slices of the existing
program \n");
printf("depending upon the variable of interest and line number. The algorithms
used in \n");
printf("producing the slices are adopted from Samadzadeh, Korel and Laski's
algorithms. \n");
printf("The cppslicer uses both static and dynamic slicing methods.\n");
printf("To produce a slice of the program, one needs to have a program which
is\n");
printf("executable. Depending on the variable of interest, the size of slice
\n");
printf("will vary. But there will be at least one slice for a program, the
program,\n");
printf("itself. ");
printf("For example consider the following example which reads ");
printf("input for two \nvariables x, y.\n");
printf("1  #include <iostream.h> \n");
printf("2 \n");
printf("3  main() \n");
printf("4  { \n");
printf("5       int x,y;\n");
printf("6       cout <<\"enter number \"; \n");
printf("7       cin>>x; \n");
printf("8       cout<<\"enter number \";\n");
printf("9       cin>>y;\n");
printf("10      cout<<\"x = \"<<x<<\" y = \"<<y<<endl; \n");
printf("11      cout<<\"x = \"<<x<<\" y = \"<<y<<endl; \n");
printf("12      return 0; \n");
printf("-------------------------------------------------------------------\n");
printf("press enter to continue ...");
getchar();
printf("\n");
printf("13  } \n");
printf("\n");
printf("Slice of the program with respect to variable y at line 11 is \n");
printf("\n");
printf("#include <iostream.h> \n");
printf("\n");
printf("main() \n");
printf("{ \n");
printf("     int x,y; \n");
printf("     cin>>y; \n");
printf("}\n");
printf("\n");
printf("The slice is executable by itself. The tool, cppslicer, can handle
programs\n");
printf("written in C/C++ either with or without classes, simple pointers \n");
printf("to int, char, class (*this). It can also handle operator
overloading.\n");
printf("This tool cannot handle functional overloading, inheritance, \n");
printf("friend functions, inline functions, main with arguments, structures,
unions.\n");
printf("\n");
```

```
printf("-----------------------------------------------------------------\n");
printf("press enter to continue ...");
getchar();
printf("\n");
printf("-----------------------------------------------------------------\n");
printf("To load an ordinary C program or programs that do not contain any
classes, type\n");
printf("\t\t\"cload <filename>\"\nat the prompt. To generate slice,
type\n\t\t\"slice <linenumber><variable>\" \nat the prompt.");
printf("Upon producing the slice, slicer displays only the part of the\ncode
which is ");
printf("of more important. To view the whole code type \"type\" at the\ncommand
prompt. To");
printf(" save it to a file type \n\t\t\"save <filename>\".\n");
printf("While loading the programs that involve classes, the format of the class
should\n");
printf("be private parts followed by public parts. It is not a restriction, but
the \n");
printf("tool works well (perfectly) under the assumed conditions. To load the
program ");
printf("\ntype \n\t\t\"load <filename><classname>\"\nat the commnand line. Once
the loading part is done, slices for the\n");
printf("program can be obtained by typing \n\t\t\"slice
<linenumber><variable>\"\nIt should be ");
printf("noted that the variables can be more than one, but the
linenumber\nshould be ");
printf("unique. \n");
printf("list of commands that are available in this tool are:\n");
printf("1. cload     2. edit     3. help\n");
printf("4. load      5. man      6. quit\n");
printf("7. save      8. slice    9. type\n");
printf("10. ! or /\n");
printf("To know more about each command type \"man <command>\"\n");
printf("If you find any information is lacking please e-mail to me at \n");
printf("ramakar@a.cs.okstate.edu\n");
printf("-----------------------------------------------------------------\n");
}

////////////////////////////////////////////////////////////////////////
// Member Function : EditProgram
// Purpose         : A user can edit his/her program while in cppslicer.
//                   To edit, one needs to type EDIT or VI and filename.
////////////////////////////////////////////////////////////////////////

void
ProcessorClass::EditProgram(char *Name)
{

       char first[20] , sec[20], third[40];
       memset(first, NULL, 20);
       memset(sec, NULL, 20);
       memset(third, NULL, 40);
       sscanf(Name, "%s%s", first, sec);
       if(strlen(sec) <=0)
       {
          printf("Usage: edit <filename> or VI <filename>\n");
          return;
       }
       strcpy(third, "vi ");
       strcat(third, sec);
       system(third);
}

////////////////////////////////////////////////////////////////////////
//   Member Function : SystemCommand
//   Purpose         : This function is to invoke the system commands on
//                     shell
////////////////////////////////////////////////////////////////////////
```

```
void
ProcessorClass::SystemCommand(char *purpose)
{
       if((purpose[0] == '/')||(purpose[0] == '!'))
       {
         char command[80];
         memset(command, NULL, 80);
         for(int i=1; i <= strlen(purpose); i++)
              command[i-1] = purpose[i];
         system(command);
       }
}

//////////////////////////// End of Proc.C ////////////////////////////

//////////////////////////////////////////////////////////////////////
//
//                                    Main.C
//
//////////////////////////////////////////////////////////////////////

#include<string.h>
#include<stdio.h>
#include"Proc.h"

main()
{
       ProcessorClass P1;
       P1.StartProcess();
}
```

```
# Makefile to compile all the files together.

OBJS     =    Main.o String.o Loader.o FlowControl.o Slice.o Proc.o
INCLUDE  =    string.h Loader.h defs.h Proc.h

CFLAGS   =    -c -g
LFLAGS   =    -g -o proj

proj:         $(OBJS)
              CC $(OBJS) $(LFLAGS)

Main.o:       Main.C $(INCLUDE)
              CC $(CFLAGS) Main.C

String.o:     String.C $(INCLUDE)
              CC $(CFLAGS) String.C

Loader.o:     Loader.C $(INCLUDE)
              CC $(CFLAGS) Loader.C

Slice.o:      Slice.C $(INCLUDE)
              CC $(CFLAGS) Slice.C

FlowControl.o:FlowControl.C $(INCLUDE)
              CC $(CFLAGS) FlowControl.C

Proc.o:       Proc.C $(INCLUDE)
              CC $(CFLAGS) Proc.C
clean:
              rm *.o \
              proj core
```

Vita

Rajeshwar Ramaka

Candidate for the Degree of

Master of Science

Thesis: TOWARDS AN INTERACTIVE DEBUGGING TOOL FOR C++ BASED ON PROGRAM SLICING

Major Field: Computer Science

Biographical:

Personal Data: Born in Karimnagar, Andhra Pradesh, India, February 22, 1971, the son of Mr. Dattaiah Ramaka. and Visheshwari Ramaka.

Education: Graduated from Loyola Academy, Hyderabad, India, in March 1987; received Bachelor of Technology in Mechanical Engineering from Jawaharlal Nehru Technological University, India, in July 1993; completed the requirements for the Master of Science Degree in Computer Science at the Computer Science department at Oklahoma State University, in December 1995.

Professional Experience: Para Professional Client Services, Computing and Information Services, Oklahoma State University, August 1994 to June 1995.