INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

FUNCTION CROSS-REFERENCE BROWSER

ZHONGDE YU

A THESIS
IN
THE DEPARTMENT
OF
COMPUTER SCIENCE

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE
CONCORDIA UNIVERSITY
MONTRÉAL, QUÉBEC, CANADA

APRIL 2002
© ZHONGDE YU, 2002

National Library of Canada
Acquisitions and Bibliographic Services
395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada
Acquisitions et services bibliographiques
395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-68491-1

Canada

Abstract

Function Cross-Reference Browser

Zhongde Yu

In this thesis, we describe the construction of a Function Cross-Reference Browser, a tool of software visualization. The Browser is able to parse all the source files of a software application written in ANSI C++, to extract the information on the function cross reference hierarchy, and to provide 3D display of the call hierarchy. The Browser can handle a proper subset of the standard C++ features (including dynamic binding), and it provides a convenient way to traverse the function call tree in a graphic window, as well as other features for clear displaying. This thesis demonstrates the design and implementation of the Browser, provides detailed analysis of its execution behavior, discusses the limitations of it, and outlines some possible enhancements in the future. In summary, the work done by this thesis adds a new member to the area of software visualization. The Browser is a useful tool for software engineers. It can help them better understand a large, complicated software application written in C++ language.

Acknowledgments

First of all, I would like to take this opportunity to thank Dr. Peter Grogono, my academic advisor, for his continuous encouragement and guidance during the whole process of my study as a Master student in the Computer Science Department, Concordia University. Without his help and support, this thesis would have been impossible.

Secondly, I also would like to thank other persons from the Computer Science Department, such as Ms. Halina Monkiewicz, for their course teaching and administration help.

Finally, I must thank my brothers and sisters for their spiritual support and great love during my study.

Contents

List of Tables

List of Figures

1 Introduction

1.1 Overview of Software Visualization

1.1.1 General Visualization

ization is a methodology of using images to convey meaningful information. Its advocates point to the following important aspects. The imagery plays an important role in human being's communication in general. The human visual system has an extraordinarily high bandwidth, hence human beings can track and detect visual patterns with very high speed. Human beings have the power of abstraction, which is inherent in pictorial representation.

Visualization can be applied in a number of fields such as science, information, geography, business, statistics, process, and software. All types of visualization share a common goal: to transforming information into a meaningful, useful representation from which a human observer can gain understanding. And all types of visualization share common foundations: all have a need to combine the graphic elements (a grammar) into meaningful visualizations (sentences, paragraphs). Software visualization is a new member of the general visualization family.

1.1.2 Software Visualization and its Categories

Software visualization (SV) refers to the visualization of computer programs and algorithms. Contrary to visual programming which is concerned with graphical specification of computer programs, SV deals with graphical presentation, monitoring and exploration of programs expressed in textual form. It can help people to better

understand algorithms, to identify bugs, to generally enhance learning and speed up development. The software visualization is believed as the art of mapping programs into pictures [1, 2].

There are six distinct, top-level categories of software visualization, together with many sub-categories under each one [1, 2]:

(1) **Scope**: What range of programs that the SV system may take as input for visualization?

The issues are Generality (Hardware, Operating System, Language, Application) and Scalability (Program, Data Set).

(2) **Content**: What subset of information about the software is visualized by the SV system?

The issues are Program (Code, Data), Algorithm (Instruction, Data), Fidelity and Completeness (Invasiveness), and Data Gathering Time (Temporal Control Mapping, Visualization Generation Time).

(3) **Form**: What are the characteristics of the output of the system (the visualization)?

The issues are Medium, Presentation Style (Graphical Vocabulary, Animation, Sound), Granularity (Elision), Multiple Views, and Program Synchronization.

(4) **Method**: How is the visualization specified?

The issues are Visualization Specification Style (Intelligence, Tailorability) and Connection Technique (Code Ignorance Allowance, System-Code Coupling).

(5) **Interaction**: How does the user of the SV system interact with and control it?

The issues are Style, Navigation (Elision Control, Temporal Control – Direction

and Speed), and Scripting Facilities.

(6) **Effectiveness**: How well does the system communicate information to the user? The issues are Purpose, Appropriateness & Clarity, Empirical Evaluation, and Production Use.

Note that the above taxonomy of software visualization covers a pretty wide range, and in practice a SV system has only part of the features mentioned above.

1.1.3 Why Software Visualization is Chosen for this Thesis

The management of source code is one of the greatest challenges facing programmers today. As programs become longer and more complex, the need to organize and manage source code increases. So we need a better way to manage the source code. Software Visualization (SV) is one of the choices.

1.2 Contribution of Thesis

1.2.1 Purpose

This thesis describes the design and implementation of a **Function Cross-Reference Browser**: specifically, it will build a tree of function calls in a software application which is written in C++, and to visualize its **function cross-reference hierarchy**, in order to help understanding the software program architecture and the working principles.

1.2.2 Description

Here are some details of this Function Cross-Reference Browser:

(A) Execution of the Browser and Graphical User Interface

After executing the Browser program, there are three windows appearing on the monitor screen (see Figure 1.1):

a) The Display Window 1: the larger graphical window in Figure 1, for showing function cross-reference relationship (i.e. the function call tree);

b) The Display Window 2: the smallller graphical window in Figure 1, for showing virtual functions;

c) The DOS Window: showing Help info.

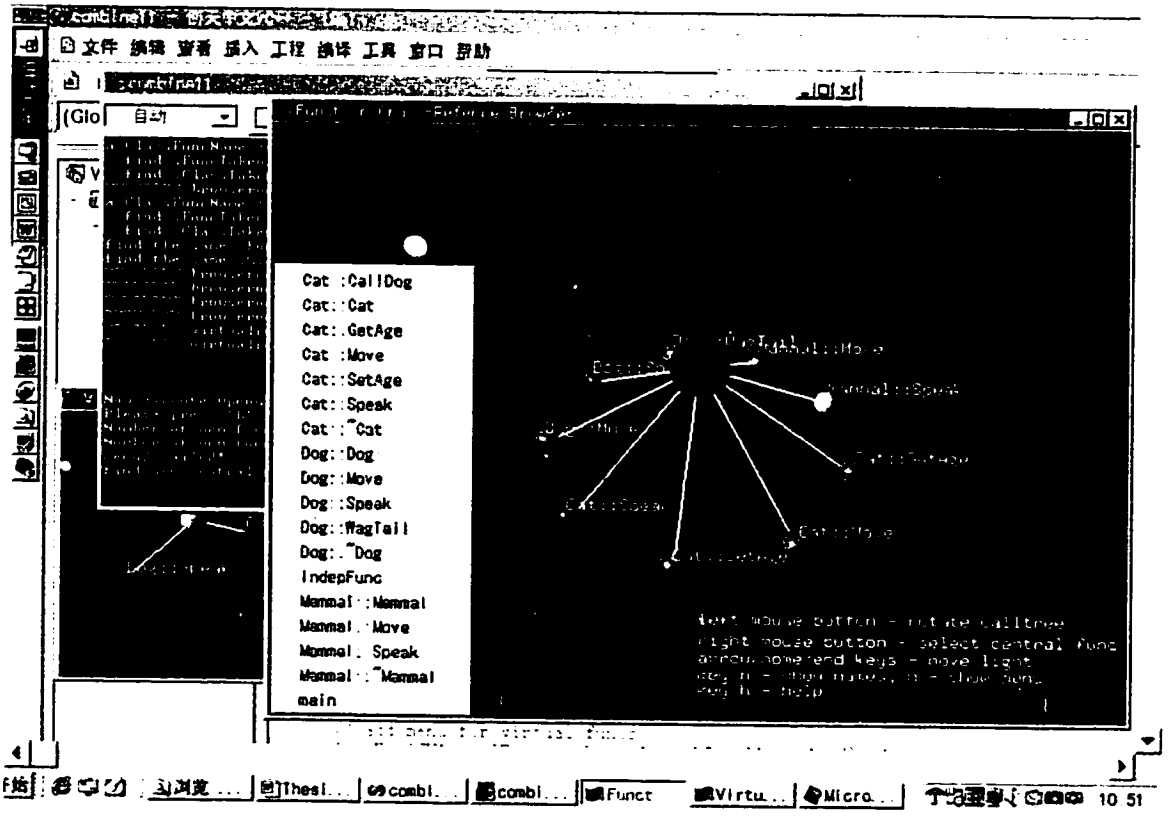


Figure 1.1 A screen shot after executing the Browser program

(B) **The function call tree window (Display Window 1)**

Generally speaking, there are three layers in this window to display the function cross-reference relationship. Each function is represented by a sphere node, and all the function names are displayed besides the function nodes (see Figure 1.2).

Layer 1: the **central function**, i.e. function selected by the user;

Layer 0: the **calling functions** (if available), i.e. functions that call the central function;

Layer 2: the **called functions** (if available), i.e. functions that are called by the central function.

All the function names are displayed besides the function nodes.

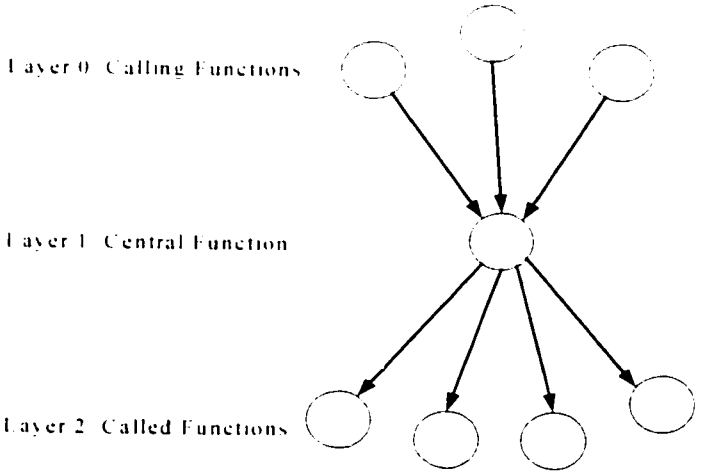


Figure 1.2 The three layers shown in the Browser program

The default status (just after system initialization) is shown in Figure 1.1:

Layer 1 contains the main() function;

Layer 0 contains nothing;

Layer 2 contains functions (if available) being called by main().

(C) Display of a list of function names

By clicking the right-hand side mouse key inside Display Window No.1, a menu is popped up showing a list of all the function names. Here assume all the source files are in the same directory).

(D) Selection of a new central function

From (B), if clicking any one of the function names, that function will be at layer 1 in the next display, so it is possible to traverse the whole function call tree (although generally the whole tree may not be displayed on the screen.)

(E) Rotation of the function call tree

The function call tree can be rotated by using the mouse in order to show those nodes which are overlapped by the nodes close to the user, when more nodes exist on a layer and some of them are overlapped by others.

(F) The Virtual function window (Display Window 2)

Similarly, this window has the same features as the function call tree window (the Display Window 1), except that there are only Layer 1 and Layer 2. The Layer 1 (central function) is always for a virtual function in base class and the Layer 2 is always for the virtual functions in derived classes. This window shows the information of possibility of dynamic binding.

In addition, by using the six special keys on the keyboard (e.g. the four arrow keys (left, right, up, down), Home and End), we can move the light source along six directions in a 3D space in order to better view the function call trees.

1.2.3 Classification

According to the taxonomy of software visualization [1] [2], this tool can be categorized as the followings:

1) **Scope**: it is a general tool to show the function call hierarchy in an application; however, it is not restricted to any particular application;

2) **Content**: it belongs to **code visualization** under program visualization;

3) **Form**: the medium is a PC with graphic display, and the presentation style is of color and of three dimensions (3D);

4) **Method**: not applicable, since no run-time, dynamic behavior will be concerned;

5) **Interaction**: the style is to use mouse and drop-down menus (as those in windows programming), and the navigation uses elision control to hide unnecessary information, i.e. only 3 layers are displayed on screen at a time;

6) **Effectiveness**: the purpose is well specified above, as helping people to better understand an application written in C++.

1.2.4 Anticipated Enhancement

This thesis can be thought (in the following aspects) as an enhancement to the System Visualizer developed by Kim Thang Vu [3]. Vu's thesis describes the development of a software visualization tool that is called System Visualizer. This System Visualizer can extract the module structure (i.e. the "module include" relationship) of software application written in C or C++, and then display the module structure as a three dimensional cone tree on the user interface (GUI). In the GUI window, up to three

levels of tree structure can be displayed by selecting a number from the menu "Levels", and different views can be displayed by selecting a choice from the menu "Views". In addition, the traversal to the whole cone tree is possible by selecting a module name from the scrolled list.

Vu's System Visualizer was developed by using PEXlib, a 3D extension to the X Window system in Unix operating system. A complicated algorithm is required for the 3D display, and the switching among different views is in a discrete (stepwise) pattern instead of a continuous one. Vu's thesis contains two main parts, the Parser and the Displayer. The Parser is relatively simple, while the Displayer is pretty complicated in algorithm during implementation.

Vu's thesis is an attempt to build a tool for 3D visualization of software; there are different possibilities to enhance his tool, and this thesis is one of the enhancements. In this thesis, the visualization is changed from displaying "module include" relationship to displaying "function cross-reference" relationship. According to my personal working experience in software industry, this type of visualization tool could be useful to the real work of software development, as the developers always need to know the function call relationships when they want to understand a large, complicated software application, which contains many modules and complex function cross-reference hierarchies.

Besides the goal of visualization has been different, there are other aspects that are different from Vu's System Visualizer. The platform is Microsoft Windows system (i.e. Windows 95/98/2000/NT) instead of Unix system. The Displayer is relatively

simple, since it was implemented by using the powerful OpenGL library. However, the Parser is rather complicated due to the intrinsic complexity of the function cross-reference hierarchy.

For the 3D display, the view of the function call tree can be continuously changed by dragging the mouse in the display window, in order to view the tree from different angles, to see the hidden nodes; this rotation is continuous, instead of the step-mode rotation, providing a finer tuning to the display. The light source can be moved by using the keyboard, since fine-tuning is not necessary.

The traversal to the call tree can be achieved by selecting a function name from a pop-up menu (by clicking mouse right key inside the Display window). For any selected function, the Display window only shows three levels, which are part of the call tree. In the Display window, there are also function names as well as a help menu, and all of them can be hidden by using designated toggle keys. There is also another Display window for showing virtual functions, so that the issue of dynamic binding can be addressed by this Browser. Note that the colors of function nodes are different, making it easy to identify different types of functions, such as called/calling functions, base/derived virtual functions.

The Browser is implemented as a Win32 console application. Both the Parser and Displayer parts are written in C++ (especially Standard Template Library, STL), Win32 API, and OpenGL 3D graphics. This makes the Browser to be used in the more popular world of PC Windows.

1.3 Organization of Thesis

This thesis has the following parts, and (from the second part) will be developed in detail in the rest of the thesis:

- Chapter 1. Introduction: This chapter first provides an overview of Software Visualization (SV), then lists the contribution of this thesis, and finally gives an outline of the thesis structure.
- Chapter 2. Related Work: This chapter introduces the research work which are related to this thesis.
- Chapter 3. Design: This chapter discusses the design for the Function Cross-Reference Browser. Basically the Browser consists of two main parts, one is the Analyzer (including Scanner and Parser, and other service functions), which is used to parse C++ source files and to construct function call hierarchy; another is the Manipulator (i.e. Displayer), which is used to display the function call hierarchy in 3D graphics.
- Chapter 4. Implementation: This chapter discusses the implementation details of the Browser. It covers the two main parts of the Browser, respectively. First, it gives brief introductions to Standard Template Library (STL) and OpenGL. Then it provides the details of various classes (such as tokens, scanner, and parser) to get primitive tokens, as well as service functions to retrieve function names and to construct function call hierarchy. Finally, this chapter provides the method of how to display the function call hierarchy in 3D graphics.
- Chapter 5. Result and Discussion: This chapter presents the accomplishment of the

design goal, the result of operation, and discusses the C++ features which the Browser can handle as well as others it cannot.

- Chapter 6, Further Development and Conclusion: This chapter proposes some possibilities to further enhance this Browser, and gives the conclusion of this thesis.
- Chapter 7, References: This chapter lists some important literatures referenced by this thesis.
- Appendix A: This Appendix lists the source code of the testing C++ application used by this thesis.
- Appendix B: This Appendix lists the textual output of the Browser after running on the testing code listed in Appendix A.

2 Related Work

The importance of visual representations in understanding computer programs is not new. Software visualization is an approach to enhance software program representation, presentation and appearance. The SV history can be sketched in terms of several major threads of activities. (All the following research on SV are from reference [2].)

1) Presentation of source code

An early attempt to improve program appearance was the development of a "presentation", or "reference" form of the programming language ALGOL60. Another idea was pretty printing, which used spacing, indentation, and layout to make source code easier to read in a structured language such as LIST or Pascal. More recent developments have used computerized typesetting and laser printing to improve the presentation of source code. An ambitious recent attempt to enhance the presentation of source code is the work of Baecker and Marcus. Their SEE Program Visualizer automatically typesets a C program according to an elaborate style guide based on graphic design principles. Knuth's WEB system also seeks to enhance program publishing, combining program source text and documentation in a single publication using a sophisticated markup language. Today, almost all the software development tools (e.g. Microsoft Visual C++) can present formatted textual styles for reserved words, strings and comments in different colors, making it easier for developers to write and recognize the contents of source code, and enhance development efficiency.

2) Representations of control flow and data structures

The role of visual representations in understanding computer programs has a long history, beginning with flowcharts demonstrated by Goldstein and von Neumann. Hait developed a system that could draw them automatically from Fortran or assembly language programs: Knuth produced a similar system which integrated documentation with the source code and could automatically generate flowcharts. Baecker's prototype interactive debugger for the TX-2 computer produced static images of high-level language data structures and of the computer graphics display file. Myers's Incense system was a more ambitious system for the display of data structures. More recently, there has been an explosion of interest in visual programming, the use of visual representations of programs as both an input and an output modality.

3) Animation of program behavior

Licklider did early experiments on the use of computer graphics to view how the contents of the memory of a computer were changing as the computer was executing. A different approach was taken with Knowlton's influential files, which demonstrated L[6], Bell Lab's low-level list processing language. This work was the first to use animation techniques to portray program behavior and the first to address the visualization of dynamically changing data structures. Other people continued this work in a pedagogical directions. Baecker outlined the potential of program animation and sketched many of the key research issues. Hopgood produced a series of short films illustrating hash coding and syntax analysis techniques. Yarwood explored the concepts of program illustration, and methods of embedding graphical representations of program state within program source text. Booth produced a short film animating

PQ-tree data structure algorithms. Baecker also reported on work in which he and his students were investigating the portrayal of data structure abstractions and algorithms. eventually leading to the important film *Sorting Out Sorting*.

4) Systems for software visualization

The availability in the 1980's of personal workstations with bit-mapped displays and graphical user interfaces allowed researchers to go beyond the prototypes and specific animations of the 70s and develop software visualization systems. One of the earliest attempts to a debugging system to aid visualization was the work done in Lisp by Lieberman. The most important and well-known system of software visualization was BALSA, followed by BALSA-II, which allowed students to interact with high level dynamic visualizations of Pascal programs. It was used as a tool in algorithm design and analysis. Literature [1, 2] gives more examples of software visualization.

A directly related work on code visualization has been done by Kim Thang Vu [3]. Vu developed a software visualization tool, which is called System Visualizer. It can extract the module structure (i.e. the "module include" relationship) of software application written in C or C++, and then display the module structure as a three dimensional cone tree on the user interface. Vu's System Visualizer is an attempt to build a tool of code visualization for 3D visualization of software.

3 Design

The principal topic of this thesis is source code management. It describes a tool that has two components: the Analyzer and the Manipulator (i.e. the Displayer). In general, source code management involves two operations: analysis and manipulation. The former is parsing the source code and building up function call tree, and the latter is displaying the call tree in graphic windows.

The analysis of the source code can yield useful information about the program that may not be readily apparent or easily obtained. For example, in a large program, it may not be easy to see, for a specific function, which functions are being called by it, and which functions call it. *A source-code analysis utility (Analyzer, which contains Scanner and Parser)* can examine the files of source code and extract the function call hierarchy information.

The manipulation of the source code is to manipulate the code to conform to some standardized style, or to display the information in a graphic format. *A source-code manipulation program (Manipulator, here is Displayer)* can display the above-mentioned function calling hierarchy information.

3.1 Analyzer --- Scanner and Parser

3.1.1 Retrieval of Primitive Tokens

The Parser shares many common tasks with compiler front-ends: both receive code as input, break the code down into tokens, and then output these tokens in a new format. The compiler's output is machine code, while here the Parser's output is just the type

of each token. The Parser is like a simplified version of compilers in some ways.

The most basic element of any programming language is called a **token**. Tokens are the smallest recognizable elements of a program: they comprise the building blocks for creating meaningful statements. Compilers generally do not work with anything smaller than tokens. Comments, constants, identifiers, numbers, punctuation, and string literals are all examples of tokens. It is the job of the **Scanner** to read the code, break it down into these elements, and return them back to the Parser. The Scanner also identifies the type of token to return. The **Parser** requests successive tokens from the Scanner and takes appropriate action before requesting the next token.

After the type of each token can be identified, there are still several tasks to be done: to retrieve function names and to build up function call hierarchy.

3.1.2 Retrieval of Function Name

First of all, we need to find the names of **classes** and their **member functions**, based on the primitive tokens. This is the foundation of building up function cross-reference hierarchy. Two global lists are maintained for the class names and member function names respectively. In the meanwhile, we also need to find the names of **independent functions**, which are not members of any class. Again, a global list is maintained for the independent function names. Here we can call these two types of functions together as the **central functions.**

Then, we need to find the names of all the **objects** which are declared in the source files. Here we need the knowledge of class names, which have been found above.

Similarly, a global list is maintained for the object names.

Next, we need to find the names of functions which are called by each of the central functions (both the class member functions and independent functions). These functions are the **called functions**. This is the crucial step in building up function call hierarchy. During the construction of this call hierarchy, for each central function, a temporary global list is maintained for the called functions. The function call hierarchy is constructed as a binary search tree (**BST**), in which each node corresponds to a central function and the node has an attribute to store its called function list.

The next step is to find the names of functions which call each of the central functions. These functions are the **calling functions**. However, we do not need to scan the source files again, because we already have the information of the calling functions for each central function in the BST, after the called functions of this central function are found and inserted into BST. What we need to do here is to traverse the BST, extract calling function information, and maintain a temporal global list of calling function names for each central function.

In addition, there is an important issue we need to resolve, namely virtual functions. This is a special but difficult issue when dynamic binding appears. Dynamic binding is a runtime behavior. Generally speaking, when virtual function is not called directly, it is impossible to know exactly which version of a virtual function will be called at compile time, either the version in base class or the version in derived classes. The solution provided by this thesis is to find all the possible versions of a virtual function.

and maintain another BST for the hierarchy relationship between base virtual functions and derived virtual functions. In summary, the followings are the steps to retrieve function names:

- Retrieval of Class Names and Member Function Names
- Retrieval of Independent Function Names
- Retrieval of Object Names
- Retrieval of Called Function Names
- Retrieval of Calling Function Names
- Retrieval of Virtual Function Names

3.1.3 Construction of Function Call Hierarchy:

Here a Binary Search Tree (**BST**) is declared and used to store the function cross-reference hierarchy. Each node in this BST corresponds to a central function, i.e. the node contains a key element to store the name of a central function; in the meanwhile, the node also contains two sets of functions names, one for the called functions of, the other for the calling functions of, the central function. In addition, the node has a flag showing the type of this central function, i.e. non-virtual, base virtual (for virtual function in base classes) or derived virtual (for virtual function in derived classes), as well as two pointers pointing to the left and right nodes of the central function.

As described above, two BSTs are maintained, one for the Function Call Hierarchy, another for Virtual Functions. The construction of these two BSTs are carried out

during the processes of function name retrievals.

3.2 Manipulator --- Displayer

3.2.1 Displaying of Function Call Hierarchy

After the function call trees (including the one for virtual functions), an OpenGL program is called to display two graphic windows, and one of them is for function call hierarchy.

1) **Function nodes, names, and colors**

As mentioned in the Introduction, generally there are three layers in the function call hierarchy window (although not always true), i.e. a central node at Layer 1 for a central function, some nodes at Layer 2 for the called function (if they exist), and some other nodes at Layer 0 for the calling functions (if they exist). Note that the nodes at Layer 2 are evenly positioned, regardless of the number of nodes; and so are the nodes at Layer 0.

The function names for all the nodes can be toggled to be displayed or to be hidden in the display window by a specific keyboard stroke; also, a help menu can be toggled to be shown or to be removed. In addition, a more detailed help menu can be shown in the DOS window.

There are also connection lines which connect each nodes at Layer 2 and Layer 0 to the central node, in order to show the function call relationships between Layer 1 and Layer 2 as well as between Layer 0 and Layer 1.

If there is no node at any layer (except Layer 1 at which there is always one and only

one node at all time), no connection line will be shown, and a dummy name such as "No Calling Func" or "No Called Func" will be shown at the Layer 0 or at Layer 2.

For clarity of visual displaying, the nodes at the three layers are in different colors, i.e. red for central node at layer 1, green for called functions at layer 2, and cyan for calling functions at layer 0. However, base virtual functions are always in yellow, and derived virtual functions are always in pink, no matter at which level they are currently displayed.

If any node is hidden by others, we can use the left mouse key to rotate the tree so that the hidden node can be moved to a position where it can be seen clearly.

Also displayed in this window is a light source, which can help to create a real 3D illusion of the function nodes. The position of the light source can be adjusted by using six special keys on the keyboard: the Left/Right/Up/Down arrow keys for moving the light source along the left/right/up/down direction, and the Home/End keys for moving the light source towards/backwards the user.

2) **Tree traversal**

By clicking the right mouse key, a list of all the function names can be shown, and any one of them can be selected as a new central function: if a new central function is selected, the display on the function call graphic window is updated, so the selected function as the central node at Layer 1, and its called and calling functions are displayed at Layer 2 and Layer 0 (if they exist), respectively. This is the way of traversing the whole function call tree, in order to view the complete call hierarchy.

3.2.2 Displaying of Virtual Functions

Another graphic window is for virtual functions. It has almost all the features of the call tree window, such as function nodes, their positions and rotation, their names and colors, the light source and moving, and way of tree traversal. However, there are some differences between the two graphic windows.

The first difference is the number of layers displayed. In this window, there are only two layers, Layer 1 for a **base virtual function** (in a base class), and Layer 2 for **the derived virtual functions** in derived classes of that base class. There is no Layer 0 at all in this window.

The second difference is the color of nodes. Here all the base virtual functions are in yellow and all the derived virtual functions in pink, the same as those in the call tree window, regardless of layer on which they are currently displayed. This allows a direct recognition of virtual functions when comparing the two display windows.

The third difference is the names of the default central nodes. In the call tree window, the name of the default central node is "main" and this is a natural choice, as every Win32 console application must have a main() function. However, there might be no virtual function in some C++ applications, so there might be no node at all in the virtual function window. However, if nothing is displayed in this window, the user may wonder whether the Browser works normally or not. One of the possible solutions is to display a default node with a name of "DummyVirtualRoot". In fact, this name is the root name of the virtual function BST during the construction of this BST: after all, this BST has to start from an initial root node.

4 Implementation

4.1 Introduction to Standard Template Library (STL)

The implementation of the Analyzer utilizes the Standard Template Library (STL) intensively, from the classes of various tokens, scanner, parser, to the service functions of retrieving function call hierarchy.

STL is considered by many to be the most important new features added to C++ in recent years. The inclusion of the STL was one of the major efforts that took place during the standardization of C++. It provides general-purpose classes and functions that implement many popular and commonly used algorithms and data structures, including, for example, support for vectors, lists, queues and stacks. It also defines various routines that access them. Because STL is constructed from template classes, the algorithms and data structures can be applied to nearly any type of data.

The STL is a complex piece of software engineering that uses some of C++'s most sophisticated features. The core of the STL are three foundational items: **containers, algorithms, and iterators**. These items works in conjunction with one another to provide off-the-shelf solutions to a variety of programming problems.

4.1.1 Containers

Containers are objects that hold other objects, and there are several different types. One type is called **sequence containers,** because in STL terminology, a sequence is a linear list: an example of such containers is the **vector** class, which defines a dynamic array. Another type is called **associative containers**, which allow efficient retrieval of

values based on keys: an example of such containers is the **map** class, which provides access to values with unique keys, since a map stores a key/value pair and allows a value to be retrieved given its key.

4.1.2 Algorithms

Algorithms act on containers. They provide the means by which you will manipulate the contents of containers. Their capabilities include initialization, sorting, searching, and transforming the contents of containers. Many algorithms operate on a range of elements within a container.

4.1.3 Iterators

Iterators are objects that are, more or less, pointers. They give you the ability to cycle through the contents of a container in much the same way that you would use a pointer to cycle through an array. There are five types of iterators: Random Access, Bidirectional, Forward, Input, and Output.

Iterators are handled just like pointers. You can increment and decrement them. You can apply the * operator to them. Iterators are declared by using the **iterator** type defined by various containers.

4.1.4 Examples of Container Classes

Containers are the STL objects that actually store data. Here are some examples of container classes used in this thesis (see Table 4.1):

Table 4.1 Examples of some container classes

Container	Description	Required Header
string	String class which manages character strings.	<string>
set	A set in which each element is unique.	<set>
map	Stores key/value pairs in which each key is associated with only one value.	<map>
vector	A dynamic array	<vector>

4.1.5 General Theory of Operation

Although the internal operation of the STL is highly sophisticated, the usage of the STL is actually quite easy. First of all, you must decide on the type of container that you wish to use, since each type offers benefits and trade-offs.

Then you will use the container's member functions to add elements to the container, access or modify those elements, and delete elements. Elements can be added to and removed from a container in a number of different ways, for example, to use the member functions called **insert**() and **erase**().

One of the most common ways to access the elements within a container is through an iterator. Both the sequence and associative containers provide the member functions **begin**() and **end**(), which return iterators to the start and end of the container, respectively. These iterators are very useful when accessing the contents of a container; for example, to cycle through a container, you can obtain an iterator to its beginning

by using begin() and then increment that iterator until its value is equal to end().

The associative containers provide the function **find**(). which is used to locate an element in an associative container given its key. Since associative containers link a key with its value. the function find() is how most elements in such a container are located.

Once you have a container that holds information. it can be manipulated by using one or more algorithms. The algorithms not only allow you to alter the contents of a container in some prescribed fashion. but they also let you transform one type of sequence into another.

4.2 Analyzer: Scanner and Parser

The Analyzer developed in this thesis is based on three class groups: **tokens. scanners.** and **parsers**. Also. some service functions are needed to pick up the required information used to construct function cross-reference hierarchy.

The design and implementation of primitive tokens retrieval are inspired with an off-the shelf C/C+ archives [5]: however. some modifications are necessary. The design and implementation of function name retrieval as well as function call tree construction are all completely and independently developed by this thesis.

4.2.1 The Token Classes

Figure 4.1 shows the hierarchical relationship of all the token classes. The base token class is CToken. and the other token classes are derived from CToken.

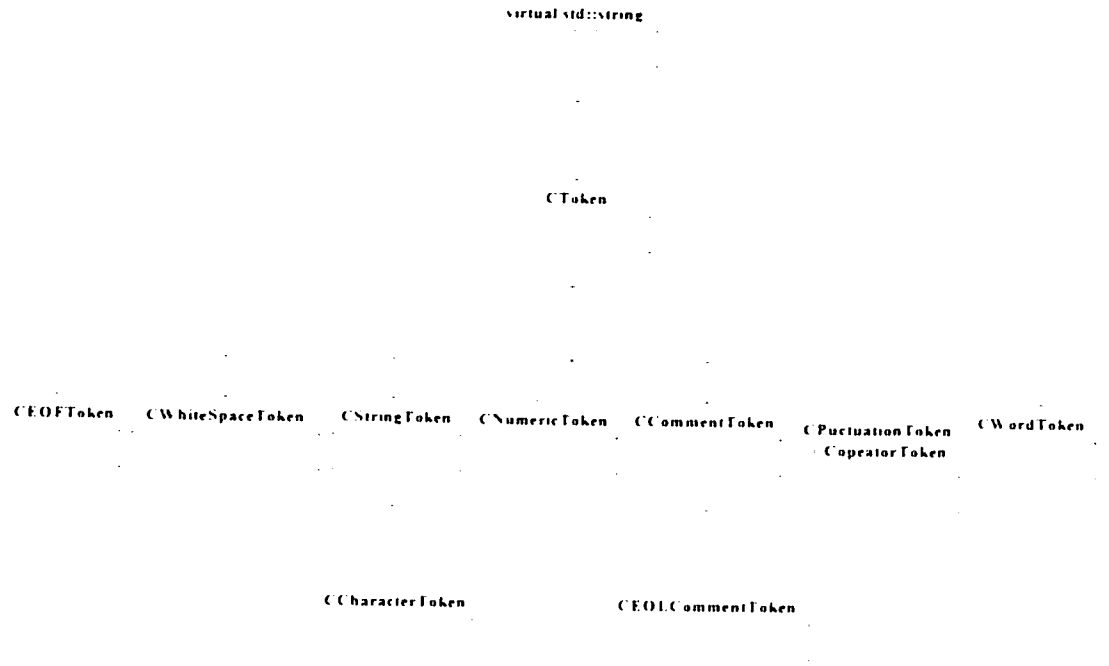


Figure 4.1 Hierarchical relationship of all the token classes

4.2.1.1 The Base Class: CToken

The **CToken** class is the generic version of a token. CToken is a subclass of std::string; this allows it to behave in the same manner as a std::string, inheriting all the operators and functions of class std::string. The details of class member functions can be found in reference [5].

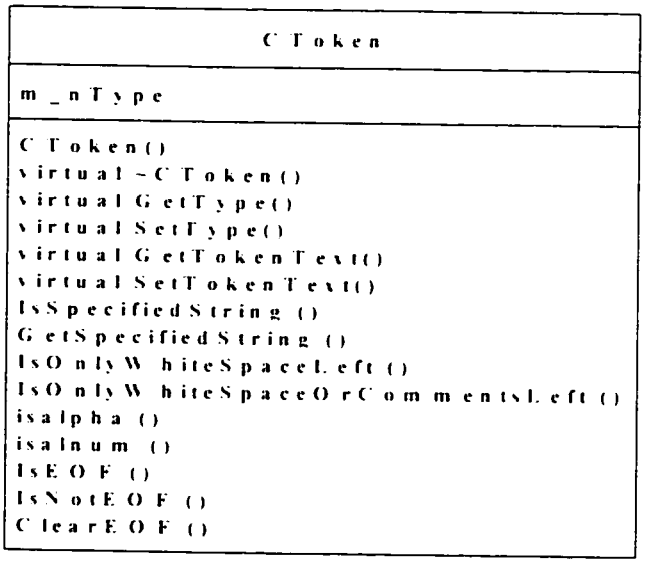


Figure 4.2 CToken class

4.2.1.2 Subclasses of CToken

There are some subclasses of CToken (see Table 4.2).

Table 4.2 Other token classes derived from CToken

Class Name	Class Description
CEOFToken	End of file
CWhiteSpaceToken	White space (a space, a tab, end of line, etc.)
CStringToken	String literal (e.g. "this is a string")
CCharacterToken 1)	Character literal (e.g. ':')
CNumericToken	Any numeric value
CCommentToken	Inline comment (e.g. /* Inline comment */)
CEOLCommentToken 2)	End of line comment (e.g. // EOL comment)
CPunctuationToken	Punctuation (e.g. '{' and '.')
COperatorToken	The same as CPuctuationToken
CWordToken	Any string of characters that does not fit any of the other categories defined in this table. Variable and function names are of this type tokens.

1): CCharacterToken is a subclass of CStringToken

2): CEOLCommentToken is a subclass of CCommentToken

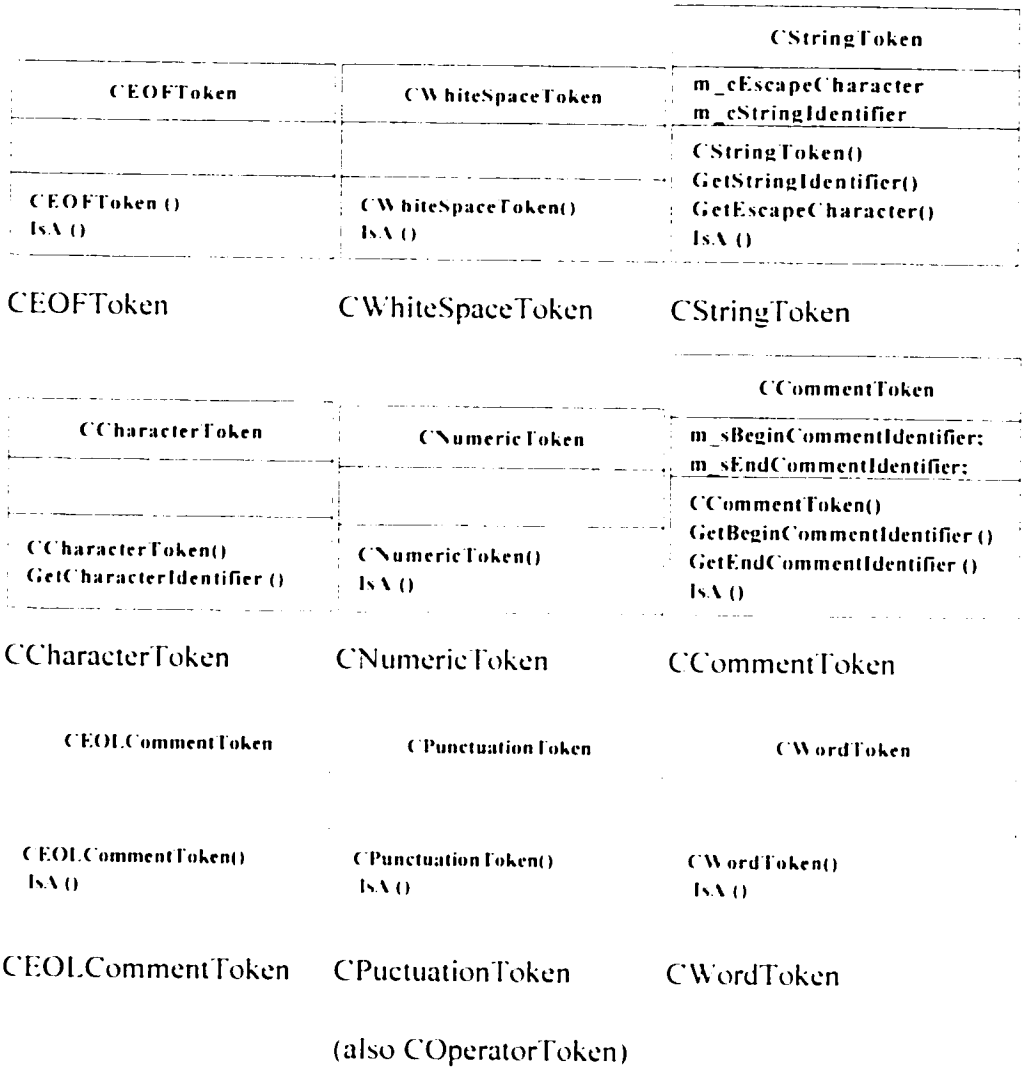
Note that each method within CToken is virtual: this allows any of the subclasses

derived from CToken to override any of these methods. In addition, CToken is not a pure virtual class, so it can be used without ever having to implement one of the derived classes. It can be used to create custom tokens without necessarily creating an entirely new class.

Each token implements two groups of functions:

1) The IsA() static function identifies the next token to be of the type specified. All tokens that are derived from CToken implement an IsA() static function. This enables the Scanner to call each token class to aid in identifying what type of token to create and return to the Parser. For example, when the Scanner checks whether the next token is end-of-file token, it simply calls CEOFToken::IsA(). This allows specific token functionality to remain local to the token.
2) The second set of functions are the constructor functions for each token type class. With the exception of CToken, no token class can be constructed without an **istream** parameter.

The followings are the Subclasses which are derived from base class CToken:



CEOFToken CWhiteSpaceToken CStringToken

CCharacterToken CNumericToken CCommentToken

CEOLCommentToken CPuctuationToken CWordToken

(also COperatorToken)

COtherToken

Figure 4.3 Subclasses derived from base class CToken

4.2.2 The Scanner Class: CScanner

The **CScanner** class is used to identify which token is next on the istream, and then to instantiate this token and pass it back to the Parser. The class is implemented as a subclass of **ifstream**.

The idea behind the CScanner class is to implement the same functionality as in istream, but with individual tokens rather than individual characters. Since the main purpose of the Scanner class is to retrieve the next token, the most important function of CScanner class is GetToken(). The Parser repeatedly calls CScanner::GetToken() in order to move through the entire file.

What the Scanner essentially does is to allow the Parser to work with individual tokens rather than individual characters. This enables the Parser to work with larger blocks of data. Spoken languages work the same way. Although it would be possible to speak in letters, this would be very impractical and extremely inefficient. Instead, the letters are combined to form words, which are then interpreted as a whole. Similarly, the Scanner combines characters to form "words", which the Parser can then interpret.

There are two tasks for the scanner to do:

1) The main task of the scanner is to break the input file down into individual tokens. The scanner identifies what the next token is going to be, instantiates a token of that type, and returns it back to the parser.

With the abundance of static IsA() functions found on each of the token classes, the Scanner simply has to ask each token class if the next token is of its type.

Complications arise, however, because of the order in which the tokens need to be checked. Ideally, they should be checked in order of frequency, to avoid needless IsA() function calls when they usually returns false. This approach presents a problem because some of the IsA() functions do not check for certain that the next token is of a particular type, so there is a chance that a token could be misidentified. If, for example, the CPunctuationToken::IsA() function were called before calling CStringToken::IsA(), the string might never get identified. The CPunctuationToken constructor would identify the next token as eTokenTypePunctuation, when instead CStringToken should be used. The Scanner therefore needs to be sure to identify the various tokens in the correct order.

The following order for identifying tokens is used by function CScanner::PeekTokenType():

a) IsWhiteSpaceToken,

b) IsWordToken,

c) IsCommentToken,

d) IsEOLCommentToken,

e) IsStringToken,

f) IsCharacterToken,

g) IsNumericToken,

h) IsPunctuationToken,

i) IsEOFToken

2) The second task of the scanner is to keep a list of the language characteristics

needed to identify a token (e.g. reserved words). Although some defaults are set for these values, it is necessary to explicitly assign the values for each type of language file that is to be scanned. There is a long list of member-variable access functions, which allows the language identifiers to be defined. Each of the methods is prefixed with "Set".

In theory, the above definitions of Parser and Scanner classes can be used to handle any language, so it is necessary to set them up so that they recognize the characteristics of C++, such as C++ language definitions (comment identifiers /* */ and //, etc.). The work is done by function LoadCPPScanner().

4.2.2.1 CScanner

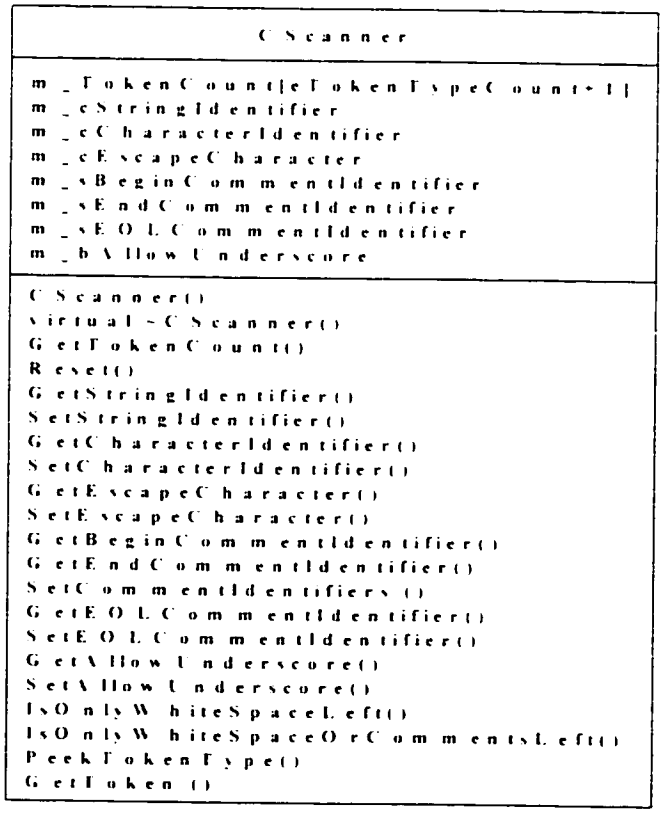


Figure 4.4 The CScanner class

4.2.3 The Parser Class. CCodePaser

The Parse class is used to parse programming source code.

4.2.3.1 CCodeParser

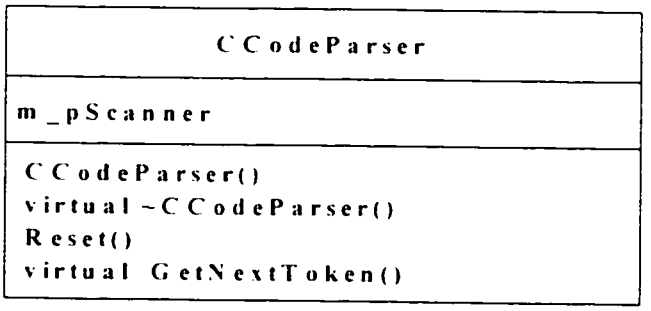


Figure 4.5 The CCodeParser class

4.2.4 Retrieval of Function Names

Similar to the implementation of the classes mentioned above, the implementation of retrieving function call hierarchy also utilizes the Standard Template Library (STL).

4.2.4.1 Retrieval of Class Names and Member Function Names

The first job when retrieving information on function call hierarchy is to find the names of all the classes and their member functions. The corresponding service function, ***GetClassMemFuncNameLists*** (), is used for this purpose. Its prototype is as follows:

bool GetClassMemFuncNameLists (CScanner &input,

ClassNameList & classNameList,

MemFuncNameList &memFuncNameList);

where *input* corresponds to a scanner object which represent the source file being

opened. *classNameList* and *memFuncNameList* correspond to global variables to store class names and member function names respectively. The types of the two name lists, *ClassNameList* and *MemFuncNameList*, are defined as container **set**<string>, a type of STL with element type of **string**, another type of STL:

typedef set< string > ClassNameList;

typedef set< string > MemFuncNameList;

The process of retrieving class and member function names is a repeated iteration. By calling the scanner's member function *input*.**GetToken**(), we can obtain a token (let's call it *token*) from the source file. The type and content of the token are obtained by calling functions *token*.**GetType**() and *token*.**GetTokenText**() respectively. If this token is not an EOF (End-Of-File) token, continue the following parsing process, until the EOF token is reached.

For every token retrieved from source file, we need to check if this token's content is the key word "class". If not, continue to get next token. If yes, ignore any white space token and find the first token of word type *eTokenTypeWord*; this should be a class name, so record it in a local variable *sClassToken*, and also call *classNameList*.**insert**(*sClassToken*) to insert the class name into *classNameList*, a container to hold all the class names.

Next, search for brace token "{", so that we know that the scanning is inside the class declaration. Keep the information on the number of braces by incrementing *NumOfOpenBraces*. Now we can get and check two successive tokens in order to find any pattern of "word + (", in which the "word" is a member function name. Record it

in another local variable *sFuncToken*.

Note that there is an issue which has to be resolved. Any of class member functions is expressed in the form of "ClassName::FuncName", so we need to concatenate the class name *sClassToken* found above with both string "::" and member function name *sFuncToken*, to have a full function name as *sClassToken* ::*sFuncToken*, and insert it into *MemFuncNameList* by calling function *memFuncNameList*.**insert**(*sClassFuncToken*).

Also, this full function name is inserted into a BST *Browser_BST*, which stores function cross-reference hierarchy information. The function call to do this is *Browser_BST*.**insert**(*sClassFuncToken*), and the details will be described in later sections.

A special case also needs to be treated, i.e. we should not pick up any initialization list as a function name, so we need to ignore any token before "{" if we find ")" and ":" after a class name.

If we find a token "}", decrement *NumOfOpenBraces*. If it is zero, we know the scanning is getting out of a class declaration, so continue to scan the source file in order to find the next class name, if it exists.

4.2.4.2 Retrieval of Independent Function Names

In addition to class member functions, in any C++ application there are possibly some other functions which are not class member functions at all. The main function **main()** is a good example. We call this type of function as **independent functions**. So a

companion job when retrieving information on function call hierarchy is to find the names of all the independent functions. The corresponding service function, ***GetIndepFuncNameLists*** (), is used for this purpose. Its prototype is as follows:

bool GetIndepFuncNameLists (CScanner &input,
MemFuncNameList &memFuncNameList,
IndepFuncNameList &indepFuncNameList);

where *input* corresponds to a scanner object which represent the source file being opened. *memFuncNameList* is still the global variable to store class member function names, and *indepFuncNameList* is a global variable to store independent function names. The type of the later list, *IndepFuncNameList*, is also defined as container **set**<string>, a type of STL with element type of **string**, another type of STL:

typedef set< string > IndepFuncNameList;

The process of retrieving independent function names is as follows. We need to find any pattern of "word + (", under the constraint of *NumOfOpenBraces* = 0; this constraint will ensure that the function name found is not a class member at all. Then record this independent function name in local variable *sFuncToken*, and insert it into *indepFuncNameList*, *memFuncNameList*, and *Browser_BST*, by calling the following functions:

indepFuncNameList.**insert**(*sFuncToken*);

memFuncNameList.**insert**(*sFuncToken*);

Browser_BST.**insert**(*sClassFuncToken*);

4.2.4.3 Retrieval of Object Names

After we find class names, we are ready to retrieve **object** names in a source file. The corresponding service function, ***GetObjectNameList*** (), is used for this purpose. Its prototype is as follows:

bool GetObjectNameList (CScanner &input,

ClassNameList &classNameList,

ObjectNameList &objectNameList);

where *input* corresponds to a scanner object which represent the source file being opened. *classNameList* is still the global variable to store class names, and *objectNameList* is a global variable to store object names. The type of the later list, *objectNameList*, is defined as a container **map**<string,string>, a type of STL with element of type **string**, another type of STL:

typedef map< string,string > ObjectNameList;

For every pair in the map, we will use the key (the first element) to store an object name, and the value (the second element) to store the class name of the object, so that we will have a correct mapping relationship between an object name and its class name.

The process of retrieving object names is as follows. For every token of word type *eTokenTypeWord*, we need to check if it is a class name. If yes, we know the parser is scanning a statement of object declaration. Then, after ignoring white spaces, comment, and pointer symbol "*", the next word token should be an object name. Record this object name by calling *objectNameList*.**insert** (**make_pair**(*sObjectToken*,

sClassToken)). This search for object names should not terminate before finding a token ":". We need to ignore any token except "." or ":", in case there are more objects being declared after the same class name: if a token "." is found, that means there are more object names on the current line.

The above algorithm has one drawback, i.e. it may pick up a wrong word token as an object name. For example, consider the following object declaration

Bus bs1(white, 12, yes);

Our algorithm will pick up "yes" as an object name, which is not correct. The solution is simple: after finding an object name, if the next token is "(" (after ignoring possible white space and comment), we continue to ignore all the token until reaching token ")".

4.2.4.4 Retrieval of Called Function Names

This is the most crucial and complicated step during the construction of function cross reference hierarchy.

There are three cases we need to treat respectively: for every function found before, we need to get its called functions 1) from outside class declaration but in class member functions, 2) from inside class declaration, and 3) from outside class declaration but in independent functions.

4.2.4.4.1 Retrieve Called Functions from outside class declaration but in class member functions

This case corresponding to any called functions **outside** class declaration but **inside** class member functions in the form of "ClassName :: FuncName ()". The service function has the following prototype:

```
bool GetCalledFuncNameList (Cscanner          &input,
                            ClassNameList     &classNameList,
                            ObjectNameList    &objectNameList,
                            IndepFuncNameList & indepFuncNameList,
                            CalledFuncNameList &calledFuncNameList);
```

where *input* corresponds to a scanner object which represent the source file being opened. *classNameList*, *objectNameList*, *indepFuncNameList* are still the global variables to store class names, object names, and independent function names; *calledFuncNameList* is a global variable to store the called function names. The type of the later list, *calledFuncNameList*, is defined as container **set**<string>, a type of STL with element types of **string**, another type of STL:

```
typedef set<string> CalledFuncNameList;
```

The process of retrieving called function names is as follows:

First of all, we need to find class name by comparing any word token with every element in the container *classNameList*. If we find a class name, we need to search the token "::", and get the word token afterwards, which is a function name. Then we construct a *sParentFuncToken* having a pattern of "ClassName :: FuncName".

Next, find the first token "{" after *sParentFuncToken*, indicating the parser enters the function definition body. Then we can search for called function names by checking every word token inside function body. Here we need to treat two different case.

One case is for the independent function names. This case is simple. For every word token inside function body, we just need to check if it is in the container *indepFuncNameList* by traversing this container (using its iterator) and calling *indepFuncNameList*.**find**(*token*): if find an independent function name, we insert it into container *calledFuncNameList*, and then insert this container into *Browser_BST*.

Another case is for class member function names. This case is a little bit complicated. For every word token inside function body, we need to check if it is in the container *objectNameList* by traversing this container (using its iterator) and calling *objectNameList*.**find**(*token*): if find an object name, we need to retrieve its class name from the map (i.e. the *objectNameList*) and temporally record its class name in a local variable *sClassToken*. Then search for token "." or "->", and the word token after them should be a called function name and it is also of class member function! Record this function name in a local variable *sFuncToken* and a full name of called function, *sCalledFuncToken*, can be obtained as "*sClassToken* ::*sFuncToken*". Finally this full name is inserted into the container *calledFuncNameList* and into *Browser_BST*.

4.2.4.4.2 Retrieve Called Functions from inside class declaration

This case corresponding to any called functions **inside** class, i.e. the inline functions inside class declaration body. The service function has the following prototype:

```
bool GetCalledFuncNameList2 (Cscanner       &input,
                             ClassNameList  &classNameList,
                             ObjectNameList &objectNameList,
                             MemFuncNameList &memFuncNameList,
                             IndepFuncNameList &indepFuncNameList,
                             CalledFuncNameList &calledFuncNameList);
```

where *input* corresponds to a scanner object which represent the source file being opened. *classNameList*, *objectNameList*, *memFuncNameList*, *indepFuncNameList* are still the global variables to store class names, object names, class member function names, and independent function names; *calledFuncNameList* is still the global variable to store the called function names.

The process of retrieving called function names is as follows:

First of all, we need to find class name; this is the same as mentioned above. Then enter class declaration body, and for every word token check to see if it is a class member function; if yes and the function has a body (i.e. it is an inline function), we enter its body and search for both the independent function names and the class member function names, and record into the *calledFuncNameList* and *Browser_BST*, in the same way as described in the previous case.

4.2.4.4.3 Retrieve Called Functions from outside class declaration but in independent functions

This case corresponds to any called functions **outside** class declaration but **inside**

independent functions. The service function has the following prototype:

```
bool GetCalledFuncNameList3 (Cscanner      &input,
                             ClassNameList     &classNameList,
                             ObjectNameList    &objectNameList,
                             MemFuncNameList   &memFuncNameList,
                             IndepFuncNameList &indepFuncNameList,
                             CalledFuncNameList &calledFuncNameList);
```

where *input* corresponds to a scanner object which represent the source file being opened. *classNameList*, *objectNameList*, *memFuncNameList*, *indepFuncNameList* are still the global variables to store class names, object names, class member function names, and independent function names; *calledFuncNameList* is still the global variable to store the called function names.

The process of retrieving called function names is as follows:

First of all, we need to find an independent function name. Then enter its body and search for both the independent function names and the class member function names, and record into the *calledFuncNameList* and *Browser_BST*, in the same way as described in the previous case.

Here a special issue need to be treated. It happens more often in independent functions and it is more or less related with dynamic binding when virtual functions are invoked. We need to pick up the correct object names but not miss member function names. For example, in the following statement

```
ObjName[i]->MemFunc();
```

we need to ignore all the tokens from "[" to "]", otherwise MemFunc will never be picked up as a member function name. The details of how to handle virtual functions are provided in later sections (see Section 4.2.4.6).

4.2.4.5 Retrieval of Calling Function Names

The service function has the following prototype:

*bool GetCallingFuncNameList (BinNode< string > * rootptr,*

CallingFuncNameList &callingFuncNameList);

where *rootptr* is the pointer to root node of *Browser_BST*, and *callingFuncNameList* is a global variable to store the calling function names. The type of *calledFuncNameList* is defined as container **set**<string >, a type of STL with element types of **string**, another type of STL:

typedef set< string > CallingFuncNameList;

The process of retrieving calling function names is fairly easier. We do not need to scan and parse the C++ source file again, because all the information on calling function names does exist in the *Browser_BST* structure. What we need to do is to traverse this BST recursively. The detailed procedure is as follows:

Starting from the root node of *Browser_BST*, for the function in this root node (say *RootFunc*), iterate through its **called function name list**. For every called function (say *CalledFunc*), search the *Browser_BST* to see if such a called function name exists in BST; if yes, the current root node function *RootFunc* should be a calling function of that called function *CalledFunc*, and we update the **calling function name list** of the

node which hold *CalledFunc*, by inserting the current *RootFunc* into that calling function name list.

4.2.4.6 Retrieval of Virtual Function Names

4.2.4.6.1 Handle Virtual Functions in Base Classes

The service function to retrieving base virtual function names has the following prototype:

bool GetVirtualFuncNameLists (Cscanner &input,
ClassNameList &classNameList,
VirtualFuncNameList &baseVirtualFuncNameList);

where *input* corresponds to a scanner object which represent the source file being opened. *classNameList, objectNameList* is still the global variable to store class names; *baseVirtualFuncNameList* is defined as container **set<string>**, a type of STL with element types of **string**, another type of STL:

typedef set< string > VirtualFuncNameList;

This process to retrieving virtual function names during the construction of Function Cross-Reference Browser is more sophisticated.

First of fall, we need to insert a dummy name of "*DummyVirtualRoot*" into *baseVirtualFuncNameList* by calling the function *baseVirtualFuncNameList*.**insert**("*DummyVirtualRoot*"); this is because there might be no virtual function at all in some C++ applications, but we need to have such a dummy node in order to better represent such a scenario on the screen.

Next, we need to enter a class declaration body, to find the token of keyword "**virtual**", in the same way of finding keyword "class". If found, then the word token after "virtual" but before "(" should be a virtual function name in this **base** class. (Note that the class here is base class since it declares virtual function by using keyword "virtual".) So, we have a full name in the form of "ClassName::FuncName", and we insert it into *baseVirtualFuncNameList* and *VirtualFunc_BST*, another BST for holding virtual functions (see the later sections for details). In the meanwhile, we need to update the node corresponding to this base virtual function in *Browser_BST* by setting the flag *enVirtualFlag* to VIRTUAL_FUNC_FLAG_BASE. Similarly, we do not pick up initialization list as function names.

4.2.4.6.2 Handling Virtual Functions in Derived Classes

The service function to retrieve the derived virtual function names has the following prototype:

*bool GetVirtualFuncNameLists2 (BinNode< string > * virtualrootptr,*
VirtualFuncNameList &baseVirtualFuncNameList);

where *virtualrootptr* is the pointer to root node of *VirtualFunc_BST*, and *baseVirtualFuncNameList* is a global variable to store the base virtual function names. Similar to the concept of retrieving the calling function names, we can retrieve the derived virtual function names of a base virtual function by traversing the two BSTs, *VirtualFunc_BST* and *Browser_BST*. Here are the steps:

1) Search *VirtualFunc_BST*, get base virtual function name in the form of

BaseClassName::VirtualFuncName", then remove "ClassName::" to get string of "VirtualFuncName";

2) Search *Browser_BST*, to get a function name in the form of "ClassName::FuncName", then remove "ClassName::" to get string of "FuncName";

3) Compare "VirtualFuncName" with "FuncName", "BaseClassName" with "ClassName": if "FuncName" = "VirtualFuncName" but "ClassName" != "BaseClassName", then we found a virtual function in derived class.

The above service function is called recursively, so that we can traverse the whole *VirtualFunc_BST*. By the way, a helper function is required during the above traversal, and its prototype is shown below:

```
bool FindVirtualFuncNameListinBrowser_BST (
        BinNode<string>*    browserrootptr,
        string              baseVirtualClassName,
        string              baseVirtualFuncName,
        VirtualFuncNameList derivedVirtualFuncNameList);
```

where the *derivedVirtualFuncNameList* is a container holding all the derived virtual function names.

4.2.5 Construction of Function Call Hierarchy

4.2.5.1 Template of Binary Search Tree (BST)

A template class for a binary search tree (BST) is defined. Each node in the BST has

an element to hold data, a flag showing if the node holds virtual functions or not, and two pointers pointing to its left and right child nodes. Also defined in the node are two containers, one for called functions, another for calling functions. There are also some member functions to manipulate the above member data.

4.2.5.1.1 Binary Node Class

Figure 4.6 shows BinNode, the class of binary node:

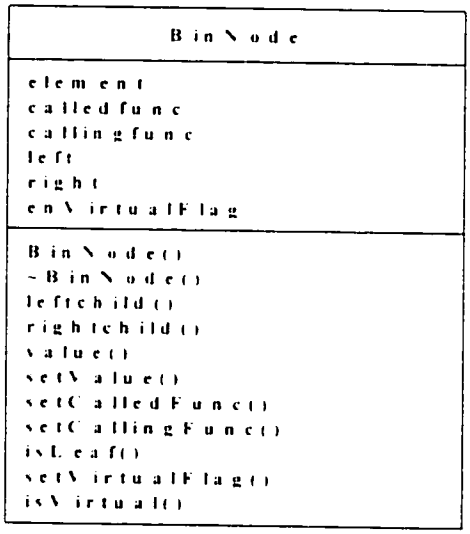


Figure 4.6 The binary node class BinNode

4.2.5.1.2 Binary Search Tree Class

Figure 4.7 shows BST, the class of binary search tree:

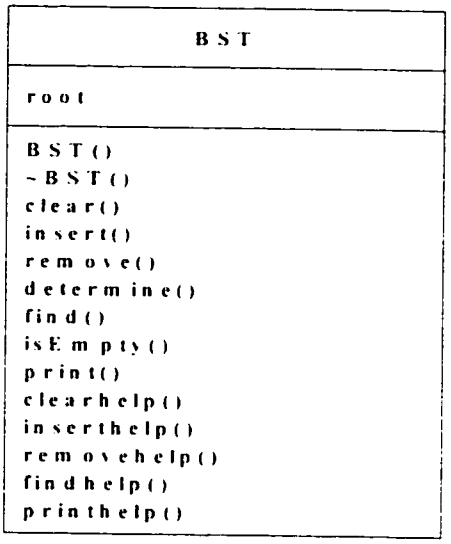


Figure 4.7 The binary search tree class BST

4.2.5.2 Construction of the Binary Search Tree for Function Call Hierarchy

The key function during the construction of *Browser_BST* is **insert**(). Whenever a function name is ready to be inserted into *Browser_BST*, this function, *Browser_BST*.**insert**(*FuncNameToken*), is called and the *FuncNameToken* will be inserted into this BST. Underlying operations include allocating memory for new node, fill in element value with function name, and adjust pointers in the tree.

4.2.5.3 Construction of the Binary Search Tree for Virtual Functions

Similaly, whenever a virtual function name is found and ready to be inserted into *VirtualFunc_BST*, the function, *VirtualFunc_BST*.**insert**(*VirtualFuncNameToken*) is called and the *VirtualFuncNameToken* will be inserted into this BST. In addition, the *Browser_BST* should be up dated at the virtual function node, to reflect the fact of

virtuality of the corresponding node.

The above sections describe the Analyzer part of the Function Cross-Reference Browser. Next the Displayer part will be introduced.

4.3 Manipulator: Displayer

4.3.1 Introduction to OpenGL

OpenGL is a software interface to graphics hardware. This interface consists of about 150 distinct commands, aiming at specifying the objects and operations needed to produce interactive three-dimensional applications.

OpenGL is designed as a streamlined, hardware-independent interface, so 1) it is not a windowing system, i.e. no commands exists for performing windowing tasks (such as opening windows) or for obtaining use input (however, portable libraries does exists, e.g. GLUT, see below); and 2) it does not provide high-level commands for describing models of three-dimensional objects (however, Open Inventors does); instead, the user must build up the desired model from a small set of geometric primitives – points, lines, and polygons.

OpenGL consists of three libraries: the Graphics Library (**GL**), the Graphics Utility Library (**GLU**), and the Graphics Library Utility Toolkit (**GLUT**). GL is the standard parts of OpenGL and are available on all SGI platforms, Sun workstations, Windows 95/98/ NT/2000, etc.

OpenGL provides the following operations:

- Construct shapes from geometric primitives, thereby creating mathematical descriptions of objects.
- Arrange the objects in 3D space and select the desired vantage point for viewing the composed scene.
- Convert the mathematical descriptions of objects and their associated color information to pixels on the screen (this process is called rasterization).
- Other operations: e.g. eliminating parts of objects that are hidden by other objects.

OpenGL is a state machine. It can be put into various states that then remain in effect until being changed. Here are some examples of state variables, such as the current color, the current viewing and projection transformations, line and polygon stipple patterns, position and characteristics of lights. Each state variable has a default value, and many state variables refer to modes that can be enabled or disabled.

Most OpenGL implementations have a similar order of operations, a series of processing stages called **OpenGL rendering pipeline**. It consists of display lists, evaluators, per-vertex operations and primitive assembly, pixel operations, texture assembly, rasterization, per-fragment operations, and finally framebuffer. Vertex data and pixel data follow the above path from the display lists to the framebuffer.

In summary, OpenGL is a powerful tool which is very suitable for 3D graphics.

4.3.2 Displaying of Function Call Hierarchy

In one of the two OpenGL graphic windows (opened in the **main()** function, see next later section), the function call hierarchy which is stored in *Browser_BST* is displayed.

The OpenGL callback function to do this is **display**(). This call back function does the following tasks:

- clear the color bit, and request hidden surface removal;
- position a white light source and show where it is;
- display a red, dull sphere at the center for a function; however, a yellow sphere is displayed if it is a base virtual function, and a pink sphere for derived virtual function;
- display cyan, polished spheres for all the upper nodes (i.e. calling functions) and display lines linking the central node to the upper nodes; all the upper node are positioned evenly in one plane; use the same treatment as above for virtual functions;
- display green, polished spheres for all the lower nodes (i.e. called functions) and display lines linking the central node to the lower nodes; all the lower node are positioned evenly in one plane; use the same treatment as above for virtual functions;
- display function names beside each node; if there is no calling or called function, a special name is displayed, either "No Calling Func" or "No Called Func";
- display a help menu in this window.

There are other callback functions provided, such as **keyboard**(), **specialkey**(), **mouse_movement**(), and **resize_window**(). Their meaning is self-explained by their names. These callback functions are all registered in the **init**() function by the following GLUT library routines before they can actually do their jobs:

glutDisplayFunc(display);

glutKeyboardFunc(keyboard);

glutSpecialFunc(specialkey);

glutMotionFunc(mouse_movement);

glutReshapeFunc(resize_window);

The **init**() function is used to initialize the browser window after it is created. Here are some helper functions: **showstring**(), **showhelp**(), **help**(), and **menu**(). The last one is used to create and attach a menu to this OpenGL window (by click the right mouse key).

4.3.3 Displaying of Virtual Functions

In another OpenGL graphics window, the relationship between a base virtual function and its derived virtual functions, which is stored in *VirtualFunc_BST*, is displayed. The OpenGL callback function to do this is **display2**(). Basically this callback function does the same thing for the virtual function window as **display**() does for the browser window, except it does not show upper layer nodes (since the central node is always a base virtual function and hence there is no upper node at all); another exception is **display2**() does not show a help menu, as this menu is shown in the browser window. The callback function **display2**() is registered in **init2**(), together with other callback functions such as **keyboard**(), **specialkey**(), **mouse_movement**(), and **resize_window**().

glutDisplayFunc(display2);

glutKeyboardFunc(keyboard);

glutSpecialFunc(specialkey);

glutMotionFunc(mouse_movement);

glutReshapeFunc(resize_window);

The function **init2()** is used to initialize the virtual function window after it is created.

4.4 The Main Flow of Browser -- main() Function

4.4.1 User Input Prompt

Before starting the browser program, we need to know the path name of a directory which contains all the C++ source files for a particular application. (This browser does not treat files in subdirectories). Then the user may enter the directory path name (a full-path name is required) after the browser executable name in the DOS window and press the ENTER key to proceed. If the path name is invalid, or if the directory is empty, the browser program exits, otherwise, it will start to parse all the C++ source files and construct function cross reference hierarchy, and finally display it in graphic windows.

4.4.2 Directory Scanning to Collect C++ Source File Names

The Win32 APIs of **FindFirstFile()** and **FindNextFile()** are used to retrieve C++ source file names under a directory. After a fine name is retrieved, it is inserted into a container of type **set**<string>. This source file list will be used later by the parser for accessing every source file.

4.4.3 Source File Parsing

There are several steps to parse the source files, and they must be done in the following order:

(1) scan the input file and get the class name list, member function name list, and insert the member function names into *Browser_BST*:

LoadCPPScanner *(*SourceCode);*

GetClassMemFuncNameLists *(*SourceCode, classlist, funclist);*

(2) scan the input file and get the independent function name list, and insert the independent function names into *Browser_BST*:

LoadCPPScanner *(*SourceCode);*

GetIndepFuncNameLists *(*SourceCode, funclist, indepfunclist);*

(3) scan the input file and get the object name list:

LoadCPPScanner *(*SourceCode);*

GetObjectNameList *(*SourceCode, classlist, objectlist);*

(4) scan the input file and get the called function name list, and insert the called function names into *Browser_BST*:

LoadCPPScanner *(*SourceCode);*

GetCalledFuncNameList *(*SourceCode, classlist, objectlist, indepfunclist, calledfunclist);*

LoadCPPScanner *(*SourceCode2);*

GetCalledFuncNameList2 *(*SourceCode2, classlist, objectlist, funclist,*

indepfunclist, calledfunclist);

LoadCPPScanner *(*SourceCode3);*

GetCalledFuncNameList3 *(*SourceCode3, classlist, objectlist, funclist,*

indepfunclist, calledfunclist);

(5) get the calling function name list and insert the calling function names into *Browser_BST*:

Note: here we do not need to scan source files, because all the information on the calling functions are already available in the *Browser_BST*.

GetCallingFuncNameList*(bstrootptr, callingfunclist);*

(6) scan the input file and get the base virtual function name list, and insert the base virtual function names into *VirtualFunc_BST*; also update the virtual function flag in *Browser_BST*;

LoadCPPScanner *(*SourceCode5);*

GetVirtualFuncNameLists *(*SourceCode5, classlist, basevirtualfunclist);*

(7) get the derived virtual function name list, and insert the derived virtual function names into *VirtualFunc_BST*; also update the virtual function flag in *Browser_BST*;

Note: here we do not need to scan source files, because all the information on the derived virtual functions are already available in the *Browser_BST*.

GetVirtualFuncNameLists2 *(virtualrootptr, basevirtualfunclist);*

4.4.4 Graphical Display

The following procedures are the steps used by OpenGL to output the 3D graphical displays:

- system initial setup and hidden surface removal

  glutInit(&argc, argv);

  glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH);

- set up the first window: Function Cross-Reference Browser

  glutInitWindowSize(window_width, window_height);

  glutInitWindowPosition(250, 20);

  glutCreateWindow("Function Cross-Reference Browser");

  init();

- add menu for all the function names

  glutCreateMenu(menu);

  *glutAddMenuEntry((*m).c_str(), f);*

  glutAttachMenu(GLUT_RIGHT_BUTTON);

- set up the second window: Virtual Function Window

  glutInitWindowSize(300, 300);

  glutInitWindowPosition(50, 80);

  glutCreateWindow("Virtual Function Window");

  init2();

- add menu for virtual functions only

  glutCreateMenu(menu2);

VirtualFuncNameList::const_iterator n;

*glutAddMenuEntry((*n).c_str(), g);*

glutAttachMenu(GLUT_RIGHT_BUTTON);

- start OpenGL

  glutMainLoop();

5 Result and Discussion

5.1 Accomplishment of Design Goal

The goal of Function Cross-Reference Browser is to help software engineers to understand the function call relationships in a large, complicated software application written in C++. Specifically, the browser is able to extract the information on function call hierarchy, to build a function call tree, and to display the call hierarchy in a 3D window. The graphical display of function call hierarchy can help software engineers to better understand the architecture and the working principle of software programs. In general, this thesis has accomplished the desired design goal. The following two sections will provide detailed descriptions about how the design goal is accomplished, from two aspects of both function call hierarchy extraction and 3D display.

5.1.1 Function Call Hierarchy Extraction

For a software application written in C++, the Function Cross-Reference Browser can extract the call hierarchical relationships among the functions. Since the C++ grammar is very complicated, this Browser will be applicable to a proper subset of C++ features. There are some preconditions or assumptions for this Browser: 1) all the source files must reside within a single directory; 2) the extension names must be either ".cpp" or ".h"; 3) the entry function must be the ordinary main function "main()"; 4) there is no function overloading (same function name, but with different parameters or return type); however, dynamic binding is really allowed; 5) there should be no such statements as "class Dummy;" (forward declaration) and "void IndepFunc(void);"

(function prototype); etc. (This issue can be addressed in the future, since it is not hard to resolve.)

The Browser works in a way as follows. First of all, the Browser opens the C++ files (either .cpp or .h) and breaks down the source files into primitive tokens. Secondly, depending on the token types, the Browser finds all the names of classes, member functions, independent functions; in the meanwhile, the Browser also constructs a *Browser_BST* for all the functions. Next, the Browser finds the names of objects, and for every function in *Browser_BST*, finds its called functions and records it into *Browser_BST*. Now for every function in *Browser_BST*, we have enough information to know its calling functions, and record the calling functions into *Browser_BST*. Finally, the Browser finds virtual functions both in base classes and in derived classes, and constructs a *VirtualFunc_BST*. Appendix B shows the textual output during the extraction process of the function call hierarchy.

5.1.2 Three Dimensional Display

After the Browser has extracted all the information on function call hierarchy, it displays the call hierarchy in two 3D windows, one for all functions in *Browser_BST*, another for virtual functions in *VirtualFunc_BST* only.

In the first window (the Browser Window, see Figure 5.1), there are 3 three layers displayed, each layer contains spheres of different colors: the middle layer (Layer 1, in red) is for a function (called Central Function) in *Browser_BST*, the lower layer (Layer 2, in green) is for Called Functions of the Central Function, and the upper layer

(Layer 0, in cyan) is for Calling Functions of the Central Function. The default Central Function is the "main". For the virtual functions, we use different colors: the base virtual functions are always in yellow, and the derived virtual functions are always in pink, no matter at which layer they are.

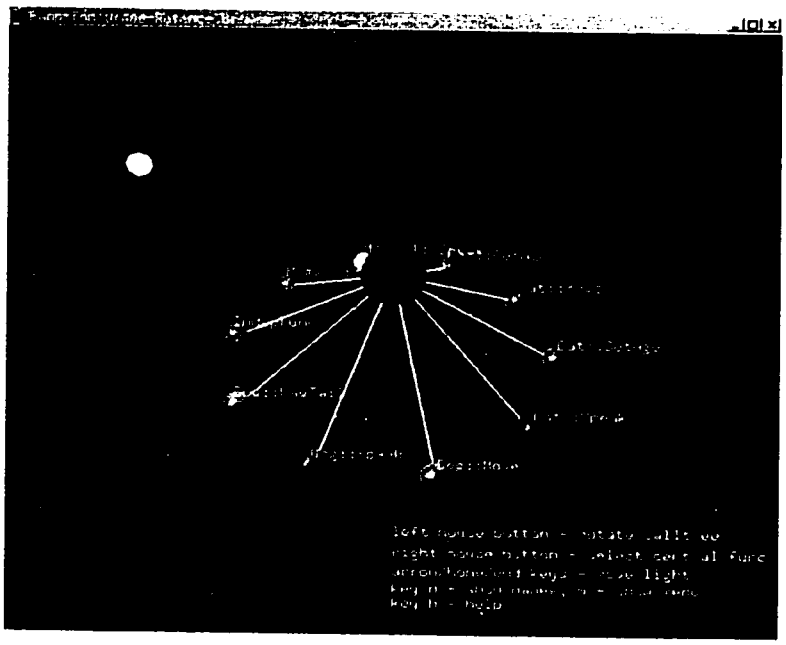


Figure 5.1 The Browser Window after executing the Browser program

The traversal of *Browser_BST* can be performed by clicking right mouse key to display a list of all the functions and then by picking up another function from the list as a new Central Function: the Browser Window is updated and redisplayed for the new Central Function as well as for its Called and Calling Functions (see Figure 5.2).

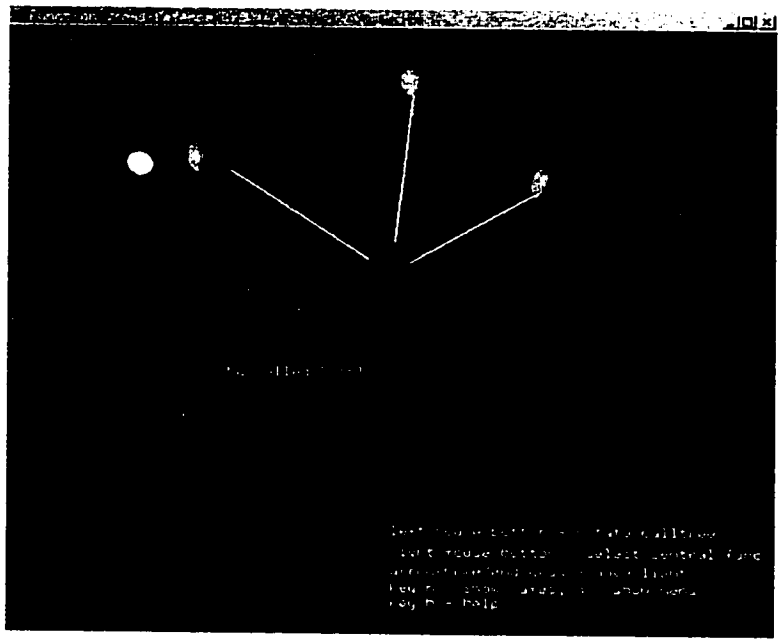


Figure 5.2 The Browser Window after selecting another central function

In the second window (the Virtual Function Window, see Figure 5.3), there are 2 layers displayed, the middle layer is for base virtual functions, and the lower layer is for derived virtual functions. There is no upper layer in this window. The default base virtual function is "DummyVirtualRoot". The colors of the spheres at 2 layers are kept consistent with those in the Browser window, i.e. base virtual function in yellow again and derived virtual function in pink again. This will help the user to identify virtual functions easily in the two windows.

The traversal of *VirtualFunc_BST* can be performed in the same way as that of *Browser_BST*.

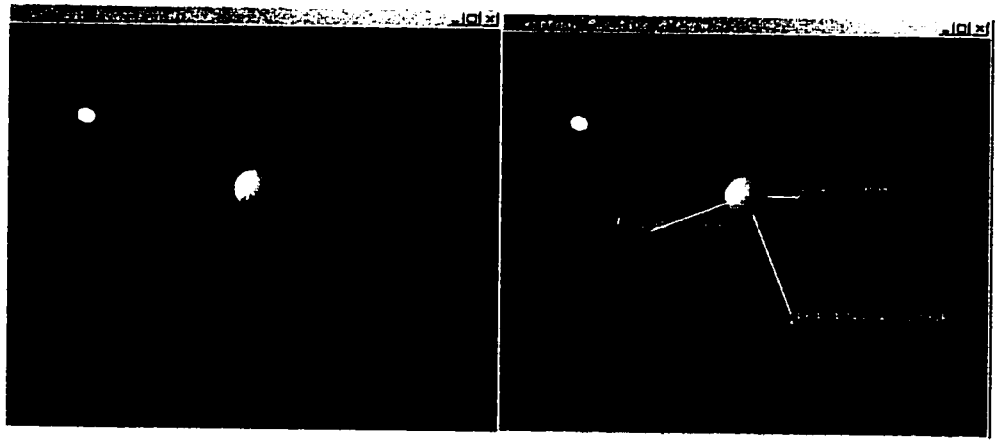


Figure 5.3 The Virtual Function Window (a) after executing the Browser program: (b) after selecting another central function.

5.2 Result of Operation on Testing C++ Application

The sample C++ application for testing this Browser is shown in Appendix A. This C++ program is specially designed for testing the working capability of the Browser. The textual output after executing the Browser program is shown in Appendix B. Figure 5.4 shows part of the event output during traversal to the call tree.

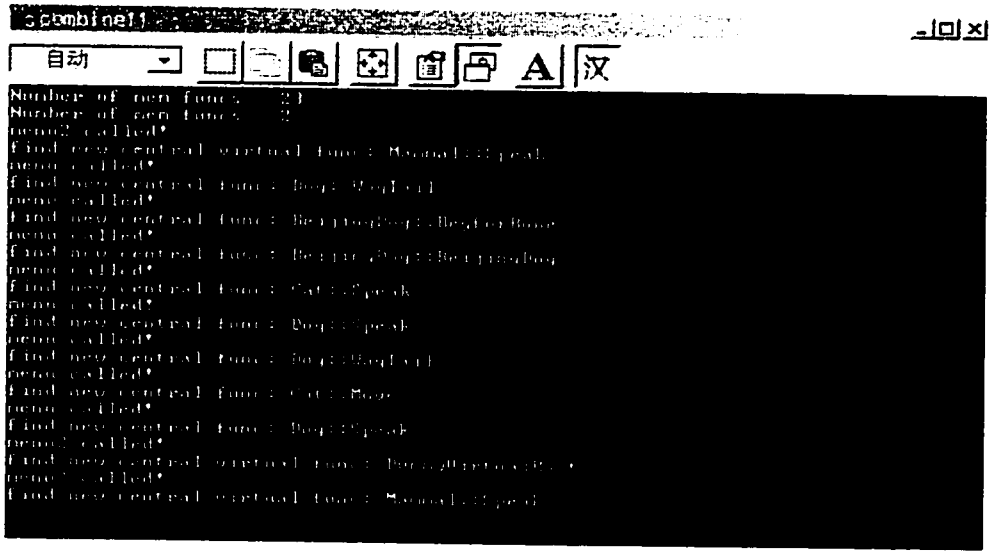


Figure 5.4 The DOS Window showing tree traversal events

5.3 Discussion

From the textual output and the graphical display, we know that the Browser can work correctly.

1. The Browser can identify C++ **source file** names correctly, i.e. it gives the whole path for each file.
2. The Browser parses each of the C++ source files and correctly finds all the **classes** declared in them.

   Note: This Browser can correctly avoid picking up other tokens as class names, and can correctly ignore the white spaces before or after class names.
3. The Browser correctly finds all the **member functions** in all the classes as well as all the **independent functions**, and constructs a BST for them. It can correctly avoid picking up the **initialization list** as a member function name. Also, it can correctly handle the **inline** member function as well as the **non-inline** member function in class declarations.
4. The Browser can correctly find all the **object** names (either objects or pointer to object) in source files, with the correct correspondence with their class types.
5. The Browser can correctly find the **called** function names of any function in the source files, and insert the called functions into corresponding node in *Browser_BST*.
6. The the Browser can correctly find the **calling** function names of any function in the source files, and insert the calling functions into corresponding node in

Browser_BST.

7. The the Browser can correctly find the **virtual** function names in the source files, and construct *VirtualFunc_BST*.

8. The Browser executes the OpenGL program and displays the function call hierarchy, as well as the virtual function hierarchy, in two separate windows, respectively. Each called function is connected to the central function with a white line, and so is each calling function. By default, the function names are displayed besides the functions.

   The central function is displayed in red, the calling function in cyan, and the called function in green. However, the virtual functions are always displayed with unique colors in the two windows, the base virtual function in yellow, and the derived virtual function in pink, so that it is very obvious to identify them.

   Now the program stops execution and enters a standby mode, waiting for any further input from the user. (Recall the OpenGL program is like a finite state machine (FSM) and it stays in one state before receiving further event.)

9. Further operation 1: the help menu. We can use the keyboard to display (or to hide) the function names and a brief help menu in the OpenGL graphic window, by pressing the key "n" and "m", respectively; also, we can press key "h" to show a detailed help menu in the DOS window.

10. Further operation 2: the tree traversal. By clicking the right mouse key, we can select another function from the function name list as a new central function, and display its called and calling functions. This is the way we traverse the tree of the

function call hierarchy.

11. Further operation 3: the tree rotation. By clicking the left mouse key, keep pressing and move the cursor in the graphical window, we can rotate the call tree, in order to view any hidden node and its name clearly.

12. Further operation 4: moving the light source. We can use the 6 special keys on the keyboard to move the position of light source, in order to have a better view of the 3D display. The 6 special keys are Left, Right, Up, Down, Home and End, for moving the light source to directions of left, right, up, down, towards the use, backwards the user, respectively.

13. Finally, the similar operations mentioned above can also be applied to the Virtual Function Window, for showing the hierarchical relationships between a base virtual function and its derived virtual functions.

In summary, this Browser can work correctly as the design indicated before.

6 Further Development and Conclusion

6.1 Possible Further Development

There are some possible enhancements which can be further achieved to this Function Cross-Reference Browser:

1. Right now the Browser requires that all the **C++ source files** must be put **within a single directory**, and it cannot handle source files inside any sub-directory. This is not a difficult issue, as we can further treat the attribute FILE_ATTRIBUTE_DIRECTORY of the Win32 API FindFirstFile() and FindNextFile(), so that we can recursively search sub-directories to find the whole path names of all the C++ source files in an application.
2. Currently the Browser can only handle such **extension names** of the C++ source files as ".cpp" or ".h": This is also an easy issue. We can enhance the Browser to handle other extension names such as "cxx".
3. Currently, the Browser can **only handle Win32 Console application**, i.e. the entry function name must be "**main()**": it **cannot handle Win32 Window application** with entry function name of "**WinMain()**". One possible solution is to provide a menu on the command line, e.g., if enter choice 1, the Browser will treat Console application, and if enter choice 2, treat Window application. Perhaps we need to write 2 different versions of the Browser in order to treat the 2 different cases.
4. Although this Browser can handle **function overriding** (changing the implementation of a base class function in a derived class, i.e. the case of dynamic

binding), it cannot handle **function overloading**, i.e. functions with the same name but different parameters or return type. It seems this is not an easy issue, as it may require more strict parsing to the source files (e.g. checking the number of input attributes, the types of attributes, and the types of return values). As we mentioned earlier, C++ grammar is not context-free, and usually a tool cannot handle all the features in C++. This issue is left for further study.

5. The Browser still cannot handle **class forward declaration** (e.g. the statement of "class Dummy;") and **function prototype** (e.g. the statement *of "void IndepFunc(void);"*). This is also not a very hard issue, as it just requires a more careful parsing to the source file. This Browser has already set up a working foundation of token parsing, and with a further step of trying, it will be able to handle such an issue. However, the solution is not provided here in the Browser due to the time limit of this thesis; instead, it is left for further study.

6.2 Conclusion

The Function Cross-Reference Browser described in this thesis continues the research on three dimensional software visualization, and makes some achievements in this field. Specifically, this Function Cross-Reference Browser has the following characteristics:

1. The Browser is able to parse the C++ source files (in a single directory, with extension names of ".cpp" and ".h") of a Win32 Console application, and to extract the function cross reference hierarchy (i.e. function call relationships). The

Browser can handle a proper subset of the standard C++ features, including dynamic binding.

2. Then the Browser is able to display the function cross reference hierarchy in 3D format, and it allows a convenient traversal to the function call tree. The Browser also has other features to allow clear displaying of the function call tree.

In summary, this Browser is a useful tool for software engineers. It can help them better understand a large, complicated software application written in C++ language. This thesis work adds a new member to the software visualization.

7 References

[1]. "**A Principled Taxonomy of Software Visualization**", by Blaine A. Price, et al., *Journal of Visual Languages and Computing*, Vol. 4, No. 3, Sep 1993, pp. 211-266. Also see: http://kmi.open.ac.uk/~doc/jvlc/JVLC-Body.html

[2]. "**Software Visualization—Programming as a Multimedia Experience**", edited by John Stasko, et al., MIT Press, 1998.

[3]. "**System Visualizer**", by Kim Thang Vu, Thesis of Master in Computer Science, Dept. of Computer Science, Concordia University, 1997.

[4]. "**C++: The Complete Reference**", 3rd ed., by Herbert Schildt, McGraw-Hill, 1998.

[5]. "**C/C++ Annotated Archives**", by Art Friedman, Lars Klander, Mark Michaelis, and Herb Schildt, McGraw-Hill, 1999.

[6]. "**Teach Yourself C++ in 21 Days**", 2nd ed., by Jesse Liberty, Sams Publishing, 1997.

[7]. "**OpenGL Programming Guide**", 2nd ed., by Mason Woo, Jakie Neider, and Tom Davis, Addison-Wesley, 1997.

[8]. "**The Unified Modeling Language User Guide**", by Grady Booch, Ivar Jacobson, and James Rumbaugh, Addison Wesley Longman, 1999.

8 Appendix A Test Case

This appendix gives a C++ application as a test case. It contains two files, one header file (Mammal.h) and one source file (Mammal.cpp).

8.1 A.1 Mammal.h

```
#include <iostream.h>

//class Dummy, Dog;   ---this is a compilation error
//Sun-10-24 Note:
//  the current Browser cannot handle the following two:
  class Dummy;
  class Dog;

  class     Mammal
{
public:
    Mammal
        : itsAge( 1 ) { cout << "Mammal constructor... \n";
    ~Mammal    { cout << "Mammal destructor... \n";

    void Move    const { cout << "Mammal move 1 step. \n";
    virtual void Speak    const { cout <<  "Mammal speak! \n";

protected:
    int itsAge;

 };


class Dog:public Mammal
{
public:
    Dog    { cout << "Dog constructor... \n";
    ~Dog    { cout << "Dog destructor... \n";

    void Move    const { cout << "Dog moves 1 step. \n";
    void Speak    const
    {
        Cat CatinDog ;
        CatinDog.Move  ;
       cout << "Dog woof! \n";
```

```
    }

    void WagTail  {"Dog wagging tail....n";
};


class Cat : public Mammal
{
public:
    Cat int initialAge ;
    ~Cat  ;

    void Move   const {cout << "Cat moves " steps. n";
    void Speak   const {cout << "Cat meow' n";}

    int  GetAge  ;
    void SetAge int  age ;
    void Call()   { Dog Dog.hiCat;
                    Dog.hiCat.Speak  ;
                  }
};


class BeijingDog : public Dog
{
public:
    BeijingDog   {cout << "BeijingDog constructor.... n";
    ~BeijingDog   {cout << "BeijingDog destructor.... n";

    void Move   const {cout <  "BeijingDog moves " steps. n";
    void Speak   const {cout << "BeijingDog in..woof' n";

    void BetterBone  {"BeijingDog begs for bone... n";

};
```

8.2 A.2 Mammal.cpp

```
#include <iostream>
#include "Mammal.h"

// 2001-12-26 Note:
//   the current Browser cannot handle the following case:
// void IndepFunc(void ;
```

```
Cat::Cat(int initialAge)
{
    Dog DogObj;
    DogObj.Speak();
    DogObj.WagTail();

    IndepFunc();
    cout << "Cat constructor... \n";
    itsAge = initialAge;
    cout << "Cat initial age is " << itsAge << "\n";
}

Cat::~Cat()
{
    cout << "Cat destructor... \n";
}

int Cat::GetAge()
{
    return itsAge;
}

void Cat::SetAge(int age)
{
    itsAge = age;
}

//-------------------------------------
// int ClassNone::main()
int main()
{
    // Dog Max, Mid, *pMid = new Dog,
    //    Min;
    // Max.Move();
    // Max.Speak();
    // Max.WagTail();
    // cout << "\n";

    // Mammal *pDog = new Dog;
    // pDog->Move();
```

```
pDog->Speak  ;
// pDog->WagTail  ;
 * Note: "WagTail" is not a member function of "Mammal". *
cout << "\n";

}
Cat Frisky 5 ;
Frisky.Move  ;
Frisky.Speak  ;
cout << "Frisky's initial age is ";
cout << Frisky.GetAge  << " \n";
Frisky.SetAge 7 ;
cout << "Frisky's new age is ";
cout << Frisky.GetAge  << " \n";
cout << " \n";

}
Mammal *pCat = new Cat ;
pCat->Move  ;
pCat->Speak  ;
cout << " \n";

{
int pFun;  // 0x401b1e

}
Mammal *MammalArray[7];
Mammal Felix;
BeijingDog John;
Mammal *pBeijingDog = new BeijingDog;

cout << "\n------ start of dynamic binding test ------" << endl;
MammalArray[0] = &Felix;          // class Mammal
MammalArray[1] = &Max;            // class Dog
MammalArray[2] = pDog;            // class Mammal
MammalArray[3] = &Frisky;         // class Cat
MammalArray[4] = pCat;            // class Mammal
MammalArray[5] = &John;           // class BeijingDog
MammalArray[6] = pBeijingDog;     // class Mammal

for (int counter = 0; counter < 7; counter++)
    MammalArray[counter]->Speak  ;

cout << "------ end of dynamic binding test ------\n" << endl;
```

```
    return 1;
}

void IndepFunc(void)
{
...
    Mammal *pDog = new Dog  ;
    //pDog->WagTail  ;
    //Mammal has no member func as WagTail; it failing it,
    // my program will crash; in fact, cann't pass compilation.

    pDog->Move  ;
    cout << "Indep func" << endl;
}
```

8.3 A.3 The textual output of testing application

```
Mammal constructor...
Dog constructor...
Mammal constructor...
Dog constructor...
Mammal constructor...
Dog constructor...
Mammal constructor...
Dog constructor...
Dog moves 2 step.
Dog woof!

Mammal constructor...
Dog constructor...
Mammal moves 1 step.
Dog woof!

Mammal constructor...
Mammal constructor...
Dog constructor...
Dog woof!
Mammal constructor...
Dog constructor...
Mammal moves 1 step.
Indep func!
Cat constructor...
Cat initial age is 5
```

```
Dog destructor...
Mammal destructor...
Cat moves 3 steps.
Cat meow!
Frisky's initial age is 5
Frisky's new age is 7

Mammal constructor...
Mammal constructor...
Dog constructor...
Dog woof!
Mammal constructor...
Dog constructor...
Mammal moves 1 step.
Indep fun!
Cat constructor...
Cat initial age is 9
Cat destructor...
Mammal destructor...
Mammal moves 1 step.
Cat meow!

Mammal constructor...
Dog constructor...
Mammal moves 1 step.
Indep fun!
Mammal constructor...
Mammal constructor...
Dog constructor...
BeijingDog constructor...
Mammal constructor...
Dog constructor...
BeijingDog constructor...

------ start of dynamic binding test ------
Mammal speaks!
Dog woof!
Dog woof!
Cat meow!
Cat meow!
BeijingDog sh..woof!
BeijingDog sh..woof!
------ end of dynamic binding test ------
```

```
BeijingDog destructor...
Dog destructor...
Mammal destructor...
Mammal destructor...
Cat destructor...
Mammal destructor...
Dog destructor...
Mammal destructor...
Dog destructor...
Mammal destructor...
Dog destructor...
Mammal destructor...
```

9 Appendix B The Textual Output of the Browser

The following is the textual output in the DOS window after running the Browser to the testing C++ application listed in Appendix A. The testing application is put under the directory E:\TestSrc6\ .

```
Enter directory name: e:\TestSrc6
Now search the directory: e:\testsrc6
File name: e:\testsrc6\Mammal.cpp
File name: e:\testsrc6\Mammal.h

Parse C++ source files:

inputfile: e:\testsrc6\Mammal.cpp
  1  Now search class names and fun() names:
'class name set' classlist is empty'
'fun() name set' funclist is empty'

inputfile: e:\testsrc6\Mammal.h
  1  Now search class names and fun() names:

display 'class name set' classlist:
BeijingDog
Cat
Dog
Mammal

display fun() name set funclist:
BeijingDog::BeijingDog
BeijingDog::BeijingDog
BeijingDog::Move
BeijingDog::Speak
BeijingDog::~BeijingDog
Cat::CallDog
Cat::Cat
Cat::GetAge
Cat::Move
Cat::SetAge
Cat::Speak
Cat::~Cat
Dog::Dog
Dog::Move
Dog::Speak
```

```
Dog::WagTail
Dog::~Dog
Mammal::Mammal
Mammal::Move
Mammal::Speak
Mammal::~Mammal

Now print BST Browser_BST from main  :
main
--------Mammal::~Mammal
----------------Mammal::Speak
------------Mammal::Move
----Mammal::Mammal
------------Dog::~Dog
------------------------Dog::WagTail
--------------------Dog::Speak
----------------Dog::Move
--------[illegible]
----------------[illegible]
--------------------------[illegible]
------------------------------[illegible]
--------------------[illegible]
------------------------[illegible]
------------[illegible]
----------------[illegible]
------------------------Beijing[illegible] Beijing[illegible]
----------------------------------Beijing[illegible]Speak
------------------------------Beijing[illegible]Move
--------------------Beijing[illegible]Beijing[illegible]
------------------------Beijing[illegible]BeijingBing

Parse 1++ source files:

inputfile: e: testsrc\Mammal.cpp
 lb  Now search indep fun: names:
'''find :: '''
'''find :: '''
### insert indepfun:: IndepFunc

display indep func name set indepfunclist:
IndepFunc
main

Now print BST Browser_BST from main  :
```

```
main
--------Mammal:::~Mammal
----------------Mammal:::Speak
------------Mammal:::Move
----Mammal:::Mammal
----------------IndepFun:
------------Dog:::~Dog
------------------------Dog:::WagTail
--------------------Dog:::Speak
----------------Dog:::Move
--------Dog:::Dog
----------------Cat:::~Cat
------------------------Cat:::Speak
----------------------------Cat:::GetAge
--------------------Cat:::Move
------------------------Cat:::SetAge
------------Cat:::Cat
----------------Cat:::Cat(int)
------------------------Brother:::~Brother()
--------------------------------Brother:::Speak
------------------------------Brother:::Move
--------------------Brother:::Brother()
------------------------Brother:::BrotherBrother
In fun: IndepFun: in Brother_BJT
```

```
Inputfile: ./mammal.in
 In  New search IndepFun: names:

Display IndepFun: Number and IndepFun(i):
IndepFun:
main
```

```
Now print BJT Browser_BJT from main  :
main
--------Mammal:::~Mammal
----------------Mammal:::Speak
------------Mammal:::Move
----Mammal:::Mammal
----------------IndepFun:
------------Dog:::~Dog
------------------------Dog:::WagTail
--------------------Dog:::Speak
----------------Dog:::Move
--------Dog:::Dog
```

```
----------------Cat::~Cat
------------------------Cat::Speak
----------------------------Cat::GetAge
--------------------Cat::Move
------------------------Cat::GetAge
------------Cat::Cat
----------------Cat::CallDog
------------------------BeijingDog::~BeijingDog
--------------------------------BeijingDog::Speak
----------------------------BeijingDog::Move
--------------------BeijingDog::BeijingDog
------------------------BeijingDog::BeijingDog
(gdb) find InvokeFunc in Browser_BJT
No symbol "InvokeFunc" in current context.
(gdb) info source files
Source files for which symbols have been loaded:

/home/test/Browser_BJT/Browser_BJT.cpp
/home/test/Browser_BJT/Cat.cpp
/home/test/Browser_BJT/Mammal.cpp
/home/test/Browser_BJT/BeijingDog.cpp
/home/test/Browser_BJT/Browser_BJT.h
/home/test/Browser_BJT/Cat.h
/home/test/Browser_BJT/Mammal.h
/home/test/Browser_BJT/BeijingDog.h
(gdb) find class name: Cat
Find class name: Cat
Find class name: Cat
Find class name: Cat
--Find object name: Cat::Cat
Find class name: Cat
Find class name: Cat
Find class name: Cat
Find class name: Cat
--Find object name: Max
--Find object name: Mia
--Find object name: pMia
--Find object name: Mia
Find class name: Mammal
--Find object name: pCat
Find class name: Cat
--Find object name: Frisky
find class name: Mammal
--find object name: pCat
find class name: Mammal
--find object name: MammalArray
find class name: Mammal
--find object name: Felix
find class name: BeijingDog
--find object name: John
find class name: Mammal
--find object name: pBeijingDog
find class name: Mammal
```

```
--find object name: pDog

Object = DogObj, Class = Dog
Object = Felix, Class = Mammal
Object = Frisky, Class = Cat
Object = John, Class = BeijingDog
Object = MammalArray, Class = Mammal
Object = Max, Class = Dog
Object = Mid, Class = Dog
Object = Min, Class = Dog
Object = pBeijingDog, Class = Mammal
Object = pCat, Class = Mammal
Object = pDog, Class = Mammal
Object = pMid, Class = Dog

-- Now set object names:
find class name: Mammal
find class name: Mammal
find class name: Dog
find class name: Mammal
find class name: Dog
find class name: Cat
find class name: Mammal
find class name: Cat
find class name: Dog
--find object name: BeijingDog
find class name: BeijingDog
find class name: Dog
find class name: BeijingDog

Object = DogObj, Class = Dog
Object = OriginCat, Class = Dog
Object = Felix, Class = Mammal
Object = Frisky, Class = Cat
Object = John, Class = BeijingDog
Object = MammalArray, Class = Mammal
Object = Max, Class = Dog
Object = Mid, Class = Dog
Object = Min, Class = Dog
Object = pBeijingDog, Class = Mammal
Object = pCat, Class = Mammal
Object = pDog, Class = Mammal
Object = pMid, Class = Dog
```

```
Parse C++ source files:

  3  Now get called func names:

Now build calledFuncNameList and insert into BST'


sParentFuncToken = Token::Int
InFuncDefinition = 1
find Object name Sub(b) and sClassToken = 0;
find Object name Sub(b) and sClassToken = 0;
--- sCalledFuncToken = Token::Sub
--- and sCalledFuncToken inserted into BST'
find Object name Sub(b) and sClassToken = 0;
--- sCalledFuncToken = 0 Token::Null
--- and sCalledFuncToken inserted into BST'
find independent func name IndepFunc
--- sCalledFuncToken = IndepFunc
--- and sCalledFuncToken inserted into BST'

sParentFuncToken = Token::Int
InFuncDefinition = 1

sParentFuncToken = Token::Int
InFuncDefinition = 1

sParentFuncToken = Token::Int
InFuncDefinition = 1
InFuncDefinition = 1
InFuncDefinition = 1
InFuncDefinition = 1
InFuncDefinition = 1
InFuncDefinition = 1
InFuncDefinition = 1
InFuncDefinition = 1
InFuncDefinition = 1
InFuncDefinition = 1

Now build calledFuncNameList  2  and insert into BST'


Now build calledFuncNameList  3  and insert into BST'

****** 88888***** find indepfunc: sParentFuncToken = IndepFunc
```

```
-- ****@@@@***** find indepfunc: sParentFuncToken = main
find object name Max and sClassToken = Dog
find object name Mid and sClassToken = Dog
find object name pMid and sClassToken = Dog
find object name Min and sClassToken = Dog
find object name Max and sClassToken = Dog
--- sCalledFuncToken = Dog::Move
--- and sCalledFuncToken inserted into BST
find object name Max and sClassToken = Dog
--- sCalledFuncToken = Dog::Speak
--- and sCalledFuncToken inserted into BST
find object name Max and sClassToken = Dog
--- sCalledFuncToken = Dog::WagTail
--- and sCalledFuncToken inserted into BST
find object name pDog and sClassToken = Mammal
find object name pDog and sClassToken = Mammal
--- sCalledFuncToken = Mammal::Move
--- and sCalledFuncToken inserted into BST
find object name pDog and sClassToken = Mammal
--- sCalledFuncToken = Mammal::Speak
--- and sCalledFuncToken inserted into BST
find object name Frisky and sClassToken = Cat
find object name Frisky and sClassToken = Cat
--- sCalledFuncToken = Cat::Move
--- and sCalledFuncToken inserted into BST
find object name Frisky and sClassToken = Cat
--- sCalledFuncToken = Cat::Speak
--- and sCalledFuncToken inserted into BST
find object name Frisky and sClassToken = Cat
--- sCalledFuncToken = Cat::GetAge
--- and sCalledFuncToken inserted into BST
find object name Frisky and sClassToken = Cat
--- sCalledFuncToken = Cat::GetAge
--- and sCalledFuncToken inserted into BST
find object name pCat and sClassToken = Mammal
find object name pCat and sClassToken = Mammal
--- sCalledFuncToken = Mammal::Move
--- and sCalledFuncToken inserted into BST
find object name pCat and sClassToken = Mammal
--- sCalledFuncToken = Mammal::Speak
--- and sCalledFuncToken inserted into BST
```

```
find independent func name IndepFunc
--- sCalledFuncToken = IndepFunc
--- and sCalledFuncToken inserted into BST'
find object name MammalArray and sClassToken = Mammal
find object name Felix and sClassToken = Mammal
find object name John and sClassToken = BeijingDog
find object name pBeijingDog and sClassToken = Mammal
find object name MammalArray and sClassToken = Mammal
find object name Felix and sClassToken = Mammal
find object name MammalArray and sClassToken = Mammal
find object name Max and sClassToken = Dog
find object name MammalArray and sClassToken = Mammal
find object name pDog and sClassToken = Mammal
find object name MammalArray and sClassToken = Mammal
find object name Felicky and sClassToken = Cat
find object name MammalArray and sClassToken = Mammal
find object name pCat and sClassToken = Mammal
find object name MammalArray and sClassToken = Mammal
find object name John and sClassToken = BeijingDog
find object name MammalArray and sClassToken = Mammal
find object name pBeijingDog and sClassToken = Mammal
find object name MammalArray and sClassToken = Mammal
--- sCalledFuncToken = Mammal::Speak
--- and sCalledFuncToken inserted into BST'
--- find IndepFunc: sParentFuncToken = IndepFunc
find object name pDog and sClassToken = Mammal
find object name pDog and sClassToken = Mammal
--- sCalledFuncToken = Mammal::Move
--- and sCalledFuncToken inserted into BST'


Now display calledFuncNameList for each node:


There are mem funcs  i.e. parent funcs :
--parent fucn = BeijingDog::BegforBone
----no calledfunc for this node'
--parent fucn = BeijingDog::BeijingDog
----no calledfunc for this node'
--parent fucn = BeijingDog::Move
----no calledfunc for this node'
--parent fucn = BeijingDog::Speak
----no calledfunc for this node'
--parent fucn = BeijingDog::~BeijingDog
```

```
----no calledfunc for this node'
--parent func = Cat::CallDog
----no calledfunc for this node'
--parent func = Cat::Cat
----its called func = Dog::Speak
----its called func = Dog::WagTail
----its called func = IndepFunc
--parent func = Cat::GetAge
----no calledfunc for this node'
--parent func = Cat::Move
----no calledfunc for this node'
--parent func = Cat::GetAge
----no calledfunc for this node'
--parent func = Cat::Speak
----no calledfunc for this node'
--parent func = Cat::Cat
----no calledfunc for this node'
--parent func = Cat::Cat
----no calledfunc for this node'
--parent func = Cat::Move
----no calledfunc for this node'
--parent func = Cat::Speak
----no calledfunc for this node'
--parent func = Cat::WagTail
----no calledfunc for this node'
--parent func = Cat::Cat
----no calledfunc for this node'
--parent func = IndepFunc
----its called func = Mammal::Move
--parent func = Mammal::Mammal
----no calledfunc for this node'
--parent func = Mammal::Move
----no calledfunc for this node'
--parent func = Mammal::Speak
----no calledfunc for this node'
--parent func = Mammal::Mammal
----no calledfunc for this node'
--parent func = main
----its called func = Cat::GetAge
----its called func = Cat::Move
----its called func = Cat::GetAge
----its called func = Cat::Speak
----its called func = Dog::Move
----its called func = Dog::Speak
```

```
----its called func = Dog::WagTail
----its called func = IndepFunc
----its called func = Mammal::Move
----its called func = Mammal::Speak

  }  Now get called func names:

Now build calledFuncNameList and insert into BST'

InFuncDefinition = 0
InFuncDefinition = 0
InFuncDefinition = 0
InFuncDefinition = 0
InFuncDefinition = 0

Now build calledFuncNameList  2  and insert into BST'


sClassToken    Mammal
find mem func name'
--sParentFuncToken    Mammal::Mammal
InFuncDefinition = 1
find mem func name'
--sParentFuncToken = Mammal::~Mammal
InFuncDefinition = 1
find mem func name'
--sParentFuncToken = Mammal::Move
InFuncDefinition = 1
find mem func name'
--sParentFuncToken = Mammal::Speak
InFuncDefinition = 1

sClassToken = Dog
find mem func name'
--sParentFuncToken = Dog::Dog
InFuncDefinition = 1
find mem func name'
--sParentFuncToken = Dog::~Dog
InFuncDefinition = 1
find mem func name'
--sParentFuncToken = Dog::Move
InFuncDefinition = 1
find mem func name'
--sParentFuncToken = Dog::Speak
```

```
InFuncDefinition = 1
find mem func name'
--sParentFuncToken = Dog::WagTail
InFuncDefinition = 1

sClassToken = Cat
find mem func name'
--sParentFuncToken = Cat::Cat
InFuncDefinition = 1
find mem func name'
--sParentFuncToken = Cat::~Cat
InFuncDefinition = 1
find mem func name'
--sParentFuncToken = Cat::Move
InFuncDefinition = 1
find mem func name'
--sParentFuncToken = Cat::Speak
InFuncDefinition = 1
find mem func name'
--sParentFuncToken = Cat::DoAdd
InFuncDefinition = 1
find mem func name'
--sParentFuncToken = Cat::DoAdd
InFuncDefinition = 1
find mem func name'
--sParentFuncToken = Cat::GetID
InFuncDefinition = 1
find object name Domin1' and sClassToken = Dog
FindCalledFunc = 1
find object name Domin1' and sClassToken = Dog
FindCalledFunc = 1
--- sCalledFuncToken = Dog::Speak
--- and sCalledFuncToken inserted into SCT'

sClassToken = BeijingDog
find mem func name'
--sParentFuncToken = BeijingDog::BeijingDog
InFuncDefinition = 1
find mem func name'
--sParentFuncToken = BeijingDog::~BeijingDog
InFuncDefinition = 1
find mem func name'
--sParentFuncToken = BeijingDog::Move
InFuncDefinition = 1
```

```
find mem func name'
--sParentFuncToken = BeijingDog::Speak
InFuncDefinition = 1
find mem func name'
--sParentFuncToken = BeijingDog::BegforBone
InFuncDefinition = 1

Now build calledFuncNameList  3  and insert into BST'



Now display calledFuncNameList for each node:


There are mem funcs  i.e. parent funcs :
--parent func = BeijingDog::BegforBone
----no calledfunc for this node'
--parent func = BeijingDog::BeijingDog
----no calledfunc for this node'
--parent func = BeijingDog::Move
----no calledfunc for this node'
--parent func = BeijingDog::Speak
----no calledfunc for this node'
--parent func = BeijingDog::~BeijingDog
----no calledfunc for this node'
--parent func = Cat::CallDog
----its called func = Dog::Speak
--parent func = Cat::Cat
----its called func = Dog::Speak
----its called func = Dog::WagTail
----its called func = IndepFunc
--parent func = Cat::GetAge
----no calledfunc for this node'
--parent func = Cat::Move
----no calledfunc for this node'
--parent func = Cat::GetAge
----no calledfunc for this node'
--parent func = Cat::Speak
----no calledfunc for this node'
--parent func = Cat::~Cat
----no calledfunc for this node'
--parent func = Dog::Dog
----no calledfunc for this node'
--parent func = Dog::Move
```

```
----no calledfunc for this node!
--parent fucn = Dog::Speak
----no calledfunc for this node!
--parent fucn = Dog::WagTail
----no calledfunc for this node!
--parent fucn = Dog::~Dog
----no calledfunc for this node!
--parent fucn = IndepFunc
----its called func = Mammal::Move
--parent fucn = Mammal::Mammal
----no calledfunc for this node!
--parent fucn = Mammal::Move
----no calledfunc for this node!
--parent fucn = Mammal::Speak
----no calledfunc for this node!
--parent fucn = Mammal::~Mammal
----no calledfunc for this node!
--parent fucn = main
----its called func = Cat::GetAge
----its called func = Cat::Move
----its called func = Cat::GetAge
----its called func = Cat::Speak
----its called func = Dog::Move
----its called func = Dog::Speak
----its called func = Dog::WagTail
----its called func = IndepFunc
----its called func = Mammal::Move
----its called func = Mammal::Speak

 4  Now get calling fun' names:
--------- Current root func: main
---------- find node: Cat::GetAge
read callingfunc
Update its callingfunc
Write back its callingfunc
---------- find node: Cat::Move
read callingfunc
Update its callingfunc
Write back its callingfunc
---------- find node: Cat::GetAge
read callingfunc
Update its callingfunc
Write back its callingfunc
---------- find node: Cat::Speak
```

```
read callingfunc
Update its callingfunc
Write back its callingfunc
---------- find node: Dog::Move
read callingfunc
Update its callingfunc
Write back its callingfunc
---------- find node: Dog::Speak
read callingfunc
Update its callingfunc
Write back its callingfunc
---------- find node: Dog::WagTail
read callingfunc
Update its callingfunc
Write back its callingfunc
---------- find node: IndepFunc
read callingfunc
Update its callingfunc
Write back its callingfunc
---------- find node: Mammal::Move
read callingfunc
Update its callingfunc
Write back its callingfunc
---------- find node: Mammal::Speak
read callingfunc
Update its callingfunc
Write back its callingfunc
--------- current root func: Mammal::Mammal
--------- current root func: Dog::Dog
--------- current root func: Cat::Cat
---------- find node: Dog::Speak
read callingfunc
Update its callingfunc
Write back its callingfunc
---------- find node: Dog::WagTail
read callingfunc
Update its callingfunc
Write back its callingfunc
---------- find node: IndepFunc
read callingfunc
Update its callingfunc
Write back its callingfunc
--------- current root func: Cat::CallDog
---------- find node: Dog::Speak
```

```
read callingfunc
update its callingfunc
write back its callingfunc
-------- current root func: BeijingDog::BeijingDog
-------- current root func: BeijingDog::BegforBone
-------- rootptr is NULL!
-------- rootptr is NULL!
-------- current root func: BeijingDog::~BeijingDog
-------- current root func: BeijingDog::Move
-------- rootptr is NULL!
-------- current root func: BeijingDog::Speak
-------- rootptr is NULL!
-------- rootptr is NULL!
-------- rootptr is NULL!
-------- rootptr is NULL!
-------- current root func: Cat::~Cat
-------- current root func: Cat::Move
-------- current root func: Cat::GetAge
-------- rootptr is NULL!
-------- rootptr is NULL!
-------- current root func: Cat::Speak
-------- current root func: Cat::GetAge
-------- rootptr is NULL!
-------- rootptr is NULL!
-------- rootptr is NULL!
-------- rootptr is NULL!
-------- current root func: Dog::~Dog
-------- current root func: Dog::Move
-------- rootptr is NULL!
-------- current root func: Dog::Speak
-------- rootptr is NULL!
-------- current root func: Dog::WagTail
-------- rootptr is NULL!
-------- rootptr is NULL!
-------- current root func: IndepFunc
---------- find node: Mammal::Move
read callingfunc
update its callingfunc
write back its callingfunc
-------- rootptr is NULL!
-------- rootptr is NULL!
-------- current root func: Mammal::~Mammal
-------- current root func: Mammal::Move
-------- rootptr is NULL!
```

```
-------- current root func: Mammal::Speak
-------- rootptr is NULL!
-------- rootptr is NULL!
-------- rootptr is NULL!
-------- rootptr is NULL!

Parse C++ source files:

inputfile: ei testsrc Mammal.cpp
 4  Now search class names and virtual func names:

display virtual func name set funclist:
DummyVirtualRoot

Now print BST VirtualFunc_BST from main  :
DummyVirtualRoot

inputfile: ei testsrc Mammal.h
 4  Now search class names and virtual func names:

Class name set size: ------ Mammal
--Find virtual func name: Speak
-- ClassStr:Func = Mammal::Speak

--- setVirtualFlag for this virtual func is set in BrowseBST!

Class name returned: ------ Dog

Class name returned: ------ Cat

Class name returned: ------ BeijingDog

display virtual func name set funclist:
DummyVirtualRoot
Mammal::Speak

Now print BST VirtualFunc_BST from main  :
----Mammal::Speak
DummyVirtualRoot

 4  Now get virtual func names:
-------- current virtual root func: DummyVirtualRoot
DummyVirtualRoot is found, go to subtrees...
-------- virtualrootptr is NULL!
```

```
******** current virtual root func: Mammal::Speak
a BaseVirtualClassFunc is found, extract BaseClassName and BaseVirtualFuncName...

**********************************
Base virtual func:
   BaseVirtualFuncName = Speak
   BaseClassName       = Mammal
**********************************

main is found, go to subtrees...
a ClassFuncName is found in Browser_BST, extract ClassName and FuncName...
  find sFuncToken = Mammal
  find sClassToken      = Mammal
a ClassFuncName is found in Browser_BST, extract ClassName and FuncName...
  find sFuncToken = Dog
  find sClassToken      = Dog
a ClassFuncName is found in Browser_BST, extract ClassName and FuncName...
  find sFuncToken = Cat
  find sClassToken      = Cat
a ClassFuncName is found in Browser_BST, extract ClassName and FuncName...
  find sFuncToken = Yellow
  find sClassToken      = Cat
a ClassFuncName is found in Browser_BST, extract ClassName and FuncName...
  find sFuncToken = BeijingDog
  find sClassToken      = BeijingDog
a ClassFuncName is found in Browser_BST, extract ClassName and FuncName...
  find sFuncToken = BeijingDog
  find sClassToken      = BeijingDog
******** browserbstptr is NULL!
******** browserbstptr is NULL!
a ClassFuncName is found in Browser_BST, extract ClassName and FuncName...
  find sFuncToken = BeijingDog
  find sClassToken      = BeijingDog
a ClassFuncName is found in Browser_BST, extract ClassName and FuncName...
  find sFuncToken = Move
  find sClassToken      = BeijingDog
******** browserbstptr is NULL!
a ClassFuncName is found in Browser_BST, extract ClassName and FuncName...
  find sFuncToken = Speak
  find sClassToken      = BeijingDog
find the same sFuncToken as baseVirtualFuncName!
find different sClassToken from baseVirtualClassName, record it...
  so find a derivedVirtualClassFuncName = BeijingDog::Speak
```

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
derived virtual func:
   DerivedVirtualFuncName = Speak
   DerivedClassName       = BarkingDog

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
OK: baseVirtualClassFuncName found in Browser_BST!
    insert derivedVirtualClassFuncName into VirtualFunc_BST!
--- enVirtualFlag for this virtual func is set in Browser_BST!
-------- browserootptr is NULL!
-------- browserootptr is NULL!
-------- browserootptr is NULL!
-------- browserootptr is NULL!
a ClassFuncName is found in Browser_BST, extract ClassName and FuncName...
  find sFuncToken = Eat
  find sClassToken     = Cat
a ClassFuncName is found in Browser_BST, extract ClassName and FuncName...
  find sFuncToken = Move
  find sClassToken     = Cat
a ClassFuncName is found in Browser_BST, extract ClassName and FuncName...
  find sFuncToken = GetAge
  find sClassToken     = Cat
-------- browserootptr is NULL!
-------- browserootptr is NULL!
a ClassFuncName is found in Browser_BST, extract ClassName and FuncName...
  find sFuncToken = Speak
  find sClassToken     = Cat
find the same sFuncToken as baseVirtualFuncName!
find different sClassToken from baseVirtualClassName, return it...
  so find a derivedVirtualClassFuncName = Cat::Speak

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
derived virtual func:
   DerivedVirtualFuncName = Speak
   DerivedClassName       = Cat

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
OK: baseVirtualClassFuncName found in Browser_BST!
    insert derivedVirtualClassFuncName into VirtualFunc_BST!
--- enVirtualFlag for this virtual func is set in Browser_BST!
a ClassFuncName is found in Browser_BST, extract ClassName and FuncName...
  find sFuncToken = GetAge
  find sClassToken     = Cat
-------- browserootptr is NULL!
```

```
-------- browserootptr is NULL'
-------- browserootptr is NULL'
-------- browserootptr is NULL'
a ClassFuncName is found in Browser_BST, extract ClassName and FuncName...
  find sFuncToken = ~Dog
  find sClassToken      = Dog
a ClassFuncName is found in Browser_BST, extract ClassName and FuncName...
  find sFuncToken = Move
  find sClassToken      = Dog
-------- browserootptr is NULL'
a ClassFuncName is found in Browser_BST, extract ClassName and FuncName...
  find sFuncToken = Speak
  find sClassToken      = Dog
find the same sFuncToken as BaseVirtualFuncName'
find different sClassToken from BaseVirtualClassName, insert it...
  so find a DerivedVirtualClassFuncName   Dog::Speak

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
Derived Virtual Funcs:
  DerivedVirtualFuncName   Speak
  DerivedClassName         Dog

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
>>> BaseVirtualClassFuncName found in Browser_BST'
    insert DerivedVirtualClassFuncName into VirtualFunc_BST'
>>> onVirtualFlag for this virtual func is set in Browser_BST'
-------- browserootptr is NULL'
a ClassFuncName is found in Browser_BST, extract ClassName and FuncName...
  find sFuncToken = WagTail
  find sClassToken      = Dog
-------- browserootptr is NULL'
-------- browserootptr is NULL'
a ClassFuncName is found in Browser_BST, extract ClassName and FuncName...
-------- browserootptr is NULL'
-------- browserootptr is NULL'
a ClassFuncName is found in Browser_BST, extract ClassName and FuncName...
  find sFuncToken = ~Mammal
  find sClassToken      = Mammal
a ClassFuncName is found in Browser_BST, extract ClassName and FuncName...
  find sFuncToken = Move
  find sClassToken      = Mammal
-------- browserootptr is NULL'
a ClassFuncName is found in Browser_BST, extract ClassName and FuncName...
  find sFuncToken = Speak
```

```
  find sClassToken       = Mammal
find the same sFuncToken as BaseVirtualFun::Name'
find the same sClassToken as BaseVirtualClassName, i.more i'...
-------- browserootptr is NULL'
-------- browserootptr is NULL'
-------- browserootptr is NULL'
-------- browserootptr is NULL'
-------- virtualrootptr is NULL'
-------- virtualrootptr is NULL'



Now execute 'penGL program'
Please press 'h' for Help menu'
Number of mem funcs = 23
Number of mem funcs = 3
menuL called'
find new central virtual func: Mammal::Speak
menu called'
find new central func: Mammal::Speak
menu called'
find new central func: Dog::Speak
menu called'
find new central func: IndepFunc
menu called'
find new central func: Cat::Speak
menu called'
find new central func: Dog::WagTail
menu called'
find new central func: Cat::Cat
menu called'
find new central func: IndepFunc
menu called'
find new central func: Dog::WagTail
menu called'
find new central func: main
menu called'
find new central func: BeijingDog::BeatDrBone
menu called'
find new central func: BeijingDog::~BeijingDog
menu called'
find new central func: BeijingDog::Move
menu called'
find new central func: BeijingDog::Speak
```

```
menu called!
find new central func: BeijingDog::BegforBone
menu called!
find new central func: main
menu1 called!
find new central virtual func: DummyVirtualFoo
menu1 called!
find new central virtual func: Mammal::Speak
menu called!
find new central func: Dog::Dig
menu called!
find new central func: BeijingDog::Speak
menu called!
find new central func: Dog::Speak
menu called!
find new central func: Dog::Speak
menu called!
find new central func: Cat::Speak
```