BROWN UNIVERSITY
Department of Computer Science
Master's Thesis
CS-92-M5

"Bat:
A Source-level Debugger for C"

by

Katie Mohrfeld and David Ross

Bat:
A Source-level Debugger for C

Katie Mohrfeld
David Ross

Department of Computer Science
Brown University

Submitted in partial fulfillment of the requirements for the
Degree of Master of Science in the Department of Computer Science
at Brown University

April 1992

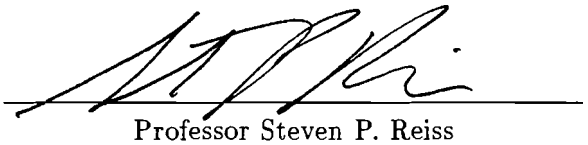
This research project by Katie Mohrfeld is accepted in its present form
by the Department of Computer Science at Brown University
in partial fulfillment of the requirements for the Degree of Master of Science.

Professor Steven P. Reiss
Advisor

4/21/91

Date

This research project by David Ross is accepted in its present form
by the Department of Computer Science at Brown University
in partial fulfillment of the requirements for the Degree of Master of Science.



Professor Steven P. Reiss
Advisor

4/21/92

Date

Contents

List of Figures

1 Introduction

Bat is a source-level debugger for C, similar to Dbx [5] and Gdb [3], which runs on SunOS. It is composed of two tools, a debugger and a graphical user interface, which cooperate via the FIELD message server [9]. The graphical user interface provides visual feedback and mouse input. The user can set and remove breakpoints, step through the execution of a program, evaluate C expressions, catch/ignore signals, and examine and traverse the runtime stack.

After reviewing the main objectives of the project in Section 2, Section 3 describes the internal architecture of the graphical user interface and the debugger. Section 4 discusses how to use the Bat debugger. And, Section 5 closes with some final remarks including what we learned. Appendix A contains a listing of the available resources. Appendix B contains the source code.

2 Objectives

The goals of this project are multifaceted. A primary goal is to gain some insight into the inner workings of the FIELD environment [8]. We are particularly interested in the benefits and logistics of an integration mechanism called *selective broadcasting* [9]. We also want to delve into the software design of visual source-level debuggers with respect to the object-oriented paradigm. Finally, this project gives us the opportunity to manage the complexities of a nontrivial software development project using object-oriented design and programming techniques.

3 Internal Architecture

The Bat debugger consists of two tools: the graphical user interface (Cave) and the debugger (Bat). The FIELD message facility serves as the integration mechanism which allows the two tools to communicate.

3.1 Message facility

The integration framework is based on a simple communications mechanism called *selective broadcasting.* In selective broadcasting, the tools communicate with each other through a central message server. Each tool defines a set of messages that it believes may be of interest to other tools. Other tools register patterns (with the message server) describing the messages that interest them. When a tool sends a message, the message server selectively rebroadcasts the message to those tools who have registered a matching pattern. The FIELD message facility is implemented via Berkeley UNIX sockets.

3.2 Cave

Cave is responsible for translating the user's input into requests to the debugger, Bat. Cave reflects Bat's current state such as the current source file, position, breakpoints, variable values, etc.
Cave broadcasts the following set of messages:

```
CAVE COMMAND RUN <executable> <args...>
CAVE COMMAND CONT
CAVE COMMAND STEP
CAVE COMMAND NEXT
CAVE COMMAND STOP AT <line number> IN <source file>
CAVE COMMAND STOP IN <function name>
CAVE COMMAND STATUS
CAVE COMMAND DELETE <status item number>
CAVE COMMAND WHERE
CAVE COMMAND UP
CAVE COMMAND DOWN
CAVE COMMAND IGNORE <signal number>
CAVE COMMAND CATCH  <signal number>
CAVE COMMAND PRINT <expression>
CAVE COMMAND DISPLAY <expression>
CAVE COMMAND UNDISPLAY <expression>
CAVE COMMAND FILE <source file>
CAVE COMMAND QUIT
```

In addition, Cave registers patterns matching all messages defined by the debugger and responds to each message appropriately.

Cave was designed and implemented as an independent component using object-oriented techniques. To better manage user interface software, separation of interface and application functionality has become an accepted design and software engineering goal. Implementing the user interface as an independent component has several advantages. First, the user interface can be subdivided into components that can be used to create complex objects without detailed knowledge of the underlying implementation. The interface can be quickly modified to be reused in other applications that have similar interaction requirements. The interface can also be changed or replaced with no adverse effects on the application. Separability also encourages prototyping which is a crucial element of user interface design. The interface can be developed in an iterative manner in which successive prototypes are produced until a design is found that satisfies the needs of the application and its users. Object-oriented programming allows one to build the complex systems required to create simple, easy to use user interface environments.

Cave is defined in terms of Motif [2] objects (often called *widgets* for "window widgets") which communicate with the window and operating system. The OSF Motif toolkit, based on the X Window System's [11] Xt Intrinsics [6], offers a collection of interactive objects, such as buttons and scrollbars, and many more sophisticated objects. All of these objects have been designed to enforce a specific user interface policy that is described in the Motif Style Guide [7]. The toolkit itself consists of a class hierarchy comprised of three types of classes: primitive subclasses (labels, buttons, scrollbars, etc.), manager subclasses (control grouping and layout of primitive subclasses), and shell subclasses (handle menus, dialog boxes, and communication with Motif Window Manager). Cave classes are built using primitive and manager subclasses.

Two object models are included to visually show Cave's classes and their relationships to one another. The classes are described below. Rumbaugh's Object Modeling Technique (OMT) notation is used in the diagrams. [4] Figure 1 is an object model illustrating Cave's inheritance tree. Figure 2 is an object model depicting aggregations (or "parts-of") relationships. AnnotMgr, MsgView, etc. are parts of the App object; and AnnotView and

CodeView are parts of the AnnotMgr object.

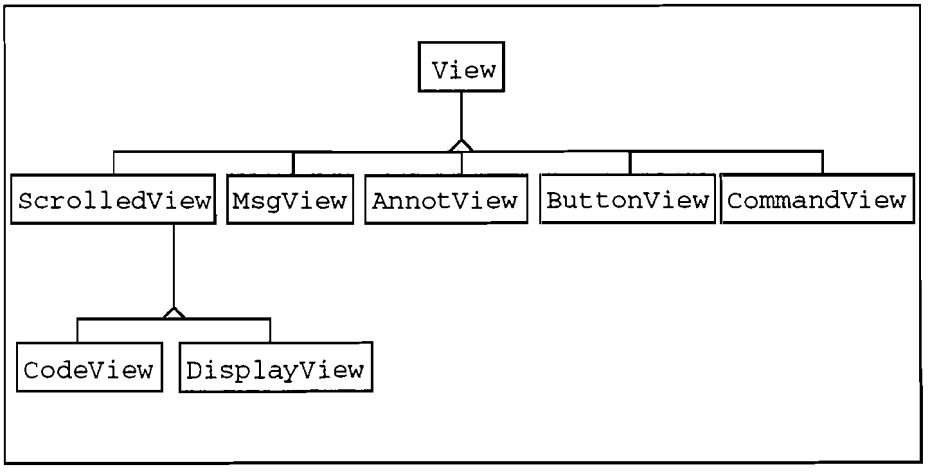


Figure 1: Cave's inheritance tree.

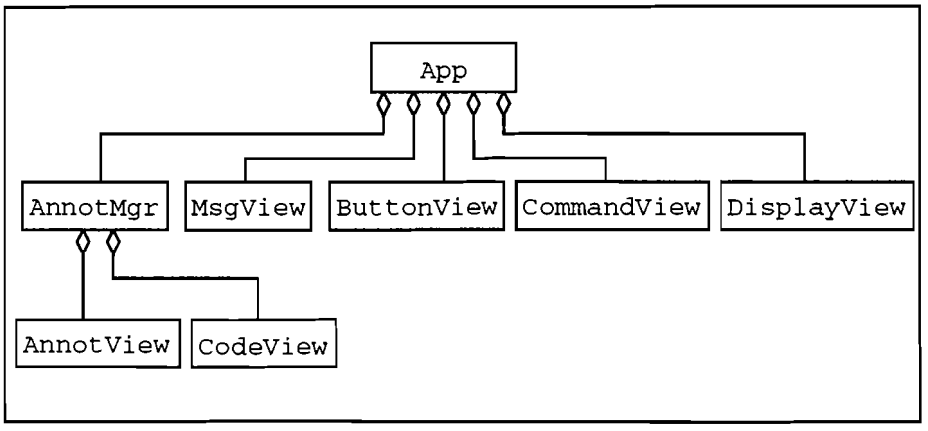


Figure 2: Cave's parts-of relationships.

3.2.1 View

View is an abstract base class that encapsulates features common to all the views in the user interface. Specific views are subclasses of View. These

include: MsgView, ScrolledView AnnotView, CommandView, and ButtonView. Each of these views is composed of one or more Motif widgets. View's primary function is to serve as a template for objects having specific functionality.

3.2.2 ScrolledView

ScrolledView is an abstract class from which scrollable views are subclassed. These include CodeView and DisplayView which are discussed below.

3.2.3 MsgView

A MsgView object registers for the BAT FOCUS message in order to display the current source file and line number of the debuggee. It is composed of a single text widget.

3.2.4 DisplayView

A DisplayView object responds to BAT DISPLAY messages from the Bat debugger. These messages may add a new expression to the display area, remove an expression from the display area, or update the value of an existing expression. DisplayView inherits its ability to scroll from the ScrolledView base class.

3.2.5 ButtonView

A ButtonView object is composed ot a set of Motif pushbutton widgets. It is responsible for translating pushbutton events into command messages to the Bat debugger.

3.2.6 CommandView

A CommandView object is composed of a command line and an output area. The functionality of these areas is controlled by a Motif command widget. The CommandView object translates command line input into command messages and sends them to the debugger. OUTPUT messages from the debugger are echoed in the command area.

3.2.7 AnnotMgr

The AnnotMgr manages an AnnotView and a CodeView object. It is responsible for keeping the two objects in sync during scrolling, FOCUS updates, etc.

CodeView

A CodeView object is responsible for presenting the source code to the user. It is composed of an array of Motif text widgets which are shrinkwrapped in a form widget. In order to allow the user to quickly scroll through a source file, a CodeView object relies on the SFile class for buffered file access. A CodeView object receives messages from the debugger concerning the current source file and line number (the FOCUS). Upon receiving such a FOCUS message, it updates the text widgets to position the appropriate file such that the current line is visible.

AnnotView

An AnnotView object is typically positioned by the AnnotMgr to the left of the CodeView object. It is responsible for representing BAT EVENT ADD and BAT EVENT REMOVE messages from the debugger. It accomplishes this task via graphical markers, annotations, which are associated with a particular location of a source file [10]. The annotation panel is comprised of a drawing area widget and the annotations are pixmaps.

3.2.8 App

App is the application object which instantiates and positions all of the components of the user interface. It is also responsible for initiating communication with the message server.

3.3 Bat

The Bat debugger server is responsible for all interactions with the inferior process. It processes incoming requests and broadcasts events to any clients which have expressed an interest in the inferior process' state.

Bat broadcasts the following set of messages:

```
BAT OUTPUT  <output string>
BAT DISPLAY <display string>
BAT CLEARDISPLAY
BAT EVENT ADD <event> <source file> <function name> <line number>
BAT EVENT REMOVE <event> <source file> <function name> <line number>
   <event> = BREAK | BOMB
BAT FOCUS <source file> <function name> <line number>
```

After studying the internals of Dbx and Gdb, we realized that debuggers in general seem to have a rich set of fundamental data types in common: symbol tables, source files, executables, the run-time stack, the inferior process, commands, breakpoints, and expressions. Therefore, this set of data types make up the set of object classes in the Bat debugger.

3.3.1 Symbol Table

A symbol table object relates an object file or shared library to the source code. Unlike many debuggers, Bat's symbol table objects are lazily evaluated; they only load information when it is needed. Symbol table objects support many operations. For example, query operations are concerned with searching for a source filename and line number given a program counter address, searching for a function object and program counter address given a source filename, searching for a source file given the name of a global variable, etc. Each symbol table object is composed of several types of objects:

Executable
: See section on executable object class.

SourceFile
: Associated with each symbol table is a list of source files making up the object file. See the section on the SourceFile objects class for a more detailed description.

FunctionTab
: A FunctionTab object supports queries for functions by name or by address.

AddrList

AddrList objects are mappings from global variable names to their addresses.

3.3.2 SourceFile

The SourceFile object class represents the lexical semantics of a C source file. It supports queries concerning the semantic contents of .c (source) and .h (include) files. SourceFile objects also maintain a data structure containing the minimal information needed to search for variables, functions, etc. This structure is lazily evaluated, which makes searching very inexpensive. For example, symbols are never loaded unless absolutely necessary. A SourceFile is composed of the following:

Block

Each block object represents a lexical block in a source file. It supports queries concerning variables and types declared within the block, and the start/end line numbers of the block. The lexical block structure of a source file is represented by a tree of these block objects.

FunctionList

This is simply a list of all functions objects declared within the source-file.

3.3.3 Executable

An executable object represents the executable file of the symbol table. It may be an a.out file or a SunOS shared library. This object encapsulates the low level opening and reading of an executable and its symbols and text.

3.3.4 Run-time Stack

The run-time stack object is composed of a list of stack frame objects. Each stack frame represents a frame in the call stack of the process. Many objects in the debugger access the run-time stack object to get information concerning the current stack frame, and to traverse the call stack.

3.3.5 Inferior Process

The process object is responsible for interactions with the inferior process (the debuggee). It has methods for reading/writing data, text, u_area and

registers of the inferior process. For process control, the process object has methods such as attach/detach, step, next, and continue.

3.3.6 Command

There is a class for each command in Bat's language. These objects are all derived from the abstract class, *command*. Thus, Bat's event handler manipulates *command* objects without any knowledge of what type of command it is executing. Command objects are created by the command parser.

3.3.7 Expression

The command language for Bat contains C expressions which must be evaluated with respect to an inferior process's current state. Bat builds an expression tree where each subtree is an expression. Each node in the tree is an object derived from the abstract class, *expression*. Each subclass defines the inherited method, *evaluate*, to evaluate the expression represented by its subtree. Expression trees are built by the command language parser.

4 Using Bat

4.1 Graphical User Interface

Bat's graphical user interface is comprised of three main subwindows: the source/annotation area, command area, and display area.

Message Area
: Displays name of current source file and line number.

SourceWindow
: Displays contents of current source file.

Annotation Panel
: The annotation panel is adjacent to the source window. Currently, the annotation panel displays graphical icons representing the current line number (an arrow), breakpoints (stop signs), and execution errors (bombs).

  The annotation panel provides a simple interface to the breakpoint facility of the debugger. For example, to set a breakpoint, the user can click in the annotation panel next to the appropriate line number.

A stop sign will appear. To remove a breakpoint, click on the stop sign icon and it will disappear.

Command Area

Consists of command buttons, command history/output area, and a command line.

Command Buttons

Provides a list of command buttons which are invoked by clicking the left mouse button. A user can select text in the source window and click on a button. The selected text will be treated as the arguments to the command button.

Command History/Output Area

Commands entered in the command-line are recorded here as well as the command output.

Command-line

Provide typing interface to Bat.

Display Area

Displays specified variables' values dynamically as the inferior process runs.

4.1.1 Resizing

The relative heights of the subwindows can be adjusted by dragging the *control sashes* (small squares near the right of the horizontal borders) with the left mouse button down. The entire application can be resized by adjusting the appropriate window manager's resize mechanism.

4.1.2 Selecting Text

To select some text, press the left mouse button and drag the pointer across the text. The selected text is displayed in reverse video. Alternatively, double-clicking the left mouse button selects a word (bounded by whitespace). Only one word may be selected at a given time.

4.1.3 Scrolling

The source window, command history window, and display window contain scrollbars on the right side of the window. They work as expected except

for the source window scrollbar which simultaneously scrolls the contents of the annotation panel and the source window. Pressing the left mouse button on the top/bottom arrow scrolls the contents backwards/forwards. Dragging the slider scrolls the contents in the appropriate direction. Clicking in the scrolling region above/below the slider will scroll the contents backwards/forwards.

4.2 Command Language

All commands may be typed into the command window. Also, the commands enclosed in a box may be executed by pressing the appropriate command button.

4.2.1 Execution Commands

run [**executable** [args ...]]
Begin program execution.

cont
Continue execution from where the inferior process stopped.

step
Execute one source line, stepping into a function if the current source line contains a function call.

next
Execute one source line, stepping over function calls.

4.2.2 Breakpoint Commands

stop at **line number** [**in** **filename**]
Set a breakpoint at specified line in specified file. If the filename is not specified, then the breakpoint is set in the current file. A *stop sign* annotation will appear in the annotation panel at that line.

stop in **function**
Stop program execution in specified function. A *stop sign* annotation will appear in the annotation panel at the first statement in the function.

delete **status item number**
Remove the breakpoint specified by the item number. Use the *status*

command to see an enumerated listing of current breakpoints. The corresponding annotation item will be deleted from the annotation panel.

status
Print enumerated listing of current breakpoints.

4.2.3 Stack Commands

where
Print the current stack trace.

up
Move up one level in the stack. Updates the source and annotation windows appropriately if source information is available for the stack frame.

down
Move down one level in the stack. Updates the source and annotation windows appropriately if source information is available for the stack frame.

4.2.4 Signal Commands

ignore signal number
Stop catching the signal represented by signal number.

catch signal number
Catch the signal represented by signal number.

4.2.5 Printing and Displaying Commands

The following commands print the result of evaluating expressions. Bat currently supports C expressions involving most primitve and derived data types. Bat supports the following operators: the address of ('&') operator, the contents of ('*') operator, the field reference operator ('.'), and the pointed-to-field reference operator ('->').

print [expression]
Evaluate the expression and print the result.

display [**expression**]
Display the result of evaluating the expression in the display window whenever execution stops. Multiple expressions may be displayed at once.

undisplay [**expression**]
Stop displaying the expression in the display window.

4.2.6 Miscellaneous Commands

file filename
Change the current source file to filename. Updates the source window.

quit
Exit Bat.

5 Conclusions

Designing and implementing the Bat debugger has given us the opportunity to learn through experience. The project required that we develop two tools, a debugger server and a graphical user interface client, that communicate via selective broadcasting. The completed project consists of 17,000 lines of C++ code. The graphical user interface was implemented using Motif. This experience has taught us some valuable lessons.

5.1 C++ Learning Curve

We underestimated the time needed to become proficient in C++.

> Learning the fundamentals of the programming language is one thing: learning how to design and write *effective* programs in that language is something else entirely. This is especially true of C++, a language that boasts an uncommon range of power and expressiveness....The trick, then, is to discover those aspects of C++ that are likely to trip you up, and to learn how to avoid them [12].

5.2 Debuggers are Naturally Object-Oriented

Perhaps our biggest lesson is that debuggers are complicated! An object-oriented perspective serves to cut through this complexity. By abstracting

out the fundamental objects, the problem quickly becomes more manageable. Bat's fundamental objects include symbol tables, source files, executables, the run-time stack, the inferior process, commands, breakpoints, and expressions.

5.3 Object-Oriented Software Development is an Iterative Process

Object-oriented software development is clearly an iterative process. If this were a commercial project, now would be the time to pause and reconsider the design of the system. Given the opportunity to iterate again, we would definately do some things differently.

5.4 Object-Oriented Design in the Real World

Faced with a nontrivial software development project and an ambitious schedule (3 months), we quickly learned that one can only take academic ideology so far. For example, there is no need to force the use of inheritance on an application. Deep inheritance trees facilitate reusability and extensibility, while increasing complexity [1]. A project can benefit greatly from the object-oriented paradigm just by making use of object-based programming techniques. Inheritance requires a great deal of forethought.

5.5 Integration via Message Passing

Typically, the integration phase is a time consuming and stressful experience for a development team. The FIELD message facility allows members of a team to develop software components independently and then integrate them easily and quickly. Our experience with this integration technique was very positive; we integrated our two tools in a matter of hours. In addition, the architecture facilitates extensibility and prototyping.

References

[1] S. R. Chidamer and C. F. Kemerer, "Towards a Metric Suite for Object Oriented Design," in *Proceedings of OOPSLA '91,* published as *SIGPLAN Notices*, May 1991, pp. 197–211.

[2] Dan Heller, *Motif Programming Manual*, O'Reilly and Associates, Inc., Sebastopol, CA, 1991.

[3] Free Software Foundation, Inc., *Working in GDB*, 1991.

[4] James Rumbaugh and others, *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, NJ, 1991.

[5] M. A. Linton, "The Evolution of Dbx," in *USENIX Summer Conference*, June 1990, pp. 211–220.

[6] Massachusetts Institute of Technology and Digital Equipment Corporation, *X Toolkit Intrinsics - C Language Interface*, 1991.

[7] Open Software Foundation, *OSF/Motif Programmer's Guide*, 1989.

[8] S. P. Reiss, "Interacting with the FIELD Environment," Technical Report CS-89-51, Brown University, 1989.

[9] S. P. Reiss, "Connecting Tools Using Message Passing in the Field Environment," *IEEE Software*, Vol. 7, No. 4, July 1990, pp. 57–66.

[10] S. P. Reiss, "On the Use of Annotations for Integrating the Source in a Program Development Environment," Technical Report CS-91-32, Brown University, 1991.

[11] R. W. Scheifler and J. Gettys, *X Window System*, 2nd ed., Digital Press, 1990.

[12] Scott Meyers, *Effective C++*, Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1992.

A Resources

The default resources may be set in the file *bat.xrdb*. The names of the widgets are:

cave
ApplicationShell widget containing all the subwindows.

ap_app_mgr
XmPanedWindow widget managing all of the pane subwindows.

ap_msg_source_mgr
XmForm widget constraining message and source area.

mv_text_w
XmText widget displaying message.

ap_source_mgr
XmForm widget constraining AnnotView and CodeView.

AnnotView
XmScrolledWindow widget displaying annot.

annot
XmDrawingArea widget displaying the annotation panel.

CodeView
XmScrolledWindow widget displaying source code.

mainform
XmForm widget constraining text widgets (fields) for source code.

cv_vsb,cv_hsb
XmScrollBar widgets for CodeView.

ap_command_mgr
XmForm widget constraining command area widgets.

ap_button_mgr
XmForm widget constraining command buttons.

RUN...QUIT
XmPushButton widgets in command area.

cd_command_w
XmCommand widget for command area.

DisplayView
XmScrolledWindow widget for display area.

Example of resource file:

```
! Bat resources

*foreground:                    DarkOrange
*allowShellResize:              true
*borderWidth:                   0
*highlightThickness:            2
*traversalOn:                   true
*keyboardFocusPolicy:           explicit
*menuAccelerator:               <Key>KP_F2

!! cave (xmfonts)
cave*CodeView.fontList:                 9x15
cave*ap_app_mgr.height:         1400
cave*ap_app_mgr.width:          1000
cave*ap_source_mgr.height:      400
cave*ap_source_mgr.width:       700
cave*annot.height:              400
cave*annot.width:               700
cave*mv_text_w.foreground:      white

!! Buttons
cave*RUN.bottomShadowColor:     green
cave*STOP.bottomShadowColor:    red
*XmPushButton.fontList:         8x13

*XmDrawingArea.borderWidth:     0

*XmText.fontList:               9x15
*XmText.borderColor:            DarkOrange
*XmText.borderWidth:            0
```

B Source Code

The code consists of the following source files:

- as.h
- bat.h
- block.h
- breakpoint.h
- command.h
- data.h
- executable.h
- expression.h
- function.h
- gendefs.h
- gui.h
- language.h
- mreg.h
- preamble.h
- proc.h
- sfile.h
- sourcefile.h
- stack.h
- strcache.h
- symtab.h

- target.h
- type.h
- variable.h
- AnnotMgr.C
- AnnotView.C
- App.C
- ButtonView.C
- CodeView.C
- CommandView.C
- DisplayView.C
- MsgView.C
- ScrolledView.C
- View.C
- as.C
- bat.C
- bp.C
- command.C
- data.C
- executable.C
- expression.C
- function.C
- function_map.C
- gui.C
- proc.C

- sfile.C
- sharedlib.C
- sourcefile.C
- stack.C
- strcache.C
- symtab.C
- symtab_misc.C
- symtab_parse.C
- target.C
- type.C
- variable.C
- commands.l
- parser.y

```
/*-----------------------------------------------------------------
 |       FILE: gui.h
 |    PURPOSE: Class declarations for cave gui.
 |      NOTES:
  -----------------------------------------------------------------*/

#include <iostream.h>
#include <strstream.h>
#include <stddef.h>
#include <X11/Intrinsic.h>
#include <X11/Xlib.h>
#include <X11/StringDefs.h>
#include <Xm/Xm.h>
#include <Xm/ScrolledW.h>
#include <Xm/ScrollBar.h>
#include <Xm/Text.h>
#include <Xm/Form.h>
#include <Xm/PanedW.h>
#include <Xm/DrawingA.h>
#include <Xm/RowColumn.h>
#include <Xm/PushB.h>
#include <Xm/Command.h>
#include <Xm/List.h>
#include <ctype.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "Model.h"
#include "sfile.h"
#include "msg.h"
#include "cmpx.h"


#ifdef max  /* just in case--we don't know, but these are commonly set */
   def max  /* by arbitrary unix systems.  Also, we cast to int! */
  dif
/* redefine "max" and "min" macros to take into account "unsigned" values */
#define max(a,b) ((int)(a)>(int)(b)?(int)(a):(int)(b))
#define min(a,b) ((int)(a)<(int)(b)?(int)(a):(int)(b))


#define NBUTTONS        11
#define NLinesInFile    71
#define MaxColsInFile   80
// Annotation panel width is hard-coded and may not be changed
#define ANNOT_WIDTH     60
#define DS_HEIGHT       80

// CodeView state flags
#define CV_SCROLL_CB    1
#define CV_RESIZED      2
// fillfieldswithtext
#define CV_REFRESH      4
// read in new file and reset scrollbars
#define CV_CHANGED_FILE 8
//#define ?             16

// AnnotView state flags
#define AV_RESIZED      1
// fillfieldswithtext
#define AV_REFRESH      2
// read in new file and reset scrollbars
#define AV_CHANGED_FILE 4

   fine FOCUS_ANNOT_COL 45
   fine STOP_ANNOT_COL  25
#define BOMB_ANNOT_COL  5

// Annotation masks
#define AV_STOP         1
#define AV_FOCUS        2
#define AV_BOMB         4
```

```
class AnnotView;
class CodeView;
class CommandView;
class ButtonView;
class DisplayView;
class MsgView;
class ScrolledView;
class View;
class AnnotMgr;

void widgetmap(Widget, int);


/*-------------------------------------------------------------------------
 |      CLASS: App
 |    PURPOSE: Application object instantiates all the parts of the gui
 |             and starts up message server.
 -------------------------------------------------------------------------*/
class App
{
  public:
    // Constructors
    App( Cardinal argc, char **argv );

    // Destructor
    ~App() {cout << "App destructing!" << endl;}

    static App         *getCurrentApp()   { return ap_current_app;     }
    ButtonView         *getButtonView()   { return ap_ptrButtonView;   }
    XtAppContext        getAppContext()   { return ap_app_context;    }
    CommandView        *getCommandView()  { return ap_ptrCommandView;  }
    DisplayView        *getDisplayView()  { return ap_ptrDisplayView;  }
    MsgView            *getFileMsgView()  { return ap_ptrFileMsgView;  }
    AnnotMgr           *getAnnotMgr()     { return ap_ptrAnnotMgr;     }
    static MSG_HANDLE   getMsgHandle()    { return ap_msg_handle;      }
    static Widget       getAppWidget()    { return ap_toplevel;        }

  protected:
    static Widget               ap_toplevel;     // Application shell
    Widget                      ap_app_mgr;      // Paned window manager bonding
                                                 // source_mgr, command_mgr,
                                                 // OutputView, and DisplayView
    Widget                      ap_command_mgr;  // Form manager managing buttons
                                                 // and command area
    Widget                      ap_button_mgr;   // Form manager managing command
                                                 // buttons
    XtAppContext                ap_app_context;  // Application context
    Widget                      ap_source_mgr;   // Form manager bonding CodeView
                                                 // and AnnotView
    Widget                      ap_msg_source_mgr; // Form manager bonding file
                                                   // message area and source area
    AnnotMgr*                   ap_ptrAnnotMgr;  // Manages CodeView and AnnotView
    ButtonView*                 ap_ptrButtonView;
    CommandView*                ap_ptrCommandView;
    DisplayView*                ap_ptrDisplayView;
    MsgView*                    ap_ptrFileMsgView;
    MsgView*                    ap_ptrStatMsgView;
    static App                 *ap_current_app;
    static MSG_HANDLE           ap_msg_handle;
};

/*-------------------------------------------------------------------------
 |      CLASS: AnnotMgr
 | SUPERCLASS:
 |    PURPOSE: Manages AnnotView and CodeView objects
 -------------------------------------------------------------------------*/
class AnnotMgr
{
  friend void scrolledCB( Widget scrollbar, XtPointer o, XtPointer c );
  friend void resizeCB( CompositeWidget widget , XtPointer user_data, XEvent *e, Boolean  *);
```

```
  public:

  // Constructors
  nnotMgr( Widget src_mgr );           // pass in form widget

  // Destructor
  ~AnnotMgr();
  static void FocusUpdateCB( int line_num, const char* src_file_name, void *user_data );

  AnnotView *getAnnotView()     { return ptrAnnotView; }
  CodeView  *getCodeView()      { return ptrCodeView; }

  protected:
  CodeView*             ptrCodeView;
  AnnotView*            ptrAnnotView;
  };


/*-------------------------------------------------------------------------
 |      CLASS: View
 |    PURPOSE: Abstract base class for various view objects.
 -------------------------------------------------------------------------*/
class View
{
  public:
  // Constructors
  View( Widget& ancestor );

  // Destructor
  ~View() {cout << "View destructing!" << endl;}

  protected:
  Widget                        *parent;
  Widget                        *widget;
   idget                        scrolled_w;             // scrolled window widget
  .idget                        mainform;               // Form widget holds
                                                        // text lines
  Widget                        annot;                  // Drawing area widget
                                                        // for annotations
  Widget                        work_w;                 // Text widget for DisplayView
  static Widget                 *annot_scrolled;        // so CodeView knows what to attach itse
  };

/*-------------------------------------------------------------------------
 |      CLASS: ScrolledView
 | SUPERCLASS: View
 |    PURPOSE: Abstract base class for Scrolled view objects.
 -------------------------------------------------------------------------*/
class ScrolledView : public View
{
  public:
  // Constructors
  ScrolledView ( Widget& ancestor, char *name );

  // Destructor
  ~ScrolledView ();
  };

/*-------------------------------------------------------------------------
 |      CLASS: CodeView
 | SUPERCLASS: ScrolledView
 |    PURPOSE: Data and methods for source code object
 -------------------------------------------------------------------------*/
class CodeView : public ScrolledView

  ublic:
  // Constructors
  CodeView( Widget& ancestor );

  // Destructor
  ~CodeView();
```

```
  void  SetResized()     { cv_state |= CV_RESIZED; }
  int   GetState()       { return cv_state; }
  int   GetNumFields()   { return cv_num_fields; }
  int   GetLineHeight()  { return cv_field_height; }

  void  SetFocus( int focus, const char* name );
  int   GetFocus( int& focus, const char*& name )
    { focus= cv_window_focus; name = cv_current_file->getName(); }

  void  GetMainformSize( Dimension& w, Dimension& h );
  void  SetMainformSize( Dimension w, Dimension h );

  void  FillFieldsWithText();

  void  GetFileCoords ( int& line_num, int& col_num )
    { line_num = cv_top_file_line_num; col_num=cv_left_file_col_num;}
  //  Get (line#, col#) of line in file that's at top of view window
  void  SetFileCoords (int line_num, int col_num)
    { cv_top_file_line_num=line_num; cv_left_file_col_num = col_num; }

  void getSelected( char*& text, const char*& fname, int& f_lnum )
    { text = cv_selected; fname = cv_current_file->getName();
      f_lnum = cv_selected_file_lnum; }

  void  Update ();                //  Update "image" in CodeView window

  // Callbacks
  static void beforeModifiedCB( Widget text_w, XtPointer, XtPointer);
  static void motionVerifyCB( Widget text_w, XtPointer, XtPointer);
  void SetScrollbarCBfunc( XtCallbackProc cb, XtPointer );
  void SetResizeCBfunc( XtEventHandler cb, XtPointer user_data );

  protected:
  int CalcNumFields( int view_height, int char_height );
  Widget                cv_vsb;                 //scrollbar widget
  XtActionsRec          cv_actions_rec;
  XColor                color, unused;
  Colormap              cmap;
  Widget                *fields;                // Array of one-line text widgets
  int                   cv_num_fields;
  int                   cv_window_focus;
  Dimension             cv_mainform_height, cv_mainform_width;
  int                   cv_state;               // flag
  static SFile          *cv_current_file;       // the source file object
  static int            cv_top_file_line_num;   // current line number in file
  static int            cv_left_file_col_num;   // current column number
  static char           cv_selected[100};       // selected text
  int                   cv_selected_file_lnum;  // file line number of selected text

  private:
  Dimension             cv_field_height, cv_field_width;
  static void scrolledHCB( Widget scrollbar, XtPointer, XtPointer);

  };


/*-------------------------------------------------------------------------
  |      CLASS: AnnotView
  | SUPERCLASS: View
  |    PURPOSE: Object for annotation panel.
  -------------------------------------------------------------------------*/
class AnnotView : public View
{
  public:
  // Constructors
  AnnotView ( Widget& ancestor );
  // Destructor
  ~AnnotView ();

  void  SetFocus( int focus, const char* name );
  void  GetFocus( int& focus, const char*& name )
    { focus= av_window_focus; name = av_current_file->getName(); }
```

```
  void GetFileCoords (int& line_num, int& col_num)
    { line_num = av_top_file_line_num; col_num=av_left_file_col_num;}
  '/  Get (line#, col#) of line in file that's at top of view window
   oid SetFileCoords (int line_num, int col_num)
    { av_top_file_line_num=line_num; av_left_file_col_num = col_num; }

  void FillAnnotWithAnnots();
  void Update ();                  //  Update "image" in CodeView window
  void SetLineHeight( int height ) { av_field_height = height; }

  void GetAnnotSize( Dimension& w, Dimension& h);
  void SetAnnotSize( Dimension w, Dimension h );

  void SetNumAnnots( int num ) { av_num_annots = num; }

  void addAnnotation( int annot_type, const char *file, const char *func, int line_number );
  void removeAnnotation( int annot_type, const char *file, const char *func, int line_number );

  void drawAnnot( int annot_type, bool on, int line_number );

  protected:
  static void drawing_areaCB( Widget draw_w, XtPointer, XtPointer);

  int                   av_window_focus;
  int                   av_state;               // flag
  static SFile          *av_current_file;       // the source file object
  static int            av_top_file_line_num;   // current line number in file
  static int            av_left_file_col_num;   // current column number
  Dimension             av_annot_height, av_annot_width;
  Pixmap                av_pixmap;              // for redrawing annotview
  int                   av_num_annots;          // number of annotations

  private:
   nt                   av_field_height;
  GC                    gc;                     // graphics context

  void  setColor( Widget widget, String color );
};

/*-------------------------------------------------------------------------
 |      CLASS: DisplayView
 | SUPERCLASS: ScrolledView
 |    PURPOSE: Object class for area that displays variable values.
 -------------------------------------------------------------------------*/
class DisplayView : public ScrolledView
{
 public:
   DisplayView( Widget& ancestor );
   ~DisplayView();

  void Clear();
  void AddTextToDisplay( char *text );

  protected:
  static XmTextPosition         ds_wpr_position;  // WHY IS THIS STATIC?
  Widget                        ds_vsb;           //scrollbar widgets
};


/*-------------------------------------------------------------------------
 |      Class: ButtonView
 | SUPERCLASS: View
 |    PURPOSE: Class for push buttons.
  -------------------------------------------------------------------------*/
  .ss ButtonView : public View
{
  public:
  ButtonView ( Widget& ancestor );
  ~ButtonView ();

  protected:
```

```
  //  Selected commands.  User-configurable via resource file.
  Widget                        *cd_buttons;    // Array of button widgets
  static void buttonCB( Widget button_w, XtPointer, XtPointer);
};

/*-------------------------------------------------------------------------
 |      CLASS: CommandView
 | SUPERCLASS: View
 |    PURPOSE: Class for command and history area
 -------------------------------------------------------------------------*/
class CommandView : public View
{
  public:
  CommandView ( Widget& ancestor, Widget& top_widget );
  ~CommandView ();
  void AddTextToOutput( char *text );

  protected:
  static void ExecCmdCB( Widget cmd_widget, XtPointer, XtPointer );
  void AddCommandToOutput( char *text );
  //  Selected commands.  User-configurable via resource file.
  Widget                        cd_command_w;   // One-line text widget
                                                // where user enters command
  Widget                        cd_output_w;    // Scrolled list widget
  Widget                        cd_text_w;      // Text widget

  private:
  void MakeItemVisible( Widget list_w, XmString item );
};

/*-------------------------------------------------------------------------
 |      CLASS: MsgView
 | SUPERCLASS: View
 |    PURPOSE: Class for message area.
 -------------------------------------------------------------------------*/
class MsgView : public View
{
 public:
   MsgView ( Widget& ancestor );
   ~MsgView();
   Widget& getWidget( ) { return mv_text_w; }

 protected:
   Widget  mv_text_w;  // text field widget
};
```

```
/*-----------------------------------------------------------------
 |       FILE: gui.C
 |    PURPOSE: Main entrpoint for the cave gui.
  -----------------------------------------------------------------*/
#include "gui.h"


main( int argc, char **argv )
{
  Cardinal n = argc;

  // Construct instance of App object
  App bat_app( n, argv);
}
```

```
/*-------------------------------------------------------------------
 |       FILE: AnnotMgr.C
 |    PURPOSE: Manages CodeView and AnnotView
  -------------------------------------------------------------------*/

#include "gui.h"

/*-------------------------------------------------------------------
 |     METHOD: focusHandler
 |
 |    PURPOSE: Handles BAT FOCUS messages.
 -------------------------------------------------------------------*/
static void
focusHandler( char *file, char *function, int line_number )
{
   App      *AP = App::getCurrentApp();
   AnnotMgr *AM = AP->getAnnotMgr();

   AM->FocusUpdateCB( line_number, file, (void *)AM );
}

typedef void (*FOCUSfptr)(...);


/*-------------------------------------------------------------------
 |     METHOD: AnnotMgr
 |
 |    PURPOSE: Constructor
 | PARAMETERS:  Widget  src_mgr;        Form manager widget
 -------------------------------------------------------------------*/
AnnotMgr::AnnotMgr( Widget src_mgr )
{
  int                   ln, cn;

   / Create AnnotView BEFORE CodeView
  ptrAnnotView = new AnnotView( src_mgr );

  // Create CodeView AFTER AnnotView
  ptrCodeView = new CodeView( src_mgr );

  // Set codeview vertical scrollbar callback function.
  // Callback is registered with CodeView object but the callback
  // routine belongs to the AnnotMgr object.  This allows AnnotMgr
  // to have control over the callback, thus it can scroll both
  // CodeView and AnnotView.
  // A pointer to callback function and a pointer to "this" (AnnotMgr)
  // object are passed in.  "This" must be passed in as user data so
  // that it is registered with the scrollbar callback and then can
  // be used in the callback routine to access the appropriate CodeView
  // and AnnotView methods and instance variables.

  ptrCodeView->SetScrollbarCBfunc( (XtCallbackProc)scrolledCB, (XtPointer)this);

  // Set add resize event to CodeView->mainform translation table.
  ptrCodeView->SetResizeCBfunc( (XtEventHandler)resizeCB, (XtPointer)this);

  Dimension h = ptrCodeView->GetLineHeight();
  ptrAnnotView->SetLineHeight( h );

  ptrAnnotView->SetNumAnnots( ptrCodeView->GetNumFields() );

  ptrCodeView->GetFileCoords( ln, cn );
  ptrAnnotView->SetFileCoords( ln, cn );

  '/ register focus handler with the mesg server
  ISGregister( App::getMsgHandle(), "BAT FOCUS %ls %2s %3d", (FOCUSfptr)focusHandler, 3, NULL );
}



/*-------------------------------------------------------------------
 |     METHOD: ~AnnotMgr
```

```
 |
 |    PURPOSE: Destructor
 ------------------------------------------------------------------------*/
AnnotMgr::~AnnotMgr( )
{
}

/*-------------------------------------------------------------------------
 |     METHOD: scrolledCB
 |
 |    PURPOSE: React to scrolling actions.  Reset position of Scrollbars;
 |             call redraw() to do actual scrolling.  cbs.value is
 |             Scrollbar's new position.
 ------------------------------------------------------------------------*/
void scrolledCB(Widget scrollbar, XtPointer user_data, XtPointer c )
{
  AnnotMgr                      *AM = (AnnotMgr *)user_data;
  CodeView                      *CV = AM->ptrCodeView;
  AnnotView                     *AV = AM->ptrAnnotView;
  XmScrollBarCallbackStruct     *cbs=(XmScrollBarCallbackStruct *)c;
  int                           line_num, col_num;


  CV->GetFileCoords( line_num, col_num );
  CV->SetFileCoords( cbs->value, col_num );

  AV->GetFileCoords( line_num, col_num );
  AV->SetFileCoords( cbs->value, col_num );

  CV->FillFieldsWithText();

  AV->Update();
}



/*-------------------------------------------------------------------------
 |     METHOD: resizeCB
 |
 |    PURPOSE: Callback for resize events.
 ------------------------------------------------------------------------*/
void resizeCB( CompositeWidget widget , XtPointer user_data,
                              XEvent *event, Boolean *continue_to_dispatch)
{
  AnnotMgr                      *AM = (AnnotMgr *)user_data;
  CodeView                      *CV = AM->ptrCodeView;
  AnnotView                     *AV = AM->ptrAnnotView;
  Dimension                           h, w;


  // Get previously saved height and width
  CV->GetMainformSize( w, h);

  switch (event->type)
    {
    // resize event
    case ConfigureNotify:
    if ( (event->xconfigure.width == w) &&
        (event->xconfigure.height == h) )
      { // Not resized
      }
    else
      { // Resized
      // Save new height and width
      CV->SetMainformSize( w, h );
      // Set resized flag
      CV->SetResized();
      // Update fields
      CV->Update();
      // Now tell AnnotView to resize itself

      AV->SetNumAnnots( CV->GetNumFields() );
```

```
      AV->Update();
      }
    break;
    }

  }

/*-----------------------------------------------------------------------
 |     METHOD: FocusUpdateCB
 |
 |    PURPOSE: Update forcus of both annotview and codeview objects.
 -----------------------------------------------------------------------*/
void AnnotMgr::FocusUpdateCB( int line_num, const char* src_file_name, void *user_data )
{
  AnnotMgr                      *AM = (AnnotMgr *)user_data;
  CodeView                      *CV = AM->ptrCodeView;
  AnnotView                     *AV = AM->ptrAnnotView;
  int                           code_ln, code_cn;

  // set focus for both av and cv
  CV->SetFocus( line_num, src_file_name );
  AV->SetFocus( line_num, src_file_name );

  CV->GetFileCoords( code_ln, code_cn );
  AV->SetFileCoords( code_ln, code_cn );

  CV->Update();
  AV->Update();
}
```

```
/*-----------------------------------------------------------------------
 |       FILE: AnnotView.C
 |    PURPOSE: Object class which handles annotation panel.
 |
 |   CONTENTS: -methods-        -------------purpose-----------------
 |             AnnotView        Constructs annotation panel
 |             ~AnnotView       Destructor
 |             annotAddHandler  Adds event (annotation) to specified
 |                              file
 |             annotRemoveHandler       Removes event (annotation)
 |                                      from specified file
 |             drawAnnot        Draws/erases annotation from file
 |             GetAnnotSize
 |             SetAnnotSize
 |             drawing_areaCB
 |             addAnnotation    Calls drawAnnot to add annotation
 |             removeAnnotation Calls drawAnnot to remove annotation
 |             FillAnnotWithAnnots
 |             SetFocus         Sets current filename to be viewed
 |                              in AnnotView and sets the focus
 |             Update           Update "image" in AnnotView
 |             setColor Sets global gc.
 |
 |
 |    NOTES:
 | ** All MSG and SFile calls deal with 1-based line numbers.
 | ** av_top_file_line_num is also 1-based.y
 | ** Everything else is 0-based.
 -----------------------------------------------------------------------*/

#include "gui.h"

static void
annotAddHandler( char *event, char *file, char *function, int line_number, AnnotView *AV );
static void
annotRemoveHandler( char *event, char *file, char *function, int line_number, AnnotView *AV );

typedef void (*ANNOTATIONfptr)(...);

/*-----------------------------------------------------------------------
 |     METHOD: AnnotView
 |
 |    PURPOSE: Constructor
 -----------------------------------------------------------------------*/
AnnotView::AnnotView( Widget& ancestor ):
                      View( ancestor )
{
  Dimension w,h;

  av_current_file = NULL;

  // create the scrolled drawing area, etc...
  scrolled_w = XtVaCreateWidget( "AnnotView",
                                 xmScrolledWindowWidgetClass, ancestor,
                                 XmNshadowThickness, 3,
                                 NULL );

  // create draw widget
  annot = XtVaCreateWidget( "annot",
                            xmDrawingAreaWidgetClass, scrolled_w,
                            NULL );

  XtVaGetValues( ancestor,
                 XmNheight, &h,
                 NULL );

  XtVaSetValues( annot,
                 XmNheight,  h,
                 XmNwidth,  ANNOT_WIDTH,
                 NULL );

  // Set saved size
```

```
  SetAnnotSize( ANNOT_WIDTH, h );
  GetAnnotSize( w, h );

  XtAddCallback(annot, "inputCallback",
                  (XtCallbackProc)AnnotView::drawing_areaCB,
                  (XtPointer)this);
  XtAddCallback( annot, "exposeCallback",
                  (XtCallbackProc)AnnotView::drawing_areaCB,
                  (XtPointer)this);
  //Create gc intially
  gc = XCreateGC(XtDisplay(annot),
                 RootWindowOfScreen(XtScreen(annot)), 0, NULL);
  XtVaSetValues(annot, XmNuserData, gc, NULL);

  // create a pixmap the same size as annotation area
  av_pixmap = XCreatePixmap(XtDisplay(annot),
                        RootWindowOfScreen(XtScreen(annot)), w, h,
                        DefaultDepthOfScreen(XtScreen(annot)));

  setColor(annot,"Black");
  // clear pixmap with black
  XFillRectangle(XtDisplay(annot), av_pixmap, gc, 0, 0, w, h);

  // scarf away widget so CodeView knows what to attach itself to
  annot_scrolled = &scrolled_w;

  XtVaSetValues( scrolled_w,
                XmNleftAttachment,      XmATTACH_FORM,
                XmNtopAttachment,       XmATTACH_SELF,
                XmNbottomAttachment,    XmATTACH_FORM,
                NULL );

  // Must specify orientation when creating scrollbars or else sizes
  // get screwed up

  // Allow scrolled window to initialize itself accordingly
  XmScrolledWindowSetAreas(scrolled_w, NULL, NULL, annot);

  XtManageChild(annot);
  XtManageChild(scrolled_w);

  // register focus handler with the mesg server
  int def_args[5];
  def_args[0] = NULL;
  def_args[1] = NULL;
  def_args[2] = NULL;
  def_args[3] = 0;
  def_args[4] = (int)this;
  MSGregister( App::getMsgHandle(), "BAT EVENT ADD %1s %2s %3s %4d",
              (ANNOTATIONfptr)annotAddHandler, 5, (void**)def_args );

  // register focus handler with the mesg server
  MSGregister( App::getMsgHandle(), "BAT EVENT REMOVE %1s %2s %3s %4d",
              (ANNOTATIONfptr)annotRemoveHandler, 5, (void**)def_args );
}


/*-----------------------------------------------------------------------
 |     METHOD: ~AnnotView
 |
 |    PURPOSE: Destructor...currently does nothing.
 -----------------------------------------------------------------------*/

AnnotView::~AnnotView( )
{
}

/*-----------------------------------------------------------------------
 |     METHOD: annotAddHandler
 |
 |    PURPOSE: To add the event (annotation) at the specified file,
 |             function, or line.  Called by the message server.
```

```
 |      NOTES: Currently just handles breakpoints and bombs.
 ------------------------------------------------------------------------*/
static void
   otAddHandler( char *event, char *file, char *function, int line_number, AnnotView *AV )

   int annot_type;
   if( !strcmp( event, "BREAK" ) ) annot_type = AV_STOP;
   else if( !strcmp( event, "BOMB" ) ) annot_type = AV_BOMB;

   AV->addAnnotation( annot_type, file, function, line_number );
}


/*-----------------------------------------------------------------------
 |     METHOD: annotRemoveHandler
 |
 |    PURPOSE: To remove the event (annotation) from the specified file,
 |             function, or line.  Called by the message server.
 |      NOTES: Currently just handles breakpoints and bombs.
 ------------------------------------------------------------------------*/
static void
annotRemoveHandler( char *event, char *file, char *function, int line_number, AnnotView *AV )
{
   int annot_type;
   if( !strcmp( event, "BREAK" ) ) annot_type = AV_STOP;
   else if( !strcmp( event, "BOMB" ) ) annot_type = AV_BOMB;

   AV->removeAnnotation( annot_type, file, function, line_number );
}


/*-----------------------------------------------------------------------
 |     METHOD: drawAnnot

      PURPOSE: Either draws or erases an annotation from the file line
 |            number.  If on==TRUE, then it draws the annotation,
 |            otherwise it erases it.
 |      NOTES: This method also updates the sfile accordingly.
 |            Also, note the lnum is the line number in the file (1-based),
 |            as opposed to the line number on the screen (0-based)!
 ------------------------------------------------------------------------*/
void
AnnotView::drawAnnot( int annot_type, bool on, int file_lnum )
{
  int                   x_pos, y_pos;   // upper left corner of dest rectangle
  int                   window_lnum;  // window line number
  Pixmap                bitmap;
  unsigned int          width, height;
  Dimension             w,h;
  unsigned              bitmap_error;
  char                  *icon;

  if( (annot_type!=AV_BOMB) && (annot_type != AV_STOP) && (annot_type != AV_FOCUS) | av_current_
    {
    return;
    }

  // Get the screen linenumber of the file line number, file_lnum
  window_lnum = file_lnum - av_top_file_line_num;
  y_pos = window_lnum * av_field_height;

  if( annot_type == AV_STOP | annot_type==AV_BOMB )
    {
    int mask = av_current_file->getAnnotations( file_lnum );

    x_pos = (annot_type==AV_STOP) ? STOP_ANNOT_COL : BOMB_ANNOT_COL;

    // ok...either draw annotation or erase it depending on "on"
    if( on )
      {
      // off so toggle on
      mask |= annot_type;
```

```
      icon = (annot_type==AV_STOP) ? "/u/khm/project/src/bitmaps/stop"
                                   : "/u/khm/project/src/bitmaps/bomb";
      setColor(annot,"Red");
      }
    else
      {
      // on so toggle off
      mask &= ~annot_type;
      icon = "/u/khm/project/src/bitmaps/clear";
      setColor(annot,"Black");
      }
    av_current_file->setAnnotations( file_lnum, mask  );

    // if line is not currently on the screen, then you are done!
    if( window_lnum < 0 || window_lnum > av_num_annots-1 )
      return;
    }
  else if (annot_type == AV_FOCUS)
    {
    icon = "/u/khm/project/src/bitmaps/focus";
    x_pos = FOCUS_ANNOT_COL;
    setColor( annot, "Green" );
    }

  // load bitmap
  if ((bitmap_error = XReadBitmapFile( XtDisplay(annot),
                                       DefaultRootWindow(XtDisplay(annot)),
                                       icon, &width, &height, &bitmap,
                                       0, 0)) == BitmapSuccess)
    {
    unsigned long plane_mask = 1;
    int center;
    center = (int)(.5 * ( av_field_height - height ) );
    y_pos += center;

    // Copy to drawing area and to pixmap for refreshing
    XCopyPlane( XtDisplay(annot), bitmap, XtWindow( annot ), gc, 0, 0,
                width, height, x_pos, y_pos, plane_mask );
    XCopyPlane( XtDisplay(annot), bitmap, av_pixmap, gc, 0, 0,
                width, height, x_pos, y_pos, plane_mask );

    XFreePixmap(XtDisplay(annot), bitmap);
    } /* load bitmap */
}



/*-------------------------------------------------------------------------
  |     METHOD: GetAnnotSize
  |
  |    PURPOSE: Get the pixel width and height of the annotview object.
  -------------------------------------------------------------------------*/
void
AnnotView::GetAnnotSize( Dimension& w, Dimension& h )
{
  h = av_annot_height;
  w = av_annot_width;
}

/*-------------------------------------------------------------------------
  |     METHOD: SetAnnotSize
  |
  |    PURPOSE: Set height and width of AnnotView object.
  -------------------------------------------------------------------------*/
void
AnnotView::SetAnnotSize( Dimension w, Dimension h )
{
  av_annot_height = h;
  av_annot_width = w;
}

/*-------------------------------------------------------------------------
```

```
|     METHOD: drawing_areaCB
|
|    PURPOSE: Callback routine for AnnotView's input and expose callbacks.
|             Determine which it is by testing the cbs->reason field.
 ------------------------------------------------------------------------*/
void
AnnotView::drawing_areaCB( Widget widget, XtPointer av, XtPointer cbs_vptr )
{
  AnnotView                      *AV = (AnnotView *)av;
  XmDrawingAreaCallbackStruct    *cbs = (XmDrawingAreaCallbackStruct *)cbs_vptr;

  static Position        x, y, x_pos, y_pos;
  XEvent                 *event = cbs->event;
  Display                *dpy = event->xany.display;
  Dimension              w,h;
  int                    line_num;
  Pixmap                 bitmap;
  unsigned int           width, height;
  unsigned int           bitmap_error;

  if( av_current_file==NULL ) return;

  AV->GetAnnotSize( w, h );

  if (cbs->reason == XmCR_INPUT)
    {
    // Activated by AnnotView drawing_area input event.
    // Button Down events draw annotation
    if (event->xany.type == ButtonPress)
      {
      // Save locator position
      x = event->xbutton.x;
      y = event->xbutton.y;
      line_num = (int)(y / AV->av_field_height);
      if (line_num > (AV->av_num_annots-1) ) return;

      // request annotation change from server
      ostrstream command;
      command << "STOP AT " << line_num+AV->av_top_file_line_num
         << " IN " << AV->av_current_file->getName() << ends;

      MSGsenda( App::getMsgHandle(), "CAVE COMMAND %S", command.str() );
      delete command.str();
      }

    // Button Up events are ignored
    else if (event->xany.type == ButtonRelease)
      {
      }
    } /* if (cbs->reason == XmCR_INPUT) */

  // Handle expose events by recopying from pixmap
  if (cbs->reason == XmCR_EXPOSE )
    {
    XtVaGetValues( AV->annot,
                XmNwidth, &w,
                XmNheight, &h,
                NULL );

    XCopyArea( XtDisplay( AV->annot ), AV->av_pixmap, XtWindow( AV->annot ),
               AV->gc, 0, 0, w, h, 0, 0);
    }
  }
/*------------------------------------------------------------------------
      METHOD: addAnnotation

     PURPOSE: Calls drawAnnot method to add annotation.
 ------------------------------------------------------------------------*/
void
AnnotView::addAnnotation( int annot_type, const char *file,
                          const char *func, int line_number )
{
```

```
  switch( annot_type )
    {
    case( AV_STOP ):
    case( AV_BOMB ):
       SFile *sf=SFile::lookUpFile( file, (so_line_callback_t)NULL );

       if( av_current_file==sf )
          drawAnnot( annot_type, TRUE, line_number );
       else
       {
          int mask = av_current_file->getAnnotations( line_number );
          mask |= annot_type;
          sf->setAnnotations( line_number, mask  );
       }

    break;

    default:
       cout << "AnnotView::addAnnotation()...unknown annotation type" << endl;
    }
}

/*-------------------------------------------------------------------------
 |     METHOD: removeAnnotation
 |
 |    PURPOSE: Calls drawAnnot to remove annotation.
 -------------------------------------------------------------------------*/
void
AnnotView::removeAnnotation( int annot_type, const char *file,
                             const char *func, int line_number )
{
   cout << "AnnotView::removeAnnotation called...";
   cout << ((annot_type==AV_STOP) ? "AV_STOP" : "AV_BOMB") << endl;


  switch( annot_type )
    {
    case( AV_STOP ):
    case( AV_BOMB ):
       SFile *sf=SFile::lookUpFile( file, (so_line_callback_t)NULL );

       if( av_current_file==sf )
          drawAnnot( annot_type, FALSE, line_number );
       else
       {
          int mask = av_current_file->getAnnotations( line_number );
          mask &= ~annot_type;
          sf->setAnnotations( line_number, mask  );
       }

    break;

    default:
       cout << "AnnotView::addAnnotation()...unknown annotation type" << endl;
    }
}

/*-------------------------------------------------------------------------
 |     METHOD: FillAnnotWithAnnots
 |
 |    PURPOSE: Callback routine for AnnotView's input and expose callbacks.
 |             Determine which it is by testing the cbs->reason field.
 -------------------------------------------------------------------------*/
void
AnnotView::FillAnnotWithAnnots()
{
  int                   i;
  char                  *stop = "/u/khm/project/src/bitmaps/stop";
  char                  *focus = "/u/khm/project/src/bitmaps/focus";
  int                   mask;
  int                   y_pos;
  unsigned int          width, height;
```

```
  Dimension             w,h;

  if (av_current_file==NULL) return;

   or (i=0; i<av_num_annots; i++)
    {
    // get mask
    mask = av_current_file->getAnnotations( i + av_top_file_line_num );
    y_pos = i*av_field_height;


    // BOMB ANNOT
    // check annotation - if on toggle off; if off toggle on
    if( mask&AV_BOMB )
      {
      drawAnnot( AV_BOMB, TRUE, i + av_top_file_line_num);
      } /* if bomb */

    // STOP ANNOT
    // check annotation - if on toggle off; if off toggle on
    if( mask&AV_STOP )
      {
      drawAnnot( AV_STOP, TRUE, i + av_top_file_line_num);
      } /* if stop */

    // FOCUS ANNOT
    if( av_window_focus == i + av_top_file_line_num )
      {
      drawAnnot( AV_FOCUS, TRUE, i + av_top_file_line_num);
      } /* if focus */
    } /* for */
  SetAnnotSize( w, h );
}

  ------------------------------------------------------------------------
      METHOD: SetFocus
  |
  |    PURPOSE: Sets current filename to be viewed in AnnotView and focus.
  |             Focus is the line in the file that is within viewing area
  |             of AnnotView.
 ------------------------------------------------------------------------*/
void
AnnotView::SetFocus( int focus, const char* name )
{
  // check to see if this is the current file
  // if it's not, then need to get the new one

  SFile *newsf = SFile::lookUpFile( name, (so_line_callback_t)NULL );

  if( av_current_file!=NULL && av_current_file==newsf )
     {
     // they're the same
     // Is the line already visible in AnnotView?

     if( (av_top_file_line_num > focus) || ( av_top_file_line_num+av_num_annots <= focus) )
       {
       // set flag telling me I need to scroll the line into view in update
       av_top_file_line_num = max(1, focus - ((int)(av_num_annots/2)));
       av_left_file_col_num = 1;
       av_state |= AV_REFRESH;
       }
     }
  else
    {
    // Adds filename to cache if it's not already there
    av_current_file = newsf;
    // SET FLAG TO SIGNAL CHANGED FILE  (refresh flag)
    av_state |= AV_CHANGED_FILE;
    av_top_file_line_num = max(1, focus - ((int)(av_num_annots/2)));

    // for now
    av_left_file_col_num = 1;
```

```
    }
  av_window_focus = focus;

}

/*-----------------------------------------------------------------------
 |     METHOD: Update
 |
 |    PURPOSE: Update "image" in AnnotView window
 |      NOTES: Resizes AnnotView pixmap and redraws annotations
 -----------------------------------------------------------------------*/
void
AnnotView::Update()
{
  Pixmap                pixmap_temp;
  Dimension             w,h;
  XEvent                event;          // Event structure


  // Get new width and height
  XtVaGetValues( annot,
                 XmNwidth, &w,
                 XmNheight, &h,
                 NULL );

  SetAnnotSize( w, h );

  // Create and initialize pixmap
  // create a pixmap the same size as annotation area
  // Delete old and recreate
  XFreePixmap(XtDisplay(annot), av_pixmap);
  av_pixmap = XCreatePixmap(XtDisplay(annot),
                        RootWindowOfScreen(XtScreen(annot)), w, h,
                        DefaultDepthOfScreen(XtScreen(annot)));
  setColor( annot, "Black" );
  XFillRectangle(XtDisplay(annot), av_pixmap, gc, 0, 0, w, h);

  FillAnnotWithAnnots();

  // Send an expose event to cause AnnotView to redraw itself and flush out
  // the buffer
  XExposeEvent exp_event;               // Event structure
  // Initialize event structure
  exp_event.type = Expose;
  exp_event.serial = 0;
  exp_event.send_event = TRUE;
  exp_event.display = XtDisplay( annot );
  exp_event.window = XtWindow( annot );
  exp_event.x = 0;
  exp_event.y = 0;
  exp_event.width = 0;
  exp_event.height = 0;
  exp_event.count = 0;
  if (!XSendEvent(XtDisplay(annot),XtWindow(annot),FALSE,ExposureMask,(XEvent *) &exp_event))
    cout << "AnnotView::Update -- XSendEvent Failed"<<endl;
}
/*-----------------------------------------------------------------------
 |     METHOD: setColor
 |
 |    PURPOSE: Method used to set the global gc's color directly.  Just
 |             provide a widget and a color name.
 -----------------------------------------------------------------------*/
void
AnnotView::setColor(Widget widget, String color)
{
  Display       *dpy = XtDisplay(widget);
  Colormap      cmap = DefaultColormapOfScreen(XtScreen(widget));
  XColor        col, unused;

  if (!XAllocNamedColor(dpy, cmap, color, &col, &unused))
    {
    XtWarning("Can't alloc xxx color");
```

```
/*-----------------------------------------------------------------------
 |       FILE: App.C
 |    PURPOSE: Application object for Cave.
  -----------------------------------------------------------------------*/

#include "gui.h"

/*-----------------------------------------------------------------------
 |     METHOD: CAVEAppMainLoop
 |
 |    PURPOSE: Replacement for XtAppMainLoop() that not only processes
 |             X events, but also processes message events.
 |      NOTES:
 -----------------------------------------------------------------------*/

static Boolean doMsg( XtPointer user_data )
{
   CMPXselect(-2);
   return( FALSE );
}

void CAVEAppMainLoop( XtAppContext app )
{

   MSGsenda( App::getMsgHandle(), "BAT INITIALIZE" );

   XtAppAddWorkProc( app, doMsg, NULL );
   XtAppMainLoop( app );
}



/*-----------------------------------------------------------------------
     METHOD: App

    PURPOSE: Constructor
      USAGE: App      the_app( n, argv );
 PARAMETERS: Cardinal         n;
             char             **argv;
      NOTES:
 -----------------------------------------------------------------------*/
App::App( Cardinal argc, char **argv )
{
  Dimension     width, height;
  Model         mod;

  // set the current app pointer
  ap_current_app = this;

  // connect with the message server
  ap_msg_handle = MSGconnect( NULL, NULL );
  if( !ap_msg_handle )
  {
     cout << "CAVE: Couldn't connect to message server." << endl;
     exit(1);
  }

  // Let the server know that Cave is up and running
  if( !MSGservice_register( ap_msg_handle, "CAVE" ) )
  {
     cout << "CAVE server is already up!  Exiting..." << endl;
     exit(1);
  }


   / add path of executable to path list for searching for files
  int len = strlen( argv[1] );
  char *fname = new char[len+1];
  strcpy( fname, argv[1] );
  for( char *p=fname+len; *p!='/' && p!=fname; p-- )
     ;
  if( *p=='/' )
```

```
{
   p[0] = '\0';
   SFile::addToSourcePath( fname );
}
delete fname;



// initialize toolkit and create ap_toplevel shell
ap_toplevel = XtVaAppInitialize(&ap_app_context, "bat", NULL, 0, &argc,
                                argv, NULL,
                                XmNallowShellResize, TRUE ,
                                NULL );

// create paned manager object
ap_app_mgr = XtVaCreateWidget( "ap_app_mgr",
                                xmPanedWindowWidgetClass, ap_toplevel,
                                XmNallowResize, TRUE ,
                                NULL );

// this works for resources
XrmDatabase cur, ndb;
cur = XtDatabase( XtDisplay( ap_toplevel ) );
ndb = XrmGetFileDatabase("bat.xrdb");
XrmMergeDatabases(ndb,&cur);

// get width/height of main window
XtVaGetValues( ap_app_mgr,
               XmNwidth,  &width,
               XmNheight, &height,
               NULL );

// create message form manager object
ap_msg_source_mgr = XtVaCreateWidget( "ap_msg_source_mgr",
                                      xmFormWidgetClass, ap_app_mgr,
                                      NULL );

// Create file message area
ap_ptrFileMsgView = new MsgView( ap_msg_source_mgr );

// create source (code&annot) form manager object
ap_source_mgr = XtVaCreateWidget( "ap_source_mgr",
                                  xmFormWidgetClass,    ap_msg_source_mgr,
                                  XmNtopAttachment,     XmATTACH_WIDGET,
                                  XmNtopWidget,         ap_ptrFileMsgView->getWidget(),
                                  XmNbottomAttachment, XmATTACH_FORM,
                                  XmNleftAttachment,    XmATTACH_FORM,
                                  XmNrightAttachment,   XmATTACH_FORM,
                                  NULL );

// Create annotation manager which will create
// codeview and annotview
ap_ptrAnnotMgr = new AnnotMgr( ap_source_mgr );

// create message form manager object
ap_command_mgr = XtVaCreateWidget( "ap_command_mgr",
                                   xmFormWidgetClass,          ap_app_mgr,
                                   XmNallowResize,             TRUE,
                                   NULL );

//Create
ap_button_mgr = XtVaCreateWidget("ap_button_mgr",
                                  xmFormWidgetClass,          ap_command_mgr,
                                  XmNallowResize,             TRUE,
                                  NULL );

ap_ptrButtonView = new ButtonView( ap_button_mgr );
ap_ptrCommandView = new CommandView( ap_command_mgr, ap_button_mgr );
ap_ptrDisplayView = new DisplayView( ap_app_mgr );

XtManageChild( ap_command_mgr );
```

```
XtManageChild( ap_msg_source_mgr );
XtManageChild( ap_button_mgr );
XtManageChild( ap_source_mgr );
XtManageChild( ap_app_mgr );

// Recursively traverse widget hierarchy creating windows for
// each one
XtRealizeWidget(ap_topleve1);

typedef char *cptr;
cptr *ap_argv = new cptr[argc+1];
for( int j=1; j<argc; j++ )
   ap_argv[j] = argv[j];

ap_argv[0] = "bat";
ap_argv[argc] = NULL;


// make sure the Bat debug server is up and running
char execpath[50];
getcwd( execpath, 50 );
strcat( execpath, "/bat" );

MSGservice_start( ap_msg_handle, "BAT", NULL, execpath, ap_argv );


// Turn control over to our event processing routine
// which processes both X and mesg events.
CAVEAppMainLoop(ap_app_context);
}
```

```
/*-------------------------------------------------------------------
 |       FILE: ButtonView.C
 |    PURPOSE: Object class handling for command area (buttons and
 |             command input line) of gui.
  -------------------------------------------------------------------*/

#include "gui.h"

//Initialize button_names
static char *button_names[] = { "RUN", "CONT", "STEP", "NEXT",
                                "STATUS","WHERE","UP","DOWN",
                                    "PRINT", "DISPLAY", "QUIT"
                                  };

/*-------------------------------------------------------------------
 |     METHOD: ButtonView
 |
 |    PURPOSE: Constructor for Button area of gui.  Creates buttons and
 |             command input area.
 -------------------------------------------------------------------*/
ButtonView::ButtonView( Widget& ancestor ) : View( ancestor )
{
  int           i;


  // Set values of button form manager
  XtVaSetValues( ancestor,
                 XmNrightAttachment,    XmATTACH_FORM,
                 XmNleftAttachment,     XmATTACH_FORM,
                 XmNallowResize,        TRUE,
                 XmNfractionBase,       NBUTTONS,
                 NULL );

  cd_buttons = new Widget[ NBUTTONS ];

  // Create buttons
  for( i = 0; i < NBUTTONS; i++ )
    {
    // create unmanaged and manage at the end if more parameters are to be
    // set - this makes it faster
    cd_buttons[i] =
      XtVaCreateManagedWidget( button_names[i],
                               xmPushButtonWidgetClass,  ancestor,
                               XmNtopAttachment,         XmATTACH_FORM,
                               XmNbottomAttachment,      XmATTACH_FORM,
                               XmNrightAttachment,       XmATTACH_POSITION,
                               XmNrightPosition,         (i + 1),
                               // first one should be attached for form
                               XmNleftAttachment, i? XmATTACH_WIDGET : XmATTACH_FORM,
                               // others are attached to the previous button
                               XmNleftWidget, i? cd_buttons[i-1] : (Widget)NULL,
                               NULL );

  XtAddCallback( cd_buttons[i], XmNactivateCallback,
                 (XtCallbackProc)ButtonView::buttonCB, (XtPointer)this );
  } /* for */

}

/*-------------------------------------------------------------------
 |     METHOD: ~ButtonView
 |
 |    PURPOSE: Destructor...currently does nothing.
 -------------------------------------------------------------------*/
  tonView::~ButtonView( )

}

/*-------------------------------------------------------------------
 |     METHOD: buttonCB
 |
 |    PURPOSE: Callback routine for ButtonView's input and expose callbacks.
```

```
  |             Determine which it is by testing the cbs->reason field.
  |
  ------------------------------------------------------------------------*/
void
ButtonView::buttonCB( Widget widget, XtPointer bv, XtPointer cbs_vptr )
{
  ButtonView                    *BV = (ButtonView *)bv;
  App                           *AP = App::getCurrentApp();
  CodeView                      *CV = AP->getAnnotMgr()->getCodeView();
  XmPushButtonCallbackStruct    *cbs = (XmPushButtonCallbackStruct *)cbs_vptr;
  static Position               x, y, y_pos;
  XEvent                        *event = cbs->event;
  char                          *name;


  name = XtName( widget );
  ostrstream command;

  if (cbs->reason == XmCR_ACTIVATE)
  {
     command << name << " ";

     // if something is selected, then append it to the command
     char *text;
     const char *fname;
     int f_lnum;
     CV->getSelected( text, fname, f_lnum );
     command << text;

     // close the string and send it

     command << ends;

     MSGsenda( AP->getMsgHandle(), "CAVE COMMAND %S", command.str() );

     delete command.str();

     if(!strcmp(name, "QUIT"))
        exit(0);
  }
}
```

```
/*-------------------------------------------------------------------
 |       FILE: CodeView.C
 |    PURPOSE: Handles source code view and user interaction with view
  `------------------------------------------------------------------*/

#include "gui.h"
#define FONT_DIR_NAME   "/usr/lib/X11/fonts/misc"

// initialize static variables
int     CodeView::cv_top_file_line_num = 1;
int     CodeView::cv_left_file_col_num = 1;


/* Globals: the ap_toplevel window/widget and the label for the bitmap.
 * "colors" defines the colors we use, "cur_color" is the current
 * color being used, and "cur_bitmap" references the current bitmap file.
 */
String          colors[] = { "Black", "Red", "Green", "Blue" };
Pixel           fg, fg_edit;


/*-------------------------------------------------------------------
 |     METHOD: CodeView
 |
 |    PURPOSE: Codeview Object constructor.
 -------------------------------------------------------------------*/
CodeView::CodeView( Widget& ancestor ):
                    ScrolledView( ancestor, "CodeView" )
{
  Dimension             h,w;
  int                   i;
  Dimension             view_width, view_height;


  / Initialize
  cv_num_fields = 0;
  cv_state      = 0;

  XtVaSetValues( scrolled_w,
                 XmNleftAttachment, XmATTACH_WIDGET,
                 XmNleftWidget, *annot_scrolled ,
                 XmNtopAttachment, XmATTACH_FORM ,
                 XmNbottomAttachment,XmATTACH_FORM ,
                 XmNrightAttachment,XmATTACH_FORM ,
                 NULL );

  // query ancestor (ap_source_mgr) width/height
  XtVaGetValues( ancestor,
                XmNwidth,  &view_width,
                XmNheight, &view_height, NULL );

  mainform = XtVaCreateWidget( "mainform",
                   xmFormWidgetClass, scrolled_w,
                   XmNwidth, view_width,
                   XmNheight, view_height,
                   XmNallowResize, True,
                   NULL );

  // Scarf away width and height for resize callback
  XtVaGetValues( mainform,
                 XmNheight, &cv_mainform_height,
                 XmNwidth, &cv_mainform_width,
                 NULL );

   / query colormap and foreground color
  tVaGetValues(mainform, XmNcolormap, &cmap,
                XmNforeground, &fg,
                NULL);

  /* convert "red" to a pixel value from given colormap */
  XAllocNamedColor(XtDisplay(mainform), cmap, "white", &color, &unused);
  fg_edit = color.pixel;
```

```
  cv_current_file = NULL;

  // Determine field height by creating temporary text widget
  Widget temp_field_w;
  temp_field_w = XtVaCreateManagedWidget("temp_field_w",
                                  xmTextWidgetClass,mainform,
                                  XmNeditMode, XmSINGLE_LINE_EDIT,
                                  XmNshadowThickness,0,
                                  XmNhighlightThickness,0,
                                  XmNborderWidth,0,
                                  XmNmarginHeight,0,
                                  XmNmarginWidth,3,
                                  NULL);

  XtVaGetValues(temp_field_w,
                XmNwidth, &cv_field_width,
                XmNheight, &cv_field_height,
                NULL );
  XtDestroyWidget( temp_field_w );

  // Application-defined ScrolledWindows won't create their own
  // Scrollbars.  So, we create them ourselves as children of the
  // ScrolledWindow widget.  The vertical Scrollbar's maximum size is
  // the number of rows that exist (in unit values).  The horizontal
  // Scrollbar's maximum width is represented by the number of columns.
  // Create scrollbar widgets -- Must specify orientation at
  // creation time!!
  cv_vsb = XtVaCreateManagedWidget( "cv_vsb",
                                  xmScrollBarWidgetClass, scrolled_w,
                                  XmNorientation, XmVERTICAL,
                                  NULL );

  // set flags
  cv_state |= CV_RESIZED;

  // Allow the ScrolledWindow to initialize itself accordingly
  XmScrolledWindowSetAreas(scrolled_w, (Widget)NULL, cv_vsb, mainform);

  XtManageChild(mainform);
  XtManageChild(scrolled_w);
  }

/*-------------------------------------------------------------------------
  |     METHOD: ~CodeView
  |
  |    PURPOSE: Destructor...currently does nothing.
  -------------------------------------------------------------------------*/
CodeView::~CodeView( )
{
}

/*-------------------------------------------------------------------------
  |     METHOD: SetFocus
  |
  |    PURPOSE: Sets current filename to be viewed in CodeView and focus.
  |             Focus is the line in the file that is within viewing area
  |             of CodeView.
  -------------------------------------------------------------------------*/
void CodeView::SetFocus( int focus, const char* name )
{
  // check to see if this is the current file
  // if it's not, then need to get the new one

  SFile *newsf = SFile::lookUpFile( name, (so_line_callback_t)NULL );


  if( cv_current_file!=NULL && cv_current_file==newsf )
    // they're the same
    {
    // Is the line already visible in CodeView?
    // If not,  then set flag telling me I need to scroll the line into view
```

```
    // in update  (set refresh flag)
    if( (cv_top_file_line_num > focus) || ( cv_top_file_line_num+cv_num_fields <= focus) )
      {
      cv_top_file_line_num = max(1, focus - ((int)(cv_num_fields/2)));
      cv_left_file_col_num = 1;
      cv_state |= CV_REFRESH;
      }
    }
  else
    {
    // Adds filename to cache if it's not already there
    cv_current_file = newsf;
    // SET FLAG TO SIGNAL CHANGED FILE  (refresh flag)
    cv_state |= CV_CHANGED_FILE;
    cv_top_file_line_num = max(1,focus - ((int)(cv_num_fields/2)));
    // for now
    cv_left_file_col_num = 1;
    }

  cv_window_focus = focus;
}



/*-------------------------------------------------------------------------
  |     METHOD: motionVerifyCB
  |
  |    PURPOSE: Callback for field text widgets.  It is called when the
  |             user uses the mouse or arrow keys to change the cursor
  |             location or the user drags the mouse or multi-clicks.
  -------------------------------------------------------------------------*/
void CodeView::motionVerifyCB( Widget text_w, XtPointer cv, XtPointer cbs_vptr )
{
  CodeView                        *CV = (CodeView *)cv;
   mTextVerifyCallbackStruct&   cbs = *((XmTextVerifyCallbackStruct *)cbs_vptr);

  char *temp;
  int  len;

  temp = XmTextGetSelection( text_w  );


  if( temp!=NULL )
    {
    len = strlen( temp );

    if (len < 100)
      strcpy(CV->cv_selected, temp );

    XtFree( temp );

    int val;
    istrstream( XtName(text_w), sizeof( XtName(text_w)) ) >> val;
    CV->cv_selected_file_lnum = CV->cv_top_file_line_num + val;
    }
  else
  {
     CV->cv_selected[0] = '\0';
     CV->cv_selected_file_lnum = 0;
  }


}


   -------------------------------------------------------------------------
  |     METHOD: FillFieldsWithText
  |
  |    PURPOSE: Fill the codeview with text from the appropriate lines
  |             in the file.
  -------------------------------------------------------------------------*/
void CodeView::FillFieldsWithText()
```

```
{
  int i;

  if( cv_current_file==NULL )
     return;

  // turn on scrolled flag
  cv_state |= CV_SCROLL_CB;
  cv_state |= CV_RESIZED;

  for (i=0; i<cv_num_fields; i++)
    {
    XtVaSetValues( fields[i],
                   // SFile is 1-based
                   XmNvalue, (*cv_current_file)[cv_top_file_line_num+i],
                   NULL );
    }

  // turn off scrolled flag
  cv_state &= ~CV_SCROLL_CB;
  cv_state &= ~CV_RESIZED;
  }


/*-------------------------------------------------------------------------
  |     METHOD: GetMainformSize
  |
  -------------------------------------------------------------------------*/
void
CodeView::GetMainformSize( Dimension& w, Dimension& h)
{
  h = cv_mainform_height;
  w = cv_mainform_width;
}

/*-------------------------------------------------------------------------
 |     METHOD: SetMainformSize
 |
 |    PURPOSE: Set height and width of mainform
 -------------------------------------------------------------------------*/
void CodeView::SetMainformSize( Dimension w, Dimension h )
{
  cv_mainform_height = h;
  cv_mainform_width = w;
}
/*-------------------------------------------------------------------------
 |     METHOD: CalcNumFields
 |
 |    PURPOSE: Calculate number of fields necessary to fill source window
 -------------------------------------------------------------------------*/
int CodeView::CalcNumFields( int view_height, int char_height )
{
  return( cv_num_fields =
            (int)floor(( (double)view_height/(double)(char_height) )) );
}

/*-------------------------------------------------------------------------
 |     METHOD: Update
 |
 |    PURPOSE: Update "image" in CodeView window
 -------------------------------------------------------------------------*/
void CodeView::Update()
{
  int           i;
  Dimension     h,w;

  if (cv_state&CV_RESIZED)
    {
    // Delete fields
    for (i  = 0; i < GetNumFields(); i++)
      {
      // Destroy widgets
```

```
    XtDestroyWidget( fields[i] );
    }
  // Deallocate Baum memory
  if( GetNumFields() )
    delete [] fields;

  // Calculate number of fields necessary to fill window
  XtVaGetValues( mainform,
                 XmNheight, &h,
                 NULL );

  cv_num_fields = CalcNumFields( h, cv_field_height );

  // Allocate new fields
  fields = new Widget[ cv_num_fields ];

  for (i = 0; i < cv_num_fields; i++)
    {
    ostrstream        fieldn;
    fieldn << i << ends;
    fields[i] =
      XtVaCreateManagedWidget( fieldn.str(),
                               xmTextWidgetClass, mainform,
                               XmNeditMode, XmSINGLE_LINE_EDIT ,
                               XmNeditable, False,
                               // XmNforeground, fg ,
                               XmNshadowThickness,0,
                               XmNhighlightThickness,0,
                               XmNborderWidth,0,  // ADD THIS IN WHEN READY!
                               XmNmarginHeight,0,
                               XmNmarginWidth,3,
                               /* first one should be attached for form */
                               XmNtopAttachment, i? XmATTACH_WIDGET : XmATTACH_FORM ,
                               /* others are attached to the previous field */
                               XmNtopWidget,        i? fields[i-1] : (Widget)NULL,
                               XmNleftAttachment, XmATTACH_FORM ,
                               XmNrightAttachment, XmATTACH_FORM ,
                               NULL );


    XtAddCallback( fields[i], XmNmotionVerifyCallback,
                   (XtCallbackProc)CodeView::motionVerifyCB, (XtPointer)this);

    // Also advance focus to next Text widget, which is in the
    // next Tab Group because each Text widget is in a Form by
    // itself.  If they were all in the same manager, we'd just
    // use XmTRAVERSE_NEXT instead.
    //    fields[i].addCallback("activateCallback",(BaumCallbackProc)XmProcessTraversal, this)
    // XtAddCallback(fields[i], XmNactivateCallback, XmProcessTraversal, XmTRAVERSE_NEXT_TAB_G

    XtManageChild(fields[i]);
    delete fieldn.str();
    } /* for */

  } /* resized */

if(cv_current_file!=NULL && cv_state&CV_RESIZED || cv_state&CV_REFRESH || cv_state&CV_CHANGED_
  {
  FillFieldsWithText();

  XtVaSetValues( cv_vsb,
                 XmNvalue,             cv_top_file_line_num,
                 XmNminimum,           1,
                 XmNmaximum,           cv_current_file->getNLines()+1,
                 XmNsliderSize,        cv_num_fields,
                 XmNpageIncrement,     cv_num_fields-1,
                 NULL );
  // unset flags
  cv_state &=~CV_REFRESH;
  cv_state &=~CV_CHANGED_FILE;
  cv_state &=~CV_RESIZED;
  }
```

```
}

/*-------------------------------------------------------------------------
 |     METHOD: SetScrollbarCBfunc
 |
 |    PURPOSE: Set scrollbar callback.
 |      NOTES: Adds vertical scrollbar callbacks to callback list of
 |             vertical scrollbar widget.
 -------------------------------------------------------------------------*/
void CodeView::SetScrollbarCBfunc( XtCallbackProc cb, XtPointer user_data )
{
  XtAddCallback(cv_vsb, "valueChangedCallback", (XtCallbackProc)cb,
                    (XtPointer)user_data);
  XtAddCallback(cv_vsb, "dragCallback", (XtCallbackProc)cb,
                    (XtPointer)user_data);

}


/*-------------------------------------------------------------------------
 |     METHOD: SetResizeCBfunc
 -------------------------------------------------------------------------*/
void CodeView::SetResizeCBfunc( XtEventHandler cb, XtPointer user_data )
{
  XtAddEventHandler( mainform, StructureNotifyMask, False,
                           ( XtEventHandler)cb, (XtPointer)user_data );
}


/*-------------------------------------------------------------------------
 |     METHOD: cv_scrolledHCB
 |
 |    PURPOSE: Currently, this does nothing.  Horz scrollbars aren't wired.
 -------------------------------------------------------------------------*/
void CodeView::scrolledHCB(Widget scrollbar, XtPointer cv, XtPointer cbs_vptr )
{
}
```

```
/*-------------------------------------------------------------------
 |       FILE: CommandView.C
 |    PURPOSE: Object class handling command area (buttons and command
 |             input line) of gui.
  -------------------------------------------------------------------*/

#include "gui.h"

typedef void (*OUTPUTfptr)(...);
static void OutputCB( char *text, CommandView *CV );

/*-------------------------------------------------------------------
 |     METHOD: CommandView
 |
 |    PURPOSE: Constructor for Command area of gui.  Creates buttons and
 |             command input area.
 -------------------------------------------------------------------*/
CommandView::CommandView( Widget& ancestor, Widget& top_widget ) : View( ancestor )
{
  // Create command line
  XmString      prompt;
  prompt = XmStringCreateSimple("Command:");

  cd_command_w = XtVaCreateWidget( "cd_command_w",
                                   xmCommandWidgetClass,        ancestor,
                                   XmNtopAttachment,            XmATTACH_WIDGET,
                                   XmNtopWidget,                top_widget,
                                   XmNbottomAttachment,         XmATTACH_FORM,
                                   XmNrightAttachment,          XmATTACH_FORM,
                                   XmNleftAttachment,           XmATTACH_FORM,
                                   XmNpromptString,             prompt,
                                   XmNallowResize,              TRUE,
                                   NULL );


  XmStringFree(prompt);
  // Get handle to history list area
  cd_output_w = XmCommandGetChild( cd_command_w, XmDIALOG_HISTORY_LIST );

  // Get handle to text area
  cd_text_w = XmCommandGetChild( cd_command_w, XmDIALOG_COMMAND_TEXT );
  XtAddCallback(cd_command_w, XmNcommandEnteredCallback,
                (XtCallbackProc)CommandView::ExecCmdCB, (XtPointer)this);



  XtManageChild( cd_command_w );

  // register with the message server
  int def_args[2];
  def_args[0] = NULL;
  def_args[1] = (int)this;

  MSGregister( App::getMsgHandle(), "BAT OUTPUT %1s", (OUTPUTfptr)OutputCB, 2, (void **)def_args
}


static void
OutputCB( char *text, CommandView *CV )
{
   CV->AddTextToOutput( text );
}

/*-------------------------------------------------------------------
       METHOD: ~CommandView

 |    PURPOSE: Destructor...currently does nothing.
 -------------------------------------------------------------------*/
CommandView::~CommandView( )
{
}
```

```
/*-----------------------------------------------------------------------
 |     METHOD: ExecCmdCB
 |
 |    PURPOSE: Execute the command and redirect output to the ScrolledText
 |             window
 |      USAGE:
 | PARAMETERS: Widget           cmd_widget;     Command widget
 |             XtPointer        cd;             CommandView object passed
 |                                              as client data
 |             XtPointer        cbs_vptr;       Pointer to callback struct
 |      NOTES:
 -----------------------------------------------------------------------*/
void
CommandView::ExecCmdCB( Widget cmd_widget, XtPointer cd, XtPointer cbs_vptr )
{
  App                           *AP = App::getCurrentApp();
  CommandView                   *CD = (CommandView *)cd;
  XmCommandCallbackStruct       *cbs = ((XmCommandCallbackStruct *)cbs_vptr);
  char                          *cmd;

  XmStringGetLtoR(cbs->value, XmSTRING_DEFAULT_CHARSET, &cmd);

  if (!cmd || !*cmd)
    { /* nothing typed? */
    if (cmd)
      XtFree(cmd);
    return;
    }
  else
    {
    MSGsenda( AP->getMsgHandle(), "CAVE COMMAND %S", cmd );
    XtFree( cmd );
    }
}


/*-----------------------------------------------------------------------
 |     METHOD: MakeItemVisible
 |
 |    PURPOSE: Scrolls list item into the viewport
 -----------------------------------------------------------------------*/
void
CommandView::MakeItemVisible( Widget list_w, XmString item )
{
  int   *pos_list;
  int   pos_count;

  // Find all occurrences of item in list
  Boolean found = XmListGetMatchPos( list_w, item, &pos_list, &pos_count );

  // Scroll last item into view
  if (found)
    XmListSetBottomPos( list_w, pos_list[pos_count-1] );
}

/*-----------------------------------------------------------------------
 |     METHOD: AddTextToOutput
 |
 |    PURPOSE: Appends text string to the output window history list.
 -----------------------------------------------------------------------*/
void
CommandView::AddTextToOutput( char *text )
{
  XmString              text_string;

  if( !text ) return;

  // Convert to Motif string - use this routine so that newlines
  // are handled properly
  text_string = XmStringCreateLtoR(text, XmSTRING_DEFAULT_CHARSET);

  // 0 means append to the end of command history list
```

```
  XmListAddItemUnselected( cd_output_w, text_string, 0 );

  MakeItemVisible( cd_output_w, text_string );

   mStringFree(text_string);
}

/*-----------------------------------------------------------------------
 |     METHOD: AddCommandToOutput
 |
 |    PURPOSE: Appends command string to command line and "executes" it
 |      NOTES: DELETE THIS METHOD BECAUSE IT'S NO DIFFERENT THAN AddTextToOutput
 |             This method is bogus and not used anymore!
 -----------------------------------------------------------------------*/
void
CommandView::AddCommandToOutput( char *text )
{
  XmString              text_string;
  char  *new_string = new char[strlen(text)+1];

  // Append carriage-return to text
  strcpy(new_string, text);
  strcat(new_string, "\n");

  // Convert to Motif string
  text_string = XmStringCreateLtoR(new_string, XmSTRING_DEFAULT_CHARSET);

  // 0 means append to the end of command history list
  XmCommandAppendValue( cd_command_w, text_string);

  XmStringFree(text_string);
}
```

```
/*-----------------------------------------------------------------------
 |       FILE: DisplayView.C
 |    PURPOSE: Object class for expression display area.
  -----------------------------------------------------------------------*/

#include "gui.h"

typedef void (*DISPLAYfptr)(...);
static void DisplayCB( char *text, DisplayView *DV );
static void ClearCB( DisplayView *DV );

/*-----------------------------------------------------------------------
 |     METHOD: DisplayView
 |
 |    PURPOSE: Constructor for Display area of gui.  Creates read-only
 |             scrolled window where values of variables will be
 |             continuously displayed.
 -----------------------------------------------------------------------*/
DisplayView::DisplayView( Widget& ancestor ) :
                          ScrolledView( ancestor, "DisplayView" )
{
  Arg   args[10];
  int   n;

  // must use varargs with XmCreateScrolledText
  // Create text_w as a ScrolledText window
  n = 0;
  XtSetArg(args[n], XmNrows,              6); n++;
  XtSetArg(args[n], XmNcolumns,           80); n++;
  XtSetArg(args[n], XmNeditable,          False); n++;
  XtSetArg(args[n], XmNeditMode,          XmMULTI_LINE_EDIT); n++;
  XtSetArg(args[n], XmNwordWrap,          True); n++;
  XtSetArg(args[n], XmNcursorPositionVisible, False); n++;

   ' create scroled text widget
  ork_w = XmCreateScrolledText( ancestor, "work_w", args, n );

  XtManageChild(work_w);

  // register with the message server
  int def_args[2];
  def_args[0] = NULL;
  def_args[1] = (int)this;

  MSGregister( App::getMsgHandle(), "BAT DISPLAY %1s", (DISPLAYfptr)DisplayCB, 2, (void **)def_a
  def_args[0] = (int)this;
  MSGregister( App::getMsgHandle(), "BAT CLEARDISPLAY", (DISPLAYfptr)ClearCB, 1, (void **)def_ar
}


static void
DisplayCB( char *text, DisplayView *DV )
{
   DV->AddTextToDisplay( text );
}
static void
ClearCB( DisplayView *DV )
{
   DV->Clear();
}

/*-----------------------------------------------------------------------
 |     METHOD: AddTextToDisplay
 |
 |    PURPOSE: Appends text string to the display window.
  -----------------------------------------------------------------------*/
void
DisplayView::AddTextToDisplay( char *text )
{
  XmString              text_string;

  if( !text ) return;
```

```
  XmTextInsert( work_w, ds_wpr_position, text );
  ds_wpr_position += strlen(text);
  XtVaSetValues( work_w, XmNcursorPosition, ds_wpr_position, NULL );
  XmTextInsert( work_w, ds_wpr_position, "\n" );
  ds_wpr_position += 1;
  XtVaSetValues( work_w, XmNcursorPosition, ds_wpr_position, NULL );
}

void
DisplayView::Clear()
{
   ds_wpr_position = 0;
   XtVaSetValues( work_w, XmNcursorPosition, ds_wpr_position, NULL );
   char *text="";
   XmTextSetString( work_w, text );
}


/*-----------------------------------------------------------------------
 |     METHOD: ~DisplayView
 |
 |    PURPOSE: Destructor...currently does nothing.
 -----------------------------------------------------------------------*/
DisplayView::~DisplayView( )
{
}
```

```
/*-----------------------------------------------------------------
 |       FILE: MsgView.C
 |    PURPOSE: Object class for the msg area.
  -----------------------------------------------------------------*/

#include "gui.h"

static void
setMessageHandler( char *file, char *function, int line_number );

typedef void (*FOCUSfptr)(...);

/*-----------------------------------------------------------------------
 |     METHOD: MsgView
 |
 |    PURPOSE: Constructor of msg area object.
 -----------------------------------------------------------------------*/
MsgView::MsgView( Widget& ancestor ) : View( ancestor )
{
  mv_text_w = XtVaCreateManagedWidget( "mv_text_w",
                                       xmTextWidgetClass,       ancestor,
                                       XmNeditMode,             XmSINGLE_LINE_EDIT,
                                       XmNeditable,             False,
                                       XmNshadowThickness,      0,
                                       XmNhighlightThickness,   0,
                                       XmNborderWidth,          0,
                                       XmNcursorPositionVisible,False,
                                       XmNmarginHeight,         0,
                                       XmNshadowThickness,      3,
                                       XmNmarginWidth,          3,
                                       XmNtopAttachment,        XmATTACH_FORM,
                                       XmNleftAttachment,       XmATTACH_FORM,
                                       XmNrightAttachment,      XmATTACH_FORM,
                                       NULL );

  // Register focus handler with the mesg server
  MSGregister( App::getMsgHandle(), "BAT FOCUS %1s %2s %3d", (FOCUSfptr)setMessageHandler, 3, NU
}


/*-----------------------------------------------------------------------
 |     METHOD: ~MsgView
 |
 |    PURPOSE: Destructor...does nothing.
 -----------------------------------------------------------------------*/
MsgView::~MsgView( )
{
}

/*-----------------------------------------------------------------------
 |     METHOD: setMessageHandler
 |
 |    PURPOSE: Send text to message area
 | PARAMETERS: int              line_num;       // line number of annotation
 |             const char*      funcn;          // function name
 |             const char*      src_filen;      // source file name
 -----------------------------------------------------------------------*/
static void
setMessageHandler( char *file, char *function, int line_number )
{
  MsgView *MV = App::getCurrentApp()->getFileMsgView();

  ostrstream    text_stream;

  text_stream << "Stopped in " << function << " at line " << line_number << " in file " << file

  XmTextSetString( MV->getWidget(), text_stream.str() );

  // My responsibility to delete this string
  delete text_stream.str();
```

```
/*-------------------------------------------------------------------
 |       FILE: ScrolledView.C
 |    PURPOSE: Abstract base class for scrolled views.
  -------------------------------------------------------------------*/

#include "gui.h"

/*-------------------------------------------------------------------
 |     METHOD: ScrolledView
 |
 |    PURPOSE: Constructor
 -------------------------------------------------------------------*/
ScrolledView::ScrolledView( Widget& ancestor, char *name):
                            View( ancestor )
{

  // Create unmanaged scrolled window
  scrolled_w = XtVaCreateWidget( name,
                                 xmScrolledWindowWidgetClass, *parent,
                                 XmNscrollingPolicy, XmAPPLICATION_DEFINED,
                                 XmNshadowThickness, 3,
                                 NULL );
}


/*-------------------------------------------------------------------
 |     METHOD: ~ScrolledView
 |
 |    PURPOSE: Destructor...Currently does nothing.
 -------------------------------------------------------------------*/
ScrolledView::~ScrolledView ()
{
}
```

```
/*-------------------------------------------------------------------------
 |       FILE: View.C
 |    PURPOSE: Abstract base class for views.
  -------------------------------------------------------------------------*/

#include "gui.h"
/*-------------------------------------------------------------------------
 |     METHOD: View
 |
 |    PURPOSE: Constructor
 -------------------------------------------------------------------------*/
View::View( Widget& ancestor )
{
  parent = &ancestor;
}
```

```
// as.h - header file for the assembler routines
// Based on code from gnu gdb.

   Type of a jump.
//
typedef enum jumptypeen
{
   JT_BRANCH,
   JT_CALL,
   JT_END
} jumptype_t;

//
//  An element in the list of jumps.
//
typedef struct jumpst
{
   jumptype_t ju_type;          // call or branch
   taddr_t ju_addr;             // instruction address - 0 for end of jump array
   taddr_t ju_dstaddr;          // target of jump - 0 if unknown
   bool ju_unconditional;       // is the jump unconditional?
} jump_t;

jump_t *get_jumps(taddr_t addr, const char *text, int len, int want_calls, int want_branches);
taddr_t get_next_pc(Process *proc, taddr_t pc);
```

```
/*-----------------------------------------------------------------
 |       FILE: as.C
 |    PURPOSE: Machine dependent (sparc) assembler routines, and code
 |             for finding jumps in sparc code.
 |      NOTES:   Based on code from gnu gdb.
  -----------------------------------------------------------------*/

#include "symtab.h"
#include "target.h"
#include "as.h"


//-------------------------------------------------------------------------
//
// Code for finding the jumps in SPARC machine code
//
//-------------------------------------------------------------------------

typedef unsigned long psr_t, fpsr_t;

//  A SPARC instruction.
//
typedef union sparc_instun
{
   unsigned        word;
   struct
   {
      unsigned        op:2;
      unsigned        disp30:30;
   } fmt1;
   struct
   {
      unsigned        op:2;
      unsigned        rd:5;
      unsigned        op2:3;
      unsigned        imm22:22;
   } fmt2_sethi;
   struct
   {
      unsigned        op:2;
      unsigned        a:1;
      unsigned        cond:4;
      unsigned        op2:3;
      unsigned        disp22:22;
   } fmt2;
   struct
   {
      unsigned        op:2;
      unsigned        rd:5;
      unsigned        op3:6;
      unsigned        rs1:5;
      unsigned        i:1;
      unsigned        asi:8;
      unsigned        rs2:5;
   } fmt3r;
   struct
   {
      unsigned        op:2;
      unsigned        rd:5;
      unsigned        op3:6;
      unsigned        rs1:5;
      unsigned        i:1;
      unsigned        simm13:13;
   } fmt3i;
   struct
   {
      unsigned        op:2;
      unsigned        rd:5;
      unsigned        op3:6;
      unsigned        rs1:5;
      unsigned        opf:9;
      unsigned        rs2:5;
```

```
   } fmt3f;
} sparc_inst_t;

typedef enum
{
UNIMP = 0, BICC = 2, SETHI = 4, FBFCC = 6, CBCCC = 7} btype_t;

//
//  Some opcodes that we special case.
//
#define OR      002
#define SUBCC   024
#define TICC    072
#define JMPL    070
#define RESTORE 075
#define FPOP1   064
#define FPOP2   065

typedef enum
{
   CC_N, CC_E, CC_LE, CC_L, CC_LEU, CC_LU, CC_NEG, CC_VS,
   CC_A, CC_NE, CC_G, CC_GE, CC_GU, CC_GEU, CC_POS, CC_VC
} cond_t;

typedef enum
{
   FC_N, FC_NE, FC_LG, FC_UL, FC_L, FC_UG, FC_G, FC_U,
   FC_A, FC_E, FC_UE, FC_GE, FC_UGE, FC_LE, FC_ULE, FC_O
} fp_cond_t;

static int sign_ext (unsigned u, int bit);
static int cond_holds (psr_t psr, cond_t cond);
static int fp_cond_holds (fpsr_t psr, fp_cond_t cond);



static int
sign_ext( unsigned u, int bit )
{
   if (u & (1 << (bit - 1)))
      u |= ~(~0 & ((1 << bit) - 1));
   return u;
}



static int
fp_cond_holds( psr_t psr, fp_cond_t fp_cond )
{
   int cc = (int)(psr >> 10) & 3;
   bool e = cc == 0;
   bool l = cc == 1;
   bool g = cc == 2;
   bool u = cc == 3;

   switch (fp_cond)
   {
   case FC_N:
      return 0;
   case FC_NE:
      return l | g | u;
   case FC_LG:
      return l | g;
   case FC_UL:
      return l | u;
   case FC_L:
      return l;
   case FC_UG:
      return g | u;
   case FC_G:
      return g;
   case FC_U:
```

```
      return u;
   case FC_A:
      return 1;
   case FC_E:
      return e;
   case FC_UE:
      return u | e;
   case FC_GE:
      return e | g;
   case FC_UGE:
      return e | g | u;
   case FC_LE:
      return e | l;
   case FC_ULE:
      return e | l | u;
   case FC_O:
      return e | l | g;
   }

   panic("cond botch in fp_cond_holds()");
   return 0;
}
```

```
static int
cond_holds( psr_t psr, cond_t cond )
{
   bool n = (psr & (1 << 23)) != 0;
   bool z = (psr & (1 << 22)) != 0;
   bool v = (psr & (1 << 21)) != 0;
   bool c = (psr & (1 << 20)) != 0;

   switch (cond)
   {
    case CC_N:
      return 0;
    case CC_E:
      return z;
    case CC_LE:
      return z | (n ^ v);
    case CC_L:
      return n ^ v;
    case CC_LEU:
      return c | z;
    case CC_LU:
      return c;
    case CC_NEG:
      return n;
    case CC_VS:
      return v;
    case CC_A:
      return TRUE;
    case CC_NE:
      return !z;
    case CC_G:
      return !(z | (n ^ v));
    case CC_GE:
      return !(n ^ v);
    case CC_GU:
      return !(c | z);
    case CC_GEU:
      return !c;
    case CC_POS:
      return !n;
    case CC_VC:
      return !v;
   }
```

```
    panic("cond botch in cond_holds()");
    return 0;
}



taddr_t
get_next_pc( Process *proc, taddr_t addr )
{
    sparc_inst_t    si;
    psr_t           psr;

    if (proc->readText(addr, (char *) &si, sizeof(si)) != 0)
        panic("Process::readText() failed in get_next_pc");

    switch (si.fmt1.op)
    {
    case 0:
        // Try to cope with annulled branches.  Comments in gdb say you add
        // four to the predicted target address in this case, so we do this.
        // This behaves better than before but it still gets things wrong (e.g.
        // when stepping over the first few lines of TypeId() when ups has been
        // compiled with gcc -O.
        //

        if (si.fmt2.a)
            addr += 4;

        switch ((btype_t) si.fmt2.op2)
        {
        case CBCCC:
            panic("cbcc NYI in get_next_pc");
            break;
        case FBFCC:
            psr = proc->getRegister(REG_FP_CONDITION_CODES);
            if (fp_cond_holds(psr, (fp_cond_t) si.fmt2.cond))
                return addr + (sign_ext(si.fmt2.disp22, 22) << 2);
            break;
        case BICC:
            psr = proc->getRegister(REG_CONDITION_CODES);
            if (cond_holds(psr, (cond_t) si.fmt2.cond))
                return addr + (sign_ext(si.fmt2.disp22, 22) << 2);
            break;
        default:
            break;
        }
        break;
    case 1:
        return addr + (si.fmt1.disp30 << 2);
    case 2:
        if (si.fmt3i.op3 == JMPL)
        {
            int             reg1;

            reg1 = proc->getRegister(si.fmt3i.rs1);
            if (si.fmt3i.i)
                return reg1 + sign_ext(si.fmt3i.simm13, 13);
            else
                return reg1 + proc->getRegister(si.fmt3r.rs2);
        }
        break;
    case 3:
        break;
    }
    return addr + 4;
}
```

```
jump_t *
get_jumps( taddr_t addr, const char *ctext, int len, int want_calls, int want_branches )
{
   static jump_t  *jtab;
   static int      jtab_size = 0;
   jumptype_t      jtype;
   sparc_inst_t   *p_si,
                  *lim,
                   si;
   bool            unconditional;
   int             njumps;

   if (jtab_size == 0)
   {
      jtab_size = 16;
      jtab = (jump_t *)malloc((jtab_size + 1) * sizeof(jump_t));
   }
   if (((int) ctext & 03) != 0 || (len & 03) != 0)
      panic("align/len botch in gj");
   p_si = (sparc_inst_t *) ctext;
   lim = (sparc_inst_t *) (ctext + len);

   njumps = 0;

   for (; p_si < lim; ++p_si, addr += 4)
   {
      taddr_t         jdest;

      si = *p_si;
      switch (si.fmt1.op)
      {
      case 0:
         switch ((btype_t) si.fmt2.op2)
         {
         case CBCCC:
         case FBFCC:
         case BICC:
            jdest = addr + (sign_ext(si.fmt2.disp22, 22) << 2);
            unconditional = FALSE;
            break;
         default:
            continue;
         }
         jtype = JT_BRANCH;
         break;
      case 1:
         jdest = addr + (si.fmt1.disp30 << 2);
         unconditional = TRUE;
         jtype = JT_CALL;
         break;
      case 2:
         if (si.fmt3i.op3 == JMPL)
         {
            jdest = 0;
            jtype = (si.fmt3i.rd != 0) ? JT_CALL : JT_BRANCH;
            unconditional = TRUE;
         } else
            continue;
         break;
      default:
         continue;
      }

      if ((want_calls && jtype == JT_CALL) ||
          (want_branches && jtype == JT_BRANCH))
      {
         if (njumps >= jtab_size)
         {
            jtab_size *= 2;
            jtab = (jump_t *)realloc((char *) jtab,
                                     (jtab_size + 1) * sizeof(jump_t));
         }
```

```
            jtab[njumps].ju_addr = addr;
            jtab[njumps].ju_type = jtype;
            jtab[njumps].ju_dstaddr = jdest;
            jtab[njumps].ju_unconditional = unconditional;
            ++njumps;
        }
    }
    jtab[njumps].ju_type = JT_END;
    return jtab;
}
```

```
/*-------------------------------------------------------------------
 |       FILE: bat.h
 |    PURPOSE: Header file for Bat class.
  -------------------------------------------------------------------*/
   fine BAT_H_INCLUDED

#include "msg.h"
#include "cmpx.h"
#include "target.h"
#include <strstream.h>
#include "command.h"

class BAToutstream : public ostrstream
{
 public:
   BAToutstream() : ostrstream() { bo_msg_handle=0; }

   ~BAToutstream();
   void send();
   static void setMsgHandle( MSG_HANDLE h ) { bo_msg_handle = h; }
   static MSG_HANDLE getMsgHandle()         { return bo_msg_handle; }
   static void send( char *msg );


 protected:
   static MSG_HANDLE     bo_msg_handle;
};


class BATdispstream : public ostrstream
{
 public:
   BATdispstream() : ostrstream() { bo_msg_handle=0; }

   ~BATdispstream();
   void send();
   static void setMsgHandle( MSG_HANDLE h ) { bo_msg_handle = h; }
   static MSG_HANDLE getMsgHandle()         { return bo_msg_handle; }
   static void send( char *msg );


 protected:
   static MSG_HANDLE     bo_msg_handle;
};


class Bat
{
 public:

   Bat( int argc, char **argv );
   static void BATinit( Bat *bat );

   Target *getTarget() { return ba_target; }

   void setCommand( Command *cmd ) {ba_current_command = cmd;}

   static void commandHandler( char *line, Bat *bat );
   static Bat *getCurrentBat() { return ba_current_bat; }
   void setFocus();

   Target        *ba_target;
   char         **ba_argv;
   MSG_HANDLE     ba_msg_handle;
   static Bat    *ba_current_bat;
   Command       *ba_current_command;
};


#endif
```

```
/*-------------------------------------------------------------------
 |       FILE: bat.C
 |    PURPOSE: Main application object for the Bat debug server.
  -------------------------------------------------------------------*/
  clude "symtab.h"
#include "proc.h"
#include "target.h"
#include "bat.h"
#include "stack.h"
#include "expression.h"
#include "command.h"

void print_stopres( stopres_t why )
{
   char *s;
   switch( why )
   {
    case( SR_SIG ):
    s = "Got a signal";
    break;
  case( SR_BPT ):
    s = "Hit a breakpoint";
    break;

  case( SR_DIED ):
    s = "Process exited";
    break;
  case( SR_SSTEP ):
    s = "Stopped after a single step";
    break;

  case( SR_USER ):
    s = "User requested stop";
    break;

  ase( SR_FAILED ):
    s = "Couldn't restart target (breakpoint problems)";
    break;

  default:
    s = "Don't know why?";

    }
}



void
Bat::BATinit( Bat *bat )
{
   int fd;

   char *name = new char[50];
   strcpy( name, bat->ba_argv[0] );

   // open the executable file...make sure it exists

   fd = Executable::open( name );
   if( fd<=0 )
   {
      BAToutstream bout;
      bout << name << " not found." << ends;
      bout.send();
      return;
   }

   // install the symbol table

   int i;
   if( i=SymTab::getAndInstallSymTabs( name, fd ) )
      return;
```

```
   // get main funcion
   int junk;
   Function *f = SymTab::nameToFunction("MAIN", &junk);
   if (f == NULL || f->getLanguage() != LANG_FORTRAN)
      f = SymTab::nameToFunction("main", &junk);
   if (f == NULL)
   {
      BAToutstream::send( "Cannot find function 'main' in the symbol table." );
      return;
   }


   // create the target object
   bat->ba_target = new Target( name, bat->ba_argv, environ );
   Command::setTarget( bat->ba_target );


   // add path of executable to path list for searching for files
   int len = strlen( bat->ba_argv[0] );
   char *fname = strsave( bat->ba_argv[0] );
   for( char *p=fname+len; *p!='/' && p!=fname; p-- )
      ;
   if( *p=='/' )
   {
      p[0] = '\0';
      SFile::addToSourcePath( fname );
   }


   //
   // set the focus to the first line in main
   //
   taddr_t pc = f->minBreakpointAddress();
   int lnum = f->addrToLNumber( pc );
   const char *file = f->getSourceFile()->getName();

   // send focus message
   MSGsenda( bat->ba_msg_handle, "BAT FOCUS %s %s %d", file, f->getName(), lnum );
}

void yyparse();


extern "C" {
   void init_input( char *s, MSG_HANDLE h );
}


/*-------------------------------------------------------------------------
  |     METHOD: commandHandler
  |
  |    PURPOSE: Parses and carries out a text command.
  |
  |       RUN   <executable-file   arguments>
  |       CONT
  |       STEP
  |       NEXT
  |
  |       STOP AT  line-number IN sourcefile
  |       STOP IN  function-name
  |
  |       STATUS
  |       DELETE   status-number
  |
  |       WHERE
  |       UP
  |       DOWN
  |
  |       PRINT     C-expression
  |       DISPLAY   C-expression
  |       UNDISPLAY C-expression
```

```
|
|       TRACE    ???...not done yet...
-------------------------------------------------------------------*/
void Bat::commandHandler( char *line, Bat *bat )

   BAToutstream bout;
   bout << "(BAT) " << line << ends;
   bout.send();

   char *arg1=NULL;

   if( !line )
      return;

   if( bat->ba_current_command )
      delete bat->ba_current_command;
   bat->ba_current_command = NULL;

   init_input( line, bat->ba_msg_handle );

   yyparse();

   if( bat->ba_current_command )
   {
      bat->ba_current_command->setCommandString( line );
      bat->ba_current_command->execute();
   }
}



typedef void (*COMMANDfptr)(...);
Bat::Bat( int argc, char **argv )
{
   ba_current_bat = this;

   if( argc < 2 )
   {
      cout << "USAGE:: bat executable [arg1] [arg2] ..." << endl;
      exit(1);
   }

   // connect with the message server
   ba_msg_handle = MSGconnect( NULL, NULL );
   BAToutstream::setMsgHandle( ba_msg_handle );
   BATdispstream::setMsgHandle( ba_msg_handle );

   if( !ba_msg_handle )
   {
      cout << "BAT: Couldn't connect to message server." << endl;
      exit(1);
   }

   // tell message server that the debug server is ready for action

   if( !MSGservice_register( ba_msg_handle, "BAT" ) )
   {
      cout << "BAT server is already running!  Exiting..." << endl;
      exit(1);
   }

   // register CAVE COMMAND with server
   int def_args[2];
   def_args[0] = NULL;
   def_args[1] = (int)this;

   MSGregister( ba_msg_handle, "CAVE COMMAND %1s", (COMMANDfptr)commandHandler, 2, (void **)def_
   def_args[0] = (int)this;
   MSGregister( ba_msg_handle, "BAT INITIALIZE", (COMMANDfptr)BATinit, 1, (void **)def_args );

   //--- save arguments ------------------------------------------
   typedef char *cptr;
```

```
   ba_argv = new cptr[argc-1];
   for( int j=1; j<argc; j++ )
      ba_argv[j-1] = strsave(argv[j]);
   ba_argv[argc-1] = NULL;


   //----- Bring up CAVE if not already up! ----------------------
   typedef char *cptr;
   cptr *my_argv = new cptr[argc+1];
   for( j=1; j<argc; j++ )
      my_argv[j] = argv[j];

   my_argv[0] = "cave";
   my_argv[argc] = NULL;

   // make sure the CAVE gui is up and running
   char execpath[50];
   getcwd( execpath, 50 );
   strcat( execpath, "/cave" );

   MSGservice_start( ba_msg_handle, "CAVE", NULL, execpath, my_argv );
}



BAToutstream::~BAToutstream()
{
   delete str();
}



void
BAToutstream::send()
{
   // str() may contain several '\n' characters
   // this just means to break these up into several lines.
   char *line = str();
   char *nextline=line;
   char *end = line+pcount();
   while( line < end && nextline )
   {
      nextline = index( line, '\n' );
      if( nextline == NULL )
         nextline = index( line, '\0' );
      if( nextline )
      {
         *nextline = '\0';
         nextline++;
      }

      if( strlen( line ) > 0 )
         MSGsenda( bo_msg_handle, "BAT OUTPUT %S", line );
      line = nextline;
   }
}

void
BAToutstream::send( char *msg )
{
   MSGsenda( bo_msg_handle, "BAT OUTPUT %S", msg );

}


BATdispstream::~BATdispstream()
{
   delete str();
}

void
BATdispstream::send()
```

```
{
   // str() may contain several '\n' characters
   // this just means to break these up into several lines.
   char *line = str();
   char *nextline=line;
   char *end = line+pcount();
   while( line < end && nextline )
   {
      nextline = index( line, '\n' );
      if( nextline == NULL )
         nextline = index( line, '\0' );
      if( nextline )
      {
         *nextline = '\0';
         nextline++;
      }

      if( strlen( line ) > 0 )
         MSGsenda( bo_msg_handle, "BAT DISPLAY %S", line );
      line = nextline;
   }
}

void
BATdispstream::send( char *msg )
{
   MSGsenda( bo_msg_handle, "BAT DISPLAY %S", msg );
}



void Bat::setFocus()
{
   char fullname[80];

   // get the current PC
   Process *proc = ba_target->getProcess();
   if( !proc ) return;

   // get the current stack frame
   Frame *frame = Stack::getCurrentStack()->getCurrentFrame();

   taddr_t pc = frame->fr_pc;

   // what function is this pc in?

   Function *f = frame->fr_func;

   // what line number?
   int lnum = frame->fr_lnum;

   // stream object for broadcasting output messages
   BAToutstream myout;

   SourceFile *sf = f->getSourceFile();
   if( !sf )
   {
      myout << "No source file information available for " << f->getName() << ends;
      myout.send();
      return;
   }

   strcpy( fullname, sf->getPathHint() );
   strcat( fullname, sf->getName() );

   // send focus message
   MSGsenda( ba_msg_handle, "BAT FOCUS %s %s %d", fullname, f->getName(), lnum );

   myout << "Stopped in " << fullname << " at line " << lnum << " in function " << f->getName()
   myout.send();
```

```
    // do whatever needs to be done with displaying expressions
    Display::DoDisplays();
}

main( int argc, char **argv )
{

  // create the bat server
  Bat *bat = new Bat(argc, argv );

  // now process messages til it hurts
  while(1) CMPXselect(-1);
}
```

```
#ifndef BLOCK_H_INCLUDE
#define BLOCK_H_INCLUDE

  ------------------------------------------------------------------------
  |      CLASS: Block
  |    PURPOSE: The information for a block consists of the start/end
  |             line numbers for the block, a pointer to the list of
  |             variables declared in the block, a pointer to the next
  |             block at the same level as this one, and a pointer to a
  |             list of blocks declared in this one, and a list of types
  |             declared in this block.  Finally a pointer to the parent
  |             block.
  |
  ------------------------------------------------------------------------*/


class Block
{
 public:
   Block( Block *parent );

   void iterateOverVars(void (*func)(Variable *v, char *c_args), char *args );
   void pushTypedefsAndAggrs( Block *bl );

   // access methods
   Variable *getVars() { return bl_vars;}
   void setVars( Variable *vars ) { bl_vars = vars;}

   int getStartLNumber() { return bl_start_lnum; }
   void setStartLNumber( int lnum ) { bl_start_lnum = lnum; }

   int getEndLNumber() { return bl_end_lnum; }
   void setEndLNumber( int lnum ) { bl_end_lnum = lnum; }

   typedef_t *getTypeDef() { return bl_typedefs;}
   void setTypeDef( typedef_t *t) { bl_typedefs = t;}

   aggr_or_enum_def_t *getAggrOrEnum() { return bl_aggr_or_enum_defs; }
   void setAggrOrEnum( aggr_or_enum_def_t *ae ) { bl_aggr_or_enum_defs = ae; }

   Block *getNext() { return bl_next; }
   void setNext( Block *next ) { bl_next = next; }

   Block *getSubBlocks() { return bl_blocks; }
   void setSubBlocks( Block *sbl ) { bl_blocks = sbl; }



   int                     bl_start_lnum;          // first line #
   int                     bl_end_lnum;            // last line #
   class Variable          *bl_vars;               // vars declared at in this block
   typedef_t               *bl_typedefs;           // typedefs declared in this block
   aggr_or_enum_def_t      *bl_aggr_or_enum_defs;
   class Block             *bl_next;               // next block at this level
   class Block             *bl_blocks;             // sub blocks
   class Block             *bl_parent;             // parent block
};
typedef class Block block_t;

//  Maximum block nesting level.
#define MAX_BLOCK_LEVEL 32



   dif
```

```
/*-----------------------------------------------------------------
 |       FILE: breakpoint.h
 |    PURPOSE: Breakpoint class
  -----------------------------------------------------------------*/

#ifndef BREAKPOINT_H_INCLUDED
#define BREAKPOINT_H_INCLUDED

#include "utils.h"
#include "gendefs.h"
#include "sfile.h"
#include "/pro/forest/basis/src/list.H"
#include "/pro/forest/basis/src/dynarray.H"
#include "preamble.h"
#include <errno.h>
#include <iostream.h>
#include "mreg.h"
#include "proc.h"
#include <strstream.h>


class Breakpoint;
typedef Breakpoint *bpPtr;
class_List(bpPtr);
typedef List(bpPtr) BreakpointList;


//typedef struct bpst
class Breakpoint
{
 public:
   //  Create an uninstalled breakpoint at address addr.
   //
   Breakpoint( taddr_t addr, char *description=NULL );

   //  Remove a breakpoint. Uninstall it first if necessary.
   //
   ~Breakpoint();

   //  If there is a breakpoint installed at addr in proc, return the breakpoint.
   //  Otherwise, return 0.
   //
   static Breakpoint *getBreakpointAtAddress( Process *proc, taddr_t addr );

   //  Run through the breakpoint list marking breakpoints as not installed.
   //  Called when the target process exits.
   //
   static void markAsUninstalled( Process *proc );

   //  Return the breakpoint corresponding to address addr, even if breakpoint
   //  is not currently installed.
   //
   //  If there is no breakpoint at addr, return 0.
   //
   static Breakpoint *addrToBreakpoint( taddr_t addr );
   taddr_t breakpointToAddr()                  { return bp_addr; }

   //  Install a breakpoint (e.g. write TRAP opcode into target).
   //
   int install( Process *proc );

   //  Install any uninstalled breakpoints.
   //
   static int installAllBreakpoints( Process *proc );

   //  Uninstall a breakpoint.
   //
   int uninstall();

   //  Uninstall any breakpoints installed in proc.
   //
   static int uninstallAllBreakpoints( Process *proc );
```

```
    // Get and set User data for a breakpoint
    long getUserData( )                 { return bp_user_data; }
    void setUserData( long data )       { bp_user_data = data; }

    //  Return non zero if breakpoint is installed.
    //
    int breakpointIsInstalled()         { return (bp_proc != 0); }

    // Write out a textual description of the breakpoint
    // to the supplied stream.
    void describe( int& id, ostrstream& rout );

    // get the breakpoint associated with an id, or NULL
    static Breakpoint *idToBreakpoint( int id );

    // Get the current list of all breakpoints
    static BreakpointList *getBreakpointList() { return bp_list; }

    // get the address of this breakpoint
    taddr_t getAddress() { return bp_addr; }

  protected:
    int                  bp_id;         // id if user created...otherwise, -1
    static BreakpointList *bp_list;     // list of all breakpoints
    Process              *bp_proc;      // process that bpt is (will be) inserted in
    taddr_t               bp_addr;      // the text address of the breakpoint
    opcode_t              bp_code;      // the opcode that has been replaced
    long                  bp_user_data; // currently not used...hook for tracing, etc.
    char                 *bp_description;// command asociated with the breakpoint
    static int            bp_next_id;   // bp id counter
};
typedef class Breakpoint bp_t;


#endif
```

```
/*-------------------------------------------------------------------
 |       FILE: bp.c
 |    PURPOSE: Breakpoint class methods
 -------------------------------------------------------------------*/

#include "proc.h"
#include "breakpoint.h"

int Breakpoint::bp_next_id = 1;
BreakpointList *Breakpoint::bp_list = new BreakpointList;


//  Create an uninstalled breakpoint at address addr.
//  Include a description of the command that caused the breakpoint
//  if it is a user created breakpoint (for use with status/delete).
Breakpoint::Breakpoint( taddr_t addr, char *descr )
{
   // add the breakpoint to the breakpoint list
   bp_list->listAppend( this );

   bp_addr = addr;
   bp_proc = 0;
   bp_user_data = 0;
   bp_description = strsave( descr );
   bp_id = ( bp_description==NULL ) ? -1 : bp_next_id++;
}

// Remove a breakpoint. Uninstall it first if necessary.
//
Breakpoint::~Breakpoint()
{
   // if its installed, then uninstall it.
   if( breakpointIsInstalled() )
      if( uninstall() != 0 )
         panic( "Breakpoint::~Breakpoint() could not uninstall breakpoint." );

   // remove it from the list
   bp_list->listRemob( this );
}

//  Install a breakpoint (e.g. write TRAP opcode into target).
//
int
Breakpoint::install( Process *proc )
{
   if( proc->textSwap( bp_addr, (opcode_t) 0, &bp_code) != 0 )
   {
      cout << "can't install breakpoint" << endl;
      return -1;
   }
   bp_proc = proc;
   return 0;
}

//  Uninstall a breakpoint.
//
int
Breakpoint::uninstall()
{
   if( bp_proc->textSwap( bp_addr, bp_code, (opcode_t *) NULL) != 0 )
   {
      cout << "can't uninstall breakpoint" << endl;
      return -1;
   }
   bp_proc = 0;
   return 0;
}

//  Uninstall any breakpoints installed in proc.
//
int
Breakpoint::uninstallAllBreakpoints( Process *proc )
```

```
{
   ListIter(bpPtr) iterator;

   for( iterator=*bp_list; iterator.iterMore(); iterator++ )
   {
      Breakpoint *bp = *iterator;
      if( bp->bp_proc == proc )
         if( bp->uninstall() )
            return -1;
   }

   return 0;
}

//  Install any uninstalled breakpoints.
//
int
Breakpoint::installAllBreakpoints( Process *proc )
{
   ListIter(bpPtr) iterator;

   for( iterator=*bp_list; iterator.iterMore(); iterator++ )
   {
      Breakpoint *bp = *iterator;
      if( !bp->breakpointIsInstalled() )
         if( bp->install( proc ) )
            return -1;
   }
   return 0;
}

//  If there is a breakpoint installed at addr in proc, return the breakpoint.
//  Otherwise, return 0.
//
Breakpoint *
Breakpoint::getBreakpointAtAddress( Process *proc, taddr_t addr )
{
   ListIter(bpPtr) iterator;

   for( iterator=*bp_list; iterator.iterMore(); iterator++ )
   {
      Breakpoint *bp = *iterator;
      if( bp->bp_proc == proc && bp->bp_addr == addr )
         return bp;
   }

   return NULL;
}

//  Return the breakpoint corresponding to address addr, even if breakpoint
//  is not currently installed.
//
//  If there is no breakpoint at addr, return 0.
//
Breakpoint *
Breakpoint::addrToBreakpoint( taddr_t addr )
{
   ListIter(bpPtr) iterator;

   for( iterator=*bp_list; iterator.iterMore(); iterator++ )
   {
      Breakpoint *bp = *iterator;
      if( bp->bp_addr == addr )
         return bp;
   }

   return NULL;
}

//  Run through the breakpoint list marking breakpoints as not installed.
//  Called when the target process exits.
//
```

```
void
Breakpoint::markAsUninstalled( Process *proc )
{
   ListIter(bpPtr) iterator;

   for( iterator=*bp_list; iterator.iterMore(); iterator++ )
   {
      Breakpoint *bp = *iterator;
      if( bp->bp_proc == proc )
         bp->bp_proc = 0;
   }
}

// returns the id of the breakpoint and appends descriptive
// text to rout.  If not associated with a command, then id==-1
// and nothing is appended to rout.
void Breakpoint::describe( int& id, ostrstream& rout )
{
   rout << bp_description << ends;
   id = bp_id;
}

Breakpoint *
Breakpoint::idToBreakpoint( int id )
{
   ListIter(bpPtr) iterator;

   for( iterator=*bp_list; iterator.iterMore(); iterator++ )
   {
      Breakpoint *bp = *iterator;
      if( bp->bp_id == id )
         return bp;
   }

   return (Breakpoint *)NULL;
}
```

```
/*-------------------------------------------------------------------
 |       FILE: command.h
 |    PURPOSE: Command object class definitions.
 |     NOTES:  One object class for each command all derived from
 |             the base class, Command.
 -------------------------------------------------------------------*/
#ifndef COMMAND_H_INCLUDED
#define COMMAND_H_INCLUDED

#include "msg.h"
#include "cmpx.h"
#include "target.h"
#include <strstream.h>
#include "expression.h"

class BAToutstream;
class Bat;

class Command
{
 public:
   Command() {;}
   ~Command(){delete co_command_str;}

   static void setTarget( Target *t ) {co_target = t; } // GROSS!!!!

   virtual void execute()=0;
   void setCommandString( char *comstr ){ co_command_str = strsave( comstr ); }

 protected:
   void           setFocus();
   void           doWhy();
   char          *co_command_str; // string representing the command
   static Target *co_target;
   static bool    co_bomb_on;
   static Frame  *co_bomb_frame;
};

// run [executable [args...]]
class Run : public Command
{
 public:
   Run();
   virtual void execute();
};

// step
class Step : public Command
{
 public:
   Step();
   virtual void execute();
};

// continue
class Continue : public Command
{
 public:
   Continue();
   virtual void execute();
};

// next
class Next : public Command
{
   blic:
   Next();
   virtual void execute();
};

// up
class Up : public Command
```

```
{
 public:
   Up();

   virtual void execute();
};

// down
class Down : public Command
{
 public:
   Down();

   virtual void execute();
};

// where
class Where : public Command
{
 public:
   Where();

   virtual void execute();
};

// print expression
class Print : public Command
{
 public:
   Print( Expression *exp );
   ~Print() { if( co_expr ) delete co_expr; }

   virtual void execute();

 protected:
   Expression *co_expr;
};

// display expression
typedef Expression *EPtr;
class_List(EPtr);
typedef List(EPtr) ExpressionList;

class Display : public Command
{
 public:
   Display( Expression *exp );

   virtual void execute();

   static void DoDisplays();


 protected:
   static ExpressionList  *co_explist;  // list of expressions to be displayed
   bool                    co_ok;
};

// trace ??
class Trace : public Command
{
 public:
   Trace();

   virtual void execute();
};

// status
class Status : public Command
{
 public:
   Status();
```

```
    virtual void execute();
};

// delete
class Delete : public Command
{
 public:
   Delete( int num );

   virtual void execute();

 protected:
   int     co_id;
};


// ignore
class Ignore : public Command
{
 public:
   Ignore( int signal );
   virtual void execute();
 protected:
   int     co_signal;
};

// Catch
class Catch : public Command
{
 public:
   Catch( int signal );
   virtual void execute();

 protected:
   int     co_signal;

};

// quit
class Quit : public Command
{
 public:
   Quit();

   virtual void execute();
};

// stop in <function>
// stop at <line_number> in <file_name>
class Stop : public Command
{
 public:
   Stop( int lnum, char *file_name );
   Stop( char *function_name );

   virtual void execute();

 protected:
   int       co_lnum;
   char     *co_file;
   char     *co_func_name;
};

// file <file_name>
//
class File : public Command
{
 public:
   File( char *file_name );

   virtual void execute();
```

```
%{
#include "defs.h"
typedef void Expression;
   clude "y.tab.h"
   clude "msg.h"
#include "cmpx.h"

extern char *   strcpy();

%}

%%
[ \t\n] ;
"->"                {return( ARROW );}
"AT"|"at"           {return( AT );}
"IN"|"in"           {return( IN );}
"CATCH"|"catch"     {return( CATCH );}
"CONT"|"cont"       {return( CONT ); }
"DELETE"|"delete"   {return( DELETE );}
"DOWN"|"down"       {return( DOWN ); }
"DISPLAY"|"display" {return( DISPLAY ); }
"FILE"|"file"       {return( FILE ); }
"IGNORE"|"ignore"   {return( IGNORE );}
"NEXT"|"next"       {return( NEXT ); }
"PRINT"|"print"     {return( PRINT );}
"QUIT"|"quit"       {return( QUIT ); }
"RUN"|"run"         {return( RUN );  }
"STEP"|"step"       {return( STEP ); }
"STATUS"|"status"   {return( STATUS );}
"STOP"|"stop"       {return( STOP ); }
"TRACE"|"trace"     {return( TRACE );}
"UP"|"up"           {return( UP  );  }
"WHERE"|"where"     {return( WHERE );}
[^-9]+              { yylval.y_long = atol( yytext ); return( INT );  }
   zA-Z1-9/_]+".c"  { strcpy( yylval.y_string, yytext ); return( FILENAME ); }
a-zA-Z1-9/_]+".h"   { strcpy( yylval.y_string, yytext ); return( FILENAME ); }
[a-zA-Z1-9/_]+      { strcpy( yylval.y_string, yytext ); return( STRING ); }
"."                 { return DOT; }
"*"                 { return STAR; }
"&"                 { return AMPERSAND; }
"["                 { return LBRAC; }
"]"                 { return RBRAC; }
%%

static char             OutputBuffer[1024];
static char OutputPos;
static char CurrentInput[1024];
static int  CurrentPos;
MSG_HANDLE  msg_handle;

#undef input
#undef unput
#undef output
int
yywrap()
{
   return (1);
}

void
yyerror( msg )
char *msg;
{
   MSGsenda( msg_handle, "BAT OUTPUT %S", msg );


int input()
{
  char c;

  c = CurrentInput[CurrentPos];
  if( c=='\0' )
```

```
    return( 0 );
  else
  {
     CurrentPos++;
     return( (int)c );
  }
}


void output( c )
 char c;
{
   if( c=='\n' || c=='\0' )
   {
      OutputBuffer[OutputPos] = '\0';
      MSGsenda( msg_handle, "BAT OUTPUT %S", OutputBuffer );
      OutputPos = 0;
   }
   else
      OutputBuffer[OutputPos++] = c;
}


void unput( c )
char c;
{
   CurrentInput[--CurrentPos] = c;
}

void init_input( s, h )
char *s;
MSG_HANDLE h;
{
   strcpy( CurrentInput, s );
   CurrentPos = 0;
   OutputPos  = 0;
   msg_handle = h;
}
```

```
%{

/*
   Yacc grammar for debugger commands.


#include "symtab.h"
#include "expression.h"
#include "bat.h"
#include "defs.h"


extern "C" {
   void yyerror( char *msg );
   int yylex();
}


%}

%start command

%term
    ALIAS AND ASSIGN AT CALL CATCH CONT DEBUG DELETE DIV DISPLAY DOWN DUMP
    EDIT FILE FUNC GRIPE HELP IF IGNORE IN LIST MOD NEXT NEXTI NIL NOT OR
    PRINT PSYM QUIT RERUN RETURN RUN SET SH SKIP SOURCE STATUS STEP STEPI
    STOP STOPI TRACE TRACEI UNALIAS UNSET UP USE
    WHATIS WHEN WHERE WHEREIS WHICH

%term INT CHAR REAL NAME STRING FILENAME
%term ARROW DOT STAR AMPERSAND LBRAC RBRAC

%right INT
   nary REDIRECT
%nary '<' '=' '>' '!' IN
%left '+' '-' OR
%left UNARYSIGN
%left '*' STAR '/' DIV MOD AND
%left '\\'
%left NOT '(' '[' DOT '^' ARROW



%union {
    void *y_command;
    Expression *y_node;
    int y_int;
    long y_long;
    char y_char;
    double y_real;
    char y_string[100];
    Boolean y_bool;
};

%type <y_long>      INT signal
%type <y_char>      CHAR
%type <y_real>      REAL

%type <y_command>   DELETE CATCH CONT DOWN IGNORE QUIT PRINT RUN STATUS STOP TRACE DISPLAY UP WH
%type <y_string>    STRING FILENAME
%type <y_string>    ALIAS AND ASSIGN AT CALL
%type <y_string>    DEBUG DIV DUMP
%type <y_string>    EDIT FUNC GRIPE HELP IF IN LIST MOD
   pe <y_string>    NEXTI NIL NOT OR
   pe <y_string>    PSYM RERUN RETURN SET SH SKIP SOURCE
%type <y_string>    STEP NEXT STEPI STOPI TRACEI
%type <y_string>    UNALIAS UNSET USE WHATIS WHEN WHEREIS WHICH
%type <y_node>      exp AMPERSAND STAR constant
%type <y_command>   command
%%
```

```
command:
/*    ASSIGN exp '=' exp
 *{
 *      $$ =
 *}
 *|
 */
    CATCH signal
{
   $$ = (void *)new Catch( $2 );
   Bat::getCurrentBat()->setCommand( (Command *)$$ );

}
|
    CONT
{
   $$ = (void *)new Continue();
   Bat::getCurrentBat()->setCommand( (Command *)$$ );
}
|
   DELETE INT
{
   $$ = (void *)new Delete( $2 );
   Bat::getCurrentBat()->setCommand( (Command *)$$ );
}
|
   DISPLAY exp
{
   $$ = (void *)new Display( $2 );
   Bat::getCurrentBat()->setCommand( (Command *)$$ );
}
|
   STATUS
{
   $$ = (void *)new Status();
   Bat::getCurrentBat()->setCommand( (Command *)$$ );
}
|
    DOWN
{
   $$ = (void *)new Down( );
   Bat::getCurrentBat()->setCommand( (Command *)$$ );
}
|
    FILE FILENAME
{
   $$ = (void *)new File( $2 );
   Bat::getCurrentBat()->setCommand( (Command *)$$ );
}
|

    IGNORE signal
{
   $$ = (void *)new Ignore( $2 );
   Bat::getCurrentBat()->setCommand( (Command *)$$ );
}

|
    QUIT
{
   $$ = (void *)new Quit();
   Bat::getCurrentBat()->setCommand( (Command *)$$ );
}
|
    RUN
{
   $$ = (void *)new Run();
   Bat::getCurrentBat()->setCommand( (Command *)$$ );
}
|
    STEP
```

```
{
   $$ = (void *)new Step();
   Bat::getCurrentBat()->setCommand( (Command *)$$ );
}
|
    NEXT
{
   $$ = (void *)new Next();
   Bat::getCurrentBat()->setCommand( (Command *)$$ );
}
|
    STOP IN STRING
{
   $$ = (void *)new Stop( $3 );
   Bat::getCurrentBat()->setCommand( (Command *)$$ );
}
|
   STOP AT INT IN FILENAME
{
      $$ = (void *)new Stop( $3, $5 );
      Bat::getCurrentBat()->setCommand( (Command *)$$ );
}
|
    UP
{
   $$ = (void *)new Up();
   Bat::getCurrentBat()->setCommand( (Command *)$$ );
}
|
    PRINT exp
{
      $$ = (void *)new Print( $2 );
      Bat::getCurrentBat()->setCommand( (Command *)$$ );
      /*        $$ = build(O_PRINT, $2); */
(
    WHERE
{
      $$ = (void *)new Where();
      Bat::getCurrentBat()->setCommand( (Command *)$$ );
}
;


signal:
    INT
{
        $$ = $1;
}
;


arglist:
     arglist arg
|
    arg
;
arg:
    NAME
|
    STRING
;


c  stant:
    INT
{
        $$ = new ExpConstant( $1 );
}
;

exp:
```

```
    STRING
{
   $$ = (Expression *)new ExpVariable( (const char *)$1 );
}
|
    exp LBRAC exp RBRAC
{
   $$ = new ExpArrayElement( $1, $3 );
}
|
    exp DOT STRING

{
   $$ = (Expression *)new ExpDot( $1, $3 );
}
|
    exp ARROW STRING
{
   $$ = (Expression *)new ExpArrow( $1, $3 );
}
|
    STAR exp %prec UNARYSIGN
{
   $$ = (Expression *)new ExpStar( $2 );
}
|
    AMPERSAND exp %prec UNARYSIGN
{
   $$ = (Expression *)new ExpAmpersand( $2 );
}
|
    constant
{
   $$ = $1;
}

;
```

```
/*-------------------------------------------------------------------
 |       FILE: command.C
 |    PURPOSE: Command object class.
     NOTES:    One object class for each command all derived from
               the base class, Command.
 -------------------------------------------------------------------*/
#include "symtab.h"
#include "proc.h"
#include "target.h"
#include "bat.h"
#include "stack.h"
#include "expression.h"
#include "command.h"

Frame *Command::co_bomb_frame = new Frame();

Run::Run()
{
}


void
Command::doWhy()
{
   stopres_t why = Process::getWhyStopped();
   if( why==SR_SIG )
   {
      BAToutstream bout;

      int sig = (int)Process::getLastSignal();
      bout << "Signal " << Process::getSignalString( sig ) << " received." << ends;
      bout.send();

      if( !co_target || co_target->cannotContinue() )
      {
         Stack *s = Stack::getCurrentStack();
         Frame *f = s->getCurrentFrame();
         while( f->fr_lnum <= 0 )
         {
            if( !f->fr_next_up )
               break;
            f = f->fr_next_up;
         }
         MSGsenda( BAToutstream::getMsgHandle(),
                  "BAT EVENT ADD %s %s %s %d", "BOMB",
                  f->fr_func->getSourceFile()->getName(), f->fr_func->getName(), f->fr_lnum );
         *co_bomb_frame = *f;
         co_bomb_on     = TRUE;
      }
   }
   else if( why==SR_DIED )
      BAToutstream::send( "Target exited." );
}



void Run::execute()
{
   if( co_target->getProcess()!=NULL )
      co_target->detach();

   if( co_bomb_on )
   {
      co_bomb_on = FALSE;
      Frame *f = co_bomb_frame;
      MSGsenda( BAToutstream::getMsgHandle(),
               "BAT EVENT REMOVE %s %s %s %d", "BOMB",
               f->fr_func->getSourceFile()->getName(), f->fr_func->getName(), f->fr_lnum );
   }

   co_target->run( (Breakpoint *)NULL, RT_CONT );
   Bat::getCurrentBat()->setFocus();
```

```
    doWhy();
}


Continue::Continue()
{
}

void Continue::execute()
{
    if( !co_target->getProcess() )
    {
        BAToutstream bout;
        bout << "Cannot continue.  No target is running." << ends;
        bout.send();
        return;
    }

    co_target->run( (Breakpoint *)NULL, RT_CONT );
    Bat::getCurrentBat()->setFocus();

    doWhy();
}

Next::Next()
{


}

void Next::execute()
{
    if( !co_target->getProcess() )
    {
        BAToutstream bout;
        bout << "Cannot continue.  No target is running." << ends;
        bout.send();
        return;
    }

    co_target->run( (Breakpoint *)NULL, RT_NEXT );
    Bat::getCurrentBat()->setFocus();
    doWhy();
}

Step::Step()
{


}

void Step::execute()
{
    if( !co_target->getProcess() )
    {
        BAToutstream bout;
        bout << "Cannot continue.  No target is running." << ends;
        bout.send();
        return;
    }

    co_target->run( (Breakpoint *)NULL, RT_STEP );
    Bat::getCurrentBat()->setFocus();

    Bat::getCurrentBat()->setFocus();
    doWhy();
}

Up::Up()
{
}
```

```
void Up::execute()
{
   if( Stack::getCurrentStack()->up() )
   {
      BAToutstream bout;
      bout << "Already at top of stack." << ends;
      bout.send();
   }
   else
      Bat::getCurrentBat()->setFocus();

}

Down::Down()
{
}

void Down::execute()
{
   if( Stack::getCurrentStack()->down() )
   {
      BAToutstream bout;
      bout << "Already at bottom of stack." << ends;
      bout.send();
   }
   else
      Bat::getCurrentBat()->setFocus();
}
Where::Where()
{


}

void Where::execute()
{
   if( !co_target->getProcess() )
   {
      BAToutstream bout;
      bout << "Cannot print the stack.  No target is running." << ends;
      bout.send();
      return;
   }

   // get the pc
   Process *proc = co_target->getProcess();
   Stack *s = Stack::getCurrentStack();

   for( Frame *frame=s->getCurrentFrame(); frame; frame=frame->fr_next_up )
   {
      // stream object for broadcasting output messages
      BAToutstream myout;
      myout << "    " << frame->fr_func->getName() << "()";

      SourceFile *sf = frame->fr_func->getSourceFile();
      if( sf != NULL )
         myout << ", line " << frame->fr_lnum << " in \"" << sf->getName() << "\"";
      else
         myout << " at " << "0x" << hex << frame->fr_pc;
      myout << ends;

      // send it to the message server
      myout.send();
   }
}

Print::Print( Expression *exp )
{

   co_expr = exp;
```

```
   return;
}

void Print::execute()
{
   BAToutstream bout;

   if( !co_expr->ok() )
   {  bout << co_expr->getError() << ends;
      bout.send();
      return;
   }

   // evaluate the expression
   co_expr->evaluate();
   if( !co_expr->ok() )
      bout << co_expr->getError();
   else
   {
      bout << co_expr->getExpression() << " = ";
      co_expr->getResultString( bout );
   }

   bout << ends;
   bout.send();
   return;

}


/*-------------------------------------------------------------------------
 |     METHOD: Display
 |
 |    PURPOSE: Display command constructor.
 |             The display object class keeps a static list of display
 |             command objects which are currently active.  Everytime
 |             the target stops, the static routine, Display::DoDisplays().
 |             This command is responsible for executing each display
 |             command object, thus updating the expression's values.
 -------------------------------------------------------------------------*/
ExpressionList *Display::co_explist = new ExpressionList;

Display::Display( Expression *exp )
{
   // if there is no active stack frame, then backit
   if( !Stack::getCurrentStack() )
   {
      BAToutstream::send( "Cannot display expression.  No target is running." );
      co_ok = FALSE;
      return;
   }

   // simply add the expression to the explist
   co_ok = exp->ok();
   if( co_ok )
      co_explist->listAppend( exp );
   else
   {
      BAToutstream bout;
      bout << "Cannot evaluate expression, " << exp->getExpression() << "." << ends;
      bout << exp->getError() << ends;
      bout.send();
   }
}

void Display::execute()
{
   if( co_ok )
      DoDisplays();
}
```

```
void Display::DoDisplays()
{
   // clear the display
   MSGsenda( BAToutstream::getMsgHandle(), "BAT CLEARDISPLAY" );
   ListIter(EPtr) iterator;

   if( !co_target )
      return;

   for( iterator=*co_explist; iterator.iterMore(); iterator++ )
   {
      Expression *exp = *iterator;
      BATdispstream bout;

      // evaluate the expression
      exp->evaluate();
      bout << exp->getExpression() << " = ";
      if( !exp->ok() )
         bout << exp->getError();
      else
         exp->getResultString( bout );

      bout << ends;
      bout.send();
   }
}


// Trace not implemented yet.
Trace::Trace()
{
}

// Trace not implemented yet.
   i Trace::execute()

}

Status::Status()
{
}

void Status::execute()
{
   BreakpointList *bplist = Breakpoint::getBreakpointList();

   ListIter(bpPtr) iterator;

   int id;
   for( iterator=*bplist; iterator.iterMore(); iterator++ )
   {
      BAToutstream bout;
      ostrstream descr;

      Breakpoint *bp = *iterator;
      bp->describe( id, descr );
      if( id>0 )
      {
         char *p = descr.str();
         bout << "(" << id << ") " << p << ends;
         bout.send();
         delete p;
      }
   }



Delete::Delete( int num )
{
   co_id = num;
}
```

```
void Delete::execute()
{

   Breakpoint *bp = Breakpoint::idToBreakpoint( co_id );
   if( !bp )
   {
      BAToutstream bout;
      bout << "No events exist with identification number, " << co_id << "." << ends;
      bout.send();
      return;
   }

   // we need to know the file, function name, and line number of the breakpoint
   taddr_t addr = bp->getAddress();
   Function *f = SymTab::addrToFunc( addr );
   lno_t *lno = f->addrToLno( addr );

   // tell clients about removal of breakpoint
   MSGsenda( BAToutstream::getMsgHandle(),
             "BAT EVENT REMOVE %s %s %s %d", "BREAK",
             f->getSourceFile()->getName(), f->getName(), lno->getLNumber() );

   // delete the breakpoint
   delete bp;
}

Ignore::Ignore(int signal)
{
   co_signal = signal;
}

void Ignore::execute()
{
   int flags = Process::getSignalFlags( co_signal );
   flags &= ~SIG_CATCH;
   flags |= SIG_IGNORE;
   Process::setSignalFlags( co_signal, flags );
}

Catch::Catch(int signal)
{
   co_signal = signal;
}

void Catch::execute()
{
   int flags = Process::getSignalFlags( co_signal );
   flags &= ~SIG_IGNORE;
   flags |= SIG_CATCH;
   Process::setSignalFlags(co_signal, flags);
}

Stop::Stop( int lnum, char *file )
{
   co_lnum = lnum;
   co_file = strsave( file );
   co_func_name = NULL;
}

Stop::Stop( char *function )
{
   co_func_name = strsave( function );
   co_lnum = 0;
   co_file = NULL;
}

void Stop::execute()
{

   int rc = 0;
   Function *f;
```

```
    taddr_t addr;
    Breakpoint *bp=NULL;
    BAToutstream bout;

    if( co_func_name )
    {
       // STOP IN function

       // find the address of the function
       int ambig;
       f = SymTab::nameToFunction( (const char *)co_func_name, &ambig );
       if( !f )
       {
          // print a syntax error
          bout << "Usage:: STOP AT <line number> IN <source file> or STOP IN <function name>" <<
          bout.send();
          return;
       }

       addr = f->minBreakpointAddress();
       co_file = (char *)f->getSourceFile()->getName();
       co_lnum = f->addrToLNumber( addr );

       bout << "Breakpoint set at line "<< co_lnum << " in " << co_file << " in function " << co_
    }
    else
    {
       // STOP AT line-number IN filename

       // get the Function and address where the breakpoint should fall
       int closest_lnum;
       if( (closest_lnum = SymTab::lNumToFuncAndAddr( co_file, co_lnum, &f, &addr )) < 0 )
       {
          BAToutstream bout;
          bout << "Cannot set breakpoint at line " << co_lnum << " in " << co_file << ends;
          bout.send();
          return;
       }
       co_func_name = (char *)f->getName();
       co_lnum = closest_lnum;

       bout << "Breakpoint set at line "<< co_lnum << " in " << co_file << " in function " << co_
    }
    bout.send();

    // things parsed ok...so set a breakpoint
    // is there already breakpoint at addr?
    bp = Breakpoint::addrToBreakpoint( addr );
    if( bp )
    {
       // There's already a breakpoint installed, so remove the breakpoint and alert
       // all clients.
       delete bp;

       // tell clients about removal of breakpoint
       MSGsenda( BAToutstream::getMsgHandle(),
                "BAT EVENT REMOVE %s %s %s %d", "BREAK", co_file, co_func_name, co_lnum );
    }
    else
    {
       // create a new breakpoint at addr and alert all clients
       bp = new Breakpoint( addr, co_command_str );

       // tell clients about removal of breakpoint
       MSGsenda(BAToutstream::getMsgHandle(),
                "BAT EVENT ADD %s %s %s %d", "BREAK", co_file, co_func_name, co_lnum );
    }

    return;
}

Quit::Quit()
```

```
{
}

void Quit::execute()
{
   // for now...just quit...
   co_target->detach();

   exit(0);
}

File::File( char *filename )
{
   co_file = strsave( filename );
}

void
File::execute()
{
   // send focus message
   MSGsenda( BAToutstream::getMsgHandle(), "BAT FOCUS %s %s %d", co_file, NULL, 1 );
}
```

```
// data.h - header file for data.c

#ifndef VALUE_H_INCLUDED
   #define VALUE_H_INCLUDED

class Variable;
typedef union
{
   char           vl_char;
   unsigned char  vl_uchar;
   short          vl_short;
   unsigned short vl_ushort;
   int            vl_int;
   unsigned int   vl_uint;
   int            vl_long;
   unsigned long  vl_ulong;
   int            vl_ints[2];
   float          vl_float;
   double         vl_double;
   int            vl_logical;
   taddr_t        vl_addr;
} value_t;

class Value
{
 public:
   // variable of the value, and base value of structure if this is
   // a member variable.  Otherwise, addr==NULL.

   Value( Variable *va );
   ~Value();
   value_t getValue();

   protected:
   value_t        vl_u;
   Variable      *vl_var;        /* variable for this value */
};

int dgets(taddr_t addr, char *optr, int max_nbytes);
int dread(taddr_t addr, char *buf, int nbytes);
int dread_fpval (taddr_t addr, bool is_reg, bool is_double, char *buf);
int dwrite(taddr_t addr, const char *buf, int nbytes);
#endif
```

```
// data.c - higher level routines to read and write target data

#include <ctype.h>
#include "symtab.h"
#include "data.h"
#include "proc.h"

//  Read a string from the data area.
//
int
dgets( taddr_t addr, register char *optr, int max_nbytes )
{
   char            ibuf[4],
                  *olim;
   register char  *iptr;

   olim = optr + max_nbytes - 1;
   iptr = ibuf + sizeof(ibuf);
   while (optr < olim)
   {
      if (iptr == ibuf + sizeof(ibuf))
      {
         if (dread(addr, ibuf, sizeof(ibuf)) != 0)
            return -1;
         iptr = ibuf;
         addr += sizeof(ibuf);
      }
      if (*iptr == '\0')
         break;
      *optr++ = *iptr++;
   }
   *optr++ = '\0';
   return 0;
}


//  Nominal address of the registers. Recognised by dread()
//
//  Just pick a value that can't be confused with a stack or data address.
//
#define REG_ADDR                ((taddr_t)0xf0000000)

//  Maximum number of registers.
//
#define MAX_REGS                256

//  Is addr an encoded register address?
//
#define IS_REG_ADDR(addr)       (addr >= REG_ADDR && addr < REG_ADDR + MAX_REGS)


int
dread_fpval( taddr_t addr, bool is_reg, bool is_double, char *buf )
{
   int             res;
   fpval_t         fpval;

   Process *proc = Process::getCurrentProcess();

   if (IS_REG_ADDR(addr))
   {
      res = proc->readFloatingPointRegister( addr - REG_ADDR, is_double, &fpval);
   } else if (is_reg)
   {
      // A register that's been saved on the stack.
      res = proc->readFloatingPointValue( addr, is_double, &fpval );
   } else
      return dread(addr, buf, is_double ? sizeof(double) : sizeof(float));

   if (res != 0)
      return -1;
```

```
    if (is_double)
        *(double *) buf = fpval.d;
    else
        *(float *) buf = fpval.f;

    return 0;
}


//  Read n bytes into buf from the data or stack area of the target process.
//  We deduce from addr whether we are supposed to read the stack or
//  the data area.
//
//  Return 0 for success, -1 for failure.
//
int
dread( taddr_t addr, char *buf, int nbytes )
{
    taddr_t         regval;
    int             regno;

    Process *proc = Process::getCurrentProcess();

    if (IS_REG_ADDR(addr))
    {
        regno = (taddr_t)addr - REG_ADDR;

        if (proc->readRegister(regno, &regval) != 0)
            return -1;

        switch (nbytes)
        {
        case 8:
            // We assume this is a pair of registers
            memcpy(buf, (char *) &regval, 4);
            if (proc->readRegister(regno + 1, &regval) != 0)
                return -1;
            memcpy(buf + 4, (char *) &regval, 4);
            break;
        case 4:
            memcpy(buf, (char *) &regval, 4);
            break;
        case 2:
            memcpy(buf, ((char *) &regval) + 2, 2);
            break;
        case 1:
            *buf = ((char *) &regval)[3];
            break;
        default:
            panic("nbytes botch in dread");
            break;
        }
        return 0;
    }
    return proc->readData(addr, buf, nbytes);
}


int
dwrite( taddr_t addr, const char *buf, int nbytes )
{
    taddr_t         regval;
    int             regno;

    Process *proc = Process::getCurrentProcess();

    if(proc == 0)
        panic("dwrite called on a core file");

    if(IS_REG_ADDR(addr))
    {
```

```
        regno = (taddr_t)addr - REG_ADDR;

        switch (nbytes)
        {
        case 4:
            regval = *(int *) buf;
            break;
        case 2:
            regval = *(short *) buf;
            break;
        case 1:
            regval = *buf;
            break;
        default:
            panic("nbytes botch in dw");
            regval = UNSET;         /* to satisfy gcc */
            break;
        }
        return proc->setRegister(regno, regval);
    }
    return proc->writeData(addr, buf, nbytes);
}
```

```
#ifndef EXECUTABLE_H_INCLUDED
#define EXECUTABLE_H_INCLUDED

#include "strcache.h"

/*-------------------------------------------------------------------
 |      CLASS: Executable
 |    PURPOSE: Methods for object class Executable providing low level
 |             opening and reading of executables and their symbols
 |             and text.
 |
 |   CONTENTS: -methods-        -------------purpose-----------------
 |             Executable()     constructor for Executable object
 |             ~Executable()    destructor
 |             open             open an executable file
 |             adjustAddrOffset change address of offset
 |             getNSyms()       get number of symbols in executable
 |             getSym()         get nlist_t for a symbol
 |             getString()      get string from string section of exec
 |             findSym()        find symbol of particular type
 |             getSymString()   return the string for a symbol
 |             readText()       read from the text (code) section
 -------------------------------------------------------------------*/

#define STRPAGESIZE     256
typedef const char *strpage_t[STRPAGESIZE];

typedef struct nlist nlist_t;
#define SYMSIZE         sizeof(nlist_t)

// We fake N_DATA, N_TEXT and N_EXT in getsym and findsym, so define
// them here.
//
#define n_offset        n_un.n_strx
#define n_dbx_type      n_type


class Executable
{
 public:
    //  Open the a.out file execfile and check that it's reasonable.
    //
    //  If the file is OK, return a file descriptor referring to execfile
    //  open for reading and positioned at the start of the file.
    //
    //  Otherwise give an error message and return -1.

    static int open( const char *execfile );


    // Constructor
    //  Given an executable file name and some other information, make a
    //  Executable object which can be use to read symbols and their strings
    //  from the text file.

    Executable(const char *execfile,
          int        fd,
          off_t      syms_offset,
          int        nsyms,
          off_t      strings_offset,
          long       addr_to_fpos_offset );

    // Destructor
    ~Executable();

    //  Change the address offset of the executable by delta.

    void adjustAddrOffset( long delta ) { ex_addr_to_fpos_offset += delta; }


    //  Return the number of symbols in the text file referred associated
    //  with the Executable.
```

```
    int getNSyms() {return ex_nsyms;}

    //  Get symbol symnum from the symbol table, and copy to p_nm.

    void getSym( int symno, nlist_t& p_nm );

    //  Return a copy of the NUL terminated string at the given offset
    //  from the start of the strings for the executable.

    const char *getString( off_t offset );

    //  Search the symbol table starting from symbol number symno for a
    //  symbol whose type is in symset.  A symbol is wanted if symset[nm.n_type]
    //  is non zero.  If we find a wanted symbol, copy it to p_nm and return
    //  its symbol number.  Otherwise return a symbol number one larger than
    //  that of the last symbol in the table.
    //

    int findSym( int symno, nlist_t& p_res, const char *symset );

    //  Return the string for symbol symno.

    const char *getSymString( int symno );

    //  Read n bytes of text starting at virtual address addr from the
    //  a.out file.
    //

    int readText( taddr_t addr, char *buf, int nbytes );

protected:

    //
    // Convert a virtual text address to the offset into the a.out file.
    //
    off_t VirtualAddressToFilePosition( taddr_t addr ) { return addr - ex_addr_to_fpos_offset; }

    //  Fill the symbols buffer so that symbol symno is at the start of
    //  the buffer.

    void fillSymBuf( int symno );

    int                     ex_fd;                  // File descriptor of the a.out file
    off_t                   ex_addr_to_fpos_offset; // Offset in file of start of text
                                                    // file_pos = virt_addr - ex_addr_to_fpos
    class StrCache          *ex_strcache;           // Symbol strings cache handle
    strpage_t               **ex_strpage_tab;       // Table of cached sym string values
    off_t                   ex_syms_offset;         // File offset of start of symbols
    off_t                   ex_strings_offset;      // File offset of symtab strings
    int                     ex_nsyms;               // Number of symbols in this a.out file
    int                     ex_symbuf_nsyms;        // Size of symbol buffer in symbols
    nlist_t                 *ex_symbuf;             // Symbol buffer
    int                     ex_symbase;             // Symno of first symbol in buffer
    int                     ex_symlim;              // Symno of one after the last sym in the
};

typedef class Executable Executable_t;


#endif
```

```
/*-------------------------------------------------------------------
 |       FILE: executable.C
 |    PURPOSE: Methods for object class Executable providing low level
 |             opening and reading of executables and their symbols
 |             and text.
 |
 |   CONTENTS: -methods-         -------------purpose-----------------
 |             Executable()      constructor for Executable object
 |             ~Executable()     destructor
 |             open              open an executable file
 |             adjustAddrOffset change address of offset
 |             getNSyms()        get number of symbols in executable
 |             getSym()          get nlist_t for a symbol
 |             getString()       get string from string section of exec
 |             findSym()         find symbol of particular type
 |             getSymString()    return the string for a symbol
 |             readText()        read from the text (code) section
 -------------------------------------------------------------------*/

#include <a.out.h>
#include <errno.h>
#include <iostream.h>
#include "utils.h"
#include "executable.h"


//  Size of the buffer used by getsym, in units of SYMSIZE.
#define S_SYMBUF_NSYMS  256
#define R_SYMBUF_NSYMS  42

//  A name for this machine, used in the error message in open_textfile().
#define MNAME   "a Sun 4"


   ---------------------------------------------------------------------
    |     METHOD: open
    |
    |  PURPOSE:
    |    Open the a.out file execfile and check that it's reasonable.
    |    If the file is OK, return a file descriptor referring to execfile
    |    open for reading and positioned at the start of the file.
    |
    |    Otherwise give an error message and return -1.
    |
    |    RETURNS: A file descriptor or -1.
    |      NOTES:
    ---------------------------------------------------------------------*/
int
Executable::open( const char *execfile )
{
   struct exec hdr;
   int fd, n_read;
   struct stat stbuf;

   // open execfile

   if ((fd = ::open(execfile, 0)) < 0)
      cout <<"Can't open " << execfile <<endl;
   else if (fstat(fd, &stbuf) == 0 && ((stbuf.st_mode & S_IFMT) != S_IFREG))
      cout<< execfile << " is a not a regular file" <<endl;
   else if ((n_read = read(fd, (char *)&hdr, sizeof(hdr))) != sizeof(hdr))
   {
      if (n_read == -1)
         cout << "Error reading in " << execfile << endl;
      else
         cout << "EOF in a.out header of " << execfile << endl;
   }
   else if (N_BADMAG(hdr))
      cout << execfile << " is not " << MNAME << " a.out file" << endl;
   else if (hdr.a_syms == 0)
      cout << execfile << " has been stripped" << endl;
   else if (hdr.a_syms % SYMSIZE != 0)
```

```
      cout << "bad symbol table size " << hdr.a_syms << " (not a multiple of " << SYMSIZE << end
   else if (lseek(fd, L_SET, 0) == -1)
      cout << "can't lseek in " << execfile << endl;
   else
   {
      return fd;
   }

   (void) close(fd);
   return -1;
}

#undef MNAME

/*-----------------------------------------------------------------------
 |     METHOD: Executable constructor
 |
 |    PURPOSE: Given an executable file's name, a file descriptor to this
 |             file, offset into the file of the symbols, the number of
 |             symbols, the offset into the executable of the string
 |             section, and the amount to be subtracted from a virtual
 |             address to give you the equivalent file position in the
 |             text section, construct a new Executable
 -----------------------------------------------------------------------*/
Executable::Executable(const char *execfile,
             int fd,
             off_t syms_offset,
             int nsyms,
             off_t strings_offset,
             long addr_to_fpos_offset )
{
   // construct a string cache for the file
   ex_strcache = new StrCache( fd );

   // allocate & initialize page table
   int npages = nsyms / STRPAGESIZE + 1;
   ex_strpage_tab = (strpage_t **)malloc(sizeof(strpage_t *) * npages);
   for (int pagenum = 0; pagenum < npages; ++pagenum)
      ex_strpage_tab[pagenum] = NULL;

   // scarf away all this useful information for later!
   ex_fd = fd;
   ex_strings_offset = strings_offset;
   ex_syms_offset = syms_offset;
   ex_nsyms = nsyms;
   ex_symbuf_nsyms = S_SYMBUF_NSYMS;
   ex_symbuf = new nlist_t[ex_symbuf_nsyms];
   ex_symbase = ex_nsyms + 1;
   ex_symlim = 0;

   ex_addr_to_fpos_offset = addr_to_fpos_offset;
}


/*-----------------------------------------------------------------------
 |     METHOD: ~Executable object destructor
 |
 |    PURPOSE: Close down the executable file object and free all
 |             associated memory.
 |      NOTES:
 -----------------------------------------------------------------------*/
Executable::~Executable()
{
   delete ex_strcache;
   close( ex_fd );

   delete ex_symbuf;
   free( (char *)ex_strpage_tab );
}
```

```
/*-----------------------------------------------------------------------
       METHOD: fillSymBuf

    PURPOSE: Fill the symbols buffer of object so that symbol symno is
             at the start of the buffer.
      NOTES:
-----------------------------------------------------------------------*/
void
Executable::fillSymBuf( int symno )
{

   if (symno < 0 || symno >= ex_nsyms)
      panic("symno botch in Executable::fillSymBuf()");
   if (lseek(ex_fd, ex_syms_offset + symno * SYMSIZE, L_SET) == -1)
      panic("lseek failed in Executable::fillSymBuf()");

   int nsyms_to_read = ex_symbuf_nsyms;
   if (ex_nsyms - symno < nsyms_to_read)
      nsyms_to_read = ex_nsyms - symno;

   int n_read = read(ex_fd, (char *)ex_symbuf, nsyms_to_read * SYMSIZE);

   if (n_read == -1)
      panic("read error in symbol table");
   if (n_read == 0)
      panic("unexpected EOF in symbol table");
   if (n_read % SYMSIZE != 0)
      panic("n_read botch in Executable::fillSymBuf()");

   ex_symbase = symno;
   ex_symlim = ex_symbase + n_read / SYMSIZE;
}



/*-----------------------------------------------------------------------
     METHOD: getSym

    PURPOSE: Get symbol symnum from the executable file's symbol table
             and copies to the supplied nlist_t structure.
      NOTES:
-----------------------------------------------------------------------*/
void
Executable::getSym( int symno, nlist_t& p_nm )
{
   if (symno < ex_symbase || symno >= ex_symlim)
      fillSymBuf( symno );

   p_nm = ex_symbuf[ symno - ex_symbase ];
}


/*-----------------------------------------------------------------------
     METHOD: getString

    PURPOSE: Return the NULL terminated at the given offset from the
             start of the strings section of the executable.
-----------------------------------------------------------------------*/
const char *
Executable::getString( off_t offset )

   ong len;

   const char *s =
      ex_strcache->getString( ex_strings_offset + offset, '\0', len);
   if (s == NULL)
      panic("str botch in Executable::getString");
```

```
   return s;
}


/*-------------------------------------------------------------------------
 |     METHOD: findSym
 |
 |    PURPOSE: Search the executable file's symbol table starting from
 |             symbol number symno for a symbol whose type is in symset.
 |             A symbol is wanted if symset[nm.n_type] is non zero.  If
 |             we find a wanted symbol, copy it to p_nm and return its
 |             symbol number.  Otherwise, return a symbol number one
 |             larger than that of the last symbol in the table.
 |
 -------------------------------------------------------------------------*/
int
Executable::findSym( int symno, nlist_t& p_res, const char *symset )
{
   if( symno < ex_symbase )
      fillSymBuf( symno );

   nlist_t *p_nm = &ex_symbuf[symno - ex_symbase];

   for( ; symno < ex_nsyms; ++p_nm, ++symno )
   {
      if (symno >= ex_symlim)
      {
         fillSymBuf( symno );
         p_nm = ex_symbuf;
      }

      // if flag is set for this symbol's type, then return it
      if (symset[p_nm->n_type] != 0)
      {
         p_res = *p_nm;
         return symno;
      }
   }

   ex_symbuf_nsyms = R_SYMBUF_NSYMS;

   return symno;
}


/*-------------------------------------------------------------------------
 |     METHOD: getSymString
 |
 |    PURPOSE: Return the string for symbol, symno.
 -------------------------------------------------------------------------*/
const char *
Executable::getSymString( int symno )
{
   nlist_t nm;

   if (symno < 0 || symno >= ex_nsyms)
      panic("Executable::getSymString() botch...bad symbol number.");

   int pagenum = symno / STRPAGESIZE;
   strpage_t *page = ex_strpage_tab[ pagenum ];

   if (page == NULL)
   {
      page = (strpage_t *)malloc( sizeof(strpage_t) );
      for( int i = 0; i < STRPAGESIZE; ++i )
         (*page)[i] = NULL;
      ex_strpage_tab[pagenum] = page;
   }

   int slot = symno % STRPAGESIZE;
   if( (*page)[slot] == NULL )
```

```
    {
       const char *line;

       getSym( symno, nm );
       if( nm.n_offset == 0 )
          line = NULL;
       else
       {
          long len;

          line = ex_strcache->getString( ex_strings_offset + nm.n_offset, '\0', len);
          line = strsave( line );
       }
       (*page)[slot] = line;
    }
    return (*page)[slot];
}


/*-------------------------------------------------------------------------
  |     METHOD: readText
  |
  |    PURPOSE: Read n bytes of text from the executable file starting at
  |             virtual address addr.  This method is used whenever it is
  |             necessary to search through the code of the process.
  |             For example, it is used to find out where a floating point
  |             register is saved.
  -------------------------------------------------------------------------*/
int
Executable::readText( taddr_t addr, char *buf, int nbytes )
{
   off_t fpos = VirtualAddressToFilePosition( addr );
   if (lseek(ex_fd, fpos, 0) != fpos)
      return -1;
   return (read(ex_fd, buf, nbytes) == nbytes) ? 0 : -1;
}
```

```
/*-------------------------------------------------------------------
 |       FILE: expresion.h
 |    PURPOSE: Expression object class declaration.
   -------------------------------------------------------------------*/

#ifndef EXPRESSION_H_INCLUDED
#define EXPRESSION_H_INCLUDED

#include "symtab.h"
#include "data.h"
#include "target.h"
#include "stack.h"
#include <strstream.h>

class Expression
{
 public:
   Expression()                {ex_stack_frame=NULL; ex_ok=TRUE; ex_error=NULL;}
   ~Expression()               { if( ex_base ) delete ex_base; }

   bool ok()                   {return ex_ok;}
   const char *getError()      { return ex_error; }
   typecode_t getResultType()  { return ex_result_type; }

   // evaluate the expression
   virtual int evaluate()=0;

   // get result
   void getResultString( ostrstream& result_str );
   value_t *getResultValue()   { return &ex_result_value; }
   taddr_t getResultAddress()  { return ex_addr;          }

   const char *getExpression() { return ex_expr_str.str(); }

   // for indentation
   void indent( ostrstream &rout );

   // parse the expression
   void parse();

   // get the variable representing this expression
   Type *getType() { return ex_type; }

   Variable *findLocalVar( const char *vname );
   Variable *findGlobalVar( const char *vname );

   Type       *ex_type;              // type object for result of this expression
   typecode_t ex_result_type;        // typecode of result
   int ex_ok;                        // TRUE if no errors have taken place concerning this expr
   const char *ex_error;             // error message if ex_error==TRUE
   Variable   *ex_var;               // variable for this expression
   Frame      *ex_stack_frame;       // stack frame where the expression is active (only used f
   taddr_t     ex_addr;              // address in inferior processes from where value was fetc
   value_t     ex_result_value;      // value result of expression evaluation
   Expression *ex_base;              // base expression (if needed)
   ostrstream  ex_expr_str;          // string representing the expression being evaluated (lhs
   static      ex_indent_level;      // level of indentation for string results.
};


class ExpVariable : public Expression
{
 public:

   ExpVariable( const char *var_name );
   int evaluate();

};


class ExpDot : public Expression
{
```

```
 public:

    ExpDot( Expression *base, const char *field_name );
    int evaluate();

};

class ExpAmpersand : public Expression
{
 public:

    ExpAmpersand( Expression *exp );
    int evaluate();
};



class ExpArrow : public Expression
{
 public:

    ExpArrow( Expression *base, const char *field_name );
    int evaluate();
};


class ExpStar : public Expression
{
 public:

    ExpStar( Expression *base );
    int evaluate();
};



class ExpArrayElement : public Expression
{
 public:

    ExpArrayElement( Expression *base, Expression *index );
    int evaluate();
    void setIndex();

    Expression *ex_index;
};



class ExpConstant : public Expression
{
 public:

    ExpConstant( int con );
/*-------------------------
 *  ExpConstant( char con );
 *  ExpConstant( char *con );
 *  etc...
 *-------------------------*/
    int evaluate();
};

#endif
```

```
/*-------------------------------------------------------------------
 |       FILE: expresion.C
 |    PURPOSE: Expression object class methods.
  -------------------------------------------------------------------*/

#include "expression.h"
#include "bat.h"


/*------------------------------ ExpVariable --------------------------*/

/*-------------------------------------------------------------------
 |     METHOD: ExpVariable
 |
 |    PURPOSE: Constructor for expressions consisting of a variable.
 |             Find the variable.
 |             Set result type to type of variable.
 |             If no variable is found, then set ex_ok to FALSE
 |             and ex_error.
 -------------------------------------------------------------------*/
ExpVariable::ExpVariable( const char *vname ) : Expression()
{
   ex_stack_frame = NULL;

   ex_var = findLocalVar( vname );
   if( !ex_var )
      ex_var = findGlobalVar( vname );

   if( !ex_var )
   {
      ex_ok = FALSE;
      ex_error = strsave( "Could not find variable." );
      ex_expr_str << vname << ends;
   }
   else
   {
      // get the type of the result
      ex_type = ex_var->getType();
      ex_result_type = ex_type->getTypeCode();
      ex_expr_str << ex_var->getName() << ends;
   }
}


/*-------------------------------------------------------------------
 |     METHOD: evaluate()
 |
 |    PURPOSE: Evaluate the expression object. If this method returns 0
 |             then one may call getResultValue() or getResultString()
 |             to get the result.  Otherwise, getError() will return
 |             an error message.
 |
 -------------------------------------------------------------------*/
int ExpVariable::evaluate()
{
   int rc=0;
   ostrstream eout;

   // find the stack frame for the variable
   ex_stack_frame = NULL;
   Variable *var = findLocalVar( ex_var->getName() );
   if( !var )
      var = findGlobalVar( ex_var->getName() );

   if( !var )
   {
      ex_ok = FALSE;
      if( ex_error ) delete (void *)ex_error;
      eout << "Variable, " << ex_expr_str << ", is not active." << ends;
      ex_error = eout.str();
      return 1;
   }
```

```
   else
   {
      // get the type of the result
      ex_var = var;
      ex_type = ex_var->getType();
      ex_result_type = ex_type->getTypeCode();

      // get the result value
      ex_addr = ex_var->getValue( ex_stack_frame, ex_result_value, NULL );
      if( ex_addr==NULL )
      {
         eout << "Error reading " << ex_expr_str << " from inferior process." << ends;
         if( ex_error ) delete (void *)ex_error;
         ex_error = eout.str();
         ex_ok = FALSE;
         return 1;
      }
   }

   return 0;
}



/*------------------------------ ExpConstant ---------------------------*/

/*-------------------------------------------------------------------------
 |     METHOD: ExpConstant
 |
 |    PURPOSE: Constructor for expressions consisting of a variable.
 |             Find the variable.
 |             Set result type to type of variable.
 |             If no variable is found, then set ex_ok to FALSE
 |             and ex_error.
 -------------------------------------------------------------------------*/
ExpConstant::ExpConstant( int con ) : Expression()
{
   ex_stack_frame  = NULL;
   ex_result_type  = TY_INT;
   ex_type         = Type::codeToType( ex_result_type );
   ex_expr_str << con;
   ex_result_value.vl_int = con;
   ex_error = NULL;
}

/*-------------------------------------------------------------------------
 |     METHOD: evaluate()
 |
 |    PURPOSE: Evaluate the expression object. If this method returns 0
 |             then one may call getResultValue() or getResultString()
 |             to get the result.  Otherwise, getError() will return
 |             an error message.
 |
 -------------------------------------------------------------------------*/
int ExpConstant::evaluate()
{
   return 0;
}


/*------------------------------ ExpDot ------------------------------*/

/*-------------------------------------------------------------------------
 |     METHOD: ExpDot
 |
 |    PURPOSE: Constructor for unary dot expressions.
 |             "base.field_name".
 -------------------------------------------------------------------------*/
ExpDot::ExpDot( Expression *base, const char *field_name ) : Expression()
{
```

```
   ostrstream eout;
   ex_base = base;

   // if base if no good then bag it
   ex_ok = base->ok();
   if( !ex_ok )
   {
      if( ex_error ) delete (void *)ex_error;
      eout << base->getError();
      ex_error = eout.str();
      return;
   }

   // get the base variable and make sure it's a struct or union
   Type *btype = base->getType();
   if( btype->getTypeCode() != TY_STRUCT && btype->getTypeCode() != TY_UNION )
   {
      ex_ok = FALSE;
      if( ex_error ) delete (void *)ex_error;
      eout << "Bad dot expression. Base is not a structure or union." << ends;
      ex_error = eout.str();
      return;
   }

   // find the field variable in the bvar
   aggr_or_enum_def_t *ae = btype->getAggrOrEnumDef();
   Variable *fvar = NULL;
   for( fvar=ae->ae_u.aeu_aggr_members; fvar; fvar=fvar->getNext() )
      if( !strcmp( fvar->getName(), field_name ) )
         break;

   if( !fvar )
   {
      ex_ok = FALSE;
      if( ex_error ) delete (void *)ex_error;
      eout << "Bad dot expression. " << field_name << " is not a member of structure, "
           << ex_base->getExpression() << "." << ends;
      ex_error = eout.str();
      return;
   }

   // save this field variable
   ex_var  = fvar;
   ex_type = fvar->getType();
   ex_result_type = ex_var->getType()->getTypeCode();
   ex_expr_str << ex_base->getExpression() << "." << ex_var->getName() << ends;
}

/*-------------------------------------------------------------------------
 |     METHOD: evaluate()
 |
 |    PURPOSE: Evaluate the expression object. If this method returns 0
 |             then one may call getResultValue() or getResultString()
 |             to get the result.  Otherwise, getError() will return
 |             an error message.
 |
 -------------------------------------------------------------------------*/
int ExpDot::evaluate()
{
   // verify that all is ok!
   if( !ok() || !ex_base->ok() )
      return 1;

   ostrstream eout;

   // get the address of the base variable
   if( ex_base->evaluate() )
   {
      if( ex_error ) delete (void *)ex_error;
      eout << "Error reading " << ex_expr_str << " from inferior process." << ends;
      ex_error = eout.str();
      ex_ok = FALSE;
```

```
      return 1;
   }
   value_t *bvalue = ex_base->getResultValue();

   // bvalue->vl_addr contains the address of the structure or union (the base address)

   // get the value of the field
   ex_addr = ex_var->getValue( ex_stack_frame, ex_result_value, bvalue->vl_addr );
   if( ex_addr == NULL )
   {
      if( ex_error ) delete (void *)ex_error;
      eout << "Error reading " << ex_expr_str << " from inferior process." << ends;
      ex_error = eout.str();
      ex_ok = FALSE;
      return 1;
   }

   return 0;
}


/*------------------------------ ExpArrayElement ------------------------------*/

/*-----------------------------------------------------------------------
 |     METHOD: ExpArrayElement
 |
 |    PURPOSE: Constructor for array expressions.
 |             "base_exp[ index_exp ]"
 -----------------------------------------------------------------------*/
ExpArrayElement::ExpArrayElement( Expression *base, Expression *index ) : Expression()
{
   ostrstream eout;
   ex_base = base;
   ex_index = index;

   // if base if no good then bag it
   ex_ok = base->ok();
   if( !ex_ok )
   {
      eout << base->getError();
      ex_error = eout.str();
      return;
   }


   // get the base variable and make sure it's an array
   Type *btype = base->getType();
   if( btype->getTypeCode() != DT_ARRAY_OF )
   {

      ex_ok = FALSE;
      eout << "Bad array expression. " << ex_base->getExpression() << " does not evaluate to an
      ex_error = eout.str();
      return;
   }

   // make sure the index expression evaluates to an int
   typecode_t rcode = ex_index->getResultType();
   if( rcode!=TY_SHORT && rcode!=TY_USHORT && rcode!=TY_INT &&
       rcode!=TY_UINT  && rcode!=TY_LONG   && rcode!=TY_ULONG )
   {
      ex_ok = FALSE;
      eout << "Bad array index.  " << ex_index->getExpression() << " is not an integer." << ends
      ex_error = eout.str();
      return;
   }

   // save this field variable
   ex_type = ex_base->getType()->getBase();
   ex_result_type = ex_type->getTypeCode();
   ex_expr_str << ex_base->getExpression() << "[" << ex_index->getExpression() << "]" << ends;
}
```

```
/*-------------------------------------------------------------------------
 |     METHOD: evaluate()
 |
 |    PURPOSE: Evaluate the expression object. If this method returns 0
 |             then one may call getResultValue() or getResultString()
 |             to get the result.  Otherwise, getError() will return
 |             an error message.
 |
 -------------------------------------------------------------------------*/
int ExpArrayElement::evaluate()
{
   // verify that all is ok!
   if( !ok() || !ex_base->ok() )
      return 1;

   ostrstream eout;

   // get the address of the base expression
   if( ex_base->evaluate() )
   {
      eout << ex_base->getError();
      ex_error = eout.str();
      ex_ok = FALSE;
      return 1;
   }
   value_t *bvalue = ex_base->getResultValue();

   // bvalue->vl_addr contains the address of the first element of the array

   // get the index
   if( ex_index->evaluate() )
   {
      eout << ex_index->getError() << ends;
      ex_error = eout.str();
      ex_ok = FALSE;
      return 1;
   }
   value_t *ivalue = ex_index->getResultValue();
   int index;
   switch( ex_index->getResultType() )
   {
      case( TY_UINT ):
      case( TY_LONG ):
      case( TY_ULONG):
      case( TY_INT ):
         index = ivalue->vl_int;
      break;

      case TY_USHORT:
      case TY_SHORT:
         index = (int)ivalue->vl_short;
      break;
    default:
      panic( "ExpArrayElement::evaluate():  bad index type" );
   }

   // make sure the index is within the dimensions of the array
   Type *btype = ex_base->getType();
   dim_t *dim = btype->getDimension();
   if( index < dim->di_low || index >= dim->di_high )
   {
      eout << "Array index " << ex_index->getExpression() << "==" << index << " is out of range.
      ex_error = eout.str();
      ex_ok = FALSE;
      return 1;
   }

   // now calculate the address of the element
   taddr_t daddr = ex_type->typeSize() * index + bvalue->vl_addr;

   // get the value of the element
```

```
   ex_addr = ex_type->getValue( ex_result_value, daddr );

   if( ex_addr == NULL )
   {
      eout << "Error reading " << ex_expr_str << " from inferior process." << ends;
      ex_error = eout.str();
      ex_ok = FALSE;
      return 1;
   }

   return 0;
}



/*------------------------------ ExpAmpersand ---------------------------*/

/*-------------------------------------------------------------------------
 |     METHOD: ExpAmpersand()
 |
 |    PURPOSE: Gets the address of an expression.
 -------------------------------------------------------------------------*/
ExpAmpersand::ExpAmpersand( Expression *exp ) : Expression()
{
   ostrstream eout;
   ex_base = exp;

   // if base if no good then bag it
   ex_ok = ex_base->ok();
   if( !ex_ok )
   {
      eout << ex_base->getError();
      ex_error = eout.str();
      return;
   }

   // ex_var     -> variable whose address is being returned
   // ex_type    -> type of the resulting expression
   // ex_addr    -> NULL 'cus you can't take the address of an & expression!
   ex_addr = NULL;
   ex_var = ex_base->ex_var;
   Type *t = ex_base->getType();
   ex_type = Type::makePointerType( t, QU_CONST );
   ex_result_type = ex_type->getTypeCode();
   ex_expr_str << "&" << ex_base->getExpression() << ends;
}

/*-------------------------------------------------------------------------
 |     METHOD: evaluate()
 |
 |    PURPOSE: Evaluate the expression object. If this method returns 0
 |             then one may call getResultValue() or getResultString()
 |             to get the result.  Otherwise, getError() will return
 |             an error message.
 |
 -------------------------------------------------------------------------*/
int ExpAmpersand::evaluate()
{

   // verify that all is ok!
   if( !ok() || !ex_base->ok() )
      return 1;

   ostrstream eout;

   // get the address of the base variable
   if( ex_base->evaluate() )
   {
      eout << ex_base->getError() << ends;
      ex_error = eout.str();
      ex_ok = FALSE;
```

```
      return 1;
   }

   // now get the address of the value
   ex_result_value.vl_addr = ex_base->getResultAddress();

   return 0;
}




/*------------------------------ ExpArrow ------------------------------*/

ExpArrow::ExpArrow( Expression *base, const char *field_name ) : Expression()
{
   ostrstream eout;

   ex_base = base;

   // if base if no good then bag it
   ex_ok = base->ok();
   if( !ex_ok )
   {
      eout << base->getError() << ends;
      ex_error = eout.str();
      return;
   }

   // get the base variable and make sure it's a pointer to a struct or union
   Type *ptype = base->getType();
   Type *btype = ptype->getBase();
   if( ptype->getTypeCode() != DT_PTR_TO ||
      (btype->getTypeCode() != TY_STRUCT && btype->getTypeCode() != TY_UNION) )
   {
      ex_ok = FALSE;
      eout << "Bad -> expression. " << base->getExpression() << " is not a structure or union."
      ex_error = eout.str();
      return;
   }

   // find the field variable in the bvar
   aggr_or_enum_def_t *ae = btype->getAggrOrEnumDef();
   Variable *fvar = NULL;
   for( fvar=ae->ae_u.aeu_aggr_members; fvar; fvar=fvar->getNext() )
      if( !strcmp( fvar->getName(), field_name ) )
         break;

   if( !fvar )
   {
      ex_ok = FALSE;
      eout << "Bad -> expression. " << field_name << " is not a member of structure, " << base->
      ex_error = eout.str();
      return;
   }

   // save this field variable
   ex_var  = fvar;
   ex_type = fvar->getType();
   ex_result_type = ex_var->getType()->getTypeCode();
   ex_expr_str << ex_base->getExpression() << "->" << ex_var->getName() << ends;
}

   ---------------------------------------------------------------------
   |     METHOD: evaluate()
   |
   |    PURPOSE: Evaluate the expression object. If this method returns 0
   |             then one may call getResultValue() or getResultString()
   |             to get the result.  Otherwise, getError() will return
   |             an error message.
```

```
 |
 ---------------------------------------------------------------------*/
int ExpArrow::evaluate()
{
   ostrstream eout;

   // verify that all is ok!
   if( !ok() || !ex_base->ok() )
      return 1;

   // get the address of the base variable
   if( ex_base->evaluate() )
   {
      eout << ex_base->getError() << ends;
      ex_error = eout.str();
      ex_ok = FALSE;
      return 1;
   }
   value_t *bvalue = ex_base->getResultValue();

   // bvalue->vl_addr contains the address of the structure or union (the base address)

   // get the value of the field
   ex_addr = ex_var->getValue( ex_stack_frame, ex_result_value, bvalue->vl_addr );
   if( ex_addr == NULL )
   {
      eout << "Error reading " << getExpression() << " from inferior process." << ends;
      ex_error = eout.str();
      ex_ok = FALSE;
      return 1;
   }

   return 0;
}



/*------------------------------ ExpStar --------------------------------*/

ExpStar::ExpStar( Expression *base ) : Expression()
{
   ex_base = base;
   ex_var  = NULL;
   ostrstream eout;

   ex_expr_str << "*" << ex_base->getExpression() << ends;

   // if base if no good then bag it
   ex_ok = base->ok();
   if( !ex_ok )
   {
      eout << base->getError();
      ex_error = eout.str();
      return;
   }

   // get the type and make sure it's a pointer
   Type *btype = base->getType();
   if( btype->getTypeCode() != DT_PTR_TO )
   {
      ex_ok = FALSE;
      eout << "Bad indirection expression. " << base->getExpression() << " is not a pointer." <<
      ex_error = eout.str();
      return;
   }

   // get the type for this expression
   ex_type = btype->getBase();
   ex_result_type = ex_type->getTypeCode();
}
```

```
/*-----------------------------------------------------------------------
 |     METHOD: evaluate()
 |
 |    PURPOSE: Evaluate the expression object. If this method returns 0
 |             then one may call getResultValue() or getResultString()
 |             to get the result.  Otherwise, getError() will return
 |             an error message.
 |
 -----------------------------------------------------------------------*/
int ExpStar::evaluate()
{
   ostrstream eout;

   // First you will get the value of the base expression.
   // The returned "lv.addr" will be the address of a variable of type btype->getBase()

   // verify that all is ok!
   if( !ok() || !ex_base->ok() )
      return 1;

   // get the address of the base variable
   if( ex_base->evaluate() )
   {
      eout << ex_base->getError() << ends;
      ex_error = eout.str();
      ex_ok = FALSE;
      return 1;
   }

   // now get the value of the base expression (which is the address of this expression's result
   value_t *bvalue = ex_base->getResultValue();

   // fetch the actual value using this address
   // the problem is that we have no variable...only the type

   if( !ex_type->getValue( ex_result_value, bvalue->vl_addr ) )
   {
      eout << "Error reading " << getExpression() << " from inferior process." << ends;
      ex_error = eout.str();
      ex_ok = FALSE;
      return 1;
   }

   return 0;
}




/*-------------------- BASE EXPRESSION ---------------------------------------*/

Variable *
Expression::findLocalVar( const char *vname )
{
   // construct the stack trace and find the current stack frame
   Frame *frame;
   Stack *s = Stack::getCurrentStack();
   if( !s )
      return NULL;

   // Start the search at the current stack frame.
   frame = ex_stack_frame = s->getCurrentFrame();

   // search up the stack for the var
```

```
   bool Found = FALSE;
   Variable *v=NULL;
   for( ; frame && !Found; frame=frame->fr_next_up )
   {
      Function *f = frame->fr_func;
      int lnum    = frame->fr_lnum;

      // if we don't have line info about this frame then move on
      if( !f || !lnum )
         continue;

      // find the lexical block of lnum so we know where to start looking
      class Block *blocks = f->getBlocks();
      Block *b=blocks;
      while( !Found )
      {
         int start_lnum = b->getStartLNumber();
         int end_lnum = b->getEndLNumber();

         if( lnum >= start_lnum &&  lnum <= end_lnum )
         {
            if( b->getSubBlocks() != NULL )
               b = b->getSubBlocks();
            else
               break;
         }
         else if( lnum > end_lnum )
            b = b->getNext();
         else // lnum < start_lnum
         {
            b = b->bl_parent;
            break;
         }

         if( b==NULL ) panic( "Botch in Bat while searching for block of current pc" );
      }

      // ok...we've got the block of the current frame's line number.
      // start in this block, and search out until we find the var OR run out of blocks

      for( v=NULL; b && !Found; b=b->bl_parent )
      {
         for( v = b->getVars(); v; v=v->getNext() )
         {
            if( !strcmp( v->getName(), vname ) )
            {
               Found = TRUE;
               ex_stack_frame = frame;        // scarf away the stack frame for later
               break;
            }
         }
      }
   }

   if( !Found )
      v = NULL;

   return v;
}


Variable *
Expression::findGlobalVar( const char *vname )
{
   SourceFile *sf;
   Variable   *v=NULL;
   SymTab::findFileOfGlobal( vname, &sf, &v );

   return v;
}
```

```
void
Expression::indent( ostrstream &rout )
{
   for(int i=0; i<ex_indent_level; i++ )
      rout << "     ";
}


/*-------------------------------------------------------------------------
 |     METHOD: getResultString()
 |
 |    PURPOSE: Adds the string representation of the result of this
 |             expression to the ostrstream parameter.
 -------------------------------------------------------------------------*/
void
Expression::getResultString( ostrstream &result_str )
{
   int rc=0;

   switch( ex_result_type )
   {
    case( DT_ARRAY_OF ):
       // ex_result_value.vl_addr is the address of the first element of the array
       // We need to fetch all the elements of the array

       Type *elemType = getType()->getBase();
       dim_t *dim = getType()->getDimension();
       bool IsString = ( elemType->getTypeCode()==TY_CHAR || elemType->getTypeCode()==TY_UCHAR )

       if( IsString ) result_str << "\"";
       else           result_str << "( ";


       for( int i=(int)dim->di_low; i<dim->di_high; i++ )
       {
          // create an array expression [i]
          Expression *cexp = new ExpConstant( i );
          Expression *array_exp = new ExpArrayElement( this, cexp );
          array_exp->evaluate();
          if( elemType->getTypeCode()==TY_CHAR || elemType->getTypeCode()==TY_UCHAR )
             if( array_exp->getResultValue()->vl_char=='\0' )
                break;

          array_exp->getResultString( result_str );

          if( i+1<dim->di_high )
             switch( elemType->getTypeCode() )
             {
              case( DT_ARRAY_OF ):
              case( TY_UNION ):
              case( TY_STRUCT ):
                 result_str << "," << ends;
              break;

              case TY_CHAR:
              case TY_UCHAR:
              break;

              default:
              result_str << ", ";
             }
       }
       if( IsString ) result_str << '"';
       else           result_str << " )";

       result_str << ends;
    break;

    case( DT_PTR_TO ):
       result_str << "0x" << hex << ex_result_value.vl_addr; // << ends;
    break;
```

```
case( TY_CHARACTER ):
   result_str << "TY_CHARACTER not done yet"; // << ends;
break;

case( TY_UINT ):
case( TY_LONG ):
case( TY_ULONG):
case( TY_INT ):
   result_str << ex_result_value.vl_int; // << ends;

break;

case TY_USHORT:
case TY_SHORT:
   result_str << ex_result_value.vl_short; // << ends;
break;

case TY_CHAR:
case TY_UCHAR:

   result_str << ex_result_value.vl_char; // << ends;
break;

case TY_BITFIELD:
   result_str << "TY_BITFIELD not done yet"; // << ends;
break;

case TY_UNION:
case TY_STRUCT:
case TY_U_STRUCT:
case TY_U_UNION:
   // ok...we need to do a little work here.
   // for each field (variable) that makes up this structure/union
   // we must create an expression.

   aggr_or_enum_def_t *ae = ex_type->getAggrOrEnumDef();
   if( ae->ae_u.aeu_aggr_members==NULL )
   {
      result_str << "Source information not available for this structure or union." << ends;
      return;
   }

   ex_indent_level++;
   result_str << endl;
   indent( result_str );
   result_str << "{" << ends;
   for( Variable *fvar=ae->ae_u.aeu_aggr_members; fvar; fvar=fvar->getNext() )
   {
      indent( result_str );
      result_str << fvar->getName() << " = ";
      Expression *field_exp = new ExpDot( this, fvar->getName() );
      field_exp->evaluate();
      field_exp->getResultString( result_str );
      result_str << ends;

      // memory leak here!!
      // delete field_exp;
   }
   indent( result_str );
   result_str << "}";   // << ends;
   ex_indent_level--;

break;

case TY_FLOAT:
   result_str << ex_result_value.vl_float; // << ends;
break;
case TY_DOUBLE:
   result_str << ex_result_value.vl_double; // << ends;
break;
case TY_ENUM:
case TY_U_ENUM:
```

```
      // ex_result_value.vl_int contains the value.
      // Convert this value to the appropriate enumeration string.
      enum_member_t *em = ex_type->getAggrOrEnumDef()->ae_u.aeu_enum_members;
      for( ; em != NULL; em = em->em_next )
      {
         if( ex_result_value.vl_int == em->em_val )
         {
            result_str << em->em_name;
            break;
         }
      }
      if( !em )
         result_str << "BAD ENUMERATION VALUE";
   break;

   case TY_REAL:
      result_str << ex_result_value.vl_float; // << ends;
   break;

   default:
      panic( "Expression::getResultString(): unknown type..." );
  }
  if( rc )
  {
     ex_ok = 0;
     ex_error = strsave( "Error in Expression::getResultString()." );
 }

  return;
}
```

```
/*-------------------------------------------------------------------
 |      CLASS: Function and FunctionTab
 |    PURPOSE: Objects representing functions supporting lookup of
 |             various information concerning functions.
 |    NOTES:
 -------------------------------------------------------------------*/

#ifndef FUNCTION_H_INCLUDED
#define FUNCTION_H_INCLUDED


// two kinds of function lists
//  Dynamic arrays

class Function;
typedef class Function *FuncPtr;
class_DynArray(FuncPtr,);
typedef DynArray(FuncPtr) FunctionArray;
typedef FunctionArray funcarray_t;

//  Linked lists of function object pointers

class_List(FuncPtr);
typedef List(FuncPtr) FunctionList;
typedef FunctionList funclist_t;



typedef class FSymInfo
{
 public:
   FSymInfo( int symno ) {fs_symno = symno;}

   int getSymNumber() { return fs_symno;}
   void setSymNumber( int n ) { fs_symno = n;}
   int getSymLimit() { return fs_symlim;}
   void setSymLimit( int n ) { fs_symlim = n;}

 protected:
   int             fs_symno;     // Start of syms in file for this func
   int             fs_symlim;    // ditto
} fsyminfo_t;


// line number

typedef struct lnost
{
   taddr_t getAddress() { return ln_addr; }
   int getLNumber()     { return ln_num; }
   struct lnost *getNext() { return ln_next; }

   void setAddress( taddr_t addr ) { ln_addr = addr; }
   void setLNumber( int lnum ) { ln_num = lnum; }
   void setNext( struct lnost *next ) { ln_next = next; }

   taddr_t                     ln_addr;         // text offset
   int                         ln_num;          // line number
   struct lnost                *ln_next;        // next line
} lno_t;



    Bits in fu_flags -- what information has been loaded

// No symbol table info
#define FU_NOSYM          0x1

// Have processed line number info
#define FU_DONE_LNOS      0x2
```

```
// Have processed block info
#define FU_DONE_BLOCKS    0x4

// Function doesn't set up frame pointer
#define FU_NO_FP          0x8

// Function is declared static (C interpreter)
#define FU_STATIC         0x10



/*-------------------------------------------------------------------------
  |      CLASS: Function
  |    PURPOSE: This object describes a function.  It contains a pointer
  |             to the source file in which it is defined, the address
  |             in the target of the function, and pointers to line
  |             numbers and local variable information.
  |
  |             Much of the information in this object is only loaded
  |             on demand.  It attempts to be quite lazy...
  -------------------------------------------------------------------------*/
class Function
{
 public:

   Function(const char *name,
            int symno,
            taddr_t addr,
            class SymTab *symtab,
            SourceFile *fil,
            Function *next );

   void getLNumberSymRange( int *p_start, int *p_lim );

   void fixRegisterParams();



   //  This function is an attempt to improve the accuracy of block start line
   //  numbers.
   int backupLNumToCurly(int orig_lnum );

   static int loadInfo( Function *f, taddr_t unused_addrlim, char *verbose, char *unused_arg2 );

   taddr_t minBreakpointAddress();



   // ---------------Line to address mappings-----------------------------

   //  Return line number info given a function and an address.
   //  Return NULL if no line number information is available.
   //
   //  This method searches for the first lnum that has an address larger
   //  than text_addr, and returns the one before that.
   //

   lno_t * addrToLno( taddr_t text_addr );


   //  Return the lnum that matches address text_addr.
   //
   int rbracAddrToLNum(taddr_t text_addr);

   int addrToLNumber( taddr_t text_addr );

   //  Map line number to the closest linenumber and address, given that the
   //  line is within the function.
   //
   //  Return 0 if there is no line number information, 1 if we can't find
   //  the line number.
   //
```

```
  taddr_t lNumberToClosestAddr( int lnum, int *closest_line );


  //  Map line number to an address, given that the line is within
  //  function f.
  //
  //  Return 0 if there is no line number information, 1 if we can't find
  //  the line number.
  //
  taddr_t lNumberToAddr( int lnum );


  // Set *p_addr to the address corresponding to line lnum in the function.
  // Return 0 for success, -1 and an error message otherwise.
  int mapLNumberToAddr( int lnum, taddr_t *p_addr );

  Preamble *getPreamble()
     { if( !fu_preamble ) fu_preamble = new Preamble(this);  return fu_preamble;}

  // Both of these routines load the appropriate info if necessary.
  // Otherwise, they just return it.
  class Block *getBlocks();
  lno_t *getLineNumbers();

  //  Comparison function for sorting by address.  The list may contain
  //  duplicate entries for a given function (this comes from the dbx
  //  type entry in the symbol table and the linker N_TEXT entry).
  //  Do the comparison so that entries with symbol table information
  //  appear before entries without).
  //
  static int addrCompare( void *p1, void *p2 );

  taddr_t getAddress() { return fu_addr;}
  void setAddress( taddr_t addr ) { fu_addr = addr;}
  const char *getName() {return fu_name;}
  void setName( const char *name ) { fu_name = strsave( name ); }
  class SymTab *getSymTab() {return fu_symtab_id;}
  void setSymTab( class SymTab *st ) { fu_symtab_id = st; }
  class Function *getNext() { return fu_next; }
  void setNext( class Function *next ) { fu_next = next; }
  bool hasFramePointer() { return getPreamble(), framePointerSetup(); }

  int getSymNumber()           { return fu_fsyminfo->getSymNumber();}
  void setSymNumber( int n )   { fu_fsyminfo->setSymNumber( n );}
  int getSymLimit()            { return fu_fsyminfo->getSymLimit();}
  void setSymLimit( int n )    { fu_fsyminfo->setSymLimit( n );}
  short getFlags( )            { return fu_flags;}
  void setFlags( short flags ) { fu_flags = flags; }
  FunctionList *getFunctionList() { return fu_funclist; }
  void setFunctionList( FunctionList *fl ) { fu_funclist = fl; }
  language_t getLanguage()     { return fu_language; }
  void setLanguage( language_t l ) { fu_language = l; }
  SourceFile *getSourceFile() { return fu_fil; }
  void setSourceFile( SourceFile *sf ) { fu_fil = sf; }
  int getMaxLNumber()
     {  if( !lineNumbersLoaded() )
           getLineNumbers();

        return fu_max_lnum;
     }

  // Booleans
  int symbolsLoaded()            { return !(fu_flags & FU_NOSYM); }
  int lineNumbersLoaded()        { return (fu_flags & FU_DONE_LNOS); }
  int blocksLoaded()             { return (fu_flags & FU_DONE_BLOCKS);}
  int framePointerSetup()        { return !(fu_flags & FU_NO_FP);}
  int staticFunction()           { return (fu_flags & FU_STATIC);}

protected:

  //  Always valid.
  short                   fu_flags;                  // flags, see below
```

```
    const char              *fu_name;                   // function name
    class Type              *fu_type;
    lexinfo_t               *fu_lexinfo;
    taddr_t                 fu_addr;                    // function text area address
    class SourceFile        *fu_fil;                    // source file
    language_t              fu_language;                // source language
    class SymTab            *fu_symtab_id;              // symbol table of this function
    FunctionList            *fu_funclist;               // pointer to function list this object
    class Function          *fu_next;                   // next function in this list
    FSymInfo                *fu_fsyminfo;               // file symtab stuff
    Preamble                *fu_preamble;               // used for recognizing functions preamb

    void *                  fu_statement_id;

    //  Valid if FU_DONE_LNOS set

    lno_t                   *fu__lnos;                  // list of line numbers, if any
    short                   fu_max_lnum;

    //  Valid if FU_DONE_BLOCKS set

    class Block             *fu__blocks;                // block list
};
typedef class Function func_t;

#define FU_PREAMBLE(f)  ((preamble_t *)(((f)->fu_preamble_id != NULL) \
                                ? ((f)->fu_preamble_id) :  get_startup_code(f)))

#define FU_SYMNO(f)             ((fsyminfo_t *)(f)->fu_fsyminfo_id)->fs_symno
#define FU_SYMLIM(f)            ((fsyminfo_t *)(f)->fu_fsyminfo_id)->fs_symlim
#define FU_CBLIST(f)            ((fsyminfo_t *)(f)->fu_fsyminfo_id)->fs_cblist
#define FU_COFF_LNO_START(f)    ((fsyminfo_t *)(f)->fu_fsyminfo_id)->fs_coff_lno_start
#define FU_COFF_LNO_LIM(f)      ((fsyminfo_t *)(f)->fu_fsyminfo_id)->fs_coff_lno_lim


typedef int (*iof_func_t)(Function *f, taddr_t addr, char *arg1, char *arg2);


/*-------------------------------------------------------------------------
 |      CLASS: FunctionTab
 |    PURPOSE: An array of Function object pointers which are lazily
 |             sorted by address.  Supports searching for functions
 |             by address.
 |
 -------------------------------------------------------------------------*/
class FunctionTab
{
 public:
    // Constructor for FunctionTab class
    //  Create a function table representing the list of functions in flist.
    //  Flist is unsorted (actually in symbol table order).
    //
    //  We just save away the parameters in the structure.  We don't sort
    //  the functions by address until the first addr_to_func() call.
    //
    FunctionTab(SymTab    *st,
                Function  *flist,
                int       flist_len,
                taddr_t   first_addr,
                taddr_t   last_addr );


    //  Adjust the segment base functab by delta.  This involves changing the
    //  function and line number addresses for all functions in the table.
    //  We also have to change the offsets of any static local variables.
    //
    //  This is called when a shared library's base address changes.
    //
    void adjustTextAddrBase( long delta );

    // Build a list of functions whose name matches the parameter, name.
    //
```

```
    int nameToFunctionList( const char *name, FunctionList *fl_list );

    //  Return the highest text address that is still within function f.
    //  Used for getting the last text line of a function.
    //
    taddr_t getAddrLim( func_t *f );

    int iterateOverFunction( iof_func_t func, char *arg1, char *arg2 );


    //  Look up the function by address and return the function with address
    //  greatest but <= addr.
    //
    Function *addrToFunc( taddr_t addr );

  protected:
    //  Create a table of functions sorted by address in functab->ft_tab.
    //  We eliminate duplicate function entries.
    //
    //  This called the first time we need to look up a function by address.
    //
    void sort();


    taddr_t             ft_first_addr;   // First valid address of segment
    taddr_t             ft_last_addr;    // Last ditto

    SymTab              *ft_symtab;      // Symtab this functab belongs to
    Function            *ft_flist;       // list of functions (unsorted)
    int                 ft_flist_len;    // Number of entries in list

    FunctionArray       *ft_tab;         // Sorted array of function pointers
                                         // may be < ft_flist_len (duplicates)

    edef FunctionTab functab_t;


#endif
```

```
/*-------------------------------------------------------------------
 |       FILE: function.C
 |    PURPOSE: Object class representing a function.
  -------------------------------------------------------------------*/

#include "symtab.h"

Function::Function(const char *name,
                   int symno,
                   taddr_t addr,
                   class SymTab * symtab,
                   SourceFile * fil,
                   Function * next )
{
   fu_flags = 0;
   fu_name = name;
   fu_type = NULL;
   fu_lexinfo = NULL;
   fu_addr = addr;
   fu_fil = fil;
   fu_language = (fil != NULL) ? fil->fi_language : LANG_UNKNOWN;
   fu_symtab_id = symtab;
   fu_next = next;
   fu_funclist = NULL;
   fu_statement_id = NULL;
   fu__blocks = NULL;
   fu__lnos = NULL;
   fu_fsyminfo = new FSymInfo( symno );
}

int
Function::loadInfo( Function *f, taddr_t unused_addrlim, char *verbose, char *unused_arg2 )
{
   unused_addrlim = (taddr_t)NULL;
   unused_arg2 = (char *)NULL;

   if( (bool) verbose)
      cout << " loading symbols of function " << f->fu_name << endl;
   (void) f->minBreakpointAddress();
   (void) f->getLineNumbers();
   (void) f->getBlocks();
   return 0;
}


//  We found that sometimes the compiler gives two symbol table entries
//  for a register parameter - a stack copy of the parameter and a
//  register copy.  This routine zaps the stack copy...makes life much
//  simpler!

void
Function::fixRegisterParams()
{
   //  We are only interested in the vars at the outermost scope.

   if (fu__blocks == NULL)
      return;

   Variable *head = fu__blocks->getVars();

   Variable *prev = NULL;
   Variable *next = NULL;
   for( Variable *vreg = head; vreg != NULL; vreg = next )
   {
      next = vreg->getNext();
      int zapped_vreg = FALSE;
      if( vreg->getClass() == CL_REG || vreg->getClass() == CL_REGPARM )
      {
         for( Variable *varg = head; varg != NULL; varg = varg->getNext() )
         {
            if( varg->getClass() == CL_ARG &&
                strcmp( vreg->getName(), varg->getName() ) == 0 )
```

```
            {
               vreg->setNext( varg->getNext() );

               // could be a problem here!!!!
               *varg = *vreg;
               if( prev != NULL )
                  prev->setNext( next );
               else
                  head = next;

               //  free vreg at this point
               delete vreg;

               zapped_vreg = TRUE;
               break;
            }
         }
      }
      if (!zapped_vreg)
         prev = vreg;
   }

   fu__blocks->setVars( head );
}


//  This function is an attempt to fix a problem we had with the accuracy
//  of block start line numbers.  It seems to work pretty well...better
//  than before.
//
//  The way we do this is to search back through the source lines, looking
//  for the '{' character that starts a block.  We only go back a small
//  number of lines - if we don't find anything by then we just return orig_lnum.
//
int
Function::backupLNumToCurly(int orig_lnum )
{
   const int lbrace = '{'; // to avoid upsetting vi's '}' matching

   if( fu_fil == NULL || fu_fil->open( FALSE ) != 0 )
      return orig_lnum;

   SFile *sf = fu_fil->getSFile();

   if( orig_lnum > sf->getNLines() )
      return orig_lnum;

   //  Make sure that we only go back a reasonable distance, and that we
   //  don't go out of range.

   int min_lnum = orig_lnum - 50;
   if( min_lnum < 1 )
      min_lnum = 1;
   if( orig_lnum < 1 )
      orig_lnum = 1;

   int lnum = orig_lnum;

   //  If the opening line doesn't have an lbrace, or has an
   //  lbrace with nothing but whitespace after it, back up
   //  a line.
   //
   //  Again, this code is not bulletproof (we'll get confused by
   //  if the lbrace is inside a comment, for example), but it doesn't
   //  matter much if it gets things wrong.

   const char *cptr = strrchr( sf->getLine( lnum - 1 ), lbrace );
   if( cptr != NULL )
   {
      int lastch;
      bool incomment;
```

```
      incomment = FALSE;
      lastch = *cptr++;
      for( ; ; ++cptr )
      {
         if( incomment && lastch == '*' && *cptr == '/' )
            incomment = FALSE;
         else if( !incomment && lastch == '/' && *cptr == '*' )
            incomment = TRUE;
         else
         {
            if( !incomment && isalnum(*cptr) )
               break;
            lastch = *cptr;
         }
         if (*cptr == '\0')
         {
            --lnum;
            break;
         }
      }
   }
   if (cptr == NULL || *cptr == '\0')
      --lnum;

   for( ; lnum >= min_lnum; --lnum )
      if( strchr(sf->getLine( lnum ), lbrace) != NULL )
         return lnum;
   return orig_lnum;
}



// load up the block information.
// Much of this code is based on gdb and lots of experimentation!

   ck_t *
Function::getBlocks()
{
   nlist_t nm, extra_nm;
   class_t aclass;
   type_t *type;
   const char *name, *s;
   var_t *v;
   bool skipthis;
   int lnum, next_lnum;

   if( fu_flags & FU_DONE_BLOCKS )
      return fu__blocks;

   //  We want to use typedef names if possible for structures and
   //  enums (see ci_basetype_name), so we need any type information
   //  in the file.  Thus we load the file types.

   fu_fil->getTypes( FALSE );

   stf_t *stf = fu_fil->getStf();

   Block *blocklists_tab[MAX_BLOCK_LEVEL + 1];
   Block **blocklists = blocklists_tab + 1;
   blocklists[-1] = fu_fil->getBlock();
   blocklists[0] = NULL;

   int level = 0;

   Variable *varlists[MAX_BLOCK_LEVEL];
   varlists[0] = NULL;

   aggr_or_enum_def_t *aelists[MAX_BLOCK_LEVEL];
   aelists[0] = NULL;

   typedef_t *tdlists[MAX_BLOCK_LEVEL];
   tdlists[0] = NULL;
```

```
int symlim = getSymLimit();
int extra_sym = 0;

SymTab *symtab = stf->getSymTab();
Executable *exec = symtab->getExecutable();
Block *bl;
for( int symno = getSymNumber(); symno < symlim + extra_sym; symno++ )
{
   if (symno == symlim)
      nm = extra_nm;
   else
      exec->getSym( symno, nm );

   switch(nm.n_type)
   {
    case N_LBRAC:
      if (level == MAX_BLOCK_LEVEL)
         panic("block nesting too deep");

      lnum = addrToLNumber( stf->getAddress() + (taddr_t)nm.n_value );

      bl = new Block( blocklists[level - 1] );
      bl->setStartLNumber( backupLNumToCurly( lnum ) );

      bl->setVars( varlists[level] );
      varlists[level] = NULL;
      bl->setTypeDef( tdlists[level] );
      tdlists[level] = NULL;
      bl->setAggrOrEnum( aelists[level] );
      aelists[level] = NULL;

      bl->setNext( blocklists[level] );
      blocklists[level] = bl;

      ++level;
      blocklists[level] = NULL;
      varlists[level] = NULL;
      aelists[level] = NULL;
      tdlists[level] = NULL;

      break;

    case N_RBRAC:
      if( level <= 0 )
         panic( "missing LBRAC" );

      //  The logically last RBRAC symbol of a function
      //  (the one that takes us from level 1 to level 0)
      //  sometimes appears earlier than we'd expect.
      //  We arrange that we always process this symbol
      //  last.

      if( level == 1 && symno < symlim - 1 )
      {
         extra_nm = nm;
         extra_sym = 1;
         break;
      }

      blocklists[level - 1]->setSubBlocks( blocklists[level] );
      --level;
      lnum = rbracAddrToLNum( stf->getAddress() + nm.n_value );
      blocklists[level]->setEndLNumber( lnum - 1 );

      //  Some heuristics to try and cope with bogus block
      //  start lines.  If a block starts on the same line
      //  as the following block at the same level, back up
      //  one then go back to the next previous curly.
      //

      bl = blocklists[level]->getSubBlocks();
```

```
      if( bl == NULL )
         break;

      next_lnum = bl->getStartLNumber() + 1;
      for( ; bl != NULL; bl = bl->getNext() )
      {
         if( next_lnum != 0 && bl->getStartLNumber() >= next_lnum )
            bl->setStartLNumber( backupLNumToCurly( next_lnum - 1) );

         next_lnum = bl->getStartLNumber();
      }
      if (blocklists[level]->getStartLNumber() >= next_lnum)
         blocklists[level]->setStartLNumber( backupLNumToCurly( next_lnum - 1) );

      break;
    case N_LSYM:
    case N_PSYM:
    case N_STSYM:
    case N_RSYM:
    case N_FUN:
    case N_LCSYM:
      s = exec->getSymString( symno );

      //  Skip any names that start with '#'.
      //  These are temporaries...I think.

      if (*s == '#')
         break;

      name = parseName( &s );
      scheck( &s, ':');
      type = Class( stf, &symno, &s, &aclass );

      if( aclass == CL_FUNC && symno == getSymNumber() &&
          strcmp( name, fu_name ) == 0 )
      {
         fu_type = type;
         break;
      }

      switch (aclass)
      {
       case CL_ARG:
         if( nm.n_type == N_RSYM )
            aclass = CL_REGPARM;
         skipthis = FALSE;
         break;
       case CL_AUTO:
         if (nm.n_type == N_RSYM)
            aclass = CL_REG;
         skipthis = FALSE;
         break;
       case CL_REG:
       case CL_REGPARM:
       case CL_LSTAT:
       case CL_REF:
         skipthis = FALSE;
         break;
       case CL_TYPEDEF:
         type->getTypeDef()->setNext( tdlists[level] );
         tdlists[level] = type->getTypeDef();
         skipthis = TRUE;
         break;
       case CL_TAGNAME:
         type->getAggrOrEnumDef()->setNext( aelists[level] );
         aelists[level] = type->getAggrOrEnumDef();
         skipthis = TRUE;
         break;
       default:
         skipthis = TRUE;
         break;
      }
```

```
    // discard duplicate variables with same name and address.

    if( level == 0 && aclass == CL_AUTO )
    {
       for( v = varlists[level]; v != NULL; v = v->getNext() )
       {
          if( v->getAddress() == nm.n_value &&
              (v->getClass() == CL_ARG ||
               v->getClass() == CL_REGPARM) &&
              strcmp(v->getName(), name) == 0 )
          {
             skipthis = TRUE;
             break;
          }
       }
    }

    if( level == 0 && aclass == CL_AUTO )
    {
       for( v = varlists[level]; v != NULL; v = v->getNext() )
       {
          if( v->getAddress() == nm.n_value &&
              (v->getClass() == CL_REG ||
               v->getClass() == CL_REGPARM) &&
              strcmp(v->getName(), name) == 0 )
          {
             skipthis = TRUE;
             break;
          }
       }
    }

    if (skipthis)
       break;

    v = new Variable( name, aclass, type, nm.n_value );
    v->setLanguage( stf->getLanguage() );
    if( v->getClass() == CL_LSTAT)
       v->setAddress( v->getAddress() + (int)symtab->getTextOffset() );
    v->setNext( varlists[level] );
    varlists[level] = v;

    //  Special case code for structures passed by value.
    //  The Sun C compiler copies these to the stack and
    //  passes a pointer to this copy to the called function.
    //  In the symbol table for the called function we get
    //  a normal symbol for the variable, followed by a
    //  register symbol for it.  We spot the situation by
    //  looking for register vars that are too big to go
    //  in a register.
    //
    //  We push an extra level of indirection on to the
    //  variable's type, and drop the symbol table entry before
    //  this one if it refers to a variable of the same name.

    if ((aclass == CL_REG || aclass == CL_REGPARM) &&
        (type->getTypeCode()!=TY_FLOAT && type->getTypeCode()!=TY_DOUBLE) &&
        type->typeSize() > sizeof(long))
    {
       type = new Type( DT_PTR_TO );
       type->setQualifiers( 0 );
       type->setBase( v->getType() );
       v->setType( type );
       v->setFlags( v->getFlags() | VA_HIDE_PTR );

       if( v->getNext() != NULL &&
          strcmp(name, v->getNext()->getName()) == 0)
          v->setNext( v->getNext()->getNext() );
    }
    break;
}
```

```
    }
    if (level != 0)
       panic("missing RBRAC");

    //  Functions with parameters but no local variables sometimes
    //  don't get an LBRAC symbol.  If this is one of those cases,
    //  then fix it by building a level 0 block for it.

    if( blocklists[0] == NULL && varlists[0] != NULL )
    {
       bl = new Block( fu_fil->getBlock() );
       bl->setStartLNumber( lineNumbersLoaded() ? fu__lnos->getLNumber() : 1 );
       bl->setEndLNumber( fu_max_lnum );
       bl->setVars(  varlists[0] );
       bl->setTypeDef( tdlists[0] );
       bl->setAggrOrEnum( aelists[0] );
       blocklists[0] = bl;
    }

    fu__blocks = blocklists[0];
    fu_flags |= FU_DONE_BLOCKS;

    //  If a function has parameters, no outer level locals but some
    //  inner locals, we can get a situation where the inner locals
    //  are noted at one level too low.  Fix this if necessary.

    if( fu__blocks != NULL && fu__blocks->getSubBlocks() == NULL &&
        fu__blocks->getNext() != NULL)
    {
       fu__blocks->setSubBlocks( fu__blocks->getNext() );
       fu__blocks->setNext( NULL );
    }

    fixRegisterParams();
    return fu__blocks;




FuncPtr listPrev( FunctionList *list, FuncPtr value )
{
   ListIter(FuncPtr) iterator;
   FuncPtr prev = NULL;

   iterator = *list;
   for( iterator = *list; iterator.iterMore(); iterator++ )
   {
      if( value == *iterator )
         break;

      prev = *iterator;
   }

   return prev;
}


void
Function::getLNumberSymRange( int *p_start, int *p_lim )
{
   if (fu_fil == NULL)
      panic("NULL fil in Function::getLNumberSymRange()");
   stf_t *stf = fu_fil->getStf();

   // If line numbers precede function
   if( stf->getLNumbersPrecedeFunctions() )
   {
      // get the previous node in this function's FunctionList

      Function *prev = listPrev( fu_funclist, this );
```

```
      // if 'this' is first node then
      if( prev != NULL )
         *p_start = prev->getSymNumber();
      else
         *p_start = stf->getSymNumber();
      *p_lim = getSymLimit();
   }
   else
   {
      *p_start = getSymNumber();
      *p_lim = getSymLimit();
   }
}



lno_t *
Function::getLineNumbers()
{
   lno_t *lno, *first;
   int symno, symno_start, symno_lim;

   // if already loaded, then just return them!
   if (fu_flags & FU_DONE_LNOS)
      return fu__lnos;

   Executable *exec = fu_symtab_id->getExecutable();


   //  First grab all the N_SLINE symbols in the function - we possibly
   //  discard some of them later.

   lno_t *last = NULL;
   getLNumberSymRange( &symno_start, &symno_lim );
   for( symno = symno_start; symno < symno_lim; ++symno )
   {
      nlist_t nm;

      exec->getSym( symno, nm );
      if(nm.n_type == N_SLINE )
      {
         lno = new lnost;
         lno->setLNumber( nm.n_desc );
         lno->setAddress( nm.n_value + fu_symtab_id->getTextOffset() );
         if( last != NULL )
            last->setNext( lno );
         else
            fu__lnos = lno;
         last = lno;
      }
   }
   if (last == NULL)
      fu__lnos = NULL;
   else
      last->setNext( NULL );

   //  FRAGILE CODE
   //
   //  Some compilers helpfully put out junk symbols at the start
   //  of functions.  We try to get rid of these.
   //
   switch(fu_language)
   {
    case LANG_C:
      lno = fu__lnos;

      //  Skip any symbols whose address is the same as the
      //  function address.
```

```
      for( ; lno != NULL && lno->getAddress() == fu_addr; lno = lno->getNext() )
         ;

      //  Skip all but the last of a set of lnos with the same
      //  address.

      if( lno != NULL )
      {
         taddr_t base_addr;
         lno_t *next;

         base_addr = lno->getAddress();
         for (;;)
         {
            next = lno->getNext();
            if( next == NULL || next->getAddress() != base_addr )
               break;
            lno = next;
         }
      }

      fu__lnos = lno;
      break;
    case LANG_UNKNOWN:
      break;
    default:
      panic("unknown language in gfl");
   }

   if (fu__lnos != NULL && last != NULL)
      fu_max_lnum = last->getLNumber();
   fu_flags |= FU_DONE_LNOS;
   return fu__lnos;
}
(


//  Return an line number information given a function and an address.
//  Return NULL if no line number information is available.
//
//  This function searches for the first lno that has an address larger
//  than text_addr, and returns the one before that.
//
lno_t *
Function::addrToLno( taddr_t text_addr )
{
   Preamble *pr = getPreamble();
   if (text_addr < fu_addr + pr->pr_bpt_offset)
      return NULL;
   lno_t *last = NULL;
   for (lno_t *ln = getLineNumbers(); ln != NULL; ln = ln->getNext())
   {
      if (ln->getAddress() > text_addr)
         return last;
      last = ln;
   }
   return last;
}

//  Return the lnum that matches address text_addr.
//
//  This function is used to matching RBRAC addresses to line numbers.
//  Some compilers emit several lnos for a for loop, all with the same
//  source line number.  For the end of a block we want a source line
//  number near the end of the loop, hence the odd way this routine works.

int
Function::rbracAddrToLNum(taddr_t text_addr)
{
   int             max_lnum;
   lno_t          *ln;
```

```
    Preamble *pr = getPreamble();

    if (text_addr < fu_addr + pr->pr_bpt_offset)
        return 0;

    max_lnum = 0;
    for (ln = getLineNumbers(); ln != NULL; ln = ln->getNext())
    {
        if (ln->getAddress() > text_addr)
            break;
        if (ln->getLNumber() > max_lnum)
            max_lnum = ln->getLNumber();
    }
    return max_lnum;
}



int
Function::addrToLNumber( taddr_t text_addr )
{
    lno_t *lno = addrToLno( text_addr );
    return (lno != NULL) ? lno->getLNumber() : 0;
}



//  Map a line number to an address, given that the line is within
//  the function.
//
//  Return 0 if there is no line number information, 1 if we can't find
//  the line number.
//
taddr_t
Function::lNumberToAddr( int lnum )
{
    register struct lnost *ln;

    if ((fu_flags & FU_NOSYM) || getLineNumbers() == NULL)
        return 0;
    for (ln = getLineNumbers(); ln != NULL; ln = ln->getNext())
    {
        if (ln->getLNumber() == lnum)
            return ln->getAddress();
    }
    return 1;
}


//  Map line number lnum to the closest linenumber and address, given that the
//  line is within the function.
//
//  Return 0 if there is no line number information, 1 if we can't find
//  the line number.
//
taddr_t
Function::lNumberToClosestAddr( int lnum, int *closest_line )
{
    register struct lnost *ln;

    if ((fu_flags & FU_NOSYM) || getLineNumbers() == NULL)
        return 0;
    lnost *prevln = NULL;
    lnost *ret_ln = NULL;
    for (ln = getLineNumbers(); ln != NULL; ln = ln->getNext())
    {
        // cout << "          line==" << ln->getLNumber() << "  addr==" << ln->getAddress() <<
        if (ln->getLNumber() == lnum)
        {
            ret_ln = ln;
            break;
        }
```

```
      else if( ln->getLNumber() > lnum )
      {
         ret_ln = (prevln) ? prevln : ln;
         break;
      }
      prevln = ln;
   }
   if( ret_ln == NULL ) return 1;

   *closest_line = ret_ln->getLNumber();

   return ret_ln->getAddress();
}



//  Determine the minimum displacement into the function code at which a
//  breakpoint can be placed. This should be after the registers have
//  been saved if they are saved. Return the address.
//
//  If the function has line number information, just return the address
//  of the first line.
//
taddr_t
Function::minBreakpointAddress()
{
   Preamble *pr = getPreamble();
   if (getLineNumbers() != NULL)
      return getLineNumbers()->getAddress();
   else
      return fu_addr + pr->pr_bpt_offset;
}


// Set *p_addr to the address corresponding to line lnum in the function.
// Return 0 for success, -1 and an error message otherwise.
int
Function::mapLNumberToAddr( int lnum, taddr_t *p_addr )
{
   taddr_t         addr;

   if ((addr = lNumberToAddr(lnum)) == 1)
   {
      cerr << "no executable code at line " << lnum << " of " << fu_fil->getName() << endl;
      return -1;
   }
   if (addr == 0)
   {
      cerr << "no line number information for " << fu_fil->getName() << endl;
      return -1;
   }

   taddr_t t = minBreakpointAddress();
   if (addr < t)
      addr = t;
   *p_addr = addr;
   return 0;
}
```

```
/*-------------------------------------------------------------------
 |       FILE: function_map.C
 |    PURPOSE: object class mapping names and addresses to Function
 |             objects.
  -------------------------------------------------------------------*/
#include "symtab.h"



//  Comparison function for sorting by address.  The list may contain
//  duplicate entries for a given function (this comes from the dbx
//  type entry in the symbol table and the linker N_TEXT entry).
//  Do the comparison so that entries with symbol table information
//  appear before entries without).
//
int
Function::addrCompare( const void *p1, const void *p2 )
{
   Function *f1 = (Function *)p1;
   Function *f2 = (Function *)p2;

   if( f1->fu_addr == f2->fu_addr )
      return( f1->fu_flags & FU_NOSYM ) - ( f2->fu_flags & FU_NOSYM );
   return( f1->fu_addr < f2->fu_addr ) ? -1 : 1;
}


static int
addrCompare2( const void *p1, const void *p2 )
{
   Function *f1 = *(Function **)p1;
   Function *f2 = *(Function **)p2;

   if( f1->getAddress() == f2->getAddress() )
      return( f1->getFlags() & FU_NOSYM ) - ( f2->getFlags() & FU_NOSYM );
   return( f1->getAddress() < f2->getAddress() ) ? -1 : 1;
}




// Constructor for FunctionTab class
//  Create a function table representing the list of functions in flist.
//  Flist is unsorted (actually in symbol table order).
//
//  We just save away the parameters in the structure.  We don't sort
//  the functions by address until the first addr_to_func() call.
//

FunctionTab::FunctionTab(SymTab    *st,         // symbol table
                         Function  *flist,      // linked list of Function objects
                         int       flist_len,   // how many Function objects in list
                         taddr_t   first_addr,  // first valid address of segment
                         taddr_t   last_addr )  // last    "      "     "    "
{
   Function     *f;

   // offset each function's address by first_addr
   for( f = flist; f != NULL; f = f->getNext() )
      f->setAddress( f->getAddress() + first_addr );


    / scarf away the information in the Function Table
   ft_first_addr = first_addr;
   ft_last_addr = last_addr;

   ft_symtab = st;
   ft_flist = flist;
   ft_flist_len = flist_len;
```

```
    ft_tab = NULL;
}


//  Create a table of functions sorted by address in functab->ft_tab.
//  We eliminate duplicate function entries.
//
//  This called the first time we need to look up a function by address.
//

void
FunctionTab::sort()
{
    int i;

    // firstly, create a new FunctionArray

    if( ft_tab!=NULL )
        ft_tab->dynTruncate();
    else
        ft_tab = new FunctionArray;

    // Now just add each element of the singly linked list to
    // the FunctionArray dynamic array.

    for( Function *f=ft_flist; f!=NULL; f=f->getNext() )
        ft_tab->dynAppend( f );

    // sort FunctionArray using quicker sort

    Function **base = ft_tab->dynData();
    f = (*ft_tab)[0];
    qsort( (char *)base, ft_tab->dynSize(), sizeof( Function * ), addrCompare2 );

    // remove duplicate values from the array

    taddr_t prevaddr = ~0;
    FunctionArray *nodupList = new FunctionArray;
    for( i=0; i<ft_tab->dynSize(); i++ )
    {
        f = (*ft_tab)[i];
        if( f->getAddress() != prevaddr )
        {
            nodupList->dynAppend( f );
        }
        prevaddr = f->getAddress();
    }
    delete ft_tab;
    ft_tab = nodupList;
}



//  Adjust the segment base of the function table by delta.
//  This involves changing the function and line number addresses
//  for all functions in the table.
//  We also have to change the offsets of any static local variables.
//
void
FunctionTab::adjustTextAddrBase( long delta )
{
    Function *f;
    lno_t *lno;

    for( f = ft_flist; f != NULL; f = f->getNext() )
    {
        f->setAddress( f->getAddress() + delta );

        if( f->lineNumbersLoaded() )
        {
            for( lno = f->getLineNumbers(); lno != NULL; lno->ln_next )
```

```
            lno->ln_addr += delta;
        }

        if( f->blocksLoaded() )
        {
            Block *b = f->getBlocks();
            b->iterateOverVars( adjustLStatAddr, (char *)delta );
        }
    }
    ft_first_addr += delta;
    ft_last_addr += delta;
}


// Build a list of functions whose name matches the parameter, name.
//
int
FunctionTab::nameToFunctionList( const char *name, FunctionList *fl_list )
{
    int len = strlen(name);

    // setup and iterator to the beginning of fl_list
    ListIter(FuncPtr) fl_iterator;
    fl_iterator = *fl_list;

    Function *f;
    int nmatches = 0;

    for( f=ft_flist; f!=NULL; f=f->getNext() )
    {
        // if this Function object matches name,
        // then add it to beginning of fl_list

        if( (strlen( f->getName() ) >= len) && (strncmp( f->getName(), name, len ) == 0) )
        {
            fl_list->listAppend( f );
            // fl_iterator.iterInsertBefore( f );
            nmatches++;
        }
    }

    // return the number of matches added to the fl_list
    return nmatches;
}



//  Return the highest text address that is still within function f.
//  Used for getting the last text line of a function.
//
taddr_t
FunctionTab::getAddrLim( func_t *f )
{
    taddr_t    addr;

    // sort the function table if necessary
    if( ft_tab == NULL )
        sort();

    // find f's position in the sorted array
    int i;
    for( i=0; i < ft_tab->dynSize(); i++ )
        if( (*ft_tab)[i]==f ) break;

    // return the next one in the array
    // or last valid address in segment if last function
    if( i>=ft_tab->dynSize() )
        addr = ft_last_addr;
    else
    {
        Function *nextf = (*ft_tab)[i+1];
        addr = nextf->getAddress();
```

```
    }

    return addr;
}



//  Call (*func)(f, addrlim(f), arg1, arg2) for all functions in
//  the function table of symtab_id.
//

int
FunctionTab::iterateOverFunction( iof_func_t func, char *arg1, char *arg2 )
{
    // sort the function table if necessary
    if( ft_tab == NULL )
        sort();

    // call func() for each function in sorted array
    int i;
    for( i=0; i < ft_tab->dynSize(); i++ )
    {
        Function *f = (*ft_tab)[i];

        int res = (*func)( f, f->getAddress(), arg1, arg2 );
        if( res != 0 )
            return res;
    }
    return 0;
}



//  Look up the function by address and return the function with address
//  greatest but <= addr.  Uses binary chop search.
//

Function *
FunctionTab::addrToFunc( taddr_t addr )
{
    //  Don't bother with the search if the address is out of range.
    //
    if( addr < ft_first_addr || addr >= ft_last_addr )
        return NULL;

    // sort if necessary
    if( ft_tab == NULL )
        sort();

    //  Use binary chop search to find function by address.

    Function **mid, **low, **max, **high;
    low = ft_tab->dynData();
    high = max = low + ft_tab->dynSize()-1;
    while( low <= high )
    {
        mid = low + (high - low) / 2;
        taddr_t maddr = (*mid)->getAddress();
        if( addr >= maddr && (mid == max || addr < mid[1]->getAddress()) )
            return (mid == (ft_tab->dynData() + ft_tab->dynSize()-1)) ? NULL : *mid;
        else if( maddr > addr )
            high = mid - 1;
        else
            low = mid + 1;
    }
    return NULL;
}
```

```
#ifndef FALSE
#define FALSE 0
#endif

   ndef TRUE
#define TRUE 1
#endif

typedef unsigned long taddr_t;
typedef int bool;

#define UNSET 0

const char *save_fname( const char *name, int ext );

#define UNKNOWN_SIZE      (-1)

#define BAD_ADDR       ((taddr_t)0x80000000)
```

```
#ifndef LANGUAGE_H_INCLUDED
#define LANGUAGE_H_INCLUDED

   Languages
   edef enum languageen
{
   LANG_C,                 // C
   LANG_FORTRAN,           // FORTRAN
   LANG_UNKNOWN            // Unknown language
} language_t;

#endif
```

```
// mreg.h - machine dependent register information

#ifndef MREG_H_INCLUDED
#define MREG_H_INCLUDED

#include "gendefs.h"

#define N_SUN_GREGS             32

typedef struct sunregsst
{
   struct regs sr_regs;
   struct rwindow sr_rwindow;
   struct fpu sr_fpu;
   bool sr_need_fpu;
} sunregs_t;

#endif
```

```
// preamble.h - used for recognizing function preamble code

#ifndef PREAMBLE_H_FILE
#define PREAMBLE_H_FILE

class Function *f;

class Preamble
{
 public:
   Preamble( Function *f );

   unsigned pr_bpt_offset;      // offset from func addr of first bpt loc
   unsigned pr_rsave_mask;      // register save mask
   int pr_rsave_offset;         // offset from fp of start of reg save area
};
typedef class Preamble preamble_t;

#endif
```

```
/*-----------------------------------------------------------------
 |       FILE: proc.h
 |    PURPOSE: Inferior process class.  For manipulation and
 |             examination of the debuggee process.
  -----------------------------------------------------------------*/
#ifndef PROC_H_INCLUDED
#define PROC_H_INCLUDED

#include <sys/types.h>
#include <sys/file.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <sys/param.h>
#include <signal.h>
#include <sys/user.h>
#include <sun4/reg.h>
#include <errno.h>
#include "mreg.h"
#include "utils.h"
#include "gendefs.h"
#include "sfile.h"
#include "/pro/forest/basis/src/list.H"
#include "/pro/forest/basis/src/dynarray.H"
#include "preamble.h"
#include <errno.h>
#include <iostream.h>


typedef struct sigst {
        const char *sig_name;
        int         sig_flags;
} sig_t;

// Firstly, we either catch or ignore a signal.

// If we catch a signal, then we either terminate the process or not.
// Firstly, a signal either terminates (and possibly core dumps) or not.
// Secondly, we either ignore a signal or we catch it.

#define SIG_IGNORE              0x1    /* continue automatically after sig */
#define SIG_CATCH               0x2    /* stop target after sig            */
#define SIG_TERMINATES          0x4    /* catching signal terminates process */
#define SIG_CORE                0x8    /* catching signal causes core dump */

//  Type for a target opcode.  Used in textSwap().
//
typedef unsigned opcode_t;

//  Which way to continue a process for proc_cont().
//
typedef enum cont_typeen {
        PR_STEP,        /* single step target */
        PR_CONT         /* continue target until signal or exit */
} cont_type_t;

//  What state signal handling is in for a given signal.
//
typedef enum sigstateen {
        SGH_DEFAULT,    /* SIG_DFL - default */
        SGH_IGNORED,    /* SIG_IGN - signal is ignored */
        SGH_CAUGHT      /* signal is caught */
} sigstate_t;

//  Why the process stopped after a Continue() or Start().

   edef enum stopresen {
        SR_SIG,         /* got a signal */
        SR_BPT,         /* hit a breakpoint */
        SR_DIED,        /* process exited */
        SR_SSTEP,       /* stopped after a single step */
        SR_USER,        /* user requested stops */
        SR_FAILED       /* couldn't restart target (breakpoint problems) */
```

```
} stopres_t;

enum which_regset {
        SUN_GEN_REGS,
        SUN_FP_REGS,
        SUN_FPA_REGS
};

typedef union {
        float f;
        double d;
} fpval_t;

//  Register type for readRegister().  Positive values are general
//  purpose registers.
//
#define REG_PC                  (-1)    /* program counter */
#define REG_SP                  (-2)    /* stack pointer */
#define REG_AP                  (-3)    /* argument pointer (VAX only) */
#define REG_FP                  (-4)    /* frame pointer */
#define REG_CONDITION_CODES     (-5)    /* condition codes */
#define REG_FP_CONDITION_CODES  (-6)    /* floating point condition codes */

typedef enum ptracereq ptracereq_t;
typedef void *code_id_t;


class Process
{
 public:

   // Constructor
   //  Make a new proc object for a running process pid.
   //  textfile is the name of the binary file.
   //  attached is TRUE if we attached to this process with a PTRACE_ATTACH.
   //
   Process( const char *textfile, int pid, int attached );

   // Destructor
   //
   ~Process();

   // creates a new process object and forks off an inferior process
   //
   static Process *start(const char *path, const char **argv,
                         const char **envp, long rdlist,
                         taddr_t stop_addr, const char **p_errmesg,
                         stopres_t *p_whystopped );

   // Creates a new process object and attaches to process, pid.
   //
   static Process *attach( const char *path, int pid, stopres_t *p_whystopped );

   //  Detach from a process which we earlier attached to.
   //  Leave the process running.
   //
   void detach( int sig );

   //  Kill off the target process
   //
   void terminate();

   taddr_t getRegisterValue( int reg );
   int     setRegisterValue( int regno, taddr_t val );

   //  Restart process proc.  Either continue or single step it, depending on ctype.
   //
   //  If the process was stopped at a breakpoint, we remove the breakpoint,
   //  single step the process and replace the breakpoint.  If ctype was PR_CONT
   //  and the single step stopped normally we then continue the process with
   //  a PR_CONT.
   //
```

```
//  The intended effect is that breakpoints that are at the current pc value
//  are ignored.
//
//  We return the reason why the process stopped - one of SR_SIG, SR_DIED,
//  SR_SSTEP, SR_BPT and SR_FAILED.  We map SIGTRAP into SR_BPT
//
stopres_t Continue( int sig, cont_type_t ctype );

//  Wait for the process to stop.  If the process didn't die, update the
//  stored register values.   Set ip_stopres to the reason why the
//  process stopped, and ip_restart_pc to the address the process
//  should be restarted from.
//
void waitForTarget();

int readFloatingPointValue( taddr_t addr, bool is_double, fpval_t *p_val );
int readFloatingPointRegister( int regno, bool is_double, fpval_t  *p_val );

//  Get the value currently stored in register regno of process proc.
//  Return 0 and set *p_val for success, -1 for failure.
//
//  regno is either a general purpose register, or one of REG_PC, REG_SP,
//  REG_FP and REG_AP.  See mreg.h.
//
int readRegister( int regno, taddr_t *p_val );

//  Interface to readRegister() that aborts on failure.
//
taddr_t getRegister( int regno );

int setRegister( int regno, taddr_t value );

//  Call a function in the target.
//
//  There must be no breakpoints installed when this routine is called
// NOT IMPLEMENTED YET!!!
//
int callFunction( code_id_t       code_id,
                  taddr_t         addr,
                  taddr_t        *args,
                  int            *argsizes,
                  int             nargs,
                  taddr_t        *p_res,
                  const char    **p_mesg );


//  Insert opcode into the text area at addr.  If p_old_opcode is
//  non NULL we ignore the supplied opcode and insert a breakpoint
//  opcode.  Otherwise insert opcode and set *p_old_opcode to the
//  old opcode.
//
//  We treat it as a fatal error if we are installing a breakpoint
//  and one is already there, or if we are installing other than
//  a breakpoint and there isn't one already there.
//
int textSwap( taddr_t addr, opcode_t opcode, opcode_t *p_old_opcode );

int writeCoreFile( const char *name );

//  Return the current state of signal handling for signal sig in process
//  Returns SGH_CAUGHT, SGH_IGNORED or SGH_DEFAULT.
//
sigstate_t getSignalState( int sig );

//  Read nbytes bytes into buf starting at address addr in the text area of process
//  proc.
//  The byte count is returned or -1 case of error.
//
int readText( taddr_t addr, char *buf, int nbytes );

//  Read nbytes of data into buf from process proc, starting at target
//  address addr.
```

```
    //  Return the number of bytes read, or -1 if there was an error.
    //  We never return a short count - the return value is always nbytes or -1.
    //
    int readData( taddr_t addr, char *buf, int nbytes );

    int writeData( taddr_t addr, const char *buf, int nbytes );

    int getRestartSignal();


    //-------------- Access methods ---------------------------------

    static Process *getCurrentProcess()     { return ip_current_proc; }

    int getPID()                            { return ip_pid; }
    void setPID( int pid )                  { ip_pid = pid; }

    // Return the filename of the binary of process proc.
    //
    char *getTextFile()                     { return ip_textfile;}
    void setTextFile( char * file )         { ip_textfile = file; }

    taddr_t getBaseSP()                     { return ip_base_sp;}
    void setBaseSP( taddr_t sp )            { ip_base_sp = sp; }
    taddr_t getRestartSP()                  { return ip_restart_pc;}
    void setRestartSP( taddr_t sp )         { ip_restart_pc = sp; }
    static short getLastSignal()            { return ip_lastsig;}
    void setLastSignal( short sig )         { ip_lastsig = sig; }

    //  Return TRUE if we attached to process proc with a PTRACE_ATTACH.
    //
    short getAttached()                     { return ip_attached;}
    void setAttached( short attached )      { ip_attached=attached; }

    // Return the reason why the process stopped.  This is the reason for the
    // most recent stop.
    //
    static stopres_t getWhyStopped()        { return ip_whystopped; }
    void setWhyStopped( stopres_t reason ) { ip_whystopped = reason; }
    static int getSignalFlags( int sig ) { return ip_Sigtab[sig].sig_flags; }
    static void setSignalFlags( int sig, int flags ) { ip_Sigtab[sig].sig_flags = flags; }

    static const char *getSignalString( int sig ) { return ip_Sigtab[sig].sig_name; }

  protected:

    // -------------- Protected methods ---------------------------
    static int ptrace( ptracereq_t req, int pid, char *addr, int data );
    int updateRegisterValues();
    void stopTarget();
    int waitWithInterrupt( union wait *p_status );
    char *getRestartPC();
    void singleStep( int sig );

    // -------------- Instance variables ------------------------
    int              ip_pid;
    char            *ip_textfile;
    taddr_t          ip_base_sp;
    taddr_t          ip_restart_pc;
    static short     ip_lastsig;
    short            ip_attached;
    static stopres_t ip_whystopped;

    sunregs_t        ip_sunregs;
    static Process  *ip_current_proc;
    static sig_t Process::ip_Sigtab[32];
};

typedef class Process iproc_t;
```

```
/*-----------------------------------------------------------------
 |       FILE: proc.C
 |    PURPOSE: Inferior process class.  For manipulation and
 |             examination of the debuggee process.
 -----------------------------------------------------------------*/

#include "proc.h"
#include <setjmp.h>
#include "breakpoint.h"
#include "as.h"
#include <sys/ptrace.h>

// Firstly, we either catch or ignore a signal.
//
// If we catch a signal, then we either terminate the process or not.
// Firstly, a signal either terminates (and possibly core dumps) or not.
// Secondly, we either ignore a signal or we catch it.

sig_t Process::ip_Sigtab[] = {
        "",                 SIG_IGNORE,
        "SIGHUP    (1) ", SIG_IGNORE,
        "SIGINT    (2) ", SIG_IGNORE,
        "SIGQUIT   (3) ", SIG_CATCH|SIG_TERMINATES|SIG_CORE,
        "SIGILL    (4) ", SIG_CATCH|SIG_TERMINATES|SIG_CORE,
        "SIGTRAP   (5) ", SIG_CATCH|SIG_TERMINATES|SIG_CORE,
        "SIGIOT    (6) ", SIG_CATCH|SIG_TERMINATES|SIG_CORE,
        "SIGEMT    (7) ", SIG_CATCH|SIG_TERMINATES|SIG_CORE,
        "SIGFPE    (8) ", SIG_CATCH|SIG_TERMINATES|SIG_CORE,
        "SIGKILL   (9) ", SIG_CATCH|SIG_TERMINATES,
        "SIGBUS    (10)", SIG_CATCH|SIG_TERMINATES|SIG_CORE,
        "SIGSEGV   (11)", SIG_CATCH|SIG_TERMINATES|SIG_CORE,
        "SIGSYS    (12)", SIG_CATCH|SIG_TERMINATES|SIG_CORE,
        "SIGPIPE   (13)", SIG_IGNORE,
        "SIGALRM   (14)", SIG_IGNORE,
        "SIGTERM   (15)", SIG_IGNORE,
        "SIGURG    (16)", SIG_IGNORE,
        "SIGSTOP   (17)", SIG_CATCH|SIG_TERMINATES|SIG_CORE,
        "SIGTSTP   (18)", SIG_CATCH|SIG_TERMINATES|SIG_CORE,
        "SIGCONT   (19)", SIG_IGNORE,
        "SIGCHLD   (20)", SIG_IGNORE,
        "SIGTTIN   (21)", SIG_CATCH|SIG_TERMINATES|SIG_CORE,
        "SIGTTOU   (22)", SIG_CATCH|SIG_TERMINATES|SIG_CORE,
        "SIGIO     (23)", SIG_IGNORE,
        "SIGXCPU   (24)", SIG_IGNORE,
        "SIGXFSZ   (25)", SIG_IGNORE,
        "SIGVTALARM(26)", SIG_IGNORE,
        "SIGPROF   (27)", SIG_IGNORE,
        "SIGWINCH  (28)", SIG_IGNORE,
        "SIGLOST   (29)", SIG_IGNORE,
        "SIGUSR1   (30)", SIG_IGNORE,
        "SIGUSR2   (31)", SIG_IGNORE,
};


typedef enum ptracereq ptracereq_t;

//  Return the offset into the u area of member.
//  Used when doing PEEKUSER ptrace requests.
//
#define U_OFFSET(member)        ((char *)&((struct user *)NULL)->member)

#define u_tsize         u_procp->p_tsize
#define u_dsize         u_procp->p_dsize
#define u_ssize         u_procp->p_ssize


//  Macros for getting/setting registers in a Sun regs structure.
//
#define IS_FLOAT_REG(regno)     ((regno) >= 32 && (regno) < 64)
#define FLOAT_REG_OFFSET        32
#define FRAME_REG(sr)           (sr.sr_rwindow.rw_fp)
#define INTEGER_REG(sr, regno)  ((&sr.sr_regs.r_y)[regno])
```

```
//  Breakpoint opcodes
//
#define BPT                                 0x91d02001      /* What gdb uses */
#define BPT_PC_OFFSET                       0       /* WRONG */
#define PUT_OPCODE_IN_WORD(w, data)         (data)
#define GET_OPCODE_FROM_WORD(w)             (w)

//  Special value for ptrace restart requests meaning restart the target
//  from where it stopped.
//
#define RESTART_FROM_WHERE_STOPPED      ((taddr_t)1)

//
//  Call ptrace(2), but abort if errno gets set.
//
int
Process::ptrace( ptracereq_t req, int pid, char *addr, int data )
{
   errno = 0;
   int res = ::ptrace( req, pid, addr, data, (char *)NULL );
   if (errno != 0)
      panic("ptrace failed in Process::ptrace");
   return res;
}


//
//  Return the current value of Sun register regno.
//  regno is a machine independent register number.
//
taddr_t
Process::getRegisterValue( int reg )
{
   switch (reg)
   {
   case REG_PC:
      return ip_sunregs.sr_regs.r_pc;
   case REG_SP:
      return ip_sunregs.sr_regs.r_sp;
   case REG_FP:
   case REG_AP:
      return FRAME_REG(ip_sunregs);

   case REG_CONDITION_CODES:
      return ip_sunregs.sr_regs.r_psr;
   default:
      if (reg == REG_FP_CONDITION_CODES || (reg >= 32 && reg < 64))
      {
         if (ip_sunregs.sr_need_fpu)
         {
            ptrace(PTRACE_GETFPREGS, ip_pid, (char *) &ip_sunregs.sr_fpu, 0);
            ip_sunregs.sr_need_fpu = FALSE;
         }
         if (reg == REG_FP_CONDITION_CODES)
            return ip_sunregs.sr_fpu.fpu_fsr;
         return ip_sunregs.sr_fpu.fpu_regs[reg - FLOAT_REG_OFFSET];
      }

      if (reg < 0 || reg >= N_SUN_GREGS)
         panic("bad reg in Process:getRegisterValue()");

      if (reg == 0)
         return 0;

      return INTEGER_REG(ip_sunregs, reg);
   }
}

int
Process::setRegisterValue( int regno, taddr_t val )
```

```
{
   switch (regno)
   {
   case REG_PC:
      ip_sunregs.sr_regs.r_pc = val;
      break;
   case REG_FP:
   case REG_AP:
      FRAME_REG(ip_sunregs) = val;
      break;
   case REG_SP:
      ip_sunregs.sr_regs.r_sp = val;
      break;
   default:
      if( IS_FLOAT_REG(regno) )
      {
         if( ip_sunregs.sr_need_fpu )
         {
            ptrace(PTRACE_GETFPREGS, ip_pid, (char *) &ip_sunregs.sr_fpu, 0);
            ip_sunregs.sr_need_fpu = FALSE;
         }
         ip_sunregs.sr_fpu.fpu_regs[regno] = val;
         break;
      }

      if (regno < 0 || regno >= N_SUN_GREGS)
         panic("bad regno in Process::setRegisterValue()");
      INTEGER_REG(ip_sunregs, regno) = val;
      break;
   }
   errno = 0;
   (void)ptrace(PTRACE_SETREGS, ip_pid, (char *)&ip_sunregs, 0);
   return (errno == 0) ? 0 : -1;
}


//  Update the stored machine register values.  This is called after the
//  target has been run and has thus changed the register values.
//
//  The pc is a special case.
//
//  Return TRUE if the target stopped because it hit a breakpoint.
//
int
Process::updateRegisterValues()
{
   ptrace(PTRACE_GETREGS, ip_pid, (char *) &ip_sunregs.sr_regs, 0);

   if( ::ptrace(PTRACE_READDATA, ip_pid, (char *) ip_sunregs.sr_regs.r_sp,
            sizeof(ip_sunregs.sr_rwindow), (char *)&ip_sunregs.sr_rwindow) != 0 )
      panic("rwindow read botch in Process::updateRegisterValues");

   taddr_t pc = (taddr_t) ip_sunregs.sr_regs.r_pc;
   Breakpoint *bp = Breakpoint::getBreakpointAtAddress( this, pc - BPT_PC_OFFSET );
   if( bp != NULL )
   {
      pc -= BPT_PC_OFFSET;
      ip_sunregs.sr_regs.r_pc = pc;
   }

   ip_sunregs.sr_need_fpu = TRUE;

   ip_restart_pc = pc;
   return bp != NULL;
}



jmp_buf Longjmp_env;

void
```

```
Process::stopTarget()
{
   longjmp(Longjmp_env, 1);
}

int
Process::waitWithInterrupt( union wait *p_status )
{
   if( setjmp(Longjmp_env) == 1 )
      return 0;

   return wait( p_status );
}

//  Wait for the process to stop.  If the process didn't die, update the
//  stored register values.   Set ip_stopres to the reason why the
//  process stopped, and ip_restart_pc to the address the process
//  should be restarted from.
//
void
Process::waitForTarget()
{
   static int      want_errors = -1;
   stopres_t       whystopped;
   int             pid,
                   at_bpt;
   bool            user_stopped_target;
   union wait      status;

   if( want_errors == -1 )
      want_errors = getenv("WANT_ERRORS") != NULL;

   user_stopped_target = FALSE;
   for (;;)
   {
      pid = wait(&status);

      if( pid == 0 )
         continue;

      if( pid == ip_pid )
         break;
      else if( pid == -1 )
      {
         if( errno != EINTR )
            cout << "wait returned -1" << endl;
      }
      else
      {
         if( want_errors )
            cout << "wait returned bad pid "
               << pid << " (expected " << ip_pid << ")" <<endl;
      }
   }

   if( WIFSTOPPED(status) )
   {
      ip_lastsig = 0;
      at_bpt = updateRegisterValues();
      if( status.w_stopsig != SIGTRAP )
      {
         whystopped = SR_SIG;
         ip_lastsig = status.w_stopsig;
      }
      else if( user_stopped_target )
         whystopped = SR_USER;
      else if( at_bpt )
         whystopped = SR_BPT;
      else
         whystopped = SR_SSTEP;
   }
   else
```

```
    {
       ip_lastsig = -1;
       ip_restart_pc = 0;
       whystopped = SR_DIED;
       Breakpoint::markAsUninstalled( this );
    }
    ip_whystopped = whystopped;
}

char *
Process::getRestartPC()
{
    if (ip_whystopped != SR_BPT)
       return (char *) 1;

    return (char *) ip_restart_pc;
}

// Constructor
//  Make a new proc object for a running process pid.
//  textfile is the name of the binary file.
//  attached is TRUE if we attached to this process with a PTRACE_ATTACH.
//
Process::Process( const char *textfile, int pid, int attached )
{
    ip_pid = pid;
    ip_textfile = strsave(textfile);
    ip_attached = attached;
    ip_current_proc = this;
}

// creates a new process object and forks off an inferior process
//
Process *
Process::start( const char *path, const char **argv, const char *envp[],
          long rdlist, taddr_t stop_addr, const char **p_errmesg, stopres_t *p_whystopped )
{
    static char     errbuf[128];

    int             pid;

    fflush(stdout);
    if( (pid = vfork()) == 0 )
    {

       if( ::ptrace(PTRACE_TRACEME, 0, (char *)NULL, 0) != 0 )
          panic("ptrace TRACEME request failed in child");

       execve(path, argv, envp);
       cerr << "!!!!!!!!!!!!!Child can't exec " << path << endl;
       cout << "!!!!!!!!!!!!!Child can't exec " << path << endl;
       cout << "Errno == " << errno << endl;
       _exit(1);
    }

    if (pid == -1)
    {
       cerr << "Can't fork to run " << path << endl;
       *p_errmesg = strsave( "Can't fork." );
       *p_whystopped = SR_DIED;
       return 0;
    }
    Process *ip = new Process( path, pid, FALSE );

    ip->waitForTarget();

    if( ip->ip_whystopped == SR_DIED )
    {
       cerr << "Can't start " << path << endl;
       *p_errmesg = strsave( "Can't start process." );
       *p_whystopped = SR_DIED;
       delete ip;
```

```
      return 0;
   }
   if( ip->ip_whystopped == SR_SSTEP )
   {
      Breakpoint *bp = new Breakpoint( stop_addr );
      if( bp->install( ip ) != 0 )
      {
         cerr << "Can't install initial breakpoint in " << path << endl;

         delete bp;
         if( !ip->getAttached() )
            ip->terminate();
         *p_errmesg = strsave( "Can't install initial breakpoint." );
         *p_whystopped = SR_DIED;
         delete ip;
         return 0;
      }
      (void) ip->Continue( 0, PR_CONT );
      delete bp;
   }
   *p_whystopped = ip->ip_whystopped;
   return ip;
}


// Creates a new process object and attaches to process, pid.
//
Process *
Process::attach( const char *path, int pid, stopres_t *p_whystopped )
{
   if( ptrace(PTRACE_ATTACH, pid, (char *)NULL, 0) != 0 )
      return 0;
   Process *ip = new Process( path, pid, TRUE );
   ip->waitForTarget();
   *p_whystopped = ip->ip_whystopped;
   return ip;
}


//  Kill off the target process
//
void
Process::terminate()
{
   union wait      status;

   ptrace(PTRACE_KILL, ip_pid, (char *)NULL, 0);
   if( ip_attached )
   {
      // fake the status
      status.w_T.w_Termsig = SIGKILL;
      status.w_T.w_Retcode = 0;
   } else
      waitForTarget();
}

// Destructor
//
Process::~Process()
{
   ip_current_proc = NULL;
}

//  Detach from a process which we earlier attached to.
//  Leave the process running.
//
void
Process::detach( int sig )
{
   if( !ip_attached )
      panic("proc_detach called but not attached to proc");
   if( Breakpoint::uninstallAllBreakpoints(this) != 0 )
```

```
      panic("can't uninstall breakpoints in proc");

   (void)ptrace(PTRACE_DETACH, ip_pid, getRestartPC(), sig);


void
Process::singleStep( int sig )
{
   //
   // Ask the assembler where it thinks the pc is going.
   //
   taddr_t npc = get_next_pc(this, ip_restart_pc);

   Breakpoint *bp = Breakpoint::addrToBreakpoint(npc);
   bool must_remove_bp = bp == NULL;
   if( bp == NULL )
      bp = new Breakpoint(npc);

   bool must_uninstall_bp = !bp->breakpointIsInstalled();
   if( must_uninstall_bp && bp->install( this ) != 0 )
      panic("Can't install bp in Process::singleStep()t");

   ptrace( PTRACE_CONT, ip_pid, getRestartPC(), sig );
   waitForTarget();

   if( ip_whystopped != SR_DIED )
   {
      if( must_uninstall_bp && bp->uninstall() != 0 )
         panic("Can't uninstall bp in sst");
      if( must_remove_bp )
         delete bp;
      if( ip_restart_pc == npc && ip_whystopped == SR_BPT )
         ip_whystopped = SR_SSTEP;
   }



//  Restart process proc.  Either continue or single step it, depending on ctype.
//
//  If the process was stopped at a breakpoint, we remove the breakpoint,
//  single step the process and replace the breakpoint.  If ctype was PR_CONT
//  and the single step stopped normally we then continue the process with
//  a PR_CONT.
//
//  The intended effect is that breakpoints that are at the current pc value
//  are ignored.
//
//  We return the reason why the process stopped - one of SR_SIG, SR_DIED,
//  SR_SSTEP, SR_BPT and SR_FAILED.  We map SIGTRAP into SR_BPT
//
stopres_t
Process::Continue( int sig, cont_type_t ctype )
{
   ptracereq_t     req;
   Breakpoint     *bp;

   //
   // Step over bpt if there is one at ip_restart_pc.
   //
   if( (bp = Breakpoint::getBreakpointAtAddress( this, ip_restart_pc )) != NULL )
   {
      if( bp->uninstall() != 0 )
         return SR_FAILED;

      singleStep( sig );

      if(bp->install(this) != 0)
         panic("can't install breakpoint in proc_cont");
      if(ip_whystopped != SR_SSTEP || ctype == PR_STEP)
         return ip_whystopped;
      if(ctype != PR_CONT)
         panic("bad ctype in proc_cont");
```

```
      req = PTRACE_CONT;
      sig = 0;
   }
   else
   {
      switch (ctype)
      {
      case PR_CONT:
         req = PTRACE_CONT;
         break;
      case PR_STEP:
         req = PTRACE_SINGLESTEP;
         break;
      default:
         panic("bad ctype in proc_cont");
         req = (ptracereq_t) UNSET;     /* to satisfy gcc */
      }
   }

   if (req == PTRACE_SINGLESTEP)
      singleStep( sig );
   else
   {
      ptrace(req, ip_pid, getRestartPC(), sig);
      waitForTarget();
   }

   return ip_whystopped;
}

int
Process::readFloatingPointValue( taddr_t addr, bool is_double, fpval_t *p_val )
{
   if(is_double)
      return readData(addr, (char *) &p_val->d, sizeof(double));
   else
      return readData(addr, (char *) &p_val->f, sizeof(float));
}

int
Process::readFloatingPointRegister( int regno, bool is_double, fpval_t  *p_val )
{
   if(is_double)
   {
      taddr_t        *buf = (taddr_t *) & p_val->d;

      if(readRegister(regno, buf) != 0 ||
         readRegister(regno + 1, buf + 1) != 0)
         return -1;
   }
   else
   {
      if(readRegister(regno, (taddr_t *) & p_val->f) != 0)
         return -1;
   }
   return 0;
}

//  Get the value currently stored in register regno of process proc.
//  Return 0 and set *p_val for success, -1 for failure.
//
//  regno is either a general purpose register, or one of REG_PC, REG_SP,
//  REG_FP and REG_AP.  See mreg.h.
//
int
Process::readRegister( int regno, taddr_t *p_val )
{
   *p_val = getRegisterValue(regno);
   return 0;
}
```

```
//  Interface to readRegister() that aborts on failure.
//
taddr_t
Process::getRegister( int regno )
{
   taddr_t         val;

   if(readRegister(regno, &val) != 0)
      panic("proc_readreg failed");
   return val;
}

int
Process::setRegister( int regno, taddr_t value )
{
   return setRegisterValue(regno, value);
}

#define N_REG_ARGS      6
#define RETURN_REGNO    8
#define ALIGN_STACK(n)  ((n) & ~07)


//  Call a function in the target.
//
//  There must be no breakpoints installed when this routine is called
// NOT IMPLEMENTED YET!!!
//
int
Process::callFunction( code_id_t       code_id,
                       taddr_t         addr,
                       taddr_t        *args,
                       int            *argsizes,
                       int             nargs,
                       taddr_t        *p_res,
                       const char    **p_mesg )
{
   //
   // NOT IMPLEMENTED YET!
   //
   return 0;
}



//  Insert opcode into the text area at addr.  If p_old_opcode is
//  non NULL we ignore the supplied opcode and insert a breakpoint
//  opcode.  Otherwise insert opcode and set *p_old_opcode to the
//  old opcode.
//
//  We treat it as a fatal error if we are installing a breakpoint
//  and one is already there, or if we are installing other than
//  a breakpoint and there isn't one already there.
//
int
Process::textSwap( taddr_t addr, opcode_t opcode, opcode_t *p_old_opcode )
{
   if(p_old_opcode != NULL)
      opcode = BPT;

   errno = 0;
   int temp = ptrace(PTRACE_PEEKTEXT, ip_pid, (char *) addr, 0);

   opcode_t old_opcode = GET_OPCODE_FROM_WORD(temp);
   temp = PUT_OPCODE_IN_WORD(temp, opcode);

   (void)ptrace(PTRACE_POKETEXT, ip_pid, (char *) addr, temp);

   if(errno != 0)
      return -1;
```

```
   if((opcode == BPT) == (old_opcode == BPT))
      panic("duplicate or vanished breakpoint in Process::textSwap()");
   if(p_old_opcode != NULL)
      *p_old_opcode = old_opcode;
   return 0;
}

int
Process::writeCoreFile( const char *name )
{
   if(ptrace(PTRACE_DUMPCORE, ip_pid, (char *)name, 0) == 0)
      return 0;

   cerr << "Can't dump core file to " << name << endl;

   return -1;
}

//  Return the current state of signal handling for signal sig in process
//  proc.  Returns SGH_CAUGHT, SGH_IGNORED or SGH_DEFAULT.
//
//
sigstate_t
Process::getSignalState( int sig )
{
   if(sig < 1 || sig >= NSIG)
      panic("sig botch in Process::getSignalState()");

   caddr_t uaddr = U_OFFSET(u_signal[sig]);

   errno = 0;
   taddr_t handler = ptrace(PTRACE_PEEKUSER, ip_pid, (char *) uaddr, 0);

   // if the ptrace above failed we try again, subtracting KERNSTACK bytes to
   // compensate for the missing kernel stack member.
   if(errno != 0)
   {
      errno = 0;
      handler = ptrace(PTRACE_PEEKUSER, ip_pid, uaddr - KERNSTACK, 0);
   }

   if(errno != 0)
      panic("sig handler botch in Process::getSignalState()");

   sigstate_t res;
   if(handler == (taddr_t) SIG_IGN)
      res = SGH_IGNORED;
   else if(handler == (taddr_t) SIG_DFL)
      res = SGH_DEFAULT;
   else
      res = SGH_CAUGHT;

   return res;
}


//  Read nbytes bytes into buf starting at address addr in the text area of
//  the process.
//  The byte count is returned or -1 case of error.
//
int
Process::readText( taddr_t addr, char *buf, int nbytes )
{
   if(::ptrace(PTRACE_READTEXT, ip_pid, (char *) addr, nbytes, buf) == 0)
      return 0;
   else
      return -1;
}

//  Read nbytes of data into buf from the process, starting at target
//  address addr.
//  Return the number of bytes read, or -1 if there was an error.
```

```
//  We never return a short count - the return value is always nbytes or -1.
//
int
P ocess::readData( taddr_t addr, char *buf, int nbytes )
{
   if(::ptrace(PTRACE_READDATA, ip_pid, (char *) addr, nbytes, buf) == 0)
      return 0;
   else
      return -1;
}

int
Process::writeData( taddr_t addr, const char *buf, int nbytes )
{
   if(::ptrace(PTRACE_WRITEDATA, ip_pid, (char *) addr, nbytes, (char *) buf) == 0)
      return 0;
   else
      return -1;
}


int
Process::getRestartSignal()
{
   int sig = getLastSignal();

   return sig ;
}
```

```
/*-------------------------------------------------------------------
 |       FILE: sfile.C
 |    PURPOSE: File access with caching.
 |
 |   CONTENTS: -methods-          ------------purpose-----------------
 |             lookUpFile         looks up sfile or constructs new one
 |             SFile              constructor...use lookupFile instead
 |             ~SFile             destructor
 |             addToSourcePath    add path to paths to search for files
 |             getName()          get file name of the sfile object
 |             getNLines()        return number of lines in file
 |             getLine()          return the nth line in the file
 |             operator[]         overloaded to call getline()
 |
 |    NOTES:
 -------------------------------------------------------------------*/
#ifndef SO_H_INCLUDED
#define SO_H_INCLUDED

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/file.h>
#include <sys/types.h>
#include <sys/stat.h>

#include "/pro/forest/basis/src/list.H"

#include <string.h>

#include <sysent.h>
#include <memory.h>

#include "gendefs.h"
   clude "strcache.h"

//
// some defines
//
#ifndef O_RDONLY
#define O_RDONLY        0
#endif
#ifndef L_SET
#define L_SET           0
#endif

// Annotation structure
//  associates annotations with a source file line
//
typedef struct
{
   int an_annot_mask;
   int an_line_number;
} so_annotation_t;

typedef so_annotation_t *so_annotptr_t;
class_List(so_annotptr_t);
typedef List(so_annotptr_t) so_AnnotList_t;



typedef struct {
        long si_mtime;
        long si_size;
} so_info_t;

class SFile;

//
//  A type for the line callback function passed to open().
//
typedef int (*so_line_callback_t)( long nlines, long pos );
```

```
typedef int (*so_input_func_t)(char *arg, off_t offset, char *buf, int nbytes);
typedef void (*so_close_func_t)(char *arg);
typedef int (*so_info_func_t)(char *arg, so_info_t *si);



//
// LBlock Object Class
//
#define LINES_PER_BLOCK         128
class LBlock
{
 protected:
   off_t bl_offset;
   LBlock *bl_next;

 public:

   // Destructor - frees all sucessive LBlock objects
   //              in the linked list
   LBlock()                      { ;                                 }
   ~LBlock()                     { if( bl_next ) delete bl_next;     }

   void setOffset(off_t offset) { bl_offset = offset;              }
   int  getOffset()             { return bl_offset;                }
   void setNext( LBlock *bl )    { bl_next = bl;                     }
   LBlock *getNext()             { return bl_next;                   }

   off_t bl_lines[LINES_PER_BLOCK];

};


//  Maximum number of simultaneously open file descriptors.
#define MAXFDS  10

class MFDobj : public FDobj
{
 public:

   MFDobj( const char *n, int fd );
   ~MFDobj();

   int getInfo( so_info_t *si );
   int getInput( off_t offset, char *buf, int nbytes );
   void getSlot();
   void freeSlot();
   int getFD() { return fd_fd; }

 protected:
   //
   // Static variables shared between all MFDobj objects
   //
   //  Array for keeping track of least recently used file descriptors.
   static class MFDobj *mf_Mfdtab[MAXFDS];

   // Use_time keeps track of the last used time of a file descriptor.
   // Each time we use a file descriptor, we set mf_last_used to
   // this variable and increment it.
   static long mf_Use_time;

   int fd_fd;
   char *mf_name;
   int mf_last_used;
};

typedef class MFDobj mfd_t;

//  Maximum number of paths
#define MAXPATHS        20
```

```
//
// SFile Object Class
//
class SFile;
typedef SFile *SFilePtr;

// linked list of SFile object pointers...
class_List(SFilePtr);


class SFile
{
 public:
   //
   // CONSTRUCTORS
   //
   SFile( const char *name, so_line_callback_t line_callback );

   //
   // DESTRUCTOR
   //
   ~SFile();

   //
   // Use this routine instead of constructor.
   // It looks thru list of currently opened files and
   // returns the appropriate sfile object.  If the file
   // has not been opened, then it constructs one for you.
   //
   static SFile *lookUpFile( const char *name, so_line_callback_t line_callback );


   //
   // Public Methods
   //
   static void addToSourcePath( const char *path );
   long getSize()               {return (long)so_size;  }
   const char *getName()        {return so_name;        }
   int getMaxLineLen()          {return so_max_linelen; }

   int getNLines()              {return so_nlines;      }
   void readMore( int reread );
   int hasChanged( int& p_reread );
   long modTime();

   // return the lnum'th line in the file.  1 based!!
   char *getLine( int lnum );
   char *operator[]( int lnum ) { return getLine( lnum ); }

   //
   // annotations
   //
   int getAnnotations( int line );
   void setAnnotations( int line, int mask );


 protected:

   void openViaFunc(const char *name,
                    so_line_callback_t line_callback );
   void readLine( off_t offset );
   void getLineOffsets();
   int lastLineChanged();


   static int so_Default_tabwidth;
   static char *so_Paths[MAXPATHS];
   static int so_Max_pathlen;
   static int so_Npaths;

   static List(SFilePtr) *so_SFile_list;
```

```
    //
    // Private variables
    //
    char *so_name;
    so_line_callback_t so_line_callback;
    so_info_func_t so_get_info;
    char *so_arg;
    mfd_t *so_mf;

    class StrCache *so_strcache;

    time_t so_mtime;
    off_t so_size;
    long so_nlines;
    long so_max_linelen;
    LBlock *so_block;
    char *so_lastline;
    long so_lastline_size;
    off_t so_lastline_offset;
    char *so_peekbuf;
    int so_peekbuf_size;
    int so_tabwidth;
    so_AnnotList_t *so_annot_list;
};
#endif /* !SO_H_INCLUDED */
```

```
/*-----------------------------------------------------------------
 |       FILE: sfile.C
 |    PURPOSE: File access with caching.
 |
 |    CONTENTS: -methods-         ------------purpose----------------
 |              lookUpFile        looks up sfile or constructs new one
 |              SFile             constructor...use lookupFile instead
 |              ~SFile            destructor
 |              addToSourcePath   add path to paths to search for files
 |              getName()         get file name of the sfile object
 |              getNLines()       return number of lines in file
 |              getLine()         return the nth line in the file
 |              operator[]        overloaded to call getline()
 |
 |    NOTES:
 -----------------------------------------------------------------*/
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/file.h>
#include <sys/types.h>
#include <sys/stat.h>

#include <sysent.h>
#include <memory.h>
#include <malloc.h>

#include "strcache.h"
#include "sfile.h"
#include "utils.h"

// SFile object static initializations
long MFDobj::mf_Use_time = 0;
int SFile::so_Default_tabwidth = 8;
    SFile::so_Max_pathlen = 0;
    SFile::so_Npaths = 0;
List(SFilePtr)* SFile::so_SFile_list = new List(SFilePtr);


/*-----------------------------------------------------------------------
 |      CLASS: lookUpFile
 |    PURPOSE: Searches list of currently opened files for "name."  If
 |             found, then it simply returns a pointer to this sfile
 |             object.
 |
 |        TBD: Fix the callbacks.  If callback is !NULL then the line
 |             offsets should be recomputed and the callback called.
 |
 -----------------------------------------------------------------------*/
SFile *SFile::lookUpFile( const char *name, so_line_callback_t line_callback )
{
   ListIter(SFilePtr) iterator;
   SFilePtr s;

   // just compare the last component of the filenames
   for( const char *np = name + strlen(name); np!=name && *np!='/'; np-- );
   if( *np=='/' ) np++;

   for( iterator = *so_SFile_list; iterator.iterMore(); iterator++ )
   {
      s = *iterator;

      for( const char *sp = s->getName() + strlen(s->getName());
          sp!=s->getName() && *sp!='/'; sp-- );
      if( *sp=='/' ) sp++;

      if( !strcmp( sp, np ) )
      {
        return s;
      }
   }
```

```
    //
    // not found...so construct a new object
    //
    s = new SFile( name, line_callback );

    return s;
}


SFile::SFile(const char *name, so_line_callback_t line_callback)
{
    char *nbuf;
    int fd=-1;
    int i;

    if (*name == '/' || so_Npaths == 0)
    {
        if ((fd = open(name, O_RDONLY)) == -1)
            return;
        nbuf = strsave(name);
    }
    else
    {
        nbuf = new char[so_Max_pathlen + 1 + strlen(name) + 1];

        for (i = 0; i < so_Npaths; i++) {
            (void) sprintf(nbuf, "%s/%s", so_Paths[i], name);
            if ((fd = open(nbuf, O_RDONLY)) != -1)
                break;
        }
        if (i == so_Npaths)
        {
            delete [] nbuf;
            return;
        }
    }

    so_mf = new class MFDobj( nbuf, fd );
    so_mf->getSlot();

    so_line_callback = line_callback;
    openViaFunc(nbuf, so_line_callback);

    // finally...add the newly constructed object to the linked list
    so_SFile_list->listAppend(this);

    // construct an empty annotation list
    so_annot_list = new so_AnnotList_t;
    free( nbuf );

    return;
}

// Annotations
//
int
SFile::getAnnotations( int line )
{
    if( !so_annot_list )
        return 0;

    ListIter(so_annotptr_t) iterator;
    iterator = *so_annot_list;
    for( ; iterator.iterMore(); iterator++ )
    {
        if( (*iterator)->an_line_number==line )
            return (*iterator)->an_annot_mask;
    }
    return 0;
}
```

```
void
SFile::setAnnotations( int line, int mask )
{
   so_annotptr_t annot = NULL;

   if( !so_annot_list )
      so_annot_list = new so_AnnotList_t;

   ListIter(so_annotptr_t) iterator;
   iterator = *so_annot_list;
   for( ; iterator.iterMore(); iterator++ )
   {
      if( (*iterator)->an_line_number==line )
         annot = *iterator;
   }
   if( !annot )
   {
      annot = new so_annotation_t;
      annot->an_line_number = line;
      so_annot_list->listAppend( annot );
   }

   annot->an_annot_mask = mask;
}




// add path to the path list for searching for files
// The path must not have a '/' at the end of it!
void
SFile::addToSourcePath( const char *path )
{
   if (so_Npaths >= MAXPATHS)
      panic("Too many source paths");

   so_Paths[so_Npaths++] = strsave(path);
   if (strlen(path) > so_Max_pathlen)
      so_Max_pathlen = strlen(path);
}



// 0-based!!!!
void
SFile::readLine(off_t offset)
{
   int oldindex;
   long ilen;

   const char *iptr = so_strcache->getString( offset, (int)'\n', ilen );
   if (iptr == NULL)
      iptr = "";
   const char *ilim = iptr + ilen;

   char *optr = so_peekbuf;
   char *obuf = so_peekbuf;
   char *olim = obuf + so_peekbuf_size - 1;
   int tabspaces = 0;
   int ch;

   for (;;)
   {
      if (tabspaces > 0)
      {
         ch = ' ';
         --tabspaces;
      }
      else
      {
         if (iptr == ilim)
```

```
            break;
         ch = *iptr++;
         if (ch == '\t')
         {
            tabspaces = (so_tabwidth - 1) -
               (optr - obuf) % so_tabwidth;
            ch = ' ';
         }
      }
      if (optr == olim)
      {
         so_peekbuf_size *= 2;
         oldindex = optr - obuf;
         obuf = so_peekbuf = realloc(obuf, (size_t)so_peekbuf_size);
         optr = obuf + oldindex;
         olim = obuf + so_peekbuf_size - 1;
      }
      if (ch == '\b' && optr > obuf)
         --optr;
      else if (ch == '\0')
         *optr++ = 128;
      else
         *optr++ = ch;
   }
   *optr++ = '\0';
}

void
SFile::openViaFunc(const char *name,
                   so_line_callback_t line_callback )
{
   so_name = strsave(name);
   so_line_callback = line_callback;
   so_nlines = 0;
   so_max_linelen = 0;
   so_block = NULL;
   so_strcache = new StrCache( so_mf );
   so_peekbuf_size = 128;
   so_peekbuf = malloc(so_peekbuf_size);
   so_lastline_size = 128;
   so_lastline = malloc((unsigned int)so_lastline_size + 1);
   so_tabwidth = so_Default_tabwidth;

   getLineOffsets();
}

//
// Destructor
//
SFile::~SFile()
{
   delete so_block;
   delete so_strcache;
   free(so_name);
   free(so_peekbuf);
   free(so_lastline);
}

void
SFile::getLineOffsets()
{

   long len;

   so_info_t sibuf;


   off_t *lptr, *llim;
   off_t pos;
   LBlock *bl;

   so_nlines = 0;
```

```
LBlock *last_block = so_block;
if (last_block == NULL) {
   lptr = llim = NULL;
   pos = 0;
}
else
{
   for (;;)
   {
      bl = last_block->getNext();
      if (bl == NULL)
         break;
      so_nlines += LINES_PER_BLOCK;
      last_block = bl;
   }
   lptr = last_block->bl_lines;
   llim = last_block->bl_lines + LINES_PER_BLOCK;
   pos = last_block->getOffset();

   // tell string cache that file has grown
   so_strcache->fileHasGrown();
}


const char *line;

while((line = so_strcache->getString(pos, (int)'\n', len)) != NULL)
{
   int specials_offset = 0;
   const char *lim = line + len;
   for (const char *cptr = line; cptr < lim; ++cptr)
   {
      if (*cptr == '\b')
      {
         if (cptr - line + specials_offset > 1)
            specials_offset -= 2;
      }
      else if (*cptr == '\t')
      {
         int cpos;

         cpos = cptr - line + specials_offset;
         specials_offset += (so_tabwidth - 1) - cpos % so_tabwidth;
      }
   }
   if (len + specials_offset > so_max_linelen)
      so_max_linelen = len + specials_offset;

   if (lptr == llim)
   {
      bl = new LBlock;
      bl->setOffset( pos );
      bl->setNext( (LBlock *)NULL );

      if (so_block == NULL)
         so_block = bl;
      else
         last_block->setNext( bl );
      last_block = bl;

      lptr = bl->bl_lines;
      llim = bl->bl_lines + LINES_PER_BLOCK;
   }

   *lptr++ = pos;
   pos += len + ((*cptr != '\0') ? 1 : 0);

   if (so_line_callback != NULL && (*so_line_callback)( so_nlines, pos ))
      break;
   ++so_nlines;
}
```

```
   so_size = pos;
   if (so_mf != NULL && so_mf->getInfo(&sibuf) == 0)
      so_mtime = sibuf.si_mtime;
   else
      so_mtime = 0;

   if (so_nlines != 0)
   {
      const char *cptr;
      long len;

      so_lastline_offset = lptr[-1];
      cptr = so_strcache->getString(so_lastline_offset, '\n', len);

      if (len >= so_lastline_size)
      {
         free(so_lastline);

         do
         {
            so_lastline_size *= 2;
         } while (len >= so_lastline_size);

         so_lastline = malloc((unsigned int)so_lastline_size + 1);
      }

      if (cptr == NULL)
         so_lastline[0] = '\0';
      else
      {
         memcpy(so_lastline, cptr, (unsigned int)len);
         so_lastline[len] = '\0';
      }
   }
}


int
SFile::lastLineChanged()
{
   const char *line;
   long junk_len;

   if (so_nlines == 0)
      return FALSE;
   if (so_lastline == NULL)
      return TRUE;
   line = so_strcache->getString( so_lastline_offset, '\n', junk_len );
   return strncmp(so_lastline, line, strlen(so_lastline)) != 0;
}

void
SFile::readMore(int reread)
{
   if (reread)
   {
      delete so_block;
      so_block = NULL;
      so_nlines = 0;
      so_strcache->forgetBuffers();
   }
   getLineOffsets();
}

int
SFile::hasChanged(int& p_reread)
{
   so_info_t sibuf;

   if (so_mf == NULL || so_mf->getInfo(&sibuf) != 0)
      return FALSE;
```

```
    if (sibuf.si_mtime == so_mtime && sibuf.si_size == so_size)
        return FALSE;

    so_reread = sibuf.si_size <= so_size || lastLineChanged();
    so_size = sibuf.si_size;
    return TRUE;
}



long
SFile::modTime()
{
    so_info_t sibuf;

    if (so_get_info == NULL || so_mf->getInfo(&sibuf) != 0 )
        return 0;
    return sibuf.si_mtime;
}

static char *empty="";



/*-----------------------------------------------------------------------
 |      METHOD: getLine
 |
 |     PURPOSE: Returns the lnum'th line in the file.  Note that this is
 |              1 based!
 -----------------------------------------------------------------------*/
char *
SFile::getLine(int lnum)
{
    if (lnum <= 0 || lnum > so_nlines)
    {
        printf( "SFILE:: line %d out of range, maxlines==%d\n",
                lnum, so_nlines );
        return empty;
    }

    lnum--;

    LBlock *bl = so_block;
    for (int nblocks = lnum / LINES_PER_BLOCK; nblocks > 0; --nblocks)
        bl = bl->getNext();

    readLine(bl->bl_lines[lnum % LINES_PER_BLOCK]);
    return so_peekbuf;
}

//
// -------------------- MFD object class methods -------------------------
//

MFDobj::MFDobj( const char *n, int fd ) : FDobj( fd )
{
    mf_name = strsave( n );
    fd_fd = fd;
    mf_last_used = (int)mf_Use_time++;
}

MFDobj::~MFDobj()
{
    if (fd_fd != -1)
    {
        close(fd_fd);
        freeSlot();
    }
    free( mf_name );
}

int
```

```
MFDobj::getInfo(so_info_t *si)
{
   struct stat stbuf;

   if (fstat(fd_fd, &stbuf) != 0)
      return FALSE;
   si->si_mtime = stbuf.st_mtime;
   si->si_size = stbuf.st_size;
   return 0;
}

int
MFDobj::getInput(off_t offset, char *buf, int nbytes )
{
   if (fd_fd == -1)
   {
      getSlot();
      if ((fd_fd = open(mf_name, O_RDONLY)) == -1)
         panic("can't reopen %s");
   }
   int fd = (int)fd_fd;
   mf_last_used = (int)mf_Use_time++;

   if (lseek(fd, offset, L_SET) == -1)
      return -1;
   return read(fd, buf, nbytes);
}


void
MFDobj::freeSlot()
{
   for (int i = 0; i < MAXFDS; i++)
   {
      if (mf_Mfdtab[i] == this)
      {
         mf_Mfdtab[i] = NULL;
         return;
      }
   }
   panic("so not in Mfdtab in free_fd_slot");
}

void
MFDobj::getSlot()
{
   int oldest = 0;
   long oldest_time = mf_Use_time;
   for (int i = 0; i < MAXFDS; i++)
   {
      if (mf_Mfdtab[i] == NULL)
      {
         mf_Mfdtab[i] = this;
         return;
      }
      if (mf_Mfdtab[i]->mf_last_used < oldest_time)
      {
         oldest = i;
         oldest_time = mf_Mfdtab[i]->mf_last_used;
      }
   }

   //  Fd table full, so must close least recently used fd.
   //
   close(mf_Mfdtab[oldest]->fd_fd);
   mf_Mfdtab[oldest]->fd_fd = -1;
   mf_Mfdtab[oldest] = this;
}
```

```
/*-----------------------------------------------------------------
 |       FILE: sharedlib.C
 |    PURPOSE: Methods for the SunOS sharelib object class.
  -----------------------------------------------------------------*/
   clude <sys/types.h>
#include <sys/wait.h>
#include <sys/file.h>
#include <a.out.h>
#include <link.h>
#include "symtab.h"
#include "data.h"

static const char **add_to_env(const char *s);

//  Dig out the list of loaded shared libraries and run time linked
//  global addresses.  This basically involves walking down linked
//  lists in the target via dread().  This must be called after the
//  target has started and the mapping been done.

int
SymTab::getSharedLibsAndGlobalAddresses( taddr_t addr_dynamic, SharedLib **p_shlibs )
{
   struct nlist nm;
#define n_corename      n_un.n_name
   struct link_dynamic ldbuf;
   struct link_dynamic_1 ld1buf;
   struct link_map lmbuf,
                  *lm;
   struct ld_debug lddbuf;

   struct rtc_symb rtcbuf,
                  *rtc;
   SharedLib        *shlist,
                  *sh;
   char            buf[256];

   if(dread(addr_dynamic, (char *) &ldbuf, sizeof(ldbuf)) != 0)
      return -1;

   // Get the list of addresses of shared data objects.
   if(dread((taddr_t) ldbuf.ldd, (char *) &lddbuf, sizeof(lddbuf)) != 0)
      return -1;
   for(rtc = lddbuf.ldd_cp; rtc != NULL; rtc = rtcbuf.rtc_next)
   {
      if(dread((taddr_t) rtc, (char *) &rtcbuf, sizeof(rtcbuf)) != 0)
         return -1;
      if(dread((taddr_t) rtcbuf.rtc_sp, (char *) &nm, sizeof(nm)) != 0)
         return -1;
      if(dgets((taddr_t) nm.n_corename, buf, sizeof(buf)) != 0)
         return -1;
      st_addrlist.insert( save_fname(buf, TRUE), nm.n_value );
   }

   // Now dig out the list of loaded shared libraries.
   if(dread((taddr_t) ldbuf.ld_un.ld_1, (char *) &ld1buf, sizeof(ld1buf)) != 0)
      return -1;
   shlist = NULL;
   for(lm = ld1buf.ld_loaded; lm != NULL; lm = lmbuf.lm_next)
   {
      if(dread((taddr_t) lm, (char *) &lmbuf, sizeof(lmbuf)) != 0)
         return -1;
      if(dgets((taddr_t) lmbuf.lm_name, buf, sizeof(buf)) != 0)
         return -1;
      sh = (SharedLib *) new SharedLib;
      sh->setName( strsave(buf) );
      sh->setAddress( (taddr_t) lmbuf.lm_addr );
      sh->setNext( shlist );
      shlist = sh;
   }

   *p_shlibs = shlist;
   return NULL;
```

```
}


//  Load the shared libraries, and point *p_stlist
//  at the loaded list.  Return 0 for success, -1 for failure.
//  In the normal case all the shared libraries will be in the
//  cache, either from the preload or from a previous run of the
//  target.
int
SharedLib::loadSymTabs( target_info_t *target_info, SymTab **p_stlist )
{
   SymTab *st;

   SymTab *stlist = NULL;
   for( SharedLib *sh = this; sh != NULL; sh = sh->sh_next)
   {
      if( (st = SymTab::getSymTabFromCache( sh->sh_name )) != NULL )
         st->changeTextOffset( sh->sh_addr );
      else
      {
         int shlib_fd;

         if( (shlib_fd = Executable::open( sh->sh_name )) == -1)
            return -1;
         st = new SymTab( sh->sh_name, STT_SHLIB, shlib_fd,
                                 sh->sh_addr, target_info );
      }
      st->setNext( stlist );
      stlist = st;
   }
   *p_stlist = stlist;
   return 0;
}



//  Return a pointer to an array of strings which consists of the
//  environment plus string s, which should be of the form
//  "name=value".

static const char **
add_to_env( const char *s )
{
   extern char **environ;
   const char    **sptr,
                 **envp,
                 **src,
                 **dst;

   for(sptr = environ; *sptr != NULL; ++sptr)
      ;
   envp = dst = (const char **)malloc(sizeof(char *) * (sptr - environ + 2));
   src = environ;
   while (*src != NULL)
      *dst++ = *src++;
   *dst++ = s;
   *dst = NULL;
   return envp;
}

//  Get the list of shared libraries that the execfile will load
//  when it is run.  We do this in the same way as ldd(1), by
//  running the target with LD_TRACE_LOADED_OBJECTS set in its
//  environment and parsing the output.
//
int
SharedLib::getPreloadShLibList( const char *execfile, SharedLib **p_shlibs )
{
   int             fds[2];
   char            buf[256],
                   name[256];
   char           *pos;
```

```
FILE            *fp;

const char **envp = add_to_env("LD_TRACE_LOADED_OBJECTS=1");

if(pipe(fds) != 0)
{
   cerr << "pipe failed in SharedLib::getPreloadShLibList()" << endl;
   free((char *) envp);
   return -1;
}
if((fp = fdopen(fds[0], "r")) == NULL)
{
   cerr << "fdopen failed in SharedLib::getPreloadShLibList()" << endl;

   free((char *) envp);
   return -1;
}

int pid;
if((pid = vfork()) == -1)
{
   cerr << "vfork failed in SharedLib::getPreloadShLibList()" << endl;
   fclose(fp);
   close(fds[1]);
   free((char *) envp);
   return -1;
}
if(pid == 0)
{
   close(fds[0]);
   if(fds[1] != 1)
      dup2(fds[1], 1);
   execle(execfile, execfile, (char *) NULL, envp);
   cerr << "SharedLib::getPreloadShLibList() : Can't exec " << execfile << endl;
   exit(1);
}
free((char *) envp);
close(fds[1]);

SharedLib *shlist = NULL;
while(fgets(buf, sizeof(buf), fp) != NULL)
{
   // We seem to get carriage returns as well as newlines
   if((pos = strchr(buf, '\r')) != NULL)
      *pos = '\0';
   if((pos = strchr(buf, '\n')) != NULL)
      *pos = '\0';
   if(sscanf(buf, "%*s => %s", name) == 1)
   {
      SharedLib *sh = new SharedLib;
      sh->sh_name = strsave(name);
      sh->sh_addr = 0;
      sh->sh_next = shlist;
      shlist = sh;
   }
}
fclose(fp);

int wpid;
while((wpid = wait((union wait *) NULL)) != pid && wpid != -1)
   ;
*p_shlibs = shlist;
return 0;
```

```
/*-------------------------------------------------------------------
 |      CLASS: SourceFile
 |    PURPOSE: Each symbol table has a list of source file objects
 |             associated with it. Each source file object has a list of
 |             functions defined in that source file, a table mapping line
 |             number offsets into the source file (built by callback from
 |             sfile object), and a list of global variables defined in
 |             the source file.  Many of these things are lazily
 |             evaluated.  SourceFile objects are linked together.
 |
 |   CONTENTS: -methods-         -------------purpose-----------------
 |             SourceFile()        constructor
 |             adjustVarAddresses  adjust addresses of files vars
 |             findVar()           find & return variable object in file
 |             getTargetName()     return name of target's binary
 |             open()              open the source file
 |             srcType()           deduce language from file name
 |             getTypes()          load type information for file
 |             getVars()           load variable information for file
 |
 |
 |    NOTES:
 -------------------------------------------------------------------*/

#ifndef SOURCEFILE_H_INCLUDED
#define SOURCEFILE_H_INCLUDED

class SourceFile
{
 public:

   SourceFile( class STFile *stf, Block *parblock, const char *path_hint, SourceFile *next );

   Variable *findVar( const char *name );

   const char *getTargetName();

   long getTargetModTime();


   void adjustVarAddresses( long delta );

   void getTypes( bool is_header );

   Variable *getVars();

   int open( bool want_error_messages );

   static language_t srcType( const char *name );

   //------------ access methods --------------------------------

   SourceFile *getNext() { return fi_next; }
   void setNext( SourceFile *next ) { fi_next = next; }

   void setUseSrcpathOnlyFlag() { fi_use_srcpath_only = TRUE;}

   const char *getName() { return fi_name;}
   void setName( const char *name ) { fi_name=name;}

   language_t getLanguage() { return fi_language;}
   void setLanguage( language_t l ) { fi_language=l;}

   FunctionList *getFunctionList() { return fi_funclist; }
   void setFunctionList( FunctionList *funclist) { fi_funclist = funclist; }

   SFile *getSFile() { return fi_so; }
   void setSFile( SFile *so ) { fi_so = so; }

   Block *getBlock() { return fi_block;}
   void setBlock( Block *bl ) { fi_block = bl;}
```

```
    STFile *getStf() { return fi_stf; }
    void setStf( STFile *stf ) { fi_stf = stf; }

    const char *getPathHint(){ return fi_path_hint; }


    const char                  *fi_name;           // file name
    const char                  *fi_path_hint;      // possible directory of source file
    language_t                  fi_language;        // programming language of the file
    short                       *fi_flags;          // how lazy are we?
    STFile                      *fi_stf;            // symbol table internal source file object
    SFile                       *fi_so;             // handle on file for displaying source
    long                        fi_editblocks_id;   // handle on added editable lines
    class Block                 *fi_block;          // vars and defs with file scope
    FunctionList                *fi_funclist;       // list of functions defined in this file
    class SourceFile            *fi_next;           // next source file
    static bool                 fi_use_srcpath_only;

};
typedef class SourceFile fil_t;

#define FI_DONE_VARS            0x1
#define FI_DONE_TYPES           0x2
#define FI_DOING_TYPES          0x4


#endif
```

```
/*-----------------------------------------------------------------
 |       FILE: sourcefile.C
 |    PURPOSE: Each symbol table has a list of source file objects
 |             associated with it. Each source file object has a list of
 |             functions defined in that source file, a table mapping line
 |             number offsets into the source file (built by callback from
 |             sfile object), and a list of global variables defined in
 |             the source file.  Many of these things are lazily
 |             evaluated.  SourceFile objects are linked together.
 |
 |   CONTENTS: -methods-         -------------purpose-----------------
 |             SourceFile()        constructor
 |             adjustVarAddresses  adjust addresses of files vars
 |             findVar()           find & return variable object in file
 |             getTargetName()     return name of target's binary
 |             open()              open the source file
 |             srcType()           deduce language from file name
 |             getTypes()          load type information for file
 |             getVars()           load variable information for file
 |
 |
 |    NOTES:
 -----------------------------------------------------------------*/

#include <ctype.h>
#include <a.out.h>
#include <stab.h>
#include <errno.h>
#include <iostream.h>
#include "symtab.h"

bool SourceFile::fi_use_srcpath_only = FALSE;


   ---------------------------------------------------------------------
       METHOD: adjustVarAddresses
 |
 |    PURPOSE: Adjust the addresses of all the global variables associated
 |             with the list of sourcefiles.  This method is called when
 |             a shared library mapping address changes across runs of the
 |             target.  It is only necessary to actually adjust the
 |             variables if they have already been loaded.  Otherwise,
 |             just store away the new address.
 ---------------------------------------------------------------------*/
void
SourceFile::adjustVarAddresses( long delta )
{
   var_t          *v;

   for( SourceFile *fil = this; fil != NULL; fil = fil->fi_next)
   {
      STFile *stf = fil->fi_stf;

      stf->setAddress( stf->getAddress() + delta );

      // iff vars are already loaded, then adjust their addresses.
      if( fil->fi_flags & FI_DONE_VARS )
         for( v = fil->fi_block->getVars(); v != NULL; v = v->getNext() )
            v->setAddress( (int)v->getAddress() + (int)delta );
   }
}


/*-----------------------------------------------------------------
       METHOD: findVar

 |    PURPOSE: Search file for a variable.  Return a pointer to the
 |             variable object if found, otherwise return NULL.
 |
 |             If the variables have not yet been loaded, then this method
 |             scans the symbol list for the variable name and doesn't
 |             bother loading in the vars if name is not found.
```

```
 ---------------------------------------------------------------------*/
Variable *
SourceFile::findVar( const char *name )
{
   // For files where the variables have not yet been read, scan the symbols
   // list for name and don't bother to read the vars if it's not there.

   if( !(fi_flags & FI_DONE_VARS))
   {
      for( snlist_t *sn = fi_stf->getSnList(); sn != NULL; sn = sn->getNext() )
         if( strcmp( sn->getName(), name) == 0 )
            break;
      // The var is not in the symbol list, so bag out...
      if( sn == NULL )
         return NULL;
   }

   for( Variable *v = getVars(); v != NULL; v = v->getNext() )
      if( strcmp(v->getName(), name) == 0 )
         return v;
   return NULL;
}


/*---------------------------------------------------------------------
 |     METHOD: getTargetName
 |
 |    PURPOSE: Return name of target's binary.
 |
 ---------------------------------------------------------------------*/
const char *
SourceFile::getTargetName()
{
   return fi_stf->getSymTab()->getTargetInfo()->ti_name;
}


/*---------------------------------------------------------------------
 |     METHOD: getTargetModTime
 |
 |    PURPOSE: Return modification time of target binary.
 |
 ---------------------------------------------------------------------*/
long
SourceFile::getTargetModTime()
{
   return fi_stf->getSymTab()->getTargetInfo()->ti_mod_time;
}




/*---------------------------------------------------------------------
 |     METHOD: open
 |
 |    PURPOSE: Open the source file if it is not already open.
 |             Return 0 if sucessful, otherwise return -1.
 |             Print error messages iff want_error_messages==TRUE.
 |
 ---------------------------------------------------------------------*/
int
SourceFile::open( bool want_error_messages )
{
   if (fi_so != 0)
      return 0;

   if (*fi_name != '/' && fi_path_hint != NULL && !fi_use_srcpath_only)
   {
      char fullname[50];
      strcpy( fullname, fi_path_hint );
      strcat( fullname, fi_name );
```

```
      fi_so = SFile::lookUpFile( fullname, (so_line_callback_t)NULL );

      if( fi_so == 0 && errno != ENOENT )
      {
         if (want_error_messages)
            cout << "Cannot open source file "<< fi_name << endl;
         return -1;
      }
   }

   if( fi_so == 0 )
   {
      fi_so = SFile::lookUpFile( fi_name, (so_line_callback_t)NULL );

      //  If the file doesn't exist try knocking any directory
      //  components off it.

      if( fi_so == 0 && errno == ENOENT )
      {
         const char *basename = strrchr(fi_name, '/');

         if( basename != NULL && basename[1] != '\0' )
         {
            fi_so = SFile::lookUpFile( basename + 1, (so_line_callback_t)NULL);

            //  We want the error for the original
            //  file if we don't find the basename.

            if( fi_so == 0 )
               errno = ENOENT;
         }
      }

      if( fi_so == 0 && want_error_messages )
         cout << "Cannot open source file " << fi_name << endl;
   }

   return (fi_so != 0) ? 0 : -1;
}


/*-------------------------------------------------------------------------
 |     METHOD: srcType
 |
 |    PURPOSE: Deduce language of source file from the source file name.
 |
 -------------------------------------------------------------------------*/
language_t
SourceFile::srcType( const char *name )
{
   char *suf;

   if( (suf = strrchr(name, '.') ) == NULL )
      return LANG_UNKNOWN;
   else
      ++suf;
   if(strcmp(suf, "c") == 0 )
      return LANG_C;
   if( strcmp(suf, "f") == 0 )
      return LANG_FORTRAN;
   return LANG_UNKNOWN;
}


   -------------------------------------------------------------------------
 |     METHOD: SourceFile
 |    PURPOSE: Constructor
 -------------------------------------------------------------------------*/
SourceFile::SourceFile(stf_t *stf,
                       Block *parblock,
                       const char *path_hint,
```

```
                        SourceFile *next )
{

   fi_name = strsave( stf->getName() );
   fi_path_hint = strsave( path_hint );
   fi_language = stf->getLanguage();
   fi_flags = 0;
   fi_stf = stf;
   fi_so = 0;
   fi_editblocks_id = NULL;
   fi_block = new Block( parblock );
   fi_funclist = new FunctionList;

   fi_next = next;
}



/*-----------------------------------------------------------------------
 |     METHOD: getTypes
 |
 |    PURPOSE: Load the type information for the sourcefile.  Of course,
 |             if it's already loaded, then just return.
 -----------------------------------------------------------------------*/
void
SourceFile::getTypes( bool is_header )
{
   symtab_t *st;
   int symno, func_symno;
   bool push_aggr;

   if (fi_flags & FI_DOING_TYPES)
      panic("duplicate file in SourceFile::getTypes");
   if (fi_flags & FI_DONE_TYPES)
      return;
   fi_flags |= FI_DOING_TYPES;

   // get the symtab object
   st = fi_stf->getSymTab();

   // setup an iterator to the beginning of fl_list
   ListIter(FuncPtr) fl_iterator;
   fl_iterator = *fi_funclist;

   func_symno = ( !fi_funclist->listEmpty() )
              ? fi_funclist->listCar()->getSymNumber()
              : -1;

   //  Note that we skip the 0th symbol in the source file, which
   //  is the N_SO or N_BINCL symbol.  This is essential for N_BINCL -
   //  if we didn't skip it we would recurse forever.

   for( symno = fi_stf->getSymNumber() + 1; ; ++symno )
   {
      nlist_t nm;
      const char *name, *s;
      class_t aclass;

      //  Skip over the symbols of functions - we don't
      //  want type information from functions.

      fl_iterator = *fi_funclist;
      for( ; fl_iterator.iterMore() && symno==func_symno; fl_iterator++ )
      {
         Function *f = *fl_iterator;
         symno = f->getSymLimit();

         func_symno = (fl_iterator.iterMore()) ? f->getSymNumber() : -1;
      }

      if( symno >= fi_stf->getSymLimit() )
         break;
```

```
    st->getExecutable()->getSym( symno, nm );
    if( (nm.n_type == N_BINCL || nm.n_type == N_EXCL) && nm.n_value != 0 )
    {
       stf_t *hdrstf;
       Block *hdrbl;
       SourceFile *sf;

       hdrstf = fi_stf->idToHdrStf( nm.n_value );
       sf = hdrstf->getSourceFile();
       sf->getTypes( TRUE );
       hdrbl = sf->getBlock();

       hdrbl->pushTypedefsAndAggrs( fi_block );

       if( nm.n_type == N_BINCL )
          symno = hdrstf->getSymLimit() - 1;
    }

    if (nm.n_type != N_LSYM)
       continue;

    name = fi_stf->getSymTab()->getExecutable()->getSymString( symno );

    //  Skip junk symbols, and ones that aren't TypeDefs (see st_parse.c)

    for (s = name; *s == '_' || isalnum(*s); ++s)
       ;
    if (*s != ':' || (s[1] != 'T' && s[1] != 't'))
       continue;

    ++s;

    type_t *type = Class( fi_stf, &symno, &s, &aclass );

    switch(aclass)
    {
     case CL_TYPEDEF:
       type->ty_typedef->td_next = fi_block->getTypeDef();
       fi_block->setTypeDef( type->getTypeDef() );

       //  Unnamed aggregate types don't get a CL_TAGNAME
       //  entry, so if we get one that hasn't already been
       //  typedefed, fall through to the aggregate pushing code.

       switch (type->ty_code)
       {
        case TY_STRUCT:
        case TY_UNION:
        case TY_ENUM:
          push_aggr = type->ty_typedef == NULL &&
             type->ty_aggr_or_enum->ae_tag == NULL;
          break;
        default:
          push_aggr = FALSE;
          break;
       }
       if (!push_aggr)
          break;
       //  fall though ...
     case CL_TAGNAME:
       type->ty_aggr_or_enum->ae_next = fi_block->bl_aggr_or_enum_defs;
       fi_block->bl_aggr_or_enum_defs = type->ty_aggr_or_enum;
       break;
     case CL_NOCLASS:
       break;
     default:
       panic("bad class in gft");
    }
}

if (!is_header)
```

```
      fi_block->getAggrOrEnum()->applyToAelist( aggr_or_enum_defst::fixUndefAggr,
                                          (char *)fi_block->getAggrOrEnum() );

   fi_flags &= ~FI_DOING_TYPES;
   fi_flags |= FI_DONE_TYPES;
}



/*-------------------------------------------------------------------------
 |     METHOD: getVars
 |
 |    PURPOSE: If the variables are not already loaded for this file,
 |             then this method loads them.  It then returns a pointer
 |             to the first variable object in the list.
 -------------------------------------------------------------------------*/
Variable *
SourceFile::getVars()
{
   nlist_t nm;
   class_t aclass;
   type_t *type;
   taddr_t addr;

   if(fi_flags & FI_DONE_VARS )
      return fi_block->getVars();

   getTypes( FALSE );

   fi_block->setVars( (Variable *)NULL );
   Variable *last = (Variable *)NULL;

   SymTab *symtab = fi_stf->getSymTab();
   Executable *exec = symtab->getExecutable();
   for( snlist_t *sn = fi_stf->getSnList(); sn != NULL; sn = sn->getNext() )
   {
      int symno = sn->getSymNumber();
      exec->getSym( symno, nm );

      const char *s = exec->getSymString( symno );
      (void) parseName( &s );

      //  Ignore symbols with funny characters in their names.
      if (*s != ':')
         continue;
      ++s;

      type = Class( fi_stf, &symno, &s, &aclass );
      if (nm.n_value != 0)
         addr = (taddr_t)nm.n_value + symtab->getTextOffset();
      else
         addr = symtab->lookupGlobalAddr( sn->getName() );

      if (addr != 0 && (aclass == CL_EXT || aclass == CL_STAT))
      {
         Variable *v = new Variable( sn->getName(), aclass, type, addr );
         v->setLanguage( fi_language );
         v->setNext( fi_block->getVars() );
         if( last != NULL )
            last->setNext( v );
         else
            fi_block->setVars( v );
         last = v;
      }
   }
   if (last != NULL)
      last->setNext( NULL );
   fi_flags |= FI_DONE_VARS;

   return fi_block->getVars();
}
```

```
/*-----------------------------------------------------------------
 |       FILE: Stack.h
 |    PURPOSE: Declarations for stack trace class and frames class.
   -----------------------------------------------------------------*/

#ifndef STACK_H_INCLUDED
#define STACK_H_INCLUDED

#include "proc.h"

class SigInfo
{
 public:

   SigInfo( int signo ) { si_signo = signo; si_fp = 0; }

   int        si_signo;
   taddr_t    si_fp;
   Preamble  *si_prbuf;
};
typedef class SigInfo siginfo_t;


//
//  Element of the linked list describing the stack.
//

#define stk_ap  stk_fp

class Frame
{
 public:
   Frame() {;}
   Frame( Function *f, taddr_t pc, int lnum, Frame *last );
    Frame();

   taddr_t getRegisterAddress( int reg );

 public:
   Function *fr_func;
   taddr_t   fr_pc;           // saved program counter
   taddr_t   fr_fp;           // frame pointer
   int       fr_lnum;         // source lnum corresponding to fr_pc
   SigInfo  *fr_siginfo;      // signal that cause func call, if not 0
   short     fr_bad;          // stack corrupted after this frame
   Frame    *fr_next_down;    // next frame down
   Frame    *fr_next_up;      // next frame up
};
typedef class Frame frame_t;



class Stack
{
 public:

   // static Stack buildTrace()
   Stack( Process *proc );

   // void freeTrace()
   ~Stack();

   // Up/down - moves the current frame "up" and "down" the stack trace
   // returns 0 if successful, 1 if unsucessful (e.g. already at the top)
   int up();
   int down();

   static Stack *getCurrentStack() { return stk_current_stack; }

   // get the current frame
   Frame *getCurrentFrame()    { return stk_current_frame; }
```

```
    void getCurrentFunction( Function **p_func, taddr_t *p_fp, taddr_t *p_pc, taddr_t *p_adjusted

  protected:

    void discardJunkFrames();

    //  Pointer to the stack frame of the "current function".  This is the
    //  function that next and step refer to, and where local variables
    //  are initially searched for.
    //
    Frame        *stk_current_frame;

  public:

    //  Pointers to the top and bottom frame in the list of stack frames.
    //  NULL if there is no stack trace.
    //
    Frame        *stk_bottom;    // bottom frame in list of stack frames
    Frame        *stk_top;       // top frame in list of stack frames

    static Stack *stk_current_stack;
};
typedef class Stack stack_t;


#endif
```

```
/*-----------------------------------------------------------------
 |       FILE: Stack.C
 |    PURPOSE: Methods for stack trace class and frames class.
 |      NOTES:   Stack objects are composed of one or more frame objects
 -----------------------------------------------------------------*/
#include <signal.h>
#include <sun4/frame.h>

#include "symtab.h"
#include "proc.h"
#include "data.h"
#include "preamble.h"
#include "stack.h"
#include "target.h"

Stack *Stack::stk_current_stack = NULL;

/*-----------------------------------------------------------------
 |     METHOD: Frame()
 |
 |    PURPOSE: Constructs a stack frame in a stack trace, where last is
 |             the last stack frame added to the trace.
 |      NOTES:
 -----------------------------------------------------------------*/
Frame::Frame( Function *f, taddr_t pc, int lnum, Frame *last )
{
   fr_bad     = FALSE;
   fr_pc      = pc;
   fr_siginfo = NULL;
   fr_func    = f;
   fr_lnum    = lnum;
   if( last ) last->fr_next_up = this;
   fr_next_up   = NULL;
}

/*-----------------------------------------------------------------
 |     METHOD: ~Frame()
 |
 |    PURPOSE: Destructs a stack frame in a stack trace.
 |      NOTES:
 -----------------------------------------------------------------*/
Frame::~Frame()
{
   if( fr_siginfo )
      delete fr_siginfo;
}

/*-----------------------------------------------------------------
 |     METHOD: ~Stack()
 |
 |    PURPOSE: Free a complete stack trace.
 |      NOTES:
 -----------------------------------------------------------------*/
Stack::~Stack()
{
   stk_current_stack = NULL;

   // free the stack frames in the trace
   Frame *next;
   for( Frame *frame=stk_top; frame; frame=next )
   {
      next = frame->fr_next_up;
      delete frame;
   }

/*-----------------------------------------------------------------
 |     METHOD: Stack()
```

```
  |
  |    PURPOSE: Build a stack trace for a process.  The stack trace is a
  |             list of frame objects linked by the fr_next_up field.
  |             Initialized the current frame to the most outer frame.
  |      NOTES: Deletes current_stack if not null.
  ------------------------------------------------------------------------*/
Stack::Stack( Process *proc )
{
   //  FRAGILE CODE
   //
   //  Build a stack trace from a core file or process.
   //
   //  If bat breaks when you get a new release of an OS, this is the
   //  place to look.  This function has lots of machine and compiler
   //  dependent assumptions about stack layout (especially when signals
   //  are involved).
   //

   Frame          *frame, *last;
   Function       *f;
   struct frame    fbuf;
   bool            must_delete_f=FALSE;

   if( stk_current_stack )
   {
      delete stk_current_stack;
      stk_current_stack = NULL;
   }

   stk_top = stk_bottom = stk_current_frame = NULL;

   taddr_t pc    = proc->getRegister(REG_PC);
   taddr_t fp    = proc->getRegister(REG_FP);
   taddr_t savpc = proc->getRegister(31);

   for (frame = last = NULL;; last = frame)
   {
      if ((f = SymTab::addrToFunc(pc)) == NULL)
      {
         must_delete_f = TRUE;
         f = new Function( "{DUMMY}", 0 , (taddr_t)NULL,
                           (SymTab *)NULL, (SourceFile*)NULL, (Function *)NULL );
         f->setFlags(FU_NOSYM | FU_DONE_LNOS | FU_DONE_BLOCKS );
      }

      if (strcmp(f->getName(), "_sigtramp") == 0 && last != NULL)
      {
         struct
         {
            int             signo,
                            code;
            taddr_t         scp;
         }               sigargs;
         struct sigcontext scbuf;

         // From looking at stack dumps ...
         if (dread(fp + 0x40, (char *) &sigargs, sizeof(sigargs)) != 0)
            break;
         last->fr_siginfo = new SigInfo(sigargs.signo);
         last->fr_siginfo->si_fp = fp;

         if (dread(sigargs.scp, (char *) &scbuf, sizeof(scbuf)) != 0)
            break;
         if (dread(fp, (char *) &fbuf, sizeof(fbuf)) != 0)
            break;

         fp = (taddr_t) fbuf.fr_savfp;
         pc = (taddr_t) scbuf.sc_pc;
         if ((f = SymTab::addrToFunc(pc)) == NULL)
         {
            must_delete_f = TRUE;
            f = new Function( "{DUMMY}", 0 , (taddr_t)NULL,
```

```
                                (SymTab *)NULL, (SourceFile*)NULL, (Function *)NULL );
            f->setFlags(FU_NOSYM | FU_DONE_LNOS | FU_DONE_BLOCKS );
         }

         if (f->hasFramePointer())
            savpc = fbuf.fr_savpc;
      }
      // Sun is unusual in that saved pc values point to the address of
      // the call instruction, rather than the one after the call.  Thus we
      // don't have to offset the pc to get the right line number.

      int lnum = f->addrToLNumber(pc);

      frame = new Frame(f, pc, lnum, last);
      frame->fr_fp = fp;

      // if this is the first (top) frame, then set stk_bottom
      if( !stk_bottom ) stk_bottom = frame;

      // If we are not the innermost function, adjust fr_pc so that it
      // points to the location where the called function will jump to on
      // return.

      if (last != NULL && last->fr_siginfo == NULL)
         frame->fr_pc += 8;

      if (fp == NULL)
         break;

      // Get the next stack frame
      //
      // The pc > f->fu_addr test is to cover the case where a signal has
      // interrupted a function before the frame setup preamble has executed.

      if (f->hasFramePointer() && pc > f->getAddress())
      {
         if (dread(fp, (char *) &fbuf, sizeof(fbuf)) != 0)
            break;
         fp = (taddr_t) fbuf.fr_savfp;
         pc = savpc;
         savpc = fbuf.fr_savpc;
      }
      else
      {
         if (last == NULL)
            pc = proc->getRegister(15);
         else
         {
            // We can't cope with non frame pointer function that calls
            // another function.

            if (last->fr_siginfo == NULL)
               break;
            pc = savpc;
            savpc = fbuf.fr_savpc;
         }
      }
   }

   if (fp != NULL && frame != NULL)
      frame->fr_bad = TRUE;

   if( must_delete_f )
      delete f;

   stk_top = frame;
   discardJunkFrames();


   // build up links
   Frame *prev = NULL;
   for( frame=stk_bottom; frame; frame=frame->fr_next_up )
```

```
    {
        frame->fr_next_down = prev;
        prev = frame;
    }
    stk_current_stack = this;
    stk_current_frame = stk_bottom;
}



/*-----------------------------------------------------------------------
 |      METHOD: discardJunkFrames()
 |
 |     PURPOSE: This method removes all frames that appear in the
 |              trace before the main() stack frame.  So after calling
 |              this method, the bottom frame in the trace will be main.
 |       NOTES:
 -----------------------------------------------------------------------*/
void
Stack::discardJunkFrames()
{
    // start at the bottom and walk up stack until we come to main
    Function *main_func = Target::getMainFunction();
    if (main_func != NULL)
    {
        Frame *frame = stk_bottom;
        while( frame!=NULL && !frame->fr_bad && frame->fr_func!=main_func )
            frame = frame->fr_next_up;

        // now delete all frames that appear above main in the stack trace
        if( frame!=NULL && frame->fr_func==main_func )
        {
            stk_top = frame;
            Frame *next = frame->fr_next_up;
            frame->fr_next_up = NULL;
            for( frame=next; frame; frame=next )
            {
                next = frame->fr_next_up;
                delete frame;
            }
        }
    }

}



taddr_t
Frame::getRegisterAddress( int reg )
{
    if (this == NULL)
        return 0;
    else if (reg < 32)
    {
        taddr_t         fp;

        Frame *frame = this;

        if (reg < 8)
            panic("reg_addr: can't do global regs");

        if (reg < 16)
        {
            frame = frame->fr_next_up;
            reg += 16;
        }

        // Integer register.  Just return the address in the register window
        // (as it appears on the stack).
        //
        // On the SPARC, a called function conceptually saves the caller's local
        // and in registers on the stack, as the first 16 words of it's stack
```

```
      // frame (the kernel hides the register window gunk from us).

      if (frame->fr_siginfo != NULL && frame->fr_siginfo->si_fp != 0)
         fp = frame->fr_siginfo->si_fp;
      else if (frame->fr_func->hasFramePointer())
         fp = frame->fr_fp;
      else
         return frame->fr_next_up->getRegisterAddress(reg);
      return fp + (reg - 16) * 4;
   }
   else if (reg < 64)
   {
#define STF     ((3 << 30) | (044 << 19))

// Max 32 fp regs (paranoia)
#define NINST   32

// SPARC reg %o6
#define SPREG   14

      int             text[NINST];
      int             i;

      // Floating point register.  Search back through the instructions for
      // the floating point save.

      reg -= 32;

      if (fr_next_down->fr_func->getSymTab()->getExecutable()->readText( fr_next_down->fr_pc - s
                                                         (char *) text,
                                                         sizeof(text)) != 0)
      {
         return 0;              // BUG: not right
      }
      for (i = NINST - 1; i > 0; --i)
      {
         int inst = text[i];
         int val = inst & ~((1 << 12) - 1);
         int expected = STF | (reg << 25) | (SPREG << 14) | (1 << 13);
         if (val == expected)
         {
            return fr_fp + (inst & ((1 << 12) - 1));
         }
      }
   }
   else
      panic("reg_addr: reg > 63");

   return 0;
}



/*-----------------------------------------------------------------------
 |     METHOD: getCurrentFunctiion
 |
 |    PURPOSE: Return the "current" function, and the pc and fp values
 |             for that function.
 |      NOTES:
 -----------------------------------------------------------------------*/
void
Stack::getCurrentFunction( Function **p_func, taddr_t *p_fp, taddr_t *p_pc, taddr_t *p_adjusted_
{
   if (stk_bottom == NULL)
      *p_func = NULL;
   else
   {
      *p_func = stk_bottom->fr_func;
      *p_pc = stk_bottom->fr_pc;
      *p_fp = stk_bottom->fr_fp;
      *p_adjusted_pc = *p_pc;
```

```
      // If this is not the top frame, the pc is pointing just after
      // a call instruction.  This means it may already be pointing at the
      // next line, so drop it by one to ensure that it's pointing at the
      // right line.

      if (stk_bottom != stk_top)
         --* p_adjusted_pc;
   }
}

int
Stack::up()
{
   int rc = 0;
   if( stk_current_frame==stk_top )
      rc = 1;
   else
      stk_current_frame = stk_current_frame->fr_next_up;

   return rc;
}


int
Stack::down()
{
   int rc = 0;
   if( stk_current_frame==stk_bottom )
      rc = 1;
   else
      stk_current_frame = stk_current_frame->fr_next_down;

   return rc;
}
```

```
/*-----------------------------------------------------------------
 |       FILE: strcache.h
 |    PURPOSE: class for random file access via LRU buffer cache.
 -----------------------------------------------------------------*/

#ifndef SC_H_INCLUDED
#define SC_H_INCLUDED

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/file.h>
#include <sys/types.h>
#include <sys/stat.h>



#include <string.h>
#include "gendefs.h"

#include <sysent.h>
#include <memory.h>

typedef int (*sc_input_func_t)(char *arg, off_t pos, char *buf, int nbytes);
typedef void (*sc_close_func_t)(char *arg);



//  A type for offsets.
//
typedef unsigned long offset_t;

typedef long timestamp_t;

typedef struct bufst {
        timestamp_t b_last_used;
        offset_t b_offset;
        offset_t b_lim;
        int b_len;
        char *b_data;
} buf_t;

#ifdef STATS
typedef struct statsst {
        int finds;
        int incache;
        int reads;
        int calls;
        int ifinds;
        int onebuf;
        int multi;
        int gt2;
} stats_t;
#endif

class FDobj
{
 public:
   FDobj( int fd ) { fd_fd = fd;}
   ~FDobj() {;}

   virtual int getInput( off_t offset, char *buf, int nbytes );
   virtual int getFD() { return fd_fd; }

   protected:
   int fd_fd;
};

//
// StrCache object class
//
```

```
class StrCache
{
 public:

   // constructor
   StrCache( int fd );

   StrCache( FDobj *fdo ){ sc_fdobj = fdo; makeStrCache( ); }

   // copy constructor
   StrCache( const StrCache& sc );

   // destructor
   ~StrCache( );

   off_t getOffset() { return sc_offset; }

   void setOffset( off_t offset ) { sc_offset = offset; }

   int setBufs( int nbufs, int bufsize );

   StrCache *dup_strcache();

   void fileHasGrown();

   void forgetBuffers();

   const char *getString( off_t offset, int endchar, long& p_len );
   char *getBytes( off_t offset, int nbytes, long& p_len );

 protected:

   void makeStrCache();

   buf_t *findBuf( offset_t offset );

   FDobj *sc_fdobj;
   buf_t *sc_last_buf;
   buf_t *sc_bufs;
   int sc_nbufs;
   unsigned long sc_bufsize;
   off_t sc_offset;
   char *sc_saved_line;
   long sc_saved_line_len;
   timestamp_t sc_current_stamp;
#ifdef STATS
   stats_t sc_stats;
#endif
};


#endif
```

```
/*-----------------------------------------------------------------
 |       FILE: strcache.C
 |    PURPOSE: class for random file access via LRU buffer cache.
  +-----------------------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/file.h>
#ifdef STATS
#include <sys/stat.h>
#endif
#include <sysent.h>
#include <memory.h>
#include "strcache.h"

#ifndef L_SET
#define L_SET   0
#endif

#ifdef STATS
#define DBINC(smem)     ++sc_stats.smem
#else
#define DBINC(smem)
#endif

#ifdef STATS

typedef unsigned long bitmap_t;

#define BMBITS (sizeof(bitmap_t) * 8)
#endif

   Default number of buffers.  Change via the NBUFS env var.

#define DEFAULT_NBUFS   4

//  Default size of a single buffer.
//  Can be made changed by the environment for tuning.
//
//  This must be a power of two.  See the check below.
//
#define DEFAULT_BUFSIZE 2048

//  Compile time check that BUFSIZE is a power of two.
//
#if (BUFSIZE & (BUFSIZE - 1)) != 0
#include "bufsize is not a power of two"
#endif


void
panic(const char *mesg)
{
   fprintf(stderr, "Fatal internal error: %s (aborting)\n", mesg);
   fflush(stderr);
   abort();
}

// ------------------- FDobj method -----------------------------
int
FDobj::getInput( off_t offset, char *buf, int nbytes )
{
   if (lseek(fd_fd, offset, L_SET) == -1)
      return -1;
   return read(fd_fd, buf, nbytes);
}


// ------------------- StrCache methods -------------------------
void
StrCache::makeStrCache( )
```

```
{
   static buf_t dummy_buf;      /* we rely on this being initialised to zeroes */

   sc_last_buf = sc_bufs = &dummy_buf;
   sc_nbufs = 0;
   sc_bufsize = 0;
   sc_offset = 0;
   sc_saved_line = NULL;
   sc_saved_line_len = 0;
   sc_current_stamp = 0;

#ifdef STATS
   {
      static stats_t zero_stats;

      sc_stats = zero_stats;
   }

#endif
}



StrCache::StrCache( int fd )
{
   sc_fdobj = new FDobj( fd );
   makeStrCache();
}


// Close up the StrCache
StrCache::~StrCache()
{
   delete sc_fdobj;
   delete [] sc_bufs->b_data;
   delete [] ((char *)sc_bufs);
   if (sc_saved_line != NULL)
      delete [] sc_saved_line;
}

// Copy Constructor
//  This is used when you want lots of interleaved accesses to different
//  parts of a file - it reduces contention for the buffers.  It is also
//  useful when you want different buffer sizes/counts for the different
//  bits of the file.
//
StrCache::StrCache( const StrCache& sc )
{
   sc_fdobj = new FDobj( sc.sc_fdobj->getFD() );
   makeStrCache();

   sc_offset = sc.sc_offset;
}


int
StrCache::setBufs(int nbufs, int bufsize)
{
   if (nbufs == 0)
      nbufs = DEFAULT_NBUFS;

   if (bufsize == 0)
      bufsize = DEFAULT_BUFSIZE;


   if ((bufsize & (bufsize - 1)) != 0)
      panic("bufsize not a power of two");

   buf_t *bufs = new buf_t[nbufs];

   char *data = new char[nbufs * bufsize];
```

```
    if( data == NULL )
       return -1;
    for (buf_t *b = bufs; b < bufs + nbufs; ++b) {
       b->b_offset = b->b_lim = 0;
       b->b_len = 0;
       b->b_last_used = 0;
       b->b_data = data;
       data += bufsize;
    }

    sc_bufs = bufs;
    sc_last_buf = bufs;
    sc_nbufs = nbufs;
    sc_bufsize = bufsize;

    return 0;
}



buf_t *
StrCache::findBuf( offset_t offset )
{
    register buf_t *b;
    timestamp_t lru_time;
    offset_t buf_offset;
    int len;
    buf_t *lru_buf, *maxbuf;

    DBINC(finds);

    if( sc_nbufs == 0 )
    {
       if( setBufs( 0, 0 ) != 0 )
          return NULL;
    }

    b = sc_bufs;
    maxbuf = b + sc_nbufs;
    lru_time = b->b_last_used;
    lru_buf = b;
    buf_offset = offset & ~(sc_bufsize - 1);

    for (; b < maxbuf; ++b) {
       if (b->b_len > 0 && b->b_offset == buf_offset) {
          if (offset < b->b_lim) {
             DBINC(incache);
             return b;
          }
          return NULL;
       }
       if (b->b_last_used < lru_time) {
          lru_time = b->b_last_used;
          lru_buf = b;
       }
    }

    //  Not found - reallocate the least recently used buffer.

    b = lru_buf;

    //  Get the data.

    DBINC(reads);

    len = sc_fdobj->getInput( buf_offset, b->b_data, (int)sc_bufsize );
    if (len < 0)
       return NULL;

    b->b_offset = buf_offset;
    b->b_len = len;
```

```
    b->b_lim = buf_offset + len;
    b->b_last_used = sc_current_stamp++;

    return (offset < b->b_lim) ? b : NULL;
}


//  This is called by the application when it thinks that the file
//  has got bigger (but is unchanged up to the original size).
//
void
StrCache::fileHasGrown()
{
    buf_t *b;

    for (b = sc_bufs; b < sc_bufs + sc_nbufs; ++b) {
        if (b->b_len < sc_bufsize) {
            b->b_lim = b->b_offset;
            b->b_len = 0;
        }
    }
}

void
StrCache::forgetBuffers()
{
    buf_t *b;

    for (b = sc_bufs; b < sc_bufs + sc_nbufs; ++b) {
        b->b_lim = b->b_offset;
        b->b_len = 0;
    }
}


const char *
StrCache::getString( off_t offset, int endchar, long &p_len )
{
    register buf_t *b;
    char *start, *end, *res;
    int res_len, alloc_len;

    DBINC(calls);

    offset += sc_offset;
    if (offset < 0)
        panic("offset < 0 in StrCache::getString");
    b = sc_last_buf;
    if (b->b_offset > offset || offset >= b->b_lim) {
        if ((b = findBuf( offset )) == NULL) {
            p_len = 0;
            return NULL;
        }
        sc_last_buf = b;
    }
    else {
        DBINC(ifinds);
    }

    //  Try for the simple case, where the string doesn't cross a
    //  buffer boundary.
    //
    start = b->b_data + (offset - b->b_offset);
    end = (char *)memchr(start, endchar, b->b_data + b->b_len - start);

    //  Check for the case where we don't find the terminating
    //  character because we hit EOF on the block.
    //
    if (end == NULL && b->b_len < sc_bufsize)
        end = b->b_data + b->b_len;

    if (end != NULL) {
```

```
        DBINC(onebuf);
        p_len = end - start;
        return start;
    }

    //  we've got a string that crosses a buffer boundary.
    //
    //  Copy it chunk by chunk into the sc_saved_line buffer.
    //
    DBINC(multi);

    res = sc_saved_line;
    res_len = (int)sc_saved_line_len;
    if (res == NULL) {
        res_len = (int)sc_bufsize;
        res = new char[res_len];
        sc_saved_line = res;
        sc_saved_line_len = res_len;
    }

    alloc_len = b->b_len - (start - b->b_data);
    memcpy(res, start, alloc_len);

    do {
        int chunk_len;

        if ((b = findBuf(offset + alloc_len)) == NULL)
            break;

        if ((end = (char *)memchr(b->b_data, endchar, b->b_len)) != NULL)
            chunk_len = end - b->b_data;
        else
            chunk_len = b->b_len;

        if (alloc_len + chunk_len > res_len) {
            res_len *= 2;
            res = realloc(res, res_len + 1);
            sc_saved_line = res;
            sc_saved_line_len = res_len;
        }
        memcpy(res + alloc_len, b->b_data, chunk_len);

        alloc_len += chunk_len;
    } while (end == NULL);

#ifdef STATS
    if (alloc_len >= sc_bufsize * 2) {
        DBINC( gt2);
    }
#endif

    //  We don't put the terminator character on the end of the string
    //  if we hit EOF without finding it.
    //
    res[alloc_len] = (b != NULL) ? endchar : '\0';

    p_len = alloc_len;
    return res;
}

char *
StrCache::getBytes(off_t offset, int nbytes, long& p_len )
{
    register buf_t *b;
    char *start, *res;
    int res_len, ncopied, remaining;

    DBINC( calls);

    offset += sc_offset;
    if (offset < 0)
        panic("offset < 0 in scb");
```

```
    b = sc_last_buf;
    if (b->b_offset > offset || offset >= b->b_lim) {
        if ((b = findBuf(offset)) == NULL) {
            p_len = 0;
            return NULL;
        }
        sc_last_buf = b;
    }
    else {
        DBINC( ifinds);
    }

    //  Try for the simple case, where the string doesn't cross a
    //  buffer boundary.
    //
    start = b->b_data + (offset - b->b_offset);
    if (start + nbytes <= b->b_data + b->b_len) {
        DBINC( onebuf);
        p_len = nbytes;
        return start;
    }

    //  we've got a string that crosses a buffer boundary.
    //
    //  Copy it chunk by chunk into the sc_saved_line buffer.
    //
    DBINC( multi);

    res = sc_saved_line;
    res_len = (int)sc_saved_line_len;
    if (res == NULL) {
        res_len = (int)((nbytes > sc_bufsize) ? nbytes : sc_bufsize);
        res = new char[res_len];
        sc_saved_line = res;
        sc_saved_line_len = res_len;
    }
    else if (nbytes > res_len) {
        while (nbytes > res_len)
            res_len *= 2;
        delete [] res;
        res = new char[res_len];
        sc_saved_line = res;
        sc_saved_line_len = res_len;
    }

    ncopied = b->b_len - (start - b->b_data);
    remaining = nbytes - ncopied;
    memcpy(res, start, ncopied);

    do {
        int to_copy;

        if ((b = findBuf(offset + ncopied)) == NULL)
            break;

        to_copy = (remaining < b->b_len) ? remaining : b->b_len;
        memcpy(res + ncopied, b->b_data, to_copy);
        ncopied += to_copy;
        remaining -= to_copy;
    } while (remaining > 0);

#ifdef STATS
    if (ncopied >= sc_bufsize * 2) {
        DBINC( gt2);
    }
#endif

    p_len = ncopied;
    return res;
}
```

```
/*-------------------------------------------------------------------
 |       FILE: symtab.h
 |    PURPOSE: symbol table objects
  -------------------------------------------------------------------*/

#ifndef SYMTAB_H_INCLUDED
#define SYMTAB_H_INCLUDED

#include "utils.h"
#include "gendefs.h"
#include "sfile.h"
#include "/pro/forest/basis/src/list.H"
#include "/pro/forest/basis/src/dynarray.H"
#include "preamble.h"
#include <errno.h>
#include <iostream.h>
#include <ctype.h>
#include <a.out.h>
#include <stab.h>

class Function;
class Executable;
class Type;
class Variable;
class SourceFile;
class Block;
class FSymInfo;
class SymTab;
class FunctionTab;
class STFile;
class SharedLib;
struct Header;
struct typedefst;

   clude "language.h"
   clude "type.h"
#include "executable.h"
#include "function.h"
#include "sourcefile.h"
#include "variable.h"
#include "block.h"
#include "shit.h"


typedef struct ftypest
{
   ftypest(){ ft_tnum=0; ft_type=NULL; ft_next=NULL;}

   int            ft_tnum;
   class Type     *ft_type;
   struct ftypest *ft_next;
} ftype_t;


// This represents a quick scan of the symbol list. For each symbol,
// it contains its number and name.  This allows us to scan through
// the symbol list quickly when we are searching for a symbol.  We
// can then decide if it is necessary to load the symbol list.
// For an example, check out SourceFile::findVar()...

typedef struct snliststst
{
   snliststst *getNext() { return sn_next; }
   void setNext( snliststst *n ) { sn_next = n; }
   int getSymNumber() { return sn_symno; }//
   void setSymNumber( int symno ) { sn_symno = symno; }
   const char *getName() { return sn_name; }
   void setName( const char *name ) { sn_name = name; }

   int              sn_symno;
   const char       *sn_name;
   struct snliststst  *sn_next;
```

```
} snlist_t;



/*-------------------------------------------------------------------------
 |      CLASS: AddrList
 |
 |    PURPOSE: A list of global variable names and their addresses.
 -------------------------------------------------------------------------*/

//  Element in the AddrList.
//

typedef class AddrNode
{
 public:

   AddrNode( const char *name, taddr_t addr ) { an_name = name; an_addr = addr; }


   const char         *an_name;
   taddr_t            an_addr;
   struct addrliststst  *an_next;
} addrnode_t;

typedef addrnode_t *addrptr;
class_List(addrptr);

class AddrList
{
 public:

   void insert( const char *name, taddr_t addr );

   void adjustAddrOffset( long delta );

   taddr_t search( const char *name );

 protected:

   // linked list of addrnode_t pointers
   List(addrptr)     al_list;

};


typedef enum symtab_typeen { STT_MAIN, STT_SHLIB } symtab_type_t;


// Target information
typedef struct
{
   const char                *ti_name;                // Name of target binary
   long                      ti_mod_time;             // Mod time of executable file
   int                       ti_highlighted_lnum;     // Current highlighted line of source
   class SourceFile          *ti_highlighted_fil;     // File the above line is in
} target_info_t;



/*-------------------------------------------------------------------------
 |      CLASS: SymTab
 |    PURPOSE: One per a.out format file, and one for each Sun shared
 |             library.  These objects represent the "root" of the
 |             symbol table "tree".  Some of the information stored here
 |             is the list of source files, addr->function name mappings,
 |             addr->global variable names & types mappings.
 -------------------------------------------------------------------------*/
class SymTab
{
```

```
public:
  // Constructor
  //  Do a prescan of the symbol table of execfile, which is assumed to
  //  be an a.out file.
  //
  //  Fd is a read only file descriptor referring to execfile.
  //
  SymTab(const char *execfile,
         symtab_type_t symtab_type,
         int fd,
         taddr_t text_addr_offset,
         target_info_t *ti );


  // Destructor
  ~SymTab();

  // Given a filename and a line number, this method returns the line number of
  // the closest statement.  It also returns the function object and address
  // of the returned line number.  If unsuccessful, it returns -1.
  static int lNumToFuncAndAddr( const char *file, int lnum, Function **p_func, taddr_t *p_addr


  //  Dig out the list of loaded shared libraries and run time linked
  //  global addresses.
  //
  int getSharedLibsAndGlobalAddresses( taddr_t addr_dynamic, SharedLib **p_shlibs );


  //  Get the main symbol table for execfile, via fd which should
  //  be a file descriptor referring to execfile.  The file pointer
  //  is assumed to be at the start of the file.
  //
  //  If execfile is a shared library, we try to guess which
  //  shared libraries it will use and preload these into the
  //  shared library cache.  This is to allow the user to set
  //  breakpoints in shared library functions before the target
  //  is started, and to avoid a long pause when the target is
  //  started for the first time.

  static int getAndInstallSymTabs( const char *execfile, int fd );

  //  Look up the symbol table for name in the cache.  Remove
  //  it from the cache and return it if found, otherwise
  //  return NULL.
  //
  //  Name is the name of the executable or shared library file
  //  that the symbol table came from.

  static SymTab *getSymTabFromCache( const char *name );

  static void freeCache();

  void skim(taddr_t first_addr,
            taddr_t last_addr,
            Block *rootblock,
            SourceFile **p_sfiles,              // return param
            FunctionTab **p_functab_id );       // return param

  void debugLoadSymbols( const char *name );


  //  Call( *func)() for each source file with the source file as an argument.
  //
  //  Do the source files in the cache as well.
  static void iterateOverSourceFiles( void (*func)(SourceFile *fil) );


  //  Load the symbol tables for the shared libraries of an object.
  //  This is called just after the target starts, because only
  //  then have the shared libraries been loaded and mapped.
  static int loadSharedLibrary();
```

```
  //  Unload shared library symbol tables.  This is called when the
  //  target dies.  We don't free the symbol tables at this point - we
  //  just put them in the cache in anticipation of using them again
  //  when the target is re-run.

  static void unloadSharedLibrary();

  //  Find the file where the global variable called name is defined.
  //
  //  If we find the file, point *p_fil at it and set *p_v to the var,
  //  otherwise set both *p_fil and *p_var to NULL.

  static void findFileOfGlobal( const char *name, SourceFile **p_fil, Variable **p_v );

  //  Return a pointer to the Function object for function name,
  //  or NULL if there is no such function or name is ambiguous.
  //  We allow matches where name is the first part of the full
  //  function name if there is only one such match (if there is
  //  more than one we return NULL and set *p_ambig to TRUE).
  //
  //  We search all the symbol tables including those in the cache.

  static Function * nameToFunction( const char *name, int *p_ambig );

  //  Deal with a change in the text offset of a symbol table.  This may
  //  be necessary when re-running the target as shared libraries may be
  //  mapped at different addresses.  It's also necessary when we have
  //  preloaded symbol tables with a nominal offset of zero.
  //
  //  We adjust the following:
  //
  //   function and line number addresses
  //   symbol table address to text file offset
  //   addresses of global variables
  //
  //  We don't change breakpoint addresses here - we do that by removing
  //  and recreating all breakpoints just after starting the target
  //
  void changeTextOffset( taddr_t new_addr );

  //  Return a pointer to the Function object of the function wherein
  //  lies address addr, or NULL if addr does not lie within any
  //  known function.  We search all the symbol tables.

  static Function *addrToFunc( taddr_t addr );

  void removeBreakpoints();

  //   Find the address of a global.  Return 0 if not found (this will happen
  //   for an extern with no definition).
  //
  taddr_t lookupGlobalAddr( const char *name );

  Executable *getExecutable() { return st_exec; }
  void setExecutable( Executable *exec ) { st_exec = exec; }

  long getTextOffset() { return st_text_addr_offset; }
  void setTextOffset( long offset ) { st_text_addr_offset = offset; }

  target_info_t *getTargetInfo() { return st_target_info; }
  void setTargetInfo( target_info_t *t ) { st_target_info = t; }

  SymTab *getNext() { return st_next; }
  void setNext( SymTab *next ) { st_next = next; }

  FunctionTab *getFunctionTab() { return st_functab; }
  void setFunctionTab( FunctionTab *ftab) { st_functab = ftab; }

protected:
  static bool       st_table_loaded;          // has the symbol table been loaded yet?
```

```
    const char        *st_name;                 // Name of the a.out file
    symtab_type_t     st_type;                  // Type (STT_MAIN or STT_SHLIB)
    int               st_dynamic;               // TRUE if dynamically linked
    target_info_t     *st_target_info;          // Per target information

    long              st_text_addr_offset;      // Offset in process of start of text

    Executable        *st_exec;                 // Symbol input from a.out file stuff
    class SourceFile  *st_sfiles;               // all source files
    FunctionTab       *st_functab;              // Addr --> func mapping table
    AddrList          st_addrlist;              // all addresses of globals

    class SymTab      *st_next;                 // Next symbol table

    static class SymTab *st_main_symtab;        // main symbol table list

    // Cached list of free symbol tables.  We keep this only between one run
    // of the target and the next.

    static SymTab *st_symtab_cache_list;

};
typedef class SymTab symtab_t;


/*-------------------------------------------------------------------------
 |      CLASS: STFile
 |    PURPOSE: Internal symbol table source file object.
 -------------------------------------------------------------------------*/
//  Flag bits in stf_flags.
#define STF_LNOS_PRECEDE_FUNCS  0x01

// don't display this under source files
#define STF_HIDE                0x02

class STFile
{
 public:
   STFile(const char *name,
          symtab_t *st,
          int symno,
          language_t language,
          taddr_t addr );

   void wrapupStf( Header **fmap, int mapsize );
   Type *lookupTnum( int tnum );
   ftype_t *addType( int tnum );

   void addToSnList( snlist_t *sn ) { sn->sn_next = stf_sn; stf_sn=sn;}

   Type *tnumToType( int tnum );


   SourceFile *getSourceFile() { return stf_fil; }
   void setSourceFile( SourceFile *fil) { stf_fil = fil; }
   int getFNumber() { return stf_fnum; }
   void setFNumber( int n) { stf_fnum = n; }
   bool getLNumbersPrecedeFunctions() { return (stf_flags&STF_LNOS_PRECEDE_FUNCS);}
   void setLNumbersPrecedeFunctions() { stf_flags|=STF_LNOS_PRECEDE_FUNCS;}
   bool getHide() { return (stf_flags==STF_HIDE);}
   void setHide() { stf_flags=STF_HIDE;}
   const char *getName() { return stf_name;}
   void setName( const char *name ) { stf_name=name;}
   int getSymLimit() { return stf_symlim; }
   void setSymLimit( int limit ) { stf_symlim = limit; }
   int getSymNumber() { return stf_symno; }
   void setSymNumber( int symno ) { stf_symno = symno; }
   SymTab *getSymTab() { return stf_symtab; }
   void setSymTab( SymTab *symtab ) { stf_symtab = symtab; }
   STFile *idToHdrStf( taddr_t id );
   snlist_t *getSnList() { return stf_sn; }
   void setSnList( snlist_t *sn ) { stf_sn = sn; }
```

```
   language_t getLanguage() { return stf_language;}
   void setLanguage( language_t l ) { stf_language = l;}

   taddr_t getAddress() { return stf_addr;}
   void setAddress( taddr_t addr ) { stf_addr = addr;}

   int getMapSize() { return stf_mapsize;}

   Header **getMap() { return stf_fmap; }
   void setMap( Header **map, int mapsize );

 protected:

   //  Return the symbol number for the definition of type number tnum
   //  in file stf.
   //  Search the symbols between min and lim.
   //
   int getDef( int tnum, int min, int max, int *p_symno, const char **p_s );

   Type *getType( int tnum, int min, int max );

   Type *TypeDef( int *p_symno, const char **p_s, int tnum, int eval );
   var_t *Field( int *p_symno, const char **p_s, int is_struct, var_t *next, bool eval );

   const char                *stf_name;
   language_t                stf_language;
   symtab_t                  *stf_symtab;
   SourceFile                *stf_fil;
   int                       stf_symno;       // first symbol in this source file
   int                       stf_symlim;      // last symbol in this source file
   taddr_t                   stf_addr;
   unsigned                  stf_flags;
   snlist_t                  *stf_sn;         // quick symbol list
   ftype_t                   *stf_types;
   Header                    **stf_fmap;      // array of pointers to include files
   int                       stf_mapsize;     // number in stf_fmap array
   int                       stf_fnum;
};
typedef STFile stf_t;


// file headers
//   One of these for each file included in a source file.
struct Header
{
 public:
   Header *lookup( int id );

   STFile *getStf() { return hf_stf;}
   void setStf( STFile *stf ) { hf_stf = stf; }
   int getID() { return hf_id;}
   void setID( int id ) { hf_id = id; }
   Header *getNext() { return hf_next; }
   void setNext( Header *next ) { hf_next = next; }

 protected:
   STFile                    *hf_stf;
   int                       hf_id;
   Header                    *hf_next;
};
typedef Header hf_t;


//
// Shared Library object class
//
typedef class SharedLib
{
 public:

   //  Load the shared libraries desribed in shlibs, and point *p_stlist
   //  at the loaded list.  Return 0 for success, -1 for failure.
```

```
    //  In the normal case all the shared libraries will be in the
    //  cache, either from the preload or from a previous run of the
    //  target.
    //
    int loadSymTabs( target_info_t *target_info, SymTab **p_stlist );

    //  Get the list of shared libraries that the execfile will load
    //  when it is run.  We do this in the same way as ldd(1), by
    //  running the target with LD_TRACE_LOADED_OBJECTS set in its
    //  environment and parsing the output.
    //
    static int getPreloadShLibList( const char *execfile, SharedLib **p_shlibs );

    const char *getName() { return sh_name; }
    void setName( const char *name ) { sh_name = name; }

    taddr_t getAddress() { return sh_addr; }
    void setAddress( taddr_t addr ) { sh_addr = addr; }

    SharedLib *getNext() { return sh_next; }
    void setNext( class SharedLib *next) { sh_next = next; }

  protected:

    const char                  *sh_name;
    taddr_t                     sh_addr;
    class SharedLib             *sh_next;
} shlib_t;

long ci_typesize( lexinfo_t *lx, Type *type);

#endif
```

```
/*-----------------------------------------------------------------
 |       FILE: symtab.C
 |    PURPOSE: Symbol table object class methods.
  -----------------------------------------------------------------*/
   clude "symtab.h"

static int      Verbose;

// Static var initialization
SymTab * SymTab::st_main_symtab = NULL;
bool SymTab::st_table_loaded = FALSE;
SymTab * SymTab::st_symtab_cache_list = NULL;


//   Find the address of a global.  Return 0 if not found (this will happen
//   for an extern with no definition).
//
taddr_t
SymTab::lookupGlobalAddr( const char *name )
{
   taddr_t         addr;

   addr = st_addrlist.search( name );

   if( addr == 0 && this != st_main_symtab )
      addr = st_main_symtab->st_addrlist.search( name );

   return addr;
}


//  Load all symbol table information.  This is used for debugging.
//
void
ab::debugLoadSymbols( const char *name )
{
   if( *name != '\0' )
   {
      int             ambig;
      Function        *f;

      if( (f = nameToFunction( name, &ambig )) == NULL )
      {
         if (ambig)
            cout << name << " is ambiguous" << endl;
         else
            cout << "unknown function " << name << end;
      } else
         f->loadInfo( f, (taddr_t) 0, (char *) FALSE, (char *) UNSET );
   }
   else
   {
      for( SymTab *st = st_main_symtab; st != NULL; st = st->st_next )
      {
         cout << "Loading entire symbol table" << endl;

         st->st_functab->iterateOverFunction( Function::loadInfo,
                                              (char *) TRUE, (char *) UNSET );
         for( SourceFile *fil = st->st_sfiles; fil != NULL; fil = fil->getNext() )
            (void) fil->getVars();
         cout << "Entire symbol table loaded" << endl;

      }
   }


// Destructor
//  Close down a symbol table.  This is called if we rerun a
//  target and find that it uses a different shared library
//  from the last run (rare, but could happen if the
//  environment variable LD_LIBRARY_PATH is changed).
```

```
//
//
SymTab::~SymTab()
{
   // Delete any breakpoints from this symbol table
   removeBreakpoints();

   // Close any source files we have open.  Also delete edit blocks belonging
   // to the source files.
   SourceFile *next;
   for( SourceFile *fil = st_sfiles; fil != NULL; fil = next )
   {
      next = fil->getNext();
      delete fil;
   }

   // delete the Executable object
   delete st_exec;
}

//  Free any symbol tables in the cache.  This is called just
//  after startup of the target to flush any cached symbol
//  tables that weren't in fact used.
void
SymTab::freeCache()
{
   SymTab *next;

   for( SymTab *st = st_symtab_cache_list; st != NULL; st = next )
   {
      next = st->st_next;
      delete st;
   }
}


//  Look up the symbol table for name in the cache.  Remove
//  it from the cache and return it if found, otherwise
//  return NULL.
//
//  Name is the name of the executable or shared library file
//  that the symbol table came from.

SymTab *
SymTab::getSymTabFromCache( const char *name )
{
   symtab_t       *st, *prev;

   prev = NULL;
   for( st = st_symtab_cache_list; st != NULL; st = st->st_next )
   {
      if( strcmp(name, st->st_name) == 0 )
      {
         if (prev != NULL)
            prev->st_next = st->st_next;
         else
            st_symtab_cache_list = st->st_next;
         st->st_next = (SymTab *)NULL;
         return st;
      }
      prev = st;
   }
   return NULL;
}


//  Deal with a change in the text offset of the symbol table.  This may
//  be necessary when re-running the target as shared libraries may be
//  mapped at different addresses.  It's also necessary when we have
//  preloaded symbol tables with a nominal offset of zero.
//
//  We adjust the following:
```

```
//
//      function and line number addresses
//      symbol table address to text file offset
//      addresses of global variables

//  We don't change breakpoint addresses here - we do that by removing
//  and recreating all breakpoints just after starting the target
//
void
SymTab::changeTextOffset( taddr_t new_addr )
{
   long delta = new_addr - st_text_addr_offset;
   if (delta != 0)
   {
      st_addrlist.adjustAddrOffset( delta );

      st_exec->adjustAddrOffset( delta );
      st_functab->adjustTextAddrBase( delta );

      st_sfiles->adjustVarAddresses( delta );
      st_text_addr_offset = new_addr;
   }
}

//  Load the symbol tables for the shared libraries of an object.
//  This is called just after the target starts, because only
//  then have the shared libraries been loaded and mapped.

int
SymTab::loadSharedLibrary()
{
   static char     dynamic[] = "__DYNAMIC";

   shlib_t        *shlibs;
   symtab_t       *stlist;

   if( st_main_symtab == NULL || st_main_symtab->st_next != NULL)
      panic("shared lib botch");

   if( st_main_symtab->st_dynamic )
   {
      const char *sym0_name = st_main_symtab->st_exec->getSymString( 0 );
      if( strcmp(sym0_name, dynamic) != 0 )
      {
         cout << "First symbol in " << st_main_symtab->st_name << " is " <<
            sym0_name <<" (expected " << dynamic << ")" << endl;
         return -1;
      }

      nlist_t         nm;
      st_main_symtab->st_exec->getSym( 0, nm );
      taddr_t addr = nm.n_value;

      if( Verbose )
         fputs("Get shlib names and global addresses...", stderr);
      if( st_main_symtab->getSharedLibsAndGlobalAddresses( addr, &shlibs ) != 0 )
         return -1;
      if( Verbose )
         fputs("done\n", stderr);
      if( shlibs->loadSymTabs( st_main_symtab->st_target_info, &stlist ) != 0 )
         return -1;
      freeCache();
      st_main_symtab->st_next = stlist;
   }
   return 0;



//  Unload shared library symbol tables.  This is called when the
//  target dies.  We don't free the symbol tables at this point - we
//  just put them in the cache in anticipation of using them again
//  when the target is re-run.
```

```
void
SymTab::unloadSharedLibrary()
{
   SymTab *prev = NULL;
   SymTab *next;
   for( SymTab *st = st_main_symtab; st != NULL; st = next )
   {
      next = st->st_next;
      if( st->st_type == STT_SHLIB )
      {
         if( prev != NULL )
            prev->st_next = st->st_next;
         else
            st_main_symtab = st->st_next;
         st->st_next = st_symtab_cache_list;
         st_symtab_cache_list = st;
      }
      else
         prev = st;
   }
}


//  Get the main symbol table for execfile, via fd which should
//  be a file descriptor referring to execfile.  The file pointer
//  is assumed to be at the start of the file.
//
//  If execfile is a shared library, we try to guess which
//  shared libraries it will use and preload these into the
//  shared library cache.  This is to allow the user to set
//  breakpoints in shared library functions before the target
//  is started, and to avoid a long pause when the target is
//  started for the first time.
//
int
SymTab::getAndInstallSymTabs( const char *execfile, int fd )
{
   shlib_t        *shlibs;
   symtab_t       *stlist;
   symtab_t       *st;
   struct stat     stbuf;
   target_info_t  *target_info;

   if( fstat(fd, &stbuf) != 0)
      panic("fstat failed");

   target_info =( target_info_t *) malloc( sizeof(target_info_t) );
   target_info->ti_name = execfile;
   target_info->ti_mod_time = stbuf.st_mtime;
   target_info->ti_highlighted_fil = NULL;

   st = new SymTab( execfile, STT_MAIN, fd,( taddr_t) 0, target_info );

   st->st_next = NULL;
   st_main_symtab = st;


   if( st->st_dynamic)
   {
      SharedLib::getPreloadShLibList( execfile, &shlibs );
      if( shlibs->loadSymTabs( target_info, &stlist ) != 0)
         return -1;
      st_symtab_cache_list = stlist;
   }

   return 0;
}


//  Find the sourcefile object where the global variable called name is defined.
//
```

```
//  If we find the file, point *p_fil at it and set *p_v to the var,
//  otherwise set both *p_fil and *p_var to NULL.
//
void
SymTab::findFileOfGlobal( const char *name, SourceFile **p_fil, Variable **p_v )
{
   static var_t   *globals = NULL;
   taddr_t         addr;
   Variable *v;

   for( SymTab *st = st_main_symtab; st != NULL; st = st->st_next )
   {
      for( SourceFile *fil = st->st_sfiles; fil != NULL; fil = fil->getNext() )
      {
         if( (v = fil->findVar( name )) != NULL )
         {
            *p_fil = fil;
            *p_v = v;
            return;
         }
      }
   }

   for( v = globals; v != NULL; v = v->getNext() )
      if( strcmp(v->getName(), name) == 0)
         break;

   if( v == NULL && (addr = st_main_symtab->lookupGlobalAddr( name )) != 0 )
   {
      v = new Variable( strsave(name), CL_EXT,
                        Type::codeToType( TY_INT_ASSUMED ), addr );
      v->setLanguage( LANG_UNKNOWN );
      v->setNext( globals );
      globals = v;
   }
   *p_fil = NULL;
   *p_v = v;
}



/*-------------------------------------------------------------------------
 |     METHOD: sortFunctionList
 |
 |    PURPOSE: An insertion sort for FunctionLists.
 |      NOTES: Should be a merge sort, but time is limited.  Maybe later.
 -------------------------------------------------------------------------*/
typedef int (*compareFunction)( Function *, Function * );

void sortFunctionList( FunctionList *oldlist, compareFunction func  )
{
   FunctionList newlist;
   ListIter(FuncPtr) olditer;
   ListIter(FuncPtr) newiter;

   olditer = *oldlist;

   newlist.listAppend( *olditer );
   newiter = newlist;

   for( olditer++; olditer.iterMore(); olditer++ )
   {
      // insert *olditer into newlist
      newiter = newlist;
      for( int done=FALSE; newiter.iterMore(); newiter++ )
      {
         if( func( *newiter, *olditer ) >= 0 )
         {
            done = TRUE;
            newiter.iterInsertBefore( *olditer );
            break;
         }
```

```
      }
      if( !done )
         newlist.listAppend( *olditer );
   }
   *oldlist = newlist;
}



//  Comparison function for the sort used in name_to_func below.
//  Primary key is function name, secondary is whether the
//  function has symbol table information( functions with symbol
//  table information come first).

static int flcmp( Function *f1, Function *f2 )
{
   int len1 = strlen(f1->getName());
   int len2 = strlen(f2->getName());

   if( len1 == len2)
      return Function::addrCompare( (void *)f1, (void *)f2 );
   else
      return( len1 > len2) ? 1 : -1;
}

//  Return a pointer to the function object for function name,
//  or NULL if there is no such function or name is ambiguous.
//  We allow matches where name is the first part of the full
//  function name if there is only one such match (if there is
//  more than one we return NULL and set *p_ambig to TRUE).
//
//  We search all the symbol tables including those in the cache.

Function *
SymTab::nameToFunction( const char *name, int *p_ambig )
{
   Function *f;

   FunctionList flhead;
   SymTab *st;

   for( st = st_main_symtab; st != NULL; st = st->st_next )
      st->st_functab->nameToFunctionList( name, &flhead );

   for( st = st_symtab_cache_list; st != NULL; st = st->st_next )
      st->st_functab->nameToFunctionList( name, &flhead );

   if( flhead.listEmpty() )
   {
      *p_ambig = FALSE;
      return NULL;
   }
   sortFunctionList( &flhead, flcmp );

   ListIter(FuncPtr) iter;
   iter = flhead;

   // How many unique matches do we have?
   int nmatches = 1;
   if( strlen((*iter)->getName()) != strlen(name) )
   {

      // walk through the list counting number of elements
      // with different addresses
      f = *iter;
      for( iter++; iter.iterMore(); iter++ )
      {
         if( (*iter)->getAddress() != f->getAddress() )
            nmatches++;
         f = *iter;
      }
```

```
    }
    if( *p_ambig = nmatches > 1) // note '='
       return NULL;
    iter = flhead;
    f = *iter;

    return f;
}

//  Return a pointer to the Function object of the function wherein
//  lies address addr, or NULL if addr does not lie within any
//  known function. Search all the symbol tables.

Function *
SymTab::addrToFunc( taddr_t addr )
{
    func_t          *f;

    for( SymTab *st = SymTab::st_main_symtab; st != NULL; st = st->st_next)
       if( (f = st->st_functab->addrToFunc( addr )) != NULL )
          return f;

    return NULL;
}

//  Call( *func)() for each source file with the source file as an argument.
//
//  Do the source files in the cache as well.

void
SymTab::iterateOverSourceFiles( void (*func)(SourceFile *fil) )
{
    SymTab          *st;
    SourceFile      *fil;

    for( st = st_main_symtab; st != NULL; st = st->st_next )
       for( fil = st->st_sfiles; fil != NULL; fil = fil->getNext() )
          if( fil->getStf()->getHide() == 0 )
            ( *func)( fil );

    for( st = st_symtab_cache_list; st != NULL; st = st->st_next )
       for( fil = st->st_sfiles; fil != NULL; fil = fil->getNext() )
          if( fil->getStf()->getHide() == 0 )
            ( *func)( fil );
}


//  Do a prescan of the symbol table of execfile, which is assumed to
//  be an a.out file which has been checked by check_execfile.
//
//  Fd is a read only file descriptor referring to execfile, and a is
//  a pointer to its exec header.
//
SymTab::SymTab(const char *execfile,
               symtab_type_t symtab_type,
               int fd,
               taddr_t text_addr_offset,
               target_info_t *ti )
{
    struct exec     hdr;

    if( read(fd,( char *) &hdr, sizeof(hdr)) != sizeof(hdr) )
    {
       cout << "read error in " << execfile << endl;
       panic( "in SymTab::Symtab()" );
    }

    st_name = strsave( execfile );
    st_type = symtab_type;
    st_target_info = ti;
    st_dynamic = hdr.a_dynamic;
```

```
   taddr_t mem_text_offset =( taddr_t) N_TXTADDR(hdr);
   off_t file_text_offset = N_TXTOFF(hdr);
   taddr_t text_size = hdr.a_text;
   off_t file_syms_offset = N_SYMOFF(hdr);
   int nsyms = (int)hdr.a_syms / SYMSIZE;
   off_t file_strings_offset = N_STROFF(hdr);

   long addr_to_fpos_offset = text_addr_offset - file_text_offset;
   if( symtab_type == STT_MAIN)
      addr_to_fpos_offset += mem_text_offset;

   st_text_addr_offset = text_addr_offset;

   st_exec = new Executable( execfile, fd, file_syms_offset, nsyms,
                   file_strings_offset, addr_to_fpos_offset );

   skim(text_addr_offset,
        text_addr_offset + mem_text_offset + text_size,
        (Block *)NULL,
        &st_sfiles,
        &st_functab );
}



static int
compFileNames( const char *f1, const char *f2 )
{
   if( !strcmp( f1, f2 ) )
      return 0;

   if( *f1!='/' && *f2!='/' )
      return 1;

   // try comparing that last components of the name
   const char *p1 = f1 + strlen( f1 );
   for( p1 = f1 + strlen(f1); p1!=f1 && *(p1-1) != '/'; p1-- );
   const char *p2 = f2 + strlen( f2 );
   for( p2 = f2 + strlen(f2); p2!=f2 && *(p2-1) != '/'; p2-- );

   return strcmp( p1, p2 );
}



// Given a filename and a line number, this method returns the line number of
// the closest statement.  It also returns the function object and address
// of the returned line number.  If unsuccessful, it returns -1.
int
SymTab::lNumToFuncAndAddr( const char *file, int lnum, Function **p_func, taddr_t *p_addr )
{
   SymTab *st;
   SourceFile *fil;
   bool Found = FALSE;

   *p_func = (Function *)NULL;
   *p_addr = (taddr_t)NULL;

   // get the SourceFile

   for( st = st_main_symtab; st != NULL && !Found; st = st->st_next )
      for( fil = st->st_sfiles; fil != NULL; fil = fil->getNext() )
      {
         if( !compFileNames( fil->getName(), file ) )
         {
            Found = TRUE;
            break;
         }
      }


   if( !Found )
```

```
      for( st = st_symtab_cache_list; st != NULL && !Found; st = st->st_next )
         for( fil = st->st_sfiles; fil != NULL; fil = fil->getNext() )
         {
            if( !compFileNames( fil->getName(), file ) )
            {
               Found = TRUE;
               break;
            }
         }

   // if we could not find the sourcefile, then we're out of luck
   if( !Found )
   {
      cout << "SymTab::LNumToFunc() could not find sourcefile " << file << endl;
      return -1;
   }

   // Now find which function this line falls into.
   // Note that the functions are sorted by line number (and thus address).

   Function *f;
   ListIter(FuncPtr) iterator;
   iterator = *fil->getFunctionList();
   for( Found=FALSE; iterator.iterMore(); iterator++ )
   {
      f = *iterator;
      if( f->getMaxLNumber() > lnum )
      {
         Found = TRUE;
         break;
      }
   }

   taddr_t addr;

   // does line number fall within this function?
   //cout << "Searching function " << f->getName() << endl;

   int closest_lnum;
   addr = f->lNumberToClosestAddr( lnum, &closest_lnum );

   if( addr!=(taddr_t)0 && addr!=(taddr_t)1 )
   {
      *p_addr = addr;
      *p_func = f;
      return closest_lnum;
   }


   return -1;
}



#define SYMSET_SIZE     256

//  Build the map of interesting symbols for skimming.
//
static void
build_symset( char *symset )
{
        static int wanted_syms[] = {
                N_SO, N_SOL, N_BCOMM, N_STSYM, N_GSYM, N_LCSYM, N_FUN,
                N_TEXT, N_TEXT | N_EXT, N_BSS | N_EXT, N_DATA | N_EXT,
                N_BINCL, N_EXCL, N_EINCL,
        };
        int i;

        memset(symset, '\0', SYMSET_SIZE);
        for (i = 0; i < sizeof(wanted_syms) / sizeof(wanted_syms[0]); ++i)
                symset[wanted_syms[i]] = TRUE;
}
```

```
//  Do a prescan of a symbol table.  Lazily evaluated since most symbols
//  are never touched in most debugger runs.
//
void
SymTab::skim(taddr_t first_addr,
             taddr_t last_addr,
             Block *rootblock,
             SourceFile **p_sfiles,              // return
             FunctionTab **p_functab_id )         // return
{
   static char symset[SYMSET_SIZE];

   ListIter(FuncPtr) fl_iterator;

   snlist_t *sn;
   funclist_t *fl;
   const char *name, *cptr;
   int nsyms;

   // which symbols do we want?
   if( !st_table_loaded )
   {
      build_symset(symset);
      st_table_loaded = TRUE;
   }

   STFile *stf = NULL;
   Function *flist = NULL;
   Function *f = NULL;
   int flist_len = 0;
   SourceFile *sfiles = NULL;
   Header *headers=NULL, *hf=NULL;
   FunctionTab *functab;

   // allocate array of header pointers (fmap)
   int mapsize = 0;
   int max_mapsize = 32;
   Header **fmap = (Header **)malloc( max_mapsize * sizeof(Header *) );

   // allocate another array of header pointers (istack)
   int isp = 0;
   int istack_size = 8;
   Header **istack = (Header **)malloc( istack_size * sizeof(Header *) );

   int symno = -1;
   nsyms = st_exec->getNSyms();

   const char *path_hint = NULL;
   int seen_func = 0;
   nlist_t nm;
   while( (symno = st_exec->findSym( symno+1, nm, symset)) != nsyms )
   {
      switch(nm.n_type)
      {
       case N_SO:
         // N_SO marks the beginning of a new sourcefile.
         // So close down the current sourcefile and create a new one.

         // If processing a function symbol, then scarf away the number of
         // the last symbol for the function (symno).

         if( f != NULL )
         {
            f->setSymLimit( symno );
            f = NULL;
         }


         if( stf != NULL )
```

```
    {
       stf->setMap( fmap, mapsize );
       stf->setSymLimit( symno );
    }

    // get the name of the file OR the path hint of where to find the file
    name = st_exec->getSymString( symno );

    //  4.0 cc puts paths ending in '/' just before
    //  the source files that follow.

    if( name[strlen(name) - 1] == '/' )
    {
       path_hint = name;
       break;
    }

    stf = new STFile( name, this, symno, SourceFile::srcType( name ), nm.n_value );
    sfiles = new SourceFile( stf, rootblock, path_hint, sfiles );
    stf->setSourceFile( sfiles );
    if (isp != 0)
       panic("unmatched N_BINCL");
    stf->setFNumber( 0 );

    // allocate and initialize header
    fmap[0] = (Header *)malloc( sizeof( Header ) );
    fmap[0]->setStf( stf );
    fmap[0]->setID( -1 );
    fmap[0]->setNext( NULL );
    mapsize = 1;
    path_hint = NULL;
    seen_func = FALSE;
    symset[N_SLINE] = TRUE;
    break;
  case N_SLINE:
    if( !seen_func && stf != NULL )
       stf->setLNumbersPrecedeFunctions();
    symset[N_SLINE] = FALSE;
    break;
  case N_SOL:
    //  Basic support for #line construct.
    //
    if( stf != NULL )
    {
       stf->setName( st_exec->getSymString( symno ) );
       sfiles->setName( stf->getName() );
    }
    break;

  case N_BINCL:
  case N_EXCL:
    if (sfiles == NULL)
       panic("header outside source file in st_skim");
    if( mapsize == max_mapsize )
    {
       max_mapsize *= 2;
       fmap = (Header **)realloc((char *)fmap,
                                 max_mapsize * sizeof(Header *));
    }


    if( nm.n_type == N_EXCL )
    {
       // N_EXCL means we've seen this header file before so
       // its excluded here.  Look it up using n_value.

       hf = headers->lookup( (int)nm.n_value );
       fmap[mapsize++] = hf;
       break;
    }
    if( isp == istack_size )
    {
```

```
      istack_size *= 2;
      istack = (Header **)realloc((char *)istack,
                                  istack_size * sizeof(Header *));
    }
    istack[isp] = (Header *)malloc( sizeof(Header) );
    istack[isp]->setNext( headers );
    headers = istack[isp++];
    headers->setStf( new STFile( st_exec->getSymString( symno ),
                                 this, symno, sfiles->getLanguage(),
                                 (taddr_t)nm.n_value ) );

    STFile *tmp_stf = headers->getStf();
    if( nm.n_type == N_EXCL )
      tmp_stf->setSourceFile( (SourceFile *)NULL );
    else
    {
      SourceFile *newSF = new SourceFile(tmp_stf, (Block *)NULL,
                                         (const char *)NULL,
                                         (SourceFile *)NULL );
      tmp_stf->setSourceFile( newSF );
    }

    tmp_stf->setFNumber( mapsize );
    headers->setID( (int)nm.n_value );
    fmap[ mapsize++ ] = headers;
    break;
  case N_EINCL:
    if( isp == 0)
      panic("unmatched N_EINCL");
    istack[--isp]->getStf()->setMap( fmap, mapsize);
    istack[isp]->getStf()->setSymLimit( symno );
    break;

  case N_STSYM:
  case N_GSYM:
  case N_LCSYM:
    name = st_exec->getSymString( symno );
    sn = (snlist_t *)malloc( sizeof(snlist_t) );
    sn->sn_symno = symno;
    sn->sn_name = parseName( &name );
    stf->addToSnList( sn );
    break;

  case N_FUN:
    name = st_exec->getSymString( symno );

    //  Some compilers (e.g. gcc) put read only strings
    //  in the text segment, and generate N_FUN symbols
    //  for them.  We're not interested in these here.

    if( (cptr = strchr(name, ':')) == NULL ||
        (cptr[1] != 'F' && cptr[1] != 'f') )
      break;

    if (f != NULL )
    {
      f->setSymLimit( symno );
      f = NULL;
    }

    name = save_fname( name, FALSE );

    // create a new function object and link it into the
    // sourcefile object's function list
    flist = new Function( name, symno, nm.n_value,
                          this, sfiles, flist );
    ++flist_len;

    // link this function into sfiles's function list
    fl = sfiles->getFunctionList();
    fl->listAppend( flist );
```

```
      // tell the Function which list it's stored in
      flist->setFunctionList( fl );

      f = flist;
      seen_func = TRUE;
      break;
    case N_TEXT:
    case N_TEXT | N_EXT:
      name = st_exec->getSymString( symno );

      //  Some compilers put N_TEXT symbols out with object
      //  file names, often with the same addresses as real
      //  functions.  We don't want these.
      //
      //  We also don't want N_TEXT symbols which immediately
      //  follow an N_FUN symbol with the same address.

      if( nm.n_type == N_TEXT )
      {
         if( *name != '_' )
         {
            //  A .o file symbol is definitely
            // the end of the current function,
            //  ifa ny.

            if( f != NULL )
            {
               f->setSymLimit( symno );
               f = NULL;
            }
            break;
         }
         if( f != NULL && (f->getAddress() == nm.n_value) )
            break;
      }

      //  You'd expect that we'd close down the current
      //  function here, but some C compilers put out
      //  N_TEXT symbols in the middle of the N_SLINE
      //  symbols for a function.
      //
      //  Thus we leave f alone, and rely on an N_SO or
      //  a .o file N_TEXT to terminate the current
      //  function.
      //
      name = save_fname( name, TRUE );
      flist = new Function( name, symno, (taddr_t)nm.n_value,
                            this, (SourceFile *)NULL,
                            flist );

      flist->setFlags( FU_NOSYM | FU_DONE_BLOCKS | FU_DONE_LNOS );

      ++flist_len;
      break;

    case N_BSS | N_EXT:
    case N_DATA | N_EXT:
      name = st_exec->getSymString( symno );
      if (*name != '_')
         break;
      ++name;
      st_addrlist.insert( name, st_text_addr_offset + (taddr_t)nm.n_value );
      break;
    default:
      panic("unexpected symbol in SymTab::skmi()");
   }
}

if( f != NULL )
{
   f->setSymLimit( symno );
```

```
      f = NULL;
   }
   if( stf != NULL )
   {
      stf->setMap( fmap, mapsize );
      stf->setSymLimit( symno );
   }

   free((char *)fmap);
   free((char *)istack);

   // construct the function table
   functab = new FunctionTab( this, flist, flist_len,
                              first_addr, last_addr );

   // return function table and source files
   *p_sfiles = sfiles;
   *p_functab_id = functab;
}
```

```
/*-------------------------------------------------------------------
 |       FILE: symtab_misc.C
 |    PURPOSE: Methods for various object classes internal to the
 |             symbol table object class.
 |
 |   CONTENTS: -classes-        ------------purpose-----------------
 |             STFile           internal source file object
 |             Header           manages list of include files
 |             Block            manages lexical info concerning a file
 |             AddrList         list of global vars and their addrs
 -------------------------------------------------------------------*/
#include "symtab.h"

/*-------------------------------------------------------------------
 |      CLASS: STFile
 |
 |    PURPOSE: Source file object internal to symbol table.
 |
 -------------------------------------------------------------------*/


/*-------------------------------------------------------------------
 |     METHOD: STFile
 |
 |    PURPOSE: Constructor
 |
 -------------------------------------------------------------------*/
STFile::STFile(const char *name,
               symtab_t *st,
               int symno,
               language_t language,
               taddr_t addr )
{
   stf_symtab = st;
   stf_symno = symno;
   stf_name = name;
   stf_language = language;
   stf_addr = addr;
   stf_flags = 0;
   stf_sn = NULL;
   stf_types = NULL;
}



/*-------------------------------------------------------------------
 |     METHOD: setMap
 |
 |    PURPOSE: called when we have collected all the information
 |             concerning an STFile.  Creates space for the header map
 |             and copies the map into this space.
 -------------------------------------------------------------------*/
void
STFile::setMap( Header **fmap, int mapsize )
{
   // allocate array of Header pointers
   if (mapsize == 0)
      panic("mapsize 0 in STFile::setMap");
   stf_mapsize = mapsize;
   stf_fmap = (Header **)malloc( mapsize * sizeof(Header *) );

   // copy fmap into this newly allocated array of pointers
   memcpy((char *)stf_fmap, (char *)fmap, mapsize * sizeof(Header *));
}


/*-------------------------------------------------------------------
 |     METHOD:tnumToType
 |
 |    PURPOSE: Given the tnum of a type appearing in this file, this
 |             method returns the corresponding type or NULL.
 |
```

```
 ------------------------------------------------------------------------*/
type_t *
STFile::tnumToType( int tnum )
{
   type_t *type;
   int deb=(tnum==8);

   ftype_t *last = NULL;
   for (ftype_t *ft = stf_types; ft != NULL; ft = ft->ft_next)
   {
      if( ft->ft_tnum > tnum)
         break;
      last = ft;
   }
   if (last != NULL && last->ft_tnum == tnum)
      type = last->ft_type;
   else
      type = getType( tnum, stf_symno, stf_symlim-1 );

   return type;
}



/*-------------------------------------------------------------------------
 |     METHOD: getType
 |
 |    PURPOSE: Get the type, tnum.  Limit the search for the tnum type
 |             to the symbols between sym numbers min through max.
 |             Return NULL if not found.
 ------------------------------------------------------------------------*/
type_t
*STFile::getType( int tnum, int min, int max )
{
   const char *s;
   int symno;

   if( getDef( tnum, min, max, &symno, &s ) == -1 )
      return NULL;
   return TypeId( this, &symno, &s, TRUE );
}


/*-------------------------------------------------------------------------
 |     METHOD: lookupTnum
 |
 |    PURPOSE: Search all types of the file for the type identified by
 |             tnum.  Return the type or NULL if not found.
 ------------------------------------------------------------------------*/
type_t *
STFile::lookupTnum( int tnum )
{
   ftype_t *ft;

   for( ft = stf_types; ft != NULL; ft = ft->ft_next )
      if( ft->ft_tnum == tnum )
         return ft->ft_type;
   return NULL;
}



/*-------------------------------------------------------------------------
 |     METHOD: addType
 |
 |    PURPOSE: Add type identified by tnum to the list of types from this
 |             file.
 ------------------------------------------------------------------------*/
ftype_t *
STFile::addType( int tnum )
{
   ftype_t *last, *ft;
```

```
   last = ft = NULL;
   for( ft = stf_types; ft != NULL; ft = ft->ft_next )
   {
      if( ft->ft_tnum > tnum )
         break;
      last = ft;
   }
   if (last != NULL && last->ft_tnum == tnum)
      panic("type already exists in STFile::insertFType");

   ftype_t *newft = new ftype_t; //(ftype_t *)malloc( sizeof(ftype_t) );
   newft->ft_next = NULL;
   newft->ft_tnum = tnum;
   newft->ft_type = new Type( TY_NOTYPE );

   if (last == NULL)
      stf_types = newft;
   else
      last->ft_next = newft;

   newft->ft_next = ft;
   return newft;
}



/*-------------------------------------------------------------------------
 |     METHOD: getDef
 |
 |    PURPOSE: Return the symbol number for the definition of type
 |             number, tnum in the file.  Limit the search between
 |             symbols min through max.
 -------------------------------------------------------------------------*/

...ile::getDef( int tnum, int min, int max, int *p_symno, const char **p_s )
{
   nlist_t nm;
   const char *s, *base, *pos;
   int symno, match;
   int f, t;

   Executable *exec = stf_symtab->getExecutable();

   for (symno = min; symno <= max; symno++)
   {

      exec->getSym( symno, nm );
      base = s = exec->getSymString( symno );
      if (base == NULL)
         continue;
      while( (pos = strchr(s, '=')) != NULL )
      {
         //  Definitions of the form "=x" are unresolved tag
         //  definitions.  We don't want these - we want the
         //  real definition.
         //
         if( pos[1] == 'x' )
         {
            s = pos + 1;
            continue;
         }

         if( pos[-1] == ')' )
         {
            while( pos >= base && *pos != '(' )
               --pos;
            *p_s = pos;
            Typenum(&pos, &f, &t);
            scheck(&pos, '=');
            match = f == stf_fnum && t == tnum && *pos != 'x';
         }
```

```
         else
         {
            while (pos >= base && isdigit(pos[-1]))
               --pos;
            *p_s = pos;
            match = parseNumber(&pos) == tnum;
         }
         if (match)
         {
            *p_symno = symno;
            return 0;
         }
         s = pos + 1;
      }
   }
   return -1;
}


/*-----------------------------------------------------------------------
 |     METHOD: idToHdrStf
 |
 |    PURPOSE: Given the id for a header file, return the header's
 |             STFile or NULL if not found.
 -----------------------------------------------------------------------*/
stf_t *
STFile::idToHdrStf( taddr_t id )
{
   Header **p_hf, **maxhf;
   stf_t *hdrstf;

   maxhf = stf_fmap + stf_mapsize;
   hdrstf = NULL;
   for( p_hf = stf_fmap; p_hf < maxhf; ++p_hf )
   {
      if( (*p_hf)->getID() == id )
      {
         hdrstf = (*p_hf)->getStf();
         break;
      }
   }

   if( hdrstf == NULL )
      panic("hdr botch in STFile::idToHdrStf");

   return hdrstf;
}


/*-----------------------------------------------------------------------
 |      CLASS: Header
 |
 |    PURPOSE: Manages the list of header files included in a source file.
 -----------------------------------------------------------------------*/

/*-----------------------------------------------------------------------
 |     METHOD: lookup
 |
 |    PURPOSE: Lookup the header file entry given it's id.  If id not
 |             found, then we panic.  This method is used when we find
 |             a N_EXCL symbol, meaning a header file is excluded
 |             because it has already been encountered.
 -----------------------------------------------------------------------*/
Header *
Header::lookup( int id )
{
   Header *hf;

   for( hf = this; hf != NULL; hf = hf->hf_next )
      if( hf->hf_id == id )
         return hf;
```

```
   panic("id not found in Header::lookup()");
   return NULL;
}


//  Return a pointer to a munged saved copy of fname.  Ext is true
//  if name is an external symbol generated by the linker.  These
//  symbols are expected to have an underscore prepended - if it
//  is there it is stripped, otherwise the returned name is
//  enclosed in brackets (e.g. "[name]").
//
const char *
save_fname( const char *name, int ext )
{
   const char *cptr;
   char *ptr;
   int len;

   if (ext && *name != '_')
   {
      ptr = malloc( strlen(name) + 3 );
      (void) sprintf(ptr, "[%s]", name);
   }
   else
   {
      if (ext)
         name++;
      for (cptr = name; *cptr != '\0' && *cptr != ':'; cptr++)
         ;
      len = cptr - name;
      ptr = malloc( len + 1 );
      memcpy( ptr, name, len );
      ptr[len] = '\0';
   }
   return ptr;


/*-------------------------------------------------------------------------
  |      CLASS: Block
  |    PURPOSE: The information for a block consists of the start/end
  |             line numbers for the block, a pointer to the list of
  |             variables declared in the block, a pointer to the next
  |             block at the same level as this one, and a pointer to a
  |             list of blocks declared in this one, and a list of types
  |             declared in this block.  Finally a pointer to the parent
  |             block.
  -------------------------------------------------------------------------*/
Block::Block( Block * parent )
{
   bl_start_lnum = 0;
   bl_vars = NULL;
   bl_typedefs = NULL;
   bl_aggr_or_enum_defs = NULL;
   bl_initlist = NULL;
   bl_parent = parent;
   bl_blocks = NULL;
   bl_next = NULL;
}


void
Block::pushTypedefsAndAggrs( Block *bl )
{
   if( bl_aggr_or_enum_defs != NULL )
   {
      aggr_or_enum_def_t *ae;

      ae = (aggr_or_enum_def_t *)malloc( sizeof(aggr_or_enum_def_t) );
      ae->ae_type = NULL;
      ae->ae_sublist = bl_aggr_or_enum_defs;
      ae->ae_next = bl->bl_aggr_or_enum_defs;
```

```
      bl->bl_aggr_or_enum_defs = ae;
   }

   if( bl_typedefs != NULL )
   {
      typedef_t *td;

      td = (typedef_t *)malloc( sizeof(typedef_t) );
      td->td_type = NULL;
      td->td_sublist = bl_typedefs;
      td->td_next = bl->bl_typedefs;
      bl->bl_typedefs = td;
   }
}


void
Block::iterateOverVars( void (*func)(Variable *v, char *c_args), char *args )
{
   if( !this )
      return;

   block_t *bl;
   var_t *v;

   for( bl = bl_blocks; bl != NULL; bl = bl->bl_next )
      bl->iterateOverVars( func, args );
   for( v = getVars(); v != NULL; v = v->getNext() )
      (*func)(v, args);
}


aggr_or_enum_def_t *
aggr_or_enum_defst::matchTag( aggr_or_enum_def_t *ae, const char *tag )
{
   if( ae->ae_is_complete == AE_COMPLETE && ae->ae_tag != NULL &&
       strcmp(ae->ae_tag, tag) == 0 )
      return ae;

   return NULL;
}

aggr_or_enum_def_t *
aggr_or_enum_defst::fixUndefAggr( aggr_or_enum_def_t *uae, const char *arg )
{
   aggr_or_enum_def_t *aelist;
   type_t *utype, *type;
   aggr_or_enum_def_t *ae;

   aelist = (aggr_or_enum_def_t *)arg;

   switch( uae->ae_type->getTypeCode() )
   {
    case TY_U_STRUCT:
    case TY_U_UNION:
    case TY_U_ENUM:
      if (uae->ae_tag == NULL)
         ae = NULL;
      else
         ae = aelist->applyToAelist( aggr_or_enum_defst::matchTag, uae->ae_tag );
      break;
    default:
      ae = NULL;
      break;
   }

   if (ae == NULL)
      return NULL;

   utype = uae->ae_type;
   type = ae->ae_type;
```

```
    utype->setTypeCode( type->ty_code );
    utype->setAggrOrEnumDef( ae );

    if(utype->getTypeDef() != NULL && type->getTypeDef() == NULL)
    {
        type->setTypeDef( utype->getTypeDef() );
        type->getTypeDef()->setType( type );
    }

    return NULL;
}




/*-------------------------------------------------------------------------
 |      CLASS: AddrList
 |
 |    PURPOSE: A list of global variable names and their addresses.
 -------------------------------------------------------------------------*/

//  Insert a new entry at the front of the globals address map
//
void
AddrList::insert( const char *name, taddr_t addr )
{
    AddrNode *anode = new AddrNode( name, addr );

    al_list.listCons( anode );
}


    i
  rList::adjustAddrOffset( long delta )
{
    ListIter(addrptr) iterator;
    for( iterator=al_list; iterator.iterMore(); iterator++ )
    {
        AddrNode *anode = *iterator;
        anode->an_addr += delta;
    }
}



//  Look up the address of global variable name.  Return the
//  address, or 0 if name is not found.
//
taddr_t
AddrList::search( const char *name )
{
    ListIter(addrptr) iterator;

    for( iterator=al_list; iterator.iterMore(); iterator++ )
    {
        AddrNode *anode = *iterator;
        if( strcmp( anode->an_name, name ) == 0 )
            return anode->an_addr;
    }

    return 0;
}
```

```
/*-------------------------------------------------------------------
 |       FILE: symtab_parse.C
 |    PURPOSE: This file contains C code which parses the symbol table
 |             Parts of this code is taken from various places
 |             (gdb and dbx) and hacked together until it seems to
 |             work pretty well.
 -------------------------------------------------------------------*/
#include <ctype.h>
#include <a.out.h>
#include <stab.h>
#include "symtab.h"

static type_t  *
TypeDef( stf_t *stf, int *p_symno, const char **p_s, int tnum, int eval );

static dim_t *
Subrange( STFile *stf, int *p_symno, const char **p_s, bool eval, bool want_subrange_type );

static aggr_or_enum_def_t *
EnumList( SymTab *st, int *p_symno, const char **p_s, type_t *type );


// The address of this variable is used as a special value for ty_typedef
// to signal that it is a basic type not a user defined typedef.
//
static typedef_t Is_basic_type;

// If **p_s is ch, move *p_s on, otherwise abort.  Used for checking
// and swallowing required terminals in the grammar.
void
scheck( const char **p_s, int ch )
{
   if (**p_s != ch)
      panic("syntax error in symtab");
   ++*p_s;
}

//  Oops - we encountered a construct which we haven't implemented in the
//  grammar.
//
static void
notImplemented( const char *s )
{
   cout << s << " not implemented" << endl;
   panic("Exit from dbx symbol table parser.");
}

#if 0
//  Class:
//      'c' = Constant ';'      { NI }
//      Variable
//      Procedure               { PI }
//      Parameter
//      NamedType
//      'X' ExportInfo          { NI }
//
//   All the ones that we implement resolve to a type letter giving the
//   class of the variable followed by a TypeId.
//
*/
#endif
Type *
Class( stf_t *stf, int *p_symno, const char **p_s, class_t *p_class )
{

   class_t         class_type;       // enumeration type
   Type            *rtype;

   const char *s = *p_s;
   switch( *s )
   {
      // 'c' '=' Constant
```

```
case 'c':
   notImplemented("constants");
   class_type = CL_NOCLASS;       /* to satisfy gcc */
break;

// Variable
case '(':
case '0':
case '1':
case '2':
case '3':
case '4':
case '5':
case '6':
case '7':
case '8':
case '9':
class_type = CL_AUTO;
   --s;
break;
case 'r':
class_type = CL_REG;
break;
case 'S':
class_type = CL_STAT;
break;
case 'V':
class_type = CL_LSTAT;
break;
case 'G':
class_type = CL_EXT;
break;

// Procedure
//
//  DEV: 'P' should be in this group according to the grammar see the
//       comment later in this function.
case 'Q':
case 'I':
   notImplemented("Q/I functions");
   class_type = CL_NOCLASS;       /* to satisfy gcc */
break;
case 'F':
class_type = CL_FUNC;
break;
case 'f':
class_type = CL_LFUNC;
break;
case 'J':
   notImplemented("internal functions");
   class_type = CL_NOCLASS;       /* to satisfy gcc */
break;

// Parameter
//
//   DEV: the grammar says 'P' introduces a global procedure, but gcc uses
//        it for parameters, so we just silently take it to mean a parameter.
case 'P':
case 'p':
   class_type = CL_ARG;
break;
case 'v':
   class_type = CL_REF;
break;

// Namedtype
case 't':
   class_type = CL_TYPEDEF;
break;
case 'T':
   class_type = CL_TAGNAME;
break;
```

```
        // 'X' ExportInfo
        case 'X':
          // ignore these
           class_type = CL_NOCLASS;
        break;

        default:
           panic("unknown type class letter in symbol table");
           class_type = CL_NOCLASS;       /* to satisfy gcc */
    }

    s++;
    rtype = TypeId(stf, p_symno, &s, TRUE);
    *p_s = s;

    //
    // The basic types (int, char etc) get entries in the symbol table that
    // look like user typedefs.  We don't want to clutter up the typedef lists
    // with these entries.
    //
    if (class_type == CL_TYPEDEF && rtype->ty_typedef == NULL)
       *p_class = CL_NOCLASS;
    else
      *p_class = class_type;

    return rtype;
}

//
//  Parse a number (INTEGER in the grammar).  No sign allowed.
//
int
parseNumber( const char **p_s )
{
    int res = 0;
    const char *s = *p_s;
    if (!isdigit(*s))
       panic("bad number in parseNumber");
    while (*s != '\0' && isdigit(*s))
       res = res * 10 + *s++ - '0';
    *p_s = s;
    return res;
}

//
//  Parse a possibly signed number.
//
static int
parseSignedNumber( const char **p_s )
{
    int neg;
    const char *s = *p_s;
    if (neg = *s == '-')
       s++;
    int num = parseNumber(&s);
    *p_s = s;
    return neg ? -num : num;
}

//
//  Parse a name (NAME in the grammar).
//  Return a pointer to a saved copy of the name.
//
c st char *
   seName( const char **p_s )
{
    const char *s = *p_s;
    if (!isalpha(*s) && *s != '_')
       panic("bad name in parse_name");
    while (isalnum(*s) || *s == '_')
       s++;
```

```
   int len = s - *p_s;
   char *name = malloc( len + 1 );
   (void) strncpy(name, *p_s, len);
   name[len] = '\0';

   *p_s = s;
   return name;
}


//
// Typenum:
//      '(' INTEGER ':' INTEGER ')'      { DEV, Sun C symbol tables }
//      INTEGER                          { f77 and all VAX symbol tables }
//
void
Typenum( const char **p_s, int *p_fnum, int *p_tnum )
{
   const char *s = *p_s;
   if (*s == '(')
    {
       ++s;
       *p_fnum = parseNumber(&s);
       scheck(&s, ',');
       *p_tnum = parseNumber(&s);
       scheck(&s, ')');
    } else
    {
       *p_fnum = 0;
       *p_tnum = parseNumber(&s);
    }
   *p_s = s;
}

static void
getBasicType( language_t language, SymTab *st, int symno, dim_t *dim, Type *type )
{
   typedef struct
   {
      const char     *a_name;
      const char     *a_alias;
   } alias_t;
   typedef struct
   {
      long            ct_low;
      long            ct_high;
      typecode_t      ct_code;
      int             ct_size;
   } c_typetab_t;
   typedef struct
   {
      const char     *ft_name;
      typecode_t      ft_code;
      int             ft_size;
   } f77_typetab_t;

   static c_typetab_t c_typetab[] = {
      -2147483648, 2147483647, TY_INT, sizeof(long),
      0, 127, TY_CHAR, sizeof(char),
      -32768, 32767, TY_SHORT, sizeof(short),
      0, 255, TY_UCHAR, sizeof(char),
      0, 65535, TY_USHORT, sizeof(short),
      0, -1, TY_UINT, sizeof(long),
      4, 0, TY_FLOAT, sizeof(float),
      8, 0, TY_DOUBLE, sizeof(double),
   };

#define N_C_TYPES       (sizeof c_typetab / sizeof *c_typetab)
   static f77_typetab_t f77_typetab[] = {
      "integer*2", TY_INTEGER_2, sizeof(short),
      "integer*4", TY_INTEGER_4, sizeof(int),
```

```
      "real", TY_REAL, sizeof(float),
      "double precision", TY_DBLPRES, sizeof(double),
      "complex", TY_COMPLEX, 2 * sizeof(float),
      "double complex", TY_DBLCOMP, 2 * sizeof(double),

      "logical", TY_LOGICAL, sizeof(int),

      "logical*1", TY_LOGICAL, sizeof(char),
      "logical*2", TY_LOGICAL, sizeof(short),
      "logical*4", TY_LOGICAL, sizeof(int),

      "char", TY_CHARACTER, sizeof(char),
      "void", TY_FVOID, 0,
   };

#define N_F77_TYPES     (sizeof f77_typetab / sizeof *f77_typetab)

   static alias_t  aliastab[] = {
      "long int", "long",
      "unsigned int", "unsigned",
      "long unsigned int", "unsigned long",
      "short int", "short",
      "short unsigned int", "unsigned short",
   };
   c_typetab_t    *ct;
   alias_t        *al;
   const char     *line, *cptr;

   if( (line = st->getExecutable()->getSymString( symno )) == NULL )
      panic("NULL line in gbt");
   if ((cptr = strchr(line, ':')) == NULL)
      panic("missing ':' in gbt");
   int len = cptr - line;

   type->setBase( NULL );
   type->setTypeDef( NULL );

   if (language != LANG_C)
      panic("unknown language in dbx symbol table parser");

   // Gcc puts rather cumbersome names for some C types in the symbol table.
   // If the name is one of these, rewrite it in the more usual C idiom,
   // otherwise just take a copy of the name.
   for( al = aliastab;; ++al )
   {
      if( al == aliastab + sizeof aliastab / sizeof *aliastab )
      {
         char *name = malloc( len + 1 );
         (void) strncpy(name, line, len);
         name[len] = '\0';
         type->setName( name );
         break;
      }
      if( strncmp(al->a_name, line, len) == 0 && al->a_name[len] == '\0' )
      {
         type->setName( al->a_alias );
         break;
      }
   }

   if( dim == NULL )
   {
      if( strcmp(type->getName(), "void") != 0 )
         panic("unknown special type in gbt");
      type->setTypeCode( TY_VOID );
      type->setSize( 0 );
      return;
   }
   for( ct = c_typetab; ct < c_typetab + N_C_TYPES; ++ct )
   {
      if( dim->di_low == ct->ct_low && dim->di_high - 1 == ct->ct_high )
      {
```

```
            type->setTypeCode( ct->ct_code );
            type->setSize( ct->ct_size );
            return;
         }
      }

      panic("unknown size range");
   }

   //
   //  TypeId:
   //      Typenum
   //      Typenum '=' TypeDef
   //      Typenum '=' TypeAttrs TypeDef    { NI }
   //
   //
   Type *
   TypeId( stf_t *stf, int *p_symno, const char **p_s, bool eval )
   {
      static char     utypechars[] = "sue";
      static typecode_t utypes[] = {TY_U_STRUCT, TY_U_UNION, TY_U_ENUM};
      char           *pos;
      stf_t          *nstf;
      typecode_t      utypecode;
      int             fnum, tnum, save_symno;
      const char     *tag;
      Type           *rtype=NULL;

      const char *s = *p_s;

      bool is_named_type = (s[-1] == 'T');
      bool is_typedefed  = (s[-1] == 't');
      bool have_typenum  = (isdigit(*s) || *s == '(');

      if(have_typenum)
      {
         Typenum(&s, &fnum, &tnum);
         if( fnum >= stf->getMapSize() )
            panic("fnum out of range in TypeId");
         Header **fmap;
         fmap = stf->getMap();
         nstf = fmap[fnum]->getStf();

         if( *s == '=' && s[1] == 'x' &&
             (pos = strchr(utypechars, s[2])) != NULL )
         {
            utypecode = utypes[pos - utypechars];
            s += 3;
            tag = parseName( &s );
            scheck(&s, ':');
         }
         else
         {
            utypecode = TY_INT_ASSUMED;
            tag = NULL;
         }
         rtype = nstf->lookupTnum( tnum );
      }
      else
      {
         rtype = NULL;
         nstf = stf;
         tnum = -1;
         utypecode = TY_NOTYPE;     /* to satisfy gcc */
         tag = UNSET;               /* to satisfy gcc */
      }

      if (is_named_type && rtype != NULL)
         return rtype;
      save_symno = *p_symno;

      if (have_typenum)
```

```
{
   if (*s == '=')
   {
      ++s;
      if (*s == '@')
         notImplemented("type attributes");
      if (rtype == NULL)
         rtype = TypeDef(nstf, p_symno, &s, tnum, eval);
      else if (!is_typedefed)
         (void) TypeDef(nstf, p_symno, &s, tnum, FALSE);
   }
   else if (eval)
   {
      rtype = nstf->tnumToType( tnum );

      if (rtype == NULL)
      {
         ftype_t *ft = nstf->addType( tnum );
         rtype = Type::makeUndefType( tag, utypecode, ft->ft_type );

         if( utypecode == TY_U_STRUCT ||
             utypecode == TY_U_UNION  ||
             utypecode == TY_U_ENUM )
         {
            // BUG: struct might not be in outermost block.
            aggr_or_enum_def_t *ae = rtype->getAggrOrEnumDef();
            Block *bl = nstf->getSourceFile()->getBlock();

            ae->setNext( bl->getAggrOrEnum() );
            bl->setAggrOrEnum( ae );
         }
      }
   }
}
else
   rtype = TypeDef(nstf, p_symno, &s, -1, eval);
*p_s = s;

if( rtype->getTypeDef() == &Is_basic_type )
{
   is_typedefed = FALSE;
   rtype->setTypeDef( NULL );
}
if (eval && (is_named_type || is_typedefed))
{
   Executable *exec = nstf->getSymTab()->getExecutable();

   const char *name = exec->getSymString( save_symno );

   if (is_typedefed)
   {
      typedef_t      *td;

      td = new typedef_t;
      td->td_name = parseName( &name );
      td->td_lexinfo = NULL;
      td->td_type = rtype;
      rtype->setTypeDef( td );
   }
   else
   {
      switch (rtype->ty_code)
      {
       case TY_STRUCT:
       case TY_UNION:
       case TY_ENUM:
         break;
       default:
         panic("bad type code in TypeId");
      }
      rtype->getAggrOrEnumDef()->ae_tag = parseName( &name );
   }
```

```
    }
    return rtype;
}

//
//  Typedef:
//      TYPENUM                         { NI }
//      Subrange
//      Array                           { PI, DEV }
//      Record
//      'e' Enumlist
//      '*' TypeId
//      'S' Typeid                      { Set of TypeId (NI) }
//      'd' Typeid                      { File of TypeId (NI) }
//      ProcedureType                   { PI }
//      'i' NAME ':' NAME ';'           { NI }
//      'o' NAME ';'                    { NI }
//      'i' NAME ':' NAME ',' TypeId ';'{ NI }
//      'o' NAME ',' TypeId             { NI }
//
static type_t  *
TypeDef( stf_t *stf, int *p_symno, const char **p_s, int tnum, int eval )
{
    ftype_t         *ft, dummy_ftype;
    Type            t;
    aggr_or_enum_def_t *ae;
    dim_t           *dim;
    int              is_struct,
    junk;
    const char      *s;

    dummy_ftype.ft_type = &t;

    if (tnum == -1)
    {
        ft = new ftype_t;
        ft->ft_tnum = -1;
        ft->ft_type->setSize( -1 );
        ft->ft_type->setTypeDef( NULL );
        ft->ft_next = NULL;
    } else
        ft = eval ? stf->addType( tnum ) : &dummy_ftype;

    s = *p_s;
    ae = NULL;
    switch (*s++)
    {
        // '*' TypeId
        case '*':
            ft->ft_type->setTypeCode( DT_PTR_TO );
            ft->ft_type->setQualifiers( 0 );
            ft->ft_type->setBase( TypeId( stf, p_symno, &s, eval ) );
        break;

        // Typenum.  We assume this is a special case basic type (e.g. void).
        case '(':
        case '0':
        case '1':
        case '2':
        case '3':
        case '4':
        case '5':
        case '6':
        case '7':
        case '8':
        case '9':
            --s;
            Typenum(&s, &junk, &junk);
            getBasicType(stf->getLanguage(), stf->getSymTab(),
                         *p_symno, (dim_t *) NULL, ft->ft_type);
        break;
```

```
      // Subrange.  We assume this means a basic type.
      case 'r':
         --s;
         dim = Subrange(stf, p_symno, &s, eval, FALSE);
         getBasicType(stf->getLanguage(), stf->getSymTab(),
                      *p_symno, dim, ft->ft_type);

         // Signal to our caller (TypeId()) that this is not a user typedef.
         ft->ft_type->setTypeDef( &Is_basic_type );

      break;

      // Array. The grammar gives < 'a' TypeId ';' TypeId >, but we currently
      // only cope with < 'a' Subrange ';' TypeId >.
      case 'a':
         ft->ft_type->setTypeCode( DT_ARRAY_OF );
         ft->ft_type->setDimension( Subrange(stf, p_symno, &s, eval, TRUE) );
         scheck(&s, ';');
         ft->ft_type->setBase( TypeId(stf, p_symno, &s, eval) );
      break;

      // ProcedureType.
      //
      case 'f':
         ft->ft_type->setTypeCode( DT_FUNC_RETURNING );
         ft->ft_type->setBase( TypeId(stf, p_symno, &s, eval) );
      break;

      // Record.
       case 's':
       case 'u':
          if (!eval)
             cout << "found aggregate with eval FALSE" << endl;
          is_struct = s[-1] == 's';
          ft->ft_type->setTypeCode( is_struct ? TY_STRUCT : TY_UNION );
          ae = aggr_or_enum_defst::record(stf, p_symno, &s, is_struct, eval, ft->ft_type);
       break;

       // 'e' Enumlist
       case 'e':
          if (!eval)
             cout << "found enum with eval FALSE" << endl;
          ae = EnumList(stf->getSymTab(), p_symno, &s, ft->ft_type);
       break;

       default:
          panic("unknown character in TypeDef");
   }
   *p_s = s;
   return ft->ft_type;
}

//  Enum:
//       'e' Enumlist ';'
//
//  Enumlist:
//       Enum
//       Enumlist Enum
//
static aggr_or_enum_def_t *
EnumList( SymTab *st, int *p_symno, const char **p_s, type_t *type )
{
   const char *s = *p_s;
   aggr_or_enum_def_t *ae = new aggr_or_enum_defst( (const char *) NULL, TY_ENUM, type );
   ae->ae_is_complete = AE_COMPLETE;
   ae->ae_enum_members = NULL;
   for (;;)
   {
      if (*s == '\\')
         s = st->getExecutable()->getSymString( ++*p_symno );
      if (*s == ';' || *s == '\0')
         break;
```

```
      ae->Enum( st, &s );
   }
   if (*s != '\0')
      scheck(&s, ';');
   *p_s = s;
   return ae;
}

//
//  Enum:
//      NAME ':' OrdValue ','
//
//  OrdValue:
//      INTEGER
//
void
aggr_or_enum_defst::Enum( symtab_t *st, const char **p_s )
{
   const char *s = *p_s;

   const char *name = parseName(&s);
   scheck(&s, ':');
   int val = parseSignedNumber(&s);
   scheck(&s, ',');

   enum_memberst *em = new enum_memberst( name, val );
   em->em_enum = this;

   em->em_next = ae_enum_members;
   ae_enum_members = em;

   *p_s = s;
}

//
//  Record:
//      's' ByteSize Fieldlist ';'
//      'u' ByteSize Fieldlist ';'
//
//  TypeDef above has already skipped the 's' or 'u'.
//
aggr_or_enum_def_t *
aggr_or_enum_defst::record( stf_t *stf,
                            int *p_symno,
                            const char **p_s,
                            int is_struct,
                            bool eval,
                            type_t *type )
{
   const char     *s;
   aggr_or_enum_def_t *ae;
   var_t          *members;

   s = *p_s;
   ae = new aggr_or_enum_defst( (const char *) NULL,
                                is_struct ? TY_STRUCT : TY_UNION, type);
   ae->ae_is_complete = AE_COMPLETE;
   ae->ae_size = parseNumber(&s);
   members = NULL;
   for (;;)
   {
      if (*s == '\\')
         s = stf->getSymTab()->getExecutable()->getSymString( ++*p_symno );
      if (*s == ';')
         break;
      members = Variable::field(stf, p_symno, &s, is_struct, members, eval);
   }
   ae->ae_aggr_members = members;

   scheck(&s, ';');
   *p_s = s;
   return ae;
```

```
}

//
// Field:
//      NAME ':' TypeId ',' BitOffset ',' BitSize ';'
//
Variable *
Variable::field( STFile *stf, int *p_symno, const char **p_s, int is_struct, Variable *next, boo
{
   const char *s = *p_s;
   const char *name = parseName(&s);
   scheck( &s, ':' );
   Type *type = TypeId( stf, p_symno, &s, eval );
   Variable *v = new Variable( name, is_struct ? CL_MOS : CL_MOU, type, (taddr_t)NULL );

   v->setLanguage( stf->getLanguage());
   v->va_flags = 0;
   scheck(&s, ',');
   int offset = parseNumber(&s);
   scheck(&s, ',');

   int width = parseNumber(&s);

   scheck(&s, ';');
   *p_s = s;
   if (!eval)
      return NULL;

   v->va_addr = offset / 8;

   if( TY_IS_BASIC_TYPE(type->ty_code) && width != type->typeSize() * 8 )
      v->va_type = Type::makeBitFieldType(type->getTypeCode(),
                                          offset - v->va_addr * 8,
                                          width);
   else
      v->va_type = type;
   v->va_next = next;
   return v;
}

//
//  Subrange:
//      'r' TypeId ';' INTEGER ';' INTEGER
//
static dim_t *
Subrange( STFile *stf, int *p_symno, const char **p_s, bool eval, bool want_subrange_type )
{
   const char     *s;
   dim_t          *dim;

   s = *p_s;
   scheck(&s, 'r');
   dim = new dimst;

   if (want_subrange_type)
      dim->di_type = TypeId(stf, p_symno, &s, eval);
   else
   {
      int             junk;

      Typenum(&s, &junk, &junk);
   }

   scheck(&s, ';');
   if (dim->di_1dynamic = *s == 'T' || *s == 'A' || *s == 'J')
      ++s;
   dim->di_low = parseSignedNumber(&s);
   scheck(&s, ';');
   if (dim->di_hdynamic = *s == 'T' || *s == 'A' || *s == 'J')
      ++s;
   dim->di_high = parseSignedNumber(&s) + 1;
   *p_s = s;
```

```
/*-------------------------------------------------------------------
 |       FILE: target.C
 |    PURPOSE: Declaration of target class object.  Hi level control
 |             of inferior process.
  -------------------------------------------------------------------*/

#ifndef TARGET_H_INCLUDED
#define TARGET_H_INCLUDED

#include "proc.h"
#include "breakpoint.h"

typedef enum rtypeen
{
        RT_STEP,
        RT_NEXT,
        RT_CONT
} rtype_t;



class Target
{
 public:

   Target( const char *path, const char **argv, const char **envp );

   int attach( const char *name, int pid );
   void detach();

   void start( stopres_t *p_whystopped );

   void Continue( Breakpoint *stop_bp, stopres_t *p_whystopped );

   void run( Breakpoint *stop_bp, rtype_t rtype );

   void bump( rtype_t rtype, stopres_t *p_whystopped );

   Process *getProcess() { return ta_process; }

   static Function *getMainFunction() { return ta_main_function; }

   bool cannotContinue() { return ta_cannot_cont; }

 protected:

   stopres_t continueOverSignals();

   static Breakpoint *ta_stuck_bp;
   Process           *ta_process;
   const char        *ta_exec_name;
   const char       **ta_argv;
   const char       **ta_envp;
   static Function   *ta_main_function;
   bool               ta_cannot_cont;
};

#endif
```

```
/*-----------------------------------------------------------------
 |       FILE: target.C
 |    PURPOSE: Methods for target class object.  Hi level control
 |             of inferior process.
 |
 |   CONTENTS: -methods-        -------------purpose-----------------
 |             Target           constructor
 |             Continue
 |             continueOverSignals
 |             bump
 |             run
 |             start
 |             attach
 |             detach
 -----------------------------------------------------------------*/
#include <signal.h>
#include "symtab.h"
#include "proc.h"
#include "breakpoint.h"
#include "as.h"
#include "target.h"
#include "stack.h"
#include "bat.h"

Breakpoint *Target::ta_stuck_bp=NULL;


/*-----------------------------------------------------------------------
 |     METHOD: Continue
 |
 |    PURPOSE: Continue running target
 |
 -----------------------------------------------------------------------*/
 d
   get::Continue( Breakpoint *stop_bp, stopres_t *p_whystopped )

   stopres_t       whystopped;
   int             stop,
                   sig;
   Breakpoint      *bp;

   taddr_t orig_fp = 0;                  /* but may be changed below */
   bool should_uninstall_stop_bp = FALSE;

   if( ta_cannot_cont )
   {
      *p_whystopped = SR_SIG;
      return;
   }

   if (ta_process != 0)
   {
      if (stop_bp != 0)
      {
         if (!stop_bp->breakpointIsInstalled())
         {
            if (stop_bp->install(ta_process) != 0)
               return;
            should_uninstall_stop_bp = TRUE;
         }
         if (ta_process->getRegister(REG_PC) == stop_bp->breakpointToAddr())
            orig_fp = ta_process->getRegister(REG_FP);
      } else
      {
         if (Breakpoint::installAllBreakpoints(ta_process) != 0)
            return;
      }
   }
   do
   {
      if (ta_process == 0)
      {
```

```
         start(&whystopped);
         if (ta_process == 0)
            break;
      } else
         whystopped = ta_process->Continue( ta_process->getRestartSignal(), PR_CONT );

      switch (whystopped)
      {
      case SR_DIED:
      case SR_USER:
      case SR_FAILED:
         ta_cannot_cont = TRUE;
         stop = TRUE;
         break;
      case SR_SSTEP:
         stop = FALSE;
         break;
      case SR_SIG:
         sig = ta_process->getLastSignal();

         int sigflags = ta_process->getSignalFlags( sig );

         if( sigflags&SIG_CATCH )
         {
            stop = TRUE;
            ta_cannot_cont = sigflags&SIG_TERMINATES;
         }
         else
            stop = FALSE; // stop_bp==0
         break;
      case SR_BPT:
         taddr_t pcaddr = ta_process->getRegister(REG_PC);
         bp = Breakpoint::getBreakpointAtAddress( ta_process, pcaddr );

         stop = TRUE;

         whystopped = ta_process->getWhyStopped();
         if (whystopped != SR_BPT)
            stop = TRUE;
         if (stop_bp != 0)
            stop = bp == stop_bp && ta_process->getRegister(REG_FP) >= orig_fp;
         break;
      default:
         panic("stopres botch in Target::continue()");
         stop = UNSET;          /* to satisfy gcc */
      }
   } while (!stop);

   if (should_uninstall_stop_bp && stop_bp->breakpointIsInstalled())
   {
      if (stop_bp->uninstall() != 0)
         ta_stuck_bp = stop_bp;
   }
   *p_whystopped = whystopped;
}


/*-----------------------------------------------------------------------
 |   FUNCTION: jump_needed
 |
 |    PURPOSE: Return TRUE if a jump is needed.  The only jumps that aren't
 |             needed are calls to a known address where the known address
 |             is within a function for which we don't have symbol table
 |             information.
 |
 |             You might think we could also skip branches to a destination
 |             within the line, but we can't because these jumps might jump
 |             over the breakpoint we have set.
 |
 -----------------------------------------------------------------------*/
static int
```

```
jump_needed( jump_t *j )
{
   if (j->ju_type == JT_CALL && j->ju_dstaddr != 0)
   {
      Function *f = SymTab::addrToFunc(j->ju_dstaddr);
      return f != NULL && f->getLineNumbers() != NULL;
   }
   return TRUE;
}



static jump_t *
get_lno_jumps( Function *f, lno_t *lno, int want_calls, taddr_t *p_addr, taddr_t *p_nextaddr )
{
   taddr_t         nextaddr;

   if (lno == NULL || f == NULL)
      panic("lno or f NULL in get_lno_jumps");
   taddr_t addr = lno->ln_addr;

   // Either get the next line's address OR
   // if this is the last line in the function, then get the address
   // limit of the function.

   if (lno->getNext() != NULL)
      nextaddr = lno->getNext()->getAddress();
   else
      nextaddr = f->getSymTab()->getFunctionTab()->getAddrLim(f);

   // fetch the text between the current lines address and the next lines address
   int len = nextaddr - addr;
   char *text = (char *)malloc(len);
   if (f->getSymTab()->getExecutable()->readText(addr, text, len) != 0)
   {
      cerr << "Can't read text of " << f->getName() << endl;
      return NULL;
   }

   // Call the assembler object to get all the jumps that occur in the
   // retrieved text (code).
   jump_t *jumps = get_jumps(addr, text, len, want_calls, TRUE);
   free(text);

   jump_t *dst = jumps;
   jump_t *src = dst;
   for (;;)
   {
      if (jump_needed(src))
      {
         if (src != dst)
            *dst = *src;
         ++dst;
      }
      if (src->ju_type == JT_END)
         break;
      ++src;
   }

   *p_addr = addr;
   *p_nextaddr = nextaddr;
   return jumps;
}



/*-------------------------------------------------------------------------
 |     METHOD: continueOverSignals
 |
 |    PURPOSE: Continue running over signals.
```

```
 ------------------------------------------------------------------------*/
stopres_t
Target::continueOverSignals()
{
   taddr_t lastsp = NULL;
   taddr_t lastpc = NULL;
   int lastsig = 0;
   stopres_t whystopped;

   for (;;)
   {
      whystopped = ta_process->Continue( ta_process->getRestartSignal(), PR_CONT );
      if (whystopped != SR_SIG)
         break;

      int sig    = ta_process->getLastSignal();
      taddr_t pc = ta_process->getRegister(REG_PC);
      taddr_t sp = ta_process->getRegister(REG_SP);

      if (pc == lastpc && sp == lastsp && sig == lastsig)
         break;
      lastpc = pc;
      lastsp = sp;
      lastsig = sig;

      if( ta_process->getSignalState(sig) == SGH_DEFAULT)
         break;
   }
   return whystopped;
}



/*------------------------------------------------------------------------
 |     METHOD: bump
 |
 |    PURPOSE: bump the target until we leave this frame.
 ------------------------------------------------------------------------*/
void
Target::bump( rtype_t rtype, stopres_t *p_whystopped )
{
   static char     error_stackfile[] = "bat_stack.error";
   jump_t                           *j;
   lno_t          *lno,
                  *old_lno;

   taddr_t         fp,
                   orig_fp,
                   adjusted_pc,
                   pc,
                   last_pc,
                   addr,
                   nextaddr,
                   bpt_addr;
   stopres_t       whystopped;
   Function       *f, *old_f;

   // get the pc
   pc = adjusted_pc = ta_process->getRegisterValue( REG_PC );

   // what function is this pc in?
   f = SymTab::addrToFunc( pc );

   // We want the adjusted pc (see get_current_func) for the breakpt addr == pc
   // test below.  This is to avoid a false positive when the saved pc after
   // a function call points at the next source line.
   //
   pc = adjusted_pc;

   if (f == NULL || (lno = f->addrToLno(pc)) == NULL)
   {
```

```
    // we don't handle this well yet...
    //
    cerr << "Can't step in function with no line number information" << endl;
    const char *errfun = f ? f->getName() : "NULL";
    cerr << "  Function is " << errfun << endl;
    cerr << "  pc address == " << pc;

    return;
}
taddr_t addrlim = f->getSymTab()->getFunctionTab()->getAddrLim(f);

jump_t *jumps = get_lno_jumps(f, lno, rtype == RT_STEP, &addr, &nextaddr);

if (jumps == NULL || Breakpoint::uninstallAllBreakpoints(ta_process) != 0)
    return;

Breakpoint *bp = NULL;
for (;;)
{
    // Find the first jump instruction after or at the pc.

    for (j = jumps; j->ju_type != JT_END && j->ju_addr < pc; ++j)
        ;
    bpt_addr = j->ju_type != JT_END ? j->ju_addr : nextaddr;
    if (bpt_addr < pc)
    {
        panic("Target::bump() addr botch");
    }

    // If we are not already at the jump, set a breakpoint at the jump and
    // continue the target to let it get there. Check that we got to the
    // breakpoint we were expecting.
    //
    if (bpt_addr == pc)
    {
        whystopped = SR_BPT;
        fp = ta_process->getRegister(REG_FP);
    }
    else
    {
        bp = new Breakpoint(bpt_addr);
        if (bp->install(ta_process) != 0)
        {
            whystopped = SR_FAILED;
            break;
        }
        for (;;)
        {
            whystopped = continueOverSignals();
            if (whystopped != SR_BPT)
            {
                fp = UNSET;       /* to satisfy gcc */
                break;
            }
            fp = ta_process->getRegister(REG_FP);
            if (fp >= orig_fp)
                break;
        }

        // Remove the breakpoint
        delete bp;

        bp = NULL;
        if (whystopped != SR_BPT)
            break;
        pc = ta_process->getRegister(REG_PC);
        if (pc != bpt_addr)
            panic("Target::bump ->unexpected pc");
    }


    // If we have reached the next line, we've finished the next or step.
```

```
//
// As we fell through to this line rather than jumping, this is the
// to execute code associated with any breakpoint that resides here.
// This note is for future reference since we don't do tracing yet.
//
if (bpt_addr == nextaddr && (rtype == RT_STEP || fp >= orig_fp))
{
   Breakpoint *bp_at_nextline;

   bp_at_nextline = Breakpoint::addrToBreakpoint(pc);
   if (bp_at_nextline != NULL)
   {
      panic( "Target::bump() -> have not implemented tracing yet!" );
   }
   break;
}

// OK, we are now sitting at a jump instruction of some sort. Single
// step to execute that instruction and find out where we jumped to.
//
whystopped = ta_process->Continue(ta_process->getRestartSignal(), PR_STEP);
if (whystopped != SR_SSTEP)
   break;

last_pc = pc;
pc = ta_process->getRegister(REG_PC);
fp = ta_process->getRegister(REG_FP);

// If we are still at the same line and stack level, continue.
//
if (pc >= addr && pc < nextaddr && fp <= orig_fp)
{
   if (pc < last_pc)
      break;
   else
      continue;
}

// Off the line - find the new lno and func. We save the old lno first
// because we want to check that we've moved onto a different source
// line (you can get multiple lnos with the same source line number).

old_lno = lno;
old_f = f;
if (pc == nextaddr && lno->getNext() != NULL)
   lno = lno->getNext();
else
{
   if (pc < f->getAddress() || pc >= addrlim)
   {
      f = SymTab::addrToFunc(pc);
      if (f == NULL)
         break;

      addrlim = f->getSymTab()->getFunctionTab()->getAddrLim(f);
   }
   lno = f->addrToLno(pc);

   // If we don't know which lno we're at after a branch, stop.
   //
   // This can happen when returning to function with no symbol table
   // information (a RET instruction is treated as a branch).
   //
   // A call to a function seems to put the pc in the function prologue,
   // so we don't mind not knowing which lno we're at after a call. We
   // move the pc onwards below in this case.
   if (j->ju_type != JT_CALL && lno == NULL)
      break;
}

// We're done if we're at a source line boundary with a different
// source line number than the current lno.
```

```
    //
    if (lno != NULL)
    {
       bool            on_boundary,
                       new_func,
                       new_lnum,
                       new_level;

       on_boundary = pc == lno->getAddress();
       new_func = f != old_f;
       new_lnum = lno->ln_num != old_lno->getLNumber();
       new_level = fp != orig_fp;

       if (on_boundary && (new_func || new_lnum || new_level))
          break;
    }

    // If we have just stepped into a function, move the pc up to the
    // minimum breakpoint address and stop.
    //
    if (j->ju_type == JT_CALL)
    {
       taddr_t call_bpt_addr = f->minBreakpointAddress();
       Breakpoint *bp_at_start_of_func = Breakpoint::addrToBreakpoint(call_bpt_addr);
       Breakpoint *tmp_bp;

       if (pc == call_bpt_addr)
          whystopped = SR_BPT;
       else
       {
          if (bp_at_start_of_func != 0)
          {
             bp = bp_at_start_of_func;
             tmp_bp = 0;
          } else
          {
             bp = new Breakpoint(call_bpt_addr);
             tmp_bp = bp;
          }
          if (bp->install(ta_process) != 0)
             break;
          whystopped = ta_process->Continue(0, PR_CONT);

          if( tmp_bp != 0 )
          {
             delete tmp_bp;
             break;
          }


          bp = NULL;
       }

       if (whystopped != SR_BPT)
          break;
       if (ta_process->getRegister(REG_PC) != call_bpt_addr)
          panic("Target::bump()->pc escaped");

       if (bp_at_start_of_func != NULL)
       {
          panic( "Target::bump() tracing not done yet." );
       }
       break;
    }
    // Move focus to the new line.
    jumps = get_lno_jumps(f, lno, rtype == RT_STEP, &addr, &nextaddr);
    orig_fp = ta_process->getRegister(REG_FP);
    if (jumps == NULL)
       break;
 }

 if (bp != 0 )
```

```
      delete bp;

   *p_whystopped = whystopped;
}



/*-----------------------------------------------------------------------
 |     METHOD: run
 |
 |    PURPOSE: Run the target as requested in rtype.  Rtype can be
 |             RT_STEP, RT_NEXT, or RT_CONT.
 -----------------------------------------------------------------------*/
void
Target::run( Breakpoint *stop_bp, rtype_t rtype )
{
   stopres_t       whystopped;

   if( ta_cannot_cont==TRUE && rtype!=RT_CONT )
   {
      BAToutstream::send( "Cannot continue target." );
      return;
   }

   if (ta_process == 0 && rtype != RT_CONT)
      panic("Target::run() ---> bad rtype");

   if (ta_stuck_bp != 0)
   {  // bogus!!!
      if( !ta_stuck_bp->uninstall() )
      {
         delete ta_stuck_bp;
         ta_stuck_bp = 0;
      }
      else
         return;
   }
   // Set whystopped to anything other than SR_DIED or SR_SIG.
   whystopped = SR_BPT;

   // reset cannot continue flag
   ta_cannot_cont = FALSE;

   if (rtype == RT_CONT)
      Continue( stop_bp, &whystopped );
   else
   {
      bump(rtype, &whystopped);
      if (whystopped != SR_DIED)
      {
         taddr_t sp = ta_process->getRegister(REG_SP);
         if (sp > ta_process->getBaseSP())
            Continue((Breakpoint *) 0, &whystopped);
      }
   }

   if (whystopped == SR_DIED)
   {
      ta_process = 0;
      SymTab::unloadSharedLibrary();
      return;
   }

   // get the current stack frame
   Stack *s = new Stack( ta_process );
}


/*-----------------------------------------------------------------------
 |     METHOD: Target
 |
```

```
 |    PURPOSE: Target object constructor.
 ------------------------------------------------------------------------*/
Target::Target( const char *path, const char **argv, const char **envp )
{
   ta_exec_name = strsave(path);
   ta_argv = argv;
   ta_envp = envp;
   ta_process = NULL;
   ta_stuck_bp = NULL;

   int junk;
   Function *f = SymTab::nameToFunction("MAIN", &junk);
   if(f == NULL || f->getLanguage() != LANG_FORTRAN)
      f = SymTab::nameToFunction("main", &junk);
   if(f == NULL)
      cout << "Target::Target()-->Can't find function 'main' in the symbol table" << endl;

   ta_main_function = f;
   ta_cannot_cont = TRUE;
}




/*------------------------------------------------------------------------
 |     METHOD: start
 |
 |    PURPOSE: Start the target process running.  Creates the process,
 |             installs all breakpoints and lets it rip.
 |
 ------------------------------------------------------------------------*/
void
Target::start( stopres_t *p_whystopped )
{
   if( !ta_main_function )
   {

      int junk;
      Function *f = SymTab::nameToFunction("MAIN", &junk);
      if(f == NULL || f->getLanguage() != LANG_FORTRAN)
         f = SymTab::nameToFunction("main", &junk);
      if(f == NULL)
         panic( "Target::start()-->Can't find function 'main' in the symbol table" );

      ta_main_function = f;
   }

   taddr_t stop_addr = ta_main_function->minBreakpointAddress();

   const char *err_mesg;
   ta_process = Process::start(ta_exec_name, ta_argv, ta_envp, 0, stop_addr,
                               &err_mesg, p_whystopped);

   while(*p_whystopped == SR_SIG)
      *p_whystopped = ta_process->Continue(ta_process->getLastSignal(), PR_CONT);

   if(*p_whystopped == SR_DIED)
   {
      // delete doesn't really detach, etc cleanly yet!!!
      if( ta_process )
      {
         delete ta_process;
         ta_process = NULL;
      }

      SymTab::unloadSharedLibrary();

      return;
   }
   if(*p_whystopped != SR_BPT)
      panic("proc not started properly on Target::start()");
```

```
   ta_process->setBaseSP( ta_process->getRegister(REG_SP) );

   SymTab::loadSharedLibrary();

   if(Breakpoint::installAllBreakpoints(ta_process) != 0)
      panic("Can't install breakpoints");

   // If we have a breakpoint at stop_addr, we don't need to continue the
   // target - we are already at the breakpoint.

   if(Breakpoint::getBreakpointAtAddress(ta_process, stop_addr) == NULL)
      *p_whystopped = ta_process->Continue(0, PR_CONT);

   if(*p_whystopped == SR_DIED)
   {
      delete ta_process;
      ta_process = NULL;

      SymTab::unloadSharedLibrary();
   }
}


/*-----------------------------------------------------------------------
 |     METHOD: attach
 |
 |    PURPOSE: Attach target to a running process.
 |
 -----------------------------------------------------------------------*/
int
Target::attach( const char *name, int pid )
{
   stopres_t       whystopped;

   if( Process::getCurrentProcess() )
      panic("Current process != NULL in Target::attach()");

   ta_process = Process::attach( name, pid, &whystopped );

   // Check that we attached sucessfully.
   if( !ta_process || !ta_process->getAttached() )
   {
      cerr << "Can't attach to process " << pid << endl;
      return -1;
   }
   if(whystopped == SR_DIED)
   {
      cerr << "process " << pid << " died on being attached to!" << endl;
      delete ta_process;
      return -1;
   }

   SymTab::loadSharedLibrary();

   return 0;
}


/*-----------------------------------------------------------------------
 |     METHOD: detach
 |
 |    PURPOSE: Detaches from process if attached.  Otherwise just kill
 |             the target's process.
 -----------------------------------------------------------------------*/
void
Target::detach()
{
   if( !ta_process )
      return;

   if( ta_process->getAttached() )
      ta_process->detach( ta_process->getLastSignal() );
```

```
    else
       ta_process->terminate();

    delete ta_process;
    ta_process = NULL;

    SymTab::unloadSharedLibrary();
}
```

```
/*-------------------------------------------------------------------
 |       FILE: type.h
 |    PURPOSE: type class declaration.  Type objects are one of the
 |             main components of the symbol table.
  -------------------------------------------------------------------*/

#ifndef TYPE_H_INCLUDED
#define TYPE_H_INCLUDED

#include "data.h"

//-------------------------------------------------------------------
// Type structures
//   There is a structure for each type derivation ('pointer to',
//   'array of', or 'function returning' or a basic type).  A basic
//   type is either one of the C standard types, or an aggregate
//   (struct, enum, or union).  These type structures follow:
//-------------------------------------------------------------------


//  Lexical information - currently a filename and line number.
//
//  Used for recording where things are in the source for better
//  error reporting.
//
typedef struct lexinfost
{
   const char    *lx_filename;
   int            lx_lnum;
   int            lx_cnum;
} lexinfo_t;


// Member of enum.

typedef struct enum_memberst
{
//   expr_id_t                          em_expr_id;
   enum_memberst( const char *name, long val );

   const char                         *em_name;
   long                               em_val;
   struct aggr_or_enum_defst          *em_enum;      // ptr back to parent enum struct
   struct enum_memberst               *em_next;
   lexinfo_t                          *em_lexinfo;
} enum_member_t;

typedef enum { AE_COMPLETE, AE_INCOMPLETE } ae_is_complete_t;

//
//  Variable code types.
//
typedef enum typecodeen
{
   // derivations
   DT_PTR_TO,           // pointer to type ty_base
   DT_FUNC_RETURNING,   // function returning type ty_base
   DT_ARRAY_OF,         // array [ty_dim] of ty_base

   // seperator between derivations and base types
   TY_NOTYPE,

   // C aggregate types etc
   TY_STRUCT,           // structure
   TY_UNION,            // union
   TY_ENUM,             // enumeration
   TY_U_STRUCT,         // undefined structure
   TY_U_UNION,          // undefined union
   TY_U_ENUM,           // undefined enum
   TY_BITFIELD,         // bitfield

   // seperator between aggregate and basic types
```

```
   TY_BASIC_TYPES,

   // C base types
   TY_VOID,              // void
   TY_CHAR,              // character
   TY_UCHAR,             // unsigned character
   TY_SHORT,             // short integer
   TY_USHORT,            // unsigned short
   TY_INT,               // integer
   TY_UINT,              // unsigned int
   TY_LONG,              // long integer
   TY_ULONG,             // unsigned long
   TY_FLOAT,             // floating point
   TY_DOUBLE,            // double word
   TY_INT_ASSUMED,       // unknown - int assumed

   // Maximum type number
   TY_MAXTYPE
} typecode_t;

#define TY_ISDERIV(code)        ((int)(code) < (int)TY_NOTYPE)
#define TY_IS_BASIC_TYPE(code)  ((int)(code) > (int)TY_BASIC_TYPES)

//  Aggregate (struct, union) or enum definition.
//
typedef struct aggr_or_enum_defst
{
   aggr_or_enum_defst( const char *tag, typecode_t typecode, Type *type );

   static aggr_or_enum_defst * record(STFile *stf,
                                      int *p_symno,
                                      const char **p_s,
                                      int is_struct,
                                      bool eval,
                                      Type *type );


   // does tag match the aggr_or_enum_defst structure?  if so return the struct
   // otherwise return NULL
   static aggr_or_enum_defst *matchTag( aggr_or_enum_defst *ae, const char *tag );

   static aggr_or_enum_defst *fixUndefAggr( aggr_or_enum_defst *uae, const char *arg );

   void Enum( SymTab *st, const char **p_s );

   aggr_or_enum_defst *
   applyToAelist( aggr_or_enum_defst *(*func)(aggr_or_enum_defst *ae, const char *farg),
                  const char *arg );

   aggr_or_enum_defst *getNext() { return ae_next; }
   void setNext( aggr_or_enum_defst *next ) { ae_next = next; }

   const char                      *ae_tag;
   ae_is_complete_t                ae_is_complete;
   int                             ae_size;
   int                             ae_alignment;
   class Type                      *ae_type;
   struct aggr_or_enum_defst       *ae_next;

   union
   {
      class Variable               *aeu_aggr_members;
      enum_member_t                *aeu_enum_members;
      struct aggr_or_enum_defst    *aeu_sublist;
   } ae_u;

   lexinfo_t                       *ae_lexinfo;
} aggr_or_enum_def_t;

#define ae_aggr_members         ae_u.aeu_aggr_members
#define ae_enum_members         ae_u.aeu_enum_members
#define ae_sublist              ae_u.aeu_sublist
```

```
//  Array size description.  For C, the type of the range must be int
//  and di_low must be zero.
//
typedef struct dimst
{
   short          di_ldynamic;                    // TRUE if lower bound is dynamic
   short          di_hdynamic;                    // TRUE if upper bound is dynamic
   long           di_low;                         // lower bound, or where same can be foun
   long           di_high;                        // upper bound, or where same can be foun
   class Type     *di_type;                       // type of an element
} dim_t;



//  Bitfield description.
//
typedef struct bitfieldst
{
   typecode_t     bf_code;
   short          bf_offset;
   short          bf_width;
} bitfield_t;



typedef struct identifierst
{
   const char                          *id_name;
   lexinfo_t                           *id_lexinfo;
   bool                                id_lparen_follows;
} identifier_t;

typedef struct identifier_listst {
   identifier_t                        *idl_id;
   struct identifier_listst            *idl_next;
} identifier_list_t;

typedef enum { FDT_IDLIST, FDT_TYPELIST } params_type_t;

//  Represents a "function returning" derivation.
//
typedef struct funcretst
{
   params_type_t                       fr_params_type;
   int                                 fr_nparams;
   bool                                fr_is_variadic;
   bool                                fr_is_old_style;
   class Variable                      *fr_params;
   identifier_list_t                   *fr_idlist;
} funcret_t;

//  Type qualifiers.  At present just a bit mask.

typedef unsigned qualifiers_t;

#define QU_VOLATILE     01
#define QU_CONST        02



/*-----------------------------------------------------------------------
  |      CLASS: Type
  |    PURPOSE: C types are represented in bat as a linked list, with an
  |             element in the list for each derivation ('pointer to',
  |             'array of', or 'function returning').  Each element in the
  |             list has a code which is either one of the above derivations
  |             or a basic type.  A basic type is either one of the C
  |             standard types, or an aggregate ('struct', 'union', or
  |             'enum').  A Type object has a union which contains pointers
  |             to more information about the type.  The ty_typecode member
```

```
 |            determines wihch element of the union should be used.
 ------------------------------------------------------------------------*/

class Type
{
 public:

   // Constructor
   Type( typecode_t typecode );
   Type() { ty_code=TY_NOTYPE; ty_size=-1; ty_base=NULL; ty_typedef=NULL; }

   static Type *makeUndefType( const char *tag, typecode_t typecode, Type *type );

   static Type *makeBitFieldType( typecode_t typecode, int bit_offset, int bit_width );

   static Type *makePointerType( Type *base, qualifiers_t qualifiers );

   taddr_t getValue( value_t& value, taddr_t daddr );

   int typeSize();

   static Type * codeToType( typecode_t typecode );

   // access methods
   typecode_t getTypeCode() { return ty_code;}
   void setTypeCode( typecode_t t ) { ty_code = t;}
   aggr_or_enum_def_t *getAggrOrEnumDef() { return ty_u.tyu_aggr_or_enum; }
   void setAggrOrEnumDef( aggr_or_enum_def_t *ae) { ty_u.tyu_aggr_or_enum = ae; }
   typedefst *getTypeDef( ) { return ty_typedef; }
   void setTypeDef( typedefst *t ) { ty_typedef = t; }
   qualifiers_t getQualifiers() { return ty_u.tyu_qualifiers; }
   void setQualifiers( qualifiers_t q ) { ty_u.tyu_qualifiers = q; }
   class Type *getBase() { return ty_base; }
   void setBase( class Type *t ) { ty_base = t; }
   int getSize() { return ty_size; }
   void setSize( int size ) { ty_size = size; }
   const char *getName() { return ty_u.tyu_name;}
   void setName( const char *name) { ty_u.tyu_name = name;}
   dim_t *getDimension() { return ty_u.tyu_dim; }
   void setDimension( dim_t *d ) { ty_u.tyu_dim = d; }

   typecode_t                  ty_code;                    // derivation or base type
   int                         ty_size;                    //  size in bytes of this type
   class Type                  *ty_base;                   // type that this type is deriv
   union
   {
      qualifiers_t             tyu_qualifiers;             // DT_PTR_TO
      dim_t                    *tyu_dim;                   // DT_ARRAY_OF
      aggr_or_enum_def_t       *tyu_aggr_or_enum;          // TY_STRUCT/TY_UNION/TY_ENUM
      bitfield_t               *tyu_bitfield;              // TY_BITFIELD
      identifier_t             *tyu_identifier;            // TY_IDENTIFIER (pseudo)
      const char               *tyu_name;                  // a base type
      funcret_t                *tyu_funcret;               // DT_FUNC_RETURNING
   } ty_u;
   struct typedefst            *ty_typedef;                // non NULL if this type is def

   static class Type           *ty_typetab;                // [TY_MAXTYPE];

};
typedef class Type type_t;

#define ty_aggr_or_enum ty_u.tyu_aggr_or_enum
#define ty_dim          ty_u.tyu_dim
#define ty_bitfield     ty_u.tyu_bitfield
#define ty_name         ty_u.tyu_name
#define ty_identifier   ty_u.tyu_identifier
#define ty_funcret      ty_u.tyu_funcret
#define ty_qualifiers   ty_u.tyu_qualifiers


typedef struct typedefst
```

```
{
   Type *getType() { return td_type;}
   void setType( Type *t ) { td_type = t;}

   typedefst *getNext() { return td_next; }
   void setNext(typedefst *next) { td_next = next; }

   const char                  *td_name;
   class Type                  *td_type;          // the type_t structure that points to this
   struct typedefst            *td_next;
   union
   {
      lexinfo_t                *tdu_lexinfo;
      struct typedefst         *tdu_sublist;
   } td_u;
} typedef_t;

#define td_lexinfo      td_u.tdu_lexinfo
#define td_sublist      td_u.tdu_sublist

#endif
```

```
/*-------------------------------------------------------------------
 |       FILE: type.h
 |    PURPOSE: type class declaration.  Type objects are one of the
 |             main components of the symbol table.
 -------------------------------------------------------------------*/
#include "symtab.h"

class Type *Type::ty_typetab = new Type[TY_MAXTYPE];

Type::Type(typecode_t typecode)
{
   ty_code = typecode;
   ty_size = -1;                /* i.e., not yet set */
   ty_base = NULL;
   ty_typedef = NULL;
}

//  Create an aggregate type for an undefined structure, union or enum.
//
Type *
Type::makeUndefType( const char *tag, typecode_t typecode, Type *type )
{
   aggr_or_enum_def_t *ae;

   switch (typecode)
   {
    case TY_U_STRUCT:
    case TY_U_UNION:
      ae = new aggr_or_enum_defst( tag, typecode, type );
      ae->ae_aggr_members = NULL;
      type = ae->ae_type;
      break;
    case TY_U_ENUM:
      ae = new aggr_or_enum_defst( tag, typecode, type );
      ae->ae_enum_members = NULL;
      type = ae->ae_type;
      break;
    case TY_INT_ASSUMED:
      if (type == NULL)
         type = codeToType( typecode );
      else
         *type = *codeToType( typecode );
      break;
    default:
      panic("unknown typecode in mut");
   }
   return type;
}

/*-------------------------------------------------------------------
 |     METHOD: getValue
 |
 |    PURPOSE: If successful, getValue() returns the
 |             address in the inferior process of the value read.
 |             Otherwise, it returns NULL.
 |       NOTE:
 -------------------------------------------------------------------*/
taddr_t
Type::getValue( value_t& value, taddr_t daddr )
{
   int rc=0;
   dim_t *dim;
   Type *elemType;

   switch( getTypeCode() )
   {
    case( DT_ARRAY_OF ):
       // for arrays, we'll return the address of the first element
       value.v1_addr = daddr;

       elemType = getBase();
       dim      = getDimension();
```

```
    break;

    // pointer to some base type
    case( DT_PTR_TO ):
       // read the address and load it into the value
       rc = (dread( daddr, (char *)&value.vl_addr, sizeof(value.vl_addr) )==-1);
    break;

    case( TY_UINT ):
    case( TY_LONG ):
    case( TY_ULONG):
    case( TY_INT ):
       rc = (dread( daddr, (char *)&value.vl_int, sizeof(value.vl_int) )==-1);
    break;

    case TY_USHORT:
    case TY_SHORT:
       rc = (dread( daddr, (char *)&value.vl_short, sizeof(value.vl_short) )==-1);
    break;

    case TY_CHAR:
    case TY_UCHAR:
       rc = (dread( daddr, (char *)&value.vl_char, sizeof(value.vl_char) )==-1);
    break;

    case TY_BITFIELD:
       daddr = NULL;
       cout << "TY_BITFIELD not implemented yet." << endl;
    break;

    case TY_UNION:
    case TY_STRUCT:
    case TY_U_STRUCT:
    case TY_U_UNION:
       value.vl_addr = daddr;
    break;

    case TY_FLOAT:
       rc = (dread_fpval( daddr, FALSE, FALSE, (char*)&value.vl_float )==-1);
    break;
    case TY_DOUBLE:
       rc = (dread_fpval( daddr, FALSE, TRUE, (char*)&value.vl_double )==-1);
    break;
    case TY_ENUM:
    case TY_U_ENUM:
       rc = (dread( daddr, (char *)&value.vl_int, sizeof(value.vl_int) )==-1);
      break;

    case TY_REAL:
       cout << "Variable::getValue():  TY_REAL not done yet." << ends;
       daddr = NULL;
       rc = 1;
    break;

    default:
       cout << "Type::getValue():  unknown type...." << getTypeCode() << endl;
       daddr = NULL;
       rc = 1;
   }

   return daddr;
}

// aggregate or enumeration defination class constructor.
//
aggr_or_enum_defst::aggr_or_enum_defst( const char *tag, typecode_t typecode, type_t *type )
{
   if (type != NULL)
   {
      type->setTypeCode( typecode );
      type->setSize( -1 );
      type->setBase( NULL );
```

```
      type->setTypeDef( NULL );
   }
   else
      type = new Type( typecode );

   type->setAggrOrEnumDef( this );

   ae_tag = tag;
   ae_is_complete = AE_INCOMPLETE;
   ae_size = -1;
   ae_type = type;
   ae_next = NULL;
   ae_lexinfo = NULL;
}


aggr_or_enum_def_t *
aggr_or_enum_defst::applyToAelist( aggr_or_enum_defst *(*func)(aggr_or_enum_defst *ae, const cha
                      const char *arg )
{
   aggr_or_enum_def_t *ae, *res;

   for(ae = this, res = NULL; ae != NULL && res == NULL; ae = ae->ae_next )
   {
      if (ae->ae_type != NULL)
         res = (*func) (ae, arg);
      else
         res = ae->ae_sublist->applyToAelist( func, arg );
   }

   return res;
}



//  Construct a bitfield type given an offset and a width.
//
Type *
Type::makeBitFieldType( typecode_t typecode, int bit_offset, int bit_width )
{
   bitfield_t *bf = new bitfieldst;

   bf->bf_code = typecode;
   bf->bf_offset = bit_offset;
   bf->bf_width = bit_width;

   Type *type = new Type( TY_BITFIELD );
   type->ty_bitfield = bf;

   return type;
}

Type *
Type::makePointerType( Type *base, qualifiers_t qualifiers )
{
   Type *type = new Type( DT_PTR_TO );
   type->ty_qualifiers = qualifiers;
   type->ty_base = base;
   return type;
}


enum_memberst::enum_memberst( const char *name, long val )
{
   em_name = name;
   em_val = val;
   em_lexinfo = NULL;
}

Type *
Type::codeToType( typecode_t typecode )
```

```
{
   // make sure typecode is in range
   if ((int) typecode < 0 || (int) typecode >= (int) TY_MAXTYPE)
      panic("bad typecode in ctt");

   if (ty_typetab[(int) typecode].ty_name == NULL)
   {
      const char     *name;
      typecode_t      typetab_code;

      typetab_code = typecode;
      switch (typecode)
      {
       case TY_VOID:
         name = "void";
         break;
       case TY_CHAR:
         name = "char";
         break;
       case TY_UCHAR:
         name = "unsigned char";
         break;
       case TY_SHORT:
         name = "short";
         break;
       case TY_USHORT:
         name = "unsigned short";
         break;
       case TY_INT:
         name = "int";
         break;
       case TY_UINT:
         name = "unsigned int";
         break;
       case TY_LONG:
         name = "long";
         break;
       case TY_ULONG:
         name = "unsigned long";
         break;
       case TY_FLOAT:
         name = "float";
         break;
       case TY_DOUBLE:
         name = "double";
         break;
       case TY_INT_ASSUMED:
         name = "(int assumed)";
         typetab_code = TY_INT;
         break;
       case TY_ELLIPSIS:
       case TY_SIGNED:
       case TY_UNSIGNED:
         name = "<internal type>";
         break;
       default:
         panic("bad typecode in ctt");
         name = UNSET;          /* to satisfy gcc */
         break;

      }
      ty_typetab[(int) typecode].ty_code = typetab_code;
      ty_typetab[(int) typecode].ty_size = -1;
      ty_typetab[(int) typecode].ty_base = NULL;
      ty_typetab[(int) typecode].ty_typedef = NULL;
      ty_typetab[(int) typecode].ty_name = name;
   }
   return &ty_typetab[(int) typecode];
}

static int
dynamic_type_size( Type *type, ilist_t *il )
```

```
{
   int             size;

   if (type->getSize() != -1)
      size = type->getSize();
   else if (type->getTypeCode() == DT_ARRAY_OF)
   {
      dim_t          *dim;
      int             basesize,
                      high,
                      low,
                      size_is_variable;

      dim = type->getDimension();
      size_is_variable = dim->di_ldynamic || dim->di_hdynamic;
      if (size_is_variable)
      {
         if (il == NULL || !il->il_low_known || !il->il_high_known)
            return UNKNOWN_SIZE;
         low = il->il_low;
         high = il->il_high;
      } else
      {
         low = dim->di_low;
         high = dim->di_high;
      }
      basesize = dynamic_type_size(type->getBase(),
                                   (il != NULL) ? il->il_next
                                   : (ilist_t *) NULL);

      if (basesize == UNKNOWN_SIZE)
         size = UNKNOWN_SIZE;
      else
         size = (high - low) * basesize;
      if (!size_is_variable)
         type->setSize( size );
   } else
      size = ci_typesize((lexinfo_t *) NULL, type);

   if (size <= 0 && size != UNKNOWN_SIZE)
      panic("non positive type size in typesize");

   return size;
}


int
Type::typeSize()
{
   return dynamic_type_size(this, (ilist_t *) NULL);

}
```

```
/*-------------------------------------------------------------------------
 |       FILE: variable.h
 |      CLASS: Variable
 |    PURPOSE: The Variable object describes a local or global variable.
 |             It stores the name, type, storage class, and address of the
 |             variable.  The address is either the absolute address,
 |             the offset from the frame pointer, the offset from the
 |             start of a structure or union or a register number,
 |             depending on the class of the variable.
 -------------------------------------------------------------------------*/

#ifndef VARIABLE_H_INCLUDED
#define VARIABLE_H_INCLUDED

#include "stack.h"
#include "data.h"

//  Class types

typedef enum classen
{
   CL_NOCLASS,                                  // out of band value
   CL_DECL,                                     // declaration only
   CL_AUTO,                                     // automatic variable
   CL_EXT,                                      // external symbol
   CL_STAT,                                     // static
   CL_REG,                                      // register variable
   CL_MOS,                                      // member of structure
   CL_ARG,                                      // function argument passed by value
   CL_REF,                                      // function argument passed by reference
   CL_MOU,                                      // member of union
   CL_MOE,                                      // member of enumeration
   CL_REGPARM,                                  // register parameter
   CL_LSTAT,                                    // local static
   CL_FUNC,                                     // function
   CL_LFUNC,                                    // static function
   CL_TYPEDEF,                                  // typedef
   CL_TAGNAME                                   // struct/union tag
} class_t;


class Variable
{
   friend void adjustLStatAddr( class Variable *v, char *c_delta );

 public:

   // use to be...ci_make_var(alloc_id, name, class, type, addr)
   Variable( const char *name, class_t c, Type *type, taddr_t addr );

   // get the dynamic address of the variable
   taddr_t getDynamicAddress( Frame *frame, taddr_t base );


   static Variable *field( STFile *stf, int *p_symno, const char **p_s,
                           int is_struct, Variable *next, bool eval );


   taddr_t getValue( Frame *frame, value_t& value, taddr_t base );

   language_t getLanguage()                     { return va_language; }
   void setLanguage( language_t l )             { va_language = l;  }
   Variable *getNext()                          { return va_next;   }
   void setNext( Variable *n )                  { va_next = n;      }
   class_t getClass()                           { return va_class;  }
   void setClass( class_t c )                   { va_class = c;     }
   const char *getName()                        { return va_name;   }
   void setName( const char *name )             { va_name = name;   }
   int getAddress()                             { return va_addr;   }
   void setAddress(int addr)                    { va_addr=addr;     }
   Type *getType()                              { return va_type;   }
```

```
   void setType( Type *t )                  { va_type = t;      }
   short getFlags()                         { return va_flags;  }
   void setFlags( short flags )             { va_flags = flags; }

 protected:
   const char           *va_name;                    // variable name
   class_t              va_class;                    // class of var (see C_ defines below)
   language_t           va_language;                 // language (C, FORTRAN etc)
   short                va_flags;                    // flags - see below
   class Type           *va_type;                    // variable type
   int                  va_addr;                     // variable address
   class Variable       *va_next;                    // next variable
   lexinfo_t            *va_lexinfo;
};

typedef class Variable var_t;

//  Flags describing a variable.

//      var has been assigned to
#define VA_SET                  0x0001
//      var is private to the C interpreter
#define VA_IS_CI_VAR            0x0002
//      var is initialised
#define VA_HAS_INITIALISER      0x0004
//      hide outer ptr type (see st_skim.C)
#define VA_HIDE_PTR             0x0008


#endif
```

```
/*-------------------------------------------------------------------------
 |       FILE: variable.C
 |    PURPOSE: Method for class variable.
 |             The Variable object describes a local or global variable.
 |             It stores the name, type, storage class, and address of the
 |             variable.  The address is either the absolute address,
 |             the offset from the frame pointer, the offset from the
 |             start of a structure or union or a register number,
 |             depending on the class of the variable.
 -------------------------------------------------------------------------*/
#include "symtab.h"

Variable::Variable(const char *name, class_t aclass, type_t * type, taddr_t addr)
{
   va_name = name;
   va_class = aclass;
   va_language = LANG_C;
   va_flags = 0;
   va_type = type;
   va_addr = (int)addr;
   va_next = NULL;
   va_lexinfo = NULL;
}

// friend function to class Variable
void
adjustLStatAddr( class Variable *v, char *c_delta )
{
   if( v->va_class==CL_LSTAT )
      v->va_addr += (int)c_delta;
}



   -------------------------------------------------------------------------
       METHOD: getDynamicAddress
 |
 |    PURPOSE: Returns the address of the variable.  This method finds
 |             the address of the variable...the location of this address
 |             depends on the class of the variable (local, extern, etc.)
 |             and the type.  If it is not extern, then the stack frame
 |             of the variable needs to be passed in.  Otherwise, just
 |             pass in NULL.
 -------------------------------------------------------------------------*/
taddr_t
Variable::getDynamicAddress( Frame *frame, taddr_t base )
{
   taddr_t addr;
   taddr_t frame_ptr = (frame) ? frame->fr_fp : (taddr_t)0;

   switch (va_class)
   {
    // out of band value
    case CL_REF:
      if( dread( frame_ptr + va_addr, (char *)&addr, sizeof(addr)) != 0 )
         addr = BAD_ADDR;
      break;

    // function argument passed by value
    case CL_ARG:
      addr = frame_ptr + va_addr;
      break;

    // automatic variable
    case CL_AUTO:
      addr = frame_ptr + va_addr;
      break;

    // member of union or member of structure
    case CL_MOS:
    case CL_MOU:
      addr = base + va_addr;
```

```
      break;

    // external symbol, static variable, or local static
    case CL_EXT:
    case CL_STAT:
    case CL_LSTAT:
      addr = va_addr;
      break;

    // register variable or register parameter
    case CL_REG:
    case CL_REGPARM:
      // !!! if this doesn't work...try stack->stk_next_inner
      frame->getRegisterAddress( va_addr );
      break;
    default:
      panic("unknown class in gda");
   }
   return addr;
}



/*-------------------------------------------------------------------------
 |     METHOD: getValue
 |
 |    PURPOSE: Given the stack frame (or NULL if var class is extern),
 |             this method fills in the value structure with the
 |             current value.  If successful, getValue() returns the
 |             address in the inferior process of the value read.
 |             Otherwise, it returns NULL.
 -------------------------------------------------------------------------*/
taddr_t
Variable::getValue( Frame *frame, value_t& value, taddr_t base )
{
   int rc=0;

   // get the true address of the variable
   taddr_t daddr = getDynamicAddress( frame, base );

   daddr = getType()->getValue( value, daddr );

   return daddr;
}
```