Intel® Debugger (IDB) Manual

Document Number: 306363-002US

Disclaimer and Legal Information

Table Of Contents

Introduction to Intel® Debugger (IDB)

The Intel® Debugger (IDB) is a component of Intel® compilers. It is a full-featured symbolic source code application debugger that helps programmers

- Debug programs
- Disassemble and examine machine code and examine machine register values
- Debug programs with shared libraries
- Debug multiple programs or processes
- Debug multithreaded applications

IDB provides debugging support for the following languages: C, C++ and Fortran (including Fortran 95/90). It also provides a choice of command-line or graphical user interface (GUI) under the Eclipse* platform.

IDB has two modes:

- dbx (default mode)
- gdb (optional mode)

IDB supports the Intel® C++ and Intel® Fortran Compilers.

IDB works on the following desktop and server platforms:

- IA-32 systems running Linux*, Windows*, and Mac OS*
- Systems using Intel® Extended Memory 64 Technology (Intel® EM64T) processors, running Linux and Windows
- Itanium® 2-based systems running Linux and Windows

Obtaining an Installation Kit

The Intel® Debugger is included in the Intel® C++ Compiler and Intel® Fortran Compiler. These Intel® Compilers are available for Linux, Windows, and Mac OS systems.

In addition, kits, documentation, and answers to frequently asked questions are available from the following sources:

- Intel compilers web pages (http://www.intel.com/software/products/compilers/)

- Intel® Developer Services forum (http://www.softwareforums.intel.com/ids/)
- Authorized retailers

About This Document

This manual describes using the Intel® Debugger on Linux* and Mac OS* systems, including preparing your program and starting the debugger, using debugger commands to accomplish various tasks, information about viewing data, viewing the call stack, controlling execution and locating problems, as well as detailed, advanced information.

Organization

The manual is organized as follows:

Part I contains a quick introduction to the Debugger

Part II contains information to help you make expert use of the Debugger

Part III contains advanced reference information

The appendixes contain the following information:

the Debugger variables

the Debugger aliases

the corefile_listobj.c example

the array navigation example

Intended Audience

This manual is intended for programmers who have a working knowledge of one of the programming languages that Intel IDB supports (C, C++, Fortran).

Notation Conventions

The following conventions are used in this manual:

Convention	**Meaning**
%	A percent sign represents the C shell system prompt.
#	A pound sign represents the default super-user prompt.
UPPERCASE lowercase	The operating system differentiates between lowercase and uppercase characters. On the operating system level, you must type examples, syntax descriptions, function definitions, and literal strings that appear in text exactly as shown.

Ctrl+C	This symbol indicates that you must press the Ctrl key while you simultaneously press another key (in this case, C).
`monospaced text`	This typeface indicates a routine, partition, pathname, directory, file, or non-terminal name. This typeface is also used in interactive examples.
`monospaced bold text`	In interactive examples, this typeface indicates input that you enter. In syntax statements and text, this typeface indicates the exact name of a command or keyword.
`monospaced italic text`	Monospaced italic type indicates variable values, place holders, and function argument names. In syntax definitions, monospaced italic text indicates non-terminal names. When a non-terminal name consists of more than one word, the words are joined using the underscore (_), for example, *breakpoint_command*.
italic text	Italic type indicates book names or emphasized terms.

`foo bar`
`    : item1`
`    | item2`
`    | item3`
	A colon (`:`) starts the syntax definition of a non-terminal name (in this example, *foo_bar*. Vertical bars (`|`) separating items that appear in syntax definitions indicate that you choose one item from among those listed.

`[ ]`	In syntax definitions, brackets indicate items that are optional.

`option ;...`
`option ,...`
`option  ...`
	A set of three horizontal ellipses indicates that you can enter additional parameters, options, or values. A semicolon, comma, or space preceding the ellipses indicates successive items must be separated by semicolons, commas, or spaces.

What's New in this Release

This release contains a number of changes, improvements, and new features. The following list describes the major 9.1 enhancements:

- Eclipse* plug-in is now available on the Intel® C++ Compiler for Linux* IA-32 and Itanium®-based systems. The Eclipse plug-in allows the user to choose the Intel® Debugger for debugging within the C/C++ Development Tools* (CDT)/Eclipse environment as described in Starting the Debugger Using Eclipse*.

- Remote debugging is now supported on Linux IA-32 systems as described in Debugging Remote Applications.

- The "%" unary operator is now available. The operator reverses all the bytes of its integer operand as described in % Unary Operator.

Note:

Please refer to the Release Notes for the most recent information about features implemented in this release.

Reporting Problems

For instructions about reporting problems, please see the Technical Support section of the compiler release notes.

What to Report

Please provide the following information when you enter your problem report. Doing so will make it easier for us to reproduce and analyze your problem. If you do not provide this information, we may have to ask you for it.

- A description of the problem. The clearer and more detailed the description, the easier it will be for us to reproduce and analyze your problem.
- A transcript of the debugger output. You can obtain this by using the `record io` debugger command or by using the `script(1)` system command.
- Operating system and version information. The output of `uname -a` is best.
- Version information. The version number is in the welcome banner that displays when you invoke the debugger. You can also obtain the version number by invoking the debugger with the `idb -V` command.
- The smallest source code example possible; build instructions; source languages, compiler versions, and so forth; and a pointer to a zip file containing sources or binaries that reproduce the problem. To obtain compiler versions, you can use the `-V` option if your compiler supports it (see the reference page for your compiler).
- The exact debugger commands that cause the problem to occur.
- Any other information that you think would be helpful.

The debugger development team can use `ftp` to fetch sources and executables if you can place them in an anonymous FTP area. If not, you may be asked to use another method.

Related Publications

The following documents contain related information:

- Man pages for the various compilers
- The release notes for the Intel® Debugger
- The Intel® Compiler documentation

Part I. A Quick Introduction to Using the Intel® Debugger

About Using the Intel® Debugger. A Quick Introduction

This section provides a quick introduction to the debugger. The user will learn how to

- Prepare a Program for Debugging
- Start the Debugger
- Enter Debugger Commands
- Script or Repeat Previous Commands
- Run the Program Under Debugger Control
- Pause the Process at the Problem
- Examine the Paused Process
- Continue Execution of the Process

Making Simple Use of the Debugger. Overview

The Intel® IDB supports DBX and GDB modes. By default, IDB operates like the DBX debugger. In the GDB mode, Intel IDB operates like the GNU* Debugger, GDB*. See the Starting the Debugger section to get to know how to launch the debugger in the required mode.

You look for a bug by doing the following:

1. Find a repeatable reproducer of the bug (the simpler the reproducer is, the easier the next steps will be).
2. Prepare your program for debugging.
3. Start the debugger.

Give commands to the debugger.

- Command the debugger to either
  - Prepare to create a process running the program, or
  - Attach to and interrupt a process that you created using normal operating system specific methods.
- Command the debugger to create breakpoints that will pause the process as close as possible to where the bug happened.
- If you are using the debugger to create the process, tell it to create the process now.

1. Do whatever it takes to reproduce the bug, so that the breakpoints will stop the process close to where the bug has caused something detectably wrong to happen.
2. Look around to determine the location of the bug:
   - If the bug is in the code where the debugger has stopped the process, exit the debugger and fix the bug.
   - If the bug has not happened yet, remove any breakpoints that are triggering too often, create other breakpoints that work better at locating the problem, and continue the process.
   - If the bug has already occurred, take the same steps of creating breakpoints and so on, but set one or more breakpoints earlier in the program before the error occurs. Rerun from an earlier position (a snapshot if you made one, or else the beginning of the program), and step through the program to determine the exact line causing the error.

Preparing a Program for Debugging. Simple Debugging

Compile and link your program using the `-g` option.

```
% icc -g tmp.c
```

For more information see also:

Preparing the Compiler and Linker Environment.

Starting the Debugger. Simple Debugging

Before you start the debugger, make sure that you have correctly set the size information for your terminal; otherwise, the debugger's command line editing support may act unpredictably. For example, if your terminal is 25x80, you may need to set the following:

```
% stty rows 25 ; setenv LINES 25
% stty cols 80 ; setenv COLS 80
```

There are four basic alternatives for running the debugger on a process (see examples below):

1. Have the debugger create the process using the shell command line to identify the executable to run. (dbx) (gdb)
2. Have the debugger create the process using the debugger commands to identify the executable to run. (dbx) (gdb)
3. Have the debugger attach to a running process using the shell command line to identify the process and the executable file that process is running. (dbx) (gdb)
4. Have the debugger attach to a running process using the debugger commands to identify the process and the executable file that process is running. (dbx) (gdb)

DBX Mode

Intel IDB starts operating in DBX mode by default, so you do not have to specify any special options in the shell command line.

Examples:

1. Creating the process using the shell command line.

```
 % idb a.out
Intel(R) Debugger for ..., Version ..., Build ...
------------------
object file name: a.out
Reading symbolic information ...done
(idb) stop in main
[#1: stop in int main(void) ]
(idb) run
```

2. Creating the process using the debugger commands.

```
% idb
Intel(R) Debugger for ..., Version ..., Build ...
------------------
(idb) load a.out
Reading symbolic information ...done
(idb) stop in main
[#1: stop in int main(void) ]
(idb) run
```

3. Attaching to a running process using the shell command line.

```
% ./a.out &
[1] 27859
% jobs
[1]+  Running                 ./a.out &
% idb a.out -pid 27859
Intel(R) Debugger for ..., Version ..., Build ...
------------------
Reading symbolic information ...done
Attached to process id 27859  ....
```

Press Ctrl+C to interrupt the process.

4. Attaching to the process using the debugger commands.

```
% ./a.out &
[1] 27859
% jobs
[1]+  Running                 ./a.out &
% idb
Intel(R) Debugger for ..., Version ..., Build ...
------------------
(idb) attach 27859 a.out
Reading symbolic information ...done
Attached to process id 27859  ....
```

Press Ctrl+C to interrupt the process.

GDB Mode

To start the debugger in the GDB mode, specify the `-gdb` option in the shell command line.

Examples:

1. Creating the process using the shell command line.

```
% idb -gdb a.out
Intel(R) Debugger for ..., Version ..., Build ...
------------------
object file name: a.out
Reading symbols from a.out...done
(idb) break main
Breakpoint 1 at 0x80484f6: file qwerty.c, line 9.
(idb) run
```

2. Creating the process using the debugger commands.

```
% idb -gdb
Intel(R) Debugger for ..., Version ..., Build ...
-------------------
(idb) file a.out
Reading symbols from a.out...done.
(idb) break main
Breakpoint 1 at 0x80484f6: file qwerty.c, line 9.
(idb) run
```

3. Attaching to a running process using the shell command line.

```
% ./a.out &
[1] 27859
% jobs
[1]+  Running                 ./a.out &
% idb -gdb a.out -pid 27859
Intel(R) Debugger for ..., Version ..., Build ...
------------------
object file name: a.out
Reading symbols from a.out...done.
Attached to process id 27859  ....
```

Press Ctrl+C to interrupt the process.

4. Attaching to the process using the debugger commands.

```
% ./a.out &
[1] 27859
% jobs
[1]+  Running                 ./a.out &
% idb -gdb
Intel(R) Debugger for ..., Version ..., Build ...
(idb) file a.out
Reading symbols from a.out...done.
```

```
(idb) attach 27859
Attached to process id 27859  ....
```

Press Ctrl+C to interrupt the process.

Note:

In the case of Fortran, routine `main` at which your program stops is not your main program unit. Rather, it is a main routine supplied by the Fortran system that performs some initialization and then calls your code. Just step forward using the **`step`** command a couple of times (probably twice) and you will soon step into your code.

Note:

If you want full compatibility with GDB output, you need to set the debugger variable `$gdb_compatible_output` to 1. Otherwise, IDB will produce the extended GDB output in some cases.

See Attaching to a Running Program for the information about how to obtain the PID of a program.

Entering Debugger Commands

The debugger issues a prompt when it is ready for the next command from the terminal:

```
(idb) you type here
```

When you enter commands, you can use the left and right arrow keys to move within the line and the up and down arrow keys to recall previous commands for editing. When you finish entering a command, press the Enter key to submit the completed line to the debugger for processing.

Typically, you enter one debugger command on one command line. However, if a debugger command requires typing on multiple lines, type a backslash (\) character at the end of the line to be continued.

To re-execute the last valid command, press Enter without typing any characters.

Following are two very useful commands available in both modes:

```
(idb) help
(idb) quit
```

Scripting or Repeating Previous Commands. Simple Debugging

DBX Mode

To execute debugger commands from a script, use the **`source`** command as follows:

```
(idb) source filename
```

The **`source`** command causes the debugger to read and execute debugger commands from the file named *`filename`*.

GDB Mode

The **`source`** command is not yet available in the GDB mode.

Context for Executing Commands

Although the debugger supports concurrently debugging multiple processes, it operates only on a single process at a time, known as the current process.

Processes contain one or more threads of execution. The threads execute functions. Functions are sequences of instructions that come from source lines within source files.

As you enter debugger commands to manipulate your process, it would be very tedious to have to repeatedly specify which thread, source file, and so on you wish the command to be applied to. To prevent this, each time the debugger stops the process, it re-establishes a static context and a dynamic context for your commands. The components of the static context are independent of this run of your program; the components of the dynamic context are dependent on this run.

The static context consists of the following:

- A current program
    - A current file
    - A current line
    - A current column, if the application is compiled with column information emitted. See Showing Column Information for details.
- The dynamic context consists of the following:
    - A current call frame
    - A current thread
    - The particular thread executing the event that caused the debugger to gain control of the process

You can change most of these individually to point to other instances, as described in the relevant portions of this manual, and the debugger will modify the rest of the static and dynamic context to keep the various components consistent.

Running a Program Under Debugger Control

You can tell the debugger to create a process or to attach to an existing process.

After you specify the program (either on the shell command line or by using the **`load`** (dbx) or **`file`** (gdb) command), but before you have requested the debugger to create the process, you can still do things that seem to require a running process; for example, you can create breakpoints and examine sources. Any breakpoints that you create will be inserted into the process as soon as possible after it executes your program.

To have the debugger create a process (rather than attach to an existing process), you request it to run, specifying, if necessary, any arguments and input and output redirection as follows:

```
% idb a.out
Intel(R) Debugger for ..., Version ..., Build ...
...Preparing the Compiler and Linker Environment>
(idb) run
```

or

```
(idb) run arguments
```

or

```
(idb) run arguments > output-file
```

or

```
(idb) run arguments < input-file > output-file
```

The result of using any of the preceding command variations is similar to having attached to a running process.

DBX Mode

The **`rerun`** command repeats the previous **`run`** command with the same arguments and file redirection.

GDB Mode

The **`run`** command without arguments repeats the previous run (with the same arguments, input and output redirections).

`r` is a shortcut for the **`run`** command.

Pausing the Process at the Problem

Following are the most common ways to pause a process:

- Press Ctrl+C. (dbx) (gdb)
- Wait until the process raises a signal. (dbx) (gdb)
- Create a breakpoint before you run or continue the process. (dbx) (gdb)
- Create a watchpoint before you run or continue the process. (dbx) (gdb)

DBX Mode

1. Pres Ctrl+C.

```
(idb) run
^C
Interrupt (for process)
Stopping process localhost:27903 (a.out).
Thread received signal INT
stopped at [int main(int):5 0x120001138]
      5      while (argc < 2 && i < 10000000)
```

2. Wait until the process raises a signal.

3. Create a breakpoint before running or continuing the process.

```
(idb) stop in main
[#1: stop in int main(void)]
(idb) run
[1] stopped at [int main(void):182 0x08052e8f]
    182      List<Node> nodeList;
```

4. Create a watchpoint before running or continuing the process.

```
(idb) watch variable nodeList. firstNode write
[#2: watch variable nodeList. firstNode write]
(idb) cont
[2] Address 0xbfffd279 was accessed at:
void List<Node>::append(class Node* const): src/x list.cxx
 [line 149, 0x0804c5ed] append(class Node* const)+0x15:
                movl     %edx, (%eax)
0xbfffd278: Old value = 0x000000000805e600
0xbfffd278: New value = 0x000000000805e600
[2] stopped at [void List<Node>::append(class Node* const):149
0x0804c5ef]
    149         _firstNode = node;
```

GDB Mode

1. Press Ctrl+C.

```
(idb) run
^C
Interrupt (for process)
```

```
Stopping process localhost:27903 (a.out).
Thread received signal INT
main(argc=1) at x whatHappensOnControlC.cxx: 5
5     while (argc < 2 && i < 10000000)
```

2. Wait until the process raises a signal.

```
(idb) run
Starting program: /home/user/examples/x segv
Program received signal SIGSEGV
buggy (input=0xbfffdf37 "/home/user/examples/x segv", output=0x0) at
src/x segv.cxx:13
13         output[k] = input[k];
```

3. Create a breakpoint before running or continuing the process.

```
(idb) break main
Breakpoint 1 at 0x8052e8f: file src/x list.cxx, line 182.
(idb) run
Starting program: /home/user/examples/x list
Breakpoint 1, main () at src/x list.cxx:182
182     List<Node> nodeList;
```

4. Creating a watchpoint before running or continuing the process. Following are the most common ways to pause a process:

```
(idb) watch nodeList. firstNode
Hardware watchpoint 2: nodeList. firstNode
(idb) continue
Continuing.
Old value = (Node *) 0x0
New value = (Node *) 0x805e600
Hardware watchpoint 2: nodeList. firstNode
List<Node>::append (node=0x805e600) at src/x list.cxx:149
149         _firstNode = node;
```

Examining the Paused Process

This section gives general information how to examine components of the paused process looking at

- the source files
- the threads
- the call stack
- the data
- the signal state

- the generated code

For the detailed information, see

Looking Around at the Code, the Data, and Other Process Information

Looking at the Source Files. Simple Debugging

You can perform the following operations on source files:

- Tell the debugger where your sources are, if it cannot find them.
- Find out the name of the current source file.
- Switch to a different source file.
- List lines in a source file.
- Search within a source file.

DBX Mode

Following is an example that shows listing lines and using the `/` command to search for a string:

```
(idb) file
src/x list.cxx
(idb) list 180:10
    180 main()
    181 {
    182     List<Node> nodeList;
    183
    184     // add entries to list
    185     //
>   186     IntNode* newNode = new IntNode(1);
    187     nodeList.append(newNode);   {static int somethingToReturnTo;
somethingToReturnTo++; }
    188
    189     CompoundNode* cNode = new CompoundNode(12.345, 2);
(idb) /CompoundNode
    192     CompoundNode* cNode1 = new CompoundNode(3.1415, 7)
```

Aliases are shorthand forms of longer commands. This example shows using the **W** alias, which lists up to 20 lines around the current line. Note that a right bracket (>) marks the current line.

```
(idb) alias W
W       list $curline - 10:20
(idb) W
    176
    177
    178 //  The driver for this test
    179 //
    180 main()
    181 {
    182     List<Node> nodeList;
    183
    184     // add entries to list
    185     //
```

```
>   186     IntNode* newNode = new IntNode(1);
    187     nodeList.append(newNode);
    188
    189     CompoundNode* cNode = new CompoundNode(12.345, 2);
    190     nodeList.append(cNode);
    191
    192     nodeList.append(new IntNode(3));
    193
    194     IntNode* newNode2 = new IntNode(4);
    195     nodeList.append(newNode2);
```

GDB Mode

Use **`info source`**, **`info line`**, and **`list`** commands for looking at source files:

```
(idb) info source
Current source file is src/x_list.cxx
(idb) list 180,+10
180 main()
181 {
182     List<Node> nodeList;
183
184     // add entries to list
185     //
186     IntNode* newNode = new IntNode(1);
187     nodeList.append(newNode);   {static int somethingToReturnTo;
somethingToReturnTo++; }
188
189     CompoundNode* cNode = new CompoundNode(12.345, 2);
(idb) forward-search CompoundNode
192     CompoundNode* cNode1 = new CompoundNode(3.1415, 7);
```

Looking at the Threads. Simple Debugging

In a multithreaded application, you can obtain information about the thread that stopped or about all the threads, and you can then change the context to look more closely at a different thread. Note that a right bracket (>) marks the current thread. And the asterisk (*) marks the thread with return status.

DBX Mode

```
(idb) thread
  Thread Name                       State           Substate    Policy
      Pri
  ------ ------------------------- --------------- ----------- ------------ --
-
>*     1 default thread            running VP 3                SCHED OTHER  19
(idb) show thread
  Thread Name                       State           Substate    Policy
      Pri
  ------ ------------------------- --------------- ----------- ------------ --
-
>*     1 default thread            running VP 3                SCHED OTHER  19
      -1 manager thread            blk SCS                     SCHED RR     19
      -2 null thread for slot 0    running VP 1                null thread  -1
      -3 null thread for slot 1    ready VP 3                  null thread  -1
      -4 null thread for slot 2     new             new         null thread   -1
      -5 null thread for slot 3     new             new         null thread   -1
       2 threads(0x140000798)       blocked         cond 3      SCHED_OTHER   19
       3 threads+8(0x1400007a0)     blocked         cond 3      SCHED_OTHER   19
       4 threads+16(0x1400007a8)    blocked         cond 3      SCHED_OTHER   19
       5 threads+24(0x1400007b0)    blocked         cond 3      SCHED_OTHER   19
       6 threads+32(0x1400007b8)    blocked         cond 3      SCHED_OTHER   19
```

You can select any thread to be the focus of commands that show things. For example:

```
(idb) thread 2
  Thread Name                        State           Substate    Policy
      Pri
  ------ ------------------------- --------------- ----------- ------------ --
-
>      2 threads(0x140000798)       blocked         cond 3      SCHED_OTHER   19
```

GDB Mode

Within the GDB mode, you can have a look at a particular thread by specifying the internal debugger thread number. The asterisk (*) marks the current thread. Or you can observe all threads while your program is running:

```
(idb) thread
   ID           State
>* 8            stopped
(idb) show thread
   ID           State
   1            stopped
   2            stopped
   3            stopped
   4            stopped
   5            stopped
   6            stopped
   7            stopped
>* 8            stopped
```

This command provides the following information about known threads:

- **Number**: Number of a thread from the debugger point of view. This number is not used for numbering again if a thread dies.
- **Thread TID**: Thread identifier. TID is an identifier (unique) assigned to each thread inside the thread library.

- **LWP PID**: Light weight process identifier (unique) assigned by the Linux kernel to each process in the system. Architecturally, `LinuxThreads` has a PID for each thread in a multithreaded application.

- **Location**: Location where the thread stopped. Its output is very similar to the backtrace location field.

You can select any thread to be the focus of commands that show things. For example:

```
(idb) thread 2
   ID           State
>  2            stopped
```

Looking at the Call Stack. Simple Debugging

You can examine the call stack of any thread. Even if you are not using threads explicitly, your process will have one thread running your code. You can move up and down the stack, and examine the source being executed at each call.

DBX Mode

```
(idb) where 3
>0  0x080535c4 in ((IntNode*)0x805e600)->IntNode::printNodeData()
"src/x list.cxx":94
#1  0x0804c6f7 in ((List<Node>*)0xbfffb068)->List<Node>::print()
"src/x list.cxx":168
#2  0x08053376 in main() "src/x list.cxx":203
(idb) up 2
>2  0x08053376 in main() "src/x list.cxx":203
    203     nodeList.print();
(idb) list $curline - 3:5
    200     CompoundNode* cNode2 = new CompoundNode(10.123, 5);
    201     nodeList.append(cNode2); {static int somethingToReturnTo;
somethingToReturnTo++; }
    202
>   203     nodeList.print();
    204 }
(idb) down 1
>1  0x0804c6f7 in ((List<Node>*)0xbfffb068)->List<Node>::print()
"src/x list.cxx":168
    168         currentNode->printNodeData();
```

GDB Mode

```
(idb) backtrace 3
#0  0x080535c4 in IntNode::printNodeData () at src/x list.cxx:94
#1  0x0804c6f7 in List<Node>::print () at src/x list.cxx:168
#2  0x08053376 in main () at src/x list.cxx:203
(idb) up 2
#2  0x08053376 in main () at src/x list.cxx:203
203     nodeList.print();
(idb) list 200,+5
200     CompoundNode* cNode2 = new CompoundNode(10.123, 5);
201     nodeList.append(cNode2); {static int somethingToReturnTo;
somethingToReturnTo++; }
202
203     nodeList.print();
204 }
(idb) down 1
```

```
#1  0x0804c6f7 in List<Node>::print () at src/x list.cxx:168
168         currentNode->printNodeData();
```

Looking at the Data. Simple Debugging

You can look at variables and evaluate expressions involving them by using the **`print`** command.

DBX Mode

```
(idb) print fdata
12.3450003
(idb) print idata
2
(idb) print idata + 59
61
(idb) print this
0x805e610
(idb) print *this
class CompoundNode {
   fdata = 12.3450003;
   data = 2;                      // class IntNode
   nextNode = 0x0;                // class IntNode::Node
}
```

GDB Mode

```
(idb) print fdata
$2 = 12.345
(idb) print idata
$3 = 2
(idb) print idata + 59
$4 = 61
(idb) print this
$5 = (CompoundNode *) 0x805e610
(idb) print *this
$6 = {<IntNode> = {<Node> = {_nextNode = 0x0}, _data = 2}, _fdata = 12.345}
```

A synonym of the **`print`** command is the **`inspect`** command. The shortcut of the **`print`** command is **`p`**.

Looking at the Signal State

The debugger shows you the signal that stopped the thread.

DBX Mode

```
(idb) run
Thread received signal SEGV
stopped at [void buggy(char*, char*):13 0x080487b8]
     13         output[k] = input[k];
```

GDB Mode

```
(idb) run
Starting program: /home/user/examples/x segv
Program received signal SIGSEGV
buggy (input=0xbfffbf37 "/home/user/examples/x segv", output=0x0) at
src/x segv.cxx:13
13         output[k] = input[k];
```

Looking at the Generated Code. Simple Debugging

You can print memory as instructions or as data.

DBX Mode

In the following example, the **`wi`** alias lists machine instructions before and after the current instruction. Note that the asterisk (`*`) marks the current instruction.

```
(idb) alias wi
wi ($curpc - 20)/10 i
(idb) wi
CompoundNode::CompoundNode(float, int): x list.cxx
 [line 105, 0x120002348] cpys $f17,$f17,$f0
 [line 105, 0x12000234c] bis r31, r18, r8
 [line 101, 0x120002350] bis r31, r19, r16
 [line 101, 0x120002354] bis r31, r8, r17
 [line 101, 0x120002358] bsr r26, IntNode::IntNode(int)
*[line 101, 0x12000235c] ldq r18, -32712(gp)
 [line 101, 0x120002360] lda r18, 48(r18)
 [line 101, 0x120002364] stq r18, 8(r19)
 [line 101, 0x120002368] sts $f0, 24(r19)
 [line 106, 0x12000236c] bis r31, r19, r0
(idb) $pc/10x
0x12000235c: 0x8038 0xa65d 0x0030 0x2252 0x0008 0xb653 0x0018 0x9813
0x12000236c: 0x0400 0x47f3
(idb) $pc/6xx
0x12000235c: 0xa65d8038 0x22520030 0xb6530008 0x98130018
0x12000236c: 0x47f30400 0x47f5041a
(idb) $pc/2X
0x12000235c: 0x22520030a65d8038 0x98130018b6530008
```

GDB Mode

Use the **`x`** command to dump memory in various formats. The **`disassemble`** command also provides disassembling capability.

```
(idb) x /10i $pc
0x08052e8f <main+27>:                    pushl    %edi
0x08052e90 <main+28>:                    leal     -160(%ebp), %eax
0x08052e96 <main+34>:                    movl     %eax, (%esp)
0x08052e99 <main+37>:                    call     0x0804c4c8
< ZN4ListI4NodeEC1Ev>
0x08052e9e <main+42>:                    addl     $0x4, %esp
0x08052ea1 <main+45>:                    movl     $0x0, -156(%ebp)
```

```
0x08052eab <main+55>:                  pushl    %edi
0x08052eac <main+56>:                  movl     $0xc, (%esp)
0x08052eb3 <main+63>:                  call     0x0804c308 < init+744>
0x08052eb8 <main+68>:                  addl     $0x4, %esp
(idb) x /10xh $pc
0x8052e8f <main+27>: 0x8d57 0x6085 0xffff 0x89ff 0x2404 0x2ae8 0xff96 0x83ff
0x8052e9f <main+43>: 0x04c4 0x85c7
(idb) x /6xw $pc
0x8052e8f <main+27>: 0x60858d57 0x89ffffff 0x2ae82404 0x83ffff96
0x8052e9f <main+43>: 0x85c704c4 0xffffff64
(idb) x /2xg $pc
0x8052e8f <main+27>: 0x89ffffff60858d57 0x83ffff962ae824044
```

To examine individual registers, use the **`print`** command with the register name prepended with the dollar sign (`$`). Commands showing all (or a subset of) the registers are specific for the mode; see examples below.

Looking at the Registers

DBX Mode

To look at all the registers, use the **`printregs`** command. For example:

```
(idb) print $sp
-1073766620
(idb) printx $sp
0xbfff9f24
(idb) printregs
$eax            0x1 1
$ecx            0xbfffa09c -1073766244
$edx            0xbfffa020 -1073766368
$ebx            0xb72dbd98 -1221739112
$esp [$sp]      0xbfff9f24 -1073766620
$ebp [$fp]      0xbfffa008 -1073766392
$esi            0xbfffa094 -1073766252
$edi            0xb72d967c -1221749124
$eip [$pc]      0x8052e8f 134557327
$eflags         0x286 646
$cs             0x23 35
$ss             0x2b 43
$ds             0x2b 43
$es             0x2b 43
$fs             0x0 0
$gs             0x33 51
$orig eax       0xffffffff -1
$fctrl          0x37f 895
$fstat          0x0 0
$ftag           0x0 0
$fiseg          0x0 0
$fioff          0x0 0
$foseg          0x0 0
$fooff          0x0 0
$fop            0x0 0
$f0             0x0 0
$f1             0x0 0
$f2             0x0 0
$f3             0x0 0
```

```
$f4             0x0 0
$f5             0x0 0
$f6             0x0 0
$f7             0x0 0
$xmm0           0x0 union {
  v4 float = [0] = 0,[1] = 0,[2] = 0,[3] = 0;
  v2 double = [0] = 0,[1] = 0;
  v16 int8 =  <repeats 15 times>0,[15] = 0;
  v8 int16 = [0] = 0,[1] = 0,[2] = 0,[3] = 0,[4] = 0,[5] = 0,[6] = 0,[7] = 0;
  v4 int32 = [0] = 0,[1] = 0,[2] = 0,[3] = 0;
  v2 int64 = [0] = 0,[1] = 0;
}
$xmm1           0x0 union {
  v4 float = [0] = 0,[1] = 0,[2] = 0,[3] = 0;
  v2 double = [0] = 0,[1] = 0;
  v16 int8 =  <repeats 15 times>0,[15] = 0;
  v8 int16 = [0] = 0,[1] = 0,[2] = 0,[3] = 0,[4] = 0,[5] = 0,[6] = 0,[7] = 0;
  v4 int32 = [0] = 0,[1] = 0,[2] = 0,[3] = 0;
  v2 int64 = [0] = 0,[1] = 0;
}
$xmm2           0x0 union {
  v4 float = [0] = 0,[1] = 0,[2] = 0,[3] = 0;
  v2 double = [0] = 0,[1] = 0;
  v16 int8 =  <repeats 15 times>0,[15] = 0;
  v8 int16 = [0] = 0,[1] = 0,[2] = 0,[3] = 0,[4] = 0,[5] = 0,[6] = 0,[7] = 0;
  v4 int32 = [0] = 0,[1] = 0,[2] = 0,[3] = 0;
  v2 int64 = [0] = 0,[1] = 0;
}
$xmm3           0x0 union {
  v4 float = [0] = 0,[1] = 0,[2] = 0,[3] = 0;
  v2 double = [0] = 0,[1] = 0;
  v16 int8 =  <repeats 15 times>0,[15] = 0;
  v8 int16 = [0] = 0,[1] = 0,[2] = 0,[3] = 0,[4] = 0,[5] = 0,[6] = 0,[7] = 0;
  v4 int32 = [0] = 0,[1] = 0,[2] = 0,[3] = 0;
  v2 int64 = [0] = 0,[1] = 0;
}
$xmm4           0x0 union {
  v4 float = [0] = 0,[1] = 0,[2] = 0,[3] = 0;
  v2 double = [0] = 0,[1] = 0;
  v16 int8 =  <repeats 15 times>0,[15] = 0;
  v8 int16 = [0] = 0,[1] = 0,[2] = 0,[3] = 0,[4] = 0,[5] = 0,[6] = 0,[7] = 0;
  v4 int32 = [0] = 0,[1] = 0,[2] = 0,[3] = 0;
  v2 int64 = [0] = 0,[1] = 0;
}
$xmm5           0x0 union {
  v4 float = [0] = 0,[1] = 0,[2] = 0,[3] = 0;
  v2 double = [0] = 0,[1] = 0;
  v16 int8 =  <repeats 15 times>0,[15] = 0;
  v8 int16 = [0] = 0,[1] = 0,[2] = 0,[3] = 0,[4] = 0,[5] = 0,[6] = 0,[7] = 0;
  v4 int32 = [0] = 0,[1] = 0,[2] = 0,[3] = 0;
  v2 int64 = [0] = 0,[1] = 0;
}
$xmm6           0x0 union {
  v4 float = [0] = 0,[1] = 0,[2] = 0,[3] = 0;
  v2 double = [0] = 0,[1] = 0;
  v16 int8 =  <repeats 15 times>0,[15] = 0;
  v8 int16 = [0] = 0,[1] = 0,[2] = 0,[3] = 0,[4] = 0,[5] = 0,[6] = 0,[7] = 0;
  v4 int32 = [0] = 0,[1] = 0,[2] = 0,[3] = 0;
  v2 int64 = [0] = 0,[1] = 0;
}
$xmm7           0x0 union {
  v4 float = [0] = 0,[1] = 0,[2] = 0,[3] = 0;
  v2 double = [0] = 0,[1] = 0;
  v16_int8 =  <repeats 15 times>0,[15] = 0;
  v8 int16 = [0] = 0,[1] = 0,[2] = 0,[3] = 0,[4] = 0,[5] = 0,[6] = 0,[7] = 0;
  v4 int32 = [0] = 0,[1] = 0,[2] = 0,[3] = 0;
  v2 int64 = [0] = 0,[1] = 0;
}
$mxcsr          0x1f80 8064
$vfp            0xbfffa010 -1073766384
```

GDB Mode

The following commands allow you to examine sets of registers:

`info registers`

For example:

```
(idb) print $sp
$14 = (void *) 0xbfff8684
(idb) print /x $sp
$15 = 0xbfff8684
(idb) info registers
$eax            0x1 1
$ecx            0xbfff87fc -1073772548
$edx            0xbfff8780 -1073772672
$ebx            0xb72dbd98 -1221739112
$esp [$sp]      0xbfff8684 (void *) 0xbfff8684
$ebp [$fp]      0xbfff8768 (void *) 0xbfff8768
$esi            0xbfff87f4 -1073772556
$edi            0xb72d967c -1221749124
$eip [$pc]      0x8052e8f (void *) 0x8052e8f
$eflags         0x286 646
$cs             0x23 35
$ss             0x2b 43
$ds             0x2b 43
$es             0x2b 43
$fs             0x0 0
$gs             0x33 51
$orig eax       0xffffffff -1
$fctrl          0x37f 895
$fstat          0x0 0
$ftag           0x0 0
$fiseg          0x0 0
$fioff          0x0 0
$foseg          0x0 0
$fooff          0x0 0
$fop            0x0 0
```

Continuing Execution of the Process. Simple Debugging

When you have finished examining the current state of the process, you can move the process forward and see what happens. The following table shows the aliases and commands you can use to do this.

Desired Behavior	Command	Alias	Can Take Repeat Cont

Continue until another interesting thing happens	**`cont`**	**`c`**	Yes*
Single step by line, but step over calls	**`next`**	**`n`**	Yes
Single step to a new line, stepping into calls	**`step`**	**`s`**	Yes
Continue until control returns to the caller	**`return`** (dbx), **`finish`** (gdb)	None	No
Single step by instruction, over calls	**`nexti`**	**`ni`**	Yes
Single step by instruction, into calls	**`stepi`**	**`si`**	Yes

***** For the **`cont`** command, in GDB mode repeat count specifies the number of times to ignore a breakpoint. For the other commands repeat count has the same meaning in both modes.

The following examples demonstrate stepping through lines of source code (dbx) (gdb) and stepping at the instruction level (dbx) (gdb).

DBX Mode

Stepping through lines of source code:

```
(idb) list $curline - 10:20
    172
    173     if (i == 1) cout << "The list is empty ";
    174     cout << endl << endl;
    175 }
    176
    177
    178 //  The driver for this test
    179 //
    180 main()
    181 {
>   182     List<Node> nodeList;
    183
    184     // add entries to list
    185     //
    186     IntNode* newNode = new IntNode(1);
    187     nodeList.append(newNode);   {static int somethingToReturnTo;
somethingToReturnTo++; }
    188
    189     CompoundNode* cNode = new CompoundNode(12.345, 2);
    190     nodeList.append(cNode); {static int somethingToReturnTo;
somethingToReturnTo++; }
    191
(idb) next
stopped at [int main(void):186 0x08052ea1]
    186     IntNode* newNode = new IntNode(1);
(idb) next 3
stopped at [int main(void):190 0x0805301a]
    190     nodeList.append(cNode); {static int somethingToReturnTo;
somethingToReturnTo++; }
(idb) step
stopped at [void List<Node>::append(class Node* const):148 0x0804c5de]
```

```
    148     if (! firstNode)
(idb) list $curline - 2:6
    146 {
    147
>   148     if (!_firstNode)
    149          firstNode = node;
    150     else {
    151         Node* currentNode = _firstNode;
(idb) next
stopped at [void List<Node>::append(class Node* const):151 0x0804c5f1]
    151         Node* currentNode =  firstNode;
(idb) list $curline - 2:5
    149          firstNode = node;
    150     else {
>   151         Node* currentNode =  firstNode;
    152         while (currentNode->getNextNode())
    153             currentNode = currentNode->getNextNode();
(idb) return
stopped at [int main(void):190 0x08053032]
    190     nodeList.append(cNode); {static int somethingToReturnTo;
somethingToReturnTo++; }
(idb) next 2
stopped at [int main(void):193 0x080530ed]
    193     nodeList.append(cNode1); {static int somethingToReturnTo;
somethingToReturnTo++; }
```

Stepping at the instruction level:

```
(idb) $pc/2i
int main(void): src/x list.cxx
*[line 193, 0x08053100] main+0x28c:                   call     append(class
Node* const)
 [line 193, 0x08053105] main+0x291:                   addl     $0x8, %esp
(idb) nexti
stopped at [int main(void):193 0x08053105] main+0x291:                   addl
    $0x8, %esp
(idb) $pc/1i
int main(void): src/x list.cxx
*[line 193, 0x08053105] main+0x291:                   addl     $0x8, %esp
(idb) rerun
Process has exited
[3] stopped at [int main(void):193 0x08053100]
    193     nodeList.append(cNode1); {static int somethingToReturnTo;
somethingToReturnTo++; }
(idb) $pc/2i
int main(void): src/x list.cxx
*[line 193, 0x08053100] main+0x28c:                   call     append(class
Node* const)
 [line 193, 0x08053105] main+0x291:                   addl     $0x8, %esp
(idb) stepi
stopped at [void List<Node>::append(class Node* const):146 0x0804c5d8]
append(class Node* const):                  pushl    %ebp
(idb) $pc/1i
void List<Node>::append(class Node* const): src/x list.cxx
*[line 146, 0x0804c5d8] append(class Node* const):                pushl
   %ebp
```

GDB Mode

Stepping through lines of source code:

```
(idb) list
172
173     if (i == 1) cout << "The list is empty ";
174     cout << endl << endl;
175 }
176
177
178 //  The driver for this test
179 //
180 main()
181 {
182     List<Node> nodeList;
183
184     // add entries to list
185     //
186     IntNode* newNode = new IntNode(1);
187     nodeList.append(newNode);   {static int somethingToReturnTo;
somethingToReturnTo++; }
188
189     CompoundNode* cNode = new CompoundNode(12.345, 2);
190     nodeList.append(cNode); {static int somethingToReturnTo;
somethingToReturnTo++; }
191
(idb) next
186     IntNode* newNode = new IntNode(1);
(idb) next 3
190     nodeList.append(cNode); {static int somethingToReturnTo;
somethingToReturnTo++; }
(idb) step
List<Node>::append (node=(Node *) 0x805e610) at src/x list.cxx:148
148     if (! firstNode)
(idb) list -2,+6
146 {
147
148     if (! firstNode)
149          firstNode = node;
150     else {
151         Node* currentNode = _firstNode;
(idb) step
151         Node* currentNode =  firstNode;
(idb) list -2,+5
149          firstNode = node;
150     else {
151         Node* currentNode =  firstNode;
152         while (currentNode->getNextNode())
153             currentNode = currentNode->getNextNode();
(idb) finish
main () at src/x list.cxx:190
190     nodeList.append(cNode); {static int somethingToReturnTo;
somethingToReturnTo++; }
(idb) next 2
193     nodeList.append(cNode1); {static int somethingToReturnTo;
somethingToReturnTo++; }
```

Stepping at the instruction level:

```
(idb) x /2i $pc
0x0804c5d8 <append>:                pushl    %ebp
0x0804c5d9 <append+1>:                movl     %esp, %ebp
(idb) nexti
146 {
(idb) x /1i $pc
0x0804c5d9 <append+1>:                movl     %esp, %ebp
(idb) run
Program exited normally.
Starting program: /home/user/examples/x list
Breakpoint 3, main () at src/x list.cxx:193
193     nodeList.append(cNode1); {static int somethingToReturnTo;
somethingToReturnTo++; }
(idb) x /2i $pc
0x08053100 <main+652>:                call     0x0804c5d8 <append>
0x08053105 <main+657>:                addl     $0x8, %esp
(idb) stepi
List<Node>::append (node=(Node *) 0xbfffc864) at src/x_list.cxx:146
146 {
(idb) x /1i $pc
0x0804c5d8 <append>:                pushl    %ebp
```

Snapshots as an Undo Mechanism

Often when you move the process forward, you accidentally go too far. For example, you may step over a call that you should have stepped into.

In a program that does not use multiple threads, you can use snapshots to save your state before you step over the call. Then clone that snapshot to position another process just before the call so you can step into it.

The following example shows the stages of a snapshot being used in this way:

1. The first stage is to build the program and start debugging.
2. The next stage is to stop the process just before the call and take a snapshot. You can see you are just before the call because the right bracket (>) to the left of the source list shows the line about to be executed.

```
(idb) next 2
stopped at [int main(void):187 0x1200024b8]
    187     nodeList.append(newNode);
(idb) list $curline - 10:20
    177
    178 //  The driver for this test
    179 //
    180 main()
    181 {
    182     List<Node> nodeList;
    183
    184     // add entries to list
    185     //
    186     IntNode* newNode = new IntNode(1);
>   187     nodeList.append(newNode);
    188
    189     CompoundNode* cNode = new CompoundNode(12.345, 2);
    190     nodeList.append(cNode);
    191
    192     CompoundNode* cNode1 = new CompoundNode(3.1415, 7);
    193     nodeList.append(cNode1);
    194
    195     nodeList.append(new IntNode(3));
    196
(idb) save snapshot
# 1 saved at 08:41:46 (PID: 1012).
    stopped at [int main(void):187 0x1200024b8]
    187     nodeList.append(newNode);
```

3. You now step over the call. The execution is now after the call, shown by the right bracket (>) being on the following source line.

```
(idb) next
stopped at [int main(void):189 0x1200024d0]
    189     CompoundNode* cNode = new CompoundNode(12.345, 2);
(idb) list $curline - 10:20
    179 //
    180 main()
    181 {
    182     List<Node> nodeList;
    183
    184     // add entries to list
    185     //
    186     IntNode* newNode = new IntNode(1);
    187     nodeList.append(newNode);
    188
>   189     CompoundNode* cNode = new CompoundNode(12.345, 2);
    190     nodeList.append(cNode);
    191
    192     CompoundNode* cNode1 = new CompoundNode(3.1415, 7);
    193     nodeList.append(cNode1);
    194
    195     nodeList.append(new IntNode(3));
    196
    197     IntNode* newNode2 = new IntNode(4);
    198     nodeList.append(newNode2);
```

4. Oh, how you wish you hadn't done that! No problem, just clone that snapshot you made.

```
(idb) clone snapshot
Process has exited
Process 1009 cloned from Snapshot 1.
# 1 saved at 08:41:46 (PID: 1012).
    stopped at [int main(void):187 0x1200024b8]
    187     nodeList.append(newNode);
```

5. Now you are in a new process before the call is executed.

```
(idb) list $curline - 10:20
    177
    178 //  The driver for this test
    179 //
    180 main()
    181 {
    182     List<Node> nodeList;
    183
    184     // add entries to list
    185     //
    186     IntNode* newNode = new IntNode(1);
>   187     nodeList.append(newNode);
```

```

    CompoundNode* cNode = new CompoundNode(12.345, 2);
    nodeList.append(cNode);

    CompoundNode* cNode1 = new CompoundNode(3.1415, 7);
    nodeList.append(cNode1);

    nodeList.append(new IntNode(3));

```

Note:

`fork()` was used by the debugger both to create the snapshot and to clone it.

Part II. A Guide to Using the Intel® Debugger

A Guide to Using the Intel® Debugger

This section provides most of the information needed to make expert use of the debugger. From this section the user will learn how to

- Prepare a program for debugging
- Start the Debugger
- Give commands to the Debugger
- Run the program under Debugger control
- Locate the site of a problem
- Look around at the code, the data, and other process information
- Modify the process
- Continue execution of the process
- Use snapshots as an undo mechanism
- Debug optimized code

This section also addresses

- Support limitations
- Context for executing commands

Preparing a Program for Debugging. Expert Debugging. Overview

To facilitate debugging, you can prepare your source code and the compiler and linker environment.

Preparing Your Source Code

You do not need to make changes to the source code to debug the program. However, you can do the following to make debugging easier:

- If the source code has functions that can be called to output data structures, you can call them from the debugger; you may want to create such functions.

- It is a good idea to make the following items part of your source code:
  - An initial stall point if you cannot stop the process easily from within the debugger.
  - Assertions sprinkled liberally through the sources to help locate errors early.

Preparing the Compiler and Linker Environment

Debugging information is put into `.o` files by compilers. The level and format of information is controlled by compiler options. Use the `-g` option with the Intel® C++ or Fortran (ifort) Compiler, for example:

```
% icc -g hello.c
...
% icpc -g hello.cpp
...
```

With the GNU* Compiler Collection (GCC, versions earlier than 3.x), use the `-gdwarf-2` option:

```
% gcc -gdwarf-2 hello.c
...
% g++ -gdwarf-2 hello.cpp
...
```

See your compiler's reference page for more details.

The debugging information is propagated into the `a.out` (executable) or `.so` (shared library) by the `link` command.

The debugging information can cause `.o` files to be very large, causing long link times, but even so it can also be incomplete.

If you are debugging optimized code, refer to the Debugging Optimized Code section of this manual and the appropriate compiler documentation for information about -g and related extended debug options and their relationship to optimization.

Starting the Debugger. Expert Debugging. Overview

You can start the debugger in the following ways:

- From a command line
- From within Emacs*
- From within DDD*
- From within Eclipse*

Starting the Debugger from a Command Line

When you invoke the debugger from a command line you can bring a program or core file under debugger control, or you can attach to a running process.

The following is the command line syntax to invoke the debugger using the **`idb`** command:

idb [*dbx_options*] [*executable_file* [*core_file*]]

Note that the set of recognized options depends on the mode. For example, **-v** option is valid in DBX mode, but not in GDB mode.

dbx_options:

[-**c** *file*]

[-cd *directory*]

[-**command** *file*]

[-**dbx**]

[-**echo**]

[-**emacs**]

[-**fullname**]

[-**gdb** [*gdb_options*]]

[-**gui**]

[-**help**]

[- **i** *file*]

[-**I** *dir*]

[-**interactive**]

[-**maxruntime** *minutes*]

[-**nosharedobjs**]

[-**parallel launcher** *launcher_args*]

[-**pid** *process_id*]

[-**prompt** *string*]

[-**quiet**]

[-**tty** *terminal_device*]

[-**V**]

[-**version**]

gdb_options:

: -**cd** *dir*

| -**command** *file*

| -**d**[**irectory**] *dir*

| -**f**[**ullname**]

| -**gdb** [*gdb_options*]

| -**help**

| -**interpreter** *name*

| -**nowindows**

| -**nw**

| -**p**[**id**] *pid*

| -**q**[**uiet**]

| -**silent**

| -**tty** *device*

| -**version**

| -**ui** *name*

DBX mode refers to the debugger's command input mode that is "dbx like" in its command syntax. It is not fully dbx compatible.

Note:

Options can be prefixed by a dash (`-`) or double dash (`--`) . Option names may be abbreviated as long as the abbreviations are unambiguous. An option and its argument are separated with one or more spaces or equal sign (`=`).

For example, to invoke the debugger on an executable file named `a.out`:

```
% idb a.out
```

To invoke the Debugger on a core file:

```
% idb a.out core
```

To invoke the debugger and attach to a running process when you do not know what file it is executing:

```
% idb -pid 8492
```

DBX mode options

The following table describes the `dbx` command options and parameters:

Options and Parameters	**Mode**	**Description**
-c *file* **-command** *file*	Default	Specifies an initialization command file. The default initialization file is `.dbxinitidbsetup.idb`. During startup, the debugger searches for this file in the current directory. If it is not there, the debugger searches your home directory. This file is processed after the target process has been loaded or attached to.
-cd *dir*	Default	Specifies a new working directory.
-echo	Default	Causes the debugger to print the prompt when running in a non-interactive session.
-gdb	Default	Causes the debugger to use GDB compatibility mode and the *gdb_options* options set.
-gui	Default	Activates the debugger's graphical user interface (GUI).
-emacs **-fullname**	Default	Output file and line number markers for Emacs*.
-help	Default	Print help message and exit.
-i *file*	Default	Specifies a pre-initialization command file. The default pre-initialization file is .idbrc. The debugger searches for this file during startup, first in the current directory and then in your home directory. This file is processed before the debugger has connected to the application being debugged, so that commands such as **`set $stoponattach = 1`** will have taken effect when the connection is made.
-I *dir*	Default	Specifies the directory containing the source code for the target program, in a manner similar to the **`use`** command. Use multiple **`-I`** options to specify more than one directory. The debugger searches directories in the order in which they were specified on the command line.
-interactive	Default	Causes the debugger to act as though `stdin` is `isatty()`, regardless of whether or not it is. This flag is sometimes useful when using `rsh` to run the debugger. Currently, the only effect is to cause the debugger to output the prompt to `stdout` when it is ready for the next line of input.
-maxruntime *minutes*	Default	Specifies the maximum allowable runtime in minutes for the debugging session.
-nosharedobjs	Default	Prevents the reading of symbol table information for any

		shared objects loaded when the process executes. Later in the debug session, you can enter the **`readsharedobj`** command to read the symbol table information for a specified object.
`-parallel` *`launcher launcher_args`*	Default	Starts a debugging session on a parallel application created by *launcher* with arguments *launcher_args*. See Debugging Parallel Applications for details on using the parallel debugging feature.
`-pid` *`pid`*	Default	Specifies the process ID of the process to be debugged.

`-prompt` *`string`*	Default	Specifies a debugger prompt. If the prompt argument contains spaces or special characters, enclose the argument in quotes (`""`). You can specify a debugger prompt when you start the debugger from a shell with the **`-prompt`** option. The default debugger prompt is `(idb)`.
		
		`% `**`idb -prompt ">> "`**
		`>> `**`quit`**
		
		Default Mode
		
		You can also change the prompt by setting the `$prompt` debugger variable. For example:
		
		`(idb) set $prompt = "newPrompt>> "`
		`newPrompt>>`
		
		GDB Mode
		
		Use **`set prompt`** *`prompt`* to specify a new prompt to use henceforth. To see the prompt used by the debugger, type the **`show prompt`** command.
		
		`(idb) `**`set prompt (gdb mode)`**
		`(gdb mode) `**`show prompt`**
		`idb's prompt is "(gdb mode) ".`
		`(gdb mode)`
		
		Note:
		
		There is a space at the end of the first line of the example above. If the space is missed, the result will be as follows:
		
		`(idb) set prompt (gdb mode)`
		`(gdb mode)show prompt`
		`idb's prompt is "(gdb mode)".`
		`(gdb mode)`

`-quiet`	Default	Causes the debugger to start but not to print sign-on message.

-tty *terminal_device*	Default	Specifies the input/output tty device for the user program.
-V **-version**	Default	Displays the banner, including the version.
executable_file	ALL	Specifies the program executable file.
core_file	ALL	Specifies the core file.

GDB mode options

The following table shows the GDB mode options:

Options and Parameters	Mode	Description
-cd *dir*	GDB	Specifies a new working directory
-command *file*	GDB	Specifies an initialization command file. The default initialization file is .dbxinit. During startup, the debugger searches for this file in the current directory. If it is not there, the debugger searches your home directory. This file is processed after the target process has been loaded or attached
-dbx	GDB	Causes the debugger to use the *idb_options* options set (default)
-/directory *dir*	GDB	Searches for source files in *dir*
-fullname	GDB	Outputs information used by Emacs*-GDB interface
-help	GDB	Print help message and exit
-interpreter *name* **-ui** *name*	GDB	Selects a mode interpreter interface
-nowindows **-nw**	GDB	Do not use a window interface
-pid *pid*	GDB	See DBX mode option **-pid**
-quiet **-silent**	GDB	Do not print copyright message
-tty *device*	GDB	Use *device* for input/output by the program being debugged
-version	GDB	Print version information and exit

Starting the Debugger Using Emacs*

You can control your debugger process entirely through the Emacs* Grand Unified Debugger (GUD) buffer mode, which is a variant of shell mode. All the debugger commands are available, and you can use the shell mode history commands to repeat them.

The debugger supports:

- GNU* Emacs* Version 19 and higher
- XEmacs* Version 19.14 and higher

The information in the following sections assumes you are familiar with Emacs and are using the Emacs notation for naming keys and key sequences.

Running IDB in Default (DBX) Mode

For each Emacs* session, before you can invoke the debugger, you must load the Intel® Debugger-specific Emacs LISP code, as follows:

```
M-x load-file
```

At the `Load file:` prompt, type the path to the Intel® Debugger-specific Emacs LISP file, which is located in the Intel IDB installation directory. For example:

```
/opt/intel_idb/bin/idb.el
```

You can also place a load-file call in your Emacs initialization file (`~/.emacs`). For example:

```
(load-file "/opt/intel_idb/bin/idb.el")
```

To start the debugger with Emacs, type:

```
M-x idb
```

The following invocation line displays:

```
Run the Debugger (like this): idb
```

Edit the invocation line by typing the target program and pressing Return. Emacs remembers the invocation. To debug the same program again, you need only press Return.

Emacs displays the GUD buffer and runs the debugger within it; the debugger starts and displays its `(idb)` prompt, indicating readiness. The GUD buffer saves all of the commands you type and the program output for you to edit. In general, interact with the debugger in the GUD buffer as you would with a debugger started from a shell.

One of the benefits of running the debugger from within Emacs is a closer correlation between program execution and source. When your program stops (for example, at a breakpoint), Emacs displays the source of your program in a second buffer (source buffer) and indicates the current execution line with `=>`.

Note:

> If the source is already loaded into a buffer, Emacs often finds that buffer. However, in some NFS mounting situations, Emacs may use an alternate name for some directories and will create a second buffer for your source (often with `<2>` appended to the name). Be careful that you do not modify the original buffer or kill it outright.

Running IDB in GDB Mode

From the Emacs* "Tools" menu, select "Debugger..." The following invocation line displays:

```
Run the debugger (like this): gdb
```

Edit the invocation line by typing the path to IDB and the IDB options followed by the target program, and pressing Return. For example:

```
 /opt/intel_idb/bin/idb -gdb -fullname myprogram
```

Emacs will remember the invocation. To debug the same program again, you need only press Return.

By default, Emacs sets its current working directory to be the directory containing the target program. Because the debugger does not do this when invoked directly, you may need to change the source code search path when using the debugger from within Emacs. To set an alternate source code search path, use the debugger **`map source directory`** command.

All Emacs* editing functions and GUD key bindings are available. For example:

- You can execute a **`step`** command by typing the command in the GUD buffer.
- You can select a line of code in the current source buffer and type a command to set a breakpoint at that position:

```
C-x SPC
```

For more information on Emacs functionality and key bindings, see the Emacs documentation. For example:

```
M-x info
```

Then select the Emacs menu, then the debuggers menu.

XEmacs will come up with the source buffer displayed. Use **`C-x 2`** and a buffer menu to select the control buffer.

Starting the Debugger Using DDD*

GNU* DDD* is a graphical front-end for command-line debuggers that can be used with Intel IDB.

DBX Mode

Specify `--ladebug` and `--debugger` *`idb`* options in the shell command line, for example:

```
$ ddd --ladebug --debugger idb a.out
```

If `idb` is not accessible through `PATH` environment variable, specify path to the debugger, absolute or relative, for example:

```
$ ddd --ladebug --debugger /opt/intel_idb/bin/idb a.out
```

GDB Mode

Specify `--debugger "`*`idb -gdb`*`"` options in the shell command line, for example:

```
$ ddd --debugger "idb -gdb" a.out
```

If `idb` is not accessible through `PATH` environment variable, specify path to the debugger, absolute or relative, for example:

```
$ ddd --debugger "/opt/intel_idb/bin/idb -gdb" a.out
```

Starting the Debugger Using Eclipse*

The Intel® C++ Compiler for Linux* includes a debugger integration with Eclipse* and the C/C++ Development Tools* (CDT). This functionality is an optional part of the debugger installation for Intel® C++ Compilers for IA-32 and Itanium®-based systems.

Eclipse is an open source software development project dedicated to providing a robust, full-featured, commercial-quality, industry platform for the development of highly integrated tools. It is an extensible, open source integrated development environment (IDE).

The CDT* (C/C++ Development Tools) project is dedicated to providing a fully functional C/C++ IDE for the Eclipse platform. CDT is layered on Eclipse and provides a C/C++ development environment perspective.

The Intel® Debugger integration with the Eclipse/CDT IDE lets you debug your C/C++ projects in a visual, interactive environment.

See also

www.eclipse.org/ for further information about Eclipse
www.eclipse.org/cdt/ for further information about CDT

Intel compilers web pages for more information about using Intel(® C++ Compiler with the Eclipse Integrated Development Environment on Linux Systems

Starting Eclipse*

After you have installed the Intel® C++ Compiler for 32-bit or Itanium®-based applications, the Intel® Eclipse integration support, and the Eclipse package, which contains a copy of Eclipse and CDT* and is provided with the Intel® C++ Compiler, you can start Eclipse in two ways: you can use the Intel supplied Eclipse launchers or invoke Eclipse directly yourself.

Eclipse Launchers

To invoke Eclipse with the Intel supplied Eclipse launchers, execute the **`iccec`** or **`idbec`** shell script. With the default Compiler installation, execute **`iccec`** or **`idbec`** as follows:

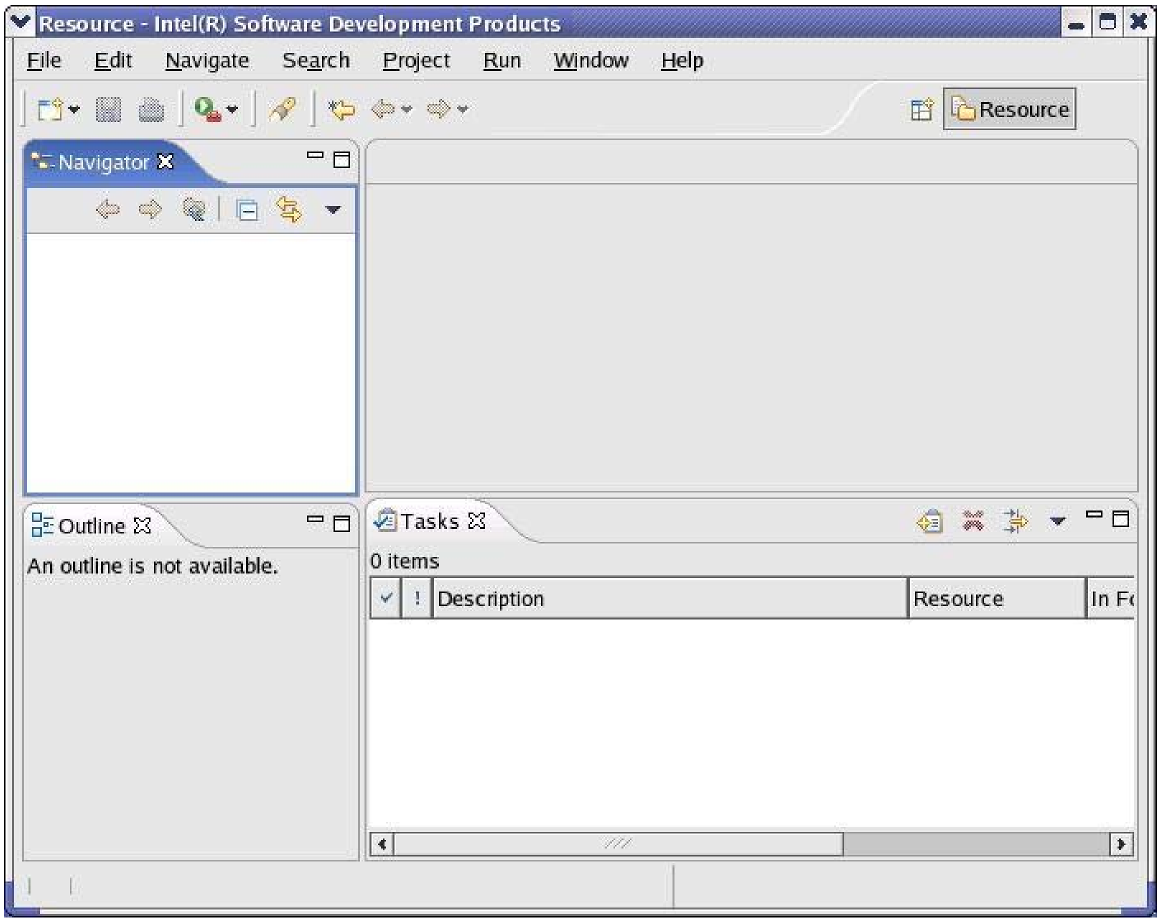
```
/opt/intel/cc/9.1.xxx/bin/iccec
```

or

```
/opt/intel/idb/9.1.xxx/idbec
```

where *xxx* indicates a package number.

The **`iccec`** or **`idbec`** script opens the Eclipse IDE:


Resource - Intel(R) Software Development Products
File Edit Navigate Search Project Run Window Help
Resource
Navigator
Outline
An outline is not available.
Tasks
0 items
Description
Resource
In F

You can also use **`iccec`** or **`idbec`** to pass Eclipse-specific parameters, such as:

- `-data <path>` - sets the location for the Eclipse workspace
- `-showlocation` - shows the location of the workspace in the Eclipse window title bar.

For example:

```
/opt/intel/cc/9.1.xxx/bin/iccec -data /cpp/eclipse -showlocation
```

From the Eclipse Help menu, select **Help Contents > Workbench User's Guide > Tasks > Running Eclipse** for the complete list of Eclipse startup parameters.

Invoke Eclipse Directly

To invoke Eclipse directly, some additional setup is required to use the Intel® C++ Compiler and the Intel® Debugger within Eclipse. This setup is required if you are using the versions of Eclipse and CDT provided by Intel in the Eclipse package and you want to invoke Eclipse directly. It is also required if you are using your own version of Eclipse and CDT obtained from www.eclipse.org/ with the Intel Eclipse compiling and debugging support being integrated.

Note:

> As an option, during installation of the compiler and debugger, you can integrate the Intel® Eclipse Compiler and Debugger support into an already existing version of Eclipse.

If you want to use the Intel® C++ Compiler and/or the Intel® Debugger within Eclipse, run the scripts **`iccvars.sh`** or **`iccvars.csh`**. The **`idbvars.sh`** or **`idbvars.csh`** scripts should be invoked prior to invoking Eclipse. These scripts are automatically provided when the compiler and debugger are installed. After running these scripts, you can invoke and run Eclipse with the Intel® Compiler and the Intel® Debugger. To invoke Eclipse, run the executable "**eclipse.**" This executable is located in the top level directory used to install Eclipse. With a default compiler installation, where the user selects to install the Intel supplied Eclipse package, invoke the executable as follows:

```
/opt/intel/eclipsepackage/eclipse version/eclipse/eclipse
```

where "*eclipse version*" is a version of Eclipse.

You can specify Eclipse-specific parameters to the `eclipse` command.

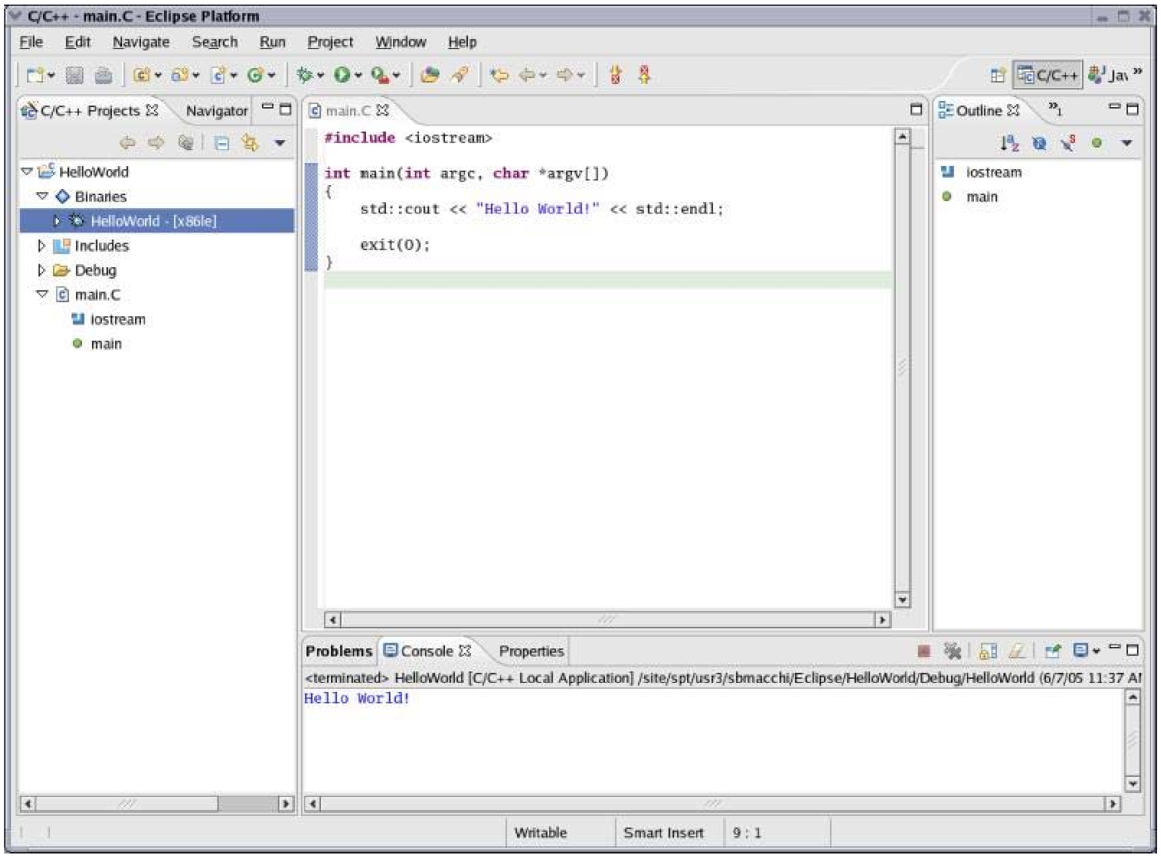
From the Eclipse Help menu, select **Help Contents > Workbench User's Guide > Tasks > Running Eclipse** for the complete list of Eclipse startup parameters.

Debugging with the Intel® Debugger in Eclipse*

When your C/C++ project is built and ready for debugging, you can follow the steps below to invoke the Intel® Debugger (See Compiler or Eclipse/CDT* documentation for details on creating and building projects):

1.Select the executable to debug.

Choose **>Run > Debug**...from the main menu.

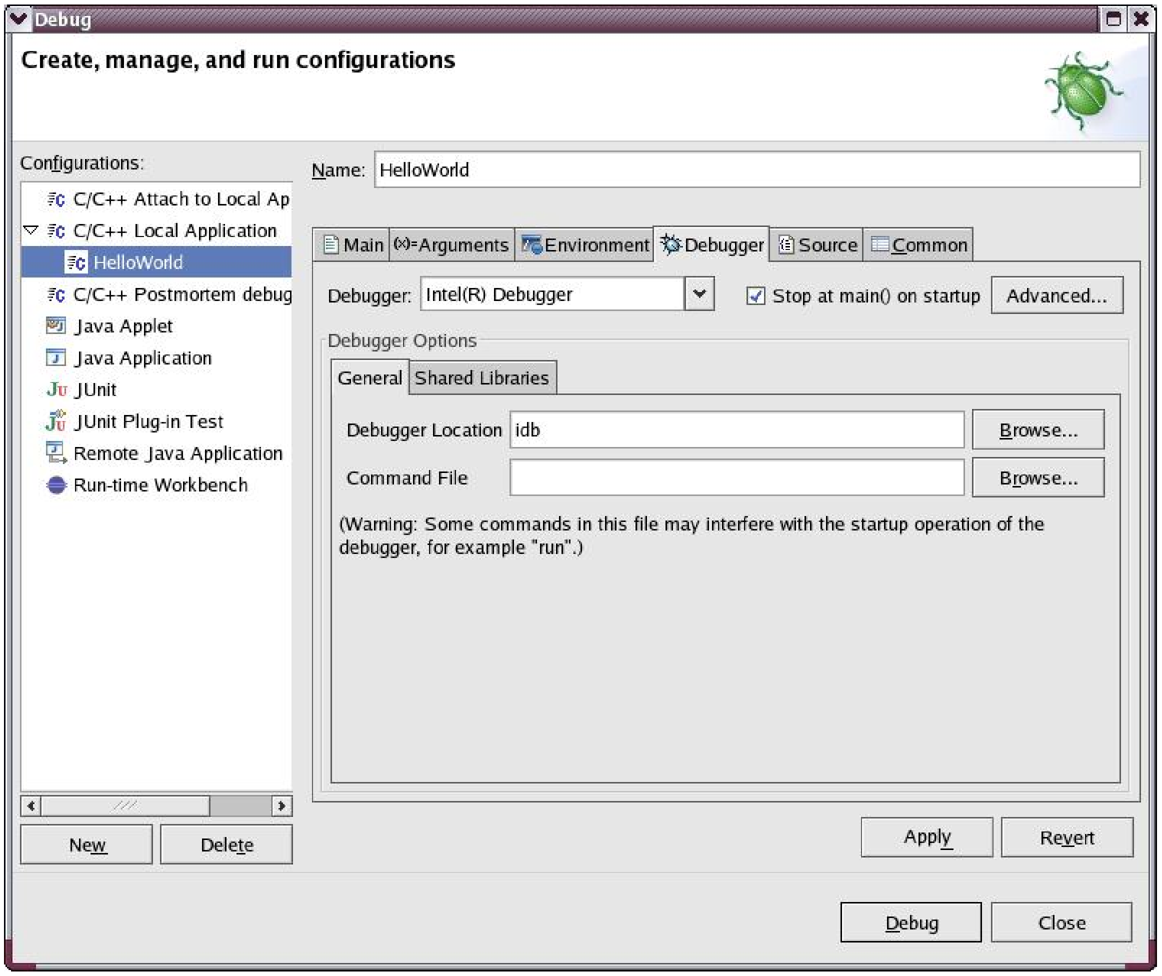


```
#include <iostream>

int main(int argc, char *argv[])
{
    std::cout << "Hello World!" << std::endl;

    exit(0);
}
```

```
<terminated> HelloWorld [C/C++ Local Application] /site/spt/usr3/sbmacchi/Eclipse/HelloWorld/Debug/HelloWorld (6/7/05 11:37 AI
Hello World!
```

2. Click the **Debugger** tab.

Choose the **Intel® Debugger** from the **Debugger** drop-down list.


Create, manage, and run configurations

Configurations:
- C/C++ Attach to Local Ap
- C/C++ Local Application
  - HelloWorld
- C/C++ Postmortem debug
- Java Applet
- Java Application
- JUnit
- JUnit Plug-in Test
- Remote Java Application
- Run-time Workbench

Name: HelloWorld

Debugger: Intel(R) Debugger

[✓] Stop at main() on startup

Debugger Options

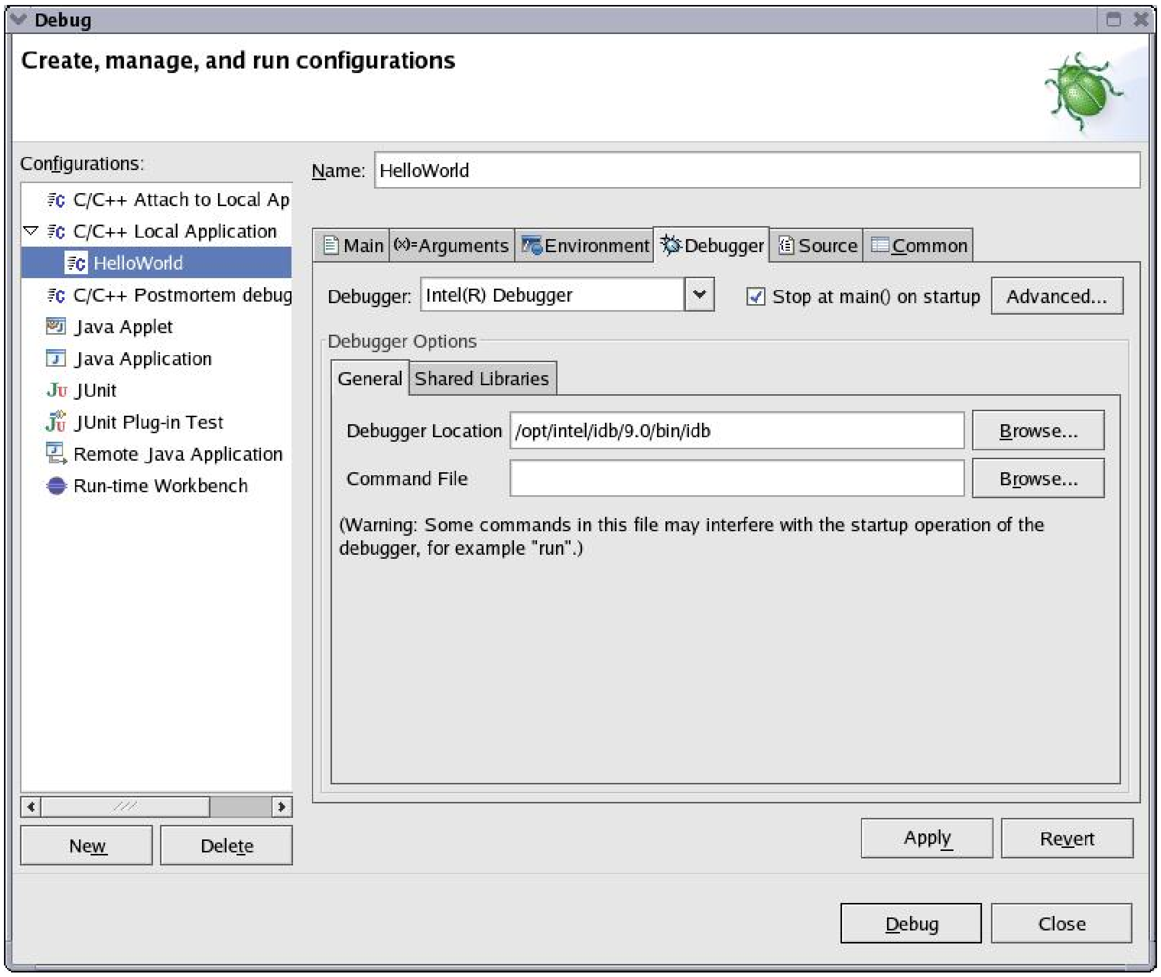
Debugger Location: idb

Command File:

(Warning: Some commands in this file may interfere with the startup operation of the debugger, for example "run".)

3. If you have started Eclipse with `iccec` or `idbec`, or you ran *idbvars.sh* or *idbvars.csh*, skip step 4.

4. If you have *not* started Eclipse with `iccec` or `idbec`, or you did not run *idbvars.sh* or *idbvars.csh*, put the full path to the debugger executable in the "Debugger Location" field:



Name: HelloWorld

Debugger: Intel(R) Debugger

☑ Stop at main() on startup

Debugger Location /opt/intel/idb/9.0/bin/idb

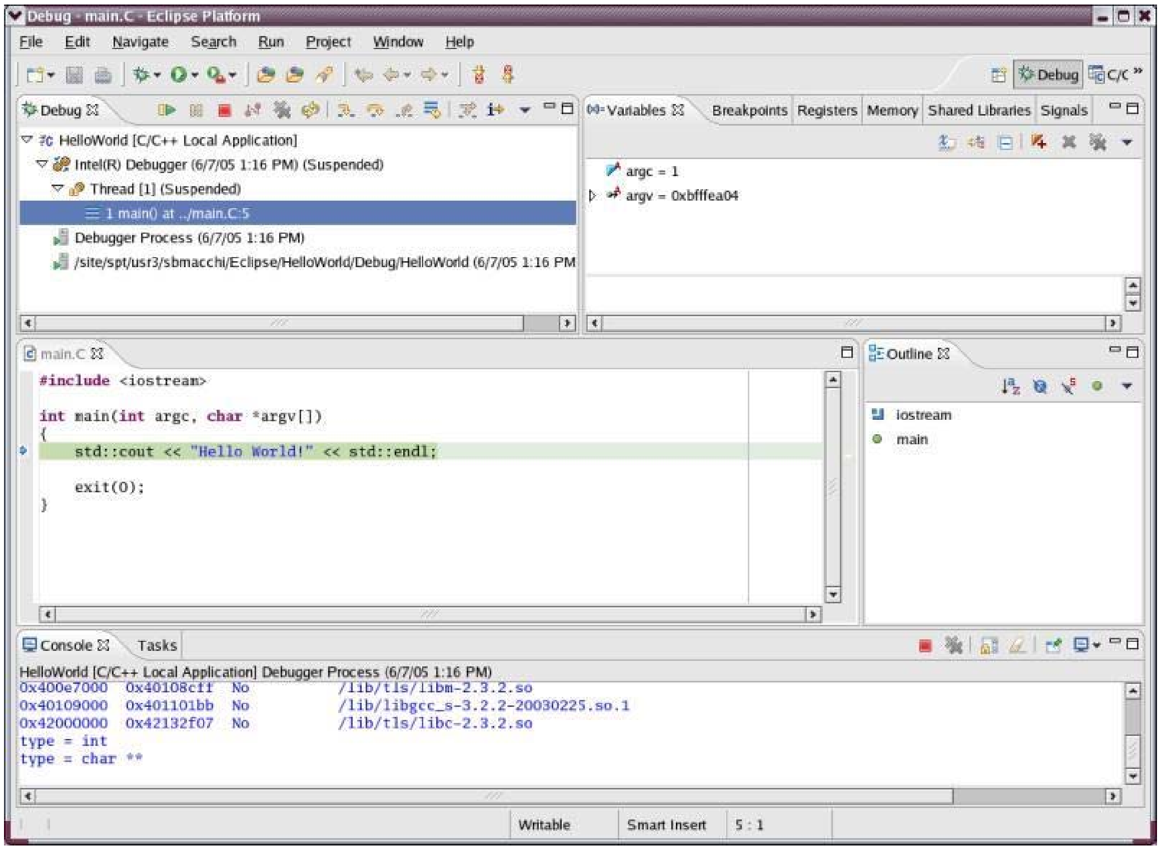
Command File

(Warning: Some commands in this file may interfere with the startup operation of the debugger, for example "run".)

5. Click the **Debug** button.

For subsequent debug sessions, you just have to select the program in the navigator tab and then select **Run > Debug As > C/C++ Local Application** (or to click the bug icon in the toolbar).

The perspective will switch to the debug perspective like the example below:


Debug - main.C - Eclipse Platform
File Edit Navigate Search Run Project Window Help
Debug C/C
Debug
Variables Breakpoints Registers Memory Shared Libraries Signals
main.C
Outline
iostream
main
Console Tasks
Writable Smart Insert 5 : 1

```
HelloWorld [C/C++ Local Application]
  Intel(R) Debugger (6/7/05 1:16 PM) (Suspended)
    Thread [1] (Suspended)
      1 main() at ../main.C:5
  Debugger Process (6/7/05 1:16 PM)
  /site/spt/usr3/sbmacchi/Eclipse/HelloWorld/Debug/HelloWorld (6/7/05 1:16 PM
```

```
argc = 1
argv = 0xbfffea04
```

```
#include <iostream>

int main(int argc, char *argv[])
{
    std::cout << "Hello World!" << std::endl;

    exit(0);
}
```

```
HelloWorld [C/C++ Local Application] Debugger Process (6/7/05 1:16 PM)
0x400e7000  0x40108c1f  No         /lib/tls/libm-2.3.2.so
0x40109000  0x401101bb  No         /lib/libgcc_s-3.2.2-20030225.so.1
0x42000000  0x42132f07  No         /lib/tls/libc-2.3.2.so
type = int
type = char **
```

Ending a Debugging Session

DBX Mode

To exit the debugger, use the **quit** command:

quit_command

                : **quit**

Alternatively, you can type **q** or **exit**, which are pre-defined aliases for **quit**.

GDB Mode

To exit the debugger, use **quit** or the **q** command.

Optionally, you can specify debugger exit status. Example:

quit_command

                : **quit** [*exit_status*]

                | **q** [*exit_status*]

```
exit_status
            : expression
```

Getting Help

To access the online help about debugger commands, use the **`help`** command.

DBX Mode

```
help_command
            : help [ topic ]
```

Enter **`help`** to see a list of help topics. Enter **`help command`** to see a list of debugger commands. Enter **`help idb`** to see a list of function-oriented debugger commands.

GDB Mode

```
help_command
            : help [ topic ]
            | h [ topic ]
            | complete [ args ]
```

Use the **`help`** or **`h`** command to display a list of command groups. Use the **`help`** command with a group name or command name to get more detailed help.

Use the **`complete`** *`args`* command to list all the possible completions for the beginning of a command, where *args* specifies the text to be completed.

Giving Commands to the Debugger. Overview

The debugger has several different mechanisms you can use to direct its behavior. It receives input from:

- Environment variables
- Command line
- `stdin`, which is usually one of the following:

- A terminal
- A file
- A pipe connecting the debugger to an editor (usually Emacs*)

- Other files:

1. At startup, before attaching to or starting the target executable and before processing command line qualifiers, commands in:
   1. `.idbrc`, if available, otherwise
   2. `~/.idbrc`, if available
2. Just before accepting command input from you:
   1. `./dbxinit`, if available, otherwise
   2. `~/.dbxinit`, if available
3. Files specified in the **`source`** command

Some examples of the difference between `.idbrc` and `.dbxinit` are shown in the following table:

Example Command	**If Used in `.idbrc`**	**If Used in `.dbxinit`**
Assume the command "**`set $stoponattach = 1`**" is in one of these files and you invoked the debugger as: `% idb -pid process_id executable_file`	The debugger attaches and stops.	The debugger attaches and waits for you to press Ctrl+C; subsequent attaches will stop.
Assume the command "**`stop in main`**" is in one of these files.	The debugger generates a message that there is no `main` in which to place a breakpoint, because there is no target yet.	The debugger sets the breakpoint (assuming there is a `main` in the target).

Debugger's Command Processing Structure. Expert Debugging

The debugger processes commands as follows:

1. Prompts for input.

- Obtains a complete line from the input file and performs:
  - History replacement of the line
  - Alias expansion of the line

1. Parses the entire line according to the parsing rules for the current language.
2. Executes the commands.

Interrupting a Debugger Action

To interrupt program execution or to abort a debugger action, press Ctrl+C. This returns the debugger to the prompt.

Entering and Editing Command Lines

The debugger reads lines from `stdin`. The debugger supports command line editing when processing `stdin` if `stdin` is a terminal and the debugger variable `$editline` is non-zero. However, by default, `$editline` is zero, and the standard Command Prompt window's capabilities are used. If the debugger's line editing is needed, use the **`set`** command to change the setting, and set the terminal width to the correct value. After editing, press the Enter key to send the line to the debugger.

- Use the left and right arrow keys to edit parts of the line.
- Use the up and down arrow keys to recall and edit earlier commands.

Note:

When you use the up and down arrow keys, the debugger skips duplicate commands. To see a complete list of the commands you have entered, use the **`history`** command.

The debugger copies each line from `stdin` to the record input file, if you have requested that file.

The debugger scans each line from the beginning, looking for backslash (`\`) characters, which 'quote' the immediately following character. If the line ends in a quoted newline, then another line is similarly processed from `stdin` and appended to the first one, with the quoted newline removed.

Whether or not command line editing is enabled, you can always use your terminal's cut-and-paste function to avoid excessive typing while entering input.

This section gives information about

History Replacement of the Line

Alias Expansion of the Line (DBX Mode only)

Environment Variable Expansion

History Replacement of the Line

Leading spaces and tabs are removed from the assembled line.

For assembled lines that begin with an exclamation point (`!`), the following rules apply:

- If the second character is also an exclamation point (`!`), the assembled line is replaced by the most-recent entry from the history list. Any remaining characters after the digits or `!` are appended to the assembled line.
- Otherwise, spaces and tabs are skipped, and one of the following actions occurs:
    - If the next character is a digit, then the digits are read as a decimal number, and the assembled line is replaced by that line from the history list, with 1 being the oldest entry.
    - If the next character is a hyphen (`-`), then the digits following it are read as a decimal number, and the assembled line is replaced by that line from the history list, with -1 being the most-recent entry.
    - Otherwise, the rest of the line is used to find the most-recent command that starts with those characters, and the assembled line is replaced by that line from the history list.

In the first two cases, any remaining characters after the digits are appended to the assembled line.

For lines that begin with a caret (`^`), these rules apply:

- The line is analyzed to extract the following:
    - The characters following the first caret but before a second caret, or until the end of line. These characters are the target string.
    - If there is a second caret, the characters following it but before a third caret, or until the end of line. These characters are the replacement string.
    - If there is a third caret, the characters following it to the end of the line. These characters are the append string.
- The most-recent entry from the history list is checked to see if it has an occurrence of the target string. If it does not, an error is reported.
- The assembled line is replaced by this most-recent entry, except that the first occurrence of the target string is replaced by the replacement string (possibly zero length), and the append string is appended to the assembled line.

The assembled line is now appended to the history list.

Exclamation points and carets cannot be used in command lists built with braces (`{}`); for example, **`{print3; !!3}`** will not parse. They may be used in scripts.

History in a command list is not limited by braces, but goes all the way back. For example:

```
(idb) print 1
1
(idb) stop at 182 { print 2; history 3 }
[#1: stop at "src/x list.cxx":182 { print 2; history 3 }]
(idb) run
2
11: print 1
12: stop at 182 {print 2; history 3}
13: run
[1] stopped at [int main(void):182 0x08052e8f]
    182     List<Node> nodeList;
```

Note:

Commands in breakpoint action lists are not entered into the history list.

Alias Expansion of the Line (DBX Mode only)

The assembled line is now subjected to alias expansion. This is done by scanning the line, looking for pound (`#`), semicolon (`;`), and left brace (`{`) characters that are not inside strings.

- Strings are recognized by their opening and closing double or single quotes. Backslash quotation causes a quote character not to terminate the string.
- Pound (`#`) characters and all that follow to the end of the line are discarded, unless the pound character is the very first character in the line. If that is the case, the pound character is not discarded because a completely empty line has special meaning. An exception is made for pound (`#`) characters that are surrounded by non-whitespace characters, such as "file#name". This is needed because the `tmpnam` standard library function generates file and directory names containing pound (`#`) characters.

The debugger performs alias expansion as follows:

1. At the beginning of the line, and immediately after semicolon (`;`) or left brace (`{`) characters not inside strings, the debugger checks for the occurrence of an alias identifier.
2. If it finds an alias identifier, it associates the formal parameters of the alias with the specified actual parameters.

   If the alias has no formal parameters, this match consumes no more of the input.

   a. If there are formal parameters, white space is skipped, and then a '`(`' character is checked for and skipped. The characters following the '`(`' up to the first non-nested '`,`' or '`)`' character are associated with the formal parameter.

      Again, the characters within strings are not tested. Nesting is caused by '`(`' and '`)`' characters outside of strings.

   b. If there are more formal parameters, the '`,`' character is treated as the terminator of the actual parameter. It is skipped and processing continues as for the first parameter.
3. After the alias and the correct number of actuals have been identified, all the characters from the start of the alias identifier to its end (no parameters) or the trailing '`)`' (one or more parameters) are replaced by the expansion.
4. Within the definition of the alias, all occurrences of the formal parameter are replaced by the actual parameter, regardless of whether or not it is in a string.

Environment Variable Expansion

The debugger expands environment variables and the leading tilde (~) in the following cases:

- As part of a command in which a file name or a directory is expected.
- In the arguments to **`run`** or **`rerun`** (dbx).

As in any shell, you can group an environment variable name using a pair of curly braces ({}), and quote a dollar sign ($) by preceding it with a backslash (\).

The following table shows how various environment variables expand. It assumes that the home directory is `/usr/users/hercules` and the environment variable BIN is `/usr/users/hercules/bin.`

Command with Environment Variable	Expands into
`load ~/a.out`	`load /usr/users/hercules/a.out`
`load $BIN/a.out`	`load /usr/users/hercules/bin/a.out`
`load ${BIN}2/a\$b`	`load /usr/users/hercules/bin2/a$b`
`map source directory $BIN ${BIN}2`	`map source directory /usr/users/hercules/bin /usr/users/hercules/bin2`
`stop at "$BIN/a.out":20`	`stop at "/usr/users/hercules/bin/a.out":20`
`run $BIN/a.out ~/core`	`run /usr/users/hercules/bin/a.out /usr/users/hercules/core`

Syntax of Commands

The debugger has different parsing rules for each of the different languages it supports. A line is processed according to the current language, even if executing the line will change the current language. This section discusses the following topics:

- Lexical Elements of Commands
- Grammar of Commands
- Categories of Commands
- Keywords Within Commands
- Using Braces to Make a Composite Command
- Conditionalizing Command Execution
- Debugger Variables

Lexical Elements of Commands

For the debugger to parse the line, it must first turn the line into a sequence of tokens, a process called "tokenizing" or "lexical analysis". Tokenizing is done with a state machine.

As the debugger starts tokenizing a line into a command, it starts processing the characters using the lexical state LKEYWORD. It uses the rules for lexical tokens in this state, recognizing the longest sequence of characters that forms a lexical token.

After the lexical token is recognized, the debugger appends it to the tokenized form of the line, perhaps changes the state of the tokenizer, and starts on the next token.

For more detailed information on lexical elements, see Lexical Elements of Commands in Part III.

Grammar of Commands

```
Some pieces of the grammar were modified from a grammar originally written by
James A. Roskind, and covered by a copyright that requires a statement that...
Portions Copyright (c) 1989, 1990 James A. Roskind
```

Each command line must parse as one of the following:

```
input

            : command_list

            | comment
```

A `command_list` is a sequence of commands that are executed one after the other.

```
command_list

            : command ;...

            | command ;

            | command
```

A `comment` is a line that begins with a pound (`#`) character.

```
comment

            : #
```

Any text after an unquoted pound character is ignored by the debugger. If the first non-whitespace character on a line is a pound character, the whole line is ignored.

Note:

The difference between a blank command line and a command line that is a comment is that a blank line entered from the keyboard will cause the debugger to repeat the previous command and the comment line will not. Blank lines not entered from the keyboard are treated as comment lines.

Categories of Commands

Commands usually start with, and often contain, keywords. These keywords must be lowercase.

DBX Mode

Following is a list of debugger command categories:

```
command

          : alias_command

          | attach_command

          | braced_command_list

          | breakpoint_command

          | browse_source_command

          | call_stack_command

          | command_repetition_command

          | continue_command

          | detach_command

          | detach_remote_command

          | dbgvar_command

          | disconnect_remote_command

          | edit_file_command

          | environment_variable_command

          | execute_commands_from_file_command

          | execute_shell_command

          | help_command
```

```
        | history_command
        | if_command
        | kill_command
        | load_command
        | look_around_command
        | machinecode_level_command
        | modifying_command
        | multiprocess_command
        | parallel_debugging_command
        | quit_command
        | record_command
        | run_command
        | snapshot_command
        | shared_library_command
        | thread_command
        | unload_command
        | while_command
```

Keywords Within Commands

If the identifiers **`thread`**, **`in`**, **`at`**, and **`if`** occur within the `expression` in the following commands, the Debugger treats them as keywords unless they are enclosed within parentheses (`()`).

- **`where`** *`expression`*
- **`stopi`** *`expression`*
- **`trace`** *`expression`*
- **`tracei`** *`expression`*
- **`wheni`** *`expression`*

For example, if your program has `thread` defined as an integer, enter the following command to inspect the first thread levels of the stack.

For example:

```
(idb) where 3
```

```
>0  0x0804868e in c() "src/x_whereAmbigParse.c":7
#1  0x080486ad in b() "src/x_whereAmbigParse.c":12
#2  0x080486bf in a() "src/x_whereAmbigParse.c":13
(idb)
(idb)
(idb)
(idb) where three(3)
>0  0x0804868e in c() "src/x_whereAmbigParse.c":7
#1  0x080486ad in b() "src/x_whereAmbigParse.c":12
#2  0x080486bf in a() "src/x_whereAmbigParse.c":13
(idb)
(idb)
(idb)
(idb) where thread (1)
Stack trace for thread 1
>0  0x0804868e in c() "src/x_whereAmbigParse.c":7
#1  0x080486ad in b() "src/x_whereAmbigParse.c":12
#2  0x080486bf in a() "src/x_whereAmbigParse.c":13
#3  0x080486da in main() "src/x_whereAmbigParse.c":17
#4  0xb739e748 in __libc_start_main(...) in /lib/tls/libc-2.3.2.so
#5  0x08048551 in _start(...) in /home/user/examples/x_whereAmbigParse
(idb)
(idb)
(idb)
(idb) where three(3) thread (1)
Stack trace for thread 1
>0  0x0804868e in c() "src/x_whereAmbigParse.c":7
#1  0x080486ad in b() "src/x_whereAmbigParse.c":12
#2  0x080486bf in a() "src/x_whereAmbigParse.c":13
(idb)
(idb)
(idb)
(idb) where (thread(3))
>0  0x0804868e in c() "src/x_whereAmbigParse.c":7
#1  0x080486ad in b() "src/x_whereAmbigParse.c":12
#2  0x080486bf in a() "src/x_whereAmbigParse.c":13
(idb)
(idb)
(idb)
```

Using Braces to Make a Composite Command

It is possible to surround a `command_list` with braces to make it work like a single command. Some places require a `braced_command_list` just for readability, or to assist the debugger in understanding your input.

braced_command_list

        : { *command_list* }

Conditionalizing Command Execution

`if` command

The debugger provides the **`if`** command , whose behavior depends on the value of an expression.

if_command

: if *expression* *braced_command_list* [*else_clause*]

else_clause

:**else** *braced_command_list*

In this command, the first `braced_command_list` is executed if `expression` evaluates to a non-zero value; otherwise, the `braced_command_list` in the `else_clause` is executed, if specified.

For example:

```
(idb) set $c = 1
(idb) assign pid = 0
(idb) if (pid < $c) { print "Greater" } else { print "Lesser" }
Greater
```

while command

In addition to the **if** command, the debugger also provides the **while** command.

while_command

: **while** *expression* *braced_command_list*

The commands in the `braced_command_list` will execute as long as expression evaluates to a non-zero value.

For example:

```
(idb) stop at 167
[#1: stop at "src/x list.cxx":167]
(idb) run
The list is:
[1] stopped at [void List<Node>::print(void):167 0x0804c632]
    167         cout << "Node " << i ;
(idb)
(idb) while (currentNode-> data != 5) { print "currentNode-> data is ",
currentNode-> data; cont }
currentNode-> data is  1
Node 1 type is integer, value is 1
[1] stopped at [void List<Node>::print(void):167 0x0804c632]
    167         cout << "Node " << i ;
currentNode-> data is  2
Node 2 type is compound, value is 12.345
       parent  type is integer, value is 2
[1] stopped at [void List<Node>::print(void):167 0x0804c632]
    167         cout << "Node " << i ;
currentNode-> data is  7
Node 3 type is compound, value is 3.1415
       parent  type is integer, value is 7
[1] stopped at [void List<Node>::print(void):167 0x0804c632]
    167         cout << "Node " << i ;
currentNode-> data is  3
Node 4 type is integer, value is 3
```

```
[1] stopped at [void List<Node>::print(void):167 0x0804c632]
    167         cout << "Node " << i ;
currentNode-> data is  4
Node 5 type is integer, value is 4
[1] stopped at [void List<Node>::print(void):167 0x0804c632]
    167         cout << "Node " << i ;
(idb)
(idb) print currentNode-> data
5
```

In this example we use the **`while`** command to continue the execution of the debuggee until the `_data` field in `currentNode` is 5.

Note that if the commands in the `braced_command_list` do not change the state of the debuggee process, such as the value of a variable or the PC register, then the **`while`** command can go into an infinite loop. In this case, press Ctrl+C to interrupt the loop, or type 'n' when you see the "More (n if no)?" prompt if your **`while`** command generates output and the paging is turned on.

Debugger Variables

Debugger variables are pseudovariables that exist within the debugger instead of within your program. They have the following uses:

- Support some limited programming capabilities within the debugger command language
- Allow you to examine and change various debugger options
- Allow you to find out exactly what various debugger commands did

The following table lists the three different varieties of debugger variables:

Kind of variable	**Purpose**
User-defined variables	You create these and can set them to a value of any type.
Preference variables	You modify these to change debugger behavior. You can only set a preference variable to a value that is valid for that particular variable.
Display/state variables	These variables display the parts of the current debugger state. You cannot modify them.

For more information about debugger variables, see Debugger Variables.

The following commands deal specifically with debugger variables:

`dbgvar_command`

: **`set`** *`dbgvar_name`* = *expression*

| **set** *dbgvar_name*

| **set**

| **unset** *dbgvar_name*

The *dbgvar_name* should not exist anywhere in your program, or you may confuse yourself about which of the occurrences you are actually dealing with. The predefined debugger variables all start with a dollar sign (`$`), to help avoid this confusion. It is strongly recommended that you follow the same practice; in a future release, all debugger variables will be required to start with a dollar sign.

Note:

> If a debugger variable exists that shares a name with a program variable, and you print an expression involving that name, which of the two variables the debugger finds is undefined.

The first form creates the debugger variable if it does not already exist. It then sets the value of the debugger variable to the result of evaluating the expression. For example:

```
(idb) set $myLoopCounter = 0
(idb) print $myLoopCounter
0
```

The second form is equivalent to the command **set** *dbgvar_name* **= 1**. For example:

```
(idb) print $stoponattach
0
(idb) set $stoponattach
(idb) print $stoponattach
1
```

The **set** form shows all the debugger variables and their values:

```
(idb) set
$ exitcode = 0
$ascii = 0
$beep = 1
$catchexecs = 0
$catchforkinfork = 0
$catchforks = 0
$childprocess = 0
$cmdset = "dbx"
$curcolumn = 0
$curevent = 0
$curfile = "src/x list.cxx"
$curfilepath = "../src/x list.cxx"
$curline = 182
$curpc = 0x8052df4
```

```
$curprocess = 17633
$cursrcline = 182
$cursrcpc = 0x8052df4
$curthread = 1
$dbxoutputformat = 0
$dbxuse = 0
$debuggerpid = 17631
$decints = 0
$disasm shows unwind = 0
$doverbosehelp = 1
$editline = 1
$eventecho = 1
$float80bit = 0
$floatshrinking = 1
$framesearchlimit = 0
$funcsig = 1
$gdb compatible output = 0
$givedebughints = 1
$hasmeta = 0
$hexints = 0
$highpc = (internal debugger function)
$historylines = 20
$indent = 1
$isaEM64T = 0
$isaIA32 = 1
$isaIPF = 0
$lang = "C++"
$lasteventmade = 0
$lc ctype = "en US.ISO8859-1"
$listwindow = 20
$main = "\"src/x list.cxx\"`main"
$maxarrlen = 1024
$maxlines = 5000
$maxstrlen = 128
$memorymatchall = 0
$myLoopCounter = 0
$octints = 0
$overloadmenu = 1
$page = 0
$pagewindow = 0
$parentprocess = 0
$pimode = 1
$prompt = "(idb) "
$readtextfile = 0
$regstyle = 1
$repeatmode = 1
$reportsotrans = 0
$showlineonstartup = 0
$showwelcomemsg = 1
$stack levels = 50
$stackargs = 1
$statusargs = 1
$stepg0 = 0
$stoponattach = 1
$stopparentonfork = 0
$symbolsearchlimit = 100
$threadlevel = "native"
$tracesyscalls = 0
$usedynamictypes = 1
$verbose = 0
```

To see the value of just one debugger variable, **`print`** it. For example:

```
(idb) print $catchexecs
0
```

The **`unset`** form deletes the debugger variable. Some predefined debugger variables either cannot be deleted or are automatically recreated in the future when needed. For example:

```
(idb) unset $myLoopCounter
(idb) print $myLoopCounter
Symbol "$myLoopCounter" is not defined.
(idb) unset $catchforks
Warning: The debugger variable "$catchforks" was not unset because it is an
idb predefined variable
```

Scripting or Repeating Previous Commands. Expert Debugging

To repeat the last command line, enter two exclamation points (`!`) or press the Enter key. You can also enter **`!-1`**.

```
command_repetition_command

        : !!

        | !  integer

        | !- integer

        | !  string
```

To repeat a command line entered during the current debugging session, enter an exclamation point followed by the integer associated with the command line. (Use the **`history`** command to see a list of commands used.) For example, to repeat the seventh command used in the current debugging session, enter **`!7`**. Enter **`!-3`** to repeat the third-to-the-last command. See also History replacement of the line.

To repeat the most-recent command starting with a string, use the last form of the command. For example, to repeat a command that started with **`bp`**, enter **`!bp`**.

Following are other ways to reuse old commands and save typing effort:

- Use a completely empty line to repeat the last command but not the last line, which could have been a comment or a syntactically invalid attempt at a command. Immediately pressing the Enter key is the recommended way of doing this.
- Use command line editing to recall and modify commands you have already entered.
- It is often useful to have a text editor up and running while debugging, and use it to assemble short scripts that you can copy and paste to the debugger. Keep a separate text file that has such scripts in it, as well as other notes you wish to keep. This provides continuity from one debugging session to the next, and from one day to the next.

If you place commands in a file, you can execute them directly from the file rather than cutting and pasting them to the terminal. For example:

execute_commands_from_file_command

: **source** *filename*

| **playback** **input** *filename*

Use the **source** command to read and execute commands from a file. (You can also execute debugger commands when you invoke the debugger by creating an initialization file named .dbxinit.) These commands can be nested, and as each comes to an end, reading resumes from where it left off in the previous file.

Be aware, however, that blank lines in these files do not repeat the last command, unlike what blank lines do when entered from the terminal. Format the commands as if they were entered at the debugger prompt.

Use the pound character (#) to create comments to format your scripts.

The following is an example debugger script:

```
(idb) sh cat ../src/myscript
step
where 2
```

The following example shows how to execute it:

```
(idb) run
[1] stopped at [int main(void):187 0x08052ec4]
    187     nodeList.append(newNode);   {static int somethingToReturnTo;
somethingToReturnTo++; }
(idb) source ../src/myscript
stopped at [void List<Node>::append(class Node* const):148 0x0804c55e]
    148     if (! firstNode)
>0  0x0804c55e in ((List<Node>*)0xbfffce78)-
>List<Node>::append(node=0x805e5f8) "src/x list.cxx":148
#1  0x08052edc in main() "src/x_list.cxx":187
```

When a command file is executed, the value of the `$pimode` debugger variable determines whether the commands are echoed. If the `$pimode` variable is set to 1, commands are echoed; if `$pimode` is set to 0 (the default), commands are not echoed. The debugger output resulting from the commands is always echoed.

Recording Input and Output

To help you make command files, as well as to help you see what has happened before, the debugger can write both its input and its output to files, as follows:

record_command

: **record io** [*filename*]

| **record input** [*filename*]

| **record** output [*filename*]

| **unrecord io**

| **unrecord input**

| **unrecord output**

Use **record input** to save debugger commands to a file. The commands in the file can be executed using the **source** command or the **playback input** command.

If no file name is specified, the debugger creates a file with a random file name in `/tmp` as the record file. The debugger issues a message giving the name of that file.

To stop recording debugger input or output, redirect as shown in the following example, use the appropriate version of the **unrecord** command, or exit the debugger:

```
(idb) record input /dev/null
(idb) record output /dev/null
```

The following example shows how to use the **record input** command to record a series of debugger commands in a file named `myscript`:

```
(idb) record input myscript
(idb) stop in main
[#1: stop in int main(void)]
(idb) run
[1] stopped at [int main(void):182 0x08052e0f]
    182     List<Node> nodeList;
(idb) unrecord input
```

This example results in the following recorded input in `myscript`:

```
(idb) sh cat myscript
stop in main
run
unrecord input
```

The **`record output`** command saves the debugger output to a file. The output is simultaneously written to `stdout` (normal output) or `stderr` (error messages). For example:

```
(idb) record output myscript
(idb) stop in List<Node>::append
[#2: stop in void List<Node>::append(class Node* const)]
(idb) cont
[2] stopped at [void List<Node>::append(class Node* const):148 0x0804c55e]
    148     if (! firstNode)
(idb) cont to 156
stopped at [void List<Node>::append(class Node* const):156 0x0804c5db]
    156 }
(idb) unrecord output
```

After the above commands are executed, `myscript` contains the following:

```
(idb) sh cat myscript
[#2: stop in void List<Node>::append(class Node* const)]
[2] stopped at [void List<Node>::append(class Node* const):148 0x0804c55e]
    148     if (!_firstNode)
stopped at [void List<Node>::append(class Node* const):156 0x0804c5db]
    156 }
```

The **`record io`** command saves both input to and output from the debugger. For example:

```
(idb) record io myscript
(idb) stop in main
[#1: stop in int main(void) ]
(idb) run
[1] stopped at [int main(void):12 0x120001130]
     12 int i;
(idb) quit
% cat myscript
(idb) stop in main
[#1: stop in int main(void) ]
(idb) run
[1] stopped at [int main(void):12 0x120001130]
     12 int i;
(idb) quit
```

If input or output is already being recorded, a new **`record input`** command will close the old file and record to a new one, rather than record simultaneously to two files. In that connection, note that **`record io`** is equivalent to the combination of **`record input`** and **`record output`**, and will cause any open recording files to be closed.

Note that the prompt itself is only recorded for **`record io`**.

Viewing the Command History

You can see all the commands you have already entered by using the **`history`** command. Use `history_number` to indicate how many commands to show, starting with the most recent. If you do not specify `history_number`, the value of the debugger variable `$historylines` (default 20) is used to determine the number of previous commands shown. See also History replacement of the line.

history_command

```
        : history [ integer_constant ]
```

For example:

```
(idb) history 7
18: stop at 182
19: run
20: stop at 103
21: delete 1
22: cont
23: print "history EXAMPLE START"
24: history 7
```

Defining Aliases (DBX mode only)

You can extend the set of debugger commands by defining aliases.

When the debugger is tokenizing a command line, it expands aliases and then retokenizes the expansion.

alias_command

```
        : alias [ alias_name ]

        | alias alias_name [ ( argument_name, ...) ] string

        | unalias alias_name
```

The following example shows how to define and use an alias:

```
(idb) alias cs
alias cs is not defined
(idb) alias cs "stop at 186; run"
(idb) cs
[#1: stop at "x list.cxx":186 ]
[1] stopped at [int main(void):186 0x120002420]
    186     IntNode* newNode = new IntNode(1);
```

The following example further modifies the **cs** alias to specify the breakpoint's line number when you enter the **cs** command:

```
(idb) alias cs (x) "stop at x; run"
(idb) cs(186)
[#2: stop at "x list.cxx":186 ]
Process has exited
[2] stopped at [int main(void):186 0x120002420]
    186     IntNode* newNode = new IntNode(1);
```

Note:

> No warning is given if the `alias_name` already has a definition as an alias. The old definition will be replaced by the new one.

Use the **unalias** command followed by an alias name to delete the specified alias.

Executing Shell Commands

You can have the debugger execute a call to the operating system's `system` function. This function is documented in system(3). The call results from the **sh** (dbx) or **shell** (gdb) commands.

DBX Mode

```
      execute_shell_command

            : sh string
```

For example, you can execute a system command through a shell from the debugger by issuing the following command:

```
(idb) sh uname -s
Linux
(idb)
```

To execute more than one command at the specified shell, spawn a shell as follows, for example:

```
(idb) sh csh -f
% ls out
out
% ls *.b
recio.b
stdio.b
% exit
(idb)
```

GDB Mode

```
    execute_shell_command

            : shell string
```

For example:

```
(idb) shell uname -s
Linux
(idb)
```

To execute more than one command at the shell, spawn a shell as follows:

```
(idb) shell bash --norc
$ ls out
out
$ ls *.b
recio.b
stdio.b
$ exit
(idb)
```

Invoking Your Editor (DBX Mode Only)

You can use the **`edit`** command to invoke the editor defined by the `EDITOR` environment variable.

```
edit_file_command

        : edit [ string ]
```

The editor is given the string as the file name to edit. If no file name is specified, the editor is given the current file. If no current file exists, the editor is started without a file.

If the `EDITOR` environment variable is undefined, the debugger invokes the `vi` editor.

The following example invokes the Emacs* editor on the file `chars.c`:

```
(idb) sh printenv EDITOR
notepad
(idb) file
chars.c
(idb) edit
```

The following example invokes the `nedit` editor on the file `~/foo/bar.f`:

```
(idb) sh printenv EDITOR
nedit
```

```
(idb) edit ~/foo/bar.f
```

Context for Executing Commands

This section describes context for executing commands and discusses the following topics:

- Loading an executable file
- Creating a new process
- Attaching the Debugger to an existing process
- Multiple processes
- Multiple call frames, threads, and sources

Loading an Executable File

Specifying an executable file on the command line or executing the **`load`** (dbx) or **`file`** (gdb) command identifies the current program.

Note:

In the background, the debugger immediately creates a process executing the program, stalls it, and uses it to answer questions about which dynamic libraries are mapped, and so on. This process never continues, and is killed when:

- The debugger exits
- You unload this executable file

Creating a New Process

Using the **`run`** command causes the debugger to create a new process running the loaded program.

Attaching the Debugger to an Existing Process

Specifying `pid <process id>` on the command line or executing the `attach` command causes the debugger to take control of the process indicated.

Multiple Processes

The debugger supports concurrently debugging multiple processes at the same time, but at any given time is only operating on a single process, known as the current process. The debugger variable `$curprocess` contains the process id for this process. Naming and switching the debugger between processes is described in Multiprocess Debugging.

Multiple Call Frames, Threads, and Sources

Processes contain one or more threads of execution. The threads execute functions. Functions are sequences of instructions that are generated by compilers from source lines within source files.

As you enter the debugger commands to manipulate your process, it would be very tedious to have to repeatedly specify which thread, source file, and so on, to which you wish the command to be applied. To prevent this, each time the debugger stops the process, it re-establishes a static context and a dynamic context for your commands. The components of the static context are independent of this run of your program; the components of the dynamic context are dependent on this run.

The components of these contexts can be displayed as debugger variables or by other commands:

- The static context consists of the following:
  - Current program - **`listobj`** (dbx) or **`info file / info sharedlibrary`** (gdb)
  - Current file - **`print`** `$curfile`
  - Current line - **`print`** `$curline`
- The dynamic context consists of the following:
  - Current call frame - **`where`**
  - Current process - **`print`** `$curprocess`
  - Current thread - **`print`** `$curthread`
  - The thread executing the event that caused the debugger to gain control of the process - **`thread`**

The debugger keeps the components of the static and dynamic contexts consistent as the contexts change. The current file and line are determined by where the debugger stops the process, but the dynamic context can be changed directly, using the **`up/down`**, **`func`** (dbx) or **`frame`** (gdb), **`process`** (dbx), and **`thread`** commands. The program can be unloaded using the **`unload`** (dbx) or **`file`** (gdb) command.

Running the Program Under Debugger Control

The program that is to be debugged can be run directly from the debugger, or it can be started separately from the debugger and then "attached to" by the debugger.

In some situations the program requires more context, or a process may already have been created. It could be part of a pipe, perhaps it is a long-running process, or perhaps it is created from a shell script or makefile . Hence, the following situations are possible:

- Running your program as a child process of the debugger process.
- Using the debugger's ability to attach to any process to which it has access.

Running the Program in the Debugger

If your program can be run using a simple command line, you can load it when you start the debugger. For example:

DBX Mode

```
% idb a.out
```

or

```
% idb
(idb) load a.out
```

GDB Mode

```
% idb -gdb a.out
```

or

```
% idb -gdb
(idb) file a.out
```

Attaching to a Running Program

If your program is already running, you can "attach to" the program to debug it. To attach to a running program you must know its process identifier or PID. In the examples that follow, the PID of the program is 8492.

Examples:

DBX Mode

```
% idb -pid 8492 a.out
```

or

```
% idb
(idb) attach 8492 a.out
```

GDB Mode

```
% idb -gdb -pid 8492 a.out
```

or

```
% idb -gdb (idb) file a.out (idb) attach 8492
```

Once you attach to a process, the process continues execution until it raises a signal that the debugger intercepts (for example, `SEGV`). If you have set the `$stoponattach` preference variable, it stops immediately.

One method you can use to attach to a process at a predictable location is to add a looping function to your program that keeps executing until the debugger takes control and you interrupt it (for example, with Ctrl+C). For example:

1. Add code such as the following to your application:

```
volatile int endStallForDebugger=0;
void stallForDebugger()
{
        while (!endStallForDebugger) ;
}
int main()
{
        ...
        stallForDebugger();
        ...
}
```

2. Run this version of your program.

3. Attach the debugger to the running process as described above.

4. Stop the program with Ctrl+C or by use of `$stoponattach`.

5. Use the debugger to assign to the `stallForDebugger` variable, and continue the execution of the process, so that it exits from the loop:

```
(idb) assign endStallForDebugger = 1
(idb) # set any needed breakpoints, and so on
(idb) cont
```

The `load`, `unload`, and `file` Commands

Using the **`load`** (dbx) and **`file`** (gdb) commands, you specify which executable file you intend to execute under control of the debugger. (This is done automatically when you give the debugger a file name on the command line.) These commands read the symbolic information for an executable file and the shared libraries it uses (if available). Objects compiled without debug information will not have symbols to load. The **`load`** (dbx) command can optionally load a core file.

DBX Mode

load_command

: **load** *filename* [*filename*]

The second file name is used to specify a core file. If you specify a core file, the debugger acts as though it is attached to the process at the point just before it died, except that you cannot execute commands that require a runnable process, such as commands that try to continue the process or evaluate function calls.

Examples:

```
% idb /home/user/examples/x_list
```

or

```
(idb) listobj
Program is not active
(idb) load /home/user/examples/x list
Reading symbolic information from /home/user/examples/x list...done
(idb) listobj
    section          Start Addr           End Addr
-------------------------------------------------------------------------------
/home/user/examples/x list
     .text           0x8048000           0x8056e3f
     .data           0x8057000           0x805deeb
      .bss           0x805deec           0x805dfb3
/lib/libdl-2.3.2.so
     .text          0xb7386000          0xb7387dc3
     .data          0xb7388dc4          0xb7388f53
      .bss          0xb7388f54          0xb7388f73
/lib/tls/libc-2.3.2.so
     .text          0xb7389000          0xb74b94f5
     .data          0xb74ba500          0xb74bcfdb
      .bss          0xb74bcfdc          0xb74bfa8b
/nfs/cmplr/icc-9.0.031/lib/libunwind.so.5
     .text          0xb74c0000          0xb74c433f
     .data          0xb74c5340          0xb74c5abb
      .bss          0xb74c5abc          0xb74c5c1b
/nfs/cmplr/icc-9.0.031/lib/libcxa.so.5
     .text          0xb74c6000          0xb74e62b3
     .data          0xb74e7000          0xb74eed37
      .bss          0xb74eed38          0xb74eeeaf
/nfs/cmplr/icc-9.0.031/lib/libcprts.so.5
     .text          0xb74ef000          0xb758d933
     .data          0xb758e000          0xb75b422f
      .bss          0xb75b4230          0xb75b4c27
/lib/tls/libm-2.3.2.so
     .text          0xb75b5000          0xb75d5dbf
     .data          0xb75d6dc0          0xb75d6f43
      .bss          0xb75d6f44          0xb75d6f8f
/lib/ld-2.3.2.so
     .text          0xb75eb000          0xb75fffcf
     .data          0xb7600000          0xb7600533
      .bss          0xb7600534          0xb7600753
```

GDB Mode

file_command

```
      : file [ filename ]
```

If *filename* is specified, the debugger loads the specified executable. Without an argument, the debugger unloads the current executable file.

Example:

```
% idb -gdb /usr/examples/x_list
```

or:

```
(idb) info files
(idb) file /home/user/examples/x list
Reading symbols from /home/user/examples/x list...done.
(idb) info files
Symbols from "/home/user/examples/x list".
Unix child process:
Using the running image of child process 19438.
While running this, IDB does not access memory from...
Local exec file:
'/home/user/examples/x_list', file type <unknown>
0x8048000 - 0x8056e40 is .text
0x8057000 - 0x805deec is .data
0x805deec - 0x805dfb4 is .bss
```

Loading a process both creates the debugger's knowledge of it and makes it the current process that the debugger is controlling.

The opposite of loading an executable file is unloading an executable file, when the debugger removes all related symbol table information that the debugger associated with the process being debugged.

DBX Mode

unload_command

```
        : unload [ pid ,... ]

        | unload [ filename ]
```

pid

```
 : integer_constant
```

Process for unloading can be specified by either a process id or an executable file name.

```
 (idb) listobj
    section         Start Addr           End Addr
```

```
-----------------------------------------------------------------------------
/home/user/examples/x list
     .text          0x8048000          0x8056e3f
     .data          0x8057000          0x805deeb
      .bss          0x805deec          0x805dfb3
/lib/libdl-2.3.2.so
     .text         0xb7386000         0xb7387dc3
     .data         0xb7388dc4         0xb7388f53
      .bss         0xb7388f54         0xb7388f73
/lib/tls/libc-2.3.2.so
     .text         0xb7389000         0xb74b94f5
     .data         0xb74ba500         0xb74bcfdb
      .bss         0xb74bcfdc         0xb74bfa8b
/nfs/cmplr/icc-9.0.031/lib/libunwind.so.5
     .text         0xb74c0000         0xb74c433f
     .data         0xb74c5340         0xb74c5abb
      .bss         0xb74c5abc         0xb74c5c1b
/nfs/cmplr/icc-9.0.031/lib/libcxa.so.5
     .text         0xb74c6000         0xb74e62b3
     .data         0xb74e7000         0xb74eed37
      .bss         0xb74eed38         0xb74eeeaf
/nfs/cmplr/icc-9.0.031/lib/libcprts.so.5
     .text         0xb74ef000         0xb758d933
     .data         0xb758e000         0xb75b422f
      .bss         0xb75b4230         0xb75b4c27
/lib/tls/libm-2.3.2.so
     .text         0xb75b5000         0xb75d5dbf
     .data         0xb75d6dc0         0xb75d6f43
      .bss         0xb75d6f44         0xb75d6f8f
/lib/ld-2.3.2.so
     .text         0xb75eb000         0xb75fffcf
     .data         0xb7600000         0xb7600533
      .bss         0xb7600534         0xb7600753
(idb) unload
(idb) listobj
Program is not active
```

GDB Mode

Use the **`file`** command without an argument to unload an executable file.

```
(idb) info files
Symbols from "/home/user/examples/x list".
Unix child process:
Using the running image of child process 19436.
While running this, IDB does not access memory from...
Local exec file:
'/home/user/examples/x list', file type <unknown>
0x8048000 - 0x8056e40 is .text
0x8057000 - 0x805deec is .data
0x805deec - 0x805dfb4 is .bss
(idb) file
No executable file now.
No symbol file now.
(idb) info files
```

The `run` and `rerun` Commands

After you have loaded a program, you can create a process executing this program using either of the following forms of the **run** command:

DBX Mode

```
run_command

        : run    [ argument_string ] [ io_redirection ... ]

        | rerun  [ argument_string ] [ io_redirection ... ]
```

If the **rerun** command is specified without arguments, the arguments and io_redirection arguments of the most recent **run** command entered with arguments are used. If there was no previous **run** command, the **rerun** command defaults to **run**.

GDB Mode

```
run_command

        : run [ argument_string ] [ io_redirection ... ]

        | r   [ argument_string ] [ io_redirection ... ]

arg_commands

        : set_args_command

        | show_args_command

set_args_command

        : set args [ argument_string ] [ io_redirection ... ]

show_args_command

        : show args
```

If the **run** (or **r**) command does not specify any arguments, default arguments are used. Default arguments are specified by the previous **run** command with arguments or by **set args** command. To view default arguments, use the **show args** command.

Note:

> The **set args** commands does not affect process currently running. New arguments will affect only the next run.

If the last modification time or size of the binary file or any of the shared objects used by the binary file has changed since the last **run** or **rerun** (dbx) command was issued, the debugger automatically rereads the symbol table information. If this happens, the old breakpoint settings may no longer be valid after the new symbol table information is read.

The `argument_string` provides both the `argc` and `argv` for the created process in the same way a shell does.

The debugger breaks up the *`argument_string`* into words, and supports several shell features, including tilde (`~`) and environment variable expansion, wildcard substitution, single quote (`'`), double quote (`"`), and single character quote (`\`).

The *`io_redirection`* argument allows you to change `stdin`, `stdout`, and `stderr`, which are otherwise inherited from the debugger process:

`io_redirection`

: `<` *`filename`*

| `>` *`filename`*

| **`1`**`>` *`filename`*

| **`2`**`>` *`filename`*

| `>`**`&`** *`filename`*

The various forms have the same effect as in the `csh`(1) shell.

Note:

Although the grammar currently allows more than the following forms of redirection, you should only use the following forms because the grammar may change in a future release of the debugger.

```
     > filename                Redirect stdout
    1> filename                Redirect stdout
    2> filename                Redirect stderr
    >& filename                Redirect stdout and stderr
    1> filename 2> filename    Redirect stdout and stderr to different files
```

Examples:

DBX Mode

```
(idb) stop at 182
[#1: stop at "src/x list.cxx":182]
(idb) run -s > prog.output
[1] stopped at [int main(void):182 0x08052e0f]
    182     List<Node> nodeList;
```

GDB Mode

```
(idb) break main
Breakpoint 1 at 0x8052e0f: file src/x list.cxx, line 182.
(idb) show args
```

```
Argument list to give program being debugged when it is started is "".
(idb) run
Starting program: /home/user/examples/x list
Breakpoint 1, main () at src/x list.cxx:182
182     List<Node> nodeList;
(idb) continue
Continuing.
The list is:
Node 1 type is integer, value is 1
Node 2 type is compound, value is 12.345
       parent  type is integer, value is 2
Node 3 type is compound, value is 3.1415
       parent  type is integer, value is 7
Node 4 type is integer, value is 3
Node 5 type is integer, value is 4
Node 6 type is compound, value is 10.123
       parent  type is integer, value is 5
Destroying nodes...
All nodes destroyed
Program exited normally.
(idb) set args -s > prog.output
(idb) show args
Argument list to give program being debugged when it is started is "-s >
prog.output".
(idb) run
Starting program: /home/user/examples/x list
Breakpoint 1, main () at src/x list.cxx:182
182     List<Node> nodeList;
```

The `kill` Command

You can kill the current process:

```
kill_command

        : kill
```

Killing a process leaves the debugger running. Any breakpoints previously set are retained. You can execute the program again later using the **`rerun`** (dbx) or the **`run`** (gdb) commands without loading it again. For example:

DBX Mode

```
(idb) show process
Current Process: localhost:19307 (/home/user/examples/x list) paused.
(idb) kill
Process has exited
(idb) rerun
[1] stopped at [int main(void):182 0x08052e0f]
    182     List<Node> nodeList;
```

GDB Mode

```
(idb) info program
Using the running image of child process 19440.
Program stopped at 0x8052e0f.
It stopped at breakpoint 1.
(idb) kill
```

```
Program exited normally.
(idb) run
Starting program: /home/user/examples/x list
Breakpoint 1, main () at src/x list.cxx:182
182     List<Node> nodeList;
```

The `attach` and the `detach` Commands

If a process already exists, you can have the debugger attach to it:

DBX Mode

attach_command

: **attach** *pid* [*filename*]

GDB Mode

attach_command

: **attach** *pid*

Note:

The **`attach`** command requires the name of executable to be specified before attaching to the process. Use the **`file`** command or shell command line to specify the filename.

The process is specified by its `pid`:

pid

: *expression*

For example:

DBX Mode

```
(idb) attach 12345 a.out
```

GDB Mode

```
(idb) file a.out
Reading symbols from a.out...done.
(idb) attach 12345
```

Note that you must specify the file name. The file name should be the executable file that is executing or a duplicate copy of it. You may omit the executable file name only if the debugger already has the file loaded. It means that if a file name is not specified, the current executable is used. If the executable contains symbolic debug information it will be read by the debugger during the attach.

Attaching to a process both creates the debugger's knowledge of it and makes it the current process that the debugger is controlling. The process continues execution until it raises a signal that the debugger intercepts. Usually you do this by pressing Ctrl+C or by using the shell command **`kill`** in another window. Any other mechanism for raising a signal within the process will also do. You can set the debugger variable `$stoponattach` to `1` to direct the debugger to immediately stop any process that it attaches to.

```
(idb) ^C
Interrupt (for process)
Stopping process localhost:16077 (loop.out).
Thread received signal INT
stopped at [int main(void):3 0x120001100]
      3     while (1) ;
```

The opposite of attaching to a process is detaching from a process. When you detach the debugger from a process, all breakpoints are removed and the process continues to run, but the debugger can no longer identify or control it:

DBX Mode

```
detach_command

        : detach pid ,...
```

For example:

```
 (idb) detach 12345, 789
```

GDB Mode

```
detach_command

        : detach
```

The **`detach`** command detaches the debugger from a current process and, therefore, does not require *pid*.

Controlling the Process Environment

You can set and unset environment variables for processes created in the future to set up an environment different from the environment of the debugger and from the shell from which the debugger was invoked. When set, the environment variables apply to all new processes you debug.

Note:

> The environment commands have **no** effect on the environment of any currently running process. The environment commands do not change or show the environment variables of the debugger or of the current process. They only affect the environment variables that will be used when a new process is created.

```
environment_variable_command

        : show_environment_variable_command

        | set_environment_variable_command

        | unset_environment_variable_command
```

To print either all the environment variables that are currently set or a specific one, use a *show_environment_variable_command*.

DBX Mode

```
show_environment_variable_command

        : printenv [ environment_variable_name ]

        | export

        | setenv
```

Note:

> The **`export`** and **`setenv`** commands without any arguments are equivalent.

GDB Mode

```
show_environment_variable_command

        : show environment [ environment_variable_name ]

        | show env         [ environment_variable_name ]
```

The **`show env`** is a synonym for the **`show environment`** command.

If you do not specify the name of the environment variable to show, the debugger will print all the environment variables.

To add or change an environment variable, use a *set_environment_variable_command*. If the *environment_variable_value* is not specified, the environment variable value is set to `""`.

DBX Mode

```
set_environment_variable_command

        : export environment_variable_name  =

environment_variable_value

        | setenv environment_variable_name

environment_variable_value
```

GDB Mode

```
set_environment_variable_command

        : set environment environment_variable_name [ [ = ]

environment_variable_value ]

        | set env         environment_variable_name [ [ = ]

environment_variable_value ]


environment_variable_value

        : string
```

To remove an environment variable, use the following commands:

DBX Mode

```
unset_environment_variable_command

        : unsetenv environment_variable_name

        | unsetenv *
```

If an asterisk (`*`) is specified, all environment variables are removed.

GDB Mode

```
unset_environment_variable_command

        : unset environment environment_variable_name

        | unset env         environment_variable_name
```

Note:

> There is no command to simply return to the initial state the environment variables had when the debugger started. You must use *set_environment_variable* commands and *unset_environment_variable* commands appropriately.

For example:

DBX Mode

```
(idb) printenv TOOLDIRECTORY
Error: Environment variable 'TOOLDIRECTORY' was not found in the environment.
(idb) setenv TOOLDIRECTORY /home/user/examples/tools
(idb) printenv TOOLDIRECTORY
TOOLDIRECTORY=/home/user/examples/tools
```

GDB Mode

```
(idb) show environment TOOLDIRECTORY
Environment variable "TOOLDIRECTORY" not defined.
(idb) set environment TOOLDIRECTORY /home/user/examples/tools
(idb) show environment TOOLDIRECTORY
TOOLDIRECTORY=/home/user/examples/tools
(idb) unset environment TOOLDIRECTORY
(idb) show environment TOOLDIRECTORY
Environment variable "TOOLDIRECTORY" not defined.
```

Multiprocess Debugging

The debugger can find and control more than one process at a time. The debugger can find and control a process for one of the following reasons:

- It created the process.
- It attached to the process.

- A process that it was controlling executed a fork, and `$catchforks` was set.

At any one time, you can examine or execute only one of the processes that the debugger controls. The rest are stalled. You must explicitly switch the debugger to the process you want to work with, stalling the one it was controlling:

```
multiprocess_command

        : show_process_command

        | switch_process_command
```

You can show the processes the debugger controls:

show_process_command

: **show process** [*all*]

| **process**

all

: **all**

| *****

For example:

```
(idb) show process
>localhost:20986 (/home/user/examples/x_list) loaded.
```

You can explicitly command the debugger to control a different process:

switch_process_command

: **process** *pid*

| **process** *filename*

The process you are switching away from remains stalled until either the debugger exits or until you switch to it and continue it.

The following example creates two processes and switches from one to the other:

```
(idb) process
There is no current process.
You may start one by using the `load' or `attach' commands.
(idb) load x list
Reading symbolic information from
/home/user/examples/x.x processes/x list...done
(idb) process
>localhost:20988 (/home/user/examples/x.x_processes/x_list) loaded.
(idb) set $old process = $curprocess
(idb) printf "$old process=%d", $old process
$old process=20988
(idb) load x segv
Reading symbolic information from
/home/user/examples/x.x processes/x segv...done
(idb) process
 localhost:20988 (/home/user/examples/x.x processes/x list) loaded.
>localhost:20990 (/home/user/examples/x.x processes/x segv) loaded.
(idb) process 20988
(idb) process
>localhost:20988 (/home/user/examples/x.x processes/x list) loaded.
 localhost:20990 (/home/user/examples/x.x_processes/x_segv) loaded.
```

Both the **load** (dbx) command and the **attach** (dbx) command switch the debugger to the process on which they operate.

Processes That Use fork()

The debugger has the following predefined variables that you can set for debugging a program that forks:

- `$catchforks` - When set to a non-zero value, this variable instructs the debugger to stop the child process on exit out of the `fork()` or `vfork()` calls. The parent process continues to run. The default is 0 (zero).
- `$stopparentonfork` - When set to a non-zero value, this variable instructs the debugger to stop the parent process on exiting out of the `fork()` or `vfork()` calls after it forks a child process. The child process continues to run if `$catchforks` is 0; otherwise, it does not. The default is 0 (zero).
- `$catchforkinfork` - When set to a non-zero value, this variable instructs the debugger to stay in the fork routine after the fork and notifies you as soon as the forked process is created; otherwise, you are notified when the call finishes. You can debug forking processes before any "atfork" handlers are run by setting `$catchforkinfork`. Because the target stops inside the system call, you will need to issue **up** commands to get to user-written code. The default is 0 (zero).

When a fork occurs, the debugger sets the debugger variables `$childprocess` and `$parentprocess` to the child and parent process IDs, respectively.

In the following example, the debugger notifies you that the child process has stopped. The parent process continues to run.

```
(idb) set $catchforks = 1
(idb) run
Process 29027 forked.  The child process is 29023.
Process 29023 stopped on fork.
stopped at [int main(void):6 0x120001178]
      6  int pid = fork();
fork.c: I am the parent.
Process has exited with status 0
(idb) show process
>localhost:29028 (/home/user/examples/fork) loaded.
 localhost:29023 (/home/user/examples/fork) paused.
```

In the preceding example, note the following:

- The debugger indicates that the child process has stopped, and shows the line number at which it is stopped.
- The last two lines show that the child process has stopped and that the parent process has completed execution.

Continuing the previous example, the following shows how to switch the debugger to the child process. Listing the source code shows the source for the child process.

```
(idb) process $childprocess
(idb) show process
 localhost:29028 (/home/user/examples/fork) loaded.
>localhost:29023 (/home/user/examples/fork) paused.
(idb) list
      7
      8   if (pid == 0)
      9    {
     10      printf("fork.c: I am the child.\n");
     11    }
     12   else
     13    {
     14      printf("fork.c: I am the parent.\n");
     15    }
     16 }
```

In the preceding example, note the following:

- The first line switches the current process context to the child process.
- The right angle bracket indicates the current process.
- The `list` command lists the source code for the current process.

Note:

If you catch the child but not the parent, and the parent code tries to execute a wait on the child, the target will get stuck if you don't let the child run to completion. This happens because the parent will be running but making no progress, and the child is stopped by the debugger. For example:

```
(idb) set $catchforks = 1
(idb) set $stopparentonfork = 0
(idb) list
     10     int new pid = 0;
     11
     12     if (pid == 0) {
     13         printf( "fork.c: I am the child.\n" );
     14 fflush( stdout );
     15
     16     } else {
     17         printf( "fork.c: I am the parent, about to wait.\n" );
     18 fflush( stdout );
     19
     20 new pid = wait( &status );
     21
     22         printf( "fork.c: I am the parent, and my wait is finished\n"
);
     23
     24         if (new pid != pid )
     25             printf( "\tthere was some error\n" );
     26         else {
     27     if (WIFEXITED(status))
     28                 printf( "\tthe child terminated normally\n" );
     29
     30             else if (WIFSIGNALED(status))
(idb) sh cat ./x.c fork hang.txt
  If we 'cont' now, the process will fork; the child will be
  caught and the parent will run to the 'wait' call and wait
  for the child to terminate.
  At that time, the child will be under debugger control,
  but the current process will be the parent, which will be
  running but making no progress.  Only a Ctrl/C will allow
  further progress.
  The example program has set up another process to simulate
  a Ctrl/C by the user.  It will send SIGINT to the parent.
(idb) cont
Process 580893 forked.  The child process is 580851.
Process 580851 stopped on fork.
stopped at [void test(void):9 0x120001318]
      9     int pid = fork();
fork.c: I am the parent, about to wait.
:
User is waiting here
:
:
Sending SIGINT to parent process
:
Thread received signal INT
stopped at [<opaque>   wait4(...) 0x3ff800d0918]
Information:  An <opaque> type was presented during execution of the previous
command.  For complete type information on this symbol, recompilation of the
program will be necessary.  Consult the compiler man pages for details on
producing full symbol table information using the '-g' (and '-gall' for cxx)
flags.
(idb) where
>0  0x3ff800d0918 in   wait4(...) in /usr/shlib/libc.so
#1  0x3ff800d668c in   wait(...) in /usr/shlib/libc.so
#2  0x120001398 in test() "c fork hang.c":20
#3  0x120001528 in main() "c fork hang.c":71
#4  0x1200012a8 in   start(...) in /home/user/examples/c fork hang
(idb) show process
>localhost:580893 (/home/user/examples/c fork hang) paused.
   \_localhost:580851 (/home/user/examples/c_fork_hang) paused.
```

Processes That Use exec()

Set `$catchexecs` to 1 to instruct the debugger to stop the process and notify you when an `exec` occurs. The process stops before executing any user program code or static initializations. You can debug the newly executed process. The debugger keeps a history of the progression of the executed files.

In the following scenario, you set the predefined variables `$catchforks` and `$catchexecs` to 1. The debugger will notify you when an execution occurs. Because `$catchforks` is set, you will also be tracking the child process and, therefore, you will be notified of any `exec` in the child process.

The following example shows an `exec` occurring on the current context and the child process stopped on the run-time loader entry point:

```
(idb) set $catchforks = 1
(idb) set $catchexecs = 1
(idb) run
Process 14839 forked.  The child process is 14835.
Process 14835 stopped on fork.
stopped at [int main(void):8 0x1200011f8]
      8   if ((pid = fork()) == 0)
```

```
x exec.c: I am the parent.
Process has exited with status 0
(idb) show process
>localhost:14918 (x exec) loaded.
 localhost:14835 (x_exec) paused.
(idb) process $childprocess
(idb) list 6:13
      6   int pid;
      7
>     8   if ((pid = fork()) == 0)
      9       {
     10       printf("About to exec \n");
     11       fflush(stdout);  /* Make sure the output gets out! */
     12       execlp("announcer", "announcer", NULL);
     13       printf("After exec \n");
     14       }
     15   else
     16      {
     17       printf("x exec.c: I am the parent.\n");
     18      }
(idb) cont
About to exec
The process 14835 has execed the image "./announcer".
Reading symbolic information ...done
stopped at [ 0x3ff8001bf48]
      5     printf("announcer.c: I am here!! \n");
```

Note the following:

- Use `process $childprocess` to set the current process context to the child process.
- Listing the source code, you can see the process is almost ready to execute.
- The debugger notifies you when the `exec` occurs.
- The child process is stopped on the run-time loader entry point. The source display shows the code in the main routine.

Core File Debugging

When the operating system encounters an unrecoverable error, for example, a segmentation violation (SEGV), the system creates a file named `core` and places it in the current directory. The core file is not an executable file; it is a snapshot of the state of your process at the time the error occurred. It allows you to analyze the process at the point it crashed. For more information on core file debugging, see Debugging Core Files

Locating the Site of a Problem. Overview

To determine why a problem is happening, you usually want to execute your program up to or just before the point at which you observe the first evidence of the problem. Then you can examine the internal state of your program and try to identify something that explains the visible problem. Possibly you will see right away how the problem occurs, in which case you are finished debugging. You then correct your program, recompile, relink, and confirm that the correction works as intended.

Often, you will see something about the program state that is wrong but you will not see how it got that way. In that case, you need to make a guess at where the mistake might have occurred. Then, repeat this whole process, trying to stop at or just before the possible trouble point.

For simple problems, it may be easy to describe the conditions under which you want to stop the program; for example, "the first time `traverse` is called" or "when `division_by_zero` occurs". Other situations may require either more complex descriptions or repeated trial-and-error attempts to discover the critical information needed to solve your problem.

Breakpoints provide the means by which you specify to the debugger an event or condition under which you want to intervene in the execution of your program and what actions you want the debugger to take when that event is detected.

You can define breakpoints based on:

- Reaching a certain place in your program (such as entering a certain function or reaching code for a particular source line number)
- Accessing the contents of a variable or other memory when it is either read or written
- Raising a specified signal

You can also enable, disable, or delete breakpoints.

Breakpoint commands include the following:

```
breakpoint_command

        : breakpoint_definition_command

        | simple_stop_command

        | signal_command

        | obsolete_breakpoint_definition_command

        | breakpoint_table_command
```

In this section, you will find information on

- breakpoints
- detectors
- breakpoints and C++
- special breakpoint commands
- obsolete breakpoint commands
- breakpoint tables

Breakpoints

This section discusses the following topics:

- Breakpoint Definitions
- Disposition
- The `quiet` Specifier
- Types of Detectors
- Thread Filter
- Logical Filter
- Breakpoint Actions
- When Multiple Breakpoints Trigger at Once
- Recursive Breakpoints
- Breakpoints and C++
- Special Breakpoint Commands
- Breakpoint Interactions with exec(), fork(), dlopen() and dlclose() System Calls
- Obsolete Breakpoint Commands (DBX Mode only)

Breakpoint Definitions

The following is a particularly common breakpoint:

DBX Mode

```
(idb) stop in main
[#1: stop in int main(void)]
```

GDB Mode

```
(idb) break main
Breakpoint 1 at 0x8052e0f: file src/x_list.cxx, line 182.
```

This command tells the debugger that when execution enters the function `main`, you want the debugger to suspend execution and return control to you.

The debugger responds to a breakpoint command by displaying how it recorded the request internally. The debugger assigns a number to the breakpoint (in this case, it is 1), which it uses later to refer to that breakpoint. The debugger does not just repeat the command as you entered it; it provides a more complete description of the function `main` to help you confirm that it has correctly identified the function you meant.

Later, after you cause the program to execute, if that event occurs, the debugger reports the event and then prompts you for what to do next. For example:

DBX Mode

```
(idb) run
[1] stopped at [int main(void):182 0x08052e0f]
    182     List<Node> nodeList;
```

GDB Mode

```
(idb) run
Starting program: /home/user/examples/x list
Breakpoint 1, main () at src/x list.cxx:182
182     List<Node> nodeList;
```

Both the event part and the action part of a breakpoint definition command consist of several subparts:

```
breakpoint_definition_command

            : disposition

            [ quiet ]

              detector

            [ thread_filter ]

            [ logical_filter ]

            [ breakpoint_actions ]
```

where the *`detector`*, *`thread_filter`* (if specified), and *`logical_filter`* (if specified) collectively specify the event part, and the *`disposition`*, **`quiet`** (if specified) and *`breakpoint_actions`* (if specified) collectively specify the action part.

Note:

Additional obsolete forms of breakpoint definition are retained only for backward compatibility with earlier versions of the debugger. These forms are explained later. The obsolete forms may be eliminated in a future release.

There are three distinct points in time at which a breakpoint definition has an effect:

1. When the command is entered

   The command is parsed, names and expressions that occur in any of the event parts are evaluated, and the breakpoint actions are parsed and checked for correctness (but not evaluated).

2. When the debugger initiates program execution

   For each breakpoint that is not disabled, appropriate modifications are made to the program to enable detection of the specified event.

3. When a detector triggers during program execution

   The thread filter specification (if present) and logical filter (if present) are evaluated to determine whether the breakpoint as a whole has triggered. If not, then execution is resumed (silently). If so, the breakpoint actions are performed, after which execution stops or resumes according to the specified disposition.

Disposition

```
disposition

        : stop

        | when
```

The **stop** command specifies that when the event specified by the breakpoint occurs and all processing for that breakpoint has been completed, the debugger should prompt for further commands.

The **when** command specifies that when the event specified by the breakpoint occurs and all processing for that breakpoint has been completed, the debugger may resume execution of the program. See the section When Multiple Breakpoints Trigger at Once for an explanation of how the debugger determines when to resume execution.

The quiet Specifier

By default, when an event is detected and the debugger determines that the breakpoint actions should be performed, the debugger prints a line that identifies the breakpoint, for example:

```
(idb) when in main { stop }
[#1: when in int main(void) { stop }]
(idb) run
[1] when [int main(void):182 0x08052e0f]
[1] stopped at [int main(void):182 0x08052e0f]
    182     List<Node> nodeList;
```

The optional **quiet** specifier tells the debugger to omit this information.

```
(idb) when quiet in main { stop }
[#11: when quiet in int main(void) { stop }]
(idb) run
(idb) list $curline:1
>   182     List<Node> nodeList;
```

Detectors

The debugger uses several kinds of detectors, each corresponding to a particular kind of event:

DBX Mode

```
detector

    : place_detector

    | watch_detector

    | signal_detector

    | unaligned_detector
```

A place detector specifies a place or location in your program. It can refer to the beginning of a function, a particular line in one of your source files, a specific value of the PC (program counter), or certain sets of these.

A watch detector specifies a variable or other memory locations that should be monitored to detect certain kinds of access (read, write, and so on).

A signal detector specifies a set of signals to be monitored.

An unaligned access detector specifies any kind of memory access using an unaligned access.

This section describes each type of detector.

Place Detectors

You can use place detectors to determine when execution reaches a particular place or location in your program:

DBX Mode

```
place_detector

        : in function_name

        | in all function_name

        | pc address_expression
```

| **at** *line_specifier*

| **every proc entry**

| **every procedure entry**

| **every instruction**

| *expression*

The **in** *function_name* detector specifies the event where execution reaches the entry of the named function. For example:

If the function name is ambiguous (more than one function can match the name in some languages, including C++), the debugger prompts you with a list of alternatives from which to choose.

```
(idb) stop in foo
Select from
----------------------------------------------------
     1 int C::foo(double*)
     2 void C::foo(float)
     3 void C::foo(int)
     4 void C::foo(void)
     5 None of the above
----------------------------------------------------
2
[#4: stop in void C::foo(float)]
```

If you choose the last option ("None of the above"), then no function is selected and no breakpoint is defined.

The **in all** *function_name* detector is the same as **in** *function_name* except that it specifies all of the functions that match the given name, whether one or more:

```
(idb) stop in all foo
[#3: stop in all foo]
```

The **pc** *address_expression* detector specifies the event where execution reaches the given machine address:

```
(idb) stop pc $pc
[#7: stop PC == 0x80534b4]
```

The **at** *line_specifier* detector specifies the event where code associated with a particular line of the source is reached:

```
(idb) stop at 190
[#8: stop at "src/x_list.cxx":190]
```

If no code is associated with the given line number, the debugger finds and substitutes the closest higher line number that has associated code.

The **`every procedure entry`** detector specifies that a breakpoint should be established for every function entry point in the program.

```
(idb) stop every procedure entry
[#9: stop every procedure entry]
```

Note:

> This command can be very time consuming because it searches your entire program - including all shared libraries that it references - and establishes breakpoints for every entry point in every executable image. This can also considerably slow execution of your program as it runs.

A disadvantage of this command is that it establishes breakpoints for hundreds or even thousands of entry points about which you have little or no information. For example, if you use **`stop every proc entry`** immediately after loading a program and then run it, the debugger will stop or trace over 100 entry points before reaching your main entry point. About the only thing that you can do if execution stops at most such unknown places is continue until some function relevant to your debugging is reached.

The **`every instruction`** detector specifies a breakpoint for every instruction in your entire program:

```
(idb) stop every instruction
[#10: stop every instruction]
```

When used with the **`stop`** disposition, a subsequent **`continue`** behaves essentially the same as a step by instruction command (see **`stepi`**).

When used with the **`when`** disposition, subsequent **`next`** and **`step`** commands allow you to trace all of the instructions that are executed as a result of those stepping commands. Beware that even when **`next`** is used to step over a called routine, the trace output includes all of the instructions that are executed within the called routine (and any routines that it calls).

Note:

> This command will slow execution of your program considerably.

The detector *`expression`* (that is, an expression not preceded by one of the keywords **`in`**, **`at`**, or **`pc`**) specifies either a function name or line number depending on how the expression is parsed and evaluated. An expression that evaluates to the name of a function is handled just like the equivalent command that uses **`in`** in the detector; otherwise, it is handled like the equivalent command that uses **`at`** in the detector.

Watch Detectors

You can use watch detectors to determine when a variable or other memory location is read or written and/or changed. Breakpoints with watch detectors are also known as *watchpoints*.

```
watch_detector

        : basic_watch_detector watch_detector_modifiers

basic_watch_detector

        : variable expression

        | memory start_address_expression

        | memory start_address_expression , end_address_expression

        | memory start_address_expression : byte_count_expression

watch_detector_modifiers

        : [ access_modifier ] [ within_modifier ]

access_modifier

        : write

        | read

        | changed

        | any

within_modifier

        : within function_name
```

You can specify a variable whose memory is to be watched, or specify the memory directly. The accesses that are considered can be limited to those that write (the default), read, write and actually change the value, or can include all accesses.

If you specify a variable, the memory to be watched includes all of the memory for that variable, as determined by the variable's type. The following example watches for write access to variable `_nextNode`, which is allocated in the 8 bytes at the address shown in the last line of the example:

```
(idb) whatis  nextNode
struct Node* Node:: nextNode
(idb) print "sizeof( nextNode) =", sizeof(( nextNode))
sizeof( nextNode) = 4
(idb) stop variable  nextNode write
[#3: stop variable _nextNode write]
```

The specified variable is watched. If "p" is a pointer, **`watch variable p`** will watch the content of the pointer, not the memory pointed to by "p". Use **`watch memory *p`** to watch the memory pointed to by "p".

If you specify memory directly in terms of its address, the memory to be watched is defined as follows:

- By default (no last address or size given), then 8 bytes beginning at the given start address:

  ```
  (idb) when memory & nextNode : 8 any
  [#4: when memory &_nextNode : 8 any]
  ```

- If an end address is given, then all bytes of memory from the start address to and including the end address:

  ```
  (idb) stop memory & nextNode, ((long)& nextNode) + 3 read
  [#5: stop memory &_nextNode, ((long)&_nextNode) + 3 read]
  ```

  This watches the 4 bytes specified on the command line.

- If you specify a byte count, then the given number of bytes starting at the given start address:

  ```
  (idb) stop memory & nextNode : 2 changed
  [#6: stop memory &_nextNode : 2 changed]
  ```

  This watches the 2 bytes specified on the command line for a change in contents.

If you specify the **`within`** modifier, then only those accesses that occur within the given function (but not any function it calls) are watched. For example:

```
(idb) whatis t
int t
(idb) stop variable t write within foo
[#2: stop variable t write within void C::foo(void)]
(idb) cont
[2] Address 0x0804d5c8 was accessed at:
void C::foo(void): src/x overload.cxx
 [line 22, 0x08048721] foo+0x3:                    movl     $0x0, 0x804d5c8
0x0804d5c8: Old value = 0x0000000f
0x0804d5c8: New value = 0x00000000
[2] stopped at [void C::foo(void):22 0x0804872b]
     22 void C::foo()         { t = 0; state++; return; }
```

Signal Detectors

You can use signal detectors to determine when a particular signal is raised:

signal_detector

        : **signal** *signal_id* ,...

signal_id

        : *integer_constant*

        | **signal_name**

You can specify signals by numeric value or by their conventional operating system names, without or without the leading "SIG":

```
(idb) stop signal SEGV, 8, SIGINT
[#2: stop signal SEGV, 8, SIGINT]
```

If the debugger catches a signal event, then a subsequent simple **continue** will resume execution without raising the signal again in your process. However, a signal can be specified as part of the **continue** command to send the signal to your process when it resumes.

Unaligned Access Detectors

You can use an unaligned access detector to determine when an unaligned memory access occurs:

unaligned_detector

        : **unaligned**

Unaligned accesses may be automatically handled by the operating system. By default, an unaligned access results in an information message and then is corrected so that your program can continue. (You or your system administrator can choose a different default. See `uac`(1) for more information.) This message looks like this:

```
Unaligned access pid=30231 <x signals> va=0x11ffff791 pc=0x120001af4
ra=0x120001b84 inst=0xa0220000
```

You can request the debugger to detect unaligned accesses:

```
(idb) stop unaligned access
[#1: stop unaligned access]
(idb) run
Thread encountered Unaligned Access
[1] stopped at [int unalignedAccess(void):27 0x120001af8]
     27     return y;
```

Unaligned Access Detector (Linux* and Mac OS* Only)

Unaligned accesses are automatically handled and quietly corrected on Linux and Mac OS. The debugger cannot detect these events.

Thread Filter

A thread filter determines whether a detected event should be further considered for breakpoint processing.

```
thread_filter

        : thread thread_id ,...
```

The `thread_id` expressions are evaluated at the time the breakpoint command is entered, and each must yield an integer value.

A detected event is retained for further consideration only if the thread in which the event occurs matches one of the given threads. If not, the detection is quietly ignored.

If the `thread_filter` does not indicate a match, then any related logical filter is not evaluated.

Logical Filter

A logical filter determines whether a detected event should be further considered for breakpoint processing:

```
logical_filter

        : if expression
```

A detected event is retained for further consideration only if the given expression evaluates to `true`. If not, the detection is quietly ignored.

The expression is checked syntactically in the context of the place where the breakpoint command is given: it must be syntactically valid according to the language rules that apply there. However, the expression is not evaluated and names that occur in the expression need not be visible. After the syntax check, the expression is remembered in an internal form and is not rechecked later when it is evaluated.

If an error occurs when the expression is evaluated, for example, because a name in the expression is not defined, then the error is reported and the value of the expression is assumed to be `true`.

An error in the expression does not change the disposition. If continuation was specified, then that is still what occurs. For example:

```
(idb) when in List<Node>::append if x
[#5: when in void List<Node>::append(class Node* const) if x]
(idb) cont
Symbol "x" is not defined.
[Error while evaluating breakpoint condition - taken as true]
[5] when [void List<Node>::append(class Node* const):148 0x0804c55e]
Symbol "x" is not defined.
[Error while evaluating breakpoint condition - taken as true]
[5] when [void List<Node>::append(class Node* const):148 0x0804c55e]
[4] stopped at [int main(void):195 0x0805308e]
    195     nodeList.append(new IntNode(3)); {static int somethingToReturnTo;
somethingToReturnTo++; }
```

It is valid for a logical filter expression to contain a call to another routine in your program. Such a call is evaluated in the same way as if it occurred in a **`call`** or **`print`** command. However, execution of the called routine might result in triggering a breakpoint; this is called a recursive breakpoint.

Breakpoint Actions

The action part of a breakpoint command specifies actions to be performed when the event part has triggered (including passing any thread and/or logical filters):

```
breakpoint_actions

        : { action_list }

action_list

        : command

        | command;

        | command;...
```

Pay special attention to the following commands within the action list:

Special Commands

The following debugger commands behave differently in some fashion when used within a breakpoint action list:

- Simple stop

  A *`simple_stop_command`* is a stop without any detector or other parameters:

```
simple_stop_command

        : stop
```

If used within a breakpoint action list, it specifies that the disposition for the breakpoint should be to stop after completion of action list processing, even if the breakpoint was specified with the **`when`** disposition. If used outside an action list, it has no effect.

A simple stop command does not terminate action list processing; it only affects the disposition that applies later. For example:

```
(idb) when in List<Node>::print { stop; print "*** stopped ***" }
[#6: when in void List<Node>::print(void) { stop; print "*** stopped ***"
}]
(idb) cont
[6] when [void List<Node>::print(void):162 0x0804c5e6]
*** stopped ***
[6] stopped at [void List<Node>::print(void):162 0x0804c5e6]
    162     Node* currentNode = _firstNode;
```

- History

  The **`history`** command does not display commands that are performed as part of the action list of a breakpoint.

Commands to Use with Caution

You must be very careful when using some commands in breakpoint action lists. The following commands cause the debugger to resume execution of your program in the midst of action list processing:

- **`call`**
- **`continue`**
- **`goto`**
- **`next`**
- **`return`**
- **`step`**
- Any command that contains an expression whose evaluation involves calling a function in your program

It is easy in such cases to lose track of just what state breakpoint processing is really in or where you really are in your program. Such confusion may mislead or misdirect your debugging effort. For further discussion, see the section on Recursive breakpoints.

Commands to Avoid

You should avoid altogether some commands in breakpoint action lists. The following are commands that directly or indirectly change the process that the debugger is controlling:

- **`attach`** and **`detach`**
- **`run`** and **`rerun`**
- **`process`** with an argument

The debugger does not explicitly prohibit these commands, but their behavior within action lists is implementation-defined and subject to change from release to release. In very specialized cases, you may be able to obtain useful results by using them in action lists, but do not expect the same behavior over the long term.

When Multiple Breakpoints Trigger at Once

It is possible for multiple breakpoints to specify the same event, or possibly overlapping events. Thus, more that one breakpoint detector may trigger at the same time.

When more than one breakpoint detector triggers, the thread filters and logical filters of all the breakpoints involved are processed before the action part of any breakpoint is performed.

After the set of breakpoints that trigger is determined, the action parts of each of them are performed in an undefined order.

After all action parts are performed, execution of the program is resumed only if all of the breakpoints so specify in their disposition. If any one of them specifies a break, the debugger prompts you for further commands.

Recursive Breakpoints

The following commands cause the debugger to resume execution of your program while in the midst of action list processing:

- **`call`**
- **`continue`**
- **`goto`**
- **`next`**
- **`return`**
- **`step`**
- Any command that contains an expression whose evaluation involves calling a function in your program

In all of these cases, the debugger temporarily suspends processing of the current breakpoint to start your program executing again and then waits for that execution to complete. As long as no new breakpoint is triggered during that execution, all will be fine. However, if a new breakpoint triggers, in particular one with the **`stop`** disposition, then you may be prompted for new command input for the *recursive breakpoint* even before the initial breakpoint has completed.

Further, continuing execution may ultimately allow the original breakpoint to complete, at which time its disposition will come into play.

It is easy in such cases to lose track of just what state breakpoint processing is really in or where you really are in your program. Such confusion may mislead or misdirect your debugging effort. See the **`call`** command example, which locates suspended execution in nested function calls.

Breakpoints and C++

This section describes how to use breakpoints when debugging C++ programs.

Member Functions

Setting breakpoints in C++ member functions is illustrated using the following program:

```
(idb) list 3:25
      3 class C {
      4 public:
      5     void foo();
      6     void foo(int);
      7     void foo(float);
      8     int  foo(double *);
      9 };
     10
     11 C  o;
     12 C* p = new C;
     13 int t     = 0;
     14 int state = 1;
     15
     16 main(){
     17     t++;
     18     o.foo();
     19
     20 }
     21
     22 void C::foo()         { t = 0; state++; return; }
     23 void C::foo(int i)    { state++; return;  }
     24 void C::foo(float f)  { state++; return;  }
     25 int  C::foo(double *) { return state;}
```

You must name member functions in a way that makes them visible at the current position, according to the normal C++ visibility rules. For example:

```
(idb) stop in main
[#1: stop in int main(void)]
(idb) run
[1] stopped at [int main(void):17 0x08048704]
     17     t++;
(idb) stop in foo
Symbol "foo" is not defined.
foo has no valid breakpoint address
Warning: Breakpoint not set
```

If not positioned within a member function of a class, it is generally necessary to name the desired member function using type qualification, an object of the class type, or a pointer to an object of the class type. For example:

```
(idb) stop in C::foo
Select from
----------------------------------------------------
     1 int C::foo(double*)
     2 void C::foo(float)
     3 void C::foo(int)
     4 void C::foo(void)
     5 None of the above
----------------------------------------------------
3
[#5: stop in void C::foo(int)]
(idb) stop in o.foo
Select from
----------------------------------------------------
     1 int C::foo(double*)
     2 void C::foo(float)
     3 void C::foo(int)
     4 void C::foo(void)
     5 None of the above
----------------------------------------------------
1
[#6: stop in int C::foo(double*)]
(idb) stop in p->foo
Select from
----------------------------------------------------
     1 int C::foo(double*)
     2 void C::foo(float)
     3 void C::foo(int)
     4 void C::foo(void)
     5 None of the above
----------------------------------------------------
4
[#7: stop in void C::foo(void)]
```

You can avoid the ambiguity associated with an overloaded function by specifying a complete signature for the function name. For example:

```
(idb) stop in C::foo(void)
[#8: stop in void C::foo(void)]
(idb) stop in C::foo(int)
[#9: stop in void C::foo(int)]
```

Templates and Instantiations

The debugger has no knowledge of templates that may occur in your program. However, you can usually debug template instantiations the same way as the equivalent non-instantiated class or function.

Debugging of template instantiations is illustrated using the following source text:

```
(idb) list 144:13
    144 template <class NODETYPE>
    145 void List<NODETYPE>::append(NODETYPE* const node)
    146 {
    147
    148     if (! firstNode)
    149          firstNode = node;
    150     else {
    151         Node* currentNode =  firstNode;
    152         while (currentNode->getNextNode())
    153             currentNode = currentNode->getNextNode();
    154 currentNode->setNextNode(node);
    155     }
    156 }
```

Normal debugging commands then apply to the instantiation (not the template as such):

```
(idb) whatis List<Node>::append
void List<Node>::append(class Node* const)
(idb) stop in List<Node>::append
[#1: stop in void List<Node>::append(class Node* const)]
(idb) run
[1] stopped at [void List<Node>::append(class Node* const):148 0x0804c55e]
    148     if (! firstNode)
(idb) where 2
>0  0x0804c55e in ((List<Node>*)0xbfffbaa8)-
>List<Node>::append(node=0x805e5f8) "src/x_list.cxx":148
#1  0x08052edc in main() "src/x_list.cxx":187
```

Exception Handlers

When working with exception handlers, you can set a breakpoint at the appropriate line to determine if an exception is thrown. In addition, you can set breakpoints in these functions that are part of the C++ library support for exceptions:

`terminate`
: Gains control when any unhandled exception occurs, which will result in program termination.

`unexpected`
: Gains control when a function containing an exception specification tries to throw an exception that is not included in that specification.

Note:

You can overwrite terminate and unexpected with your own exception handlers by using set_terminate and set_unexpected functions. This technique can be useful for debugging.

These special library functions are illustrated using the following source:

```
(idb) list 30:29
     30 // Throw an exception.  The "throw(int)" syntax tells the compiler
that
     31 // only integer exceptions can escape this method.  This will result
in
     32 // an unexpected exception from C++.
     33 //
     34 void throwAnException() throw(int)
     35 {
     36     throw "Bug";
     37 }
     38
     39 // Provide some depth to the stack, for demonstration purposes
     40 //
     41 void someOperation()
     42 {
     43     int z = unalignedAccess();  // Some tests ignore this exception
     44     throwAnException();
     45 }
     46
     47 main()
     48 {
     49     try {
     50         someOperation();
     51     }
     52     catch(char* str) {
     53         std::cout << "Caught exception [" << str << "]" << std::endl;
     54     }
     55     catch(...) {
     56         std::cout << "Caught something" << std::endl;
     57     }
     58 }
```

You can trace the flow of execution, as in the following:

```
(idb) stop at 52
[#1: stop at "x signals.cxx":52]
(idb) stop in all terminate
Symbol "terminate" is not defined.
No value for expression terminate
Warning: Breakpoint not set
(idb) stop in all unexpected
Symbol "unexpected" is not defined.
No value for expression unexpected
Warning: Breakpoint not set
(idb) run
Caught something
Process has exited with status 0
(idb) where
The "where" command has failed because there is no running program.
(idb) cont
The "cont" command has failed because there is no running program.
(idb) where
The "where" command has failed because there is no running program.
(idb) cont
The "cont" command has failed because there is no running program.
(idb) where
The "where" command has failed because there is no running program.
(idb) cont
```

```
The "cont" command has failed because there is no running program.
```

Special Signal Breakpoints

Signals are operating-system-defined events that can be handled by the debugger.

To handle signal events, you can use special breakpoint commands:

`catch` and **`ignore`** (DBX)

`info handle` and **`handle`** (GDB)

You can request the debugger to catch unaligned accesses.

If your program seems to be caught in a loop, you can press Ctrl+C.

The `catch` and `ignore` Commands

DBX mode

signal_command

        : *catch_command*

        | *ignore_command*

catch_command

        : **catch** [*signal_id*]

ignore_command

        : **ignore** [*signal_id*]

A **`catch`** command with an operand specifies that the debugger should catch and handle the given signal. You can specify the signal by integer number or by standard signal name, with or without the leading "SIG". The **`catch`** command is equivalent to the breakpoint command

```
(idb) catch ILL
```

or

```
(idb) stop signal SIGILL
[#1: stop signal SIGILL]
```

with these exceptions:

- No entry is made in the breakpoint table for a **`catch`** command.
- A catch for a signal that is already being caught does not create an additional breakpoint for that signal.

An **`ignore`** command with an operand specifies that the given signal should not be caught or handled by the debugger; rather, such a signal is passed to your program. The **`ignore`** command is equivalent to deleting the breakpoint created by a **`catch`** command for that signal:

```
(idb) ignore SIGILL
```

A **`catch`** command without an operand lists all signals that are currently being handled. Similarly, an **`ignore`** command without an operand lists the signals that are currently being ignored. Together, the two lists show all signals known to the debugger.

You can issue these commands immediately after the debugger starts to show which signals are caught and which are ignored by default:

```
(idb) catch
INT, ILL, ABRT, FPE, SEGV, TERM, QUIT, TRAP, BUS, SYS, PIPE, URG, STOP, TTIN,
TTOU, XCPU, XFSZ, PROF, USR1, USR2, VTALRM
(idb) ignore
RTMIN, RTMIN1, RTMIN2, RTMIN3, RTMIN4, RTMIN5, RTMIN6, RTMIN7, RTMAX, RTMAX7,
RTMAX6, RTMAX5, RTMAX4, RTMAX3, RTMAX2, RTMAX1, HUP, KILL, ALRM, TSTP, CONT,
CHLD, WINCH, POLL
```

The `info handle` and `handle` Commands

GDB mode

IDB has the ability to detect any occurrence of a signal in your program. You can tell IDB in advance what to do for each kind of signal. Normally, IDB is set up to let the some signals like SIGUSR1 be silently passed to user program but to stop your program immediately whenever an error signal happens. You can change these settings with the **`handle`** command.

```
handle_command

        : handle signal_name handle_keyword



handle_keyword

        : stop | nostop | print | noprint | pass | nopass | ignore | noignore



info_handle_command

        : info handle

        | info signals
```

The keywords allowed by the **`handle`** command can be abbreviated. Their full names are:

Keywords	Description
nostop	IDB should not stop your program when this signal happens. It may still print a message telling you that the signal has come in.
stop	IDB should stop your program when this signal happens. This implies the print keyword as well.
print	IDB should print a message when this signal happens.
noprint	IDB should not mention the occurrence of the signal at all. This implies the nostop keyword as well.
pass noignore	IDB should allow your program to see this signal; your program can handle the signal, or else it may terminate if the signal is fatal and not handled. pass and noignore are synonyms.
nopass ignore	IDB should not allow your program to see this signal. nopass and ignore are synonyms.

Example of using command **`info handle`** (GDB mode):

```
(idb) info handle
Signal        Stop      Print   Pass to program Description
SIGHUP        No        No      Yes             Hangup
SIGINT        Yes       Yes     No              Interrupt
SIGQUIT       Yes       Yes     No              Quit
SIGILL        Yes       Yes     No              Illegal instruction
SIGTRAP       Yes       Yes     No              Trace/breakpoint trap
SIGABRT       Yes       Yes     No              Aborted
SIGEMT        Yes       Yes     No              Emulation trap
SIGFPE        Yes       Yes     No              Arithmetic exception
SIGKILL       No        No      Yes             Killed
SIGBUS        Yes       Yes     No              Bus error
SIGSEGV       Yes       Yes     No              Segmentation fault
SIGSYS        Yes       Yes     No              Bad system call
SIGPIPE       Yes       Yes     No              Broken pipe
SIGALRM       No        No      Yes             Alarm clock
SIGTERM       Yes       Yes     No              Terminated
SIGURG        Yes       Yes     No              Urgent I/O condition
SIGSTOP       Yes       Yes     No              Stopped (signal)
SIGTSTP       No        No      Yes             Stopped (user)
SIGCONT       No        No      Yes             Continued
SIGCHLD       No        No      Yes             Child status changed
SIGTTIN       Yes       Yes     No              Stopped (tty input)
SIGTTOU       Yes       Yes     No              Stopped (tty output)
SIGIO         Yes       Yes     No              I/O possible
SIGXCPU       Yes       Yes     No              CPU time limit exceeded
SIGXFSZ       Yes       Yes     No              File size limit exceeded
SIGVTALRM     Yes       Yes     No              Virtual timer expired
SIGPROF       Yes       Yes     No              Profiling timer expired
SIGWINCH      No        No      Yes             Window size changed
```

```
SIGLOST       Yes       Yes     No              Resource lost
SIGUSR1       Yes       Yes     No              User defined signal 1
SIGUSR2       Yes       Yes     No              User defined signal 2
SIGPWR        Yes       Yes     No              Power fail/restart
SIGPOLL       No        No      Yes             Pollable event occurred
SIGWIND       Yes       Yes     No              SIGWIND
SIGPHONE      Yes       Yes     No              SIGPHONE
SIGWAITING    Yes       Yes     No              Process's LWPs are blocked
SIGLWP        Yes       Yes     No              Signal LWP
SIGDANGER     Yes       Yes     No              Swap space dangerously low
SIGGRANT      Yes       Yes     No              Monitor mode granted
SIGRETRACT    Yes       Yes     No              Need to relinquish monitor
mode
SIGMSG        Yes       Yes     No              Monitor mode data available
SIGSOUND      Yes       Yes     No              Sound completed
SIGSAK        Yes       Yes     No              Secure attention
SIGPRIO       Yes       Yes     No              SIGPRIO
SIG33         No        No      Yes             Real-time event 33
SIG34         No        No      Yes             Real-time event 34
SIG35         No        No      Yes             Real-time event 35
SIG36         No        No      Yes             Real-time event 36
SIG37         No        No      Yes             Real-time event 37
SIG38         No        No      Yes             Real-time event 38
SIG39         No        No      Yes             Real-time event 39
SIG40         No        No      Yes             Real-time event 40
SIG41         No        No      Yes             Real-time event 41
SIG42         No        No      Yes             Real-time event 42
SIG43         No        No      Yes             Real-time event 43
SIG44         No        No      Yes             Real-time event 44
SIG45         No        No      Yes             Real-time event 45
SIG46         No        No      Yes             Real-time event 46
SIG47         No        No      Yes             Real-time event 47
SIG48         No        No      Yes             Real-time event 48
SIG49         No        No      Yes             Real-time event 49
SIG50         No        No      Yes             Real-time event 50
SIG51         No        No      Yes             Real-time event 51
SIG52         No        No      Yes             Real-time event 52
SIG53         No        No      Yes             Real-time event 53
SIG54         No        No      Yes             Real-time event 54
SIG55         No        No      Yes             Real-time event 55
SIG56         No        No      Yes             Real-time event 56
SIG57         No        No      Yes             Real-time event 57
SIG58         No        No      Yes             Real-time event 58
SIG59         No        No      Yes             Real-time event 59
SIG60         No        No      Yes             Real-time event 60
SIG61         No        No      Yes             Real-time event 61
SIG62         No        No      Yes             Real-time event 62
SIG63         No        No      Yes             Real-time event 63
SIGCANCEL     Yes       Yes     No              LWP internal signal
SIG32         Yes       Yes     No              Real-time event 32
SIG64         No        No      Yes             Real-time event 64
SIG65         Yes       Yes     No              Real-time event 65
SIG66         Yes       Yes     No              Real-time event 66
SIG67         Yes       Yes     No              Real-time event 67
SIG68         Yes       Yes     No              Real-time event 68
SIG69         Yes       Yes     No              Real-time event 69
SIG70         Yes       Yes     No              Real-time event 70
SIG71         Yes       Yes     No              Real-time event 71
SIG72         Yes       Yes     No              Real-time event 72
SIG73         Yes       Yes     No              Real-time event 73
SIG74         Yes       Yes     No              Real-time event 74
SIG75         Yes       Yes     No              Real-time event 75
SIG76         Yes       Yes     No              Real-time event 76
SIG77         Yes       Yes     No              Real-time event 77
```

```
SIG78         Yes       Yes     No              Real-time event 78
SIG79         Yes       Yes     No              Real-time event 79
SIG80         Yes       Yes     No              Real-time event 80
SIG81         Yes       Yes     No              Real-time event 81
SIG82         Yes       Yes     No              Real-time event 82
SIG83         Yes       Yes     No              Real-time event 83
SIG84         Yes       Yes     No              Real-time event 84
SIG85         Yes       Yes     No              Real-time event 85
SIG86         Yes       Yes     No              Real-time event 86
SIG87         Yes       Yes     No              Real-time event 87
SIG88         Yes       Yes     No              Real-time event 88
SIG89         Yes       Yes     No              Real-time event 89
SIG90         Yes       Yes     No              Real-time event 90
SIG91         Yes       Yes     No              Real-time event 91
SIG92         Yes       Yes     No              Real-time event 92
SIG93         Yes       Yes     No              Real-time event 93
SIG94         Yes       Yes     No              Real-time event 94
SIG95         Yes       Yes     No              Real-time event 95
SIG96         Yes       Yes     No              Real-time event 96
SIG97         Yes       Yes     No              Real-time event 97
SIG98         Yes       Yes     No              Real-time event 98
SIG99         Yes       Yes     No              Real-time event 99
SIG100        Yes       Yes     No              Real-time event 100
SIG101        Yes       Yes     No              Real-time event 101
SIG102        Yes       Yes     No              Real-time event 102
SIG103        Yes       Yes     No              Real-time event 103
SIG104        Yes       Yes     No              Real-time event 104
SIG105        Yes       Yes     No              Real-time event 105
SIG106        Yes       Yes     No              Real-time event 106
SIG107        Yes       Yes     No              Real-time event 107
SIG108        Yes       Yes     No              Real-time event 108
SIG109        Yes       Yes     No              Real-time event 109
SIG110        Yes       Yes     No              Real-time event 110
SIG111        Yes       Yes     No              Real-time event 111
SIG112        Yes       Yes     No              Real-time event 112
SIG113        Yes       Yes     No              Real-time event 113
SIG114        Yes       Yes     No              Real-time event 114
SIG115        Yes       Yes     No              Real-time event 115
SIG116        Yes       Yes     No              Real-time event 116
SIG117        Yes       Yes     No              Real-time event 117
SIG118        Yes       Yes     No              Real-time event 118
SIG119        Yes       Yes     No              Real-time event 119
SIG120        Yes       Yes     No              Real-time event 120
SIG121        Yes       Yes     No              Real-time event 121
SIG122        Yes       Yes     No              Real-time event 122
SIG123        Yes       Yes     No              Real-time event 123
SIG124        Yes       Yes     No              Real-time event 124
SIG125        Yes       Yes     No              Real-time event 125
SIG126        Yes       Yes     No              Real-time event 126
SIG127        Yes       Yes     No              Real-time event 127
SIGINFO       Yes       Yes     No              Information request
EXC BAD ACCESSYes       Yes     No              Could not access memory
EXC BAD INSTRUCTIONYes       Yes     No              Illegal
instruction/operand
EXC ARITHMETICYes       Yes     No              Arithmetic exception
EXC EMULATION Yes       Yes     No              Emulation instruction
EXC SOFTWARE  Yes       Yes     No              Software generated exception
EXC_BREAKPOINTYes       Yes     No              Breakpoint
```

Example of using command **`handle`** (GDB mode):

```
(idb) info handle ILL
Unrecognized or ambiguous flag word: "ILL".
```

```
(idb) handle SIGILL nostop noprint
Signal        Stop      Print    Pass to program Description
SIGILL        No        No       No              Illegal instruction
```

Unaligned Accesses

You can request the debugger to catch unaligned accesses:

```
(idb) catch unaligned
```

This command is very much like the `stop unaligned` command:

Although this looks like a normal `catch` command, it differs in several respects:

- `unaligned` is not the name of a signal
- There is no corresponding signal number
- `unaligned` is never listed by either the `catch` or `ignore` commands without an argument

Like other `catch` commands, the following rules apply:

- No entry is made in the breakpoint table for a `catch` command
- Repeating the command does not create an additional breakpoint

Note:

You cannot specify `unaligned` in a signal detector of a normal breakpoint definition.

You can request the debugger to ignore unaligned accesses when `catch unaligned` is in effect (the default) by using the following command:

```
(idb) ignore unaligned
```

However, if a breakpoint was defined using an unaligned access detector, then it must be disabled using a disable or delete breakpoint command.

Unaligned Accesses (Linux and Mac OS Only)

Unaligned accesses are automatically handled and quietly corrected on Linux and Mac OS. The debugger cannot catch these events.

Ctrl+C

If your program seems to be caught in a loop, you can press Ctrl+C. The debugger interprets this as a command to send a signal interrupt (`SIGINT`) to your program. Because the debugger itself catches signal `SIGINT` by default, this interrupts your program and returns control to the debugger prompt.

If you give the command **`ignore SIGINT`**, then it is no longer possible to regain control of your program using Ctrl+C. In that case, signal `SIGINT` is delivered directly to your program. Unless your program has explicitly arranged otherwise, `SIGINT` will result in program termination.

Breakpoint Interactions with `exec(), fork(), dlopen()` and `dlclose()` System Calls

A process starts with a copy of its parent's memory as the result of a `fork()` system call; after running for a while within that memory, the process will often make an `exec()` system call to start a new executable file within that process.

The debugger keeps track of the `exec()` calls that occur so that it can keep track of various properties associated with each executable file. In particular, the breakpoint table is one of those properties. Thus, if you **`run`** or **`rerun`** your program, the same breakpoints can be re-established, even though a new process is initiated. Similarly, if you work with more than one process, each process has a distinct breakpoint table associated with it.

When a `dlopen()` system call occurs, the debugger reprocesses the current breakpoint table and automatically sets up the means to detect any events that apply to the newly loaded image.

When a `dlclose()` system call occurs, the debugger also reprocesses the breakpoint and de-activates any events that apply to the unloaded image.

Obsolete Breakpoint Commands

The following forms of breakpoint commands are obsolete, but are still supported for backward compatibility with earlier versions of the debugger:

```
obsolete_breakpoint_definition_command

        : obsolete_watch_breakpoint_definition_command

        | obsolete_trace_breakpoint_definition_command

        | obsolete_stopi_breakpoint_definition_command

        | obsolete_wheni_breakpoint_definition_command

        | obsolete_tracei_breakpoint_definition_command
```

Obsolete Watchpoint Definition

An obsolete watchpoint definition is similar to a **`stop variable`** or **`stop memory`** breakpoint:

```
obsolete_watch_breakpoint_definition_command

        : watch obsolete_watch_detector

            [ obsolete_watch_modifiers ]

            [ breakpoint_actions ]


obsolete_watch_detector

        : variable variable_name

        | [ memory ] start_address_expression

        | [ memory ] start_address_expression , end_address_expression

        | [ memory ] start_address_expression : byte_count_expression


obsolete_watch_modifiers

        : [ access_modifier ]

          [ thread_filter ]

          [ within_modifier ]

          [ logical_filter ]
```

An obsolete watchpoint and a **`stop`** command differ in the following respects:

- The obsolete watchpoint command begins with **`watch`** instead of **`stop`**.
- The keyword **`memory`** is optional; if omitted, it is assumed.
- The order of filters and modifiers is different.

These differences are purely syntactic; the semantics are the same.

For example:

```
(idb) watch variable  firstNode write
[#3: watch variable  firstNode write]
(idb) cont
```

```
[3] Address 0xbfffc188 was accessed at:
void List<Node>::append(class Node* const): src/x list.cxx
 [line 149, 0x0804c56d] append(class Node* const)+0x15:                movl
    %edx, (%eax)
0xbfffc188: Old value = 0x00000000
0xbfffc188: New value = 0x0805e5f8
[3] stopped at [void List<Node>::append(class Node* const):149 0x0804c56f]
    149         _firstNode = node;
```

Obsolete Tracepoint Definition

An obsolete tracepoint definition is similar to a **`when in`** or **`when at`** breakpoint, possibly combined with watching for a change of a variable's value:

obsolete_trace_breakpoint_definition_command

: **trace** [*variable_name*]

[*thread_filter*]

[*where_modifier*]

[*logical_filter*]

[*breakpoint_actions*]

| **trace** *function_name [logical_filter] [breakpoint_actions]*

| **trace** *line_specifier* [*logical_filter] [breakpoint_actions]*

where_modifier

: **in** *function_name*

| **at** *line_specifier*

line_specifier

: *quoted_filename:line_number*

| *line_number*

quoted_filename

: "*filename*"

```
        | 'filename'
```

Following are the differences between an obsolete tracepoint and a **`when`** command:

- The obsolete tracepoint command begins with **`trace`** instead of **`when.`**
- If you specify a variable name, a trace identification line is displayed only when the value of the variable changes (and the logical filter evaluates to `true`).

  The debugger implementation of **`trace`** for detecting variable changes tends to be slow - at each place where control might be stopped, as specified by the where modifier and filters, the value of the variable is compared to the value remembered at the time execution began.
- The order of filters and modifiers is different.

For example:

```
(idb) trace in List<Node>::print
[#7: trace in void List<Node>::print(void)]
(idb) trace i in List<Node>::print
[#8: trace i in void List<Node>::print(void)]
(idb) trace List<Node>::print if i { print "Test 1" }
[#9: trace in void List<Node>::print(void) if i { print "Test 1" }]
```

If the **`trace`** command is given with no arguments, the debugger prints a trace identification line when each function in your program is entered. For example:

```
(idb) trace
[#10: trace]
(idb) status
#10 at procedure entry { trace-proc }
```

This is equivalent to the **`when every proc entry`** command (with equivalent performance degradation).

Instruction-Related Breakpoint Commands

The following commands control obsolete instruction-related breakpoints:

```
obsolete_stopi_breakpoint_definition_command

        : stopi [ expression ]

            [ thread_filter ] [ match_address ] [ logical_filter ]
```

```
obsolete_tracei_breakpoint_definition_command

        : tracei [ expression ]

            [ thread_filter ] [ match_address ] [ logical_filter ]


obsolete_wheni_breakpoint_definition_command

        : wheni [ expression ]

            [ thread_filter ] [ match_address ] [ logical_filter ]

            breakpoint_actions


match_address

        : at address_expression
```

The **`stopi`**, **`tracei`**, and **`wheni`** forms of breakpoint definition are similar to the corresponding **`stop`**, **`trace`**, and **`when`** forms, with the following differences:

- They have a different initial keyword.
- If you specify a variable name, then breakpoint triggers only when the value of the variable changes (and the logical and thread filters are true).

  The debugger implementation of **`tracei`** for detecting variable changes tends to be slow: at each place where control might be stopped, as specified by the `where` modifier and filters, the value of the variable is compared to the value remembered at the time execution began.

  Most important, **the variable change and filter tests are performed after every instruction is executed**, making these definitions especially demanding on program performance.
- The order of filters and modifiers is different.
- The **`at`** keyword is followed by an address in these commands, instead of a line number.

Breakpoint Tables

As breakpoints are defined, they are recorded in a breakpoint table associated with the current program. You can display and modify this table in certain limited ways.

`breakpoint_table_command`

: *show_all_breakpoints_command*

| *delete_breakpoint_command*

| *enable_breakpoint_command*

| *disable_breakpoint_command*

Each entry in the breakpoint table has the following properties:

- A unique breakpoint number that is used to identify and refer to that breakpoint.
- An event description that characterizes the circumstances under which the breakpoint triggers.
- Actions (a possibly empty list of debugger commands) to be performed when the breakpoint triggers.
- A final disposition: either continue or break (stop).
- Enabled and disabled states.

In addition to the main effects of a breakpoint definition, as discussed in Breakpoint Definitions, a breakpoint definition also sets the debugger variable `$lasteventmade` to the breakpoint number of the breakpoint just defined. This value can be recalled for later use if desired. For example:

```
(idb) stop in List<Node>::append
[#2: stop in void List<Node>::append(class Node* const)]
(idb) cont
[2] stopped at [void List<Node>::append(class Node* const):148 0x0804c55e]
    148     if (! firstNode)
(idb) print $lasteventmade
2
(idb) set $my break = $lasteventmade
(idb) print $my break
2
```

If an error occurs in a breakpoint command, the variable `$lasteventmade` is not changed.

Showing Breakpoint Status

Use the following commands to display the current breakpoint table:

DBX Mode

show_all_breakpoints_command

: **status**

GDB Mode

show_all_breakpoints_command

```
: info breakpoints [ expression ]
| info watchpoints [ expression ]
| info break       [ expression ]
| info b           [ expression ]
| i    breakpoints [ expression ]
| i    watchpoints [ expression ]
| i    break       [ expression ]
| i    b           [ expression ]
```

All these commands are synonyms.

Specify breakpoint number to print information about particular breakpoint. If you do not specify an argument, the debugger prints information about all breakpoints.

Each entry in the current breakpoint table is displayed showing all of its properties. For example:

DBX Mode

```
(idb) status
#1 PC==0x08052e0f in int main(void) "src/x list.cxx":182 { stop }
#2 PC==0x0804c55e in void List<Node>::append(class Node* const)
"src/x list.cxx":148 { break }
#3 Access memory (write) 0xbfffc188 to 0xbfffc18b { stop }
```

GDB Mode

```
(idb) info breakpoints
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x0804b1c3 in main at src/x list.cxx:182
breakpoint already hit 1 time(s)
2   breakpoint     keep y   0x0804f20a in List<Node>::append(Node * const) at
src/x list.cxx:148
breakpoint already hit 1 time(s)
3   watchpoint     keep y              _firstNode
```

When an entry in the current breakpoint table references a shared object that is not currently mapped, its contribution to the **What** column indicates **Not Currently Mapped**.

When large or complex values are passed by value to the routine in the status line, the output can be voluminous. You can set the control variable `$statusargs` to 0 to suppress the output of argument type information in the status line.

Enabling, Disabling, and Deleting Breakpoints

When a breakpoint is defined, it is enabled by default. When the debugger starts or resumes process execution, it first adapts the process so that it can detect when the given events occur. A breakpoint can be disabled so it is not involved in determining when the process should next stop. A breakpoint that is no longer required can be deleted entirely.

DBX Mode

```
disable_breakpoint_command

        : disable all

        | disable breakpoint_number_expression ,...


enable_breakpoint_command

        : enable all

        | enable breakpoint_number_expression ,...


delete_breakpoint_command

        : delete all

        | delete breakpoint_number_expression ,...
```

GDB Mode

```
disable_breakpoint_command

        : disable [ breakpoints ] [ bpnums ]

        | dis     [ breakpoints ] [ bpnums ]


enable_breakpoint_command

        : enable [ breakpoints ] [ bpnums ]


delete_breakpoint_command

        : delete [ breakpoints ] [ bpnums ]
        | d      [ breakpoints ] [ bpnums ]
```

bpnums

```
        : bpnum ...
```

bpnum

```
        : integer
```

You can specify one or more breakpoint numbers to disable, enable or delete. If you do not specify any arguments, the debugger disables, enables or deletes **all** the breakpoints.

For example:

DBX Mode

```
(idb) disable 1
(idb) status
#1 PC==0x08052e0f in int main(void) "src/x list.cxx":182 { stop } Disabled
#2 PC==0x0804c55e in void List<Node>::append(class Node* const)
"src/x list.cxx":148 { break }
#3 Access memory (write) 0xbfffc188 to 0xbfffc18b { stop }
(idb) disable 10 - 8,1 + 1 + 1
(idb) status
#1 PC==0x08052e0f in int main(void) "src/x list.cxx":182 { stop } Disabled
#2 PC==0x0804c55e in void List<Node>::append(class Node* const)
"src/x list.cxx":148 { break } Disabled
#3 Access memory (write) 0xbfffc188 to 0xbfffc18b { stop } Disabled
(idb) delete 1
(idb) status
#2 PC==0x0804c55e in void List<Node>::append(class Node* const)
"src/x list.cxx":148 { break } Disabled
#3 Access memory (write) 0xbfffc188 to 0xbfffc18b { stop } Disabled
(idb) enable all
(idb) status
#2 PC==0x0804c55e in void List<Node>::append(class Node* const)
"src/x list.cxx":148 { break }
#3 Access memory (write) 0xbfffc188 to 0xbfffc18b { stop }
```

GDB Mode

```
(idb) disable 1
(idb) info breakpoints
Num Type           Disp Enb Address    What
1   breakpoint     keep n   0x0804b1c3 in main at src/x list.cxx:182
breakpoint already hit 1 time(s)
2   breakpoint     keep y   0x0804f20a in List<Node>::append(Node * const) at
src/x list.cxx:148
breakpoint already hit 1 time(s)
3   watchpoint     keep y               firstNode
(idb) disable 2,3
(idb) info breakpoints
Num Type           Disp Enb Address    What
1   breakpoint     keep n   0x0804b1c3 in main at src/x_list.cxx:182
```

```
breakpoint already hit 1 time(s)
2   breakpoint     keep n   0x0804f20a in List<Node>::append(Node * const) at
src/x list.cxx:148
breakpoint already hit 1 time(s)
3   watchpoint     keep n               firstNode
(idb) **delete 1**
(idb) **info breakpoints**
Num Type           Disp Enb Address    What
2   breakpoint     keep n   0x0804f20a in List<Node>::append(Node * const) at
src/x list.cxx:148
breakpoint already hit 1 time(s)
3   watchpoint     keep n               firstNode
(idb) **enable**
(idb) **info breakpoints**
Num Type           Disp Enb Address    What
2   breakpoint     keep y   0x0804f20a in List<Node>::append(Node * const) at
src/x list.cxx:148
breakpoint already hit 1 time(s)
3   watchpoint     keep y              _firstNode
```

Looking Around at the Code, the Data, and Other Process Information

This section describes how to look at the following components of a running process:

- The source files
- The threads
- The call stack of one or more threads
- The data
- The generated code
- The shared libraries that are loaded

See also

Examining the Paused Process

Looking at the Source Files. Expert Debugging

The debugger supports commands to perform the following operations with source files:

- Determine the location of the source files
- Select a particular file as the current file
- List portions of the current file
- Search through the current file for target strings

```
browse_source_command

        : source_directory_mapping_command

        | source_searchlist_command

        | select_source_file_command

        | list_source_file_command
```

| *search_source_file_command*

Special debugging information that the compiler puts in the .o files correlates the machine instructions and data back to the source files and the positions they came from.

Source files are compiled and linked into executable files. During debugging, the debugger tries to find these source files to display them for you. If the source files have moved, or if the paths to them are relative, the debugger may not be able to locate them. All the information the debugger needs comes from the executable files or shared libraries, not from the source files.

How the Debugger Finds Source Files

The debugger searches for a source file (`dir_name/base_name`) using the following algorithm:

1. If `dir_name` is mapped to another source directory (`mapped_dir_name`), look for `mapped_dir_name/base_name`.
2. If Step 1 fails to find a readable file:
   Case 1: If `dir_name` is absolute, look for `dir_name/base_name`.
   Case 2: If `dir_name` is relative, for each entry `use_dir` in `use_list`, look for `use_dir/dir_name/base_name`. Note that the `use_list` entries are tried in the order they appear in the `use_list`.
3. If Step 2 fails, for each entry `use_dir` in `use_list`, look for `use_dir/base_name`. Just as in Step 2, the `use_list` entries are tried in the order they appear in the `use_list`.
4. If Step 3 fails, the debugger cannot find any source file.

The debugger uses the first-found readable file as the source file.

The debugger has source directory mapping commands that:

- Inform you in which directories the debugger is looking for the source files.
- Allow you to designate directories in which the debugger will look for the source files.

The following example shows how to use source directory mapping. Suppose you compile `x_solarSystem` as follows:

```
% pwd
/home/user/examples
% ls -R
bin/ src/
./bin:
x solarSystem*
./src:
solarSystemSrc/
./src/solarSystemSrc:
base class includes/    main/                   star.cxx
derived class includes/ orbit.cxx
heavenlyBody.cxx        planet.cxx
./src/solarSystemSrc/base class includes:
heavenlyBody.h  orbit.h
./src/solarSystemSrc/derived class includes:
planet.h  star.h
./src/solarSystemSrc/main:
```

```
solarSystem.cxx
% cd src
% cc -g -o ../bin/x solarSystem \
  -IsolarSystemSrc/base class includes \
  -IsolarSystemSrc/derived class includes \
  main/solarSystem.cxx heavenlyBody.cxx orbit.cxx planet.cxx star.cxx
```

Then you move the directory `solarSystemSrc` elsewhere:

```
% mv solarSystemSrc movedSolarSystemSrc
```

Now debug `x_solarSystem` in `/home/users/Examples/bin`

DBX Mode

```
(idb) list $curline - 10:20
Source file not found or not readable, tried...
    ./bld/i686-Linux-
development/debuggable/solarSystemSrc/main/solarSystem.cxx
    ../src/bld/i686-Linux-
development/debuggable/solarSystemSrc/main/solarSystem.cxx
    /home/user/examples/bld/i686-Linux-
development/debuggable/solarSystemSrc/main/solarSystem.cxx
    ./solarSystem.cxx
    ../src/solarSystem.cxx
    /home/user/examples/solarSystem.cxx
```

The debugger cannot find the file because it has been moved to another directory.

The following command displays a summary of the source directories in `a.outexe`. The ellipsis (...) here means that `solarSystemSrc` contains one or more source directories.

DBX Mode

```
(idb) show source directory
.
/home/user/examples/solarSystemSrc
 ...
Information: You can further expand a '...' using the command
    show source directory <directory>
or
    show all source directory <directory>
where <directory> is the directory on the line above the '...'.
The first command displays only the children of <directory>, whereas
the second command displays all the descendants of <directory>.
```

The following command directs the debugger to look for source files originally in `solarSystemSrc` in `movedSolarSystemSrc` instead. This time, the debugger finds the source file.

DBX Mode

```
(idb) map source directory /home/user/examples/solarSystemSrc
/home/user/examples/movedSolarSystemSrc
(idb) list $curline - 10:20
    104
    105     // Insert the new entry appropriately
    106     //
    107     if (iAmBiggerThan < biggestCount) {
    108         biggestMoons[iAmBiggerThan] = moon;
    109     }
    110 }
    111
    112 int main()
    113 {
>   114     unsigned int j = 1;    // for scoping examples
    115     for (unsigned int i = 0; i < biggestCount; i++)
    116         biggestMoons[i] = NULL;
    117
    118     Star *sun = new Star("Sol", G, 2);
    119     buildOurSolarSystem(sun);
    120     sun->printBodyAndItsSatellites(j);
    121     printBiggestMoons();
    122
    123     return 0;
```

The following command gives a complete list of source directories. As you can see, `solarSystemSrc` is mapped to `movedSolarSystemSrc`. As a side effect of mapping `solarSystemSrc` to `movedSolarSystemSrc`, the subdirectories in `solarSystemSrc` are mapped to their counterparts under `movedSolarSystemSrc`.

DBX Mode

```
(idb) show all source directory
.
/home/user/examples/solarSystemSrc  *=>
 /home/user/examples/movedSolarSystemSrc
 /home/user/examples/solarSystemSrc/base class includes  =>
 /home/user/examples/movedSolarSystemSrc/base class includes
 /home/user/examples/solarSystemSrc/derived class includes  =>
 /home/user/examples/movedSolarSystemSrc/derived class includes
 /home/user/examples/solarSystemSrc/main  =>
 /home/user/examples/movedSolarSystemSrc/main
```

To summarize, the debugger provides the following four commands for checking and setting source directory mappings:

DBX Mode

source_directory_mapping_command

        : **show source directory** [*directory_name*]

        | **show all source directory** [*directory_name*]

| **`map source directory`** *`from_directory_name`* *to_directory_name*

| **`unmap source directory`** *`from_directory_name`*

Use the **`show source directory`** (dbx) command to display the directory mapping information of `directory_name` and its child directories (or immediate subdirectory). If `directory_name` is not specified, the mapping information of all the source directories whose parent is not a source directory is displayed.

The **`show all source directory`** (dbx) command is identical to the **`show source directory`** (dbx) command except that the mapping information of all the descendants of `directory_name` is displayed:

DBX Mode

```
(idb) show source directory
.
/home/user/examples/solarSystemSrc  *=>
 /home/user/examples/movedSolarSystemSrc
 ...
(idb) show all source directory
.
/home/user/examples/solarSystemSrc  *=>
 /home/user/examples/movedSolarSystemSrc
 /home/user/examples/solarSystemSrc/base class includes  =>
 /home/user/examples/movedSolarSystemSrc/base class includes
 /home/user/examples/solarSystemSrc/derived class includes  =>
 /home/user/examples/movedSolarSystemSrc/derived class includes
 /home/user/examples/solarSystemSrc/main  =>
 /home/user/examples/movedSolarSystemSrc/main
```

When you further expand ellipsis points (...) where `directory` is the directory on the line above the ellipsis points:

- The **`show source directory`** (dbx) command displays only the children of `directory_name`.
- The **`show all source directory`** (dbx) command displays all the descendants of `directory_name`.

Use the **`map source directory`** (dbx) command to tell the debugger that the source files in the directory `from_directory_name` can now be found in `to_directory_name`.

The **`unmap source directory`** (dbx) command maps `from_directory_name` back to itself; in other words, if `from_directory_name` has been mapped to some other directory, this command will restore its default mapping. For example:

DBX Mode

```
(idb) show source directory
.
/home/user/examples/solarSystemSrc  *=>
 /home/user/examples/movedSolarSystemSrc
 ...
(idb) show source directory /home/user/examples/solarSystemSrc
/home/user/examples/solarSystemSrc  *=>
 /home/user/examples/movedSolarSystemSrc
 /home/user/examples/solarSystemSrc/base_class_includes  =>
 /home/user/examples/movedSolarSystemSrc/base class includes
 /home/user/examples/solarSystemSrc/derived class includes  =>
 /home/user/examples/movedSolarSystemSrc/derived_class_includes
 /home/user/examples/solarSystemSrc/main  =>
 /home/user/examples/movedSolarSystemSrc/main
```

```
(idb) unmap source directory /home/user/examples/solarSystemSrc
(idb) show source directory /home/user/examples/solarSystemSrc
/home/user/examples/solarSystemSrc
 /home/user/examples/solarSystemSrc/base class includes
 /home/user/examples/solarSystemSrc/derived_class_includes
```

Note:

The symbol `*=>` means that you are setting the mapping explicitly using the **`map source directory`** (dbx) command, whereas `=>` means that the mapping is derived from an existing explicit mapping.

By default, the `use_list` is: (1) the current directory and (2) the directory containing the executable file (dbx). Each process has its own `use_list`. You can also use the **`idb`** command **`-I`** option to specify search directories.

The following commands let you view and modify the `use_list`.

DBX Mode

source_search_list_command

    : *use_command*

    | *unuse_command*

GDB Mode

source_search_list_command

    : *directory_command*

    | **show directories**

Enter the **`use`** (dbx) without an argument or **`show directories`** (gdb) command to list the directories in which the debugger searches for source code files. Specify a directory argument to make source code files in that directory available to the debugger. You can also use the **`idb`** command **`-I`** option to specify search directories, which puts those directories in the `use_list`.

You can customize your debugger environment source code search paths by adding commands to your `.dbxinit` file that use the **`use`** command:

DBX Mode

`use_command`

```
        : use [directory_name ...]
```

If the `directory_name` is specified, it is either appended to or replaces the `use_list`, depending on whether the value of the `$dbxuse` debugger variable is zero (append) or non-zero (replace).

```
(idb) use
Directory search path for source files:
. ../src /home/user/examples
```

```
(idb)
(idb) use aa
Directory search path for source files:
. ../src /home/user/examples aa
(idb)
(idb) use bb cc
Directory search path for source files:
. ../src /home/user/examples aa bb cc
(idb)
(idb)
(idb) use bb
Directory search path for source files:
. ../src /home/user/examples aa bb cc
```

```
(idb) use aa bb cc
Directory search path for source files:
. ../src /home/user/examples aa bb cc
(idb) set $dbxuse = 1
(idb) use aa
Directory search path for source files:
. ../src /home/user/examples aa
(idb) use aa bb cc
```

GDB Mode

`directory_command`

```
        : directory [directory_name ...]
```

Use the **`directory`** command with no argument to reset the `use_list` to empty value.

If the `directory_name` is specified, it is prepended to `use_list`. Several directory names may be given to the **`directory`** command, separated by '**:**' or whitespace. If you specify a directory, which is already in the list, it is moved forward.

To get the full list of directories in the search list, use the **`show directories`** command.

DBX Mode

`unuse_command`

```
        : unuse [directory_name ...]

        | unuse *
```

The **`unuse`** command removes entries from the `use_list`:

Enter the **`unuse`** command without the `directory_name` to set the search list to the default (the home directory, the current directory, and the directory containing the executable file). Include the directory names to remove them from the search list. The asterisk (*****) argument removes all directories from the search list.

```
(idb)
(idb) unuse aa
Directory search path for source files:
. ../src /home/user/examples bb cc
(idb)
(idb) unuse aa
aa not in the current source path
Directory search path for source files:
. ../src /home/user/examples bb cc
(idb)
(idb) unuse bb cc
Directory search path for source files:
. ../src /home/user/examples
(idb)
(idb) unuse *
Directory search path for source files:
(idb)
(idb) unuse
Directory search path for source files:
. ../src /home/user/examples
```

How the Debugger Chooses Which Source File to List

The debugger has a concept of current source file, so you do not have to explicitly specify a source file in many commands. Whenever the process stops, the current source file is set to the source file for the code currently executing. The commands **`up`**, **`down`**, **`class`** (dbx), and **`file`** (dbx) also set the current source file.

DBX Mode

You can see and modify the current source file selection:

`select_source_file_command`

```
        : file [ filename ]

        : fileexpr [ expression ]
```

Use the **`file`** command without a file name to display the name of the current file scope. Include the file name to change the file scope. Change the file scope to set a breakpoint in a function not in the file currently being executed.

To see source code for or set a breakpoint in a function not in the file currently being executed, use the **`file`** command to set the file scope.

If the file name is not a literal, use the **`fileexpr`** command. For example, if you have a script that calculates a file name in a debugger variable or in a routine that returns a file name as a string, you can use **`fileexpr`** to set the file.

The following example uses the **`file`** command to set the debugger file scope to a file different from the main program, and then stops at line number 26 in that file. This example also shows the **`fileexpr`** command setting the current scope back to the original file, which is `solarSystem.cxx`.

```
(idb) run
[1] stopped at [int main(void):114 0x08058b1e]
    114     unsigned int j = 1;    // for scoping examples
(idb) file
/home/user/examples/solarSystemSrc/main/solarSystem.cxx
(idb) set $originalFile = "solarSystem.cxx"
(idb) list 36:10
     36     Moon   *enceladus = new Moon("Enceladus",  238,  260, saturn);
     37     Moon   *tethys    = new Moon("Tethys",     295,  530, saturn);
     38     Moon   *dione     = new Moon("Dione",      377,  560, saturn);
     39     Moon   *rhea      = new Moon("Rhea",       527,  765, saturn);
     40     Moon   *titan     = new Moon("Titan",     1222, 2575, saturn);
     41     Moon   *hyperion  = new Moon("Hyperion",  1481,  143, saturn);
     42     Moon   *iapetus   = new Moon("Iapetus",   3561,  730, saturn);
     43
     44     Planet *uranus    = new Planet("Uranus",     2870990, sun);
     45     Moon   *miranda   = new Moon("Miranda",    130,  236, uranus);
(idb) file star.cxx
(idb) list 36:10
     36 void Star::printBody(unsigned int i) const
     37 {
     38     std::cout << "(" << i << ") Star [" << name()
     39           << "], class [" << stellarClassName(classification())
     40           << ((int)subclassification()) << "]" << std::endl;
     41 }
     42
     43 StellarClass Star::classification() const
     44 {
     45     return  classification;
(idb) stop at 38
[#2: stop at "/home/user/examples/solarSystemSrc/star.cxx":38]
(idb) cont
[2] stopped at [virtual void Star::printBody(unsigned int):38 0x0805605e]
     38     std::cout << "(" << i << ") Star [" << name()
(idb) file
/home/user/examples/solarSystemSrc/star.cxx
(idb) fileexpr $originalFile
(idb) file
/home/user/examples/solarSystemSrc/main/solarSystem.cxx
```

```
(idb) list 36:10
     36     Moon   *enceladus = new Moon("Enceladus",  238,  260, saturn);
     37     Moon   *tethys    = new Moon("Tethys",     295,  530, saturn);
     38     Moon   *dione     = new Moon("Dione",      377,  560, saturn);
     39     Moon   *rhea      = new Moon("Rhea",       527,  765, saturn);
     40     Moon   *titan     = new Moon("Titan",     1222, 2575, saturn);
     41     Moon   *hyperion  = new Moon("Hyperion",  1481,  143, saturn);
     42     Moon   *iapetus   = new Moon("Iapetus",   3561,  730, saturn);
     43
     44     Planet *uranus    = new Planet("Uranus",     2870990, sun);
     45     Moon   *miranda   = new Moon("Miranda",    130,  236, uranus);
```

GDB Mode

In the GDB mode, the current file can be changed, for example, using the **`list`** command.

See the example:

```
(idb) run
Starting program: /home/user/examples/x solarSystem
Breakpoint 1, main () at
/home/user/examples/solarSystemSrc/main/solarSystem.cxx:114
114     unsigned int j = 1;    // for scoping examples
(idb) info source
Current source file is /home/user/examples/solarSystemSrc/main/solarSystem.cxx
(idb)
(idb) list 36,+10
36     Moon   *enceladus = new Moon("Enceladus",  238,  260, saturn);
37     Moon   *tethys    = new Moon("Tethys",     295,  530, saturn);
38     Moon   *dione     = new Moon("Dione",      377,  560, saturn);
39     Moon   *rhea      = new Moon("Rhea",       527,  765, saturn);
40     Moon   *titan     = new Moon("Titan",     1222, 2575, saturn);
41     Moon   *hyperion  = new Moon("Hyperion",  1481,  143, saturn);
42     Moon   *iapetus   = new Moon("Iapetus",   3561,  730, saturn);
43
44     Planet *uranus    = new Planet("Uranus",     2870990, sun);
45     Moon   *miranda   = new Moon("Miranda",    130,  236, uranus);
(idb) list star.cxx:36,+10
36 void Star::printBody(unsigned int i) const
37 {
38     std::cout << "(" << i << ") Star [" << name()
39          << "], class [" << stellarClassName(classification())
40          << ((int)subclassification()) << "]" << std::endl;
41 }
42
43 StellarClass Star::classification() const
44 {
45     return  classification;
(idb) break 38
Breakpoint 2 at 0x805605e: file /home/user/examples/solarSystemSrc/star.cxx,
line 38.
(idb) continue
Continuing.
Breakpoint 2, Star::printBody (this=0x806e168, i=1) at
/home/user/examples/solarSystemSrc/star.cxx:38
38     std::cout << "(" << i << ") Star [" << name()
(idb) info source
Current source file is star.cxx
(idb) list solarSystem.cxx:36,+10
36     Moon   *enceladus = new Moon("Enceladus",  238,  260, saturn);
```

```
37     Moon   *tethys    = new Moon("Tethys",     295,  530, saturn);
38     Moon   *dione     = new Moon("Dione",      377,  560, saturn);
39     Moon   *rhea      = new Moon("Rhea",       527,  765, saturn);
40     Moon   *titan     = new Moon("Titan",     1222, 2575, saturn);
41     Moon   *hyperion  = new Moon("Hyperion",  1481,  143, saturn);
42     Moon   *iapetus   = new Moon("Iapetus",   3561,  730, saturn);
43
44     Planet *uranus    = new Planet("Uranus",     2870990, sun);
45     Moon   *miranda   = new Moon("Miranda",    130,  236, uranus);
(idb) info source
Current source file is solarSystem.cxx
```

Listing Source Files

DBX Mode

The simplest way to see a source file is to use a text editor. The **`edit`** command will display an editor on the current file, using the current definition of the `EDITOR` environment variable, if there is one.

However, some primitive inspection capabilities are built into the debugger. The **`list`** command displays source lines, which can be defined by following way:

- The position of the program counter
- The last line listed, if multiple list commands are entered
- The line number specified as the arguments to the **`list`** command

DBX Mode

list_source_file_command

: **list** [*line_expression*]

| **list** *line_expression* , *line_expression*

| **list** *line_expression* : *line_expression*

line_expression

: *expression*

If specified, the first expression must evaluate to either an integer (the line number of the first line to display within the current source file) or a function (the first line of the function).

Specify the exact range of source lines as either a comma followed by the expression for the last line, or a colon followed by the expression for the number of lines. This second expression must evaluate to an integer value.

If a second expression is not given, the debugger shows 20 lines, fewer if the end of source file is reached.

GDB Mode

list_source_file_command

        : **list** [[+ | -] *line_expression*] [, [[+ | -]

line_expression]]

        | **list** *function*

If a **`function`** name or the only one **`line_expression`** is specified as the single argument then lines centered around the function beginning or specified line are printed.
Commands **`list +`** and **`list -`** print some lines after and before the last printed correspondently.

The line in arguments can be specified by following way:

- **`integer number`** - the line in the current source file with specified number
- **`+offset`** and **`-offset`** - the line moved on 'offset' lines upper or downer from the last printed
- **`*address`** - the line which contains code for specified program address

User can specify a source file name for arguments given in form **`integer number`** and **`function`** by following way:
`filename:integer_number` and **`filename:function`**.

For example, to list lines 16 through 20:

DBX Mode

```
(idb) list 16, 20
     16
     17 class Node {
     18 public:
     19     Node ();
     20
```

GDB Mode

```
(idb) list 16,20
16
17 class Node {
18 public:
19     Node ();
20
```

For example, to list 6 lines, beginning with line 16:

DBX Mode

```
(idb) list 16:6
     16
     17 class Node {
     18 public:
     19     Node ();
     20
     21     virtual void printNodeData() const = 0;
```

GDB Mode

```
(idb) list 16,+6
16
17 class Node {
18 public:
19     Node ();
20
21     virtual void printNodeData() const = 0;
```

Showing Column Information

If you compile your application with the option **`-debug extended`** or **`-debug emit_column`** the debugger will be able to show you the column in the current line that corresponds to the current PC.

```
(idb) stop in main
[#1: stop in int main(void)]
(idb) run
[1] stopped at [int main(void):382:14 0x08049438]
    382     static S a;
(idb) next
stopped at [int main(void):384:9 0x0804949f]
    384     int i = 123;
(idb) next
stopped at [int main(void):385:5 0x080494a6]
    385     a.i = 456;
(idb) list $curline-5:10
    380 {
    381
    382     static S a;
    383     S* p;
    384     int i = 123;
>   385     a.i = 456;
    386     p = &a;
    387     p->c = S::plus1;
    388
    389     g ();               // calls a number of overloaded functions
```

Note that the character at the current column of the current line is highlighted (underlined in the example) with your terminal's default highlight mode. Also note that in a breakpoint message, the current line and column numbers are displayed in the format `<current line>:<current column>`. The same format is used to display line/column information in the stack trace and the machine code listing when the column information is available.

This feature is available in the DBX mode, and in the GDB mode when `$gdb_compatible_output` is set to 0.

Searching the Content of Source Files

The following search commands search through the current source file to help you find the lines to list:

DBX Mode

```
search_source_file_command

        : / [ string ]

        | ? [ string ]
```

GDB Mode

```
search_source_file_command

        : forward-search [ string ]

        | reverse-search [ string ]
```

DBX Mode

Note:

> The string is actually just the rest of the line, not a string literal. The rest of the line is still having alias expansion done on it.

Use a slash (`/`) to search forward from the most recently listed line; use a question mark (`?`) to search backward. Like most searches, it will stop at the end (or beginning) of the file being searched, and will wrap if the command is repeated at that point.

When the string is omitted, the previous search continues from where it found the string. When the string is present, the search starts from either the start (`/`) or the end (`?`) of the current line.

When a match is found, the debugger lists the line number and the line. That line becomes the starting point for any further searches, or for a **`list`** command. For example:

1. To locate `_firstNode`:

   ### DBX Mode

   ```
   (idb) / firstNode
        69     NODETYPE* _firstNode;
   ```

   ### GDB Mode

```
(idb) forward-search  firstNode
69     NODETYPE* _firstNode;
```

2. Then to locate `append` before line 69:

   DBX Mode

   ```
   (idb) ?append
        65     void      append  (NODETYPE* const node);
   ```

   GDB Mode

   ```
   (idb) reverse-search append
   65     void      append  (NODETYPE* const node);
   ```

3. Then to locate `append` after line 65:

   DBX Mode

   ```
   (idb) /append
       145 void List<NODETYPE>::append(NODETYPE* const node)
   ```

   GDB Mode

   ```
   (idb) forward-search append
   145 void List<NODETYPE>::append(NODETYPE* const node)
   ```

The debugger provides parameterized aliases and debugger variables of arbitrary types. You can use these to do list traversal (see the array navigation example).

Looking at the Threads. Expert Debugging

A thread is a single, sequential flow of control within a process. Each thread contains a single point of execution. Threads execute within (and share) a single address space; therefore, a process's threads can read and write the same memory locations.

Thread Levels

On Linux* and Mac OS*, the debugger supports POSIX threads, also known as pthreads. The $threadlevel debugger variable is default to "pthreads" and cannot be modified.

For example:

```
(idb) set $threadlevel = "pthreads"
```

Thread Manipulation Commands

You can use a variety of commands to manipulate the threads:

DBX Mode

`thread_command`

```
        : show_thread_command
        | switch_thread_command
```

GDB Mode

```
thread_commands
        : info_thread_command
        | thread_command
```

Thread Display Commands

You can use the following commands to display threads:

DBX Mode

```
show_thread_command
        : show thread [ thread_id_list ] [ thread-state-filter]


thread_id_list
        : thread_id ,...
        | *


thread_id
        : expression


thread_state_filter
        : with state eq thread_state
eq
        : ==              (for C and C++)
        | .eq.            (for Fortran)


thread_state
```

: **ready**

| **running**

| **terminated**

| **blocked**

Use the **`show thread`** command without parameters to list all the threads known to the debugger.

If you specify one or more thread identifiers, the debugger displays information about the threads you specify, if the thread matches what you specified in the list. If you omit a thread specification, the debugger displays information for all threads.

Use the **`show thread`** commands to list threads that have specific characteristics, such as threads that are currently blocked. For example:

```
(idb) print $threadlevel
"pthreads"
(idb) show thread
  Thread Name                      State           Substate    Policy
      Pri
  ------ ------------------------- --------------- ----------- ------------ --
-
*      1 default thread            running VP 3                SCHED OTHER  19
      -1 manager thread            blk SCS                     SCHED RR     19
      -2 null thread for slot 0    running VP 1                null thread  -1
      -3 null thread for slot 1    ready VP 3                  null thread  -1
      -4 null thread for slot 2    new             new         null thread  -1
      -5 null thread for slot 3    new             new         null thread  -1
>      2 threads(0x140000798)      blocked         cond 3      SCHED OTHER  19
       3 threads+8(0x1400007a0)    blocked         cond 3      SCHED OTHER  19
       4 threads+16(0x1400007a8)   blocked         cond 3      SCHED OTHER  19
       5 threads+24(0x1400007b0)   blocked         cond 3      SCHED OTHER  19
       6 threads+32(0x1400007b8)   blocked         cond 3      SCHED OTHER  19
(idb) set $threadlevel = "pthreads"
(idb) print $threadlevel
"pthreads"
(idb) show thread
   Id                  State
*  0x9                 stopped
*  0x9                 unstarted
   0x3                 unstarted
   0x7                 unstarted
```

GDB Mode

info_threads_command

: **info threads**

thread

: *expression*

Use the **info threads** command to list all the threads known to the debugger.

```
(idb) info threads
  0 Thread 1024 (LWP 19513)  0x804f8f6 in   sigsuspend from
/tmp/pthread manythreads
  1 Thread 2049 (LWP 19514)  0x805a42a in   clone from
/tmp/pthread manythreads
* 2 Thread 1026 (LWP 19515)  0x804f8f6 in   sigsuspend from
/tmp/pthread manythreads
  3 Thread 2051 (LWP 19516)  0x804f8f6 in   sigsuspend from
/tmp/pthread manythreads
  4 Thread 3076 (LWP 19517)  0x804f8f6 in   sigsuspend from
/tmp/pthread manythreads
  5 Thread 4101 (LWP 19518)  0x8048288 in prime search at
pthread_manythreads.c:79
```

Note:

> In the output, the right bracket indicator (`>`) marks the current thread, whereas the asterisk (`*`) indicator marks the thread with the event that stopped the application.

You can switch to a different thread as the current thread. The debugger variable `$curthread` contains the thread identifier of the current thread.

```
switch_thread_command

          : thread [ thread_id ]
```

The `$curthread` value is updated when program execution stops or completes. You can modify the current thread by assigning `$curthread` a valid thread identifier. This is equivalent to issuing the **thread** *thread_id* command. When there is no process or program, `$curthread` is set to 0.

Use the **thread** command without a thread identifier to identify the current thread. Supply a thread identifier to make another thread the current thread.

GDB Mode

```
(idb) thread 2
* 2 Thread 1026 (LWP 19515)  0x804f8f6 in   sigsuspend from
/tmp/pthread_manythreads
```

Other Thread Commands

You can use the **where** command to display the stack trace of current threads. You can specify one or more threads or all threads.

The **print** command evaluates an optional expression in the context of the current thread and displays the result.

The **`call`** command evaluates an expression in the context of the current thread and makes the call in the context of the current thread.

The **`printregs`** command prints the registers for the current thread.

Looking at the Call Stack. Expert Debugging

Most programming languages have some concept of functions, routines, or subroutines, capturing the notion of code that is invoked from many places. A running program needs a call stack of call frames for the called functions. Each call frame contains both the information needed to return to its caller and the information needed to contain the local variables of the function.

The machine code generated for these functions maintains this call stack. Some of this maintenance is done before the call, some at the start of the called function, some at the end of the called function, and some after the call.

Non-optimized machine code is usually very easy to correlate with the source code, but optimized machine code can be tricky. See Call Frames and Optimized Code and Call Frames and Machine Code Correlation for more information.

The debugger controls the call stacks of all the threads; you can use it to examine and manipulate call stacks, and use them as a basis for further queries:

```
call_stack_command

        : show_stack_command

        | change_stack_frame_command

        | pop_stack_frame_command
```

When your process is stopped by the debugger, you can show the call stack of the thread that caused the stoppage, or the call stack of any other thread.

The following commands show the most recent call frames on the call stack of the current or specified threads:

DBX Mode

```
show_stack_command

        : where [ expression ] [ thread_specifier ]



thread_specifier

        : thread thread_id ,...
        | thread all
```

thread_id

```
        : expression
```

GDB Mode

show_stack_command

```
        : Backtrace [ expression ]

        | where [ expression ]

        | info stack [ expression ]

        | bt [ expression ]
```

If specified, the expression must evaluate to a non-negative integer. You can specify the number of call frames to show. If not specified, all the call frames for the thread are shown.

You can change the control variable `$stack_levels` to control number of call stack output levels. Default value is 50.

DBX Mode

If specified, the `thread_specifier` specifies the threads whose call stacks are to be shown. If not specified, just the current thread is used.

When large and complex values are passed by value to a routine on the stack, the output of the **`where`** command can be voluminous. You can set the control variable `$stackargs` to 0 to suppress the output of argument values in the **`where`** command.

The stack trace provides the following information for each call level:

Call level	The number used to refer to a call level on the stack. The function entered most recently is at level 0. Its caller is at level 1.
Memory address	The address of the next instruction to be executed at this level.
Function name	The name of the function for the memory address.
File name	The source file for the memory address.
Line number	The number of the next source line of the memory address.

If your call stack seems to be missing routines, you may be seeing the result of a compiler optimization known as *tail calls*. When a function calls another function, typically it saves the return address, registers etc. on the stack before making a call, so that the callee can know where to return to once it is done. A new stack frame is also created for the callee. If the caller calls the callee as the last operation and just returns the result it got from callee, as is the case with some recursive functions, the compiler may convert the call to a branch or jump. This is called tail call optimization, and allows the callee to directly return results to the caller. In this case, the stack does not change and the caller function's stack frame is reused, which saves time and stack space. However, since no new frames are added, stack frames can be missing from the stack trace, which can ultimately adversely affect debugging.

If your call stack is corrupted, you may see random numbers without any routine names. In this case, it is likely that your application has gotten lost. Typically, this type of call stack display means that your application has lost track of the real stack and real code location, and is now executing random bits of memory, interpreting them as instructions.

If you are coding in C++, one of the most common ways to get a corrupt stack is for your code to try to execute a method on an invalid object. If the object has already been deleted, has not yet been initialized, is not there, or is of a completely different type, then the virtual function table will not be correct, and the application will be treating random memory as the virtual function table and calling a random place.

GDB Mode

Note:

You should change the current thread to the thread whose call stack you want to examine.

For unwinding live processes, the Linux version of IDB for Itanium®-based systems uses the Hewlett-Packard libunwind* unwinder described at the following web site: http://www.hpl.hp.com/research/linux/libunwind/.

This unwinder supports programs that generate code at run-time and register that code along with rules describing how to unwind through it.

To unwind core files, a proprietary algorithm is used. It can also be used for live processes. In the unlikely event that the libunwind algorithm fails, you can force the following range of behaviors by setting either the debugger variable *`idb_libunwind_usage`* or the environment variable *`IDB_LIBUNWIND_USAGE`* to either 0, 1, 2, or 3:

- 0: only use the proprietary algorithm; do not use the libunwind algorithm at all
- 1: use the libunwind algorithm first, then use the proprietary algorithm
- 2: use the proprietary algorithm first, then use the libunwind algorithm
- 3: only use the libunwind algorithm, do not use the proprietary algorithm at all

Navigating the Call Stack

You can select one of the call frames as the starting point for examining variables. This call frame provides the current scope in the program for which variables exist, and tells the debugger which instance of those variables whose values you want to see.

DBX Mode

```
change_stack_frame_command

        : up   [ expression]

        | down [ expression]

        | func [ loc ]
```

GDB Mode

```
change_stack_frame_command

        : up            [ expression]

        | up-silently   [ expression]

        | down          [ expression]

        | down-silently [ expression]

        | frame         [ expression]
```

Use the **up** command or the **down** command without the expression to change to the call frame located one level up or down the stack. Specify an expression that evaluates to an integer to change the call frame up or down the specified number of levels. If the number of levels exceeds the number of active calls on the stack in the specified direction, the debugger issues a warning message and the call frame does not change.

When the current call frame changes, the debugger displays the source line corresponding to the last instruction executed in the function executing the selected call frame.

DBX Mode

When large and complex values are passed by value to a routine on the stack, the output of the **up** and **down** commands can be voluminous. You can set the control variable `$stackargs` to 0 to suppress the output of argument values in the **up** and **down** commands.

Use the **func** command without the `loc` to display the current function. To change the function scope to a function that has a call frame in the call stack, specify the `loc` either as the name of the function or as an integer expression evaluating to the call level. If you specify the name, the most-recently entered call frame for that function becomes the current call frame.

If no frames are available to select from, the debugger context is set to the static context of the named function. The current scope and current language are set based on that function. Types and static variables local to that function are now visible and can be evaluated.

If you enter an integer expression, the debugger moves to the frame at level `n`, just as if you had entered **up** `n` at the level 0 function.

GDB Mode

`up-silently` and **`down-silently`** commands are similar to **`up`** and **`down`** respectively. But they work without invoking a new frame.

The **`frame`** command selects frame by given number or address. If there is no argument the command displays info about current stack frame.

In the following example, the current call frame is changed to one for method `Planet::print` so that a variable in that instance of `print()` can be displayed:

DBX Mode

```
(idb) where 4
#0  0x080553fd in ((Planet*)0x806e298)->Planet::printBody(i=2)
"/home/user/examples/solarSystemSrc/planet.cxx":19
#1  0x0804d697 in ((HeavenlyBody*)0x806e298)-
>HeavenlyBody::printBodyAndItsSatellites(i=2)
"/home/user/examples/solarSystemSrc/heavenlyBody.cxx":62
>2  0x0804d6d2 in ((HeavenlyBody*)0x806e168)-
>HeavenlyBody::printBodyAndItsSatellites(i=1)
"/home/user/examples/solarSystemSrc/heavenlyBody.cxx":68
#3  0x08058bf2 in main()
"/home/user/examples/solarSystemSrc/main/solarSystem.cxx":120
(idb) list $curline - 5:10
     63
     64     // Recursively deal with the satellites.  Redeclare i for scoping
examples.
     65     //
     66     unsigned int j = 1;
     67     for (HeavenlyBody* i =  firstSatellite; i; i = i-> outerNeighbor)
{
>    68     i->printBodyAndItsSatellites(j++);     {static int
somethingToReturnTo; somethingToReturnTo++; }
     69     }
     70 }
(idb) whatis i
class HeavenlyBody* i
(idb) print i
0x806e298
(idb) func Planet::printBody
virtual void Planet::printBody(unsigned int) in
/home/user/examples/solarSystemSrc/planet.cxx line No. 19:
     19     std::cout << "(" << i
(idb) where 4
>0  0x080553fd in ((Planet*)0x806e298)->Planet::printBody(i=2)
"/home/user/examples/solarSystemSrc/planet.cxx":19
#1  0x0804d697 in ((HeavenlyBody*)0x806e298)-
>HeavenlyBody::printBodyAndItsSatellites(i=2)
"/home/user/examples/solarSystemSrc/heavenlyBody.cxx":62
#2  0x0804d6d2 in ((HeavenlyBody*)0x806e168)-
>HeavenlyBody::printBodyAndItsSatellites(i=1)
"/home/user/examples/solarSystemSrc/heavenlyBody.cxx":68
#3  0x08058bf2 in main()
"/home/user/examples/solarSystemSrc/main/solarSystem.cxx":120
(idb) list $curline - 5:10
     14 {
     15 }
     16
     17 void Planet::printBody(unsigned int i) const
     18 {
>    19     std::cout << "(" << i
     20          << ") Planet [" << HeavenlyBody::name() << "]; ";
     21     printOrbitalParameters();
     22     std::cout << std::endl;
     23 }
(idb) whatis i
unsigned int i
(idb) print i
2
```

In the previous example, instead of entering **`func Planet::print`**, you can enter **`down 2`**. (You would use **`down`** in this case because the current call frame at the start of the example was not the bottommost frame.) Note that the final stack trace in this example lists a call frame for function `Planet::print` as the current call frame (denoted by the > character).

GDB Mode

There are some commands to print info about the selected stack frame:

`info frame`

The command prints following info about the current frame:

- the address of the frame
- the address of the next frame down (called by this frame)
- the address of the next frame up (caller of this frame)
- the language in which the source code corresponding to this frame is written
- the address of the frame's arguments
- the address of the frame's local variables
- the program counter saved in it (the address of execution in the caller frame)
- which registers were saved in the frame

`info frame addr`

The command prints a verbose description of the frame at address addr, without selecting that frame.

`info args`

The command prints the arguments of the selected frame, each on a separate line.

`info locals`

The command prints the local variables of the selected frame, each on a separate line. These are all variables (declared either static or automatic) accessible at the point of execution of the selected frame.

`info catch`

The command prints a list of all the exception handlers that are active in the current stack frame at the current point of execution.

The `pop` Command

The **`pop`** command removes one or more call frames from the call stack:

DBX Mode

```
pop_stack_frame_command

        : pop [expression]
```

The default is one call frame. The **`pop`** command undoes the work already done by the removed execution frames. It does not, however, reverse side effects, such as changes to global variables.

Note:

> Because it is extremely unlikely this will fix all the effects of a half-executed call, this command is not recommended for general use. Furthermore, the **`pop`** command does not provide a way to specify a return value when the frame being discarded corresponds to a function that should return a value. You may need to use the **`assign`** command to restore the values of global variables.

Instead of the **`pop`** command, you may want to use the **`return`** command, which finishes the call corresponding to the selected frame.

GDB Mode

```
pop_stack_frame_command

        : return [expression]
```

When you use **`return`**, the selected stack frame is discarded (and all frames within it). If you wish to specify a value to be returned, give that value as the argument to **`return`**.

The **`return`** command does not resume execution. It leaves the program stopped in the state that would exist if the function had just returned. In contrast, the **`finish`** command resumes execution until the selected stack frame returns naturally.

Call Frames and Optimized Code

When optimized machine code is generated by the compilers, the compiler generates code that maintains the call stack, but sometimes the function boundaries are changed in one of two ways:

- **Inlining** is when the compiler completely eliminates the call by instead generating the instructions for the called function at the call site, usually followed by merging those instructions with the other instructions surrounding the call site.
- **Outlining** is when the compiler creates a function where one did not exist explicitly in the source. For example, the compiler turns a loop body into a function, so that it can generate code that uses threads to execute the different iterations in parallel; or the compiler creates a single shared function to replace several sections of the source that are similar.

Depending on the information the compiler makes available to the debugger, inlined calls may or may not show up in the call stack display. Outlined calls will show up, and will be correlated to the code they came from. The compiler will probably have supplied the debugger with some invented name for the function.

Call Frames and Machine Code Correlation

For a call to a function, the machine code generated typically contains:

- The machine code before the call performs the following operations:
    - Sets some context registers
    - Puts the parameters either in registers or memory
    - Loads the address of the function into a register
    - Loads the address to return to into a register
    - Branches to the function
- The machine code at the start of the called function performs the following operations:
    - Sets some context registers
    - Allocates stack space
    - Saves some registers in the stack space
    - Performs some setup of the local variables
- The machine code at the end of the called function performs the following operations:
    - Restores the saved registers from the stack space
    - Deallocates the stack space
    - Branches to the address to return to
- The machine code at the return address of the call frame sets some context registers.

When the thread is partway through the call frame creation or tear-down, the debugger will still show the call frame, but will not be able to show correct values for the variables or parameters.

Special C++ Issues

For non-static member functions, the implicit *this* pointer is displayed as the address on the stack trace along with the class type of the object, as shown in the following example:

DBX Mode

```
(idb) stop in List<Node>::print
[#3: stop in void List<Node>::print(void)]
(idb) cont
[3] stopped at [void List<Node>::print(void):162 0x0804c5e6]
    162     Node* currentNode =  firstNode;
(idb) where 2
>0  0x0804c5e6 in ((List<Node>*)0xbfffa4f8)->List<Node>::print()
"src/x list.cxx":162
#1  0x080532f6 in main() "src/x_list.cxx":203
```

GDB Mode

```
(idb) break List<Node>::print
Breakpoint 3 at 0x804c5e6: file src/x list.cxx, line 162.
(idb) continue
Continuing.
Breakpoint 3, List<Node>::print (this=0xbfffdb98) at src/x list.cxx:162
162     Node* currentNode =  firstNode;
(idb) backtrace 2
#0  0x0804c5e6 in List<Node>::print (this=0xbfffdb98) at src/x list.cxx:162
#1  0x080532f6 in main () at src/x_list.cxx:203
```

Looking at the Data. Expert Debugging

After you have seen the call stack (`show_stack_command`), selected the call frame containing the variables you wish to examine (`change_stack_frame_command`), and looked at the source this function is executing (`looking at the source`), you usually want to examine some of the variables or even evaluate some expressions. You can use the **`print`** command and the **`call`** command to do this. You can also use the following commands to help you determine what to look at and what you are seeing:

DBX Mode

look_around_command

: *various_print_command*

| *c++_look_around_command*

| *call_command*

| *whatis_command*

| *whereis_command*

| *which_command*

various_print_command

: *print_command*

```
        | printf_command

        | printi_command

        | print_registers_command

        | printt_command

        | dump_command
```

GDB Mode

```
look_around_command

        : print_command


        | call_command

        | whatis_command
```

The `print` Command

You can print the values of one or more expressions or all local variables. You can also use the **`print`** command to evaluate complex expressions involving typecasts, pointer dereferences, multiple variables, constants, and any legal operators allowed by the language of the program you are debugging:

DBX Mode

```
print_command

        : print [ expression ,... ]

        | print rescoped_expression

        | print printable-type

        | printb [ expression ,... ]

        | printd [ expression ,... ]

        | printo [ expression ,... ]

        | printx [ expression ,... ]


rescoped_expression
```

```
        : filename ` qual_symbol
        | ` qual_symbol
```

```
qual_symbol
        : expression
        | qual_symbol ` expression
```

For an array, the debugger prints every cell in the array if you do not specify a specific cell.

Use the `$hexints`, `$decints`, or `$octints` variables to select a radix for the output of the **`print`** command. If you do not want to change the radix permanently, use the **`printx`**, **`printd`**, **`printo`**, and **`printb`** commands to print expressions in hexadecimal, decimal, octal, or binary base format, respectively.

GDB Mode

```
print_command
        : print [/format specifier] [expression]
```

```
format specifier
        : x | d | u | o | t | a | c | f
```

By default, GDB prints a value according to its data type. But user might want to print a number in hex, or a pointer in decimal. Or user might want to view data in memory at a certain address as a character string or as an instruction. In this case user should specify an output format.
To specify how to print a value already computed a user should print after the print command a slash and a format letter. The format letters supported are:

- **`x`**
  Regard the bits of the value as an integer, and print the integer in hexadecimal
- **`d`**
  Print as integer in signed decimal.
- **`u`**
  Print as integer in unsigned decimal.
- **`o`**
  Print as integer in octal.
- **`t`**
  Print as integer in binary. The letter `t' stands for "two".

- **a**
  Print as an address, both absolute in hexadecimal and as an offset from the nearest preceding symbol.
- **c**
  Regard as an integer and print it as a character constant.
- **f**
  Regard the bits of the value as a floating point number and print using typical floating point syntax.

For an array, the debugger prints every cell in the array if you do not specify a specific cell.

Use the **set output-radix xxx** command to select a radix for the output of the **print** command, where xxx can be values: 8, 10 or 16.

Consider the following declarations in a C++ program:

DBX Mode

```
(idb) list 59:2
     59 const unsigned int biggestCount = 10;
     60 static Moon *biggestMoons[biggestCount];
```

GDB Mode

```
(idb) list 59,+2
59 const unsigned int biggestCount = 10;
60 static Moon *biggestMoons[biggestCount];
```

The following example uses the **print** command to display a non-string array:

DBX Mode

```
(idb) print biggestMoons
[0] = 0x806e5b0,[1] = 0x806e8c8,[2] = 0x806e608,[3] = 0x806e500,[4] =
0x806e348,[5] = 0x806e558,[6] = 0x806ec38,[7] = 0x806eb30,[8] = 0x806e870,[9]
= 0x806eb88
```

GDB Mode

```
(idb) print biggestMoons
$4 = {0x806e5b0, 0x806e8c8, 0x806e608, 0x806e500, 0x806e348, 0x806e558,
0x806ec38, 0x806eb30, 0x806e870, 0x806eb88}
```

The following example shows how to print individual values of an array:

DBX Mode

```
(idb) print biggestMoons[3]
0x806e500
```

```
(idb) print *biggestMoons[3]
class Moon {
   radius = 1815;
   name = 0x805c5c0="Io";          // class Planet::HeavenlyBody
  _innerNeighbor = 0x0;            // class Planet::HeavenlyBody
   outerNeighbor = 0x806e558;      // class Planet::HeavenlyBody
   firstSatellite = 0x0;           // class Planet::HeavenlyBody
  _lastSatellite = 0x0;            // class Planet::HeavenlyBody
   primary = 0x806e4a8;            // class Planet::Orbit
   distance = 422;                 // class Planet::Orbit
  _name = 0x806e530="Jupiter 1";   // class Planet::Orbit
}
```

GDB Mode

```
(idb) print biggestMoons[3]
$7 = (Moon *) 0x806e500
(idb) print *biggestMoons[3]
$8 = {<Planet> = {<HeavenlyBody> = { name = 0x805c5c0 "Io",  innerNeighbor =
0x0,  outerNeighbor = 0x806e558,  firstSatellite = 0x0,  lastSatellite = 0x0},
<Orbit> = { primary = 0x806e4a8,  distance = 422,  name = 0x806e530 "Jupiter
1"}}, _radius = 1815}
```

Dereferencing Pointers

Pointers are variables that contain addresses. By dereferencing a pointer in the command interface, you can print the value at the address pointed to by the pointer. In C and C++ programs, variables containing a pointer are dereferenced using the * operator. The following example shows how to dereference a pointer in C++ programs:

```
(idb) whatis newNode
class IntNode* newNode
(idb) print newNode
0x805e5f8
(idb) print *newNode
class IntNode {
   data = 1;
   nextNode = 0x0;                 // class Node
}
```

Printing C Strings

The debugger does not print more than the first `$maxstrlen` characters of a null-terminated string. Change this debugger variable if it is showing either more or less than you wish to see.

Printing Floating Point Numbers

Floating point numbers are represented inside the computer in binary floating point. They are converted to decimal floating point when printed. The two formats are not the same, and some numbers are easily represented in decimal but not in binary (for example the number 1.1). The internal binary form for such numbers is an approximation, the closest that can be made given the number of bits available.

Normally, when a binary floating point number is printed, the shortest decimal number which would be represented by that binary number is used as the number to print, as it is a legitimate representation of the internal binary number. However, to see a more exact (extended form) representation of a binary floating point number, you can set the `$floatshrinking` debugger variable to 0 (zero).

The following example shows the result of converting 1.1 (shortened form) to the closest long double binary floating point number (extended form).

```
(idb) p $floatshrinking
1
(idb) p 1.1
1.1
(idb) set $floatshrinking = 0
(idb) p 1.1
1.1000000000000000000000000000000000008
```

Currently, the extended forms are only available for long double variables and expressions.

For more detail on floating point representation, see ANSI IEEE standard 754-1985.

Printing 80-bit Floating Point

Floating point numbers occur in different lengths. Usual forms are 32, 64, and 128 bits.

On some platforms, an 80-bit length floating point may also occur. It is stored in a 128-bit register or memory area. Currently idb cannot distinguish between a true 128-bit number and an 80-bit number held in a 128-bit container.

If an 80-bit value is printed as though it were a 128-bit floating point number, the value is incorrect (typically, it will show up as a very small positive number). To correctly print the values of 80-bit floating point numbers, set the debugger variable `$float80bit` to a non-zero value. This will cause idb to treat all 128-bit floating point numbers as 80-bit numbers in 128-bit containers.

Restrictions on the `print` Command

Expressions containing labels are not supported. Variables involving static anonymous unions and enumerated types may not be able to be printed. Printing a structure that is declared but not defined in a compilation unit may generate an error message indicating that the structure is opaque.

Extended Naming Syntax

IDB implements several extensions to the normal language expressions and names to let you reference variables, types, and so on, that are not visible by the normal language rules.

For example:

for all languages:

“file_name”`identifier

`scope_name`scope-name`identifier

for Fortran:

module_name %% identifier

For example, to set a breakpoint in the subroutine bar contained in a globally defined module foo, do:

```
(idb) stop in foo%%bar
```

The `printf` Command

Use the **`printf`** command to format and display a complex structure. The first argument is a string expression of characters and conversion specifications using the same format specifiers as the `printf` C function. The **`printf`** command requires a running target program because it uses `libc`.

`printf_command`

```
        : printf [ format_string [ , expression ,... ] ]
```

For example:

```
 (idb) printf "The PC is 0x%x", $pc
The PC is 0x80532f6
```

The `printi` Command

The **`printi`** command takes one or more numerical expressions and interprets each one as an assembly instruction, printing out the instruction, and its arguments when applicable. This command is typically used by engineers performing machine-level debugging.

`printi_command`

```
               : printi [ expression ,... ]
```

For example:

```
(idb) $curpc/1i
int main(void): src/x list.cxx
*[line 182, 0x08052e0f] main+0x1b:                       pushl    %edi
```

```
(idb) $curpc/1dd
0x08052e0f:   1619365207
(idb) printi $pc
```

The `printregs` Command

Use the **`printregs`** command to display the values of all the hardware registers. The list of registers displayed by the debugger is machine-dependent. By default, most values are displayed in decimal radix. To display the register values in hexadecimal radix, set the `$hexints` variable to 1.

print_registers_command

```
        : printregs
```

For example:

```
(idb) printregs
$eax           0x805df10 134602512
$ecx           0xb74bcd98 -1219768936
$edx           0xb74ba610 -1219779056
$ebx           0xb74bcd98 -1219768936
$esp [$sp]     0xbfffa4b0 -1073765200
$ebp [$fp]     0xbfffa598 -1073764968
$esi           0xbfffa624 -1073764828
$edi           0xb74ba67c -1219778948
$eip [$pc]     0x80532f6 134558454
$eflags        0x296 662
$cs            0x23 35
$ss            0x2b 43
$ds            0x2b 43
$es            0x2b 43
$fs            0x0 0
$gs            0x33 51
$orig eax      0xffffffff -1
$fctrl         0x37f 895
$fstat         0x0 0
$ftag          0x0 0
$fiseg         0x23 35
$fioff         0x8050129 134545705
$foseg         0x2b 43
$fooff         0xbfffa2cc -1073765684
$fop           0x48b 1163
$f0            0x0 0
$f1            0x0 0
$f2            0x0 0
$f3            0x0 0
$f4            0x0 0
$f5            0x0 0
$f6            0x0 0
$f7            0xa1f7cf0000000000 10.1230001449584961
$xmm0          0x0 union {
  v4 float = [0] = 0,[1] = 0,[2] = 0,[3] = 0;
  v2 double = [0] = 0,[1] = 0;
  v16 int8 = [0] = 0,[1] = 0,[2] = 0,[3] = 0,[4] = 0,[5] = 0,[6] = 0,[7] =
0,[8] = 0,[9] = 0,[10] = 0,[11] = 0,[12] = 0,[13] = 0,[14] = 0,[15] = 0;
  v8 int16 = [0] = 0,[1] = 0,[2] = 0,[3] = 0,[4] = 0,[5] = 0,[6] = 0,[7] = 0;
  v4 int32 = [0] = 0,[1] = 0,[2] = 0,[3] = 0;
  v2_int64 = [0] = 0,[1] = 0;
}
$xmm1          0x0 union {
  v4 float = [0] = 0,[1] = 0,[2] = 0,[3] = 0;
  v2 double = [0] = 0,[1] = 0;
  v16 int8 = [0] = 0,[1] = 0,[2] = 0,[3] = 0,[4] = 0,[5] = 0,[6] = 0,[7] =
0,[8] = 0,[9] = 0,[10] = 0,[11] = 0,[12] = 0,[13] = 0,[14] = 0,[15] = 0;
  v8 int16 = [0] = 0,[1] = 0,[2] = 0,[3] = 0,[4] = 0,[5] = 0,[6] = 0,[7] = 0;
  v4 int32 = [0] = 0,[1] = 0,[2] = 0,[3] = 0;
  v2 int64 = [0] = 0,[1] = 0;
}
$xmm2          0x0 union {
  v4 float = [0] = 0,[1] = 0,[2] = 0,[3] = 0;
  v2 double = [0] = 0,[1] = 0;
  v16 int8 = [0] = 0,[1] = 0,[2] = 0,[3] = 0,[4] = 0,[5] = 0,[6] = 0,[7] =
0,[8] = 0,[9] = 0,[10] = 0,[11] = 0,[12] = 0,[13] = 0,[14] = 0,[15] = 0;
  v8 int16 = [0] = 0,[1] = 0,[2] = 0,[3] = 0,[4] = 0,[5] = 0,[6] = 0,[7] = 0;
  v4 int32 = [0] = 0,[1] = 0,[2] = 0,[3] = 0;
  v2 int64 = [0] = 0,[1] = 0;
}
$xmm3          0x0 union {
  v4 float = [0] = 0,[1] = 0,[2] = 0,[3] = 0;
  v2 double = [0] = 0,[1] = 0;
  v16 int8 = [0] = 0,[1] = 0,[2] = 0,[3] = 0,[4] = 0,[5] = 0,[6] = 0,[7] =
0,[8] = 0,[9] = 0,[10] = 0,[11] = 0,[12] = 0,[13] = 0,[14] = 0,[15] = 0;
  v8 int16 = [0] = 0,[1] = 0,[2] = 0,[3] = 0,[4] = 0,[5] = 0,[6] = 0,[7] = 0;
  v4 int32 = [0] = 0,[1] = 0,[2] = 0,[3] = 0;
  v2 int64 = [0] = 0,[1] = 0;
}
$xmm4          0x0 union {
  v4 float = [0] = 0,[1] = 0,[2] = 0,[3] = 0;
  v2 double = [0] = 0,[1] = 0;
  v16 int8 = [0] = 0,[1] = 0,[2] = 0,[3] = 0,[4] = 0,[5] = 0,[6] = 0,[7] =
0,[8] = 0,[9] = 0,[10] = 0,[11] = 0,[12] = 0,[13] = 0,[14] = 0,[15] = 0;
  v8 int16 = [0] = 0,[1] = 0,[2] = 0,[3] = 0,[4] = 0,[5] = 0,[6] = 0,[7] = 0;
  v4 int32 = [0] = 0,[1] = 0,[2] = 0,[3] = 0;
  v2 int64 = [0] = 0,[1] = 0;
}
$xmm5          0x0 union {
  v4 float = [0] = 0,[1] = 0,[2] = 0,[3] = 0;
  v2 double = [0] = 0,[1] = 0;
  v16 int8 = [0] = 0,[1] = 0,[2] = 0,[3] = 0,[4] = 0,[5] = 0,[6] = 0,[7] =
0,[8] = 0,[9] = 0,[10] = 0,[11] = 0,[12] = 0,[13] = 0,[14] = 0,[15] = 0;
  v8 int16 = [0] = 0,[1] = 0,[2] = 0,[3] = 0,[4] = 0,[5] = 0,[6] = 0,[7] = 0;
  v4 int32 = [0] = 0,[1] = 0,[2] = 0,[3] = 0;
  v2 int64 = [0] = 0,[1] = 0;
}
$xmm6          0x0 union {
  v4 float = [0] = 0,[1] = 0,[2] = 0,[3] = 0;
  v2 double = [0] = 0,[1] = 0;
  v16 int8 = [0] = 0,[1] = 0,[2] = 0,[3] = 0,[4] = 0,[5] = 0,[6] = 0,[7] =
0,[8] = 0,[9] = 0,[10] = 0,[11] = 0,[12] = 0,[13] = 0,[14] = 0,[15] = 0;
  v8 int16 = [0] = 0,[1] = 0,[2] = 0,[3] = 0,[4] = 0,[5] = 0,[6] = 0,[7] = 0;
  v4 int32 = [0] = 0,[1] = 0,[2] = 0,[3] = 0;
  v2 int64 = [0] = 0,[1] = 0;
}
$xmm7          0x0 union {
  v4 float = [0] = 0,[1] = 0,[2] = 0,[3] = 0;
  v2 double = [0] = 0,[1] = 0;
  v16 int8 = [0] = 0,[1] = 0,[2] = 0,[3] = 0,[4] = 0,[5] = 0,[6] = 0,[7] =
0,[8] = 0,[9] = 0,[10] = 0,[11] = 0,[12] = 0,[13] = 0,[14] = 0,[15] = 0;
  v8 int16 = [0] = 0,[1] = 0,[2] = 0,[3] = 0,[4] = 0,[5] = 0,[6] = 0,[7] = 0;
  v4 int32 = [0] = 0,[1] = 0,[2] = 0,[3] = 0;
  v2 int64 = [0] = 0,[1] = 0;
}
```

```
$mxcsr          0x1f80 8064
$vfp            0xbfffa5a0 -1073764960
```

The **`printt`** Command

The **`printt`** command takes one or more numerical expressions and interprets each one as the number of seconds since the Epoch (00:00:00 UTC 1 Jan 1970; see `ctime`(3) for more information).

For example:

printt_command

        : **printt** [*expression* ,...]

```
(idb) printt 0
(UTC) Thu Jan  1 00:00:00 1970
(idb) printt 978325200
(UTC) Mon Jan  1 05:00:00 2001
```

The **`dump`** Command

Use the **`dump`** command without an argument to list the parameters and local variables in the current function. To list the parameters and local variables in an active function, specify it as an argument.

Use the **`dump .`** command (include the dot) to list the parameters and local variables for all functions active on the stack:

dump_command

        : **dump** *qual_symbol*

        | **dump .**

For example:

```
(idb) dump
>0  0x080532f6 in main() "src/x list.cxx":203
cNode=0x805e608
cNode1=0x805e620
cNode2=0x805e658
newNode=0x805e5f8
newNode2=0x805e648
nodeList=class List<Node> { ... }
```

When large and complex values are passed by value to a routine on the stack, the output of the **`dump`** command can be voluminous. You can set the control variable `$stackargs` to 0 to suppress the output of argument values in the **`dump`** command.

The `call` Command

After a breakpoint or a signal suspends program execution, you can execute a single function in your program by using the **`call`** command, or by including a function call in the expression argument of a debugger command. Calling a function lets you test the function's operation with a specific set of parameters.

```
call_command

        : call call-expression
```

Specify the function as if you were calling it from within the program. If the function has no parameters, specify empty parentheses (`()`). For multithreaded applications, the call is made in the context of the current thread. For C++ applications, when you set the `$overloadmenu` debugger variable to 1 and call an overloaded function, the debugger lists the overloaded functions and calls the function you specify. When the function you call completes normally, the debugger restores the stack and the current context that existed before the function was called.

While the program counter is saved and restored, calling a function does not shield the program state from alteration if the function you call allocates memory or alters global variables. If the function affects global program variables, for instance, those variables will be changed permanently.

Functions compiled without the debugger option to include debugging information may lack important parameter information and are less likely to yield consistent results when called.

The **`call`** command executes the specified function with the parameters you supply and then returns control to you (at the debugger prompt) when the function returns. The **`call`** command discards the return value of the function. If you embed the function call in the expression argument of a **`print`** command, the debugger prints the return value after the function returns. The following example shows both methods of calling a function:

```
(idb) call earth->distance()
(idb) print earth->distance()
149600
```

In the previous example, the **`call`** command results in the return value being discarded while the embedded call passes the return value of the function to the **`print`** command, which in turn prints the value. You can also embed the call within a more involved expression, as shown in the following example:

```
(idb) print earth->distance() - 100000
49600
(idb) print mars->distance() - earth->distance()
```

```
78340
(idb) call io->printBody(3)
(3) Moon [Io], radius [1815] km; <Jupiter 1> orbits at 422 Megameters
```

All breakpoints or tracepoints defined and enabled during the session are active when a called function is executing. When program execution halts during function execution, you can examine program information, execute one line or instruction, continue execution of the function, or call another function.

When you call a function when execution is suspended in a called function, you are nesting function calls, as shown in the following example:

```
(idb) where 2
>0  0x080581c6 in buildOurSolarSystem(sun=0x806e168)
"/home/user/examples/solarSystemSrc/main/solarSystem.cxx":55
#1  0x08058bda in main()
"/home/user/examples/solarSystemSrc/main/solarSystem.cxx":119
(idb) stop in Planet::printBody
[#2: stop in virtual void Planet::printBody(unsigned int)]
(idb) call mars->printBody(1)
[2] stopped at [virtual void Planet::printBody(unsigned int):19 0x080553fd]
     19     std::cout << "(" << i
(idb) where
>0  0x080553fd in ((Planet*)0x806e3a0)->Planet::printBody(i=1)
"/home/user/examples/solarSystemSrc/planet.cxx":19
#1  0xb7387cec in __do_global_ctors_aux(...) in /lib/libdl-2.3.2.so
(idb) next
stopped at [virtual void Planet::printBody(unsigned int):20 0x08055403]
     20          << ") Planet [" << HeavenlyBody::name() << "]; ";
(idb) stop in Orbit::distance
[#3: stop in Megameters Orbit::distance(void)]
(idb) print distance()
[3] stopped at [Megameters Orbit::distance(void):41 0x08054ea7]
     41     return  distance;
(idb) where
>0  0x08054ea7 in ((Orbit*)0x806e3b8)->Orbit::distance()
"/home/user/examples/solarSystemSrc/orbit.cxx":41
#1  0xb7387cec in   do global ctors aux(...) in /lib/libdl-2.3.2.so
(idb) disable 3
(idb) cont
Called Procedure Returned
stopped at [virtual void Planet::printBody(unsigned int):20 0x08055403]
     20          << ") Planet [" << HeavenlyBody::name() << "]; ";
(idb) where
>0  0x08055403 in ((Planet*)0x806e3a0)->Planet::printBody(i=1)
"/home/user/examples/solarSystemSrc/planet.cxx":20
#1  0xb7387cec in   do global ctors aux(...) in /lib/libdl-2.3.2.so
(idb) cont
(1) Planet [Mars]; <Sol 4> orbits at 227940 Megameters
Called Procedure Returned
stopped at [void buildOurSolarSystem(class Star*):55 0x080581c6]
     55     Planet *pluto     = new Planet("Pluto",      5913520, sun);
```

Restrictions on the `call` Command

The debugger supports function calls and expression evaluations that call functions, with the following limitations:

- The debugger does not support passing and returning structures by value.
- The debugger does not implicitly construct temporary objects for call parameters.
- Optimization can prevent the debugger from knowing the type of a function return. Therefore, the debugger assumes returns are of the type `int` if the functions are optimized. If the returns are a different type, it may be necessary to cast the result when calling the optimized functions.

The `whatis` Command

You can print information about the basic nature of a *whatis_expression*. The expression can be a normal language expression or the name of a type, function, or other language entity. The debugger shows you information about the entity rather than evaluating it. However, it will evaluate any contained expressions, such as pointers, needed to determine the entity to which you are referring.

whatis_command

        : **whatis** *whatis_expression*

The following example uses the **whatis** command to determine the storage representation for the data member `_classification`:

```
(idb) whatis sun-> classification
const enum StellarClass Star:: classification
(idb) whatis StellarClass
enum StellarClass {O, B, A, F, G, K, M, R, N, S}
(idb) print sun-> classification
G
```

The `whereis` Command

The **whereis** command lists all declarations of a variable and each declaration's fully qualified scope information.

The scope information of a variable usually consists of the name of the source file that contains the function in which the variable is declared, the name of that function, and the name of the variable. The components of the scope information are separated by back-quotes (`).

whereis_command

        : **whereis** *whereis_name*

        | **whereis** *whereis_string*

whereis_name

        : *identifier_or_typedef_name*

        | (*identifier_or_typedef_name*)

whereis_string

      : *string*

You can use the **`whereis`** command with the *`whereis_name`* to obtain information needed to differentiate overloaded identifiers that are in different units, or within different routines in the same unit. The following example shows how to set breakpoints in two C++ methods, both named `print`:

```
(idb) whereis printBody
"/home/user/examples/solarSystemSrc/heavenlyBody.cxx"`HeavenlyBody::printBody(
const class HeavenlyBody*, unsigned int)
"/home/user/examples/solarSystemSrc/heavenlyBody.cxx"`HeavenlyBody::printBody(
const class HeavenlyBody*, unsigned int)
"/home/user/examples/solarSystemSrc/heavenlyBody.cxx"`HeavenlyBody::printBody(
const class HeavenlyBody*, unsigned int)
"/home/user/examples/solarSystemSrc/heavenlyBody.cxx"`HeavenlyBody::printBody(
const class HeavenlyBody*, unsigned int)
"/home/user/examples/solarSystemSrc/heavenlyBody.cxx"`HeavenlyBody::printBody(
const class HeavenlyBody*, unsigned int)
"/home/user/examples/solarSystemSrc/heavenlyBody.cxx"`HeavenlyBody::printBody(
const class HeavenlyBody*, unsigned int)
"/home/user/examples/solarSystemSrc/heavenlyBody.cxx"`HeavenlyBody::printBody(
const class HeavenlyBody*, unsigned int)
"/home/user/examples/solarSystemSrc/heavenlyBody.cxx"`HeavenlyBody::printBody(
const class HeavenlyBody*, unsigned int)
"/home/user/examples/solarSystemSrc/heavenlyBody.cxx"`HeavenlyBody::printBody(
const class HeavenlyBody*, unsigned int)
"/home/user/examples/solarSystemSrc/heavenlyBody.cxx"`HeavenlyBody::printBody(
const class HeavenlyBody*, unsigned int)
"/home/user/examples/solarSystemSrc/heavenlyBody.cxx"`HeavenlyBody::printBody(
const class HeavenlyBody*, unsigned int)
"/home/user/examples/solarSystemSrc/heavenlyBody.cxx"`HeavenlyBody::printBody(
const class HeavenlyBody*, unsigned int)
"/home/user/examples/solarSystemSrc/heavenlyBody.cxx"`HeavenlyBody::printBody(
const class HeavenlyBody*, unsigned int)
"/home/user/examples/solarSystemSrc/heavenlyBody.cxx"`HeavenlyBody::printBody(
const class HeavenlyBody*, unsigned int)
"/home/user/examples/solarSystemSrc/heavenlyBody.cxx"`HeavenlyBody::printBody(
const class HeavenlyBody*, unsigned int)
"/home/user/examples/solarSystemSrc/heavenlyBody.cxx"`HeavenlyBody::printBody(
const class HeavenlyBody*, unsigned int)
"/home/user/examples/solarSystemSrc/heavenlyBody.cxx"`HeavenlyBody::printBody(
const class HeavenlyBody*, unsigned int)
"/home/user/examples/solarSystemSrc/heavenlyBody.cxx"`HeavenlyBody::printBody(
const class HeavenlyBody*, unsigned int)
"/home/user/examples/solarSystemSrc/heavenlyBody.cxx"`HeavenlyBody::printBody(
const class HeavenlyBody*, unsigned int)
"/home/user/examples/solarSystemSrc/heavenlyBody.cxx"`HeavenlyBody::printBody(
const class HeavenlyBody*, unsigned int)
```

```
"/home/user/examples/solarSystemSrc/heavenlyBody.cxx"`HeavenlyBody::printBody(
const class HeavenlyBody*, unsigned int)
"/home/user/examples/solarSystemSrc/heavenlyBody.cxx"`HeavenlyBody::printBody(
const class HeavenlyBody*, unsigned int)
"/home/user/examples/solarSystemSrc/heavenlyBody.cxx"`HeavenlyBody::printBody(
const class HeavenlyBody*, unsigned int)
"/home/user/examples/solarSystemSrc/heavenlyBody.cxx"`HeavenlyBody::printBody(
const class HeavenlyBody*, unsigned int)
"/home/user/examples/solarSystemSrc/heavenlyBody.cxx"`HeavenlyBody::printBody(
const class HeavenlyBody*, unsigned int)
"/home/user/examples/solarSystemSrc/planet.cxx"`Moon::printBody(unsigned int)
"/home/user/examples/solarSystemSrc/planet.cxx"`Moon::printBody(unsigned int)
"/home/user/examples/solarSystemSrc/planet.cxx"`Planet::printBody(unsigned
int)
"/home/user/examples/solarSystemSrc/planet.cxx"`Planet::printBody(unsigned
int)
"/home/user/examples/solarSystemSrc/star.cxx"`Star::printBody(unsigned int)
"/home/user/examples/solarSystemSrc/star.cxx"`Star::printBody(unsigned int)
(idb) stop in "star.h"`Star::printBody
Select from
----------------------------------------------------
     1 /home/user/examples/solarSystemSrc/main/solarSystem.cxx
     2 /home/user/examples/solarSystemSrc/star.cxx
     3 None of the above
----------------------------------------------------
1
[#2: stop in virtual void Star::printBody(unsigned int)]
```

See also the **`which`** command for another example of the **`whereis`** command.

If you are not sure how to spell a symbol, you can use the **`whereis`** command with the *`whereis_string`* to search the symbol table for the regular expression represented by the quoted string. All symbols that match the rules of the regular expression are displayed in ascending order. For example:

```
(idb) whereis planet
Symbol not found
(idb) whereis "[Pp]lanet"
"solarSystemSrc/derived class includes/planet.h"`Moon::Moon(char*, Megameters,
Kilometers, class Planet*)
"solarSystemSrc/derived class includes/planet.h"`Planet
"solarSystemSrc/derived class includes/planet.h"`Planet
"solarSystemSrc/derived class includes/planet.h"`Planet
"solarSystemSrc/derived class includes/planet.h"`Planet::Planet(char*,
Megameters, class HeavenlyBody*)
"solarSystemSrc/derived class includes/planet.h"`Planet::Planet(char*,
Megameters, class HeavenlyBody*)
"solarSystemSrc/derived class includes/planet.h"`Planet::print(unsigned int)
"solarSystemSrc/derived class includes/planet.h"`  INTER  Moon Moon Orbit Plan
et Xv
"solarSystemSrc/derived class includes/planet.h"`  INTER  Planet Planet Orbit
Xv
"solarSystemSrc/derived class includes/planet.h"`  dt  6PlanetXv
  T 6Planet
  cxxexsig6Planet
  vtbl 5Orbit6Planet
  vtbl 5Orbit6Planet4Moon
  vtbl 6Planet
solarSystemSrc/derived_class_includes/planet.h
```

```
solarSystemSrc/derived class includes/planet.h
solarSystemSrc/derived class includes/planet.h
solarSystemSrc/planet.cxx
(idb) whereis "^Planet$"
"solarSystemSrc/derived_class_includes/planet.h"`Planet
"solarSystemSrc/derived class includes/planet.h"`Planet
"solarSystemSrc/derived class includes/planet.h"`Planet
"solarSystemSrc/derived_class_includes/planet.h"`Planet::Planet(char*,
Megameters, class HeavenlyBody*)
(idb) whereis Planet
"solarSystemSrc/derived class includes/planet.h"`Planet
"solarSystemSrc/derived class includes/planet.h"`Planet
"solarSystemSrc/derived class includes/planet.h"`Planet
"solarSystemSrc/derived class includes/planet.h"`Planet::Planet(char*,
Megameters, class HeavenlyBody*)
(idb) which Planet
"solarSystemSrc/derived class includes/planet.h"`Planet
(idb) whatis Planet
class Planet : HeavenlyBody, Orbit {
  Planet(char*, Megameters, class HeavenlyBody*);
  virtual void print(unsigned int);
}
```

You can use the `$symbolsearchlimit` debugger variable to specify the maximum number of symbols that will be returned by the **`whereis`** command for a regular expression search. The default value for the `$symbolsearchlimit` variable is 100; a value of 0 indicates no limit.

The `which` Command

Use the **`which`** command to determine which declaration an identifier resolves to. The **`which`** command shows the fully qualified scope information for the instance of the specified expression visible from the current scope.

The scope information of a variable usually consists of the name of the source file that contains the function in which the variable is declared, the name of that function, and the name of the variable. The components of the scope information are separated by back-quotes (`).

which_command

        : **which** *which_name*

which_name

        : *identifier_or_typedef_name*

        | (*identifier_or_typedef_name*)

The following example shows how to use the **`whereis`** and **`which`** commands to determine a variable's scope:

```
(idb) where 4
>0  0x080553fd in ((Planet*)0x806e298)->Planet::printBody(i=2)
"/home/user/examples/solarSystemSrc/planet.cxx":19
#1  0x0804d697 in ((HeavenlyBody*)0x806e298)-
>HeavenlyBody::printBodyAndItsSatellites(i=2)
"/home/user/examples/solarSystemSrc/heavenlyBody.cxx":62
#2  0x0804d6d2 in ((HeavenlyBody*)0x806e168)-
>HeavenlyBody::printBodyAndItsSatellites(i=1)
"/home/user/examples/solarSystemSrc/heavenlyBody.cxx":68
#3  0x08058bf2 in main()
"/home/user/examples/solarSystemSrc/main/solarSystem.cxx":120
(idb) which i
"/home/user/examples/solarSystemSrc/planet.cxx"`Planet::printBody(unsigned
int)`i
(idb) assign i = 10
(idb) print i
10
(idb) whereis i
"/home/user/examples/solarSystemSrc/heavenlyBody.cxx"`HeavenlyBody::printBodyA
ndItsSatellites(unsigned int)`i
"/home/user/examples/solarSystemSrc/heavenlyBody.cxx"`HeavenlyBody::printBodyA
ndItsSatellites(unsigned int)`i
"/home/user/examples/solarSystemSrc/heavenlyBody.cxx"`HeavenlyBody::satelliteN
umber(class HeavenlyBody*)`i
"/home/user/examples/solarSystemSrc/main/solarSystem.cxx"`main`i
"/home/user/examples/solarSystemSrc/main/solarSystem.cxx"`printBiggestMoons`i
"/home/user/examples/solarSystemSrc/main/solarSystem.cxx"`trackBiggestMoons(cl
ass Moon*)`i
"/home/user/examples/solarSystemSrc/planet.cxx"`Moon::printBody(unsigned
int)`i
"/home/user/examples/solarSystemSrc/planet.cxx"`Planet::printBody(unsigned
int)`i
"/home/user/examples/solarSystemSrc/star.cxx"`Star::printBody(unsigned int)`i
(idb) func HeavenlyBody::printBodyAndItsSatellites
void HeavenlyBody::printBodyAndItsSatellites(unsigned int) in
/home/user/examples/solarSystemSrc/heavenlyBody.cxx line No. 62:
     62     printBody(i);      {static int somethingToReturnTo;
somethingToReturnTo++; }
(idb) which i
"/home/user/examples/solarSystemSrc/heavenlyBody.cxx"`HeavenlyBody::printBodyA
ndItsSatellites(unsigned int)`i
(idb) print i
2
```

Notes on C++ Debugging

The following sections describe the debugger commands specific to debugging C++ programs for

- Setting the class scope using the class command
- Displaying class Information
- Displaying object information
- Displaying static and dynamic type information
- Displaying virtual and inherited class information

- Resolving ambiguous references to overloaded functions
- Printing information on virtual base class pointers

Setting the Class Scope Using the `class` Command

The debugger maintains the concept of a current context in which to perform lookup of program variable names. The current context includes a file scope and either a function scope or a class scope. The debugger automatically updates the current context when program execution suspends.

The **`class`** command lets you set the scope to a class in the program you are debugging:

```
c++_look_around_command

        : class [ class_name ]
```

If `class_name` is not specified, the **`class`** command displays the current class context.

Setting the class scope nullifies the function scope and vice versa. To return to the default (current function) scope, use the command **`func 0`**.

Explicitly setting the debugger's current context to a class enables you to view a class to:

- Set a breakpoint in a member function
- Print static data members
- Examine any data member's type

After the class scope is set, you can set breakpoints in the class's member functions and examine data without explicitly mentioning the class name. If you do not want to affect the current context, you can use the scope resolution operator (::) to access a class whose members are not currently visible. Use the **`class`** command without an argument to display the current class scope. Specify an argument to change the class scope. After the class scope is set, refer to members of the class by omitting the `classname::` prefix.

The following example shows the use of the **`class`** command to set the class scope to `List<Node>` in order to make member function `append` visible so a breakpoint can be set in `append`:

```
(idb) stop in append
Symbol "append" is not defined.
append has no valid breakpoint address
Warning: Breakpoint not set
(idb) class List<Node>
class List<Node> {
  class Node*  firstNode;
  List(void);
  void append(class Node* const);
  void print(void);
  ~List(void);
}
```

```
(idb) stop in append
[#1: stop in void List<Node>::append(class Node* const)]
```

Displaying Class Information

The **`whatis`** and **`print`** commands display information on a class. Use the **`whatis`** command to display static information about the classes. Use the **`print`** command to view dynamic information about class objects.

The **`whatis`** command displays the class type declaration, including the following:

- Data members
- Member functions
- Constructors
- Destructors
- Static data members
- Static member functions

For classes that are derived from other classes, the data members and member functions inherited from the base class are not displayed. Any member functions that are redefined from the base class are displayed.

The **`print`** command lets you display the value of data members and static members. Information regarding the public, private, or protected status of class members is not provided, because the debugger relaxes the related access rules to be more helpful to users.

The type signatures of member functions, constructors, and destructors are displayed in a form that is appropriate for later use in resolving references to overloaded functions.

The following example shows the **`whatis`** and **`print`** commands in conjunction with a class:

```
(idb) list 43:12
     43 // Compound Node - contains integer and float data items
     44 //
     45 class CompoundNode : public IntNode {
     46 public:
     47     CompoundNode (float fdata, int idata);
     48
     49     void printNodeData() const;
     50
     51 private:
     52     float  fdata;
     53 };
     54
(idb) whatis CompoundNode
class CompoundNode : IntNode {
  float  fdata;
  CompoundNode(float, int);
  virtual void printNodeData(void);
}
(idb) whatis CompoundNode::CompoundNode
CompoundNode::CompoundNode(float, int)
(idb) stop in CompoundNode::printNodeData
[#1: stop in virtual void CompoundNode::printNodeData(void)]
```

```
(idb) run
The list is:
Node 1 type is integer, value is 1
[1] stopped at [virtual void CompoundNode::printNodeData(void):109 0x080535fa]
    109     cout << " type is compound, value is ";
(idb) print  fdata
12.3450003
```

Displaying Object Information

The **`whatis`** and **`print`** commands display information on instances of classes (objects). Use the **`whatis`** command to display the class type of an object. Use the **`print`** command to display the current value of an object.

You can also display individual object members using the member access operators, period (.) and right arrow (->), in a **`print`** command.

You can use the scope resolution operator (::) to refer to global variables, to refer to hidden members in base classes, to explicitly refer to a member that is inherited, or to name a member hidden by the current context.

When you are in the context of a nested class, you must use the scope resolution operator to access members of the enclosing class.

The following example shows how to use the **`whatis`** and **`print`** commands to display object information:

```
(idb) whatis this
const class CompoundNode* const this
(idb) whatis *this
class CompoundNode : IntNode {
  float  fdata;
  CompoundNode(float, int);
  virtual void printNodeData(void);
}
(idb) print *this
class CompoundNode {
   fdata = 12.3450003;
   data = 2;                    // class IntNode
   nextNode = 0x805e620;        // class IntNode::Node
}
(idb) print  fdata,  data
12.3450003 2
(idb) print this-> fdata, this-> data
12.3450003 2
```

Displaying Static and Dynamic Type Information

When displaying object information for C++ class pointers or references, you have the option of viewing either static type information or dynamic type information.

The static type of a class pointer or reference is its type as defined in the source code, and thus cannot change. The dynamic type is the type of the object being referenced, before any casts were made to that object, and thus may change during program execution.

The debugger provides a debugger variable, `$usedynamictypes`, which allows you to control which form of the type information is displayed. The default value for this variable is true (1), which indicates that the dynamic type information is displayed. Setting this variable to false (0) instructs the debugger to display static type information. The output of the **`print, trace, tracei,`** and **`whatis`** commands are affected.

The display of dynamic type information is supported for C++ class pointers and references. All other types display static type information. In addition, if the dynamic type of an object cannot be determined, the debugger defaults to the use of static type information.

This debugger functionality does not relax the C++ visibility rules regarding object member access through a pointer/reference (only members of the static type are accessible). For more information about the C++ visibility rules, see *The Annotated C++ Reference Manual* (by Margaret E. Ellis and Bjarne Stroustrup, 1990, Addison-Wesley Publishing Company).

In order for dynamic type information to be displayed, the object's static type must have at least one virtual function defined as part of its interface (either one it introduced or one it inherited from a base class). If no virtual functions are present for an object, only the static type information for that object is available for display.

The following example shows debugger output with `$usedynamictypes` set to 0 (false):

```
(idb) print $usedynamictypes
0
(idb) whatis *this
class HeavenlyBody {
  const char* const _name;
  class HeavenlyBody* _innerNeighbor;
  class HeavenlyBody* _outerNeighbor;
  class HeavenlyBody* _firstSatellite;
  class HeavenlyBody* _lastSatellite;
  virtual void printBody(const class HeavenlyBody*, unsigned int);
  void printBodyAndItsSatellites(unsigned int);
  HeavenlyBody(char*);
  void addSatellite(class HeavenlyBody*);
  const char* name(void);
  unsigned int satelliteNumber(class HeavenlyBody*);
}
(idb) print *this
class HeavenlyBody {
  _name = 0x805abe4="Moon";
  _innerNeighbor = 0x0;
  _outerNeighbor = 0x0;
  _firstSatellite = 0x0;
  _lastSatellite = 0x0;
}
```

The following example displays debugger output with `$usedynamictypes` set to 1 (true). The output is for the same object as the previous example, at the same point in program execution:

```
(idb) print $usedynamictypes
1
(idb) whatis *this
class Moon : Planet {
  const Kilometers  radius;
  Moon(char*, Megameters, Kilometers, class Planet*);
  Kilometers radius(void);
  virtual void printBody(unsigned int);
  virtual ~Moon(void);
}
(idb) print *this
class Moon {
   radius = 1738;
   name = 0x805abe4="Moon";        // class Planet::HeavenlyBody
   innerNeighbor = 0x0;            // class Planet::HeavenlyBody
   outerNeighbor = 0x0;            // class Planet::HeavenlyBody
   firstSatellite = 0x0;           // class Planet::HeavenlyBody
   lastSatellite = 0x0;            // class Planet::HeavenlyBody
   primary = 0x806b4e0;            // class Planet::Orbit
   distance = 384;                 // class Planet::Orbit
   name = 0x806b5c0="Earth 1";     // class Planet::Orbit
}
```

Displaying Virtual and Inherited Class Information

When you use the `print` command to display information on an instance of a derived class, the debugger displays both the new class members as well as the members inherited from a base class. Pointers to members of a class are not supported.

When you use the `print` command to display the format of C++ classes, the class name (or structure/union name) is displayed at the top of the output. Data members of a class that are inherited from another class are commented using a double slash (//). Only those data members that are inherited within the current class being printed are commented.

The following example shows how the debugger uses C++ style comments to identify inherited class members. In the example, class CompoundNode inherits from class IntNode, which inherits from class Node. When printing a class CompoundNode object, the data member _data is commented with "// class IntNode", signifying that it is inherited from class IntNode. The member _nextNode is commented with "// class IntNode::Node", showing that it is inherited from class IntNode, which inherits it from class Node. This commenting is also provided for C++ structs.

```
(idb) where 3
>0  0x0804d37e in ((IntNode*)0x805fb18)->IntNode::printNodeData()
"src/x list.cxx":93
#1  0x0804d4c6 in ((CompoundNode*)0x805fb18)->CompoundNode::printNodeData()
"src/x list.cxx":112
#2  0x08050c43 in ((List<Node>*)0xbffec758)->List<Node>::print()
"src/x list.cxx":168
(idb) whatis *this
class CompoundNode : IntNode {
  float  fdata;
  CompoundNode(float, int);
  virtual void printNodeData(void);
}
```

```
(idb) print *this
class CompoundNode {
   fdata = 12.3450003;
   data = 2;                      // class IntNode
  _nextNode = 0x805fb30;          // class IntNode::Node
}
(idb) up 1
>1  0x0804d4c6 in ((CompoundNode*)0x805fb18)->CompoundNode::printNodeData()
"src/x list.cxx":112
    112     IntNode::printNodeData();
(idb) whatis *this
class CompoundNode : IntNode {
  float  fdata;
  CompoundNode(float, int);
  virtual void printNodeData(void);
}
(idb) print *this
class CompoundNode {
   fdata = 12.3450003;
   data = 2;                      // class IntNode
   nextNode = 0x805fb30;          // class IntNode::Node
}
```

If two members in an object have the same name but different base class types (multiple inheritance), you can refer to the members using the following syntax:

`object.class::member`

or

`object->class::member`

This syntax is more effective than using the *`object.member`* and *`object->member`* syntaxes, which can be ambiguous. In all cases, the debugger uses the C++ language rules as defined in *The Annotated C++ Reference Manual* to determine which member you are specifying.

The following example shows a case in which the expanded syntax can be used:

```
(idb) print *jupiter
class Planet {
   name = 0x805c5b8="Jupiter";    // class HeavenlyBody
   innerNeighbor = 0x806e3a0;     // class HeavenlyBody
   outerNeighbor = 0x806e6b8;     // class HeavenlyBody
   firstSatellite = 0x806e500;    // class HeavenlyBody
   lastSatellite = 0x806e660;     // class HeavenlyBody
   primary = 0x806e168;           // class Orbit
   distance = 778330;             // class Orbit
   name = 0x806e4d8="Sol 5";      // class Orbit
}
(idb) print jupiter-> name
Overloaded Values
0x806e4d8="Sol 5"
0x805c5b8="Jupiter"
(idb) print jupiter->HeavenlyBody:: name
0x805c5b8="Jupiter"
(idb) print jupiter->Orbit:: name
0x806e4d8="Sol 5"
```

Member Functions on the Stack Trace

The implicit `this` pointer, which is a part of all non-static member functions, is displayed as the address on the stack trace. The class type of the object is also given.

Sometimes the debugger does not see class type names with internal linkage. When this happens, the debugger issues the following error message:

```
Name is overloaded.
```

Trying to examine an inlined member function that is not called results in the following error:

```
Member function has been inlined.
```

The debugger will report this error regardless of the setting of the `-noinline_auto compilation` flag. As a workaround, include a call to the given member function somewhere in your program. (The call does not need to be executed.)

If a program is not compiled with the -g option, a breakpoint set on an inlined member function may confuse the debugger.

Resolving Ambiguous References to Overloaded Functions

In most cases, the debugger works with one specific function at a time. In the case of overloaded function names, you must specify the desired overloaded function. Following are two ways to resolve references to overloaded function names, both under the control of the `$overloadmenu` debugger variable (the default setting of this debugger variable is 1):

- Choose the correct reference from a selection menu.

  If the `$overloadmenu` variable is set to 1 (the default), whenever you specify a function name that is overloaded, a menu appears with all the possible functions; you must select from this menu. In this example, a breakpoint is set in `foo`, which is overloaded:

```
(idb) set $overloadmenu = 1
(idb) stop in C::foo
Select from
----------------------------------------------------
     1 int C::foo(double*)
     2 void C::foo(float)
     3 void C::foo(int)
     4 void C::foo(void)
     5 None of the above
----------------------------------------------------
```

```
1
[#10: stop in int C::foo(double*)]
```

- Enter the function name with its full type signature.

  If you prefer this method, set the `$overloadmenu` variable to 0. To see the possible type signatures for the overloaded function, first display all the declarations of an overloaded function by using the **`whatis`** command.

  You cannot select a version of an overloaded function that has a type signature containing ellipsis points (...). Pointers to functions with type signatures that contain the list parameter or ellipsis points are not supported.

  Use the specific function type signature to refer to the desired version of the overloaded function. If a function has no parameter, include the void parameter as the function's type signature. In the following example, the function context is set to `foo(double *)`, as `foo` is overloaded:

```
 (idb) func foo
Error: foo is overloaded
(idb) func foo(double *)
int C::foo(double*) in src/x overload.cxx line No. 25:
     25 int  C::foo(double *) { return state;}
```

Advanced Program Information - Verbose Mode

By default, the debugger gives no information on virtual base class pointers for the following:

- Derived classes
- Virtual pointer tables for virtual functions
- Compiler-generated function members
- Compiler-generated temporary variables
- Implicit parameters in member functions

By setting the `$verbose` debugger variable to 1, you can request that this information be printed in subsequent debugger responses. When the `$verbose` debugger variable is set to 1 and you display the contents of a class using the **`whatis`** command, several of the class members listed are not in the source code of the original class definition. The following line shows specific output from the **`whatis`** command for one of the additional members:

```
(idb) whatis CompoundNode::  vptr
(array [subrange 0 ... 0 of int] of union {
  void <member function>(void)* fptr;
  int ioffset;
})* Node::__vptr
```

The `_vptr` variable contains the addresses of all virtual functions associated with the class. The compiler generates several other class members for internal use.

The compiler generates additional parameters for non-static member functions. When the `$verbose` debugger variable is set to 1, these extra parameters are displayed as part of each member function's type signature. If you specify a version of an overloaded function by entering its type signature and the variable is set to 1, you must include these parameters. Do not include these parameters if the variable is set to 0.

When the `$verbose` variable is set to 1, the output of the **`dump`** command includes not only standard program variables but also compiler-generated temporary variables.

The following example prints class information using the **`whatis`** command under different settings of the `$verbose` variable:

```
(idb) set $verbose = 0
(idb) whatis CompoundNode
class CompoundNode : IntNode {
  float  fdata;
  CompoundNode(float, int);
  virtual void printNodeData(void);
}
(idb) set $verbose = 1
(idb) whatis CompoundNode
class CompoundNode : IntNode {
  float  fdata;
  CompoundNode(class CompoundNode* const, float, int);
  virtual void printNodeData(const class CompoundNode* const);
}
```

% Unary Operator

You may wish to swap the bytes of a numeric value, rather than rely on the debugger’s knowledge of representation. The "%" unary operator reverses all the bytes of its integer operand.

For example:

```
(idb) printx %(short)258
0x201
(idb) printx %(int)258
0x2010000
(idb) printx %(long long)258
0x201000000000000
```

Looking at the Generated Code. Expert Debugging. Overview

This section discusses the following topics:

- Memory display and search commands
- Machine-level debugging

Memory Display and Search Commands

You can use the following commands to read arbitrary memory locations in your program:

```
machinecode_level_command

      : examine_command

      : search_command


examine_command

      : address_expression / [ count ] [ mode ]

      | address_expression ? [ count ] [ mode ]

      | address_expression , address_expression / [ mode ]


search_command

      : address_expression / [ count ] search_mode   value  mask

      | address_expression ? [ count ]  search_mode  value  mask

      | address_expression , address_expression / search_mode  value  mask


count

      : integer_constant


mode

      : d     Print a short word in decimal

      | dd    Print a 32-bit (4-byte) decimal display

      | D     Print a long word in decimal

      | u     Print a short word in unsigned decimal

      | uu    Print a 32-bit (4-byte) unsigned decimal display

      | U     Print a long word in unsigned decimal
```

```
    | o     Print a short word in octal

    | oo    Print a 32-bit (4-byte) octal display

    | O     Print a long word in octal

    | x     Print a short word in hexadecimal

    | xx    Print a 32-bit (4-byte) hexadecimal display

    | X     Print a long word in hexadecimal

    | b     Print a byte in hex

    | c     Print a byte as a character

    | s     Print a string of characters (a C-style string ending in null)

    | C     Print a wide character as a character

    | S     Print a null terminated string of wide characters

    | f     Print a single precision real number

    | g     Print a double precision real number

    | L     Print a long double precision real number

    | i     Disassemble machine instructions


search_mode

    : m     32-bit search mode

    | M     64-bit search mode


value

    : integer_constant

mask

    : integer_constant
```

The first `examine_command` displays the `count` number of memory values in the requested format, starting at `address_expression`. If `count` is not specified, 1 is assumed. The count value must be a positive value.

If you wish to see memory values leading up to the `address_expression`, use the second `examine_command`. The second `examine_command` displays `count` number of memory values in the requested format ending at the `address_expression`. If `count` is not specified, 1 is assumed. The count value must be a positive value.

The third `examine_command` displays memory values in the requested format starting at the smaller of the two `address_expressions` and ending at the larger `address_expression`.

You can display stored values in the following formats by specifying `mode`. If `mode` is not specified, the mode used in the previous `/` command is assumed. If no previous `/` command exists, `X` is assumed.

When disassembling machine instructions, use the `$regstyle` variable to customize how the registers are displayed.

The `search_commands` allow you to search memory. Use the `address_expression` and `count` to determine the range of memory to search. Use the `search_mode` to specify whether you want to search 32 or 64-bit chunks. The debugger will start at the specified starting address and read a chunk of memory (either 32 or 64 bits in size) and apply the mask and comparison on that chunk of memory. For example, if you want to search memory for a particular instruction or search an array of either integer or floating-point values, the 32-bit search would be efficient because machine instructions and integer and floating-point data types are 32 bits in length. Use the `value` to specify the memory value to seek. Use the `mask` to specify those bits that must match the same bits in the specified value. To ensure that a possible match will be found, the debugger applies the `mask` to the input value prior to starting the search, to remove any bits that could prevent a match from occurring. Then, for each memory location searched, the debugger applies the `mask` to the memory value and then compares it with this new input value. If a match is found, then the address and memory value are displayed.

For example, suppose the user wishes to check an array of 100 integers in memory to see if any values are NULL (0):

```
(idb) array,&(array[99])/m 0x0 0xffffff
0x1400005d0:  0x00000000
```

Suppose the user wishes to search for the `trapb` instruction:

```
(idb) printi 0x63ff0000
trapb
(idb) main/1000m 0x63ff0000 0xffffff
0x12561314: 0x63ff0000
```

Use the debugger variable `$memorymatchall` to cause the debugger to output all matches in the specified range. Suppose you want to search a long integer array of 100 values for the first value over 80, and then want to find all values in the array over 80:

```
(idb) printx 80
0x50
(idb) longarray/100M 0x50 0x50
0x140002680: 0x0000000000000050
(idb) set $memorymatchall
(idb) longarray/100M 0x50 0x50
0x140002680: 0x0000000000000050
0x140002688: 0x0000000000000051
0x140002690: 0x0000000000000052
0x140002698: 0x0000000000000053
0x1400026a0: 0x0000000000000054
0x1400026a8: 0x0000000000000055
0x1400026b0: 0x0000000000000056
0x1400026b8: 0x0000000000000057
0x1400026c0: 0x0000000000000058
0x1400026c8: 0x0000000000000059
0x1400026d0: 0x000000000000005a
0x1400026d8: 0x000000000000005b
0x1400026e0: 0x000000000000005c
0x1400026e8: 0x000000000000005d
0x1400026f0: 0x000000000000005e
0x1400026f8: 0x000000000000005f
(idb)
```

Machine-Level Debugging. Expert Debugging

The debugger lets you debug your programs at the machine-code level as well as at the source-code level. Using debugger commands, you can examine and edit values in memory, print the values of all machine registers, and step through program execution one machine instruction at a time.

Only those users familiar with machine-language programming and executable file code structure will find low-level debugging useful.

See also

Machine-Level Debugging

Looking at the Libraries

```
shared_library_command

        : listobj

        | readsharedobj filename

        | delsharedobj  filename
```

Use the **`listobj`** command to list all loaded objects, including the main image and the shared libraries. For each object, the information listed consists of the full object name (with pathname) and the starting and ending addresses for the `.text`, `.data`, and `.bss` sections.

Use the **`readsharedobj`** command to read in the symbol table information for the specified shared object. This object must be a shared library. You can use the command only when you specify the debuggee; that is, either the debugger has been invoked with it, or the debuggee was loaded by the **`load`** command.

Conversely, use the **`delsharedobj`** command to remove the symbol table information for the shared object from the debugger.

Modifying the Process

In addition to the normal side effects of evaluating expressions, including calls, you can explicitly modify the memory of the current process and also modify the actual loadable file (either executable file or shared library) that has been mapped into memory.

Note:

This section discusses these commands.

The `assign` and the `set variable` Commands

Use the **`assign`** (dbx) and the **`set variable`** (gdb) commands to change the value associated with a variable, memory address, or expression that is accessible according to the scope and visibility rules of the language. The expression can be any expression that is valid in the current context.

DBX Mode

```
modifying_command

        : assign target = expression

        | patch target = expression



target

        : unary_expression
```

GDB Mode

```
modifying_command

        : set [ variable ] expression
```

The **`set variable`** command evaluates the specified expression. If expression includes assignment operator, it executes like all other operators. This is the way to change memory.

The only difference between the **`set variable`** and the **`print`** commands is printing the value - the **`set variable`** does not print anything.

Note:

> The **`variable`** can be omitted if the beginning of *`expression`* does not confuse the debugger, for example, does not look like a valid subcommand for the **`set`** command.

The following example shows how to deposit the value `5` into the data member `_data` of a C++ object:

DBX Mode

```
(idb) print node-> data
2
(idb) assign node-> data = 5
(idb) print node-> data
5
```

GDB Mode

```
(idb) print node-> data
$2 = 2
(idb) set variable node-> data = 5
(idb) print node-> data
$3 = 5
```

The following example shows how to change the value associated with a variable and the value associated with an expression:

DBX Mode

```
(idb) print *node
class CompoundNode {
   fdata = 12.3450003;
   data = 5;                    // class IntNode
   nextNode = 0x0;              // class IntNode::Node
}
(idb) assign node-> data = -32
(idb) assign node-> fdata = 3.14 * 4.5
(idb) assign node-> nextNode =  firstNode
(idb) print *node
class CompoundNode {
   fdata = 14.1300001;
   data = -32;                  // class IntNode
   nextNode = 0x805e5e0;        // class IntNode::Node
}
```

GDB Mode

```
(idb) print *node
$6 = {<IntNode> = {<Node> = {_nextNode = 0x0}, _data = 5}, _fdata = 12.345}
(idb) set variable node->_data = -32
```

```
(idb) set variable node-> fdata = 3.14 * 4.5
(idb) set variable node-> nextNode =  firstNode
(idb) print *node
$7 = {<IntNode> = {<Node> = { nextNode = 0x805e5e0},  data = -32},  fdata =
14.13}
```

For C++, use the **`assign`** (dbx) and the **`set variable`** (gdb) commands to modify static and object data members in a class, and variables declared as reference types, type const, or type static. You cannot change the address referred to by a reference type, but you can change the value at that address.

```
assign [classname::]member = ["filename"] `expression
assign [object.]member = ["filename"] `expression
```

Note:

Do not use the **`assign`** (dbx) and the **`set variable`** (gdb) commands to change the PC. When you change the PC, no adjustment to the contents of registers or memory is made. Because most instructions change registers or memory in ways that can impact the meaning of the application, changing the PC is very likely to cause your application to do incorrect calculations and arrive at the wrong answer. Access violations and other errors and signals may result from changing the value in the PC.

The `assign` and the `set variable` Commands in Machine-Level Debugging

You can use the **`assign`** (dbx) and the **`set variable`** (gdb) commands to alter the contents of memory specified by an address as shown in the following example:

```
(idb) set $address = &(node-> data)
(idb) print $address
0x805e5f8
(idb) print *(int *)($address)
-32
(idb) assign *(int *)($address) = 1024
(idb) print *(int *)($address)
1024
```

GDB Mode

```
(idb) set variable $address = &(node-> data)
(idb) print $address
$11 = (int *) 0x805e5f8
(idb) print *(int *)($address)
$12 = -32
(idb) set variable *(int *)($address) = 1024
(idb) print *(int *)($address)
$13 = 1024
```

See Machine-Level Debugging for more information.

The `patch` Command (DBX Mode only)

Use the **`patch`** command to correct bad data or instructions in executable disk files. You can patch the text, initialized data, or read-only data areas. You cannot patch the `bss` segment, or stack and register locations, because they do not exist on disk files.

Use this command exclusively when you need to change the on-disk binary. Use the **`assign`** command when you need only to modify debuggee memory. If the image is executing when you issue the **`patch`** command, the corresponding location in the debuggee address space is updated as well. (The debuggee is updated regardless of whether the patch to disk succeeded, as long as the source and destination expressions can be processed by the **`assign`** command.) If your program is loaded but not yet started, the patch to disk is performed without the corresponding assign to memory.

```
(idb) run
[1] stopped at [int main(void):24 0x120001324]
24     return 0;
(idb) patch i = 10
0x1400000d0 = 10
(idb) patch j = i + 12
0x1400000d8 = 22
(idb)
```

Note:

When you use the **`patch`** command, the original binary is not overwritten, but is saved with the string `~backup` appended to the file name. This allows you to revert to the original binary if necessary. A file with the string `~temp` appended to the file name may also be created. It may be deleted after the debugging session is over.

Continuing Execution of the Process. Expert Debugging. Overview

This section gives a detailed description of

- The step and stepi commands
- The next and nexti commands
- The return command
- The cont command
- The goto command
- The finish command

Before continuing the process, you may make a snapshot of the current state of the process, so that in the future you can revert to this state and try a different set of steps. After creating the snapshot, use the following commands to continue executing the program:

```
continue_command

        : step_into_command

        | step_over_command

        | step_out_of_command

        | cont_command

        | cont_from_place_command

        | finish_command
```

The `step` and `stepi` Commands

Use the **`step`** command to execute a line of source code. When the line being stepped contains a function call, the **`step`** command steps into the function and stops at the first executable statement.

Use the **`stepi`** command to step into the next machine instruction. When the instruction contains a function call, the **`stepi`** command steps into the function being called.

For multithreaded applications, use the **`stepi`** command to step the current thread one machine instruction while putting all other threads on hold.

If you supply the optional expression argument, the debugger evaluates the expression as a positive integer that specifies the number of times to execute the command. The expression can be any expression that is valid in the current context.

```
step_into_command

        : step  [ step_number ]

        | stepi [ step_number ]

step_number

        :  expression
```

In the following example, two **`step`** commands continue executing a C++ program:

DBX Mode

```
(idb) list $curline:4
>   151         Node* currentNode =  firstNode;
    152         while (currentNode->getNextNode())
    153             currentNode = currentNode->getNextNode();
    154 currentNode->setNextNode(node);
(idb) step
stopped at [void List<Node>::append(class Node* const):152 0x0804c579]
    152         while (currentNode->getNextNode())
(idb) step
stopped at [class Node* Node::getNextNode(void):81 0x080534b4]
     81 Node* Node::getNextNode()            {return  nextNode; }
(idb) step
stopped at [void List<Node>::append(class Node* const):152 0x0804c585]
    152         while (currentNode->getNextNode())
(idb) step
stopped at [void List<Node>::append(class Node* const):154 0x0804c5c3]
    154 currentNode->setNextNode(node);
(idb) step
stopped at [void Node::setNextNode(class Node*):82 0x080534c3]
     82 void  Node::setNextNode(Node* next)  { _nextNode = next;}
```

GDB Mode

```
(idb) list +0,+4
151         Node* currentNode =  firstNode;
152         while (currentNode->getNextNode())
153             currentNode = currentNode->getNextNode();
154 currentNode->setNextNode(node);
(idb) step
152         while (currentNode->getNextNode())
(idb) step
Node::getNextNode (this=0xbfff9fe8) at src/x list.cxx:81
81 Node* Node::getNextNode()            {return _nextNode; }
(idb) step
List<Node>::append (this=0xbfff9fe8, node=0x805e608) at src/x list.cxx:152
152         while (currentNode->getNextNode())
(idb) step
154 currentNode->setNextNode(node);
(idb) step
Node::setNextNode (this=0xbfff9fe8, next=0x805e608) at src/x list.cxx:82
82 void  Node::setNextNode(Node* next)  { _nextNode = next;}
```

The following example shows stepping by instruction (**`stepi`**). To see stepping over calls, see the example of the **`next`** command.

DBX Mode

```
(idb) $curpc/8i
void List<Node>::append(class Node* const): src/x list.cxx
*[line 151, 0x0804c571] append(class Node* const)+0x19:                movlr
   0x8(%ebp), %eax
 [line 151, 0x0804c574] append(class Node* const)+0x1c:                movlr
   (%eax), %eax
 [line 151, 0x0804c576] append(class Node* const)+0x1e:                movl
    %eax, -16(%ebp)
 [line 152, 0x0804c579] append(class Node* const)+0x21:                pushl
   %edi
 [line 152, 0x0804c57a] append(class Node* const)+0x22:                movlr
   -16(%ebp), %eax
 [line 152, 0x0804c57d] append(class Node* const)+0x25:                movl
    %eax, (%esp)
 [line 152, 0x0804c580] append(class Node* const)+0x28:                 call
    getNextNode
 [line 152, 0x0804c585] append(class Node* const)+0x2d:                 addl
    $0x4, %esp
(idb) stepi
stopped at [void List<Node>::append(class Node* const):151 0x0804c574]
append(class Node* const)+0x1c:                 movlr    (%eax), %eax
(idb) stepi $count - 1
stopped at [void List<Node>::append(class Node* const):151 0x0804c574]
append(class Node* const)+0x1c:                 movlr    (%eax), %eax
(idb) stepi
stopped at [void List<Node>::append(class Node* const):151 0x0804c576]
append(class Node* const)+0x1e:                 movl     %eax, -16(%ebp)
```

GDB Mode

```
(idb) x /8i $pc
0x0804c571 <append+25>:                 movlr    0x8(%ebp), %eax
0x0804c574 <append+28>:                 movlr    (%eax), %eax
0x0804c576 <append+30>:                 movl     %eax, -16(%ebp)
0x0804c579 <append+33>:                 pushl    %edi
0x0804c57a <append+34>:                 movlr    -16(%ebp), %eax
0x0804c57d <append+37>:                 movl     %eax, (%esp)
0x0804c580 <append+40>:                 call     0x080534aa <getNextNode>
0x0804c585 <append+45>:                 addl     $0x4, %esp
(idb) stepi
0x0804c574      151         Node* currentNode =  firstNode;
(idb) x /1i $pc
0x0804c574 <append+28>:                 movlr    (%eax), %eax
(idb) stepi $count - 1
0x0804c574      151         Node* currentNode =  firstNode;
(idb) x /1i $pc
0x0804c574 <append+28>:                 movlr    (%eax), %eax
(idb) stepi
0x0804c576      151         Node* currentNode =  firstNode;
(idb) x /1i $pc
0x0804c576 <append+30>:                 movl     %eax, -16(%ebp)
```

The `next` and `nexti` Commands

Use the **`next`** command to execute a line of source code. When the next line to be executed contains a function call, the **`next`** command executes the function being called and stops the process at the line immediately after the function call.

Use the **`nexti`** command to execute a machine instruction. When the instruction contains a function call, the **`nexti`** command executes the function being called and stops the process at the instruction immediately after the call instruction.

If you supply the optional expression argument, the debugger evaluates the expression as a positive integer that specifies the number of times to execute the command. The expression can be any expression that is valid in the current context.

```
step_over_command

        : next  [ step_number ]

        | nexti [ step_number ]
```

step_number

        : *expression*

For example:

DBX Mode

```
(idb) list $curline:4
>   151         Node* currentNode =  firstNode;
    152         while (currentNode->getNextNode())
    153             currentNode = currentNode->getNextNode();
    154 currentNode->setNextNode(node);
(idb) next
stopped at [void List<Node>::append(class Node* const):152 0x0804c579]
    152         while (currentNode->getNextNode())
(idb) next
stopped at [void List<Node>::append(class Node* const):153 0x0804c592]
    153             currentNode = currentNode->getNextNode();
(idb) next
stopped at [void List<Node>::append(class Node* const):152 0x0804c5aa]
    152         while (currentNode->getNextNode())
(idb) next
stopped at [void List<Node>::append(class Node* const):154 0x0804c5c3]
    154 currentNode->setNextNode(node);
```

GDB Mode

```
(idb) list +0,+4
151         Node* currentNode =  firstNode;
152         while (currentNode->getNextNode())
153             currentNode = currentNode->getNextNode();
154 currentNode->setNextNode(node);
(idb) next
152         while (currentNode->getNextNode())
(idb) next
153             currentNode = currentNode->getNextNode();
(idb) next
152         while (currentNode->getNextNode())
(idb) next
154 currentNode->setNextNode(node);
```

The following example shows the difference between **`stepi`** and **`nexti`** over the same call:

DBX Mode

```
(idb) $curpc/8i
void List<Node>::append(class Node* const): src/x list.cxx
*[line 151, 0x0804c571] append(class Node* const)+0x19:                movlr
   0x8(%ebp), %eax
 [line 151, 0x0804c574] append(class Node* const)+0x1c:                movlr
   (%eax), %eax
 [line 151, 0x0804c576] append(class Node* const)+0x1e:                movl
    %eax, -16(%ebp)
 [line 152, 0x0804c579] append(class Node* const)+0x21:                pushl
   %edi
 [line 152, 0x0804c57a] append(class Node* const)+0x22:                 movlr
   -16(%ebp), %eax
 [line 152, 0x0804c57d] append(class Node* const)+0x25:                 movl
    %eax, (%esp)
 [line 152, 0x0804c580] append(class Node* const)+0x28:                 call
    getNextNode
 [line 152, 0x0804c585] append(class Node* const)+0x2d:                 addl
    $0x4, %esp
(idb) nexti
stopped at [void List<Node>::append(class Node* const):151 0x0804c574]
append(class Node* const)+0x1c:                 movlr    (%eax), %eax
(idb) nexti $count - 1
stopped at [void List<Node>::append(class Node* const):151 0x0804c574]
append(class Node* const)+0x1c:                 movlr    (%eax), %eax
(idb) nexti
stopped at [void List<Node>::append(class Node* const):151 0x0804c576]
append(class Node* const)+0x1e:                 movl     %eax, -16(%ebp)
```

GDB Mode

```
(idb) x /8i $pc
0x0804c571 <append+25>:                 movlr    0x8(%ebp), %eax
0x0804c574 <append+28>:                 movlr    (%eax), %eax
0x0804c576 <append+30>:                 movl     %eax, -16(%ebp)
0x0804c579 <append+33>:                 pushl    %edi
0x0804c57a <append+34>:                 movlr    -16(%ebp), %eax
0x0804c57d <append+37>:                 movl     %eax, (%esp)
0x0804c580 <append+40>:                 call     0x080534aa <getNextNode>
0x0804c585 <append+45>:                 addl     $0x4, %esp
(idb) nexti
0x0804c574      151         Node* currentNode =  firstNode;
(idb) x /1i $pc
0x0804c574 <append+28>:                 movlr    (%eax), %eax
(idb) nexti $count - 1
0x0804c574      151         Node* currentNode =  firstNode;
(idb) x /1i $pc
0x0804c574 <append+28>:                 movlr    (%eax), %eax
(idb) nexti
0x0804c576      151         Node* currentNode =  firstNode;
(idb) x /1i $pc
0x0804c576 <append+30>:                 movl     %eax, -16(%ebp)
```

The `return` Command

Use the **`return`** (dbx) or **`finish`** (gdb) command without an argument to continue execution of the current function until it returns to its caller.

Use **`return`** `function_name` (dbx) to continue the execution until control is returned to the specified function. The function must be active on the call stack.

DBX Mode

```
step_out_of_command

        : return

        | return [qual_symbol_opt]
```

qual_symbol_opt

    : *expression*

    | *qual_symbol_opt* ` *expression*

GDB Mode

step_out_of_command

        : **finish**

The **`return`** (dbx) / **`finish`** (gdb) command finishes the `append` method and returns control to the caller.

DBX Mode

```
(idb) cont
[1] stopped at [void List<Node>::append(class Node* const):151 0x0804c571]
    151         Node* currentNode =  firstNode;
(idb) return
stopped at [int main(void):195 0x08053142]
    195     nodeList.append(new IntNode(3)); {static int somethingToReturnTo;
somethingToReturnTo++; }
```

GDB Mode

```
(idb) continue
Continuing.
Breakpoint 1, List<Node>::append (this=0xbfff9fe8, node=0x805e638) at
src/x list.cxx:151
151         Node* currentNode =  firstNode;
(idb) finish
main () at src/x list.cxx:195
195     nodeList.append(new IntNode(3)); {static int somethingToReturnTo;
somethingToReturnTo++; }
```

The **`return`** (dbx) / **`finish`** (gdb) command is sensitive to the user's location in the call stack. Suppose function A calls function B, which calls function C. Execution has stopped in function C, and you entered the **`up`** command, so you were now in function B, at the point where it called function C. Using the **`return`** (dbx) / **`finish`** (gdb) command here would return you to function A, at the point where function A called function B. Functions B and C will have completed execution.

The `cont` Command

Use the **`cont`** (dbx) and **`continue`** (gdb) command without a parameter value to resume process execution until a breakpoint, a signal , an error, or normal process termination is encountered. Specify a signal parameter value to send an operating system signal to the process.

DBX Mode

cont_command

    : **cont** [**in** *loc*]

    | **cont** [*signal*] [*to_source_line*]

    | *number_expression* **cont** [*signal*]

    | **conti to** *address_expression*

to_source_line

    : **to** *line_specifier*

number_expression

    : *expression*

signal

    : *integer_constant*

    | *signal_name*

GDB Mode

cont_command

    : **continue** [*number_expression*]

    | **until** [*place_detector*]

When you use the **cont** (dbx) and **continue** (gdb) command, the debugger resumes execution of the entire process.

In the following example, a **cont** (dbx) / **continue** (gdb) command resumes process execution after it was suspended by a breakpoint.

DBX Mode

```
(idb) list 195:7
>   195     nodeList.append(new IntNode(3)); {static int somethingToReturnTo;
somethingToReturnTo++; }
    196
    197     IntNode* newNode2 = new IntNode(4);
    198     nodeList.append(newNode2); {static int somethingToReturnTo;
somethingToReturnTo++; }
    199
    200     CompoundNode* cNode2 = new CompoundNode(10.123, 5);
    201     nodeList.append(cNode2); {static int somethingToReturnTo;
somethingToReturnTo++; }
(idb) stop at 200
[#2: stop at "src/x list.cxx":200]
(idb) cont
[2] stopped at [int main(void):200 0x08053214]
    200     CompoundNode* cNode2 = new CompoundNode(10.123, 5);
```

GDB Mode

```
(idb) list 195,+7
195     nodeList.append(new IntNode(3)); {static int somethingToReturnTo;
somethingToReturnTo++; }
196
197     IntNode* newNode2 = new IntNode(4);
198     nodeList.append(newNode2); {static int somethingToReturnTo;
somethingToReturnTo++; }
199
200     CompoundNode* cNode2 = new CompoundNode(10.123, 5);
201     nodeList.append(cNode2); {static int somethingToReturnTo;
somethingToReturnTo++; }
(idb) break 200
Breakpoint 2 at 0x8053214: file src/x list.cxx, line 200.
(idb) continue
Continuing.
Breakpoint 2, main () at src/x list.cxx:200
200     CompoundNode* cNode2 = new CompoundNode(10.123, 5);
```

DBX Mode

The signal parameter value can be either a signal number or a string name (for example, SIGSEGV). The default is 0, which allows the process to continue execution without specifying a signal. If you specify a signal parameter value, the process continues execution with that signal.

Use the **`in`** argument to continue until the named function is reached. The function name must be valid. If the function name is overloaded and you do not resolve the scope of the function in the command line, the debugger prompts you with the list of overloaded functions bearing that name from which to choose.

Use the **`to`** parameter value to resume execution and then halt when the specified source line is reached. The form of the optional **`to`** parameter must be either:

- `quoted_filename:line_number`, which explicitly identifies both the source file and the line number where execution is to be halted.
- `line_number`, a positive numeric, which indicates the line number of the current source file where execution is to be halted.

You can repeat the **`cont`** command (*n* +1) times by entering ***`n`*** **`cont`**.

You can set a one-time breakpoint on an instruction address before continuing by entering **`conti to`** *`address_expression`*.

GDB Mode

Use the optional argument **`number_expression`** of the **`continue`** command to specify a further number of times to ignore a breakpoint at this location.

The **`until`** command continues running until a source line past the current line, in the current stack frame, is reached. This command is used to avoid single stepping through a loop more than once.

If **`place_detector`** is specified the command continues running until either the specified location is reached, or the current stack frame returns.

The `goto` Command

DBX Mode

The **`goto`** (dbx) command is intended for the users who want to 'skip over' the execution of a portion of source code.

```
cont_from_place_command

        : goto line_expression

line_expression

        : expression
```

Using Snapshots as an Undo Mechanism

DBX Mode

You can save the current state of the debuggee process in a snapshot, and later revert to that state and try a different set of steps. Conceptually speaking, this feature is similar to the undo function in text editors, except that with snapshots you have control of the granularity of each undo. See the Snapshots as an undo Mechanism for a quick overview.

```
snapshot_command

         : save_snapshot_command

         | clone_snapshot_command

         | show_snapshot_command

         | delete_snapshot_command
```

The following sections discuss these commands and address the limitations of snapshots.

The `save snapshot` Command

Use the **`save snapshot`** command to save the state of the current process in a snapshot. Snapshots are numbered sequentially starting from 1.

```
save_snapshot_command

        : save snapshot
```

In the following example, the first line of the **`save snapshot`** message shows the *snapshot_number* (1), the time it is saved, and the ID number of the process that implements the snapshot. The next two lines show the status of the snapshot.

```
(idb) save snapshot
# 1 saved at 13:27:54 (PID: 29077).
    stopped at [int main(void):182 0x1200023f8]
    182     List<Node> nodeList;
```

The `clone snapshot` Command

Use the **`clone snapshot`** command to revert the state of the debuggee process to that of a previously saved snapshot. By doing this, you can conveniently return to the state saved in the snapshot as opposed to rerunning the process and re-entering the debugger command sequence that brought you to that state.

Note that **`rerun`** and **`clone snapshot`** are different in that **`rerun`** always executes the process from the beginning, whereas **`clone snapshot`** does not execute the process at all; it simply duplicates the saved snapshot (using a mechanism similar to the `fork` system call) and behaves as though the process execution has stopped at the point when the snapshot was saved.

The **`clone snapshot`** command clones the snapshot specified by snapshot_id. If no snapshot_id is specified, the most-recently saved existing snapshot is cloned.

```
clone_snapshot_command

        : clone snapshot  [ snapshot_id ]

snapshot_id

        : expression
```

Cloning a snapshot has two side effects:

- The snapshots created after the cloned snapshot are deleted. For example, suppose four snapshots are saved from a process. Cloning the second snapshot results in the deletion of the third and fourth snapshots.

- The current process is killed and replaced by the clone process. Thus, if you enter **`show process`** after cloning a snapshot, you will see that the process ID of the current process has changed to that of the cloned process. For example:

```
(idb) show process
>localhost:29013 (/home/user/examples/x list) paused.
(idb) clone snapshot
Process has exited
Process 29089 cloned from Snapshot 1.
# 1 saved at 13:27:54 (PID: 29077).
    stopped at [int main(void):182 0x1200023f8]
    182     List<Node> nodeList;
(idb) show process
>localhost:29089 (/home/user/examples/x_list) paused.
```

The `show snapshot` Command

Use the **`show snapshot *`** and **`show snapshot all`** commands to display all the snapshots that have been saved from the current process. Use **`show snapshot`** *snapshot_id_list* to display the snapshots specified. If no snapshots are specified, the most-recently saved existing snapshot is displayed.

show_snapshot_command

        : **show snapshot** [*snapshot_id_list*]

snapshot_id_list

        : *snapshot_id* ,...

        | **all**

        | *****

snapshot_id

        : *expression*

For example:

```
(idb) show snapshot all
# 1 saved at 13:27:54 (PID: 29077).
    stopped at [int main(void):182 0x1200023f8]
    182     List<Node> nodeList;
```

The `delete snapshot` Command

Use the **`delete snapshot *`** and **`delete snapshot all`** commands to delete all the snapshots that have been saved from the current process. Use **`delete snapshot`**

snapshot_id_list to delete the specified snapshots. If no snapshots are specified, the most-recently saved existing snapshot is deleted.

```
 delete_snapshot_command

        : delete snapshot [ snapshot_id_list ]

snapshot_id_list

        : snapshot_id ,...

        | all

        | *

snapshot_id

        : expression
```

For example:

```
(idb) show snapshot all
# 1 saved at 13:27:54 (PID: 29077).
    stopped at [int main(void):182 0x1200023f8]
    182     List<Node> nodeList;
(idb) delete snapshot
(idb) show snapshot all
No snapshots have been saved.
```

Snapshot Limitations

Snapshots have the following limitations:

- Snapshots are implemented by causing the process to fork. The state saved in a snapshot does not include I/O and forks. In other words, when you clone a snapshot, the I/O that has been done since the snapshot was saved is not undone; likewise, the child processes that have been spawned since the snapshot was saved are not killed.
- The **`save snapshot`** command saves the state of the current process only. If you are doing multiprocess debugging, you might want to save a snapshot for each process.
- Snapshots on a multithreaded process are not supported.
- Snapshots are not supported for core file debugging.

Debugging Optimized Code

IDB can help debug an optimized program that is compiled with the `-g` option. However, some of the information about the program may be inaccurate. In particular, the locations and values of variables are often not correctly reported, because the common form of debug information cannot fully represent the complexity of the optimizations provided by the `-O1/2/3` options.

To avoid this limitation, compile the program with an Intel compiler, specifying both the `-g` and `-debug extended` options, in addition to the desired `-O1/2/3` optimization option. This causes the generation of more advanced (but less commonly supported) debug information that enables the following:

- Giving correct locations and values for variables, even if they are in registers or at different locations at different times. Note the following:
  - Some variables may be optimized away or converted to data of a different type, or their location may not be recorded at all points in the program. In these cases, printing a variable will yield `<no value>`.
  - Otherwise, the values and locations will be correct, though registers have no address, so a **`print &i`** command may print a warning.
  - Most variables and arguments are undefined during function prologues and epilogues, though a **`stop in main`** command will usually stop the program after the prologue.
- Shows inline functions in stack traces, identified by the inline keyword. Note the following:
  - Only the function at the top of the stack and functions that make regular (non-inline) calls show instruction pointers, because other functions share a hardware-defined stack frame with the inline functions that they called.
  - The `return` instruction will only return control to a function that made a non-inline call using a `call` instruction, because inline calls have no defined return address, particularly when their instructions are mixed with those of other functions.
  - The **`up`**, **`down`**, and **`func`** commands work as usual.
- Allows you to set breakpoints in inlined functions.

The following limitation exists:

Optimization often prevents instructions for a source line being generated in consecutive locations. This is particularly true for the Intel® Itanium® architecture. When stepping through such code, the program will tend not to stop at each source line in turn, but rather it will stop each time a change in source line occurs.

Support Limitations

This section describes the limitations on support for the following languages:

- C++
- Fortran

Limitations on Support for C++

The debugger interprets C++ names and expressions using the ANSI standard rules. C++ is a distinct language, rather than a superset of C. Where the semantics of C and C++ differ, the debugger provides the interpretation appropriate for the language of the program being debugged.

To make the debugger more useful, it relaxes some standard C++ name visibility rules. For example, you can reference public, protected, and private class members.

The following limitations apply to debugging a C++ program:

- If a program is not compiled to include debug information, do not set a breakpoint on an inline member function; it may confuse the debugger.
- When you use the debugger to display virtual and inherited class information, the debugger does not support pointers to members of a class.
- The debugger does not support calling the C++ constructs `new` and `delete`. As alternatives, use the `malloc()` and `free()` routines from C.
- Sometimes the debugger does not see class type names with internal linkage, and it issues an error message stating that the name is overloaded.
- You cannot select a version of an overloaded function that has a type signature containing ellipsis points (...).
- Pointers to functions with type signatures that contain parameter list or ellipsis arguments are not supported.
- The debugger does not support consideration of namespace using directives when performing name resolution.

Limitations for debugging templates include the following:

- You cannot specify a template by name in a debugger command. You must use the name of an instantiation of the template.
- Setting a breakpoint at a line number that is inside a template function will not necessarily stop at all instantiations of the function within the given file, but only at a randomly chosen few.

Limitations on Support for Fortran

The debugger and the operating system support the Fortran language with certain limitations. For example, you should be aware of the following data-type limitations when you debug a Fortran program:

- The debugger does not allow setting a breakpoint on a program routine named MAIN.
- Substring notation is not supported.

Following are the limitations on Fortran procedure invocations:

- The debugger does not support invocations of user-defined procedures unless they have been compiled with debug information.
- The debugger does not support complex or real*16 procedure return values.
- The debugger does not support real*16 or complex*32 procedure arguments.

The following limitations apply only to Fortran 90:

- Fortran 90 array constructors, structure constructors, adjustable arrays, and vector subscripts are not supported.
- Fortran 90 user-defined (derived) operators are not supported.
- The debugger does not handle variables of 16-bit character data types.

Part III. Advanced Topics

Advanced Topics

This section provides information to make advanced use of the debugger.

It is intended to be used as a reference and therefore contains a wide variation in the level of information provided in each of the following topics:

- Preparing Your Program for Debugging
- Debugger's Command Processing Structure
- Debugging Core Files
- Machine-Level Debugging
- Debugging Parallel Applications

Preparing Your Program for Debugging. Advanced Debugging. Overview

This section describes how you can modify your program to wait for the debugger.

Modifying Your Program to Wait for the Debugger

To modify your program to wait for the debugger, complete the following steps:

1. Add some code similar to the following :

```
void loopForDebugger()
{
    if (!getenv("loopForDebugger")) return;
    static volatile int debuggerPresent = 0;
    while (!debuggerPresent);
}
```

1. Call this function at some convenient point.
2. Use the debugger's **`attach`** capability to get the process under control.

When you have the process under debugger control, you can assign a non-zero value to debuggerPresent, and continue your program. For example:

```
% setenv loopForDebugger ""
% a.out arg1 arg2 &
% idb -pid 1709 a.out
^C
```

```
(idb) assign debuggerPresent = 1
...
(idb) cont
```

Debugger's Command Processing Structure. Advanced Debugging. Overview

This chapter is divided into the following sections:

- Lexical elements of commands
- Grammar of commands

Lexical Elements of Commands. Overview

This section describes the following topics:

- Lexical elements
- Lexical states
- Identifiers
- Embedded keywords
- Leading keywords
- Reserved identifiers
- Lexemes shared by all languages
- Lexemes that are represented differently in each language

Lexical Elements of Commands

After the debugger assembles complete lines of input from the characters it reads from `stdin` or from the file specified in the `source` command (as described in Entering and Editing Command Lines), each line is then converted into a sequence of lexical elements known as lexemes. For example, **`print(a++);`** is converted into the following lexemes:

1. `print`
2. `(`
3. `a`
4. `++`
5. `)`
6. `;`

The sequence of lexemes is then recognized as debugger commands containing language-specific expressions by a process known as parsing. The recognition is based on a set of rules known as a grammar. The debugger uses a single grammar for commands regardless of the current program's language, but it has language-specific subgrammars for some of the pieces of commands, such as names, expressions, and so on.

Lexical States

The debugger starts processing each line according to the rules of the LKEYWORD lexical state. It recognizes the longest lexeme it can, according to these rules. After recognizing the lexeme, it may stay in the same state, or it may change to a different lexical state with different rules.

The debugger moves between the following lexical states as it recognizes the lexemes:

Lexical State	**Description**
LBPT	Breakpoint commands have lexemes that change the lexical state to LBPT.
LEXPORT	The command **`export`** changes the lexical state to LEXPORT. This state recognizes an environment variable identifier.
LFILE	Commands that require file names have lexemes that change the lexical state to LFILE so that things like `mysrc/foo.txt` are recognized as a file name, and not as a variable `mysrc` being divided by a structure data member `foo.txt`.
LFINT	Commands that require either a file name or a process ID have lexemes to change the lexical state to LFINT.
LKEYWORD	All but one of the debugger commands begin with one or more keywords. The exception is the `examine` command. The debugger recognizes the '{' and ';' lexemes that must precede a command, and uses those to reset to the LKEYWORD state for the beginning of the next command.
LLINE	Commands that require arbitrary character input have lexemes that change the lexical state to LLINE.
LNORM	Language expressions involve language-specific lexemes. The lexemes that precede an expression change the lexical state to LNORM, and then LNORM recognizes the language-specific lexemes.
LSETENV	The command **`setenv`** changes the lexical state to LSETENV. This state recognizes an environment variable identifier.
LSIGNAL	Commands that require signal names have lexemes that change the lexical state to LSIGNAL.
LWORD	Commands that require shell words have lexemes that change the lexical state to LWORD.

The rules that each lexical state uses are described in the following sections, in a format known as a lex regular expression. Rather than repeating some of descriptions, a set of common subrules is described first. After the common subrules, we describe the initial state, the rules, and for each recognized token, the next lexical state.

Identifiers

All languages have a concept of an identifier, composed of letters, digits, and other special characters. The debugger also uses keywords composed of letters; therefore, rules are required to determine which identifiers are actually debugger keywords.

All debugger commands, except **`examine`**, begin with leading keywords. Because the **`examine`** command begins with an expression, all identifiers must be recognized as such from both the LKEYWORD state that starts commands and the LNORM state that the debugger uses for processing expressions.

Some debugger commands have keywords embedded in them following expressions, and the ends of expressions are hard to recognize. You can use identifiers that have the same spelling as an embedded keyword simply by enclosing the whole expression in parentheses (`()`). For more information on using keywords within commands, see Keywords Within Commands.

Furthermore, the C and C++ grammars need to know whether an identifier is a `typedef` or `struct/class` identifier. The debugger currently makes the determination at the time the whole command is parsed, rather than deferring the determination to when the expression itself is being evaluated. This can result in a misidentification of the identifier.

When in the following four lexical states, the debugger can recognize identifiers:

- LKEYWORD, LNORM, LBPT

Language	**Regular Expression**
C, C++, Fortran	`{LT}({LT}|{DG})*`

The state is changed to LNORM to process the rest of the expression.

- LSIGNAL

Language	**Regular Expression**
All	`{LT}({LT}|{DG})*`

The state is left as LSIGNAL to process the next signal.

If your operating system supports internationalization (I18N), `{LT}` equates to `{LTI18N}`.

Embedded Keywords

The complete set of embedded keywords follows:

Lexeme	**Representation**	**Initial Lexical State**	**Changed Lexical State**	**Language Specific?**

ANY	**any**	LNORM	LNORM	Shared by all
AT	**at**	LBPT, LNORM	LNORM	Shared by all
CHANGED	**changed**	LNORM	LNORM	Shared by all
IF	**if**	LBPT	LNORM	Shared by all
IN	**in**	LBPT, LNORM	LNORM	Shared by all
IN_ALL	**in**{Whitespace}**all** {Whitespace}	LBPT, LNORM	LNORM	Shared by all
POLICY	**policy**	LNORM	LNORM	Shared by all
PRIORITY	**priority**	LNORM	LNORM	Shared by all
READ	**read**	LNORM	LNORM	Shared by all
THREAD	**thread**	LNORM	LNORM	Shared by all
THREAD_ALL	**thread**{Whitespace}**all** **thread**{Whitespace} *****	LNORM	LNORM	Shared by all
TO	**to**	LNORM	LNORM	Shared by all
WITH_STATE	**with**{Whitespace}**state**	LNORM	LNORM	Shared by all
WITHIN	**within**	LNORM	LNORM	Shared by all
WRITE	**write**	LNORM	LNORM	Shared by all

NOTE:

THREAD is both a leading and an embedded keyword.

Leading Keywords

Leading keywords are recognized only at the beginning of commands. You do not need to use parentheses (()) to use them as a normal identifier, unless they occur at the start of an **examine** command.

Leading keywords may differ between languages. The complete set follows:

Lexeme	Representation (Some May Be Language Specific)	Initial Lexical State	Changed Lexical State	Language Specific?
ALIAS	**alias**	LKEYWORD	LNORM	Shared by all
ASSIGN	**assign**	LKEYWORD	LNORM	Shared by all

ATTACH	**attach**	LKEYWORD	LNORM	Shared by all
CALL	**call**	LKEYWORD	LNORM	Shared by all
CATCH	**catch**	LKEYWORD	LSIGNAL	Shared by all
CATCH_UNALIGN	**catch**{Whitespace}**unaligned**	LKEYWORD	LNORM	Shared by all
CLASS	**class**	LKEYWORD	LNORM	C++ Special Case
CLONE_SNAPSHOT	**clone**{Whitespace}**snapshot**	LKEYWORD	LNORM	Shared by all
CONDITION	**condition**	LKEYWORD	LNORM	Shared by all
CONT	**cont**	LKEYWORD	LNORM	Shared by all
CONTI	**conti**	LKEYWORD	LNORM	Shared by all
CONT_THREAD	**cont**{Whitespace}**thread**	LKEYWORD	LNORM	Shared by all
DELETE	**delete**	LKEYWORD	LNORM	Shared by all, also used for C++ special case
DELETE_ALL	**delete**{Whitespace}***** **delete**{Whitespace}**all**	LKEYWORD	LNORM	Shared by all
DELSHAREDOBJ	**delsharedobj**	LKEYWORD	LFILE	Shared by all

DETACH	**detach**	LKEYWORD	LNORM	Shared by all
DETACH_REMOTE	**detach**{Whitespace}**remote**	LKEYWORD	LNORM	Shared by all
DISABLE	**disable**	LKEYWORD	LNORM	Shared by all
DISABLE_ALL	**disable**{Whitespace}***** **disable**{Whitespace}**all**	LKEYWORD	LNORM	Shared by all
DISCONNECT_REMOTE	**detach**{Whitespace}**remote**	LKEYWORD	LNORM	Shared by all
DOWN	**down**	LKEYWORD	LNORM	Shared by all
DUMP	**dump**	LKEYWORD	LNORM	Shared by all
EDIT	**edit**	LKEYWORD	LFILE	Shared by all
ELSE	**else**	LKEYWORD	LKEYWORD	Shared by all
ENABLE	**enable**	LKEYWORD	LNORM	Shared by all
ENABLE_ALL	**enable**{Whitespace}***** **enable**{Whitespace}**all**	LKEYWORD	LNORM	Shared by all
EXPAND_AGGREGATED_MESSAGE	**expand**{Whitespace}**aggregated**{Whitespace}**message**	LKEYWORD	LNORM	Shared by all
EXPORT	**export**	LKEYWORD	LNORM	Shared by all
FILECMD	**file**	LKEYWORD	LFILE	Shared by all

FILEEXPR	**fileexpr**	LKEYWORD	LFILE	Shared by all
FOCUS	**focus**	LKEYWORD	LNORM	Shared by all
FOCUS_ALL	**focus**{Whitespace}***** **focus**{Whitespace}**all**	LKEYWORD	LNORM	Shared by all
FUNC	**func**	LKEYWORD	LNORM	Shared by all
GOTO	**goto**	LKEYWORD	LNORM	Shared by all
HELP	**help**	LKEYWORD	LLINE	Shared by all
HISTORY	**history**	LKEYWORD	LNORM	Shared by all
HPFGET	**hpfget**	LKEYWORD	LNORM	Fortran
IF	**if**	LKEYWORD	LNORM	Shared by all
IGNORE	**ignore**	LKEYWORD	LSIGNAL	Shared by all
IGNORE_UNALIGN	**ignore**{Whitespace}**unaligned**	LKEYWORD	LNORM	Shared by all
INFO	**info**	LKEYWORD	LKEYWORD	Shared by all
INPUT	**input**	LKEYWORD	LFILE	Shared by all
IO	**io**	LKEYWORD	LFILE	Shared by all

KILL	**kill**	LKEYWORD	LNORM	Shared by all
KPS	**kps**	LKEYWORD	LNORM	Shared by all
LIST	**list**	LKEYWORD	LNORM	Shared by all
LISTOBJ	**listobj**	LKEYWORD	LNORM	Shared by all
LOAD	**load**	LKEYWORD	LFILE	Shared by all
MAP_SOURCE_DIRECTORY	**map**{Whitespace}**source**{Whitespace}**directory**	LKEYWORD	LNORM	Shared by all
NEXT	**next**	LKEYWORD	LNORM	Shared by all
NEXTI	**nexti**	LKEYWORD	LNORM	Shared by all
OUTPUT	**output**	LKEYWORD	LFILE	Shared by all
PATCH	**patch**	LKEYWORD	LNORM	Shared by all
PLAYBACK	**playback**	LKEYWORD	LKEYWORD	Shared by all
POP	**pop**	LKEYWORD	LNORM	Shared by all
PRINT	**print**	LKEYWORD	LNORM	Shared by all
PRINTB	**printb**	LKEYWORD	LNORM	Shared by all

PRINTD	**printd**	LKEYWORD	LNORM	Shared by all
PRINTENV	**printenv**	LKEYWORD	LNORM	Shared by all
PRINTF	**printf**	LKEYWORD	LNORM	Shared by all
PRINTI	**printi**	LKEYWORD	LNORM	Shared by all
PRINTO	**printo**	LKEYWORD	LNORM	Shared by all
PRINTT	**printt**	LKEYWORD	LNORM	Shared by all
PRINTX	**printx**	LKEYWORD	LNORM	Shared by all
PRINTREGS	**printregs**	LKEYWORD	LNORM	Shared by all
PROCESS	**process**	LKEYWORD	LNORM	Shared by all
PROCESS_ALL	**process**{Whitespace}***** **process**{Whitespace}**all**	LKEYWORD	LNORM	Shared by all
QUESTION	**?**	LKEYWORD	LNORM	Shared by all
QUIT	**quit**	LKEYWORD	LNORM	Shared by all
READSHAREDOBJ	**readsharedobj**	LKEYWORD	LFILE	Shared by all
RECORD	**record**	LKEYWORD	LKEYWORD	Shared by all

RERUN	**rerun**	LKEYWORD	LWORD	Shared by all
RETURN	**return**	LKEYWORD	LNORM	Shared by all
RUN	**run**	LKEYWORD	LWORD	Shared by all
SAVE_SNAPSHOT	**save**{Whitespace}**snapshot**	LKEYWORD	LNORM	Shared by all
SET	**set**	LKEYWORD	LNORM	Shared by all
SETENV	**setenv**	LKEYWORD	LNORM	Shared by all
SH	**sh**	LKEYWORD	LNORM	Shared by all
SHOW	**show**	LKEYWORD	LKEYWORD	Shared by all
SHOW_AGGREGATED_MESSAGE	**show**{Whitespace}**aggregated**{Whitespace}**message**	LKEYWORD	LNORM	Shared by all
SHOW_AGGREGATED_MESSAGE_ALL	**show**{Whitespace}**aggregated**{Whitespace}**message**{Whitespace}***** **show**{Whitespace}**aggregated**{Whitespace}**message**{Whitespace}**all**	LKEYWORD	LNORM	Shared by all
SHOW_PROCESS_SET	**show**{Whitespace}**process**{Whitespace}**set**	LKEYWORD	LNORM	Shared by all
SHOW_PROCESS_SET_ALL	**show**{Whitespace}**process**{Whitespace}**set**{Whitespace}***** **show**{Whitespace}**process**{Whitespace}**set**{Whitespace}**all**	LKEYWORD	LNORM	Shared by all
SHOW_SOURCE_DIRECTORY	**show**{Whitespace}**source**{Whitespace}**directory**	LKEYWORD	LNORM	Shared by all

SHOW_ALL_SOURCE_DIRECTORY	**show**{Whitespace}**all**{Whitespace}**source**{Whitespace}**directory**	LKEYWORD	LNORM	Shared by all
SLASH	/	LKEYWORD	LNORM	Shared by all
SNAPSHOT	**snapshot**	LKEYWORD	LNORM	Shared by all
SNAPSHOT_ALL	**snapshot all**	LKEYWORD	LNORM	Shared by all
SNAPSHOT_*	**snapshot ***	LKEYWORD	LNORM	Shared by all
SOURCE	**source**	LKEYWORD	LFILE	Shared by all
STATUS	**status**	LKEYWORD	LNORM	Shared by all
STEP	**step**	LKEYWORD	LNORM	Shared by all
STEPI	**stepi**	LKEYWORD	LNORM	Shared by all
STOP	**stop**	LKEYWORD	LBPT	Shared by all
STOPI	**stopi**	LKEYWORD	LNORM	Shared by all
THREAD	**thread**	LKEYWORD	LNORM	Shared by all
TRACE	**trace**	LKEYWORD	LNORM	Shared by all
TRACEI	**tracei**	LKEYWORD	LNORM	Shared by all

UNALIAS	**unalias**	LKEYWORD	LNORM	Shared by all
UNLOAD	**unload**	LKEYWORD	LNORM	Shared by all
UNMAP_SOURCE_DIRECTORY	**unmap**{Whitespace}**source**{Whitespace}**directory**	LKEYWORD	LNORM	Shared by all
UNRECORD	**unrecord**	LKEYWORD	LNORM	Shared by all
UNSET	**unset**	LKEYWORD	LNORM	Shared by all
UNSETENV	**unsetenv**	LKEYWORD	LNORM	Shared by all
UNSETENV_ALL	**unsetenv**{Whitespace}*****	LKEYWORD	LNORM	Shared by all
UNUSE	**unuse**	LKEYWORD	LFILE	Shared by all
UP	**up**	LKEYWORD	LNORM	Shared by all
USE	**use**	LKEYWORD	LFILE	Shared by all
VERSION	**version**	LKEYWORD	LNORM	Shared by all
WATCH	**watch**	LKEYWORD	LNORM	Shared by all
WATCH_MEMORY	**watch**{Whitespace}**memory**	LKEYWORD	LNORM	Shared by all
WATCH_VARIABLE	**watch**{Whitespace}**variable**	LKEYWORD	LNORM	Shared by all

WHATIS	**whatis**	LKEYWORD	LNORM	Shared by all
WHEN	**when**	LKEYWORD	LBPTChapter	Shared by all
WHENI	**wheni**	LKEYWORD	LNORM	Shared by all
WHERE	**where**	LKEYWORD	LNORM	Shared by all
WHEREIS	**whereis**	LKEYWORD	LNORM	Shared by all
WHERE_THREAD	**where**{Whitespace}**thread**	LKEYWORD	LNORM	Shared by all
WHERE_THREAD_ALL	**where**{Whitespace}**thread**{Whitespace}***** **where**{Whitespace}**thread**{Whitespace}**all**	LKEYWORD	LNORM	Shared by all
WHICH	**which**	LKEYWORD	LNORM	Shared by all
WHILE	**while**	LKEYWORD	LNORM	Shared by all

Reserved Identifiers

Some identifiers are recognized as reserved words, regardless of whether they are inside parentheses (`()`).

The reserved words may differ between languages. The complete set follows:

Lexeme	Representation (Some May Be Language Specific)	Initial Lexical State	Changed Lexical	Language Specific?
CHAR	char	LNORM	LNORM	C, C++
CLASS	class	LNORM	LNORM	C++
CONST	const	LNORM	LNORM	C, C++
DELETE	delete	LNORM	LNORM	C++

DOUBLE	double	LNORM	LNORM	C, C++
ENUM	enum	LNORM	LNORM	C, C++
FLOAT	float	LNORM	LNORM	C, C++
INT	int	LNORM	LNORM	C, C++
LONG	long	LNORM	LNORM	C, C++
NEW	new	LNORM	LNORM	C++
OPERATOR	operator	LNORM	LNORM	C++
SHORT	short	LNORM	LNORM	C, C++
SIGNED	signed	LNORM	LNORM	C, C++
SIZEOF	sizeof	LNORM	LNORM	C, C++, Fortran
STRUCT	struct	LNORM	LNORM	C, C++
UNION	union	LNORM	LNORM	C, C++
UNSIGNED	unsigned	LNORM	LNORM	C, C++
VOID	void	LNORM	LNORM	C, C++
VOLATILE	volatile	LNORM	LNORM	C, C++

About lexemes

Because the debugger supports multiple languages, some of the rules must be language specific. To distinguish between the characters used for a particular language to represent a lexeme and the lexeme itself, the debugger names the lexemes, rather than using any one language's representation. For example, the lexeme GE corresponds to Fortran's '.GE.', and to C's '>='.

This section contains information about lexemes that have the same representation in all languages, especially those that form part of the debugger commands apart from the language-specific expressions.

There is also information about lexemes that are represented differently in each language and specific to C++, C and C++, Fortran.

Lexemes Shared by All Languages

This section gives information about

- Common elements of lexemes
- Whitespace and command separating lexemes
- LNORM lexemes
- LBPT lexemes

- LFILE lexemes
- LKEYWORD lexemes
- LLINE lexemes
- LWORD lexemes
- LSIGNAL lexemes
- LSETENV and LEXPORT lexemes

Common Elements of Lexemes

The following tables list common elements of lexemes.

Concept	Rule	Representation	Description
Decimal digit	DG	`[0-9]`	One character from '0'..'9'.
Octal digit	OC	`[0-7]`	One character from '0'..'7'.
Hexadecimal digit	HX	`[0-9a-fA-F]`	Any of the characters '0'..'9' and any of the letters 'A'..'F' and 'a'..'f'.
Single letter	LT	`[A-Za-z_$]`	Any of the characters 'A'..'Z', 'a'..'z', and the underscore (_) and dollar sign ($) characters.
Single letter from the International Character Set	LT18N	`[A-Za-z_$\200-\377]`	Any of the characters 'A'..'Z', 'a'..'z', the underscore (_) and dollar sign ($) characters, and any character in the top half of the 8-bit character set.
Shell 'word'	WD	`[^ \t;\n'"]`	Any character except space, tab, semicolon (;), linefeed, less than (<), greater than (>), and quotes (' or ").
File name	FL	`[^ \t\n\}\;\>\<]`	Any character except space, tab, semicolon (;), linefeed, right brace (}), less than (<), greater

			than (>), and tick (`).
Optional exponent	Exponent	`[eE][+-]?{DG}+`	Numbers often allow an optional exponent. It is represented as an 'e' or 'E' followed by an optional plus (+) or minus (-), and then one or more decimal digits.
Whitespace	Whitespace	`[ \t]+`	Whitespace is often used to separate two lexemes that would otherwise be misconstrued as a single lexeme. For example, `stop in` is two keywords, but `stopin` is an identifier. Apart from this separating property, Whitespace is usually ignored. Whitespace is a sequence of one or more tabs or spaces.
String literal	stringChar	`([^"\\\n]|([\\]({simpleEscape}|{octalEscape}|{hexEscape})))`	Any character except the terminating quote character ("), or a newline (\n). If the character is a backslash (\), it is followed by an escaped sequence of characters.
Character literal	charChar	`([^'\\\n]|([\\]({simpleEscape}|{octalEscape}|{hexEscape})))`	Any character except the terminating quote (') character, or a newline (\n). If the character is a backslash (\), it is followed by an escaped sequence of characters.
Environment variable identifier	EID	`[^ \t\n;='"&\\|]`	Any character except space, tab, linefeed, less-than (<), greater-than (>), semicolon (;), equal sign (=), quotes (' or "), ampersand (&), backslash (\), and bar (|).
Universal character	UCN	`\\u{HX}{4}|\\U{HX}{8}`	A universal character name is a backslash (\)

name			followed by either a lowercase 'u' and 4 hexadecimal digits, or an uppercase 'U' and 8 hexadecimal digits.

The escaped sequence of characters can be one of following three forms:

Concept	**Rule**	**Representation**	**Description**
Simple escape	simpleEscape	`([A-Za-z'"?#*\\])`	One of 'A'-'Z' or 'a'-'z'. Some of these have special meanings, the most common being 'n' for newline and 't' for tab. Can be a quote (' or ") character that does not finish the literal, a question mark (?), a pound sign (#), an asterisk (*), or a backslash (\), which then becomes part of the string literal rather than causing a further escape sequence.
Octal escape	octalEscape	`(OC{1,3})`	1 to 3 octal digits, the combined numeric value of which is the character that becomes part of the string literal.
Hexadecimal escape	hexEscape	`([xX]HX{1,8})`	An 'x' or an 'X' followed by 1 to 8 hexadecimal digits, the combined numeric value of which is the character that becomes part of the string literal.

Whitespace and Command-Separating Lexemes

In all lexical states, unescaped newlines produce the NEWLINE token and change the lexical state to be LKEYWORD.

Initial State:	**LKEYWORD, LNORM, LFILE, LLINE, LWORD, LSIGNAL, LBPT**
Regular Expression:	`[\n]`
Lexeme:	NEWLINE
Change to State:	LKEYWORD

In all lexical states except LLINE, a semicolon also changes the lexical state to be LKEYWORD.

This is because SEMICOLON is the command separator.

Initial State:	**LKEYWORD, LNORM, LFILE, LSIGNAL, LBPT, LWORD**
Regular Expression:	`";"`
Lexeme:	SEMICOLON
Change to State:	LKEYWORD

Commands can be nested, and the following transitions support this:

Initial State:	**LNORM**
Regular Expression:	`"{"`
Lexeme:	LBRACE
Change to State:	LKEYWORD

Initial State:	**LKEYWORD, LNORM, LFILE, LSIGNAL, LBPT**
Regular Expression:	`"}"`
Lexeme:	RBRACE
Change to State:	LKEYWORD

In most lexical states, the spaces, tabs, and escaped newlines are ignored. In the LLINE state, the spaces and tabs are part of the line, but escaped newlines are still ignored. In the LWORD state, the spaces and tabs are ignored, but escaped newlines are not.

Initial State:	**LKEYWORD, LNORM, LFILE, LSIGNAL, LBPT**

Regular Expression:	`[ \t]`
	`\\\n`

Lexeme:	Ignored
Change to State:	Unchanged

Initial State:	**LLINE**
Regular Expression:	`\\\n`
Lexeme:	Ignored
Change to State:	Unchanged

Initial State:	**LWORD**
Regular Expression:	`[ \t]`
Lexeme:	Ignored
Change to State:	Unchanged

LNORM Lexemes

The state stays in LNORM.

Lexeme	Regular Expression
ANY	any
AT	at
ATSIGN	"@"
CHANGED	changed
CHARACTERconstant	[lL]['']{charChar}+['']
COLON	":"
COMMA	","
DOLLAR	"$"
DOT	"."
GE	">="
GREATER	">"
HASH	*unknown*
IF	if
IN	in
IN_ALL	in{Whitespace}all{Whitespace}
LE	"<="
LESS	"<"
LPAREN	"("
POLICY	policy
PRIORITY	priority
RPAREN	")"
READ	read
SLASH	"/"
STAR	"*"
STATE	state
STRINGliteral	["]{stringChar}*["]
THREAD	thread
THREAD_ALL	thread{Whitespace}all thread{Whitespace}"*"
TICK	"`"
TO	to
WIDECHARACTERconstant	[lL]['']{charChar}+['']
WIDESTRINGliteral	[lL]["]{stringChar}*["]
WITH	with
WITHIN	within
WRITE	write

LBPT Lexemes

Lexeme	Regular Expression	Initial Lexical State	Changed Lexical State
IN	in	LBPT	LNORM
IN_ALL	in{Whitespace}all	LBPT	LNORM
AT	at	LBPT	LNORM
PC_IS	pc	LBPT	LNORM
SIGNAL	signal	LBPT	LNORM
UNALIGNED	unaligned	LBPT	LNORM
VARIABLE	variable	LBPT	LNORM
MEMORY	memory	LBPT	LNORM
EVERY_INSTRUCTION	every{Whitespace}instruction	LBPT	LNORM
EVERY_PROC_ENTRY	every{Whitespace}proc[edure]{Whitespace}entry	LBPT	LNORM
QUIET	quiet	LBPT	LBPT

LFILE Lexemes

Files are one or more characters that can appear in a file name.

The state is left as LFILE, so that commands such as **use** and **unuse** can have lists of files.

Lexeme	Regular Expression
FILENAME	{FL}+

LKEYWORD Lexemes

The state remains in LKEYWORD.

Lexeme	Regular Expression
INTEGERconstant	"0"{OC}+ "0"[xX]{HX}+ {DG}+

LLINE Lexemes

All characters up to the next newline are assembled into a STRING literal.

LWORD Lexemes

Once the lexical state has been set to LWORD, it will stay there until a NEWLINE or a SEMICOLON is found. Both of these cause the lexical state to become LKEYWORD again.

The individual words recognized can be any of the following, but in each case, the state stays LWORD:

Lexeme	Regular Expression
GREATER	`">"`
LESS	`"<"`
GREATERAMPERSAND	`">&"`
ONEGREATER	`"1>"`
TWOGREATER	`"2>"`

STRINGliteral	`[']{charChar}*[']`
	`["]{stringChar}*["]`

STRINGliteral	`{WD}* that does not end in a backslash`
WIDECHARACTERconstant	`[lL]['] {charChar}+[']`
WIDESTRINGliteral	`[lL]["] {stringChar}*["]`

LSIGNAL Lexemes

The state stays in LSIGNAL.

Lexeme	Regular Expression
INTEGERconstant	`{DG}+`
IDENTIFIER	`{LT}({LT}|{DG})*`

LSETENV and LEXPORT Lexemes

Lexeme	Regular Expression
ENVARID	`{EID}+`

Lexemes That Are Represented Differently in Each Language

The following Table gives lexemes that are represented differently in each language

Lexeme	Representation (Some May Be Language Specific)	Initial Lexical State	Changed Lexical State	Language Specific?
AMPERSAND	`"&"`	`LNORM`	Unchanged	C, C++, Fortran
ANDAND	`"&&"`	`LNORM`	Unchanged	C, C++
ANDassign	`"&="`	`LNORM`	Unchanged	C, C++

ARROW	`"->"`	LNORM	Unchanged	C, C++
ARROWstar	`"->*"`	LNORM	Unchanged	C++
ASSIGNOP	`"="`	LNORM	Unchanged	C, C++, Fortran
BRACKETS	`"[]"`	LNORM	Unchanged	C, C++
CLCL	`"::"`	LNORM	Unchanged	C++
DECR	`"--"`	LNORM	Unchanged	C, C++
DIVassign	`"/="`	LNORM	Unchanged	C, C++
DOTstar	`".*"`	LNORM	Unchanged	C++
ELLIPSIS	`"..."`	LNORM	Unchanged	C++

EQ	`"=="`
	`".EQ."`
	`(IS[ \t]+)?`
	`("="|("EQUAL"([ \t]+"TO")?))`
		LNORM	Unchanged	C, C++,
				Fortran
				Fortran

ERassign	`"^="`	LNORM	Unchanged	C, C++

GE	`".GE."`
	`(IS[ \t]+)?`
	`"NOT"[ \t]+`
	`("(IS[ \t]+)?`
	`(">="|("GREATER"([ \t]+"THAN")?[ \t]`
	`+"OR"[ \t]+"EQUAL"([ \t]+"TO")?))`
		LNORM	Unchanged	Fortran

GREATER	`".GT."`
	`(IS[ \t]+)?`
	`(">"|("GREATER"([ \t]+"THAN")?))`
		LNORM	Unchanged	Fortran

HAT	`"^"`	LNORM	Unchanged	C, C++
INCR	`"++"`	LNORM	Unchanged	C, C++
LBRACKET	`"["`	LNORM	Unchanged	C, C++, Fortran

LE	`".LE."`
	`(IS[ \t]+)?"NOT"[ \t]+`
	`(">"|("GREATER"([ \t]+"THAN")?))`
	`(IS[ \t]+)?`
	`("  "OR"[ \t]+"EQUAL"([ \t]+"TO")?))`
		LNORM	Unchanged	Fortran

LESS	`".LT."`
	`(IS[ \t]+)?`
	`("<"|("LESS"([ \t]+"THAN")?))`
		LNORM	Unchanged	Fortran

LOGAND	`".AND."`	LNORM	Unchanged	Fortran
LOGEQV	`".EQV."`	LNORM	Unchanged	Fortran
LOGNEQV	`".NEQV."`	LNORM	Unchanged	Fortran
LOGNOT	`".NOT."`	LNORM	Unchanged	Fortran
LOGOR	`".OR."`	LNORM	Unchanged	Fortran
LOGXOR	`".XOR."`	LNORM	Unchanged	Fortran
LS	`"<<"`	LNORM	Unchanged	C, C++
LSassign	`"<<="`	LNORM	Unchanged	C, C++
MINUS	`"-"`	LNORM	Unchanged	C, C++, Fortran
MINUSassign	`"-="`	LNORM	Unchanged	C, C++
MOD	`"%"` `MOD`	LNORM	Unchanged	C, C++
MODassign	`"%="`	LNORM	Unchanged	C, C++

MULTassign	`"*="`	LNORM	Unchanged	C, C++

NE	`"!="`
	`".NE."`
	`"/="`
	`(IS[ \t]+)?`
	`  "NOT"[ \t]+("="|("EQUAL"([ \t]+"TO")?))`
		LNORM	Unchanged	C, C++
				Fortran

NOT	`"!"` `NOT`	LNORM	Unchanged	C, C++
OPENSLASH	`"(/"`	LNORM	Unchanged	Fortran
OR	`"|"` `OR`	LNORM	Unchanged	C, C++
OROR	`"||"`	LNORM	Unchanged	C, C++
ORassign	`"|="`	LNORM	Unchanged	C, C++
PARENS	`"()"`	LNORM	Unchanged	C++
PERCENT	`"%"`	LNORM	Unchanged	Fortran
PLUS	`"+"`	LNORM	Unchanged	C, C++, Fortran
PLUSassign	`"+="`	LNORM	Unchanged	C, C++
QUESTION	`"?"`	LNORM	Unchanged	C, C++
RBRACKET	`"]"`	LNORM	Unchanged	C, C++, Fortran
RS	`">>"`	LNORM	Unchanged	C, C++
RSassign	`">>="`	LNORM	Unchanged	C, C++
SLASHCLOSE	`"/)"`	LNORM	Unchanged	Fortran
SLASHSLASH	`"//"`	LNORM	Unchanged	Fortran
STARSTAR	`"**"`	LNORM	Unchanged	Fortran
TWIDDLE	`"~"`	LNORM	Unchanged	C, C++

LKEYWORD Lexemes Specific to C++

If a C++ identifier is followed by a "::", it is assumed to be a class or namespace identifier.

If a C++ identifier is followed by a "<", complex and dubious checks are made to try to match a complete template instance specifier.

LNORM Lexemes Specific to C and C++

The lexemes in the following table are specific to C and C++. The state stays in LNORM.

Lexeme	**Representation**	**Language**
AMPERSAND	`"&"`	C, C++
ANDAND	`"&&"`	C, C++
ANDassign	`"&="`	C, C++
ARROW	`"->"`	C, C++
ASSIGNOP	`"="`	C, C++

BRACKETS	"[]"	C, C++
CHAR	char	C, C++
CONST	const	C, C++
DECR	"--"	C, C++
DIVassign	"/="	C, C++
DOUBLE	double	C, C++
ENUM	enum	C, C++
EQ	"=="	C, C++
ERassign	"^="	C, C++
FLOAT	float	C, C++

FLOATINGconstant	{DG}*"."{DG}*
	{DG}*"."{DG}*
	{Whitespace}?{Exponent}
	{DG}+{Whitespace}?{Exponent}
		C, C++

HAT	"^"	C, C++
IDENTIFIER, **TYPEDEFname**	{LT}|{UCN})({LT}|{UCN}|{DG})*	C, C++
INCR	"++"	C, C++
INT	int	C, C++
INTEGERconstant	"0"{OC}+ "0"[xX]{HX}+ {DG}+	C, C++
LBRACKET	"["	C, C++
LONG	long	C, C++
LS	"<<"	C, C++
LSassign	"<<="	C, C++
MINUS	"-"	C, C++
MINUSassign	"-="	C, C++
MOD	"%"	C, C++
MODassign	"%="	C, C++
MULTassign	"*="	C, C++
NE	"!="	C, C++
NOT	"!"	C, C++
OR	"|"	C, C++
ORassign	"|="	C, C++
OROR	"||"	C, C++
PLUS	"+"	C, C++
PLUSassign	"+="	C, C++
QUESTION	"?"	C, C++

RBRACKET	`"]"`	C, C++
RS	`">>"`	C, C++
RSassign	`">>="`	C, C++
SHORT	`short`	C, C++
SIGNED	`signed`	C, C++
SIZEOF	`sizeof`	C, C++
STRUCT	`struct`	C, C++
TWIDDLE	`"~"`	C, C++
UNION	`union`	C, C++
UNSIGNED	`unsigned`	C, C++
VOID	`void`	C, C++
VOLATILE	`volatile`	C, C++

The lexemes in the following table are specific to C++. The state stays in LNORM.

Lexeme	**Representation**	**Language**
ARROWstar	`"->*"`	C++
CLASS	`class`	C++
CLCL	`"::"`	C++
DELETE	`delete`	C++
DOTstar	`".*"`	C++
ELLIPSIS	`"..."`	C++
NEW	`new`	C++
OPERATOR	`operator`	C++
PARENS	`"()"`	C++
THIS	`this`	C++

LNORM Lexemes Specific to Fortran

The lexemes in the following table are specific to Fortran. The state stays in LNORM.

Lexeme	**Representation**
AMPERSAND	`"&"`
ASSIGNOP	`"="`

CHARACTERconstantWithKind	`{FortranString}`
	`{FortranCharacterKind}{FortranString}`
	`{FortranCharacterKind}?{FortranCString}`

CharDoubleDelim	`[^"\\\n]|("")`
CharSingleDelim	`[^'\\\n]|('')`

CStringDoubleDelim	["]({CharDoubleDelim}|{FortranEscapeChar})*["]
CStringSingleDelim	[']({CharSingleDelim}|{FortranEscapeChar})*[']
ExponentVal	[+-]?{DG}+
EQ	".EQ."
EQ	"=="
FortranBinaryValue	[Bb]((['][01]+['])|(["][01]+["]))
FortranCharacterKind	{FortranCharacterNamedKind} {FortranCharacterNumericKind}
FortranCharacterNamedKind	{FortranName}"_"
FortranCharacterNumericKind	{DG}+"_"
FortranCString	({CStringSingleDelim}|{CStringDoubleDelim})[Cc]
FortranEscapeChar	[\\]([AaBbFfNnRrTtVv]|{FortranOctalEscape} |{FortranHexEscape}|0|[\\])
FortranHexEscape	[Xx]{HX}{1,2}
FortranHexValue	[Zz]((['][{HX}+['])|(["]{HX}+["]))
FortranHexValueAlternative	((['][{HX}+['])|(["]{HX}+["]))[Xx]
FortranKind	{FortranNamedKind} {FortranNumericKind}
FortranName	[A-Za-z$]({LT}|{DG})*
FortranNamedKind	"_"{FortranName}
FortranNumericKind	"_"{DG}+
FortranOctalEscape	{OC}{1,3}
FortranOctalValue	[Oo]((['][{OC}+['])|(["]{OC}+["]))
FortranOctalValueAlternative	((['][{OC}+['])|(["]{OC}+["]))[Oo]
FortranString	{StringSingleDelim} {StringDoubleDelim}
GE	".GE."
GREATER	".GT."
IDENTIFIER, TYPEDEFname	{LT}({LT}|{DG})*
INTEGERconstant	".TRUE." ".FALSE." "0"{OC}+ "0X"{HX}+ {DG}+

INTEGERconstantWithKind	{DG}+{FortranKind}
	{DG}*"#"[0-9A-Za-z]+
	{FortranBinaryValue}
	{FortranOctalValue}
	{FortranHexValue}
	{FortranOctalValueAlternative}
	{FortranHexValueAlternative}

LBRACKET	"["
LE	".LE."
LESS	".LT."
LOGAND	".AND."
LOGEQV	".EQV."

LOGICALconstantWithKind	".TRUE."{FortranKind}? ".FALSE."{FortranKind}?
LOGNEQV	".NEQV."
LOGNOT	".NOT."
LOGOR	".OR."
LOGXOR	".XOR."
MINUS	"-"
NE	".NE."
NE	"/="
OPENSLASH	"(/"
PERCENT	"%"
PERCENT_PERCENT	"%%"
PLUS	"+"
RBRACKET	"]"
RealConstant	{RealSingleConstant} {RealDoubleConstant} {RealQuadConstant}
REALconstantWithKind	RealConstant
RealDExponent	[Dd]{ExponentVal}
RealDoubleConstant	({DG}+|{RealWithDecimal}){RealDExponent}
RealEExponent	[Ee]{ExponentVal}
RealQExponent	[Qq]{ExponentVal}
RealQuadConstant	({DG}+|{RealWithDecimal}){RealQExponent}
RealSingleConstant	(({DG}+{RealEExponent}) ({RealWithDecimal}{RealEExponent}?)){FortranKind}?
RealWithDecimal	({DG>}+"."DG}*) ({DG}*"."{DG}+)
SIZEOF	sizeof
SLASHCLOSE	"/)"
SLASHSLASH	"//"
STARSTAR	"**"
StringDoubleDelim	["](({CharDoubleDelim}|[\\])*["]
StringSingleDelim	['](({CharSingleDelim}|[\\])*[']

Grammar of Commands

Most of the grammar for commands has already been given in previous sections. This section concentrates on the grammar for expressions:

- Names and expressions within commands
- Expressions specific to C
- Expressions specific to C++

- Expressions specific to Fortran

Names and Expressions Within Commands

The exact syntax of expressions is specific to the current language.

Often you can omit an expression from a command or use a convenient default instead, to change the meaning of a command.

```
expression

        : expression for C

        | expression for C++

        | expression for Fortran

expression-opt

        : [ expression ]
```

Identifiers, Keyword, and Typedef Names

The debugger uses the normal language lookup rules for identifiers, (obeying scopes, and so on,) but also extends those rules as follows:

- All global variables are visible.
- If the debugger cannot find the identifier within the current lexical scopes, it will successively search the lexical scopes of each of the first `$framesearchlimit` (default is 0) callers.

These rules can be subverted by rescoping the name.

NOTE:

> The debugger does not know where in the scope a declaration occurred, so all lookups consider all identifiers in the scope, whether or not they occurred before the current line.

The lexical tokens for identifiers are specific to the current language, and also to the current lexical state.

```
IDENTIFIER

        : identifier for LSIGNAL lexical state

        | identifier for C
        | identifier for C++

        | identifier for Fortran
```

`TYPEDEFname`s are lexically just identifiers, but when looking them up in the current scope, the debugger determines that they refer to types, such as `TYPEDEF`s, classes, or `struct`s. This information is needed to correctly parse C and C++ expressions.

```
TYPEDEFname

        : IDENTIFIER
```

A few lexical tokens act as embedded keywords in some positions within expressions, but the debugger generally tries to accept them as though they were normal identifiers.

```
identifier-or-key-word

        : IDENTIFIER

        | embedded-key-word

embedded-key-word

        : ANY

        | CHANGED

        | READ

        | WRITE
```

In other contexts, the debugger is also prepared to accept `TYPEDEFname`s (for example, `int` or the name of a `class`).

```
identifier-or-typedef-name

        : identifier-or-typedef-name for C

        | identifier-or-typedef-name for C++

        | identifier-or-typedef-name for Fortran
```

Integer Constants

The lexical tokens for integer constants are specific to the current language.

```
integer_constant

    : INTEGERconstant for C and C++

    | INTEGERconstant for Fortran
```

Macros

The debugger does not currently understand usages of macros, for example, uses of C and C++ preprocessor `#define` macros, and so on.

Calls

You can call any function whose address can be taken, provided that the parameters can also be passed, and the result returned.

```
call-expression

        : call-expression for C

        | call-expression for C++

        | call-expression for Fortran
```

Parameters

Each language may impose its own restrictions on exactly what can be passed as a parameter.

Any expression can be passed 'by value', but C++ constructors and destructors will not be invoked. Evaluating parameters can involve evaluating nested calls.

Anything whose address can be taken can be passed 'by reference'.

The debugger has very limited understanding of array descriptors.

Comma is both the argument separator and a valid operator in C and C++. Hence, argument lists are comma-separated *assignment-expressions* rather than full expressions.

```
argument-expression-list

        : assignment-expression

        | assignment-expression COMMA argument-expression-list

arg-expression-list-opt

        : [ argument-expression-list ]

assignment-expression

        : assignment-expression for C

        | assignment-expression for C++

        | assignment-expression for Fortran
```

Return Results

Any scalar or structure type can be the return result of a called function. Some simple array types are also supported, but the general cases are not.

The C++ constructors and destructors are not invoked, which may cause problems.

Addresses

You can take the addresses of variables and other data that are in memory, and functions that have had code generated for them. You can also take the address of a line of source code.

Some variables may be in registers; you cannot take their addresses.

The optimizing compilers may move variables from one memory location to another, in which case you will obtain the address of the current memory location of the variable.

The optimizing compilers may eliminate unused functions, as well as functions that have had all calls inlined. Static functions in header files may result in multiple copies of the code, and the address will be of only one of those copies.

The optimizing compilers and linkers may skip some instructions on the way in during a call, so a breakpoint on the first few instructions may not be hit. When you set a breakpoint on a function, the debugger sets it deeper in the function, at the end of the entry sequence, to try to avoid this.

The address of a line of source code is the address of the first instruction in memory that came from this line, but this instruction may be branched around, so it might not be executed before any other instruction from the same line.

Address of a Source Line

The debugger has extended the syntax of most languages to allow you to get the address of the first instruction that a source line generates. If you do not specify a file via the *string*, then the current file is used. If you specify a DOLLAR as the *line-number*, then the last line in the file that generated any instructions is used.

```
line-address

         : ATSIGN string COLON line-number

         | ATSIGN line-number



line-number

         : INTEGERconstant
```

```
        | DOLLAR
```

Other Modified Forms of Expressions

The *whatis_command* supports supersets of the normal *expression* syntax of the language.

```
whatis-expressions

        : whatis-expressions for C

        | whatis-expressions for C++

        | whatis-expressions for Fortran
```

Some commands (notably the **examine** command and the **cont** command) have a syntax that inhibits the use of a full expression. In this case, a more limited form of expression is still allowed.

```
address-exp

        : address-exp for C

        | address-exp for C++

        | address-exp for Fortran
```

The **cont** command and the *change_stack_frame_commands* have a form that specifies where to continue to, or where to cut the stack back to.

```
loc

        : loc for C

        | loc for C++

        | loc for Fortran
```

The *target* of a *modifying_command* can only be a subset of the possible expressions, known as a *unary-expression*.

```
unary-expression

        : unary-expression for C

        | unary-expression for C++

        | unary-expression for Fortran
```

Strings

The syntax of strings is sensitive to the current lexical state and language.

```
string

        : LNORM string

        | LLINE string

        | LWORD string
```

Most of the languages have places where they allow a series of string literals to be equivalent to a single string formed of their concatenated characters.

```
string-literal-list

        : string-literal-list for C

        | string-literal-list for C++
```

Rescoped Expressions

Sometimes the normal language visibility rules are not sufficient for specifying the variable, function, and so on, to which you may want to refer. The debugger extends the language's idea of an expression with an additional possibility called a rescoped expression.

Rescoped expressions cause the debugger to look up the identifiers and so on in the *qual-symbol-opt*, as though it were in the source file specified by the preceding *filename-tick* or *qual-symbol-opt*.

```
rescoped-expression

        : filename-tick qual-symbol-opt

        | TICK qual-symbol-opt

rescoped-typedef

        : filename-tick qual-typedef-opt

        | TICK qual-typedef-opt

filename-tick

        : string-tick



string-tick

        : string TICK

qual-symbol-opt

        : expression                        /* Base (global) name */
        | qual-symbol-opt TICK expression   /* Qualified name */

qual-typedef-opt

        : qual-typedef-opt for C

        | qual-typedef-opt for C++

        | qual-typedef-opt for Fortran
```

In the following example, rescoped expressions are used to distinguish which `x` the user is querying, because there are two variables named `x` (one local to `main` and one global):

```
(idb) list $curline - 10:20
      1 long x = 5;             // global x
      2
      3 int main()
      4 {
      5     int x = 7;          // local x
      6     int y = x - ::x;
>     7     return (y);
      8 }
```

By default, a local variable is found before a global one, so that the plain `x` refers to the local variable.

```
(idb) whatis x
int x
(idb) which x
"x rescoped.cxx"`main`x
(idb) whatis "x rescoped.cxx"`main`x
int x
```

You may use the C++ `::` operator to specify the global `x` in C++ code or rescoped expressions in any language.

```
(idb) whatis ::x
long x
(idb) whatis "x rescoped.cxx"`x
long x
(idb) print "x rescoped.cxx"`x
5
```

In the following example, the `x` variable is used in the following places to demonstrate how rescoping expressions can find the correct variable:

- As a variable local to `main`
- As a member variable of the class `Foo`

- As a global variable
- As a local variable to `Foo`'s member function `SetandGet`
- As a local variable to the `CastAndAdd` function, but visible as a parameter

```
(idb) list $curline - 10:20
     10 double x = 3.1415;
     11
     12 int CastAndAdd(char x) {
     13     int result = ((int)::x) + x;
     14     return result;
     15 }
     16
     17 float Foo::SetandGet() {     // multiple scopes!
     18     long x = (long)::x;      // local x = global x
     19     Foo::x = (float)x;       // member x = local x
>    20     return Foo::x;           // return member x
     21 }
     22
     23 int main () {
     24     int x = 7;
     25     x -= CastAndAdd((char)1);
     26
     27     Foo thefoo;
     28     x -= (int)thefoo.SetandGet();
     29     return x;
(idb) whatis x
long x
(idb) which x
"src/x rescoped2.cxx"`Foo::SetandGet`x
(idb) whatis ::x
double x
(idb) whatis Foo::x
float Foo::x
(idb) whatis `main`x
int x
(idb) whatis `CastAndAdd`x
char x
```

Printable Types

The lexical tokens for printable types are specific to the current language.

printable-type

    : *printable-type for C*

    | *printable-type for C++*

    | *printable-type for Fortran*

Expressions Specific to C

The debugger has an almost complete understanding of C expressions, given the general restrictions.

expression

```
        : assignment-expression

constant-expression

        : conditional-expression
```

C Identifiers

The lookup rules are almost always correct for C.

```
identifier-or-typedef-name

        : identifier-or-key-word

        | TYPEDEFname
```

C Constants

The numeric constants are treated exactly the same as in C. The enumeration constant identifiers go though the same grammar paths as variable identifiers, which has basically the same effect as the C semantics.

```
primary-expression

        : identifier-or-key-word

        | constant

        | string-literal-list

        | LPAREN expression RPAREN

        | process_set

        | LPAREN process_range RPAREN

string-literal-list

        : string

        | string-literal-list string

constant

        : FLOATINGconstant

        | INTEGERconstant

        | CHARACTERconstant

        | WIDECHARACTERconstant
        | WIDESTRINGliteral
```

C Rescoped Expressions

The C implementation of rescoped expressions is the following:

```
qual-typedef-opt

        : TYPEDEFname

        | qual-typedef-opt TICK TYPEDEFname

whatis-expressions

        : expression

        | rescoped-expression

        | printable-type
```

C Calls

Following is the C implementation of calls.

```
call-expression

        : expression

function-call

        : postfix-expression LPAREN [arg-expression-list] RPAREN
```

C Addresses

Following is the C implementation of addresses.

```
address

        : AMPERSAND postfix-expression

        | line-address

        | postfix-expression

address-exp

        : address

        | address-exp PLUS  address

        | address-exp MINUS address
        | address-exp STAR  address
```

C Loc Specifications

The C implementation of *loc* is the following:

```
loc

        : expression

        | rescoped-expression
```

C Types

The debugger understands the full C type specification grammar.

```
type-specifier

        : basic-type-specifier

        | struct-union-enum-type-specifier

        | typedef-type-specifier

basic-type-specifier

        : basic-type-name

        | type-qualifier-list basic-type-name

        | basic-type-specifier type-qualifier

        | basic-type-specifier basic-type-name

type-qualifier-list

        : type-qualifier

        | type-qualifier-list type-qualifier

type-qualifier

        : CONST

        | VOLATILE

        | LE

        | _le

        | _be
```

```
basic-type-name

        : VOID

        | CHAR

        | SHORT

        | INT

        | LONG

        | FLOAT

        | DOUBLE

        | SIGNED

        | UNSIGNED

printable-type

        : rescoped_typedef

        | type_name

struct-union-enum-type-specifier

        : elaborated-type-name

        | type-qualifier-list elaborated-type-name

        | struct-union-enum-type-specifier type-qualifier

typedef-type-specifier

        : TYPEDEFname

        | type-qualifier-list TYPEDEFname

        | typedef-type-specifier type-qualifier

elaborated-type-name

        : struct-or-union-specifier

        | enum-specifier

struct-or-union-specifier

        : struct-or-union opt-parenthesized-identifier-or-typedef-name

opt-parenthesized-identifier-or-typedef-name
```

```
        : identifier-or-typedef-name

        | LPAREN opt-parenthesized-identifier-or-typedef-name RPAREN


struct-or-union

        : STRUCT

        | UNION

enum-specifier

        : ENUM identifier-or-typedef-name

type-name

        : type-specifier

        | type-specifier abstract-declarator

        | type-qualifier-list                          // Implicit "int"

        | type-qualifier-list abstract-declarator     // Implicit "int"

type-name-list

        : type-name

        | type-name COMMA type-name-list

abstract-declarator

        : unary-abstract-declarator

        | postfix-abstract-declarator

        | postfixing-abstract-declarator

postfixing-abstract-declarator

        : array-abstract-declarator

        | LPAREN RPAREN

array-abstract-declarator

        : BRACKETS

        | LBRACKET constant-expression RBRACKET

        | array-abstract-declarator LBRACKET constant-expression RBRACKET
```

```
unary-abstract-declarator

        : STAR

        | STAR type-qualifier-list

        | STAR abstract-declarator

        | STAR type-qualifier-list abstract-declarator

postfix-abstract-declarator

        : LPAREN unary-abstract-declarator RPAREN

        | LPAREN postfix-abstract-declarator RPAREN

        | LPAREN postfixing-abstract-declarator RPAREN

        | LPAREN unary-abstract-declarator RPAREN postfixing-abstract-
declarator
```

Other Forms of C Expressions

The following expressions all have their usual C semantics:

```
 assignment-expression

        : conditional-expression

        | unary-expression ASSIGNOP    assignment-expression

        | unary-expression MULTassign  assignment-expression

        | unary-expression DIVassign   assignment-expression

        | unary-expression MODassign   assignment-expression

        | unary-expression PLUSassign  assignment-expression

        | unary-expression MINUSassign assignment-expression

        | unary-expression LSassign    assignment-expression

        | unary-expression RSassign    assignment-expression

        | unary-expression ANDassign   assignment-expression

        | unary-expression ERassign    assignment-expression

        | unary-expression ORassign    assignment-expression

conditional-expression
```

```
        : logical-OR-expression

        | logical-OR-expression QUESTION expression COLON conditional-
expression


logical-OR-expression

        : logical-AND-expression

        | logical-OR-expression OROR logical-AND-expression


logical-AND-expression

        : inclusive-OR-expression

        | logical-AND-expression ANDAND inclusive-OR-expression


inclusive-OR-expression

        : exclusive-OR-expression

        | inclusive-OR-expression OR inclusive-OR-expression


exclusive-OR-expression

        : AND-expression

        | exclusive-OR-expression HAT AND-expression


AND-expression

        : equality-expression

        | AND-expression AMPERSAND equality-expression


equality-expression

        : relational-expression

        | equality-expression EQ relational-expression

        | equality-expression NE relational-expression
```

```
relational-expression

        : shift-expression

        | relational-expression LESS    shift-expression

        | relational-expression GREATER shift-expression

        | relational-expression LE      shift-expression

        | relational-expression GE      shift-expression


shift-expression

        : additive-expression

        | shift-expression LS additive-expression

        | shift-expression RS additive-expression


additive-expression

        : multiplicative-expression

        | additive-expression PLUS  multiplicative-expression

        | additive-expression MINUS multiplicative-expression


multiplicative-expression

        : cast-expression

        | multiplicative-expression STAR  cast-expression

        | multiplicative-expression SLASH cast-expression

        | multiplicative-expression MOD   cast-expression


cast-expression

        : unary-expression

        | LPAREN type-name RPAREN cast-expression
```

```
unary-expression
        : postfix-expression
        | INCR unary-expression
        | DECR unary-expression
        | AMPERSAND cast-expression
        | STAR cast-expression
        | PLUS cast-expression
        | MINUS cast-expression
        | TWIDDLE cast-expression
        | NOT cast-expression
        | SIZEOF unary-expression
        | SIZEOF LPAREN type-name RPAREN
        | line-address

postfix-expression
        : primary-expression
        | postfix-expression LBRACKET expression RBRACKET
        | function-call
        | postfix-expression LPAREN type-name-list RPAREN
        | postfix-expression DOT   identifier-or-typedef-name
        | postfix-expression ARROW identifier-or-typedef-name
        | postfix-expression INCR
        | postfix-expression DECR
```

Expressions Specific to C++

C++ is a complex language, with a rich expression system. The debugger understands much of the system, but it does not understand how to evaluate some complex aspects of a C++ expression. It can correctly debug these when they occur in the source code.

The aspects of the expression system not processed properly during debugger expression evaluation include the following:

- Many of the implicit conversions
- Program-defined operators
- Calling constructors and destructors during the debugger's own evaluation of expressions

There are also some minor restrictions in the following grammar, compared with the full C++ expression grammar, to make it unambiguous:

```
expression

        : assignment-expression



constant-expression

        : conditional-expression
```

C++ Identifiers

The debugger correctly augments the general lookup rules when inside class member functions, to look up the members correctly.

The debugger has only a limited understanding of namespaces. It correctly processes names such as `UserNameSpace::NestedNamespace::userIdentifier`, as well as C++ use-declarations, which introduce a new identifier into a scope.

The debugger does not currently understand C++ using-directives.

The debugger understands the relationship between `struct` and class identifiers and `typedef` identifiers.

```
id-or-keyword-or-typedef-name

        : identifier-or-key-word

        | TYPEDEFname
```

C++ Constants

The debugger treats numeric constants the same as C++ does. The enumeration constant identifiers go though the same grammar paths as variable identifiers, which has basically the same effect as the C++ semantics.

```
constant

        : FLOATINGconstant

        | INTEGERconstant

        | CHARACTERconstant

        | WIDECHARACTERconstant

        | WIDESTRINGliteral
```

C++ Calls

The debugger does not understand the following aspects of C++ calls:

- Invoking C++ constructors and destructors to create and destroy temporaries containing the value of parameters and results.
- Default parameters.
- Many of the implicit conversions that may be needed for the parameters.
- Overloading resolution. Instead, the debugger queries the user.

```
call-expression

        : expression
```

C++ Addresses

Following is the C++ implementation of addresses:

```
address

        : AMPERSAND postfix-expression /* Address of */

        | line-address

        | postfix-expression

address-exp

        : address

        | address-exp PLUS  address
        | address-exp MINUS address

        | address-exp STAR  address
```

C++ Loc

Following is the C++ implementation of *loc*:

```
loc

        : expression

        | rescoped-expression
```

Other Modified Forms of C++ Expressions

```
whatis-expressions

        : expression

        | printable-type
```

C++ Rescoped Expressions

The C++ implementation of rescoped expressions is as follows:

```
 qual-typedef-opt

        : type-name

        | qual-typedef-opt TICK type-name
```

C++ Strings

The C++ implementation of strings is as follows:

```
string-literal-list

        : string

        | string-literal-list string
```

C++ Identifier Expressions

The debugger understands nested names. Namespaces go through the same paths as classes, hence the unusual use of TYPEDEFname.

```
id-expression

       : id-expression-internals
```

```
id-expression-internals

        : qualified-id

        | id-or-keyword-or-typedef-name

        | operator-function-name

        | TWIDDLE id-or-keyword-or-typedef-name


qualified-id

        : nested-name-specifier qualified-id-follower


qualified-type

        : nested-name-specifier TYPEDEFname


nested-name-specifier

        : CLCL

        | TYPEDEFname CLCL

        | nested-name-specifier TYPEDEFname CLCL


qualified-id-follower

        : identifier-or-key-word

        | operator-function-name

        | TWIDDLE id-or-keyword-or-typedef-name
```

C++ Types

The debugger understands the full C++ type specification grammar.

```
type-specifier

         : basic-type-specifier
```

```
        | struct-union-enum-type-specifier

        | typedef-type-specifier


type-qualifier-list

        : type-qualifier

        | type-qualifier-list type-qualifier


type-qualifier

        : CONST

        | VOLATILE

        | LE

        | _le

        | _be

basic-type-specifier

        : basic-type-name basic-type-name

        | basic-type-name type-qualifier

        | type-qualifier-list basic-type-name

        | basic-type-specifier type-qualifier

        | basic-type-specifier basic-type-name


struct-union-enum-type-specifier

        : elaborated-type-name

        | type-qualifier-list elaborated-type-name

        | struct-union-enum-type-specifier type-qualifier


typedef-type-specifier

        : TYPEDEFname type-qualifier
        | type-qualifier-list TYPEDEFname

        | typedef-type-specifier type-qualifier


basic-type-name

        : VOID

        | CHAR

        | SHORT

        | INT

        | LONG

        | FLOAT

        | DOUBLE

        | SIGNED

        | UNSIGNED


elaborated-type-name

        : aggregate-name

        | enum-name


printable-type

        : rescoped-typedef

        | type-name


aggregate-name

        : aggregate-key opt-parenthesized-identifier-or-typedef-name

        | aggregate-key qualified-type


opt-parenthesized-identifier-or-typedef-name
```

```
        : id-or-keyword-or-typedef-name
        | LPAREN opt-parenthesized-identifier-or-typedef-name RPAREN


aggregate-key
        : STRUCT
        | UNION
        | CLASS


enum-name
        : ENUM id-or-keyword-or-typedef-name


parameter-type-list
        : PARENS type-qualifier-list-opt


type-name
        : type-specifier
        | qualified-type
        | basic-type-name
        | TYPEDEFname
        | type-qualifier-list
        | type-specifier abstract-declarator
        | basic-type-name abstract-declarator
        | qualified-type abstract-declarator
        | TYPEDEFname abstract-declarator
        | type-qualifier-list abstract-declarator


abstract-declarator
```

```
        : unary-abstract-declarator

        | postfix-abstract-declarator

        | postfixing-abstract-declarator


postfixing-abstract-declarator

        : array-abstract-declarator

        | parameter-type-list


array-abstract-declarator

        : BRACKETS

        | LBRACKET constant-expression RBRACKET

        | array-abstract-declarator LBRACKET constant-expression RBRACKET


unary-abstract-declarator

        : STAR

        | AMPERSAND

        | pointer-operator-type

        | STAR                  abstract-declarator

        | AMPERSAND             abstract-declarator

        | pointer-operator-type abstract-declarator


postfix-abstract-declarator

        : LPAREN unary-abstract-declarator RPAREN

        | LPAREN postfix-abstract-declarator RPAREN

        | LPAREN postfixing-abstract-declarator RPAREN

        | LPAREN unary-abstract-declarator RPAREN postfixing-abstract-
declarator
```

```
pointer-operator-type

        : TYPEDEFname CLCL STAR type-qualifier-list-opt

        | STAR                  type-qualifier-list

        | AMPERSAND             type-qualifier-list
```

Other Forms of C++ Expressions

The following expressions all have the usual C++ semantics:

```
primary-expression

        : constant

        | string-literal-list

        | THIS

        | LPAREN expression RPAREN

        | operator-function-name

        | identifier-or-key-word

        | qualified-id

        | process_set

        | LPAREN process_range RPAREN


operator-function-name

        : OPERATOR operator-predefined

        | OPERATOR basic-type-name

        | OPERATOR TYPEDEFname

        | OPERATOR LPAREN type-name RPAREN

        | OPERATOR type-qualifier

        | OPERATOR qualified-type


operator-predefined
```

```
        : PLUS
        | MINUS
        | STAR
        | ...
        | DELETE
        | COMMA

type-qualifier-list-opt
        : [ type-qualifier-list ]

postfix-expression
        : primary-expression
        | primary-expression LBRACKET expression RBRACKET
        | postfix-expression PARENS
        | postfix-expression LPAREN  argument-expression-list RPAREN
        | postfix-expression LPAREN  type-name-list RPAREN
        | postfix-expression DOT  id-expression
        | postfix-expression ARROW id-expression
        | postfix-expression INCR
        | postfix-expression DECR
        | TYPEDEFname LPAREN argument-expression-list RPAREN
        | TYPEDEFname LPAREN type-name-list RPAREN
        | basic-type-name LPAREN assignment-expression RPAREN

type-name-list
        : type-name
        | type-name COMMA type-name-list
        | type-name comma-opt-ellipsis

        | ELLIPSIS


comma-opt-ellipsis

        : ELLIPSIS

        | COMMA ELLIPSIS


unary-expression

        : postfix-expression

        | INCR unary-expression

        | DECR unary-expression

        | line-address

        | AMPERSAND cast-expression

        | STAR cast-expression

        | MINUS cast-expression

        | PLUS cast-expression

        | TWIDDLE LPAREN cast-expression RPAREN

        | NOT cast-expression

        | SIZEOF unary-expression

        | SIZEOF LPAREN type-name RPAREN

        | allocation-expression


allocation-expression

        : operator-new LPAREN type-name RPAREN operator-new-initializer

        | operator-new LPAREN argument-expression-list RPAREN LPAREN type-name
RPAREN operator-new-initializer
```

```
operator-new
        : NEW
        | CLCL NEW

operator-new-initializer
        : [ PARENS ]
        | [ LPAREN argument-expression-list RPAREN ]

cast-expression
        : unary-expression
        | LPAREN type-name RPAREN cast-expression

deallocation-expression
        : cast-expression
        | DELETE               deallocation-expression
        | CLCL DELETE          deallocation-expression
        | DELETE BRACKETS      deallocation-expression
        | CLCL DELETE BRACKETS deallocation-expression

point-member-expression
        : deallocation-expression
        | point-member-expression DOTstar    deallocation-expression
        | point-member-expression ARROWstar  deallocation-expression

multiplicative-expression
        : point-member-expression
        | multiplicative-expression STAR  point-member-expression
        | multiplicative-expression SLASH point-member-expression
        | multiplicative-expression MOD   point-member-expression

additive-expression
        : multiplicative-expression
        | additive-expression PLUS  multiplicative-expression
        | additive-expression MINUS multiplicative-expression

shift-expression
        : additive-expression
        | shift-expression LS additive-expression
        | shift-expression RS additive-expression

relational-expression
        : shift-expression
        | relational-expression LESS    shift-expression
        | relational-expression GREATER shift-expression
        | relational-expression LE      shift-expression
        | relational-expression GE     shift-expression

equality-expression
        : relational-expression
        | equality-expression EQ relational-expression
        | equality-expression NE relational-expression

AND-expression
        : equality-expression
        | AND-expression AMPERSAND equality-expression


exclusive-OR-expression

        : AND-expression

        | exclusive-OR-expression HAT AND-expression


inclusive-OR-expression

        : exclusive-OR-expression

        | inclusive-OR-expression OR exclusive-OR-expression


logical-AND-expression

        : inclusive-OR-expression

        | logical-AND-expression ANDAND inclusive-OR-expression


logical-OR-expression

        : logical-AND-expression

        | logical-OR-expression OROR logical-AND-expression


conditional-expression

        : logical-AND-expression

        | logical-AND-expression QUESTION expression COLON conditional-
expression


assignment-expression

        : conditional-expression

        | unary-expression ASSIGNOP    assignment-expression

        | unary-expression MULTassign  assignment-expression

        | unary-expression DIVassign   assignment-expression
        | unary-expression MODassign   assignment-expression
        | unary-expression PLUSassign  assignment-expression
        | unary-expression MINUSassign assignment-expression
        | unary-expression LSassign    assignment-expression
        | unary-expression RSassign    assignment-expression
        | unary-expression ANDassign   assignment-expression
        | unary-expression ERassign    assignment-expression
        | unary-expression ORassign    assignment-expression
```

Expressions Specific to Fortran

This section contains expressions specific to Fortran.

Fortran Identifiers

The Fortran implementation of identifiers is as follows:

```
identifier-or-typedef-name
        : identifier-or-key-word
        | TYPEDEFname
        | PROCEDUREname
```

Fortran Constants

```
real-or-imag-part
        : real_constant
        | PLUS real_constant
        | MINUS real_constant
        | integer_constant
        | PLUS integer_constant
        | MINUS integer_constant


constant
```

```
        : real_constant

        | integer_constant

        | complex-constant

        | character_constant

        | LOGICALconstantWithKind


character_constant

        : CHARACTERconstantWithKind

        | string


complex-constant

        : LPAREN real-or-imag-part COMMA real-or-imag-part RPAREN
```

Fortran Rescoped Expressions

The Fortran implementation of rescoped expressions is as follows:

```
qual-typedef-opt

        : TYPEDEFname /* Base (global) name */

        | qual-typedef-opt TICK TYPEDEFname /* Qualified name */


whatis-expressions

        : expression

        | rescoped-expression

        | printable_type
```

Fortran Calls

The Fortran implementation of calls is as follows:

```
call-expression

        : call-stmt
```

```
call-stmt

        : named-subroutine

        | named-subroutine LPAREN RPAREN

        | named-subroutine LPAREN actual-arg-spec-list RPAREN
```

Fortran Addresses

The Fortran implementation of addresses is as follows:

```
address

        : line-address

        | primary


address-exp

        : address

        | address-exp PLUS  address

        | address-exp MINUS address

        | address-exp STAR  address
```

Restrictions and limits are documented here.

Fortran Loc

The Fortran implementation of loc is as follows:

```
loc

        : expression

        | rescoped-expression
```

Fortran Types

The Fortran implementation of types is as follows:

```
type-name
```

```
        : TYPEDEFname


printable-type

        : rescoped-typedef

        | type-name
```

Other Forms of Fortran Expressions

```
expression

        : expr

        | named-procedure


assignment-expression

        : expr


constant-expression

        : constant


unary-expression

        : variable


expr

        : level-5-expr

        | expr defined-binary-op level-5-expr


level-5-expr

        : equiv-operand

        | level-5-expr LOGEQV equiv-operand
        | level-5-expr LOGNEQV equiv-operand

        | level-5-expr LOGXOR equiv-operand


equiv-operand

        : or-operand

        | equiv-operand LOGOR or-operand


or-operand

        : and-operand

        | or-operand LOGAND and-operand


and-operand

        : level-4-expr

        | LOGNOT and-operand


level-4-expr

        : level-3-expr

        | level-3-expr LESS level-3-expr

        | level-3-expr GREATER level-3-expr

        | level-3-expr LE level-3-expr

        | level-3-expr GE level-3-expr

        | level-3-expr EQ level-3-expr

        | level-3-expr NE level-3-expr


level-3-expr

        : level-2-expr

        | level-3-expr SLASHSLASH level-2-expr
```

```
level-2-expr

        : and-operand

        | level-2-expr PLUS  add-operand

        | level-2-expr MINUS add-operand


add-operand

        : add-operand-f90

        | add-operand-dec

        | unary-expr-dec


add-operand-f90

        : mult-operand-f90

        | add-operand-f90 STAR  mult-operand-f90

        | add-operand-f90 SLASH mult-operand-f90


mult-operand-f90

        : level-1-expr

        | level-1-expr STARSTAR mult-operand-f90


add-operand-dec

        : mult-operand-dec

        | add-operand-f90 STAR  mult-operand-dec

        | add-operand-f90 SLASH mult-operand-dec

        | add-operand-f90 STAR  unary-expr-dec

        | add-operand-f90 SLASH unary-expr-dec
```

```
mult-operand-dec

        : level-1-expr STARSTAR mult-operand-dec

        | level-1-expr STARSTAR unary-expr-dec


unary-expr-dec

        : PLUS  and-operand

        | MINUS and-operand


level-1-expr

        : primary

        | defined-unary-op primary


defined-unary-op

        : DOT_LETTERS_DOT


primary

        : constant

        | variable

        | function-reference

        | LPAREN expr RPAREN

        | AMPERSAND variable

        | process_set

        | LPAREN process_range RPAREN


defined-binary-op

        : DOT_LETTERS_DOT
```

```
int-expr

        : expr


scalar-int-expr

        : int-expr


variable

        : named-variable

        | subobject


named-variable

        : variable-name

        | module_prefix PERCENT_PERCENT variable-name


subobject

        : array-elt-or-sect

        | structure-component

        | known-substring


known-substring

        : disabled-array-elt-or-sect LPAREN substring-range RPAREN

        | hf-array-abomination


substring-range

        : scalar-int-expr COLON scalar-int-expr
        | scalar-int-expr COLON
        |                 COLON scalar-int-expr
        |                 COLON


hf-array-abomination
        : named-variable
          LPAREN section-subscript-list RPAREN
          LPAREN section-subscript RPAREN
        | structure PERCENT any-identifier
          LPAREN section-subscript-list RPAREN
          LPAREN section-subscript RPAREN
        | structure DOT any-identifier
          LPAREN section-subscript-list RPAREN
          LPAREN section-subscript RPAREN


disabled-array-elt-or-sect
        : DISABLER array-elt-or-sect


array-elt-or-sect
        : named-variable LPAREN section-subscript-list RPAREN
        | structure PERCENT any-identifier LPAREN section-subscript-list
RPAREN
        | structure DOT any-identifier LPAREN section-subscript-list RPAREN


section-subscript-list
        : section-subscript
        | section-subscript COMMA section-subscript-list
```

```
subscript

        : scalar-int-expr


section-subscript

        : subscript

        | subscript-triplet


subscript-triplet

        : subscript COLON subscript COLON stride

        | subscript COLON           COLON stride

        |           COLON subscript COLON stride

        |           COLON           COLON stride

        | subscript COLON subscript

        | subscript COLON

        |           COLON subscript

        |           COLON


stride

        : scalar-int-expr


structure-component

        : structure PERCENT any-identifier

        | structure DOT any-identifier


structure

        : named-variable
        | structure-component

        | array-elt-or-sect


function-reference

        : SIZEOF LPAREN expr RPAREN

        | named-function LPAREN RPAREN

        | named-function LPAREN actual-arg-spec-list RPAREN


named-procedure

        :  PROCEDUREname


named-function

        :  PROCEDUREname

        | module_prefix PERCENT_PERCENT PROCEDUREname

named-subroutine

        :  PROCEDUREname


actual-arg-spec-list

        : actual-arg-spec

        | actual-arg-spec COMMA actual-arg-spec-list


actual-arg-spec

        : actual-arg


actual-arg

        : expr
```

```
any-identifier

        : variable-name

        | PROCEDUREname



variable-name

        : identifier-or-key-word



PROCEDUREname

        : IDENTIFIER



module_prefix

        : any_identifier

        | module_prefix PERCENT_PERCENT any_identifier
```

Debugging Core Files Overview

When the operating system encounters an unrecoverable error while running a process, for example a segmentation violation (SEGV), the system creates a file named `core` and places it in the current directory. The core file is not an executable file; it is a snapshot of the state of your process at the time the error occurred. It allows you to analyze the process at the point it crashed.

This section discusses the following topics:

- Invoking the debugger on a core file
- Debugging a core file
- Transporting a core file

Invoking the Debugger on a Core File

You can use the debugger to examine the process information in a core file. Use the following debugger command syntax to invoke the debugger on a core file:

```
% idb executable_file core_file
```

or

```
 (idb) load executable_file core_file
```

The executable file is that which was being executed at the time the core file was generated.

Debugging a Core File

When debugging a core file, you can use the debugger to obtain a stack trace and the values of some variables just as you would for a stopped process.

The stack trace lists the functions in your program that were active when the dump occurred. By examining the values of a few variables along with the stack trace, you may be able to pinpoint the process state and the cause of the core dump. Core files cannot be executed; therefore the `rerun`, `step`, `cont` and so on commands will not work until you create a process using the `run` command.

In addition, if the program is multithreaded, you can examine the thread information with the `show thread` and `thread` commands. You can examine the stack trace for a particular thread or for all threads with the `where thread` command.

The following example uses a null pointer reference in the `factorial` function. This reference causes the process to abort and dump the core when it is executed. The **`dump`** command prints the value of the `x` variable as a null, and the **`print *x`** command reveals that you cannot dereference a null pointer.

```
% cat testProgram.c
#include <stdio.h>
int factorial(int i)
main() {
        int i,f;
        for (i=1 ; i<3 ; i++) {
                f = factorial(i);
                printf("%d! = %d\en",i,f);
        }
}
int factorial(int i)
int i;
{
int *x;
         x = 0;
         printf("%d",*x);
         if (i<=1)
                 return (1);
         else
                 return (i * factorial(i-1) );
}
% cc -o testProgram -g testProgram.c
% testProgram
Memory fault - core dumped.
% idb testProgram core
Welcome to the debugger Version n
------------------
object file name: testProgram
core file name: core
Reading symbolic information ...done
```

```
Core file produced from executable testProgram
Thread terminated at PC 0x120000dc4 by signal SEGV
(idb) where
>0  0x120000dc4 in factorial(i=1) testProgram.c:13
#1  0x120000d44 in main() testProgram.c:4
(idb) dump
>0  0x120000dc4 in factorial(i=1) testProgram.c:13
printf("%d",*x);
(idb) print *x
Cannot dereference 0x0
Error: no value for *x
(idb)
```

Transporting a Core File

Transporting core files is usually necessary to debug a core file on a system other than that which produced it. It is sometimes possible to debug a core file on a system other than that which produced it if the current system is sufficiently similar to the original system, but it will not work correctly in general.

When moving core files, don't forget to move the original executable and any shared components it required.

If the POSIX Threads Library is involved, set the LD_LIBRARY_PATH environment variable to the debugger subdirectory so that the debugger will use the correct `libpthreaddebug.so`.

```
% env IDB COREFILE LIBRARY PATH=applibs \
LD LIBRARY PATH=dbglibs \
idb a.out core
```

Kernel Debugging

Kernel Debugging is currently unsupported

Machine-Level Debugging. Advanced Debugging

The debugger lets you debug your programs at the machine-code level as well as at the source-code level. Using debugger commands, you can examine and edit values in memory, print the values of all machine registers, and step through program execution one machine instruction at a time.

Only those users familiar with machine-language programming and executable-file-code structure will find low-level debugging useful.

This section contains the following topics:

- Examining memory addresses
- Stepping at the machine level
- Disassembling Functions

Examining Memory Addresses

You can examine the value contained at an address in memory as follows:

- The examine commands (`/` and `?`) display the values stored in memory.
- The **`print`** `command`, with the appropriate pointer arithmetic, prints the value contained at the address in decimal.
- The **`printregs`** `command` prints the values of all machine-level registers.

In addition to examining memory, you can also search memory in 32 and 64-bit chunks.

Using the Examine Commands

You can use the examine commands (`/` and `?`) to print the value contained at the address in one of a number of formats (decimal, octal, hexadecimal, and so on). See Memory Display Commands for more information.

The debugger also maintains the `$readtextfile` debugger variable that allows you to view the data from the text section of the executable directly from the binary file, rather than reading it from memory.

Using Pointer Arithmetic

You can use C and C++ pointer-type conversions to display the contents of a single address in decimal. Using the **`print`** command, the syntax is as follows:

```
(idb) print *(int *)(address)
```

Using the same pointer arithmetic, you can use the **`assign`** command to alter the contents of a single address. Use the following syntax:

```
(idb) assign *(int *)(address) = value
```

The following example shows how to use pointer arithmetic to examine and change the contents of a single address:

```
(idb) print *(int*)(0x10000000)
 4198916
(idb) assign *(int*)(0x10000000) = 4194744
(idb) print *(int*)(0x10000000)
 4194744
(idb)
```

Examining Machine-Level Registers

The **printregs** command prints the values of all machine-level registers. The registers displayed by the debugger are machine dependent. The values are in decimal or hexadecimal, depending on the value of the `$hexints` variable (the default is 0, decimal). The register aliases are shown; for example, `$r1 [$t0]`.

Stepping at the Machine Level

The **stepi** and **nexti** commands let you step through program execution incrementally, like the **step** and **next** commands. The **stepi** and **nexti** commands execute one machine instruction at a time, as opposed to one line of source code. The following example shows stepping at the machine-instruction level:

```
(idb) stop in main
[#1: stop in main ]
(idb) run
[1] stopped at [main:4 0x120001180]
4      for (i=1 ; i<3 ; i++) {
(idb) stepi
stopped at [main:4 0x120001184] stl     t0, 24(sp)
(idb) [Return]
stopped at [main:5 0x120001188] ldl     a0, 24(sp)
(idb) [Return]
stopped at [main:5 0x12000118c] ldq     t12, -32664(gp)
(idb) [Return]
stopped at [main:5 0x120001190] bsr     ra,
(idb) [Return]
stopped at [factorial:12 0x120001210]   ldah    gp, 8192(t12)
(idb)
```

At the machine-instruction level, you can step into, rather than over, a function's prologue. While within a function prologue, you may find that the stack trace, variable scope, and parameter list are not correct. Stepping out of the prologue and into the actual function updates the stack trace and variable information kept by the debugger.

Single-stepping through function prologues that initialize large local variables is slow. As a workaround, use the **next** command.

Disassembling Functions

The alias **lfi** lets you disassemble all of the instructions in a single function. Instructions from other functions may be intermixed in the disassembly due to compiler optimizations. The following example shows how to disassemble a member function:

```
(idb)alias lfi
lfi(x)  set $lfitmp = x ; $lfitmp, $highpc($lfitmp) / i ; unset $lfitmp
(idb) lfi( Node::getNextNode )
class Node* Node::getNextNode(void): src/x list.cxx
 [line 81, 0x0804b5aa]  getNextNode:                  pushl    %ebp
 [line 81, 0x0804b5ab]  getNextNode+0x1:              movl     %esp, %ebp
 [line 81, 0x0804b5ad]  getNextNode+0x3:              movl     0x8(%ebp), %eax
 [line 81, 0x0804b5b0]  getNextNode+0x6:              movl     0x4(%eax), %eax
 [line 81, 0x0804b5b3]  getNextNode+0x9:              leave
 [line 81, 0x0804b5b4]  getNextNode+0xa:              ret
 [line 81, 0x0804b5b5]  getNextNode+0xb:              nop
(idb)
```

Debugging Parallel Applications. Overview

The Intel® IDB supports debugging of message passing interface (MPI) applications launched by

- mpirun, a MPI launcher from mpich, a public domain implementation of MPI.
- prun, a parallel launcher of Resource Management System (RMS) from Quadrics.

This chapter contains the following sections:

- Starting a parallel debugging session
- Using commands in a parallel debugging session
- Working with sets of application processes
- Working with aggregated messages
- Parallel debugging tips
- Parallel debugging examples
- Using the `mpirun_dbg.idb` startup file

Overview

The biggest challenge of debugging massively parallel applications is coping with large quantities of output from debuggers controlling the parallel application's processes. Intel Debugger helps you do this by condensing (aggregating) similar output into groups. Aggregation is performed by using the following two strategies:

- Identical output messages are condensed into a single output message. When a condensed message is displayed, it is prefixed with a range of user process IDs (not necessarily consecutive) to which this output applies. All processes with the same output are aggregated into a single and final output message, for example:

  ```
  [0-41] Intel(R) Debugger for Itanium®-based applications, Version XX
    |
  Process range
  ```

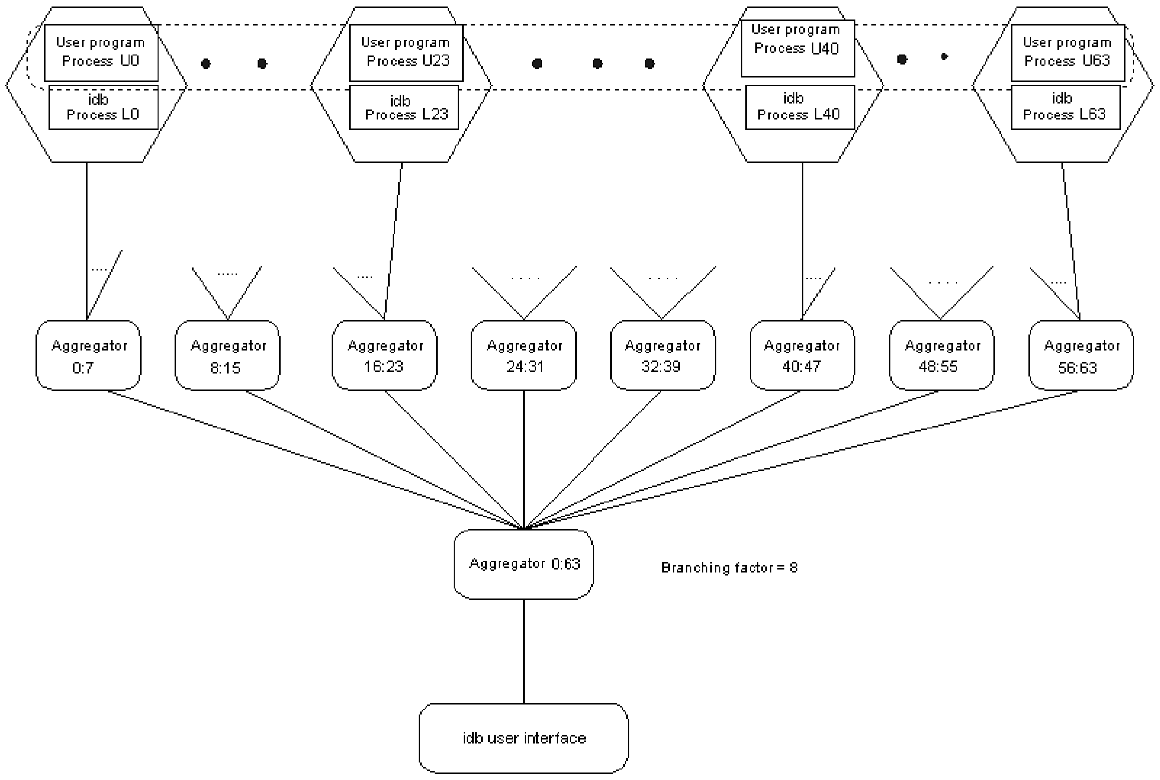
- Outputs that have different hexadecimal digits, but are otherwise identical, are condensed by aggregating the differing digits into a range, for example:

  ```
  [0-41]>2 0x120006d6c in
  feedback(myid=[0;41],np=42,name=0x11fffe018="mytest") "mytest.c":41
  ```

```
   |                                         |
Process range                          Value range
```

Another challenge of debugging massively parallel applications is controlling all processes or subsets of the parallel application's processes from the debugger in a consistent manner. The debugger allows you to control all or a subset of your processes through a single user interface. At the startup of a parallel debugging session, Intel IDB does the following:

1. Detects the topology of your application and attaches a debugger to each of your application's processes.
2. Builds an *n*-nary tree with the debuggers as root and leaves with special processes called aggregators in the middle (shown in the following diagram). You can specify the tree's branching factor and the aggregator time delay.



The root debugger is responsible for starting your parallel application and serves as your user interface. The aggregators perform output consolidation as described previously. The leaf debuggers control and query your application processes.

The branching factor is the factor used to build the *n*-nary tree and determine the number of aggregators in the tree. For example, for 16 processes:

- Using a branching factor of 8 creates 3 aggregators
- Using a branching factor of 2 creates 15 aggregators

You can set the value of the `$parallel_branchingfactor` variable from its default value of 8 to a value equal to or greater than 2 in the Intel IDB initialization file (`.idbrcidbinit.idb`, and so on).

When you delete `$parallel_branchingfactor` from the Intel IDB initialization file, the branching factor used in the startup mechanism is the default value.

Aggregator delay specifies the time that aggregators wait before they aggregate and send messages down to the next level when not all of the expected messages have been received.

You can change the value of the `$parallel_aggregatordelay` variable from its default value of 3000 milliseconds in the Intel IDB initialization file (`.idbrcidbinit.idb`, etc.). See Parallel Debugging Tips for more information.

When you delete `$parallel_aggregatordelay` from the Intel IDB initialization file, the aggregator delay used in the startup mechanism is the default value.

Note:

You can only change the values that are set for `$parallel_branchingfactor` and `$parallel_aggregatordelay` at startup, in the `.idbrcidbinit.idb` file. After the program has started up, you cannot change these values.

Note:

The Intel® Debugger uses rsh to create the leaf debugger and aggregator processes in the tree structure. Make sure that every node in your cluster has rsh privilege to all other cluster nodes for proper setup of the tree structure.

Starting a Parallel Debugging Session

To start your parallel application under debugger control, you need to have the environment variable **`IDB_HOME`** set to the directory that your debugger is in. When debugging an application launched by mpich, issue the following command at the shell, where *`N`* represents number of processes and *`application`* is the name of the MPP program you would like to debug:

```
% mpirun -dbg=idb -np N [other mpich options] application [application
arguments] [-idb idb options]
```

Make sure that there is a file called `mpirun_dbg.idb` in the directory in which `mpirun` is located. Also note that the Intel IDB option `-gdb` is not yet supported under parallel debugging session.

When the debugger starts your parallel application, it detects and attaches to all of your application's processes. At this point, your application stops before executing any user code and the debugger displays a prompt.

You can now set any necessary breakpoints and use the **`continue`** command to continue the execution of your application.

When debugging an application launched by prun, issue the following command at the shell, where n is the number of processes and N is the number of nodes running the processes.

```
% $IDB HOME/idb [idb options] -parallel `which prun` -n n -N N [other prun
options] application [application arguments]
```

Using Commands in a Parallel Debugging Session

You can use most Intel® Debugger commands just as you would when debugging a non-parallel application. Most commands are passed on to the leaf debuggers and you see aggregated output from them in your user interface. However, there are a few important exceptions.

The following two tables show the DBX and GDB debugger commands that can be accessed remotely, locally, and both remotely and locally for parallel debugging; and Intel® Debugger commands that are disabled for parallel debugging.

DBX commands:

Remote	Local	Both Remote and Local	Disabled

`#`
`/`
`?`
`assign`
`call`
`catch`
`class`
`cont/conti`
`delete`
`delsharedobj`
`disable`
`down`
`dump`
`enable`
`examine_address`
`file`
`func`
`goto`
`history`
`if`
`ignore`
`kill`
`list`
`listobj`
`map/unmap source`
`directory`
`next/nexti`
`pop`
`print`
`printb`
`printd`
`printf`
`printi`
`printo`
`printregs`
`printt`
	`!/history`
	`alias/unalias`
	`edit`
	`export`
	`help`
	`playback`
	`quit`
	`record/unrecord`
	`source`
		`export`
		`set/setenv`
		`sh`
		`unset/unsetenv`
			`attach/detach`
			`kps`
			`load/unload`
			`patch`
			`printenv`
			`rerun`
			`run`
			`snapshot`

printx process readsharedobj return show condition show process show source directory show thread status step/stepi stop/stopi thread trace/tracei use/unuse up watch whatis when/wheni where/whereis which			

GDB commands:

Remote	Local	Both Remote and Local	Disabled
advance	complete	pwd	attach/detach
awatch	echo	quit	file
backtrace/where	help	set/set variable	run
break	set editing	shell	set args
call	set height	show convenience	set debug remote
clear	set prompt	show environment	set debug serial
condition	set width	unset environment	set debug target
continue	show editing		set remotedevice
delete	show height		set remoteaddresssize
disable	show prompt		set remotecache
directory	show width		set remotelogbase
disassemble	source		set remotelogfile
disconnect			set follow-fork-mode
display			show debug remote
down			show debug serial

`down-silently`
`enable`
`finish`
`forward-search`
`frame`
`handle`
`ignore`
`info`
`jump`
`kill`
`list`
`maintenance`
`next`
`nexti`
`output`
`path`
`signal`
`print/inspect`
`printf`
`ptype`
`return`
`reverse-search`
`rwatch`
`set confirm`
`set demangle-style`
`set language`
			`show debug target`
			`show remotedevice`
			`show remoteaddresssize`
			`show remotecache`
			`show remotelogbase`
			`show remotelogfile`
			`show follow-fork-mode`
			`target remote`
			`togglegdbserver`

`set listsize`
 	
`set output-radix`
 	
`set print`
 	
`set prompt`
 	
`sharedlibrary`
 	
`show architecture`
 	
`show commands`
 	
`show demangle-style`
 	
`show directories`
 	
`show language`
 	
`show listsize`
 	
`show output-radix`
 	
`show path`
 	
`show print`
 	
`show process`
 	
`show prompt`
 	
`show values`
 	
`show version`
 	
`tstatus`
 	
`step`
 	
`stepi`
 	
`symbol-file`
 	
`tbreak`
 	
`tbreak`
 	
`thread`
 	
`undisplay`			

until **up** **up-silently** **watch** **whatis** **x**			

Remote means commands will be sent to the leaf debuggers. Local means that commands are not sent to the leaf debuggers but are processed by the local Intel® IDB.

In addition to the commands listed in the table, you can use four other Intel® Debugger commands to assist parallel debugging:

```
parallel_debugging_command

        : focus_command

        | show_process_set_command

        | show_aggregated_message_command

        | expand_aggregated_message_command
```

Working With Sets of Application Processes

When there are many processes, it can be annoying or impractical to enumerate all the processes when one needs to focus on specific processes. Therefore, Intel IDB introduces the concept of "process sets" and "process ranges" to let the user specify a group of processes in a compact form. Moreover, process sets come with the usual set operations, and both the sets and the ranges can be stored in debugger variables for manipulation, reference, or inspection at a later time.

A process set is a bracketed list of process ranges separated by commas.

Note:

> Because brackets (`[]`) are part of the process set syntax, this section shows optional syntactic items enclosed in curly braces (`{}`).

```
process_set

       : [ ]

       | [ process_range {,...} ]
```

Note:

The set can be empty.

A process range has the following three forms:

```
process_range

        : *

        | expression

        | { expression } : { expression }
```

In the first form, the star (`*`) specifies all processes.

You can use the second form as follows:

- If expression evaluates to, or can be coerced into an integer `p`, then the range contains the process with pid `p` only.
- If expression evaluates to a process range `r`, then the process range is the same as `r`.

You can use the third form to specify a contiguous range of processes. For example, `10:12` stands for the processes associated with `pids` 10, 11, and 12.

Note:

A range whose lower bound is greater than its upper bound is illegal and will be ignored.

Because both the lower bound and the upper bound are optional, you can specify ranges as follows:

Example	Represents
`:5`	All processes whose `pid` is no greater than 5.
`20:`	All processes whose `pid` is no less than 20.
`:`	The process set [`:`] is equivalent to the process set [`*`].

Using Debugger Variables to Store Process Sets and Ranges

Like storing other data types supported by the debugger, you can store process sets and process ranges in debugger variables using the **`set`** command. For example:

```
(idb) set $set1 = [:7, 10, 15:20, 30:]
(idb) print $set1
```

```
[:7, 10, 15:20, 30:]
```

In addition to using the **`print`** command, you can also use the **`show process set`** command to inspect the process set stored in a debugger variable. For example:

```
show_process_set_command

        : show process set debugvar_name

        | show process set all

        | show process set
```

If you do not specify the set name, or if you use the *`all`* specifier, the debugger displays all the process sets that are currently stored in debugger variables, as the continued example shows:

```
(idb) set $set2 = [8:9, 5:2, 22:27]
`5:2' is not a legal process range. Ignored.
(idb) show process set $set2
$set2 = [8:9, 22:27]
(idb) show process set *
$set1 = [:7, 10, 15:20, 30:]
$set2 = [8:9, 22:27]
```

Process Set Operations

You can use the following three operations on process sets:

Operation	Represents	Action
`+`	Set union	Takes two sets S1 and S2 and returns a set whose elements are either in S1 or in S2.
`-`	Difference	Takes two sets S1 and S2 and returns a set whose elements are in S1 but not in S2.
unary `-`	Negation	Takes a single set S and returns the difference of [`*`] and S.

The following example demonstrates these operations:

```
(idb) set $set1 = [:10, 15:18, 20:]
(idb) set $set2 = [10:16, 19]
(idb) set $set3 = $set1 + $set2
(idb) print $set3
[*]
(idb) print $set3 - $set2
[:9, 17:18, 20:]
(idb) print -$set2
[:9, 17:18, 20:]
```

Changing the Current Set with the `focus` Command

You can use the **`focus`** command to change the current process set, which is the set of processes whose debuggers receive the remote command entered at the root debugger:

```
focus_command

        : focus expression

        | focus all

        | focus
```

The first form of the command sets the current process set to the set resulting from the evaluation of the given expression. The second form sets the current process set to the set that includes all processes. The third form displays the current process set.

Working with Aggregated Messages

The root debugger collects the outputs from the leaf debuggers and presents you with an aggregated output. In most cases, this aggregation works fine, but it can be an impediment if you want to know the exact output from certain leaf debuggers.

To remedy this, the debugger assigns a unique number (called a `message_id`) to each aggregated message and saves the message in the `message_id_list`. You can use the following commands to inspect the message list and expand its entries:

```
show_aggregated_message_command

        : show aggregated message message_id_list

        | show aggregated message all

        | show aggregated message



message_id_list

        : expression {,...}
```

The first form of the command displays the aggregated messages in the list whose message IDs match the numbers specified in the `message_id_list`. The second form displays all the aggregated messages in the list. If no `message_id` is specified, the debugger shows the most recently added (newest) message.

```
expand_aggregated_message_command
```

```
    : expand aggregated message  message_id_list

    | expand aggregated message
```

This command expands the specified messages. If no `message_id` is specified, the debugger expands the most recently added (newest) message.

You can control the length of the message list using the `$aggregatedmsghistory` debugger variable. If you set this variable to the default (0), the debugger records as many messages as the system will allow.

Parallel Debugging Tips

This section contains the following tips for debugging parallel applications:

- Tip 1. How to Obtain Better Aggregate Outputs
- Tip 2. How to Synchronize Processes
- Tip 3. How to Find the Sources in a Parallel Debugging Session

Tip 1. How to Obtain Better Aggregate Outputs

If the debugger outputs are not aggregated as you would expect them to be, you can increase the value of the `$parallel_aggregatordelay` debugger variable, whose value is the expiration time (in milliseconds) for each of the aggregators when the aggregators have not received all the expected messages. Because the default value of the `$parallel_aggregatordelay` is 3000 milliseconds, you should not normally have a problem with the aggregation delay.

Tip 2. How to Synchronize Processes

If the processes become unsynchronized in the debugging session (for example, if you use the **`focus`** command on a subset of the total set and then use a **`next`** or some other motion command), the easiest way to get the processes back together is to use a **`cont to`** a future location where all processes have to go. The following example shows how the output from processes is not identical because different processes are at different locations in the program. Using the **`cont to`** command synchronizes the processes and aggregates the messages.

```
(idb) next
(idb)    [4:5,12] stopped at [int feedbackToDebugger(int, int, char*):17
0x120006bf4]
   [0:3,6:11] [3] stopped at [int feedbackToDebugger(int, int, char*):15
0x120006bf0]
   [4:5,12]      17   int pathSize = 1000;
   [0:3,6:11]      15   int i = 0;
(idb) l
(idb)    [0:3,6:11]      16   char path[1000];
   [4:5,12]      18   char hostname[1000];
   [0:3,6:11]      17   int pathSize = 1000;
   [4:5,12]      19   int hostnameSize = 1000;
   [0:3,6:11]      18   char hostname[1000];
   [4:5,12]      20
   [0:3,6:11]      19   int hostnameSize = 1000;
   [4:5,12]      21   volatile int debuggerAttached = 0;
   [0:3,6:11]      20
   [4:5,12]      22
   [0:3,6:11]      21   volatile int debuggerAttached = 0;
   [4:5,12]      23   gethostname(hostname,hostnameSize);
%3 [0:12]      [22;24]
   [0:3,6:11]      23   gethostname(hostname,hostnameSize);
   [4:5,12]      25   getcwd(path,pathSize);
   [0:3,6:11]      24
   [4:5,12]      26   strcat(path,"/");
   [0:3,6:11]      25   getcwd(path,pathSize);
   [4:5,12]      27   strcat(path,name);
   [0:3,6:11]      26   strcat(path,"/");
   [4:5,12]      28
   [0:3,6:11]      27   strcat(path,name);
   [4:5,12]      29   // Print myid pid into idbAttach.myid
   [0:3,6:11]      28
   [4:5,12]      30   sprintf(filename,"idbAttach.%d",myid);
   [0:3,6:11]      29   // Print myid pid into idbAttach.myid
   [4:5,12]      31   file = fopen(filename,"w");
   [0:3,6:11]      30   sprintf(filename,"idbAttach.%d",myid);
   [4:5,12]      32   if (file == NULL) {
   [0:3,6:11]      31   file = fopen(filename,"w");
   [4:5,12]      33     fprintf(stderr,"smg98: can't open %s for
%s\n",filename, "w");
   [0:3,6:11]      32   if (file == NULL) {
   [4:5,12]      34     exit(1)
   [0:3,6:11]      33     fprintf(stderr,"smg98: can't open %s for
%s\n",filename, "w");
   [4:5,12]      35   }
   [12]      36   fprintf(file," %ld %ld %s %s\n", myid, getpid(), hostname,
path);
   [12]      37   fclose(file);
   [12]      38
   [4:5]      36   fprintf(file," %ld %ld %s %s\n", myid, getpid(), hostname,
path);
   [0:3,6:11]      34     exit(1);
   [0:3,6:11]      35   }
   [4:5]      37   fclose(file);
   [0:3,6:11]      36   fprintf(file," %ld %ld %s %s\n", myid, getpid(),
hostname, path);
   [4:5]      38
(idb) cont to 36
   [0:13] stopped at [int feedbackToDebugger(int, int, char*):36 0x120006cb8]
   [0:13]      36   fprintf(file," %ld %ld %s %s\n", myid, getpid(), hostname,
path);
(idb) next
(idb)    [0:13] stopped at [int feedbackToDebugger(int, int, char*):37
0x120006d0c]
   [0:13]      37   fclose(file);
```

Tip 3. How to Find the Sources in a Parallel Debugging Session

The debugger will not be able to display the source lines if it cannot find the source file in the directory specified in the application binary file or in the directory in which the binary resides.

Specifying the `-I` option in the command line fixes the problem. Note that when launching a debugging session through mpirun, this option should be following the flag `-idb`. See Starting a Parallel Debugging Session for details on specifying idb options in a command line.

Alternatively, applying the **`use`** command or the **`map source directory`** command to all the leaf debuggers can overcome the problem as well. For example:

```
(idb) w
Source file not found or not readable, tried...
    ./cpi.c
    /home/user/Funct/bin/cpi.c
(Cannot find source file mpirun.c)
(idb) use /home/user/Funct/src
   [0:7] Directory search path for source files:
   [0:7]  . /home/user/Funct/bin /home/user/Funct/src
(idb) w
   [0:7]      20
   [0:7]      21 double f(double);
   [0:7]      22
   [0:7]      23 int main(int argc, char *argv[])
   [0:7]      24 {
   [0:7]      25     int done = 0, n, myid, numprocs, i;
   [0:7]      26     double PI25DT = 3.141592653589793238462643;
   [0:7]      27     double mypi, pi, h, sum, x;
   [0:7]      28     double startwtime = 0.0, endwtime;
   [0:7]      29     int  namelen;
```

Parallel Debugging Examples

Example 1

is a parallel debugging session started by mpirun.

```
% mpirun -dbg=idb -np 8 cpi
Intel(R) Debugger for Itanium®-based applications, Version XX
Reading symbolic information ...done
stopped at [void* MPIR Breakpoint(void):101 0x40000000000b3060]
    101 {
Process has exited
(idb)
   [0:7] Intel(R) Debugger for Itanium®-based applications, Version XX
   [0:7] ------------------
   [0:7] object file name: /home/user/examples/cpi
   [0:7] Reading symbolic information ...   [0:7] done
%1 [0:7] Attached to process id [30596;30636]  ....
   [1:7] stopped at [ 0x20000000001ef962]
   [0] stopped at [void* MPIR Breakpoint(void):101 0x40000000000b3060]
   [0]     101 {
(idb)
   [0:7] stopped at [int main(int, char**):20 0x4000000000003520]
   [0:7]      20     MPI_Init(&argc,&argv);
(idb)
   [0:7]      16     double startwtime = 0.0, endwtime;
   [0:7]      17     int  namelen;
   [0:7]      18     char processor name[MPI MAX PROCESSOR NAME];
   [0:7]      19
   [0:7] >    20     MPI_Init(&argc,&argv);
   [0:7]      21     MPI Comm size(MPI COMM WORLD,&numprocs);
   [0:7]      22     MPI Comm rank(MPI COMM WORLD,&myid);
   [0:7]      23     MPI_Get_processor_name(processor_name,&namelen);
   [0:7]      24
(idb) stop in f
```

```
(idb)
   [0:7] [#1: stop in double f(double) ]
(idb) focus [0:3]
[0:3]>
[0:3]>  cont
[0:3]>  Process 3 on nht6005.spt.intel.com
Process 2 on nht6005.spt.intel.com
Process 0 on nht6005.spt.intel.com
Process 1 on nht6005.spt.intel.com
   [0:3] [1] stopped at [double f(double):7 0x4000000000003390]
   [0:3]       7 {
[0:3]>  where
[0:3]>
   [0:3] >0  0x4000000000003390 in f(a=<no value>) "cpi.c":7
%2 [0:3] #1  0x4000000000003a30 in main(argc=0, argv=0x[0;80000fffffffba7c])
"cpi.c":51
   [0:3] #2  0x20000000000906b0 in /lib/libc.so.6.1
   [0:3] #3  0x4000000000003220 in  start(...) in /home/user/examples/cpi
[0:3]>  focus [4:7]
[4:7]>
[4:7]>  cont
[4:7]>  Process 7 on nht6005.spt.intel.com
Process 4 on nht6005.spt.intel.com
Process 6 on nht6005.spt.intel.com
Process 5 on nht6005.spt.intel.com
   [4:7] [1] stopped at [double f(double):7 0x4000000000003390]
   [4:7]       7 {
[4:7]>  where
[4:7]>
   [4:7] >0  0x4000000000003390 in f(a=<no value>) "cpi.c":7
%3 [4:7] #1  0x4000000000003a30 in main(argc=0, argv=0x[0;80000fffffffba7c])
"cpi.c":51
   [4:7] #2  0x20000000000906b0 in /lib/libc.so.6.1
   [4:7] #3  0x4000000000003220 in  start(...) in /home/user/examples/cpi
[4:7]>  focus [*]
[0:7]>
[0:7]>  next
[0:7]>
   [0:7] stopped at [double f(double):8 0x40000000000033b1]
   [0:7]       8     return (4.0 / (1.0 + a*a));
[0:7]>  where
[0:7]>
%4 [0:7] >0  0x40000000000033b1 in
f(a=[0.0050000000000000001;0.074999999999999997]) "cpi.c":8
%5 [0:7] #1  0x4000000000003a30 in main(argc=1,
argv=0x[80000fffffffb768;6000000000014a50]) "cpi.c":51
   [0:7] #2  0x20000000000906b0 in /lib/libc.so.6.1
   [0:7] #3  0x4000000000003220 in _start(...) in /home/user/examples/cpi
[0:7]>  show aggregated message
%1 [0:7] Attached to process id [30596;30636]  ....
%2 [0:3] #1  0x4000000000003a30 in main(argc=0, argv=0x[0;80000fffffffba7c])
"cpi.c":51
%3 [4:7] #1  0x4000000000003a30 in main(argc=0, argv=0x[0;80000fffffffba7c])
"cpi.c":51
%4 [0:7] >0  0x40000000000033b1 in
f(a=[0.0050000000000000001;0.074999999999999997]) "cpi.c":8
%5 [0:7] #1  0x4000000000003a30 in main(argc=1,
argv=0x[80000fffffffb768;6000000000014a50]) "cpi.c":51
[0:7]>
[0:7]>  expand aggregated message 1
%1 [0:7] Attached to process id [30596;30636]  ....
 [3] Attached to process id 30612  ....
 [2] Attached to process id 30606  ....
 [0] Attached to process id 30596  ....
 [1] Attached to process id 30600  ....
 [4] Attached to process id 30618  ....
 [5] Attached to process id 30624  ....
 [7] Attached to process id 30636  ....
 [6] Attached to process id 30630  ....
[0:7]>  disable 1
[0:7]>
[0:7]>  cont
[0:7]>  pi is approximately 3.1416009869231249, Error is 0.0000083333333318
wall clock time = 69.300781
   [0:7] Process has exited with status 0
[0:7]>  quit
```

The following are explanatory notes from the previous example:

Component of Example	Meaning
`-np 8`	This parallel session creates 8 processes.
`[0:7]`	This is a message from processes 0 to 7.
`%1`	This aggregated message contains messages with differing portions (in this case, the process id's are different from process to process), and 1 is the message id.
`focus [0:3]`	This **`focus`** command sets the current process set to include processes 0, 1, 2, and 3.
`[0:3]>`	This prompt shows the current process set.
`show aggregated message`	This **`show aggregated message`** command displays all the aggregated messages saved in the message list.
`expand aggregated message 1`	This **`expand aggregated message`** command expands the aggregated message with message id 1.

Example 2

demonstrates how to start a parallel debugging session with prun.

```
% idb -parallel `which prun` -n 16 -N 8 ./cpi
Intel(R) Debugger for Itanium®-based applications, Version 7.0,
Build 20021118
Reading symbolic information ...done
stopped at [void  rms breakpoint(void):2150 0x20000000001913e0]
Source file not found or not readable, tried...
    ./loader.cc
    /usr/bin/loader.cc
(Cannot find source file loader.cc)
stopped at [void  rms breakpoint(void):2150 0x20000000001913e0]
Source file not found or not readable, tried...
    ./loader.cc
    /usr/bin/loader.cc
(Cannot find source file loader.cc)
Process has exited
(idb)
```

```
   [0:15] Intel(R) Debugger for Itanium®-based applications,
Version 7.0, Build 20021118
   [0:15] ------------------
   [0:15] object file name: cpi
   [0:15] Reading symbolic information ...   [0:15] done
   [0:15]      13     int done = 0, n, myid, numprocs, i;
(idb) where
(idb)
   [0:15] >0  0x4000000000000c60 in  start(...) in cpi
(idb) stop in main
(idb)
   [0:15] [#1: stop in int main(int, char**) ]
(idb) cont
(idb)
   [0:15] [1] stopped at [int main(int, char**):13 0x4000000000000f52]
   [0:15]      13     int done = 0, n, myid, numprocs, i;
```

In this example, the first couple of messages about not being able to find the file `loader.cc` can be ignored; they are caused by the fact that this file usually does not exist on a production system.

Using the `mpirun_dbg.idb` Startup File

The latest mpich distribution should come with the idb startup file `mpirun_dbg.idb`. If it does not, or if you are using an older distribution of mpich, you can create the idb startup file by saving the following script as `mpirun_dbg.idb` in the directory in which `mpirun` resides:

```
#! /bin/sh
cmdLineArgs=""
p4pgfile=""
p4workdir=""
prognamemain=""
p4ssport=""
processedCmdLineArgs=""
#
# Extract -p4ssport info from the string passed in via -cmdlineargs.
#
function processCmdLineArgs()
{
    while [ 1 -le $# ] ; do
        arg=$1
        shift
        case $arg in
            -p4ssport)
               p4ssport="-p4ssport $1"
               shift
               ;;
            *)
               processedCmdLineArgs="$processedCmdLineArgs $arg"
               ;;
esac
    done
}
while [ 1 -le $# ] ; do
  arg=$1
  shift
  case $arg in
     -cmdlineargs)
```

```
        cmdLineArgs="$1"
        shift
;;
     -p4pg)
        p4pgfile="$1"
shift
;;
     -p4wd)
        p4workdir="$1"
shift
;;
     -progname)
        prognamemain="$1"
shift
;;
  esac
done
#
#
# Need to `eval echo $cmdLineArgs` to undo evil quoting done in mpirun.args
#
processCmdLineArgs `eval echo $cmdLineArgs`
#
if [ -n "$IDB HOME" ] ; then
    ldbdir=$IDB HOME
    idb=$ldbdir/idb
    if [ -f $ldbdir/idb.cat ] && [ -r $ldbdir/idb.cat ] ; then
      if [ -n "$NLSPATH" ];  then
 nlsmore=$NLSPATH
      else
 nlsmore=""
      fi
      NLSPATH=$ldbdir/$nlsmore
    fi
else
    idb="idb"
fi
#
$idb -parallel $prognamemain -p4pg $p4pgfile -p4wd $p4workdir -mpichtv
$p4ssport $processedCmdLineArgs
```

Debugging Remote Applications. Overview.

Intel® IDB supports debugging of applications running on remote machines under `gdbserver` control on IA-32 processors.

If an application you need to debug is not on the same machine that you are working on, use a special remote agent, `gdbserver`, on the target machine, and IDB on the host machine. It is important that you use the same binaries on the target and host machines, but on the target machine you can use stripped bits.

The `gdbserver` utility is a part of the GNU* debugger, GDB*. If you do not have the `gdbserver` installed, you need to install it.

GDB and `gdbserver` communicate using a Remote Serial Line protocol. IDB and `gdbserver` can be connected via TCP/IP connection.

This section shows you how to:

- Start a remote debugging session
- Connect to the remote process
- End a remote debugging session

Starting a Remote Debugging Session

To start your remote application under debugger control, use the GNU* `gdbserver` remote agent. The following example shows how to invoke `gdbserver` remote agent:

```
% gdbserver [host]:port application [application arguments]
```

When the `gdbserver` launches an application, the terminal window displays service information about the started application and the connection. Issue the following command in another shell session to connect to the remote `gdbserver`:

```
% idb [idb_options] -remote [host]:port application  [application arguments]
```

or

```
% idb [idb_options] -remote [host]:port
```

Connecting to the Remote Process

After `gdbserver` has loaded a program, use the **`attach_remote`** (dbx) or the **`target_remote`** (gdb) command to connect to the remote process:

DBX Mode

```
attach_remote_command

        : attach remote connection-info [local-copy-of-remote-file]
```

The debugger connects to the remote agent and loads debug information from a local copy of the debuggee.

GDB Mode

```
target_remote_command

         : target remote connection-info [local-copy-of-remote-file]
```

IDB connects to the remote agent.

To load the debugging information from the file, use the `symbol-file` command:

symbol-file

: **symbol-file** [*filename*]

If the *filename* matches the *connection-info* ([[*protocol*:]*host*]:*port*), use the protocol and the port. Otherwise use the *filename* as a path to the serial device supporting the interface.

local-copy-of-remote-filename

: **filename**

This file provides the debugging information on the process running on the remote system. It does not need to be exactly the same file that is running on the remote system.

Ending the Remote Debugging Session

Use either the **detach** or the **disconnect** command to break the connection between the IDB and the remote agent:

`detach` command

When you have finished debugging the remote program, use the **detach** command to free it from IDB control. In most cases, the program will resume its execution, but this depends on a particular gdbserver. IDB can connect to another program, after detach is issued.

detach_remote_command

: **detach**

`disconnect` command

The **disconnect** command behaves the same as **detach**, except the program is not resumed. The program will wait for IDB (this instance or another) to reconnect to continue debugging. IDB can connect to another program after **disconnect** is issued.

disconnect_remote_command

: **detach**

Appendixes

The Appendixes contain information about debugger variables, debugger aliases, and give `corefile_listobj.c` example and array navigation examples.

List of Debugger Variables

The debugger has the following predefined variables. Conventionally, an Intel IDB variable name is an identifier with a leading dollar sign (`$`).

Variable	Default Setting	Description
`$aggregatedmsghistory`	0	Controls the length of the aggregated message list. If set to the default (0), the debugger records as many messages as the system will allow.
`$ascii`	1	Displays whether prints ASCII or all ISO Latin-1. (see $lc_ctype)
`$beep`	1	Beeps on illegal command line editing.
`$catchexecs`	0	Stops execution on program exec.
`$catchforkinfork`	0	Notifies you as soon as the forked process is created (otherwise you are notified when the call finishes).
`$catchforks`	0	Notifies you on program fork and stops child.
`$childprocess`	0	When the debugger detects a fork, it assigns the child process ID to `$childprocess`.
`$curcolumn`	0	Displays the current column number if that information is available; 0 otherwise.
`$curevent`	0	Displays the current breakpoint number.
`$curfile`	(null)	Displays the current source file.
`$curfilepath`	(null)	Displays the current source file access path.
`$curline`	0	Displays the current source line.
`$curpc`	0	Displays the current point of program execution.
`$curprocess`	0	Displays the current process ID.

`$cursrcline`	0	Displays the last source line at end of most recent source listing.
`$cursrcpc`	0	Displays the PC address at end of most recent machine code listing.
`$curthread`	0	Displays the current thread ID.
`$dbxoutputformat`	0	Displays various data structures in `dbx` format.
`$dbxuse`	0	Replaces current use paths.
`$decints`	0	Displays integers in decimal radix.
`$doverbosehelp`	1	Displays the help menu front page.
`$editline`	1	Enables command line editing.
`$eventecho`	1	Echoes events with event numbers.
`$exitonterminationofprocesswithpid`	None	If set to process ID (`pid`), when that process terminates, the debugger exits.
`$float80bit`	1	If set to 0 (the default), prints 128-bit floating point numbers normally;if set to a non-zero value, prints 128-bit floating point numbers as 128-bit containsers holding 80-bit floating point numbers.
`$floatshrinking`	1	If set to the default (1), the debugger prints binary floating point numbers using the shortest possible decimal number. If set to 0, the debugger prints the decimal number which is the closest representation in the number of decimal digits available of the internal binary number.
`$framesearchlimit`	0	Defines the maximum number of call frames by which to extend normal language-based identifier lookups.
`$funcsig`	1	Displays function signature at breakpoint.
`$gdb_compatible_output`	0	Makes the IDB's output to be fully compatible with GDB's output.
`$givedebughints`	1	Displays hints on debugger features.
`$hasmeta`	0	Interprets multibyte characters.
`$hexints`	0	Displays integers in hex radix.
`$highpc`	(internal debugger	Returns the highest address

	variable)	associated with "function".
`$historylines`	20	Defines the number of commands to show for **`history`**.
`$indent`	1	Prints structures with indentation.
`$lang`	"None"	Defines the programming language of current routine.
`$lasteventmade`	0	Displays the number of last (successful) breakpoint definition.
`$lc_ctype`	result of setlocale(LC_CTYPE, 0L)	Defines the current locale information. If set, passes the value through setlocale() and becomes the result. "" is passed as 0L.
`$listwindow`	20	Displays the number of lines to show for **`list`**.
`$main`	"main"	Displays the name of the first routine in the program.
`$maxstrlen`	128	Defines the largest string to print fully.
`$memorymatchall`	0	When set to non-zero, displays all memory matches in the specified range. Otherwise, only the first memory match is displayed.
`$octints`	0	Displays integers in octal radix.
`$overloadmenu`	1	Prompts for choice of overloaded C++ name.
`$page`	1	Paginates debugger terminal output.
`$pagewindow`	0	Defines the number of lines per output page. The default of 0 causes the debugger to query the terminal for the page size.
`$parallel_branchingfactor`	8	Specifies the factor used to build the *n*-nary tree and determine the number of aggregators in the tree.
`$parallel_aggregatordelay`	3000 milliseconds	Specifies the length of time that aggregators wait before they aggregate and send messages down to the next level when not all the expected messages have been received.
`$parentprocess`	0	When the debugger detects a fork, it assigns the parent process ID to `$parentprocess`.

`$pimode`	0	Echoes input to log file on **`playback input`**.
`$prompt`	"(idb) "	Specifies debugger prompt.
`$readtextfile`	0	If set to non-zero, instructions are read from the text area of the binary file rather than from the memory image.
`$regstyle`	1	Controls the format of register names during disassembly. Valid settings are: • 0 = compiler names, for example, t0, ra, or zero. • 1 = hardware names, for example, r1, r26, or r31. • 2 = assembly names, for example, $1, $26, or $31.
`$repeatmode`	1	Repeats previous command when you press the Enter key.
`$reportsotrans`	0	Report when an event was changed because a shared object was either opened or closed.
`$showlineonstartup`	0	Displays the first executable line in `main`.
`$showwelcomemsg`	1	Displays welcome message at startup time.
`$stackargs`	1	Shows arguments in the call stack if 1.
`$stack_levels`	50	Controls the number of call stack output levels.
`$statusargs`	1	Prints breakpoints with parameters if 1.
`$stepg0`	0	Steps over routines with minimal symbols.
`$stoponattach`	0	Stops the running process on **`attach`**.
`$stopparentonfork`	0	Stops parent process execution on fork. When set to a non-zero value, this variable instructs the debugger to stop the parent process after it forks a child process. The child process continues to run if `$catchforks` is not set, otherwise stops. The default is 0.

`$symbolsearchlimit`	100	Specifies the maximum number of symbols that will be returned by the **`whereis`** command for a regular expression search. The default value is 100; a value of 0 indicates no limit.
`$threadlevel`	pthreads	Specifies POSIX threads or pthreads.
`$usedynamictypes`	1	Evaluates using C++ static or dynamic type.
`$verbose`	0	Produces even more output.

Debugger Aliases

The debugger has the following predefined aliases:

```
(idb) alias
F1 print
F2 print 'F2 executes the command "F2 selected-text" - define alias F2'
F3 print 'F3 executes the command "F3 selected-text" - define alias F3'
S next
Si nexti
W list $curline - 10:20
a assign
att attach
b stop at
bp stop in
c cont
d delete
det detach
e file
exit quit
f func
g goto
h history
j status
l list
lfi(x) set $lfitmp = x ; $lfitmp, $highpc($lfitmp) / i ; unset $lfitmp
li ($cursrcpc)/10 i
n next
ni nexti
p print
pb printb
pd printd
pi printi
plist show process all
po printo
pr printregs
ps printf "%s",
pt printt
px printx
q quit
r rerun
ri record input
ro record output
s step
```

```
si stepi
source playback input
sw switch
switch process
t where
tlist show thread
ts where thread all
tset thread
tstack where thread all
u list $curline - 9:10
w list $curline - 5:10
wi ($curpc - 7)/10 i
wm watch memory
wv watch variable
```

`corefile_listobj.c` Example

You can use the following example as an alternative to the **`listobj`** command for cases in which the debugger cannot be run on the original system. See the Transporting Core Files section for more information.

```
/*
  cc corefile listobj.c -lxproc -o corefile listobj
 */
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <errno.h>
typedef unsigned long vma t;
/* core file format */
#include <sys/user.h>
#include <sys/core.h>
/* dynamic loader hookup */
#include <loader.h>
typedef int (*ldr reader func)(vma t from,
                               void * to,
                               long nbytes,
                               int is string);
extern pid_t ldr_core_process();
extern int ldr set core reader(ldr reader func reader);
/*******************************************************************/
static FILE *                   corefile;
static struct core filehdr      corehdr;
static int                      nsections;
static struct core_scnhdr *     section_headers;
int
open corefile(const char * corename)
{
  size t nread;
  corefile = fopen(corename, "rb");
  if (!corefile) {
    perror("Opening corefile");
    return -1;
  }
  nread = fread(&corehdr, sizeof(corehdr), 1, corefile);
  if (nread != 1) {
    perror("fread() of corefile header");
    return -1;
  }
  if (strncmp(corehdr.magic, "Core", 4) != 0) {
    fprintf(stderr, "Corefile header magic is not \"Core\"\n");
    return -1;
  }
  nsections = corehdr.nscns;
  section headers = calloc(nsections, sizeof(section headers[0]));
  if (!section headers) {
    perror("Allocating corefile section headers");
    return -1;
  }
  nread = fread(section headers, sizeof(section headers[0]),
                nsections, corefile);
  if (nread != nsections) {
    perror("fread() of corefile section headers");
    return -1;
  }
  return 0;
}
static int
section type has memory(int type)
{
  switch (type) {
  case SCNTEXT: case SCNDATA: case SCNRGN: case SCNSTACK:
    return 1;
  case SCNREGS: case SCNOVFL:
  default:
    return 0;
  }
}
static int
read from corefile(vma t from,
                   void * to,
                   long nbytes,
                   int is string)
{
  vma t getter = from;
  char * putter = (char *) to;
  long to go = nbytes;
  int secnum;
  size t nxfer;
try for more:
  while (to go > 0) {
    for (secnum = 0; secnum < nsections; secnum += 1) {
      if (section type has memory(section headers[secnum].scntype)) {
        vma t vaddr = (vma t) section headers[secnum].vaddr;
        vma t size  = (vma t) section headers[secnum].size;
        if (vaddr <= getter && getter < vaddr+size) {
          vma t this time = (size < to go ? size : to go);
          long file offset = section headers[secnum].scnptr+(getter-vaddr);
          if (fseek(corefile, file offset, SEEK SET) != 0) {
            perror("fseek() for corefile read");
            return -1;
          }
          nxfer = fread(putter, 1, this time, corefile);
          if (nxfer != this time) {
            perror("fread() of corefile data ");
            return -1;
          }
          to go -= this time;
          getter += this time;
          putter += this time;
          goto try for more;
        }
      }
    }
    fprintf("Couldn't find core address for %#lx\n", getter);
    return -1;
  }
  return 0;
}
int
main(int argc, char* argv[])
{
  pid t process;
  if (argc != 2) {
    fprintf(stderr, "Usage is %s <corefile>\n", argv[0]);
    return 1;
  }
  if (open corefile(argv[1]) < 0)
    return -1;
  process = ldr core process();
  ldr set core reader(read from corefile);
  if (ldr xattach(process) < 0) {
    perror("Attaching to corefile");
    return 1;
  } else {
    ldr module t mod id = LDR NULL MODULE;
    ldr module info t info;
    size t ret size;
    while (1) {
      if (ldr next module(process, &mod id) < 0) {
        perror("ldr next module");
        return 1;
      }
      if (mod id == LDR NULL MODULE)
        break;
      if (ldr inq module(process, mod id, &info,
                         sizeof(info), &ret size) < 0) {
        perror("ldr inq module");
        return 1;
      }
      printf("%s\n", info.lmi name);
    }
    ldr xdetach(process);
    return 0;
  }
}

/*
  cc corefile listobj.c -lxproc -o corefile listobj
 */
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <errno.h>
typedef unsigned long vma t;
/* core file format */
#include <sys/user.h>
#include <sys/core.h>
/* dynamic loader hookup */
#include <loader.h>
typedef int (*ldr reader func)(vma t from,
                               void * to,
                               long nbytes,
                               int is string);
extern pid t ldr core process();
extern int ldr set core reader(ldr reader func reader);
/*******************************************************************/
static FILE *                   corefile;
static struct core filehdr      corehdr;
static int                      nsections;
static struct core scnhdr *     section headers;
int
open corefile(const char * corename)
{
  size t nread;
  corefile = fopen(corename, "rb");
  if (!corefile) {
    perror("Opening corefile");
    return -1;
  }
  nread = fread(&corehdr, sizeof(corehdr), 1, corefile);
  if (nread != 1) {
    perror("fread() of corefile header");
    return -1;
  }
  if (strncmp(corehdr.magic, "Core", 4) != 0) {
    fprintf(stderr, "Corefile header magic is not \"Core\"\n");
    return -1;
  }
  nsections = corehdr.nscns;
  section headers = calloc(nsections, sizeof(section headers[0]));
  if (!section headers) {
    perror("Allocating corefile section headers");
    return -1;
  }
  nread = fread(section headers, sizeof(section headers[0]),
                nsections, corefile);
  if (nread != nsections) {
    perror("fread() of corefile section headers");
    return -1;
  }
  return 0;
}
static int
section type has memory(int type)
{
  switch (type) {
  case SCNTEXT: case SCNDATA: case SCNRGN: case SCNSTACK:
    return 1;
  case SCNREGS: case SCNOVFL:
  default:
    return 0;
  }
}
static int
read from corefile(vma t from,
                   void * to,
                   long nbytes,
                   int is string)
{
  vma t getter = from;
  char * putter = (char *) to;
  long to go = nbytes;
  int secnum;
  size t nxfer;
try for more:
  while (to go > 0) {
    for (secnum = 0; secnum < nsections; secnum += 1) {
      if (section type has memory(section headers[secnum].scntype)) {
        vma t vaddr = (vma t) section headers[secnum].vaddr;
        vma t size  = (vma t) section headers[secnum].size;
        if (vaddr <= getter && getter < vaddr+size) {
          vma t this time = (size < to go ? size : to go);
          long file_offset = section_headers[secnum].scnptr+(getter-vaddr);
          if (fseek(corefile, file offset, SEEK SET) != 0) {
            perror("fseek() for corefile read");
            return -1;
          }
          nxfer = fread(putter, 1, this time, corefile);
          if (nxfer != this time) {
            perror("fread() of corefile data ");
            return -1;
          }
          to go -= this time;
          getter += this time;
          putter += this time;
          goto try for more;
        }
      }
    }
    fprintf("Couldn't find core address for %#lx\n", getter);
    return -1;
  }
  return 0;
}
int
main(int argc, char* argv[])
{
  pid t process;
  if (argc != 2) {
    fprintf(stderr, "Usage is %s <corefile>\n", argv[0]);
    return 1;
  }
  if (open corefile(argv[1]) < 0)
    return -1;
  process = ldr core process();
  ldr set core reader(read from corefile);
  if (ldr xattach(process) < 0) {
    perror("Attaching to corefile");
    return 1;
  } else {
    ldr module t mod id = LDR NULL MODULE;
    ldr module info t info;
    size t ret size;
    while (1) {
      if (ldr next module(process, &mod id) < 0) {
        perror("ldr next module");
        return 1;
      }
      if (mod id == LDR NULL MODULE)
        break;
      if (ldr inq module(process, mod id, &info,
                         sizeof(info), &ret size) < 0) {
        perror("ldr inq module");
        return 1;
      }
      printf("%s\n", info.lmi name);
    }
    ldr xdetach(process);
    return 0;
  }
}
```

Array Navigation Example

The debugger provides parameterized aliases and debugger variables of arbitrary types. Clever use of these can do almost any list traversal.

For example, here is how to navigate an array:

```
alias elt(e ) "{ p e  }"
alias pa0(a)  "{ set $a = &a[0]; set $i = 0; elt($a[$i]); set $i = $i+1 }"
alias pan     "{ elt($a[$i]); set $i = $i+1 }"
pa0
pan
pan
pan
%idb a.out
...
(idb) alias elt(e ) "{ p e  }"
(idb) alias a0(a)   "{ set $a = &a[0]; set $i = 0; elt($a[$i]); set $i = $i+1
}"
(idb) alias pan     "{ elt($a[$i]); set $i = $i+1 }"
...
(idb) pa0(a)
struct S {
  next = 0x140000178;
}
(idb) pan
struct S {
  next = 0x140000180;
}
(idb)
struct S {
  next = 0x140000188;
}
(idb)
struct S {
  next = 0x140000190;
}
```

Index