GDB Tutorial

University of Waterloo

Version 1.0

Caroline Kierstead and Peter A. Buhr ©*2002

April 1, 2002

Contents

1 Introduction

This tutorial is designed to give a very basic introduction to the GNU Source-Level Debugger. It is organized with a basic introduction to the debugger commands and then two programs with several errors are debugged using the debugger. By working through the exercises, basic concepts are introduced and can be practiced. The tutorial is not intended as a complete instructional guide. A manual on GDB is available.

GDB can be used in and out of the Emacs environment. It is recommended that GDB be run within Emacs as it is easier to trace the execution of a program. While this tutorial uses GDB within Emacs, additional instructions are given on how to run GDB outside of Emacs; it is assumed that you are familiar with Emacs. As well, you should be familiar with the UNIX environment. (UNIX consultants are available in MC3011.)

Throughout the tutorial, the following symbols are used:

$\Rightarrow$ This symbol indicates that you are to perform the action marked by the arrow.

$\bowtie$ This symbol indicates that the section explains a concept that may be unfamiliar even if you have some previous experience using a computer (e.g., DOS). Make sure you understand this concept before advancing in the tutorial.

2 Before Using GDB

Before starting any debugger it is important to understand what you are looking for. A debugger does not debug your program for you, it merely helps in the debugging process. Therefore, you must have some idea about what is wrong with a program before starting to look or you will simply waste your time. Furthermore, you should not rely solely on a debugger to debug a program. You may work on a system without a debugger or the debugger may not work for certain kinds of problems. This section discusses traditional approaches to debugging.

2.1 Debug Print Statements

The best way to debug a program is to *start* by inserting debug print statements as the program is written. It does take a few extra minutes to include debug print statements, but the alternative is wasting hours trying to figure out what the program is doing.

The two aspects of a program that you need to know about are: where the program is executing and what values it is calculating. Debug print statements show the flow of control through a program and print out intermediate values. For example, every routine should have a debug print statement at the beginning and end, as in:

```
int p( ... ) {
    // declarations
    cerr << "Enter p( ... )\n" << parameter variables << endl;
    ...
    cerr << "Exit p: ...\n" << return value(s)  << endl;
    return i;
} // p
```

This results in a high-level audit trail of where the program is executing and what values are being passed around. To get finer resolution of a program's execution, more debug print statements can be included in important control structures, as in:

```
if ( a > b ) {
    cerr << "a > b" << endl ;                          // debug print
    for ( ... ) {
        cerr << "x=" << x << ", y=" << y << endl;      // debug print
        ...
    } // for
} else {
    cerr << "a <= b" << endl;                          // debug print
    ...
} // if
```

By examining the control paths the program takes and the intermediate values calculated, it is possible to determine if the program is executing correctly.

Unfortunately, debug print statements can generate enormous amounts of output, far more than is useful.

> It is of the highest importance in the art of detection to be able to recognize out of a number of facts which are incidental and which vital.
>
> *Sherlock Holmes, The Reigate Squires*

So gradually comment out debug statements as parts of the program begin to work to remove clutter from the output, but do not delete them until the program works completely. You never know when they will be needed again.

In general, when you go for help, either from your instructor or an advisor, you should have debug print statements in your program. Their presence shows that you have at least attempted to track the problem yourself. If you have no debug print statements, you may be told to come back when you have! Finally, debug print statements never appear in the program you hand in for marking. They are only there to help get the program working.

2.2 Errors

Debug print statements do not prevent errors, they simply aid in finding the errors; your programs will still have errors. What you do about an error depends on the kind of error. Errors fall into two basic categories:

Syntax Errors are errors in the arrangement of the basic tokens of the programming language. These errors correspond to spelling or punctuation errors when writing in a human language. Fixing syntax errors is usually straight forward especially if the compiler generates a meaningful error message. Always *read* the error message carefully and check the statement in error.

> You see (Watson), but do not observe.
>
> *Sherlock Holmes, A Scandal in Bohemia*

Watch out for the following general errors:

- Forgetting a closing " or */. The remainder of the program is *swallowed* as part of the character string or comment.
- Missing a { or }. If the program is indented and closing braces are appropriately commented, it is easy to find the missing block delimiter.
- Putting a semi-colon before the keyword word else.

Semantic Errors are errors in the behaviour or logic of the program. These errors correspond to incorrect meaning when writing in a human language. Semantic errors are harder to find and fix than syntax errors. Often a semantic or execution error message from the runtime libraries only tells why the program stopped not what caused the error. Usually, you must work backwards from the error to determine the cause of the problem.

> In solving a problem of this sort, the grand thing is to able to reason backwards. This is very useful accomplishment, and a very easy one, but people do not practise it much. In the everyday affairs of life it is more useful to reason forward, and so the other comes to be neglected.
>
> *Sherlock Holmes, A Study in Scarlet*

For example, this is an infinite loop but there is nothing wrong with the loop, it is the initialization that is wrong.

```
i = 10;
while ( i != 5 ) {
    . . .
    i += 2;
} // while
```

In general, when a program stops with a semantic error, the statement that caused the error is not usually the one that must be fixed.

Watch out for the following general errors:

- Forgetting to assign a value to a variable before using it in an expression.

- Using an invalid subscript or pointer value.

Finally, if a statement appears not to be working properly, but looks correct, check the syntax.

```
if ( a = b ) {
    cerr << "a == b" << endl;
} // if
```

> When you have eliminated the impossible whatever remains, however improbable must be the truth.
> *Sherlock Holmes, Sign of Four*

An interactive debugger effectively allows debug print statements to be added and removed to/from a program dynamically, as will be seen shortly. However, a good programmer usually uses a combination of debug print statements and an interactive debugger when debugging a complex program.

3 Getting Started

You need the following files for this tutorial:

⇒ Copy the following files to some location under your home directory:

```
/u/cssystems/examples/gdb_basic.cc
/u/cssystems/examples/gdb_q1.cc
/u/cssystems/examples/gdb_q2.cc
/u/cssystems/examples/data1
/u/cssystems/examples/data2
```

To start GDB within Emacs, enter the command M-x gdb <Return> and you will be prompted in the mini-buffer for the name of an executable file to be debugged. To start GDB outside of Emacs, the general format of the command is:

gdb *executable-file-name*

4 Using GDB

⇒ Start up Emacs and create a buffer containing the file gdb_basic.cc.

This program's sole purpose is to demonstrate the debugger commands; the program itself does nothing in particular (see Appendix A).

⇒ Compile the program using the command: g++ -g gdb_basic.cc

The -g option includes additional information for symbolic debugging.

⇒ Enter the command: M-x gdb <Return>.
⇒ Enter the name of the program's executable, ./a.out, after the prompt in the mini-buffer and press <Return>.

GDB responds with a number of messages and with the GDB prompt, (gdb).

4.1 Getting Help

To obtain help in GDB use the command help. The information is divided by topic. (NOTE: All commands in GDB are executed when <Return> is pressed.)

⇒ Enter the command: help

GDB responds with the following list of command classes:

```
List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the program
user-defined -- User-defined commands

Type "help" followed by a class name for a list of commands in that class.
Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
```

4.2 Starting a Program

⇒ Enter the command: help run

GDB responds with:

```
Start debugged program.  You may specify arguments to give it.
Args may include "*", or "[...]"; they are expanded using "sh".
Input and output redirection with ">", "<", or ">>" are also allowed.

With no arguments, uses arguments last specified (with "run" or "set args").
To cancel previous arguments and run with no arguments,
use "set args" without arguments.
```

⇒ Enter the command: run

GDB responds with a message indicating which object file it is executing and that it executed normally. When there are no errors in a program, running it via GDB is the same as running it in a shell.

4.3 Setting a Breakpoint

In order to trace the execution of the program, *breakpoints* are required. A breakpoint causes suspension of the program's execution when that location is reached. Breakpoints can be set on routines, line numbers and addresses. They are numbered consecutively from 1 up and can be enabled or disabled as required by using the enable, disable, and delete commands.

In order to allow the execution to be traced, set a breakpoint in the first routine that is executed, main.

⇒ Enter the command: break main or b main

GDB responds with:

```
(gdb) break main
Breakpoint 1 at 0x10624: file gdb_basic.cc, line 24.
```

indicating that breakpoint number 1 has been set at location 0x10624, which is line 24 of the file gdb_basic.cc. If a program is not compiled with the -g flag, only the address location is given.

⇒ Enter the command: run

The program is restarted (it was run once already) and execution continues until the first breakpoint is reached. The breakpoint is at the first executable line within main, line 24. Your screen will have split horizontally and the source code *about* to be executed is displayed with an arrow (=>). (The arrow may cover the first two characters of the current line of code.)

4.4 Listing Source Code

When not executing GDB in Emacs, the source file does not appear in another window. To list source code around the execution location, use the list command.

⇒ Enter the command: list 24 or l 24.

GDB responds with:

```
(gdb) list 24
19              return r;
20          }
21  };
22
23  int main() {
24      int r, x = 1;
25      mary m;
26      r = m.bar( x );
27      return 0;
28  }
```

4.5 Printing Variables

The print command is used to print the values of variables accessible in the current routine, plus all those whose declared in the global/external area.

⇒ Print the contents of variable r by entering the command: print r or p r.

GDB responds with:

```
(gdb) p r
$1 = 0
```

The value of r is 0. The $1 is the name of a history variable (like history variables in a shell). The name $1 can be used in subsequent commands to access previous values of r.

The value to be printed may be any C++ expression, so if the variable is a pointer, the pointer and the value it references are printed with the commands:

```
(gdb) print p
(gdb) print *p
```

(Unfortunately, a list of variables is taken to be a C++ language "comma expression" and only the last value in the list is printed.)

During debugging, it is often necessary to print certain variables each time the program stops at a breakpoint. This requires typing in a series of print commands each time the program stop. The display command is like the print command, and in addition, it prints the specified variable each time the program stops.

⇒ Enter the command: display r or disp r

GDB responds with:

```
(gdb) disp r
1: r = 0
```

⇒ Enter the command: display x or disp x

Each displayed variable is numbered, in this case, r is numbered 1 and x is numbered 2. The number is used to stop displaying a variable using the undisplay *n* command.

Note: variables r and x have not yet been initialized. The values displayed are that of the memory where the variables are allocated. Sometimes the values are 0, and sometimes they are not. *Do not assume that the values are always 0.*

4.6 Controlling Execution

Once a breakpoint is reached, execution of the program can be continued in several ways.

step [n]	Execute the next n lines of the program and stop. If n is not present, 1 is assumed. If the next line is a routine call, control stops at the first line in that routine. Abbreviated to s.
next [n]	Like step, but routine calls are treated as a single statement, so control stops at the statement after the routine call instead of the first statement of the called routine. Abbreviated to n.
continue	Continue execution until the next breakpoint is reached. Abbreviated to c.
finish	Finish execution of the current routine and stop at the statment after the routine call. Print the value returned by the finished routine, if any. Abbreviated to fin.

⇒ Step to the next line to be executed.

Notice that the arrow (=>) has moved and the variables r and x are printed (x has changed). The program has now stopped execution on the line r = m.bar(x);.

⇒ Step into routine mary::bar.

GDB responds with:

```
(gdb) s
mary::bar (this=0xffbefaf7, x=1) at gdb_basic.cc:17
```

indicating that execution has stopped within mary::bar at line 17, and that bar has a single parameter, x, containing the value 1. Also the arrow has moved to line 17 in the buffer containing gdb_basic.cc.

⇒ Display the values of variables x and r.
⇒ Step to the next line.
⇒ Step into routine fred::foo.

GDB responds with:

```
(gdb) s
fred::foo (this=0xffbefaf7, x=2) at gdb_basic.cc:5
```

⇒ Display the variable i.
⇒ Set a breakpoint at line 8 (return x;).
⇒ Enter the command: step 4 or s 4.
⇒ Continue to the next breakpoint by entering the command: cont or c.
⇒ Print the contents of variable x. *Why is the value 7?*
⇒ Enter the command: step 2 or s 2

Control has returned to routine mary::bar, which is about to return a value of 7. From the display statements, it can be seen that the changes to x in fred::foo have not affected x in mary::bar because x was passed by value.

⇒ Continue execution.

GDB responds with:

```
(gdb) c
Continuing

Program exited normally
```

Notice that the arrow has not moved to the end of the routine main.

⇒ Set a breakpoint at line 28.
⇒ Set a breakpoint in routine mary::bar by entering the command: break mary::bar
⇒ Run the program again.
⇒ Enter the command step 3 to move you to the line r = f.foo(x);
⇒ Enter the command next or n to step over the call to routine fred::foo.

Why did execution stop in fred*?* (Because there is a breakpoint at line 8.)

⇒ Enter the command finish to complete execution of routine fred.

GDB responds with:

```
Run till exit from #0  fred::foo (this=0xffbefaf7, x=7) at gdb_basic.cc:8
0x10718 in mary::bar (this=0xffbefaf7, x=2) at gdb_basic.cc:18
4: r = 0
3: x = 2
Value returned is $3 = 7
```

which indicates that fred::foo is returning the value 7 as it finishes. Control now returns to routine mary::bar.

⇒ Press <Return>. This repeats the last command, which was finish.

GDB responds with:

```
(gdb)
Run till exit from #0  0x10718 in mary::bar (this=0xffbefaf7, x=2) at gdb_basic.cc:18
0x10640 in main () at gdb_basic.cc:26
2: x = 1
1: r = -268436580
Value returned is $4 = 7
```

which indicates that mary::bar is returning the value 7 as it finishes.

⇒ Continue until execution of the program completes.

4.7 Controlling Breakpoints

⇒ Enter the command info breakpoints to obtain information on the breakpoints currently set.

GDB responds with:

```
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x00010624 in main at gdb_basic.cc:24
        breakpoint already hit 1 time
2   breakpoint     keep y   0x00010778 in fred::foo(int) at gdb_basic.cc:8
        breakpoint already hit 1 time
3   breakpoint     keep y   0x0001065c in main at gdb_basic.cc:28
        breakpoint already hit 1 time
3   breakpoint     keep y   0x00010704 in mary::bar(int) at gdb_basic.cc:17
        breakpoint already hit 1 time
```

⇒ In order to avoid stopping at the breakpoint on line 8, disable it by entering the command: disable 2
⇒ Disable breakpoint 4.

4.8 Changing Values

The set command changes the values of variables in the current routine or global/external variables.

⇒ Run the program again.
⇒ Step to the next line.
⇒ Change the initial value of x to 6 by entering the command: set x = 6
⇒ Continue execution.

Notice how the value of r is larger. In this way, it is possible to change the values of variables while debugging to investigate how the program behaves with new values instead of having to restart the debugging process or change and recompile the program.

⇒ Continue the program.
⇒ Quit out of GDB. If using Emacs, enter the command C-x k; otherwise, enter the command quit or q.
⇒ Close the gdb_basic.cc buffer.

5 Debugging Example 1

⇒ Create a buffer within Emacs containing the file gdb_q1.cc.

This program calculates the sum of an array of size n, given n (see Appendix B), and it currently contains some errors. The program is terminated by entering either the end of file key sequence (C-c C-d in Emacs, or <CTRL>-d in the Unix shell) or the sentinel value -999.

⇒ Compile the program using the command: g++ -g gdb_q1.cc

Notice the compilation error message displayed in the compilation buffer.

```
gdb_q1.cc:38: unterminated string or character constant
gdb_q1.cc:22: possible real start of unterminated constant
```

The compiler believes that a string or character constant is not terminated in line 38.

⇒ Go to line 38. Since the error probably consists of a missing quotation mark, start by examining line 38.

```
cout << array[i] << "] is " << returnValue << endl;
```

Examining line 38 of the code, it is clear that the string on that line is terminated properly! Therefore, the error comes from a previous line that contained a string that is not terminated properly. Start searching from line 22 forwards, as indicated by the second compiler error message.

⇒ Make the required correction.
⇒ Save the program.
⇒ Compile the program.

Having achieved a successful compilation:

⇒ Start up GDB on the program's executable, a.out.
⇒ Enter the command: run
⇒ Enter the number 5 in response to the prompt Enter the size of the array to sum [<CTRL>-D or -999 for EOF]: and press <Return>.
⇒ Enter a number at the next prompt, Enter value [1].

GDB responds with:

```
(gdb) run
Starting program: /u/cssystems/tutorial/GDB/C++/a.out

Enter the size of the array to sum [<CTRL>-D or -999 for EOF]: 5

Enter value [1] 1

Program received signal SIGSEGV, Segmentation fault.
0xff153974 in istream::operator>> (this=0x20ed0, i=@0x4) at iostream.cc:352
352 iostream.cc: No such file or directory.
```

The program received a "Segmentation fault" signal at address 0x6ff738e in routine istream::operator>>. This message often indicates a pointer addressing problem.

⇒ Enter the command where to receive more information about the location of the error in the program.

GDB responds with:

```
(gdb) where
#0  0xff153974 in istream::operator>> (this=0x20ed0, i=@0x4) at iostream.cc:352
#1  0x108c0 in readInArrayAndSum (array=0x4, size=1) at gdb_q1.cc:11
#2  0x109cc in main () at gdb_q1.cc:32
```

This information is about the *stack frames*. A *frame* is the data associated with a call to a routine. It contains the arguments passed to the routine, variables local to the routine, and the executing routine's address.

Upon starting a program, the stack has only one frame (the *outer-most* frame) which is for routine main. A new frame is created for each routine called. The frame labeled 0 (the *inner-most* frame) is the most recently created frame.

In this example, frame 0 involves the I/O output operator, >>. This routine is invoked by the routine readInArrayAndSum in frame 1. The information associated with frame 2 gives the file name containing the program (gdb_q1.cc. Each frame also gives a line number. This number is either the line on which the error occurred, or the line from which the routine containing the error was invoked. The frame for the I/O operator >> has no file and line numbers because it was not compiled with the -g flag. As a result, GDB cannot display the source code where the error occurred. Therefore, you have to manually begin the search in your program.

⇒ Enter frame 2 by typing: frame 2 or f 2
⇒ List line 32.

```
27          if ( !cin.good() ) {
28             cerr << "\nERROR: cannot have char or string for the array size\n";
29             exit(-2);
30           } // if
31           int *array;
32           int returnValue = readInArrayAndSum(array, size);
33           cout << "\nsum of [";
34           int i;
35           for (i = 0; i < size-1; i += 1); {
36                cout << array[i] << ", ";
```

Line 32 is the invocation of readInArrayAndSum. Since it appears correct, move into the frame associated with the routine and examine its associated information.

⇒ Enter frame 1.

GDB responds with

```
(gdb) frame 1
#1  0x108c0 in readInArrayAndSum (array=0x4, size=1) at gdb_q1.cc:11
```

You should notice that the address associated with the parameter array looks a bit odd.

⇒ Try printing the element at index 1.

GDB responds with

```
(gdb) print array[1]
Cannot access memory at address 0x4
```

So, the address contained in array is invalid.

⇒ Make the buffer containing gdb_q1.cc visible.
⇒ Go to line 31 (C-x @ command).
⇒ The declaration of array doesn't actually allocate any space. Change the declaration to read int array[size]; instead.
⇒ Save and recompile the file.
⇒ Create buffer containing data1 and examine its contents.
⇒ Make the the *gdb-a.out* buffer visible in a window.
⇒ Run the program again, but now use a data file for the input. Use the command: run < data1 *What does the* < data1 *do after the* run *command?* You will be prompted to restart it. *Why is this essential?*

Is the program working correctly? No, we still have a segmentation fault.

⇒ Print the value of i. You should see that it is significantly larger than 10, the size of the array.
⇒ Correct the code by changing the increment of i on line 9 to be by one rather than by two.
⇒ Recompile the code.
⇒ Run the program as before by entering just run on the GDB command line. Be aware that entering run is equivalent to run < data1 (see help run).

Is the output correct given the contents of data1*?* No, the output just have been a single sum, not two, and the single sum should have been 10, not 9. To determine the exact location of the error, the execution of the program needs to be traced. The error is likely to be in readInArrayAndSum since that routine handles the input and summation.

⇒ Enter the command: break readInArrayAndSum.
⇒ Run the program as before.

The program continues execution until the first breakpoint is reached.

The following code is displayed in the gdb_q1.cc buffer:

```
#include <iostream.h>

int readInArrayAndSum( int *array, int size ) {
=>  int sum = 0;
    for ( int i = 1; i != size; i += 1 ) {
        cout << "\nEnter value: ";
        cin >> array[i];
        sum += array[i];
    } // for
    return sum;
} // readInArrayAndSum
```

GDB tells you the values of the parameters array and size.

To trace the values of the variables that readInArrayAndSum() uses as they change, use the display command.

⇒ Step through the next two statements by using the next 2 command.
⇒ Display the variables i, sum, and array[i].
⇒ Step through the next statement, cout << "\nEnter value [" << i << "] ";.
⇒ Step through the next statement, cin >> array[i];.
⇒ Step through the next statement, sum += array[i];.
⇒ Step to the next statement, the for loop. sum is now set to 1, the value in array[1]. So far, everything seems appropriate.
⇒ Notice that the parameter to readInArrayAndSum indicates that 10 values will be read in. We don't want to step through all of the statements, we want to concentrate on the final stages of our loop, so set a break point on line 13.
⇒ Continue to the next breakpoint.
⇒ Bypass the next 8 crossings of the breakpoint by entering cont 8

GDB responds with

```
(gdb) cont 8
Will ignore next 7 crossings of breakpoint 3.  Continuing.

Enter value:
Enter value:
Enter value:
Enter value:
Enter value:
Enter value:
Enter value:
Enter value:
Breakpoint 3, readInArrayAndSum (array=0xffbafac8, size=10) at gdb_q1.cc:13
3: array[i] = 1
2: sum = 9
1: i = 9
```

⇒ Step to the next instruction. Suddenly, we're about to return sum instead of executing the loop one more time.
⇒ Print the value of array.
⇒ Print the address of the element at index 1 of array.

GDB responds with

```
(gdb) print array
$1 = (int *) 0xffbefac8
(gdb) print &(array[1])
$2 = (int *) 0xffbefacc
```

These two addresses do not match. So, the array actually starts one position earlier, at index 0.

⇒ Change the initialization of i in the for loop to 0 rather than 1.
⇒ Save and compile the code.
⇒ Move back to the *gdb-a.out* buffer.
⇒ Run the program again. You will be prompted to restart it. The program stops at the breakpoint in readInArrayAndSum.
⇒ Delete breakpoints 1 and 2.
⇒ Continue execution.

Does the code work correctly now? The sum is calculated correctly, but the array is not printed out correctly.

⇒ List line 32, since the print loop is in that general vicinity.
⇒ Put a break point on line 34.
⇒ Run the program again.
⇒ When you reach the loop, display variable i.
⇒ Step through the loop. Notice that the value of i has suddenly reached 9.

While the for loop appears to be correct, there is a very subtle error, which is the semi-colon at the end of the for line terminating the loop body; therefore, the code in the braces is simply a block separate from the for statement. In other words, the for loop executes until it is done, and then the code within the braces is executed.

⇒ Remove the semi-colon from the end of the for statement.
⇒ Save and compile the code.
⇒ Move back to the *gdb-a.out* buffer.
⇒ Run the program again.

Does the code work correctly now?

⇒ Close all of the current buffers.

6 Debugging Example 2

⇒ Create a buffer containing the file gdb_q2.cc.

This program reads in strings, and stores the string alphabetically, along with its number of occurrences, in a linked list (see Appendix C). The program currently contains some errors.

⇒ Compile the code by using the command: g++ -g gdb_q2.cc.
⇒ Create a buffer containing file data2 and examine the data.
⇒ Start up GDB on the program's executable, a.out.
⇒ Enter the command: run data2

GDB responds with:

```
(gdb) run data2
Starting program: /u3/ctkierstead/GDB/a.out data2

Program received signal SIGSEGV, Segmentation fault.
basic_string<char, string_char_traits<char>, __default_alloc_template<false, 0> >::rep (this=0x1)
    at /.software/arch/gcc-2.95.2/distribution/lib/gcc-lib/sparc-sun-solaris2.6/2.95.2/. ./. ./. ./. ./include/g++-3/std/bastring.h:147
```

with the arrow on a line in the string library.

An error of "Segmentation Fault" usually indicates a problem with a pointer.

⇒ It is unlikely the string library has a bug, so let's determine where the offending call was made. Use the where command.

GDB responds with:

```
(gdb) where
#0  basic_string<char, string_char_traits<char>, __default_alloc_template<false, 0> >::rep (this=0x1)
    at /.software/arch/gcc-2.95.2/distribution/lib/gcc-lib/sparc-sun-solaris2.6/2.95.2/. ./. ./. ./. ./include/g++-3/std/bastring.h:147
#1  0x14b88 in basic_string<char, string_char_traits<char>, __default_alloc_template<false, 0> >::basic_string (
    this=0xeffff998, str=@0x1)
    at /.software/arch/gcc-2.95.2/distribution/lib/gcc-lib/sparc-sun-solaris2.6/2.95.2/. ./. ./. ./. ./include/g++-3/std/bastring.h:172
#2  0x14df0 in ListNode::getWord (this=0x1) at gdb_q2.cc:20
#3  0x13170 in ListADT::searchInsert (this=0xeffffb1c, word={static npos = 4294967295, static nilRep = {len = 0,
        res = 0, ref = 1, selfish = false}, dat = 0x27d20 "This"}) at gdb_q2.cc:112
#4  0x12f28 in ListADT::ListADT (this=0xeffffb1c, fileName=0xefffcc6 "data2") at gdb_q2.cc:78
#5  0x12d30 in main (argc=2, argv=0xeffffb94) at gdb_q2.cc:49
```

The call listed in frame 2 has an invalid address assigned to the internal object pointer, this. So, the error probably came from the previous frame.

⇒ Use the frame 3 command to enter the frame so we can examine the previous frame's values.

GDB puts the arrow on the following line:

```
=>  if ( head == NULL || head->getWord() > word ) {
```

Since it was the call to getWord that failed, and the method was being called on head, the problem likely lies with head.

⇒ Print the contents of head. This is the first word that we are inserting into the list, and yet head contains an address of 1 rather than the expected 0.
⇒ Find the initialization of head in ListADT's constructor. Replace the statement (ListNode*) 1 with NULL.
⇒ Recompile the code and try running the program again.

Is the program correct? No, since no information is output for the list. Either the list is empty, or there is something wrong with the printing of its contents. The execution of the program must be traced in order to pinpoint the problem by setting some breakpoints and observing what happens when the code is run.

⇒ Set a breakpoint at line 112. This location is at the start of the insertion routine (If you had written the program, you would know this.) We could have set a breakpoint at line 78, just after the first word is read in; however, when we step into searchInsert, we'd have to first get through a number of string library calls.

⇒ Run the program again.
⇒ GDB stops at the breakpoint. To ensure that the list is being correctly maintained, print the value of head, which should be 0x0/NULL.
⇒ Display the object pointed to by the pointer head using the command display *head. (GDB will complain that it cannot display a NULL pointer and will disable the display.)
⇒ Display the value of head instead. We can always turn the other display back on later. The pointer to the top of the list is still NULL, which is correct for inserting the first token.
⇒ Step through the next 3 lines using the next command (watch the arrow), carefully observing the value of head.

head suddenly went from a non-zero value back to 0x0. This indicates an error with the if statement. Close examination of the if statement reveals that an assignment operator appears where an equals operator should be. Instead of ==, = is used which assigns the value of NULL to *head and causes the program lose all previous information stored in the list.

⇒ Correct the error.
⇒ Save and compile the program.
⇒ Enter the *gdb-a.out buffer.
⇒ Run the program again. (You will be prompted to restart it.) The program stops at the breakpoint in main.
⇒ Delete breakpoint 1 and continue execution.

The program now runs properly.

Before leaving GDB, two more useful features are worth mentioning: the ability to print out structures and the ability to easily follow pointers and print out the objects pointed to.

⇒ Enter buffer gdb_q2.cc
⇒ Move the cursor down to the line myList.printList();.
⇒ Enter the command: M-x what-line <Return>. The line number of the line containing the cursor is printed in the mini-buffer.
⇒ Set a breakpoint at this line.
⇒ Run the program again.
⇒ Enter the command: p *myList.head

GDB responds with:

```
$1 = {token = {static npos = 4294967295, static nilRep = {len = 0, res = 0, ref = 1, selfish = false},
    dat = 0x27f60 "Either"}, timesFound = 1, next = 0x28388}
```

⇒ Enter the command: set print pretty to print structures in a nice format.
⇒ Print *myList.head again.

GDB responds with:

```
$2 = {
  token = {
    static npos = 4294967295,
    static nilRep = {
      len = 0,
      res = 0,
      ref = 1,
      selfish = false
    },
    dat = 0x27f60 "Either"
  },
  timesFound = 1,
  next = 0x28388
}
```

⇒ Enter the command: print *myList.head.next:

GDB responds with:

```
$3 = {
  token = {
    static npos = 4294967295,
    static nilRep = {
      len = 0,
      res = 0,
      ref = 1,
      selfish = false
    },
    dat = 0x27f00 "Some"
  },
  timesFound = 1,
  next = 0x28220
}
```

⇒ Enter the command: print *$3.next:

GDB responds with:

```
$4 = {
  token = {
    static npos = 4294967295,
    static nilRep = {
      len = 0,
      res = 0,
      ref = 1,
      selfish = false
    },
    dat = 0x27d20 "This"
  },
  timesFound = 1,
  next = 0x28268
}
```

Notice the use of a history variable to save typing. This can be continued until the end of the list is reached, allowing a user to verify that the data structure is set up correctly.

⇒ Enter the command: cont
⇒ Close all of Emacs buffers.
⇒ Terminate Emacs by using the C-x C-c command.

A Basic

```
class fred {
  public:
    int foo( int x ) {
        int i;
        for ( i = 1; i <= 5; i += 1 ) {
            x += 1;
        } // for
        return x;
    } // foo
};

class mary {
    fred f;
  public:
    int bar( int x ) {
        int r;
        x += 1;
```

```
        r = f.foo( x );
        return r;
    }
};

int main() {
    int r, x = 1;
    mary m;
    r = m.bar( x );
    return 0;
}
```

B ArraySum

```
/**************************************************************************
  GDB Test File 1 - contains several errors
**************************************************************************/

#include <iostream.h>

int readInArrayAndSum(int *array, int size) {
    int sum = 0;
    for ( int i = 1; i != size; i += 2 ) {
        cout << "\nEnter value [" << i << "] ";
        cin >> array[i];
        sum += array[i];
    } // for
    return sum;
} // readInArrayAndSum

/**************************************************************************/

int main() {
    int size;

    cout << "\nEnter the size of the array to sum [<CTRL>-D or -999 for EOF]:;
    for( ; ; ) {
       cin >> size;
       if ( cin.eof() ) break;
       if ( size == -999 ) break;
       if ( !cin.good() ) {
         cerr << "\nERROR: cannot have char or string for the array size\n";
         exit(-2);
       } // if
       int *array;
       int returnValue = readInArrayAndSum(array, size);
       cout << "\nsum of [";
       int i;
       for (i = 0; i < size-1; i += 1); {
          cout << array[i] << ", ";
       } // for
       cout << array[i] << "] is " << returnValue << endl;
    } // for
    return 0;
} // main
```

C Strings

```
/**************************************************************************
  GDB Test File 2 - contains a few errors
 **************************************************************************/

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

```
/************************** Class Declarations ***************/

class ListNode {
    string     token;
    int        timesFound;                          // Times that token has been found
    ListNode *next;                                 // Next node in the list
  public:
    ListNode(string word) : token(word), timesFound(1), next(NULL) {}
    ListNode(string word, ListNode *next) : token(word), timesFound(1), next(next) {}
    ~ListNode() { if ( next != NULL ) delete next; }

    string     getWord()             { return token; }
    int        getNumOccurrences() { return timesFound; }
    void       incNumOccurrences() { timesFound++; }
    ListNode * getNext()             { return next; }
    void       setNext(ListNode * next) { ListNode::next = next; }
};



class ListADT {
    ListNode *head;                                 // Head of the list

  public:
    ListADT( char * fileName );                     // Initialize the list from the specified file
    ~ListADT();
    bool findWord( string word );
    void searchInsert( string word );
    void printList ();
};

/*****************************************************************************/

int main( int argc, char *argv[ ] ) {
    // Must have an input file specified on the command line.
    if ( argc != 2 ) {
        cerr << argv[0] << " input-file" << endl;
        exit(-1);
    } // if

    ListADT myList( argv[1] );                      // List of words.

    cout << "\nThe linked list contains:" << endl;
    myList.printList();
} // main

/************************** Class Implementations ***************************/

/*****************************************************************************
 Builds a list of words contained in the specified input file.

 Input: name of the input file
 Output: builds a list where head contains the address of the first node
 Error: program ends with error message and exit code -2 if the file could
        not be opened for input.
 *****************************************************************************/
ListADT::ListADT( char * fileName ) {
    head = (ListNode *) 1;                          // Initialize list to empty.
    string word;

    ifstream infile( fileName, ios::in );           // Open input file
    if ( !infile ) {
        cerr << "File " << fileName << " could not be opened for input." << endl;
        exit(-2);
    } // if

    while ( true ) {
```

```
        infile >> word;
        if ( infile.eof() ) break;
        searchInsert( word );
    } // while
} // ListADT::ListADT

/*************************************************************************
  Determines if the specified word is in the list or not.

  Input: word to be searched for
  Output: flag indicating whether a word has been found or not.
  Errors: N/A
 *************************************************************************/
bool ListADT::findWord ( string word ) {
    bool foundWord = false;
    for ( ListNode *ptr = head; ptr != NULL && !foundWord; ptr = ptr->getNext() ) {
        if ( ptr->getWord() == word ) {
            foundWord = true;
        } // if
    } // for
} // ListADT::findWord

/*************************************************************************
  Searches the list for the word. If the word is in the list, the
  repetition count of the word is incremented. Otherwise, the word is
  added to the list.

  Input: Pointer to the start of the list - ListHead
  Pointer to the buffer which contains the word - Token
  Output: A list of words with their frequencies of occurrence
  Errors: If no space for new word, then print message and exit program
 *************************************************************************/
void ListADT::searchInsert ( string word ) {
    ListNode *newNode, *temp, *prev;
    int value;

    if ( head == NULL || head->getWord() > word ) {
        head = new ListNode( word, head );
        if ( head = NULL ) {                        // Ran out of space.
            cerr << "ERROR: ran out of space\n";
            exit(-3);
        } // if
    } else if ( head->getWord() == word ) {
        head->incNumOccurrences();
    } else {
        newNode = new ListNode( word );
        if ( newNode == NULL ) {                    // Ran out of space.
            cerr << "ERROR: ran out of space\n";
            exit(-3);
        } // if
        prev = head;
        for ( temp = head->getNext(); temp != NULL; temp = temp->getNext() ) {
            if ( word == temp->getWord() ) {
                temp->incNumOccurrences();
                break;
            } else if ( word < temp->getWord() ) {
                newNode->setNext( temp );
                prev->setNext( newNode );
                break;
            } // if
            prev = temp;
        } // for
        if ( temp == NULL && prev != NULL ) {       // Fell off list
            prev->setNext( newNode );
        } // if
    } // if
} // ListADT::SearchInsert
```

```
/*****************************************************************************
 Prints the list with the word repetition counts.

  Input: N/A
  Output: The words and their frequencies of occurrence.
 *****************************************************************************/
void ListADT::printList() {
    for ( ListNode *ptr = head; ptr != NULL ; ptr = ptr->getNext() ) {
        cout << "word: " << ptr->getWord() << " occurs "
             << ptr->getNumOccurrences() << " time(s)" << endl;
    } // for
} // ListADT::printList

/*****************************************************************************
  Deletes the list.

  Input: N/A
  Output: The list has been freed.
 *****************************************************************************/
ListADT::~ListADT () {
    if ( head != NULL ) {
        delete head;
    } // if
} // ListADT::~ListADT
```

Index