GDB Tutorial

University of Waterloo

Version 1.1

Caroline Kierstead and Peter A. Buhr ©[*]1993

May 1, 2000

[*]Permission is granted to make copies for personal or educational use

Contents

1 Introduction **3**

2 Before Using GDB **3**
2.1 Debug Print Statements 3
2.2 Errors 4

3 Getting Started **5**

4 Using GDB **5**
4.1 Getting Help 5
4.2 Starting a Program 6
4.3 Setting a Breakpoint 6
4.4 Listing Source Code 7
4.5 Printing Variables 7
4.6 Controlling Execution 8
4.7 Controlling Breakpoints 9
4.8 Changing Values 9

5 Debugging Example 1 **10**

6 Debugging Example 2 **12**

A Basic **15**

B Factorial **15**

C Strings **16**

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 concept 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:

⇒ This symbol indicates that you are to perform the action marked by the arrow.

⋈ 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 */
    fprintf( stderr, "Enter p( . . . )\n", parameter variables );
    . . .
    fprintf( stderr, "Exit p: . . .\n", return value(s) );
    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 ) {
    fprintf( stderr, "a > b\nn" );                 /* debug print */
    for ( . . . ) {
        fprintf( stderr,"x=%d, y=%d\n", x, y );    /* debug print */
        . . .
    } /* for */
} else {
    fprintf( stderr, "a <= b\n" );                 /* 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 ) {
    fprintf( stderr, "a == b\n" );
} /* 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.c
/u/cssystems/examples/gdb_q1.c
/u/cssystems/examples/gdb_q2.c
/u/cssystems/examples/data
```

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.c`.

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: `gcc -g gdb_basic.c`

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:

running -- Running the program
stack -- Examining the stack
data -- Examining data
breakpoints -- Making program stop at certain points
files -- Specifying and examining files
status -- Status inquiries
support -- Support facilities
user-defined -- User-defined commands
aliases -- Aliases of other commands
obscure -- Obscure features
internals -- Maintenance 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 0x2320: file gdb_basic.c, line 17.
```

indicating that breakpoint number 1 has been set at location 0x2320, which is line 17 of the file gdb_basic.c. 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 17. 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 17 or l 17.

GDB responds with:

```
(gdb) list 17
12        r = fred( x );
13        return r;
14    }
15
16    int main() {
17        int r, x = 1;
18        r = mary( x );
19        return 0;
20    }
```

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 = mary(x);.

⇒ Step into routine mary.

GDB responds with:

```
(gdb) s
mary (x=1) at gdb_basic.c:11
```

indicating that execution has stopped within mary at line 11, and that mary has a single parameter, x, containing the value 1. Also the arrow has moved to line 11 in the buffer containing gdb_basic.c.

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

GDB responds with:

```
(gdb) s
fred (x=2) at gdb_basic.c:3
```

⇒ Display the variable i.
⇒ Set a breakpoint at line 6 (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, which is about to return a value of 7. From the display statements, it can be seen that the changes to x in fred have not affected x in mary because x was passed by value.

⇒ Continue execution.

GDB responds with:

```
(gdb) c
Continuing

Progran exited normally
```

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

⇒ Set a breakpoint at line 19.
⇒ Set a breakpoint in routine mary by entering the command: break mary
⇒ Run the program again.
⇒ Enter the command step 3 to move you to the line r = fred(x);

⇒ Enter the command next or n to step over the call to routine fred.

Why did execution stop in fred*?* (Because there is a breakpoint at line 6.)

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

GDB responds with:

```
(gdb) fin
Run till exit from #0  fred (x=7) at gdb_basic.c:6
0x2304 in mary (x=2) at gdb_basic.c:12
4: r = 0
3: x = 2
Value returned is $2 = 7
```

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

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

GDB responds with:

```
(gdb)
Run till exit from #0  0x2304 in mary (x=2) at gdb_basic.c:12
0x2334 in main () at gdb_basic.c:18
2: x = 1
1: r = 0
Value returned is $3 = 7
```

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

⇒ Continue to the next breakpoint.
⇒ 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   0x00002320 in main at gdb_basic.c:17
2   breakpoint    keep y   0x000022d0 in fred at gdb_basic.c:6
3   breakpoint    keep y   0x00002338 in main at gdb_basic.c:19
4   breakpoint    keep y   0x000022ec in mary at gdb_basic.c:11
```

⇒ In order to avoid stopping at the breakpoint on line 6, 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.c buffer.

5 Debugging Example 1

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

This program calculates 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: gcc -g gdb_q1.c

Notice the compilation error message displayed in the compilation buffer.

```
gdb_q1.c:47 unterminated string or character constant
```

The compiler believes that a string or character constant is not terminated in line 47.

⇒ Go to line 47. Since the error probably consists of a missing quotation mark, start by examining line 47.

```
printf("\n%d factorial is %d\n",value, ret_value );
```

Examining line 47 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.

⇒ Go to the beginning of the file and search forward for the character " until a string is found with unmatching quotes.
⇒ 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 1 in response to the prompt Enter a number [<CTRL>-D or -999 for EOF]: and press <Return>.

GDB responds with:

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

Enter a number [<CTRL>-D or -999 for EOF]:1

Program received signal 11, Segmentation fault
0xd39c in _doscan ()
```

The program received a "Segmentation fault" signal at address 0xd39c in routine _doscan(). 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  0xd39c in _doscan ()
#1  0xcb24 in _doscan ()
#2  0x5e40 in scanf ()
#3  0x2444 in main () at gdb_q1.c:38
```

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, frames 0 and 1 involve I/O operations (the underscore in front of the routine name doscan indicates that it is a system routine). This routine is invoked by the routine scanf in frame 2. The information associated with frame 3 gives the file name containing the program (gdb_q1 and line number on which the error occurred (38). The frames for scanf and _doscan have no file and line numbers because they were 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.

⇒ List line 38.

```
33  {
34  int value, EndOfFileFlag, ret_value;
35
36  for( ; ; ) {
37      printf("\nEnter a number [<CTRL>-D or -999 for EOF]:");
38      EndOfFileFlag = scanf("%d",value);
39      if ( EndOfFileFlag == EOF ) break;
40      if ( value == -999 ) break;
41      if ( EndOfFileFlag != 1 ) {
42        fprintf(stderr,"\nERROR: cannot calculate factorial for char or string\n");
```

While line 38 appears correct, it has a common C mistake which is: passing the value of a variable to scanf instead of the address of the variable. The value of variable value was probably 0 and so scanf tried to store a value at location 0, which is protected memory.

Since the error did not occur directly in your program (it occurred in the I/O routine _doscan), the buffer for gdb_q1.c is not made visible (even though the error is due to a mistake in your program).

⇒ Make the buffer containing gdb_q1.c visible.
⇒ Go to line 38 (C-x @ command).
⇒ Insert the character & in front of the variable value.
⇒ Save and recompile the file.
⇒ Make the the *gdb-a.out* buffer visible in a window.
⇒ Run the program again. You will be prompted to restart it. *Why is this essential?*
⇒ Try re-executing the program with the following values: 1, 2, 3, -999.

Is the program working correctly? The input value is now read correctly (confirmed in the output), so the error must be in the routine Factorial(). To determine the exact location of the error, the execution of the program needs to be traced.

⇒ Enter the command: break Factorial.
⇒ Run the program.
⇒ Enter the number: 3

The program continues execution until the first breakpoint is reached.

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

```
int Factorial( int value )
{
int counter, prod;

=> ( value < 0 ) {
  fprintf(stderr,"\nERROR: cannot calculate factorial for %d < 0\n",value);
  return( ERROR_VALUE );
} else if ( value == 0 ) {
  return( 1 );
} else {
  prod = 1;
```

(Note the arrow is covering the characters if. GDB also tells you the value of the parameter value is 3.

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

⇒ Display the variables value, prod, and counter.
⇒ Step to the next statement, else if (value == 0).

⇒ Step to the next statement, `prod = 1;`. Notice that the value for `prod` has not yet changed.
⇒ Step to the next statement, the `for` loop. `prod` is now initialized to 1.
⇒ Step to the next statement, `prod *= counter;`. `counter` is set to 4.
⇒ Step to the next statement, `return( prod );`. `prod` is set to 4.

By looking at the code and the results from the `step` commands, it is obvious that the problem arises because the for loop is not executing.

⇒ Look for a problem with the for loop.

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. You will be prompted to restart it.
⇒ Try executing the program with the following values: 1, 2, 3, 4, 5, -999. The program stops at the breakpoint in `Factorial`.
⇒ delete breakpoint 1 and continue execution.

Does the code work correctly now?

⇒ Close all of the current buffers.

6 Debugging Example 2

⇒ Create a buffer containing the file `gdb_q2.c`.

This program reads in strings, up to 20 characters in length, 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: `gcc -g gdb_q2.c`.
⇒ Create a buffer containing file `data` and examine the data.
⇒ Start up GDB on the program's executable, `a.out`.
⇒ Enter the command: `run < data`

What does the `< data` *do after the* `run` *command?*
GDB responds with:

```
(gdb) run < data
Starting program: /u/ctkierstead/a.out < data

Program received signal 11, Segmentation fault
0x261c in SearchInsert (ListHead=0xf7fffbbc, Token=0xf7fffba0 "This") at gdb_q2.c:138
```

with the arrow on line:

```
=>  for (temp = *ListHead; temp->Next != NULL; temp = temp->Next) {
```

An error of "Segmentation Fault" usually indicates a problem with a pointer. Since `temp` is assigned the value of `*ListHead` and the for loop checks `temp`'s field `Next`, examine the value of the object pointed at by pointer `ListHead`. Because the pointer `ListHead` is actually the address of the pointer to the head of the linked list, dereference the pointer in order to examine the object it points to.

⇒ Enter the command: `print *ListHead`

GDB responds with:

```
$1 = (struct ListNodeType *) 0x0
```

At this point in the code, *ListHead contains the address 0x0. This is GDB's representation of a NULL pointer. So the error occurred when the program tried to dereference temp and examine the value of its field Next. This segment of code searches the ordered list until the word is found, the end of the list is reached, or the proper insertion point in the list is found. So, the program needs to know if temp is NULL, not temp->Next, if it is to properly find the end of the linked list.

⇒ Switch to buffer gdb_q2.c.
⇒ Change the first occurrence of temp->Next to temp.
⇒ Save and compile the program.
⇒ Enter the *gdb-a.out buffer.
⇒ Run the program again. You will be prompted to restart it. Be aware that entering run is equivalent to run < data (see help run).

GDB responds with:

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
'/u/cssystems/tutorial/GDB/C/a.out' has changed; re-reading symbols.
Starting program: /u/cssystems/tutorial/GDB/C/a.out < data

Program received signal 11, Segmentation fault
0x2688 in SearchInsert (ListHead=0xf7fffbbc, Token=0xf7fffba0 "This") at gdb_q2.c:144
Source file is more recent than executable.
```

Once again there is a "Segmentation fault" error.

The arrow is shown on the statement temp->TimesFound += 1;. Examine the value of the pointer temp since the assignment statement tries to access temp's field TimesFound.

⇒ Enter the command: print temp

GDB responds with:

```
$2 = (struct ListNodeType *) 0x0
```

At this point in the code, temp equals NULL. 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 breakpoints at line 44. This location is just before the first word is read in and is a good place to start debugging. (If you had written the program, you would know this.)
⇒ Run the program again.
⇒ GDB stops at the breakpoint. To ensure that the list is being correctly maintained, print the value of ListHead, which should be 0x0/NULL.
⇒ Enter the command: next to step over the execution of the routine Find_Token().
⇒ Step to the next line.
⇒ Step into the routine SearchInsert().
⇒ Display the object pointed to by the pointer ListHead using the command display *ListHead. The pointer to the top of the list is still NULL, which is correct for inserting the first token.
⇒ Step to the next line (watch the arrow).

The section of code which should be performed when *ListHead has a value of NULL is being skipped over. 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 *ListHead and causes the program to continue into an incorrect section of code.

⇒ 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.c`
⇒ Move the cursor down to the line `Print_List( ListHead );`.
⇒ 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 *ListHead`

GDB responds with:

```
$4 = {Token = {"This", '\000' <repeats 16 times>}, TimesFound = 1, Next = 0x1d1b0}
```

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

GDB responds with:

```
$5 = {
  Token = {"This", '\000' <repeats 16 times>},
  TimesFound = 1,
  Next = 0x1d1b0
}
```

⇒ Enter the command: `print *ListHead.Next:`

GDB responds with:

```
$6 = {
  Token = {"Either", '\000' <repeats 14 times>},
  TimesFound = 1,
  Next = 0x1d160
}
```

⇒ Enter the command: `print *$6.Next:`

GDB responds with:

```
$7 = {
  Token = {"Some", '\000' <repeats 16 times>},
  TimesFound = 1,
  Next = 0x1cfa8
}
```

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

```
int fred( int x ) {
    int i;
    for ( i = 1; i <= 5; i += 1 ) {
          x += 1;
    }
    return x;
}
int mary( int x ) {
    int r;
    x += 1;
    r = fred( x );
    return r;
}
int main() {
    int r, x = 1;
    r = mary( x );
    return 0;
}
```

B Factorial

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

#include <stdio.h>
#include <stdlib.h>

#define ERROR_VALUE -1
/*************************************************************************/

int Factorial( int value ) {
    int counter, prod;

    if ( value < 0 ) {
          fprintf(stderr,"\nERROR: cannot calculate factorial for %d < 0\n",value);
          return( ERROR_VALUE );
    } else if ( value == 0 ) {
          return( 1 );
    } else {
          prod = 1;
          for ( counter = 1; counter <= value ; counter += 1 );
          {
              prod *= counter;
          } /* for */
    } /* if */
    return( prod );
} /* Factorial */

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

void main( void ) {
    int value, EndOfFileFlag, ret_value;

    for( ; ; ) {
          printf("\nEnter a number [<CTRL>-D or -999 for EOF]:);
          EndOfFileFlag = scanf("%d",value);
      if ( EndOfFileFlag == EOF ) break;
      if ( value == -999 ) break;
      if ( EndOfFileFlag != 1 ) {
              fprintf(stderr,"\nERROR: cannot calculate factorial for char or string\n");
              exit(-2);
          } /* if */
          ret_value = Factorial( value );
      if (ret_value != ERROR_VALUE ) {
         printf("\n%d factorial is %d\n",value, ret_value );
      } /* if */
   } /* for */
} /* main */
```

C Strings

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

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define TOKEN_LENGTH        20                  /* Assume maximum length of token. */
#define FALSE                0
#define TRUE                 1

/************************** Typedefs and Symbolic Constants ***************/

typedef struct ListNodeType {
   char Token[TOKEN_LENGTH];                    /* Pointer to text of token. */
   int TimesFound;                              /* Times that token has been found */
   struct ListNodeType *Next;                   /* Next nodes in the list */
} LIST_NODE;

/************************** Function Prototypes ***************************/

LIST_NODE *Alloc_ListNode( void );
int Find_Token( char *Token );
void SearchInsert( LIST_NODE **ListHead, char *Token );
void Print_List ( LIST_NODE *ListHead );
void Delete_List ( LIST_NODE **ListHead );

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

void main( void ) {
   LIST_NODE *ListHead;                         /* Pointer to the root of tree */
   char Token[TOKEN_LENGTH];                    /* Token read in */
   int EndOfFileFlag;                           /* Flag => have reached eof. */

   ListHead = NULL;

   /* Read in tokens one by one. Break the loop only when have reached the
      end of the file. Pass the token to the procedure SearchInsert. */

   for( ; ; ) {
        EndOfFileFlag = Find_Token( Token );
     if ( EndOfFileFlag == TRUE ) break;
        SearchInsert( &ListHead, Token );
   } /* for */
   printf("\nThe linked list contains:\n");
   Print_List( ListHead );
   Delete_List( &ListHead );
} /* main */

/***************************************************************************
  Alloc_ListNode() returns a pointer to a freshly allocated ListNode.

  Input: none
  Output: Pointer to a new node for the list.
  Errors: no action
  *************************************************************************/

LIST_NODE *Alloc_ListNode() {
    return( ( LIST_NODE * ) malloc( (unsigned) sizeof(LIST_NODE) ) );
} /* Alloc_ListNode */

/***************************************************************************
  Find_Token finds the next white-space delimited token in a file
  and puts it into a buffer. It is assumed tokens will be less than
  TOKEN_LENGTH.

  Input: Pointer to where the token is stored - **TokenBuffer
  Output: Token.
  Flag indicating whether a token has been found or not.
  Errors: If the end of the file has been reached, return to the calling
  procedure.
 ***************************************************************************/

int Find_Token ( char *TokenBuffer) {
    int Symbol;                                         /* Symbol that was read from file. */
    int BufferLen, EndOfFileFlag;

    /* Skip leading whitespace by looping until either found the end of
       the file or reached a non-whitespace symbol. Read 1 character at a
       time.
     */

    EndOfFileFlag = FALSE;

    for( ; ; ) {
          Symbol = getchar();
      if( Symbol == EOF ) break;
      if( Symbol != ´ ´ && Symbol != ´\n´ && Symbol != ´\t´ ) break;
    } /* for */

    if( Symbol == EOF ) {
          EndOfFileFlag = TRUE;
    } else  {
          for ( BufferLen = 0 ; BufferLen < TOKEN_LENGTH - 2; BufferLen += 1 ) {
              TokenBuffer[BufferLen] = Symbol;
              Symbol = getchar();
            if( Symbol == EOF || Symbol == ´ ´ || Symbol == ´\n´
                 || Symbol == ´\t´ ) break;                /* Found end of token. */
          } /* for */
          TokenBuffer[BufferLen+1] = ´\0´;               /* End string. */
    } /* if */
    return( EndOfFileFlag );
} /* Find_Token */

/***************************************************************************
  SearchInsert searches the list for the token. If the token is in the list,
  the repetition count of the token is incremented. Otherwise, the token is
  added to the list.

  Input: Pointer to the start of the list - ListHead
  Pointer to the buffer which contains the token - Token
  Output: A list of tokens with their frequencies of occurrence
  Errors: If no space for new token, then print message and ignore token.
 ***************************************************************************/

void SearchInsert ( LIST_NODE **ListHead, char *Token ) {
    LIST_NODE *new_node, *temp, *prev;
    int value;

    if (*ListHead = NULL) {
          *ListHead = Alloc_ListNode();
          if( *ListHead == NULL ) {                      /* Ran out of space. */
              fprintf("ERROR: ran out of space\n");
              exit(-3);
          } /* if */
          (*ListHead)->Next = NULL;
          (*ListHead)->TimesFound = 1;
          strncpy((*ListHead)->Token,Token,TOKEN_LENGTH-1);
    } else {
          prev = *ListHead;
          for ( temp = *ListHead; temp->Next != NULL; temp = temp->Next ) {
              value = strcmp(temp->Token, Token);
            if ( value >= 0 ) break;
              prev = temp;
          } /* for */
          if (value == 0) {
              temp->TimesFound += 1;
          } else {
              new_node = Alloc_ListNode();
              if ( new_node == NULL ) {                    /* Ran out of space. */
                    fprintf("ERROR: ran out of space\n");
                    exit(-3);
              } /* if */
            strncpy(new_node->Token,Token,TOKEN_LENGTH-1);
              new_node->TimesFound = 1;
              if ( temp != NULL ) {
                  new_node->Next = prev->Next;
                    prev->Next = new_node;
              } else {
                    prev->Next = new_node;
                  new_node->Next = NULL;
              } /* if */
          } /* if */
    } /* if */
} /* SearchInsert */

/**************************************************************************
  Print_List prints the linked list with the token repetition counts.

  Input: Pointer to the root of the tree - ListHead
  Output: The tokens and their frequencies of occurrence.
 **************************************************************************/

void Print_List ( LIST_NODE *ListHead) {
    LIST_NODE *temp;

    for ( temp = ListHead; temp != NULL ; temp = temp->Next ) {
          printf("Token: %20s occurs %d time(s)\n",temp->Token,temp->TimesFound);
    } /* for */
} /* Print_List */

/**************************************************************************
  Delete_List deletes the binary search tree.

  Input: Pointer to the root of the tree - ListHead
  Output: Resets ListHead to NULL at the end.
 **************************************************************************/

void Delete_List ( LIST_NODE **ListHead) {
    LIST_NODE *temp, *prev;

    prev = *ListHead;
    if ( *ListHead != NULL ) {
          for ( temp = (*ListHead)->Next ; temp != NULL ; temp = temp->Next ) {
              prev = NULL;
              free(prev);
              prev = temp;
          } /* for */
    } else {
          free(*ListHead);
    } /* if */
} /* Delete_List */
```

Index

$\Rrightarrow$, 3
$\bowtie$, 3
<Return>, 9

break, 6, 11
breakpoint, 6
  continue, 8
  delete, 6
  disable, 6
  enable, 6
  finish, 8
  next, 8
  step, 8

continue, 8

debug print statements, 3
delete, 6
disable, 6, 9
display, 7, 11

enable, 6
execution error, 4

finish, 8, 9

gdb
  <Return>, 9
  break, 6
  breakpoint, 6
    continue, 8
    delete, 6
    disable, 6
    enable, 6
    finish, 8
    next, 8
    step, 8
  continue, 8
  delete, 6
  disable, 6
  disable, 9
  display, 7
  enable, 6
  finish, 8
  finish, 9
  help, 5
  info, 9
  list, 7
  next, 8
  print, 7
  run, 6
  set, 9
  stack frame, 10
  step, 8
  undisplay, 7
  where, 10
gdb, 5

help, 5

info, 9
inner-most frame, 10

list, 7

next, 8

outer-most frame, 10

print, 7

run, 6

semantic error, 4
set, 9
stack frame, 10
step, 8
syntax error, 4

undisplay, 7
unix consultant, 3

where, 10