Using the GNU debugger (GDB)

The following is a *summary* of the GNU GDB Manual. Which is part of the documentation of GNU GDB.

Introduction

The purpose of a debugger such as GDB is to allow you to see what is going on "inside" another program while it executes--or what another program was doing at the moment it crashed.

GDB can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act:

- Start your program, specifying anything that might affect its behavior.
- Make your program stop on specified conditions.
- Examine what has happened, when your program has stopped.
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

In order to debug a program effectively, you need to generate debugging information when you compile it. This debugging information is stored in the object file; it describes the data type of each variable or function and the correspondence between source line numbers and addresses in the executable code.

To request debugging information, specify the `-g' option when you run the compiler.

GDB Quickstart

We provide you with a list of common GDB operations which should get you going :

Stopping Execution

break

A *breakpoint* makes your program stop whenever a certain point in the program is reached. For each breakpoint, you can add conditions to control in finer detail whether your program stops. You can set breakpoints with the `break' command and its variants to specify the place where your program should stop by line number, function name or exact address in the program.

break FUNCTION

Set a breakpoint at entry to function FUNCTION.

break LINENUM

Set a breakpoint at line LINENUM in the current source file.

break FILENAME:LINENUM

Set a breakpoint at line LINENUM in source file FILENAME.

break FILENAME:FUNCTION

Set a breakpoint at entry to function FUNCTION found in file FILENAME.
break ... if COND
Set a breakpoint with condition COND; evaluate the expression COND each time the breakpoint is reached, and stop only if the value is nonzero--that is, if COND evaluates as true. ... stands for one of the possible arguments described above (or no argument) specifying where to break.
watch
You can use a watchpoint to stop execution whenever the value of an expression changes, without having to predict a particular place where this may happen.
watch EXPR
Set a watchpoint for an expression. GDB will break when EXPR is written into by the program and its value changes.
info
Print a table of all breakpoints and watchpoints set and not deleted, with the following columns for each breakpoint: info breakpoints [N]
info break [N]
info watchpoints [N]
clear
Delete any breakpoints at the next instruction to be executed in the selected stack frame. When the innermost frame is selected, this is a good way to delete a breakpoint where your program just stopped.
clear FUNCTION
clear FILENAME:FUNCTION
Delete any breakpoints set at entry to the function FUNCTION.
clear LINENUM
clear FILENAME:LINENUM
Delete any breakpoints set at or within the code of the specified line.
delete *[breakpoints] [BNUMS...]*
Delete the breakpoints or watchpoints of the numbers specified as arguments. If no argument is specified, delete all breakpoints (GDB asks confirmation, unless you have `set confirm off'). You can abbreviate this command as `d'.

Running/Resuming Execution
run
Use the `run' command to start your program under GDB. You must first specify the program name with an argument to GDB, or by using the `file' or `exec-file' command.
step
Continue running your program until control reaches a different source line, then stop it and return control to GDB. This command is abbreviated **s**.
step [COUNT] (shorthand s)
Continue running as in `step', but do so COUNT times. If a breakpoint is reached, or a signal not related to stepping occurs before COUNT steps, stepping stops right away.
next *[COUNT]* (shorthand n)
Continue to the next source line in the current (innermost) stack frame. This is similar to `step', but function calls that appear within the line of code are executed without stopping. Execution stops when control reaches a different line of code at the original stack level that was executing when you gave the `next' command. This command is abbreviated `n'.
continue *[IGNORE-COUNT]* (shorthand c)

Resume program execution, at the address where your program last stopped; any breakpoints set at that address are bypassed. The optional argument IGNORE-COUNT allows you to specify a further number of times to ignore a breakpoint at this location.
finish
Continue running until just after function in the selected stack frame returns. Print the returned value (if any).

Examining the Data and Source Code

print *[EXP]* (shorthand p)
EXP is an expression (in the source language). By default the value of EXP is printed in a format appropriate to its data type; If you omit EXP, GDB displays the last value again.
display EXP (shorthand disp)
Add the expression EXP to the list of expressions to display each time your program stops. `display' does not repeat if you press RET again after using it.
list (shorthand l)
To print lines from a source file, use the `list' command (abbreviated `l'). By default, ten lines are printed. There are several ways to specify what part of the file you want to print.

Here are the forms of the `list' command most commonly used:

list LINENUM
Print lines centered around line number LINENUM in the current source file.
list FUNCTION
Print lines centered around the beginning of function FUNCTION.
list
Print more lines. If the last lines printed were printed with a `list' command, this prints lines following the last lines printed; however, if the last line printed was a solitary line printed as part of displaying a stack frame (*note Examining the Stack: Stack.), this prints lines centered around that line.
list -
Print lines just before the lines last printed.

Examining the Stack

When your program has stopped, the first thing you need to know is where it stopped and how it got there.

Each time your program performs a function call, information about the call is generated. That information includes the location of the call in your program, the arguments of the call, and the local variables of the function being called. The information is saved in a block of data called a "stack frame". The stack frames are allocated in a region of memory called the "call stack".

backtrace (shorthand bt)
Print a backtrace of the entire stack: one line per frame for all frames in the stack.

GDB Example

First we need to compile our program using the -g flag. We will be considering the list abstract data type (ADT). Our main program contains a set of tests in creating, inserting, and searching for list elements.

```
$ ls
Makefile     list.c       list.h       test-list.c
$ make
gcc -Wall --strict-prototypes -g -c list.c
gcc -Wall --strict-prototypes -g test-list.c list.o -o test-list
```

We are now ready to start the debugger on the executable test-list.

```
$ gdb test-list
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.6, Copyright 1992 Free Software Foundation, Inc.
(gdb) run
Starting program: /home/akonstan/allen/cs3139/recitation3/int-list/test-list
begin test-list : insert two integer values 100, 200
L = 200 100
assigning '300' to val1 ...
L = 200 100

end test-list

Program exited normally.

(gdb) break main
Breakpoint 1 at 0x804851a: file test-list.c, line 13.
(gdb) run
Starting program: /home/akonstan/allen/cs3139/recitation3/int-list/test-list

Breakpoint 1, main () at test-list.c:13
13          List L = NULL;      /* declare (but not create) the list */
(gdb) step
16          printf("begin test-list : insert two integer values 100, 200\n");
(gdb) print L

$1 = (struct Node *) 0x0
(gdb) s
begin test-list : insert two integer values 100, 200
17          L = MakeEmpty(L);   /* create the list */
(gdb) s
MakeEmpty (L=0x0) at list.c:36
36          if( L != NULL )
(gdb) display L
1: L = (struct Node *) 0x0
(gdb) list
```

```
31            }
32        }
33
34        List MakeEmpty( List L )
35        {
36            if( L != NULL )
37              DeleteList( L );
38            L = (List) malloc( sizeof( struct Node ) );
39            assert(L != NULL);
40
(gdb) s
38            L = (List) malloc( sizeof( struct Node ) );
1: L = (struct Node *) 0x0
(gdb) s

39            assert(L != NULL);
1: L = (struct Node *) 0x8049c50
(gdb) s
41            L-&gt;Next = NULL;
1: L = (struct Node *) 0x8049c50
(gdb) s
42            return L;
1: L = (struct Node *) 0x8049c50
(gdb) display *L
2: *L = {Element = 0, Next = 0x0}
(gdb) s
43        }
2: *L = {Element = 0, Next = 0x0}
1: L = (struct Node *) 0x8049c50
(gdb) s
main () at test-list.c:18
18            assert(IsEmpty(L)); /* should be empty */

(gdb) next
19            val1 = 100;         /* assign value */
(gdb) s
20            val2 = 200;
(gdb) s
22            Insert(val1, L, Header(L));         /* insert value val1 into list */
(gdb) s
Header (L=0x8049c50) at list.c:120
120       return L;
(gdb) s
121   }
(gdb) s

Insert (X=100, L=0x8049c50, P=0x8049c50) at list.c:110
110       TmpCell = malloc( sizeof( struct Node ) );
```

```
(gdb) bt
#0  Insert (X=100, L=0x8049c50, P=0x8049c50) at list.c:110
#1  0x804859a in main () at test-list.c:22
#2  0x80484ce in ___crt_dummy__ ()
(gdb) s
111         assert(TmpCell != NULL);
(gdb) break Find
Breakpoint 2 at 0x804886e: file list.c, line 64.
(gdb) cont
Continuing.

Breakpoint 2, Find (X=100, L=0x8049c50) at list.c:64
64          P = L-&gt;Next;
(gdb) print *L
$2 = {Element = 0, Next = 0x8049c60}

(gdb) print *(L-&gt;Next)
$3 = {Element = 100, Next = 0x0}
(gdb) c
Continuing.

Breakpoint 2, Find (X=200, L=0x8049c50) at list.c:64
64          P = L-&gt;Next;
(gdb) c
Continuing.
L = 200 100
assigning '300' to val1 ...
L = 200 100

end test-list

The program is not being run.
(gdb) delete
Delete all breakpoints? (y or n) y
(gdb) break MakeEmpty
Breakpoint 2 at 0x80487df: file list.c, line 36.

(gdb) run
Starting program: /home/akonstan/allen/cs3139/recitation3/int-list/test-list
begin test-list : insert two integer values 100, 200

Breakpoint 2, MakeEmpty (L=0x0) at list.c:36
36        if( L != NULL )
(gdb) display L
1: L = (struct Node *) 0x0
(gdb) s
38        L = (List) malloc( sizeof( struct Node ) );
1: L = (struct Node *) 0x0
```

```
(gdb) s
39        assert(L != NULL);
1: L = (struct Node *) 0x8049c50
(gdb) print L = 0
$1 = (struct Node *) 0x0
(gdb) s
test-list: list.c:39: MakeEmpty: Assertion `L != ((void *)0)' failed.
Program received signal SIGABRT, Aborted.
0x4003a3c5 in __kill ()
(gdb) quit
```