Debugging C and C++ programs with gdb (and ddd)

Introduction to gdb and ddd

The purpose of a debugger is to allow you to see what is going on inside your C program while it runs. In addition, you can use gdb to see what your program was doing at the moment it crashed.

Here are some of the usful actions that gdb can perform:

- Start your program and step through it line by line
- Make your program stop on specified conditions
- Show the values of variables used by your program
- Examine the contents of any frame on the call stack
- Set breakpoints that will stop your program when it reaches a certain point. Then you can step through part of the execution using step and next, and type continue to resume regular execution.

For C and C++ programs, gdb and ddd are debuggers that you can use. `ddd` is a easy-to-use GUI wrapper around an inferior debugger (gdb for GNU compiled C or C++ code). `ddd` allows you to interact with the debugger by using either GUI menu options or the under-lying debugger's command line interface. In addition, `ddd` automatically displays source code when breakpoints are reached.

There are some example programs and some documentation on using gdb to debug them that you can copy from here:
`/home/newhall/public/gdb_examples/`

Getting started with gdb

C and C++ programs compiled with the GNU compiler and the `-g` option can be debugged using GNU's debugger `gdb` (actually, you can use gdb on code that is not compiled with -g, but unless you like trying to figure out how assembly code sequences map to your source code I wouldn't recommend doing so). Also, do not compile with an optimization flag (i.e. don't use -O2), or gdb will have a hard time mapping optimized machine code to your source code. For example:

```
% gcc -g myprog.c
```

To start gdb, invoke gdb on the executable file. For example:

```
% gdb a.out
```

If your program terminates with an error, then the operating system will often dump a core file that contains information about the state of the program when it crashed. gdb can be used to examine the contents of a core file:

```
% gdb core a.out
```

One good way to get started when you are trying to track down a bug, is to set breakpoints at the start of every function. In this way, you will quickly be able to determine which function has the problem. Then you can restart the program and step through

the offending function line-by-line until you locate the problem exactly.

ddd is invoked in a similar way:

```
% ddd a.out
```

Common gdb Commands

(printable version here)

```
Commonly used gdb commands
--------------------------
gdb also understands abreviations of commands, so you can just type up to
the unique part of a command name ("cont" for "continue", or "p" for "print")

help                  List classes of all gdb commands
help <topic>          Shows help available for topic or command

where                 Shows stack: sequence of function calls executed so far
(or backtrace)        (good for pinpointing location of a program crash)
(or bt)

frame                 Shows all stack frames
 frame <frame-num>     Sets current stack frame to <frame-num>
info frame            Show state about current stack frame

run                   Starts program at the beginning
 run command line args
continue              Continues execution from breakpoint

break
 break <line>          Sets breakpoint at line number <line>
 break <func-name>     Sets breakpoint at beginning of function <func-name>
  break main            Sets breakpoint at beginning of program

continue              Continues execution from breakpoint

condition <bp-num> <exp>   Sets breakpoint number <bp-num> to break only if
                           conditional expression <exp> is true
info break            Shows current breakpoints
disable [breakpoints] [bnums ...]  Disable one or more breakpoints
enable [breakpoints] [bnums ...]   Enable one or more breakpoints
clear <line>          Clears breakpoint at line number <line>
clear <func-name>     Clears breakpoint at beginning of function <func-name>
delete <bp-num>       Deletes breakpoint number <bp-num>
delete                Deletes all breakpoints

step (or s)           Executes next line of program (steping into functions)
  step <count>          Executes next <count> lines of program
next (or n)           Like step, but treats a function call as a single
  next <count>            instruction
until <line>          Executes program until line number <line>

list                  Lists next few lines of program
  list <line>           Lists lines around line number <line> of program
  list <start> <end>    Lists line numbers <start> through <end>
  list <func-name>      Lists lines at beginning of function <func-name>

print <exp> (or inspect <exp>  Displays the value of expression <exp>

 To print in different formats:
   print/x  <exp> print the value of the expression in hexidecimal  (e.g. print/x 123   displays 0x7b)
   print/t  <exp> print the value of the expression in binary (e.g. print/t 123   displays  1111011)
   print/d  <exp> print the value of the expression as unsigned int format (e.g. print/d 0x1c displays 28)
   print/c  <exp> print the ascii value of the expression (e.g. print/c 99  displays 'c')
   print   (int)<exp> print the value of the expression as signed int format (e.g. print (int)'c' displays 99)

 To represent different formats in the expression (the default is int):
   0x suffix for hex: 0x1c
   0b suffix for binary: 0b101      (e.g. print 0b101   displays 5,   print 0b101 + 3   displays 8)
   you can also re-cast expressions using C-style syntax  (int)'c'
```

```
 You can also use register values and values stored in memory locations in expressions
   print $eax    # print the value stored in the eax register
   print *(int *)0x8ff4bc10  # print the int value stored at memory address 0x8ff4bc10

x <var, memory address> displays the contents of the memory location given a variable name or a memory address.
  Can display in diffent formats (as an int, a char, a string, ...)

  (ex) assume s1 = "Hello There" is at memory address 0x40062d
  x/s s1      # examine the memory location associated with var s1 as a string
    0x40062d   "Hello There"
        x/4c s1     # examine the first 4 chars in s1
    0x40062d   72 'H'   101 'e' 108 'l' 108 'l'
  x/d s1      # examine the memory location assoc with var s1 as an int
    0x40062d   72
  x/8d s1     # the ascii values of the first 8 chars of s1
    0x40062d:   72      101     108     108     111     32      84      104

  # in general x takes up to three arguments:  x/nfu  address
  #    n:  the repeat count
  #    f: the display format (s:string,i:instruction,x:hex,d:decimal,t:binary,...)
  #    u: the units format (b:byte,h:2bytes,w/d:4bytes,g:8bytes)
  #  and the format f is sticky: specify once to set, subsequent x command use it
  x/10dh  0x1234  # prints out 10 short values in decimal format at address 0x1234


  # can also use the address of a varible as the argument (say temp is an int)
  x  &temp

  # NOTE: format in examine is sticky, for example if you use the command x/c
  #       subsequent executions of x will use /c format. you therefore need to
  #       explicitly change the format to /d /c /s etc. for interpreting memory
  #       contents as differnt type from the previous call to x


display <exp>         Automatic display of <exp> each time a breakpoint reached
   display i+1

whatis <exp>          Shows data type of expression <exp>
info locals           Shows local variables in current stack frame
set variable <variable> = <exp>   Sets variable <variable> to expression <exp>
  set x = 123*y                 # set var x's value to 123*y

quit                  Quits gdb
```

info commands for examining runtime and debugger state:

gdb has a large set of info X commands for displaying information about different types of runtime state and about debugger state. Here is how to list all the info commands in help, and a description of what a few of the info commands do:

```
(gdb) help status           # lists a bunch of info X commands

(gdb) info frame            # list information about the current stack frame
(gdb) info locals           # list local variable values of current stack frame
(gdb) info args             # list argument values of current stack frame
(gdb) info registers        # list register values
(gdb) info breakpoints      # list status of all breakpoints
```

using gdb to debug assembly code and examine memory and register values

ddd is probably easier to use when steping through assembly code than gdb because you can have separate windows that show the disassembled code, the register values, and the gdb prompt.

Here are some gdb commands that are useful for debugging at the assembly code level:

```
disass   list the assembly code for a function or range of addresses
  disass <func_name>           lists assembly code for function
  disass <start>  <end>  lists assembly instructions between start and end address

break   Set a breakpoint at an instruction
  break *0x80dbef10   Sets breakpoint at the machine code instruction at address 0x80dbef10
```

```
stepi   Executes the next machine code instruction
nexti   Executes the next machine code instruction treats function call as single instr

info
   info registers        # list register values

print
   print $eax    # print the value stored in the eax register
   print *(int *)0x8ff4bc10  # print the int value stored at memory address 0x8ff4bc10

x      Display the contents of the memory location given an address.
                NOTE: the format is sticky (need to explictly change it)

   x/s 0x40062d      # examine the memory location 0x40062d as a string
     0x40062d   "Hello There"
   x/4c 0x40062d     # examine the first 4 char memory locations starting at address 0x40062d
     0x40062d   72 'H'  101 'e' 108 'l' 108 'l'
   x/d s1            # examine the memory location assoc with var s1 as an int
     0x40062d   72

set set the contents of memory locations and registers
   set $eax = 10                # set the value of register eax to 10
   set $esp = $esp + 4          # pop a 4-byte value off the stack
   set *(int *)0x8ff4bc10 = 44   # at memory address 0x8ff4bc10 store int value 44

display  at every breakpoint display the given expression

   display $eax
```

Here is some more information about: Debugging IA32 Assembly Code with gdb (and ddd)

Sample gdb sessions

Below is output from two runs of gdb on programs from ~newhall/public/gdb_examples/.

1. Run 1 is a gdb run of badprog.c. It demonstrates some common gdb commands, and it finds one of the bugs in this program...there are others.
2. Run 2 is a gdb run of segfaulter.c. It demonstrates how to find out where your program is segfaulting (and perhaps why...although valgrind will help more with this type of error).

Run 1: badprog.c

```
% gcc -g badprog.c           #-- compile program with -g flag


% gdb a.out                  #-- invoke gdb with the executable

GNU gdb 6.4.90-debian
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or 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.
This GDB was configured as "i486-linux-gnu"...Using host libthread_db library "/lib/tls/libthread_db.so.1".

(gdb) break main              #-- set a breakpoint at the begining of the program's execution

Breakpoint 1 at 0x8048436: file badprog.c, line 36.


(gdb) run                     #-- run the program

Starting program: /home/newhall/public/gdb_examples/a.out

Breakpoint 1, main () at badprog.c:36      #-- gdb stops at breakpoint

36        int arr[5] = { 17, 21, 44, 2, 60 };

(gdb) list                                 #-- list the source code near the break point
```

```
31        return 0;
32      }
33
34      int main(int argc, char *argv[]) {
35
36        int arr[5] = { 17, 21, 44, 2, 60 };
37
38        int max = arr[0];
39
40        if ( findAndReturnMax(arr, 5, max) != 0 ) {
```

```
(gdb) list 11                          #-- list source code around line 11

11      // this function should find the largest element in the array and
12      // "return" it through max
13      //      array: array of integer values
14      //      len: size of the array
15      //      max: set to the largest value in the array
16      //      reuturns: 0 on success and non-zero on an error
17      //
18      int findAndReturnMax(int *array1, int len, int max) {
19
20        int i;
```

```
(gdb) list                             #-- list the next few lines of code
21
22        if(!array1 || (len <=0) ) {
23          return -1;
24        }
25        max = array1[0];
26        for(i=1; i <= len; i++) {
27          if(max < array1[i]) {
28            max = array1[i];
29          }
30        }
```

```
(gdb) next                             #-- execute the next instruction
38        int max = arr[0];

(gdb)                                 #-- hitting Enter executes the previous command (next in this case)
40        if ( findAndReturnMax(arr, 5, max) != 0 ) {
```

```
 #-- also you can use the up and down arrows to scroll through previous commands
```

```
(gdb) print max                        #-- print out the value of max
$1 = 17

(gdb) p arr                            #-- p is short for the print command
$2 = {17, 21, 44, 2, 60}
```

```
(gdb) step        #-- step into the function call
                  #-- if we had entered 'next' the entire function call would have been executed

findAndReturnMax (array1=0xbfc5cb3c, len=5, max=17) at badprog.c:22
22        if(!array1 || (len <=0) ) {             #-- 'step' takes us to the entry point of findAndReturnMax

(gdb) print array1[0]                   #-- lets see what the param values are
$3 = 17
(gdb) p max
$4 = 17

(gdb) list
17      //
18      int findAndReturnMax(int *array1, int len, int max) {
19
20        int i;
21
22        if(!array1 || (len <=0) ) {
23          return -1;
24        }
25        max = array1[0];
26        for(i=1; i <= len; i++) {
```

```
(gdb) break 26                          #-- set a breakpoint at line 26 (inside findAndReturnMax)

Breakpoint 2 at 0x80483e7: file badprog.c, line 26.

(gdb) cont                              #-- continue the execution
Continuing.

Breakpoint 2, findAndReturnMax (array1=0xbfc5cb3c, len=5, max=17)    #-- gdb hits the next breakpoint
    at badprog.c:26
26        for(i=1; i <= len; i++) {

(gdb) p i
$5 = 0

(gdb) n                                 #-- n is short for next
27          if(max < array1[i]) {

(gdb) display max                       #-- display will print out the value everytime we hit a breakpoint
1: max = 17

(gdb) display array1[i]
2: array1[i] = 21

(gdb) break 27                          #-- set a breakpoint inside the loop
Breakpoint 3 at 0x80483f0: file badprog.c, line 27.

(gdb) cont                              #-- continue execution
Continuing.

Breakpoint 3, findAndReturnMax (array1=0xbfc5cb3c, len=5, max=21)
    at badprog.c:27
27          if(max < array1[i]) {
#-- display prints these out:
2: array1[i] = 44
1: max = 21

(gdb) cont
Continuing.

Breakpoint 3, findAndReturnMax (array1=0xbfc5cb3c, len=5, max=44)
    at badprog.c:27
27          if(max < array1[i]) {
2: array1[i] = 2
1: max = 44

(gdb) cont
Continuing.

Breakpoint 3, findAndReturnMax (array1=0xbfc5cb3c, len=5, max=44)
    at badprog.c:27
27          if(max < array1[i]) {
2: array1[i] = 60
1: max = 44

(gdb) cont
Continuing.

Breakpoint 3, findAndReturnMax (array1=0xbfc5cb3c, len=5, max=60)
    at badprog.c:27
27          if(max < array1[i]) {
2: array1[i] = 17
1: max = 60                     #-- so max is 60 here

(gdb) where                 #-- show the stack frames

#-- findAndReturnMax is the active function at line 27, it was called by main at line 40:
#0  findAndReturnMax (array1=0xbfd043ec, len=5, max=60) at badprog.c:27
#1  0x08048479 in main () at badprog.c:40

 frame 1                        #-- move into main's calling context (stack frame 1) to examine main's state
#1  0x08048479 in main () at badprog.c:40
40        if ( findAndReturnMax(arr, 5, max) != 0 ) {
```

```
(gdb) print max                 #-- in main's stack frame max is 17
$1 = 17


(gdb) cont                      #-- continue execution
Continuing.
max value in the array is 17            #-- main prints out value of max after function call

#-- This looks like a bug:"
#-- findAndReturnMax set max to 60, but 60 isn't getting "passed back" to main after the call
#-- to fix this we need either have findAndReturnMax return the value of max or pass max by reference


(gdb) quit                      #-- quit gdb

The program is running.  Exit anyway? (y or n) y
```

Run 2: segfaulter.c

```
 % gdb segfaulter

GNU gdb 6.4.90-debian
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or 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.
This GDB was configured as "i486-linux-gnu"...Using host libthread_db library "/lib/tls/libthread_db.so.1".

 (gdb) run              #-- just run segfaulter and let it seg fault

Starting program: /home/newhall/public/gdb_examples/segfaulter
Failed to read a valid object file image from memory.

Program received signal SIGSEGV, Segmentation fault.
0x080483e1 in initfunc (array=0x0, len=100) at segfaulter.c:15
15          array[i] = i;

 (gdb) where           #---  let's see where it segfaulted

#0  0x080483e1 in initfunc (array=0x0, len=100) at segfaulter.c:15
#1  0x0804846e in main () at segfaulter.c:38

(gdb) list             #--- let's see code around segfaulting instruction

10      int initfunc(int *array, int len) {
11
12        int i;
13
14        for(i=1; i <= len; i++) {
15          array[i] = i;
16        }
17        return 0;
18      }
19

 (gdb) p array[0]          #--- let's print out some values and see what's going on
Cannot access memory at address 0x0

#-- it looks like array is a bad address (0x0 is NULL)

 (gdb) p array
$1 = (int *) 0x0

 (gdb) frame 1             #--- let's see what main is passing to this funtion

#1  0x0804846e in main () at segfaulter.c:38
38        if(initfunc(arr, 100) != 0 ) {

 (gdb) print arr           #--- print out arr's value (what we pass to initfunc)
$2 = (int *) 0x0
```

```
#--- oops, we are passing NULL to initfunc...we forgot to initialize arr to point to valid memory
```

Keyboard shortcuts in gdb

gdb supports **command line completion**; by typing in a prefix you can hit `TAB` and gdb will try to complete the command line for you.

Also, you can give just the **unique prefix** of a command as the command and gdb will execute it. For example, rather than entering the command `print x`, you can just enter `p x` to print out the value of x.

The **up and down arrow keys** can be used to scroll through previous command lines, so you do not need to re-type them each time.

If you just hit `RETURN` at the gdb prompt, gdb will execute the **most recent previous command** again. This is particularly useful if you are steping through the execution, then you don't have to type `next` each time you want to execute the next instruction, you can just type it one time and then hit RETURN.

Setting conditional breakpoints and some issues with setting breakpoints in C++ code

conditional breakpoints

A conditional breakpoint is one that only transfers control to gdb when a certain condition is true. This can be very useful when you only want gdb control after iteration 1000 of a loop, for example.

To set a condition on a breakpoint, use the condition command with the number of the breakpoint followed by the condition on which to trigger the breakpoint. Here is an example where I'm setting a conditional breakpoint that will only be triggered when the condition (i >= 1000) is true:

```
(gdb) break 28                                  # set breakpoint at line 28
(gdb) info break                                # list breakpoint information
  Num Type           Disp Enb Address    What
   1   breakpoint     keep y   0x080483a3 in foo at loops.c:28

(gdb) condition 1 (i >= 1000)                # set condition on breakpoint 1
(gdb) run   (or continue if already running)
```

breakpoints in C++ programs

One complication with gdb and C++ programs, is that you need to specify methods and data members using the "classname::" prefix. In addition, you often need to use a leading ' before a name for gdb to find the symbol, and if methods are overloaded, you need to specify which method it is by listing its full prototype (actually, if you hit `TAB` gdb will list all possible matches for you and you can pick one of those).

For example, to set a break point in funciton pinPage of the BufMgr class, I'd do the following:

```
(gdb) break 'BufMgr::pinPage(int, Page *&, int)'
```

This looks pretty icky, but really I just type `break 'BufMgr::p` then hit `TAB` for automatic completion.

```
(gdb) break 'BufMgr:: <tab>
```

will list all methods of the BufMgr class, then you can just pick from the list the method you want to put the breakpoint in.

---

gdb and make

Within gdb you can invoke "make" to rebuid your executable (assuming that you have a makefile to build your program). This is a nice feature in the case when you have many breakpoints set and do not want to exit gdb, recompile, re-start gdb with the new a.out, and reset all the breakpoints. However, keep in mind that modifying and recompiling your source code from within gdb may result in your breakpoints not being where you think they should be (adding/removing lines of source code can result in your

in your breakpoints no longer being where you want them to be in terms of the new version of your source code). You can use the disable or delete commands to disable or delete old breakpoints.

Some Advanced Features

attaching gdb to a running process

1. get the process's pid

```
# ps to get process's pid
$ ps                    # lists all processes started in current shell
$ ps -A | grep a.out    # list all processes pipe through grep for just those named a.out
  PID TTY          TIME CMD
12345 pts/3     00:00:00 a.out
```

2. attach gdb to the running process

```
# gdb <executable> <pid>
$ gdb a.out 12345

# OR alternative syntax:

# gdb attach <pid>  <executable>
$ gdb attach 12345 a.out
```

   At this point the process is stopped by gdb; you have the gdb prompt that you can use issue gdb commands like setting breakpoints, or printing out program state before continuing execution.

Depending on if the process was explicitly stopped before attaching gdb or not (e.g. did the process call kill(getpid(), SIGSTOP) to stop itself like in the `attach_example.c`) you can continue its execution from the gdb prompt in one of two ways:

```
(gdb) cont                # try this first
(gdb) singal SIGCONT      # try this if the cont command doesn't work
```

following a process on a fork

You can set gdb to follow either the parent or the child process on a fork (the default is to follow the parent): By setting breakpoints in the child path of code, and then doing:

```
(gdb) set follow-fork-mode child
```

gdb will follow the child process after the fork. See debugging forks for more information.

signal control

In gdb you can send the process a signal:

```
(gdb) signal SIGCONT
(gdb) signal SIGALARM
...
```

Sometimes your process receives signals and you would like to have gdb perform some action when certain signals are delived to the debugged process. For example, if your program issues a bad adress, it will receive a SIGBUS signal and usually exit. The default behavior of gdb on a SIGBUS it to let the process exit. If, however, you want to examine program state when it recieves a SIGBUS, you can specify that gdb handle this singal differently:

```
(gdb) handle SIGBUS stop             # if program gets a SIGBUS, gdb gets control
```

You can list how gdb is handling signals using info:

```
(gdb) info signal    # list info on all signals

(gdb) info SIGALRM   # list info just for the SIGALRM signal
```

---

ddd settings and bug fixes

Running ddd creates a .ddd directory in your home directory and will save settings to files here, so that you don't need to reset all your preferences from scratch. You can click and drag to change the sizes of subwindows and choose Menu options to display (or not) certain menus, register values, machine code, etc.

- To view assembly code: under Source menu choose "Display Machine Code" or Alt+4
- If ddd hangs with "Waiting until gdb ready" message, then one way to fix this is to wipe out your .ddd directory (you will lose all your saved settings):

```
rm -rf ~/.ddd
```

---

gdb Links

common gdb commands (from above)
example gdb sessions (from above)
GDB quick reference card
A very complete GDB reference
Using GDB within Emacs by Ali Erkan