CPSC 240: Lecture on GDB

Starting

Place -g in all compile, assemble, and link statements.

Eg.: nasm -f elf64 -g -o control.o control.asm -l control.lst

Execute the program with gdb on the left. The command for execution can be placed in the master bash file.

Eg.: gdb ./assign3.out

Prologue

Create some break points. Usually you will place breakpoints on statements or instructions where you suspect errors. If you don't know where to place breakpoints then at least place one break point on the first executable statement such as the statement "main".

b main //Create break on the line containing main

b 12 //Create break on line number 12

b fill.cpp:10 //Create break on line number 10 in the file fill.cpp

i b //Show a list of all existing breaks

d 12 //Delete break number 12 according to output from "i b"

clear fill.cpp:10 //Delete the break in file fill.cpp at line 10

//Notice that breaks are created according to line number (or called function name). However, breaks are only deleted by break number. If a break is created by "b main" then "d main" does nothing. That break must be deleted with a break number as in "d 3".

//The clear operator requires a file name and a line number. Clear does not use break numbers. Examples: "clear sum.cpp:18" removes the break at line 18 within the file sum.cpp. "clear 18" does nothing.

Execution controls

r //Start the program running under the control of gdb

q //Quit. Stop execution of the program. Reverse of the 'r' command.

q //Quit and exit from gdb

//Notice that there are two uses of quit: one terminates the execution of the program being debugged but gdb continues live while the other use of quit causes gdb to exit.

Stepping through execution

n //Execute the next statement or instruction. The command 'n' will execute a function call, but it will not step into that function. Suppose your program calls printf at a certain line of the source file. You want gdb to execute printf, but you probably do not wish to see all the details regarding printf.

s //Step ahead to the next instruction. Recommended for stepping through an assembly module.

c //Continue execution without pause until the next break is reached.

Display data values

Let a and b be an ordinary variables in a program being executed by gdb. For example:
long a = 37; or char b[ ] = “Hello”;

p/d a //Output contents of a in decimal (base 10)

p/t a //Output contents of a in binary (base 2)

p/s a //Output the contents of a in ascii characters.
//For example, suppose a holds 0x4142. Then the output will be AB.

Table of symbols of output formats. These govern the form of the output and are for use with the ‘p’ command.

d	decimal
c	char
a	address
s	string
i	instruction
f	float
u	unsigned integer of any size
t	binary

Pick any general register: we’ll use rbp for illustration purposes.

p/x $rbp //Output the value in rbp in hex

p/u $rbp //Output the value in rbp in unsigned decimal integer form

p/s $rbp //Output the value in rbp as a string where each byte is outputted as an ascii character

Display addresses

The value of every variable is stored in the Stack Frame associated to the process where the variable is declared. Thus every variable has an address. Registers don’t have addresses.

p/x &a //Output the location of a in hex

p/u &a //Output the location of a in unsigned integer

p/a &a //Output the address of a in address format.

Memory and Arrays

An array is simply a contiguous section of memory. For that reason the two concepts are treated together in this section.

For this narrative let these be array declarations:

```
long numb[12];              //In C++ 'long' means qword.
double number[100];    //In C++ 'double' means qword.
float num[16]                     //In C++ 'float' means dword.
unsigned int numbers[60];     //In C++ 'int' means dword.
short nu[24]                      //In C++ 'short' means word
char n[32]                    //In C++ 'char' means byte
```

In X86 assembly the declarations above would be written like the following:

```
segment .bss
        numb  resq  12
        number  resq  100
        num  resd  16
        numbers  resd  60
        nu  resw  24
        n  resb  32
```

Here are examples of outputting values from an array using the 'p' command.

```
p/x  number       //Output all the values of number in hex integers

p/d  number       //Output all the values of number in decimal integers

p/d  number[3]   //Output the one value in cell #3 of number

p/t  number       //Output all the values of number in binary integers

p/f  number       //Output all the values of number in decimal floating points

p/x  &number     //Output the starting address of number in hex

p/d  &number     //Output the starting address of number in decimal

p/a  &number[2]  //Output the address of cell #2 of number in hex

p/d  &number[3]  //Output the address of cell #3 of number in decimal
```

Here are examples of outputting values from an array using the ‘x’ command.

x/4xw number   //Output the first 4 dwords (32 bits) of the array in hex form.

x/5dh number   //Output the first 5 words (16 bits) of the array in decimal

x/7dg number   //Output the first 7 dwords (32 bits) of the array in decimal.

x/8xw number   //Output the first 8 dwords (32 bits) of the array in hex

x/9xg number   //Output the first 9 qwords (64 bits) of the array in hex

x/5xw number   //Output the first 5 dwords (32 bits) of the array in hex

x/4gx number   //Output the first 4 qwords of the array in hex

An example of changing the value in one cell of the array number.

set var number[3] = 14   //Change the value of cell #3 to become 14.

//Although the C++ source program declared number to be an array of doubles gdb has placed an integer in that array.

//Prefixing with an ‘&’ creates the same output as no prefix. Examples:
//   x/5d numb and   x/5d &numb   both produce the same output..

Table of symbols used by gdb to specify the size of a segment of memory

Symbol	Name	Bytes
b	byte	1
h	halfword	2
w	word	4
g	giant	8

The definitions of halfword, word, and giant are specific to gdb; these are not universal definitions. The symbols in the table can be used only with the ‘x’ operator, If used with the ‘p’ operator a warning message will appear.

x/40xg $rsp //Output 40 contiguous giant words in hex starting at the address in rsp.

x/30dg 0x7fffffffaa80 //Output 30 contiguous giant words in decimal starting at the address 0x7fffffffaa80

x/32xw 0x7FFFFFFF0000 //Output 32 contiguous words (4 bytes each) in hex beginning at the given address. In this course we usually don’t use 4 byte data items of any kind, but in industry they’ll use data of all sizes.

x/100xg rip //Output 100 qwords in hex starting at the address of the next instruction to be executed. This is useful for finding opcodes in the executable section of memory, which may be later modified.

x/xg rip //The same as “x/1xg rip”. When no integer indicating the number of numeric units to be displayed is given then the default value ‘1’ is assumed.

Change Values

set $r14 = 32 //Change the value stored in register r14

set $rdx = 0x46 //Change the value stored in register rdx

set var a = 29 //Change the value in program variable a to become 29

set var numb[2] = 3.75//Change the value in cell #2 to 3.75

set {unsigned long}0x7fffffff5a40 = 99 //Change value in memory to 99

//The following is invalid: set 0x7fffffff5a40 = 99
because the size of the destination is not specified. The prefix {unsigned long} is a way of specifying that 8 bytes (1 qword) will be used in the destination for storage of the value 99.

set {unsigned int}0x420a60 = 3000

//Use a 4 bytes of memory starting at 0x420a60 to store the number 3000. In this course we're not using 4 byte numbers.

To: Students enrolled in 240:

The document you are reading was created from a number of sources – some on line sources and one in paper book form. This document was used for lectures during three days of the current semester.

To give coherent, accurate, and meaningful presentations about GDB is an ongoing task. I am still refining how to best present GDB.

This document does not cover all of GDB. The document does provide sufficient basic coverage for the ordinary programmer to use GDB to debug ordinary programs. It is not finished. I will be adding to it as additional useful commands come to my attention.

Which programming language support the use of GDB? There are about 12 at the present time. Use this link to see the complete list: https://www.gnu.org/software/gdb/

If you use any part of this lecture document I welcome your comments. Feel free to tell me via email what was helpful, what was not helpful, and what should be included.

Enjoy your programming because it is fun, that is, it is fun when it finally works.

Floyd Holliday, Fall semester 2019.

Mainto: **holliday@fullerton.edu**