CPSC 240: Lecture on GDB

Starting

When compiling C or C++ files place the "-g" switch in the call to the compiler as shown in the following example.

g++ -c -m64 -Wall -no-pie -o gdb-demo.o -std=c++17 gdb-demo-arrays.cpp -g

Place -gdwarf all calls to the nasm assembler as shown in the next statement:

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

//dwarf is a kind of format for the object file that will be created. The dwarf for format is better suited for use with gdb than other formats. Use of "-g" in place of "-gdwarf" does not result in an object file satisfactory for use with gdb.

Ref: https://gcc.gnu.org/onlinedocs/gcc/Debugging-Options.html

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 to live while the other use of quit causes gdb to exit.

Stepping through execution

n //Execute the next statement or instruction. If the next statement is a call to a program function compiled with the “-g” parameter then enter the called function for debugging purposes. If the next statement or instruction is a call to a library function ‘n’ will execute that function without stepping through it.

ni //Shows the statement or instruction that ‘n’ will execute if ‘n’ were to be inputted to gdb

s /Execute the next instruction. If the next statement is a call to a program function compiled with the “-g” parameter then enter the called function for debugging purposes. If the next instruction is a call to a library function then enter that function for debugging purposes.

si Show the statement that will execute if ‘s’ is the next command entered.

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

If a member function of the program (not a library function) is compiled without the “-g” parameter then both ‘n’ and ‘s’ will execute the function without debugging the function.

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. These are the type specifies for the second parameter of the ‘x’ command that will be introduced later in this document.

d	signed integer
c	char
a	address
s	string
i	instruction
f	float
u	unsigned integer
t	binary
x	hex

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

Pick any SSE register: we’ll use xmm2 for illustration purposes

p $xmm2 //Show the contents of xmm2 using defaults

p/d $xmm2 //Show the contents of xmm2 where all values are outputted in twos complement decimal format.

p/t $xmm2 //show the contents of xmm2 where all values are outputted in twos complement binary format

p/x $xmm2 //show the contents of xmm2 where all values are outputted in twos complement hex format.

p/f $xmm2.v2_double //Outputs exactly two values: the low 64 bits of xmm2 followed by the high 64-bits of xmm2.

To view the data in an xmm register according to a specific struct it is necessary to append the name of that struct at the end of the name of the register. This is shown in the preceding example.

The examples above show some of the major forms of displaying data, but it is not an exhaustive list.

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.

Display entire sets of registers

Be aware that the registers in SSE (xmm's) and the registers in AVX (ymm's) are viewed according to various structs. The commands that output SSE and AVX registers show the stored values as formatted by these structs. At first view the output appears overwhelming. Study the output and eventually it will all start to make sense.

i r //Show all GPRs

i r f //Show all FPU registers

i r s Show all SSE registers, that is, all xmm registers

i r a //Show all registers in the X86 processor including ymm registers; 'a' = all

Memory and Arrays in C & C++

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:

Here are examples of outputting values from arrays.

```
p/x numb          //Output all the values of numb in hex

p/d numb          //Output all the values of numb as integers base 10.

p/t  numb         //Output all the values of numb in binary (big endian)

p/f number        //Output all the values of number in floating point base 10.

p/x number        //Show all values after truncating to integers
```

Introduction to the 'x' command.

In its general form the command uses 3 parameters:

- First parameter: a positive integer indicating the number of groups of output
- Second parameter: a single char indicating the type of each group of output.
- Third parameter: a single char indication the size of each group of output.

The third parameter may be omitted and gdb will output using a default size for each group. We don't investigate defaults for omitted parameters here.

```
x/4xg number      //Show first 4 values in the array number in groups of qword in each group.

x/7dw  number     //Output the first 7 dwords (32 bits) of the array in integer base 10.

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/4xg numbers //Output the first 4 qwords of the array in hex

//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. These are the size specifiers for the 'x' command.

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 groups to be displayed is given then the default value '1' is assumed.

Arrays in Assembly X86

Arrays in assembly are also contiguous regions of memory. The structure of the array is clearly seen at the time of declaration.

In segment .data we can declare initialized arrays such as the following:

```
welcome db “Welcome to Assembly Assignment”,10,0          //cell size = 1 byte

points dw 36, 44, 18, 59, 87, 3, 68, 97, 67, 31           //cell size = 2 bytes

distances dd 39000, 65500, 104475, 600                    //cell size = 4 bytes

weights dq 9677000000, -1, 44, 855999888, -2, 77    `     //cell size = 8 bytes

costs dq 0x4060000000000000, 0x4168888888888888,
0x3FD8111111111111, 0x001C000000000000                    //Array initialized with 64-
```

bit floating point values and cell size is 8 bytes. There are 16 hex digits in each cell of the array.

Here are gdb commands for viewing the array welcome defined above. The ‘8’ indicates that the first 8 bytes of the array are to be displayed.

```
p/c {char[8]}&welcome

p/s {char[8]}&welcome

p/d {char[8]}&welcome

p/x {char[8]}&welcome

p/t {char[8]}&welcome

p/x &welcome                    //Outputs address where array data begins

p/c ({char[5]}&welcome)[3]      //Outputs the char of the array at index 3.
```

Here are gdb commands for viewing the array points defined above. The ‘9’ indicates that the first 12 words of the array are to be displayed. If the 9 is omitted then the default value 1 is assumed. The type “short” specifies the cell size to be two bytes.

```
p/d {short[9]}&points

p/s {short[9]}&points

p/x {short[9]}&points

p/t {short[9]}&points
```

```
p/x &points                  //Outputs the starting address of the data of the array points

p/x ({short[8]}&points)[4]   //Outputs the single value at index number 4.
```

Here are gdb commands for viewing the array distances defined above. The '4' indicates that the first 4 words of the array are to be displayed. If the 4 is omitted then the default value 1 is assumed. The type "int" specifies the cell size to be four bytes.

```
p/d {int[4]}&distances

p/s {int[4]}&distances

p/x {int[4]}&distances

p/t {int[4]}&distances

p/x &distances               //Outputs the starting address of the data of distances

p/x ({int[2]}&distances)[3]  //Outputs the single value at index number 3.
```

Here are gdb commands for viewing the array weights defined above. The '3' indicates that the first 3 double words of the array are to be displayed. If the 3 is omitted then the default value 1 is assumed. The type "long" specifies the cell size to be eight bytes.

```
p/d {long[3]}&weights

p/s {long[3]}&weights

p/x {long[3]}&weights

p/t {long[3]}&weights

p/x &weights

p/x ({long[3]}&weights)[5]          //Outputs the single value at index number 5
```

Here are gdb commands for viewing the array costs defined above. The '4' indicates that the first 4 double words of the array are to be displayed. If the 4 is omitted then the default value 1 is assumed. The type "long" specifies the cell size to be eight bytes. That declaration 'long' is independent of the fact the array contains floating point values.

```
p/d {long[4]}&costs          //The bits are interpreted as signed decimal integers

p/s {long[4]}&costs

p/x {long[4]}&costs

p/t {long[4]}&costs
```

```
p/x &costs

p/f {long[4]}&costs          //View the data as decimal floating point numbers

p/f ({long[4]}&costs)[2]     //View only the data in the cell with index number 2
```

This is a work in progress. This is not done. The examples above show arrays declared in the segment .data.

This document needs a section on arrays declared in segment .bss. There are some small differences with arrays in .bss compared with arrays in .data.

I am simply out of time. This will have to be continued another day.

To the students of CPSC 240 in Spring 2020. I hope this document proves to be helpful. In your future career you will perhaps remember that you once learned command-line GDB.

Yes there are other debuggers in the market place of software, but GDB is the one that is free. I like free.

To all my students: this is a free document. You may modify it and you may post it anywhere. It would be nice if when you post it someplace that you leave one little line at the end with words like "Based on earlier work by Floyd Holliday (2020)".

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 number[2] = 3.75 //Change the value in cell #2 to 3.75

set var number[3] = 2.65 //Change the value of cell #3 to become 2.65 .
GDB knows to modify the qword of the array starting at quadword number 3 because gdb knows that the array number was declared to have quadword-size cells.

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 4 bytes of memory starting at 0x420a60 to store the number 3000. [In the course 240 we're not using 4 byte numbers.]

//To change the value in an SSE register one needs to name the register, the struc, and the index of the value to be modified.

set $xmm2.v2_int64[1]=400

//Change the low 64 bits (low 8 bytes) of xmm2 to 400 decimal.

set $xmm2.v2_double[1]=2.5 //?Needs to be checked.

//Change the low 64 bits of xmm2 to 2.5.

set $xmm2.uint128=126748

//Special case of unit128: Because there is only one value in the struct called unit128 it is necessary to omit the index integer. That means that $xmm2.uint128[0]=126748 is syntactically wrong.

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,
First version of this document: October 2019.
Significant upgrade: April 23, 2020.

Mainto: **holliday@fullerton.edu**