THE UNIVERSITY OF ARIZONA
UA South

CYBV 471-Week 4

GDB debugger, debug C and assembly language programs and disassemble C-programs

Agenda

- **What is a debugger**
- **Static Debugging**
- **Dynamic Debugging**
- **Symbol table**
- **Running gdb debugger**
- **Mange breakpoints**
- **Examine memory contents**
- **Display register values**
- **Disassemble c program**

Finding an Error in a Program

- What is debugging?
  - Trying to determine the root cause of an error and fixing it
- Static Debugging
- Dynamic Debugging

Static Debugging

- Review the actual programming code
- While compilation, use all types of compiler flags so that it generates all possible warnings
- To check some variables' values, insert several "printf" statements inside the code for confirming the expected outputs
- Add several comments
- Keep reviewing the code

Limitations of Static Debugging

With static debugging, there are things you cannot examine:

- Interaction of the code with libraries
- Memory layout and memory bugs

Dynamic Debugging

- Dynamic debugging allows you to:
  - Observe a program as it runs and to control the program's execution
  - Execute a code step-by-step
  - Execute group of codes (by inserting breakpoints)
  - Step in a function
  - Step-out a function
  - Examine variables' values
  - Modify variables' values
  - Check the memory contents
  - Display registers' values

Dynamic Debugging

- To execute dynamic debugging, you need:

1- Enable debugging option during the compilation step

- Use –g flag with the complier (e.g. gcc –g)

- G flag allows the compiler to keep symbol table with the executable file

2- Use a debugger

- GDB in Linux

- Microsoft Studio in Windows

Symbol Table

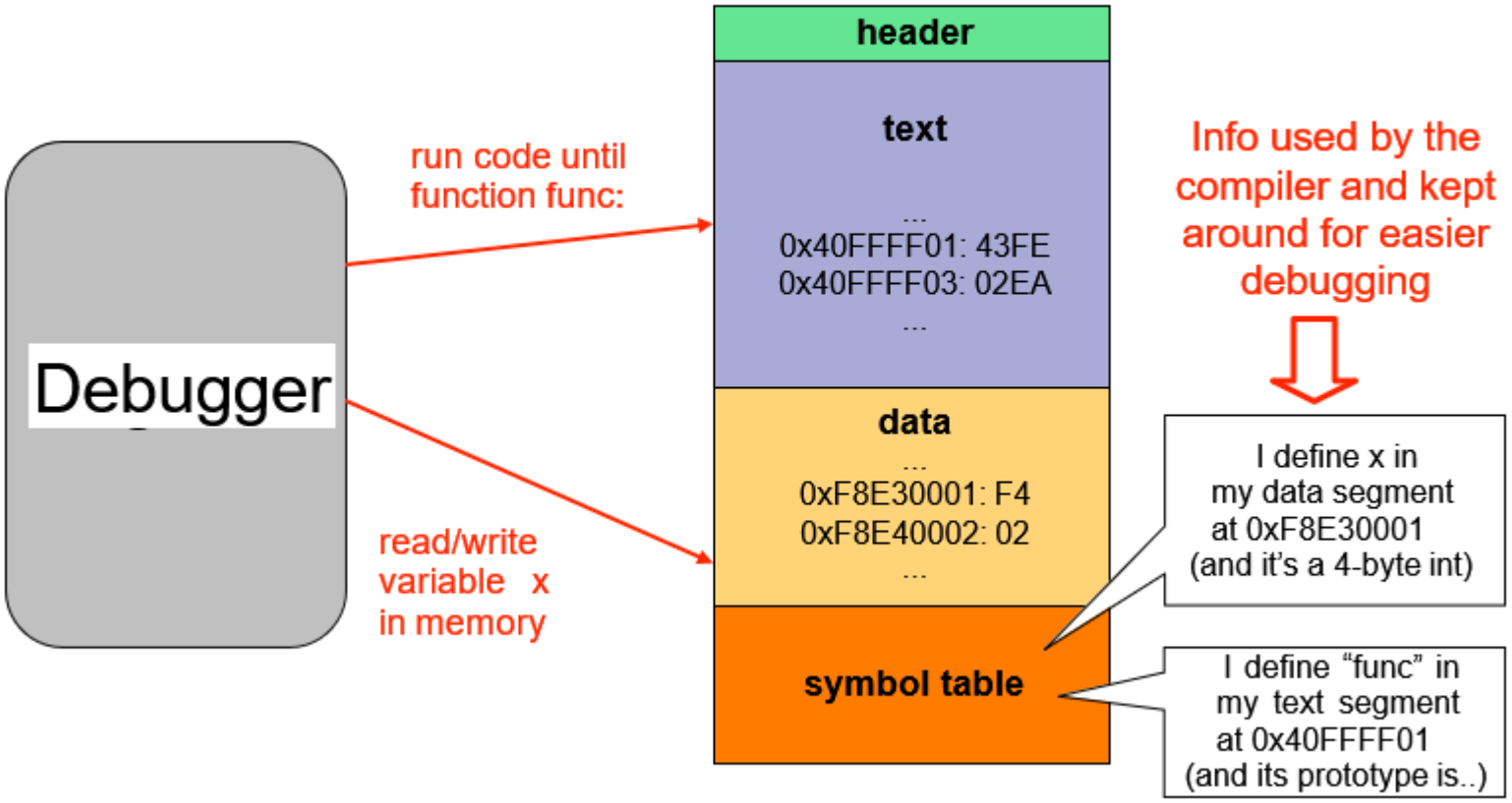
A symbol table is a data structure used by a compiler to identify each symbol in a program's code and its associated information relating to its declaration or appearance in the source.

```
// Declare an external function
extern double bar(double x);

// Define a public function
double foo(int count)
{
    double  sum = 0.0;

    // Sum all the values bar(1) to bar(count)
    for (int i = 1;  i <= count;  i++)
        sum += bar((double) i);
    return sum;
}
```

Symbol name	Type	Scope
bar	function, double	extern
x	double	function parameter
foo	function, double	global
count	int	function parameter
sum	double	block local
i	int	for-loop statement

Symbol Table

- A symbol table may only exist in memory when creating object file for later use.
- It is used during an interactive debugging session
- In general the symbol table is removed from the final executable



User Mode C-Program

- Create cprogram1.c
- Compile the program
- Run the program

```
// cprogram1.c //

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

int add (int x, int y)
{
     int result = 0;
     result = x + y;
     return result;
}

main (int argc, char **argv)
{
    int a = atoi(argv[1]);
    int b = atoi(argv[2]);
    int c;
    char buffer [100];

    gets(buffer);
    puts(buffer);

    c = add (a, b);

    printf("Sum of %d and %d = %d\n", a, b, c);
    exit (0);
}
```

Compile the C-Program

- Compile the C-program
- Run the program

```
root@kali-Test:/# gcc -g cprogram1.c -o cprogram1
cprogram1.c:13:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
 main (int argc, char **argv)
 ^~~~
cprogram1.c: In function ‘main’:
cprogram1.c:20:5: warning: implicit declaration of function ‘gets’; did you mean ‘fgets’
? [-Wimplicit-function-declaration]
     gets(buffer);
     ^~~~
     fgets
/tmp/cc2HY9jG.o: In function `main':
//cprogram1.c:20: warning: the `gets' function is dangerous and should not be used.
root@kali-Test:/#
```

```
root@kali-Test:/# ./cprogram1 10 20
Hello
Hello
Sum of 10 and 20 = 30
root@kali-Test:/#
```

Running a Debugger With Executable File

- To run the program alone, type ./cprogram1 10 20
- To debug the program and run the debugger with cprogram1, type

  gdb ./cprogrm1 (without or with arguments) gdb ./cprogram1 10 20
- In this case, the debugger (dbg) is controlling the execution of cprogram1
- We are now within the "gdb" prompt and we can use "gdb" commands to execute different tasks within cprogram1

```
File  Edit  View  Search  Terminal  Help
root@kali-Test:/# gdb ./cprogram1 10 20
Excess command line arguments ignored. (20)
GNU gdb (Debian 7.12-6) 7.12.0.20161007-git
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
---Type <return> to continue, or q <return> to quit---
```

```
File  Edit  View  Search  Terminal  Help
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./cprogram1...done.
Attaching to program: /cprogram1, process 10
ptrace: Operation not permitted.
/10: No such file or directory.
(gdb)
```

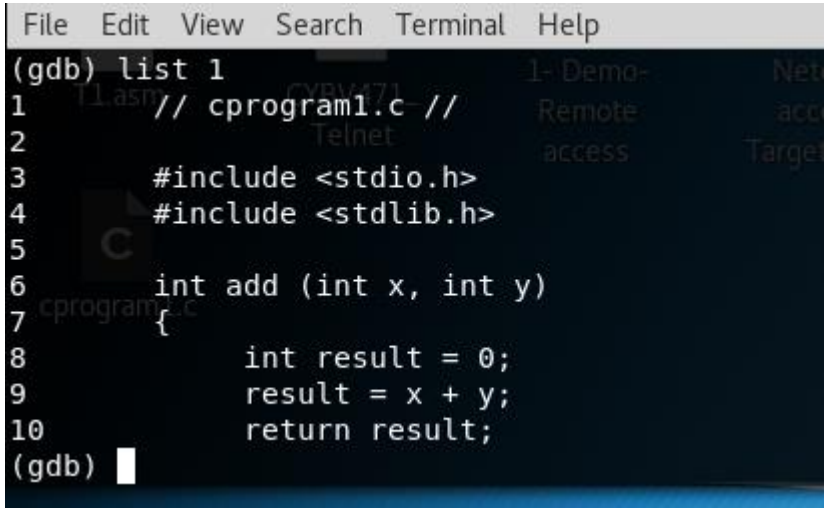
GDB Debugger Commands: list (or l)

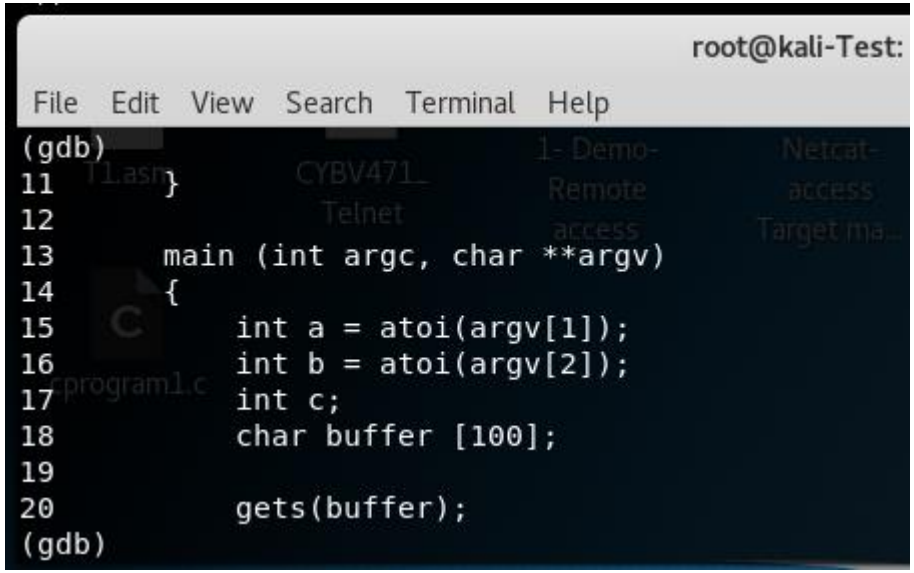
- List the source file: To list around 10 lines of the source file from line 1, type

  (gdb) list 1

  or (dgb) list

- Pressing <enter>, displays the next few lines of source code





GDB Debugger Commands: list (or l)

- List the source file: To list the source file from line n, type

  (gdb) list n or (gdb) l n

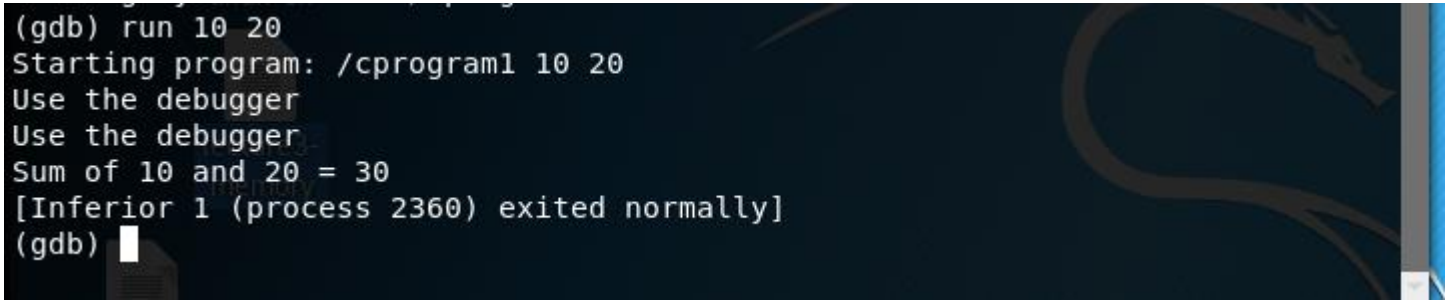
- Pressing <enter> or type list, displays the next few lines of source code

```
File  Edit  View  Search  Terminal  Help
(gdb) list 10
5
6       int add (int x, int y)
7       {
8            int result = 0;
9            result = x + y;
10           return result;
11      }
12
13      main (int argc, char **argv)
14      {
(gdb)
```

```
13      main (int argc, char **argv)
14      {
(gdb) list 10-15
Function "10-15" not defined.
(gdb) list
15           int a = atoi(argv[1]);
16           int b = atoi(argv[2]);
17           int c;
18           char buffer [100];
19
20           gets(buffer);
21           puts(buffer);
22
23           c = add (a, b);
24
(gdb) list
25           printf("Sum of %d and %d = %d\n", a, b, c);
26           exit (0);
27      }
28
(gdb)
```

GDB Debugger Commands: run (or r)

- Run (starts) the program
  - You should provide the required command-line arguments
  - The program will run all the way through to the end of the program



Run the program for specific code (**Breakpoints**)

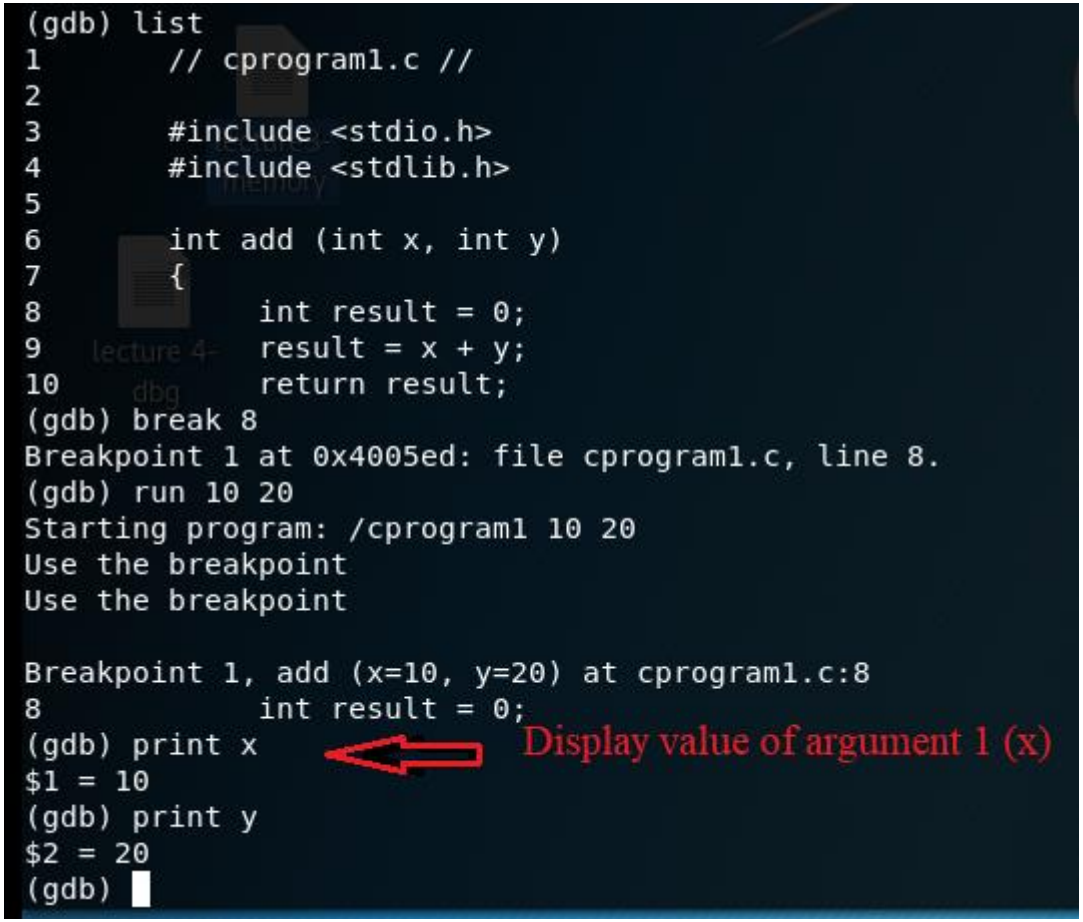
- Breakpoints allow you to stop the program execution at specific location in the program:
- The program start executing from the beginning of the program until the breakpoint

- For example, to run the program from the beginning until code line = 8, use

  (gdb) break 8
- Then run the program again.
- The program will execute all line of codes and stop at code line =8

```
(gdb) list
1       // cprogram1.c //
2
3       #include <stdio.h>
4       #include <stdlib.h>
5
6       int add (int x, int y)
7       {
8            int result = 0;
9            result = x + y;
10           return result;
(gdb) break 8
Breakpoint 1 at 0x4005ed: file cprogram1.c, line 8.
(gdb)
```

Run the program After Adding the Breakpoints



Run the program for specific code (Breakpoints)

- For example, to run the program from the beginning until certain function in the code

  (gdb) break FunctionName
- Then run the program again.
- The program will execute all line of codes and stop at the beginning of the FunctionName

  - For example, (gdb) break add

```
(gdb) list
1       // cprogram1.c //
2
3       #include <stdio.h>
4       #include <stdlib.h>
5
6       int add (int x, int y)
7       {
8            int result = 0;
9            result = x + y;
10           return result;
(gdb) break 8
Breakpoint 1 at 0x4005ed: file cprogram1.c, line 8.
(gdb)
```

Manage Breakpoints

To display all existing breakpoints you can use the info break command

```
(gdb) info break
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x001f7c in main at main.c:4
2   breakpoint     keep y   0x001f96 in main at main.c:12
3   breakpoint     keep y   0x001fa9 in main at main.c:17
```

To delete a breakpoint you can use the delete command

For example, to delete breakpoint#2, use delete 2

GDB Commands: step (s)

```
(gdb) list
1       // cprogram1.c //
2
3       #include <stdio.h>
4       #include <stdlib.h>
5
6       int add (int x, int y)
7       {
8               int result = 0;
9               result = x + y;
10              return result;
(gdb) break 8
Breakpoint 1 at 0x4005ed: file cprogram1.c, line 8.
(gdb) run 10 20
Starting program: /cprogram1 10 20
Use the breakpoint
Use the breakpoint

Breakpoint 1, add (x=10, y=20) at cprogram1.c:8
8               int result = 0;
(gdb) print x
$1 = 10
(gdb) print y
$2 = 20
(gdb)
```

Display value of argument 1 (x)

```
(gdb) info break
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x004005ed in add at cprogram1.c:8
        breakpoint already hit 1 time
(gdb) s
9               result = x + y;
(gdb) print result
$3 = 0
(gdb)
```

Useful gdb Commands

run (or ‘r’): starts the program

- with potential command-line arguments
- the program will run all the way through

print (or ‘p’): to print variable values

list (or ‘l’): shows 10 lines of code around “where we are”

break (or ‘b’): sets a breakpoint

- e.g., “break add”, break at the enter of function “add”
- e.g., “break 9, break at line#9

step (or ‘s’): runs the program step-by-step (after each breakpoint)

next (or ‘n’): like step, but skips over functions

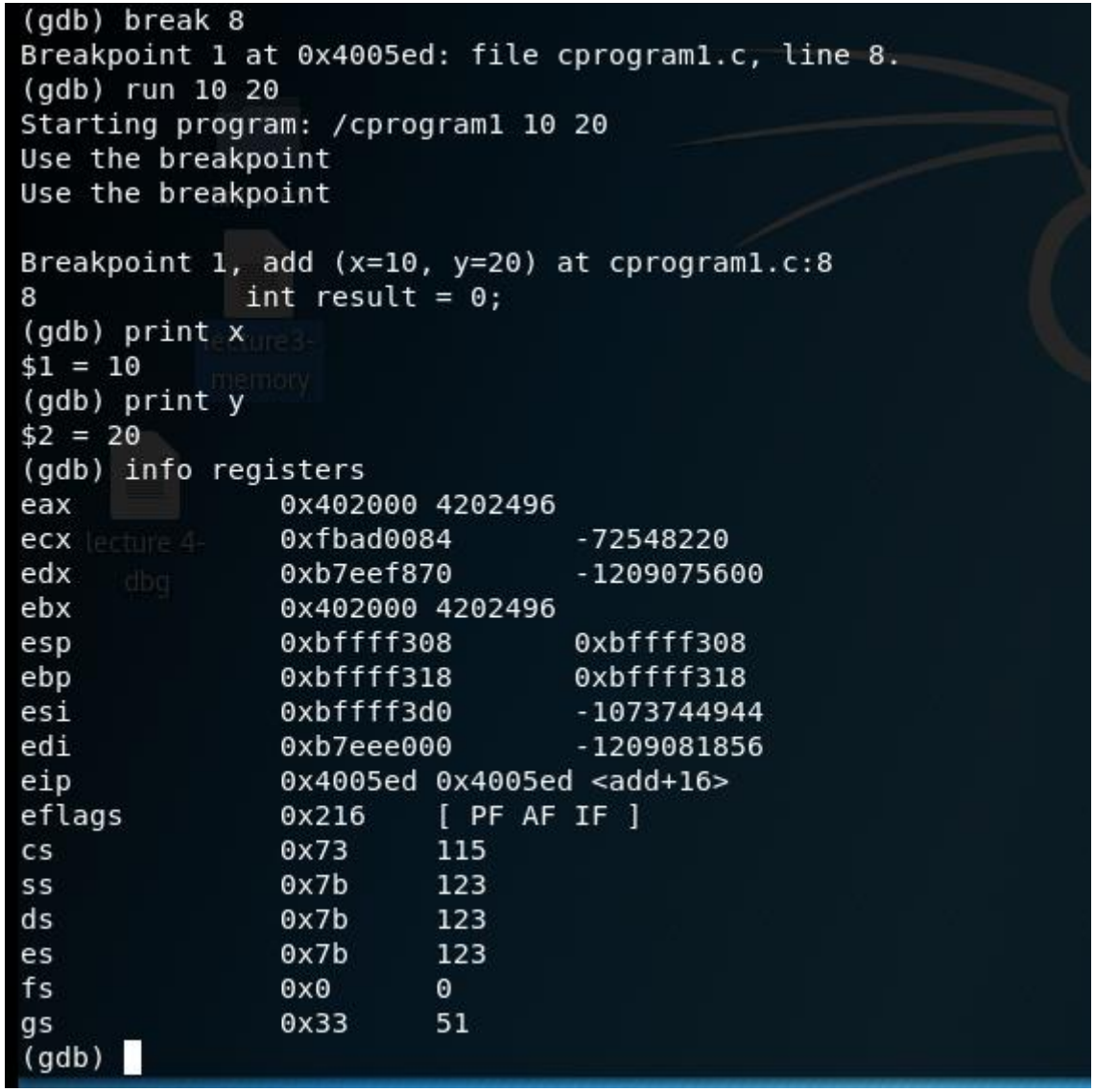
continue (or ‘c’): continues until next breakpoint

(control + L): clear debugger screen

x: examine memory location

quit (or ‘q’): quits the program/debugger

Display The values of the Registers: info registers



```
(gdb) break 8
Breakpoint 1 at 0x4005ed: file cprogram1.c, line 8.
(gdb) run 10 20
Starting program: /cprogram1 10 20
Use the breakpoint
Use the breakpoint

Breakpoint 1, add (x=10, y=20) at cprogram1.c:8
8           int result = 0;
(gdb) print x
$1 = 10
(gdb) print y
$2 = 20
(gdb) info registers
eax            0x402000 4202496
ecx            0xfbad0084       -72548220
edx            0xb7eef870       -1209075600
ebx            0x402000 4202496
esp            0xbffff308       0xbffff308
ebp            0xbffff318       0xbffff318
esi            0xbffff3d0       -1073744944
edi            0xb7eee000       -1209081856
eip            0x4005ed 0x4005ed <add+16>
eflags         0x216    [ PF AF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb)
```

Examine Memory Content of memory Location

- Examine command (x) is used to examine memory location

x/FMT Memory Address

FMT: Format to explain to the debugger how to display the content of memory

- count (how many units to display)
- Display the content of the memory format
  - o (octal)
  - x (hex)
  - d (decimal)
  - t (binary)
- Size of the unit
  - b (byte)
  - w (word, 4 bytes)
  - h (1/2 word, 2 bytes)
  - g (2 words, 8 bytes)

Display content of 10 bytes at address location 0xbffff308, display the content as hex

(dgb) x/10xb 0xbffff308

Display content of 10 words at address location 0xbffff308, display the content as hex

(dgb) x/10xw 0xbffff308

Get More information about GDB Commands

- Use “help” “command”
- Example: To get information about examining memory command (x)

  (gdb) help x

```
(gdb) help x
Examine memory: x/FMT ADDRESS.
ADDRESS is an expression for the memory address to examine.
FMT is a repeat count followed by a format letter and a size letter.
Format letters are o(octal), x(hex), d(decimal), u(unsigned decimal),
  t(binary), f(float), a(address), i(instruction), c(char), s(string)
  and z(hex, zero padded on the left).
Size letters are b(byte), h(halfword), w(word), g(giant, 8 bytes).
The specified number of objects of the specified size are printed
according to the format.  If a negative number is specified, memory is
examined backward from the address.

Defaults for format and size letters are those previously used.
Default count is 1.  Default address is following last thing printed
with this command or "print".
(gdb)
```

Examine Memory Content of memory Location

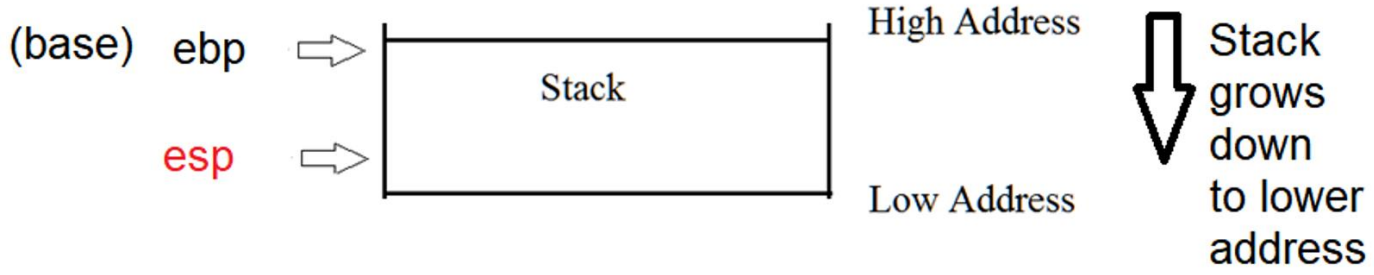
- Display content of 10 bytes at address location 0xbffff308, display the content as hex
  (dgb) x/10xb 0xbffff308

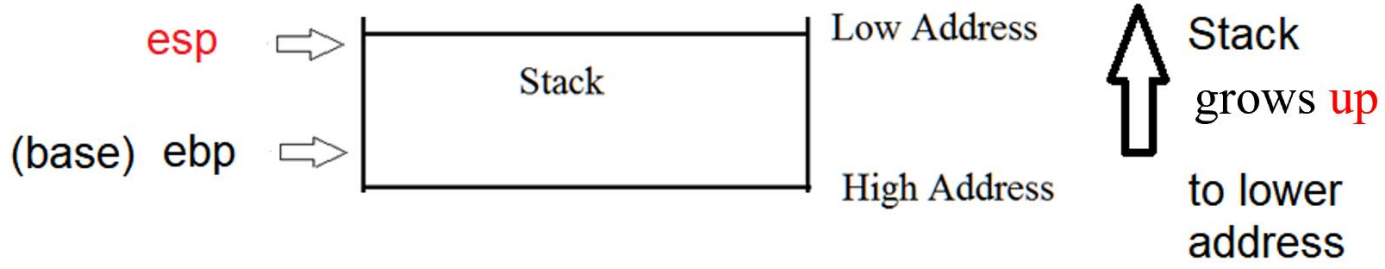
- Display content of 10 words at address location 0xbffff308, display the content as hex
  (dgb) x/10xw 0xbffff308

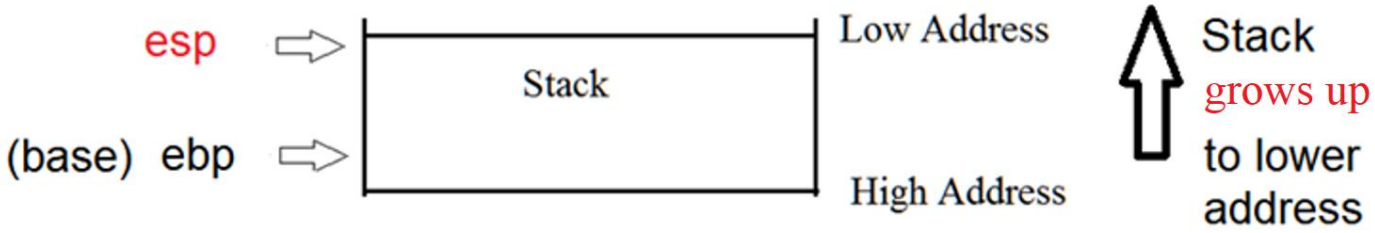
```
(gdb) x/10xb 0xbffff308
0xbffff308:     0x8b    0xa8    0xd9    0xb7    0x00    0x20    0x40    0x00
0xbffff310:     0xd0    0xf3
(gdb) x/10xw 0xbffff308
0xbffff308:     0xb7d9a88b      0x00402000      0xbffff3d0      0xb7eee000
0xbffff318:     0xbffff3b8      0x00400684      0x0000000a      0x00000014
0xbffff328:     0xb7f3e920      0x0040061c
(gdb)
```

Visualizing the stack

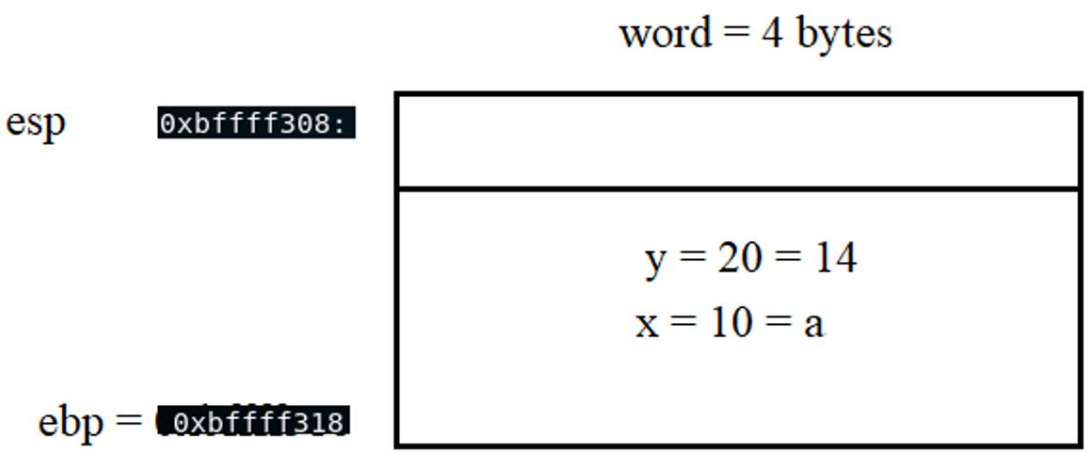
- Some books show stack grows down towards low addresses
- Some books show stack grows up towards low addresses
- Debuggers show stack grows up towards low addresses








```
(gdb) x/10xw 0xbffff308
0xbffff308:     0xb7d9a88b      0x00402000      0xbffff3d0      0xb7eee000
0xbffff318:     0xbffff3b8      0x00400684      0x0000000a      0x00000014
0xbffff328:     0xb7f3e920      0x0040061c
(gdb)
```



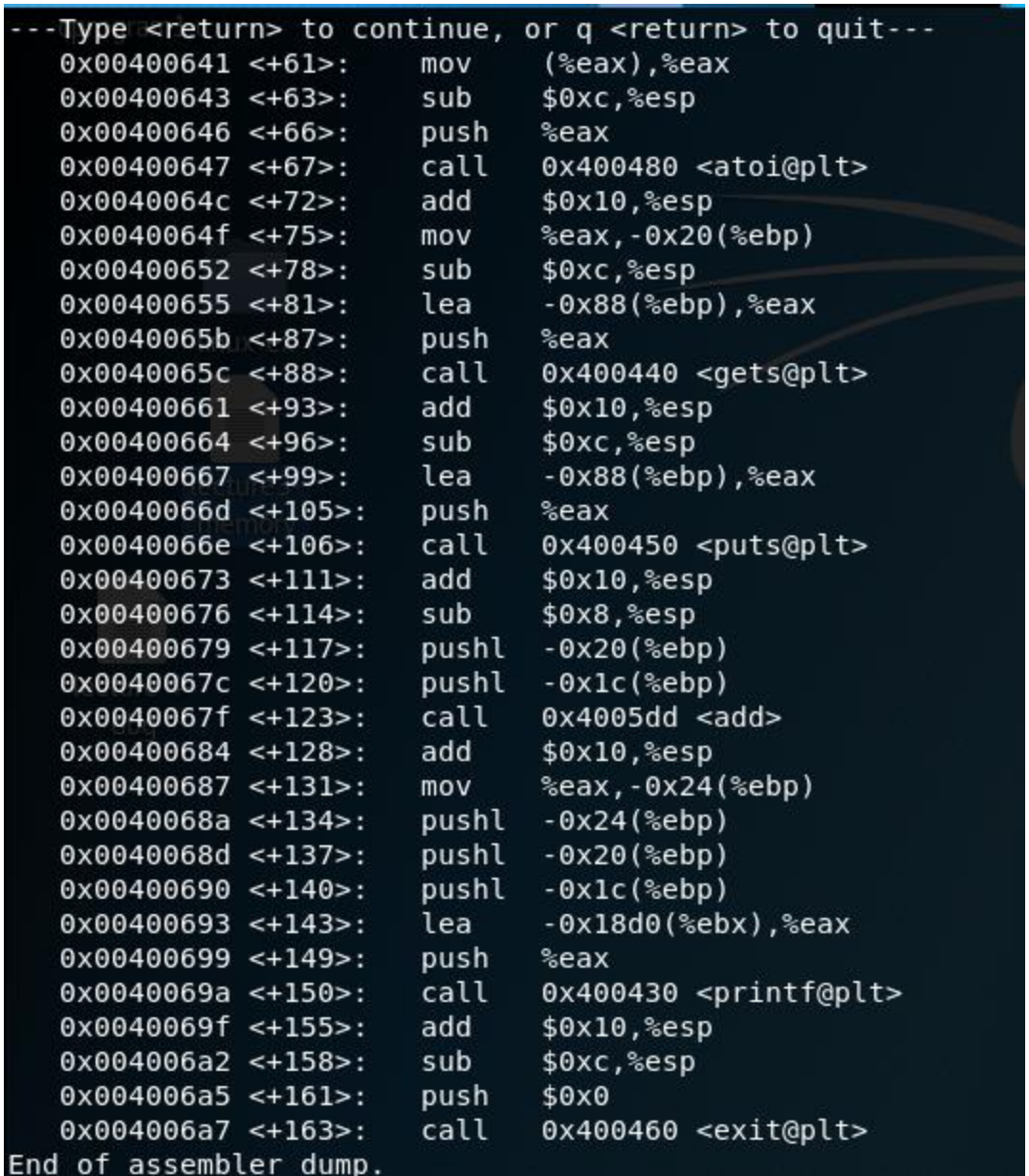

```
eax            0x402000 4202496
ecx            0xfbad0084       -72548220
edx            0xb7eef870       -1209075600
ebx            0x402000 4202496
esp            0xbffff308       0xbffff308
ebp            0xbffff318       0xbffff318
esi            0xbffff3d0       -1073744944
edi            0xb7eee000       -1209081856
eip            0x4005ed 0x4005ed <add+16>
eflags         0x216    [ PF AF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
```

Disassemble The Program: disassemble

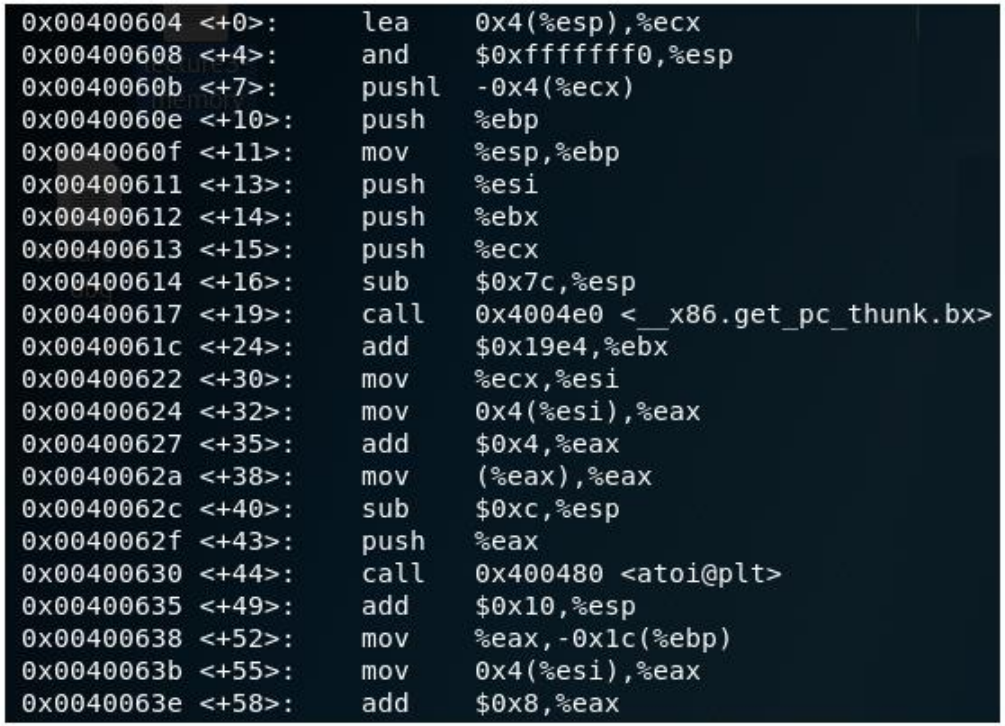
- To get the assembly language of the executable program, use
  (gdb) disassemble function name

```
(gdb) disassemble main
Dump of assembler code for function main:
   0x00400604 <+0>:     lea    0x4(%esp),%ecx
   0x00400608 <+4>:     and    $0xfffffff0,%esp
   0x0040060b <+7>:     pushl  -0x4(%ecx)
   0x0040060e <+10>:    push   %ebp
   0x0040060f <+11>:    mov    %esp,%ebp
   0x00400611 <+13>:    push   %esi
   0x00400612 <+14>:    push   %ebx
   0x00400613 <+15>:    push   %ecx
   0x00400614 <+16>:    sub    $0x7c,%esp
   0x00400617 <+19>:    call   0x4004e0 <__x86.get_pc_thunk.bx>
   0x0040061c <+24>:    add    $0x19e4,%ebx
   0x00400622 <+30>:    mov    %ecx,%esi
   0x00400624 <+32>:    mov    0x4(%esi),%eax
   0x00400627 <+35>:    add    $0x4,%eax
   0x0040062a <+38>:    mov    (%eax),%eax
   0x0040062c <+40>:    sub    $0xc,%esp
   0x0040062f <+43>:    push   %eax
   0x00400630 <+44>:    call   0x400480 <atoi@plt>
   0x00400635 <+49>:    add    $0x10,%esp
   0x00400638 <+52>:    mov    %eax,-0x1c(%ebp)
   0x0040063b <+55>:    mov    0x4(%esi),%eax
   0x0040063e <+58>:    add    $0x8,%eax
---Type <return> to continue, or q <return> to quit---
```

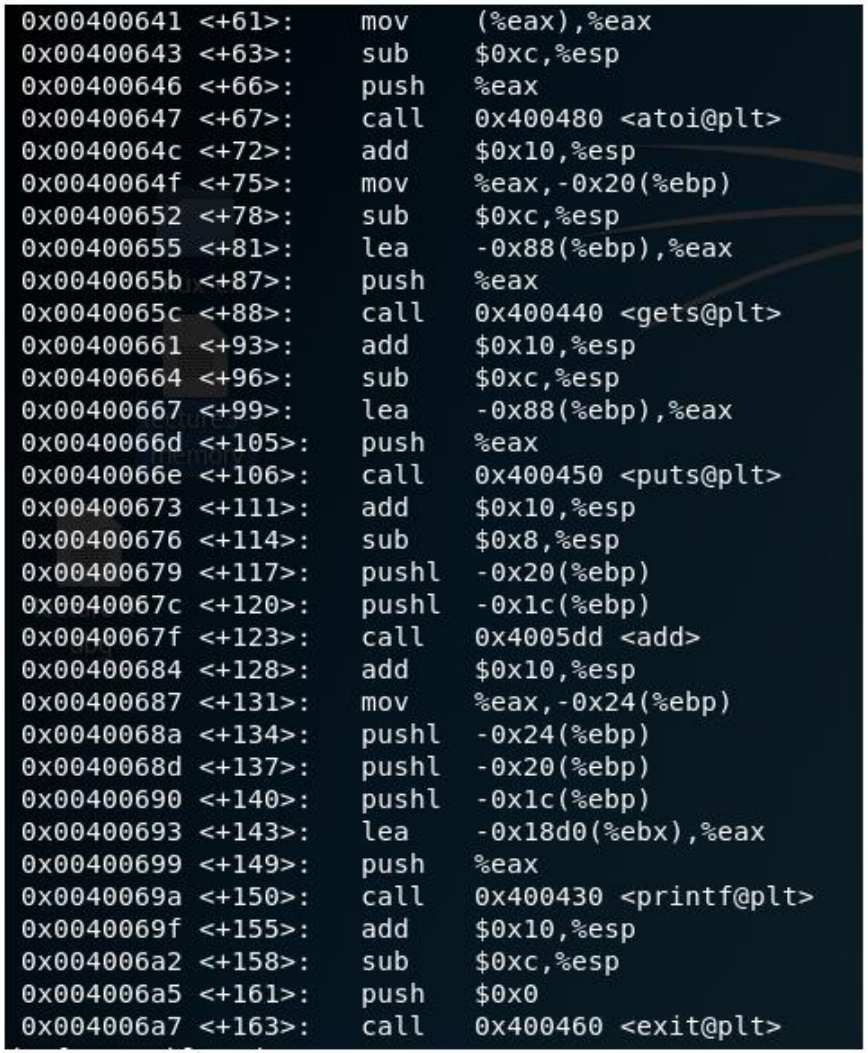
Disassemble The Program: disassemble



```
---Type <return> to continue, or q <return> to quit---
   0x00400641 <+61>:    mov    (%eax),%eax
   0x00400643 <+63>:    sub    $0xc,%esp
   0x00400646 <+66>:    push   %eax
   0x00400647 <+67>:    call   0x400480 <atoi@plt>
   0x0040064c <+72>:    add    $0x10,%esp
   0x0040064f <+75>:    mov    %eax,-0x20(%ebp)
   0x00400652 <+78>:    sub    $0xc,%esp
   0x00400655 <+81>:    lea    -0x88(%ebp),%eax
   0x0040065b <+87>:    push   %eax
   0x0040065c <+88>:    call   0x400440 <gets@plt>
   0x00400661 <+93>:    add    $0x10,%esp
   0x00400664 <+96>:    sub    $0xc,%esp
   0x00400667 <+99>:    lea    -0x88(%ebp),%eax
   0x0040066d <+105>:   push   %eax
   0x0040066e <+106>:   call   0x400450 <puts@plt>
   0x00400673 <+111>:   add    $0x10,%esp
   0x00400676 <+114>:   sub    $0x8,%esp
   0x00400679 <+117>:   pushl  -0x20(%ebp)
   0x0040067c <+120>:   pushl  -0x1c(%ebp)
   0x0040067f <+123>:   call   0x4005dd <add>
   0x00400684 <+128>:   add    $0x10,%esp
   0x00400687 <+131>:   mov    %eax,-0x24(%ebp)
   0x0040068a <+134>:   pushl  -0x24(%ebp)
   0x0040068d <+137>:   pushl  -0x20(%ebp)
   0x00400690 <+140>:   pushl  -0x1c(%ebp)
   0x00400693 <+143>:   lea    -0x18d0(%ebx),%eax
   0x00400699 <+149>:   push   %eax
   0x0040069a <+150>:   call   0x400430 <printf@plt>
   0x0040069f <+155>:   add    $0x10,%esp
   0x004006a2 <+158>:   sub    $0xc,%esp
   0x004006a5 <+161>:   push   $0x0
   0x004006a7 <+163>:   call   0x400460 <exit@plt>
End of assembler dump.
```



```
0x00400604 <+0>:     lea    0x4(%esp),%ecx
0x00400608 <+4>:     and    $0xfffffff0,%esp
0x0040060b <+7>:     pushl  -0x4(%ecx)
0x0040060e <+10>:    push   %ebp
0x0040060f <+11>:    mov    %esp,%ebp
0x00400611 <+13>:    push   %esi
0x00400612 <+14>:    push   %ebx
0x00400613 <+15>:    push   %ecx
0x00400614 <+16>:    sub    $0x7c,%esp
0x00400617 <+19>:    call   0x4004e0 <__x86.get_pc_thunk.bx>
0x0040061c <+24>:    add    $0x19e4,%ebx
0x00400622 <+30>:    mov    %ecx,%esi
0x00400624 <+32>:    mov    0x4(%esi),%eax
0x00400627 <+35>:    add    $0x4,%eax
0x0040062a <+38>:    mov    (%eax),%eax
0x0040062c <+40>:    sub    $0xc,%esp
0x0040062f <+43>:    push   %eax
0x00400630 <+44>:    call   0x400480 <atoi@plt>
0x00400635 <+49>:    add    $0x10,%esp
0x00400638 <+52>:    mov    %eax,-0x1c(%ebp)
0x0040063b <+55>:    mov    0x4(%esi),%eax
0x0040063e <+58>:    add    $0x8,%eax
```



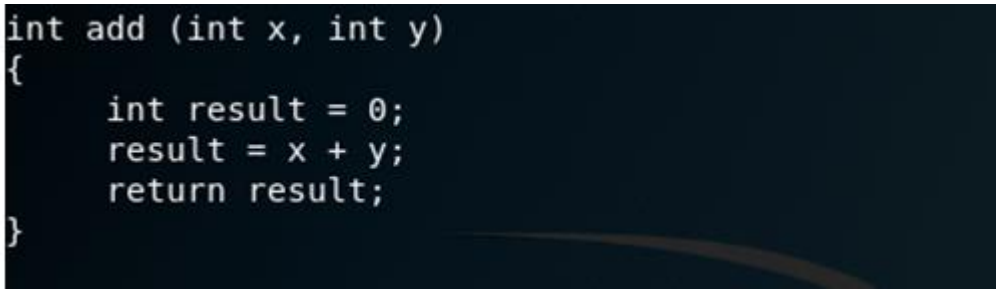
```
0x00400641 <+61>:    mov    (%eax),%eax
0x00400643 <+63>:    sub    $0xc,%esp
0x00400646 <+66>:    push   %eax
0x00400647 <+67>:    call   0x400480 <atoi@plt>
0x0040064c <+72>:    add    $0x10,%esp
0x0040064f <+75>:    mov    %eax,-0x20(%ebp)
0x00400652 <+78>:    sub    $0xc,%esp
0x00400655 <+81>:    lea    -0x88(%ebp),%eax
0x0040065b <+87>:    push   %eax
0x0040065c <+88>:    call   0x400440 <gets@plt>
0x00400661 <+93>:    add    $0x10,%esp
0x00400664 <+96>:    sub    $0xc,%esp
0x00400667 <+99>:    lea    -0x88(%ebp),%eax
0x0040066d <+105>:   push   %eax
0x0040066e <+106>:   call   0x400450 <puts@plt>
0x00400673 <+111>:   add    $0x10,%esp
0x00400676 <+114>:   sub    $0x8,%esp
0x00400679 <+117>:   pushl  -0x20(%ebp)
0x0040067c <+120>:   pushl  -0x1c(%ebp)
0x0040067f <+123>:   call   0x4005dd <add>
0x00400684 <+128>:   add    $0x10,%esp
0x00400687 <+131>:   mov    %eax,-0x24(%ebp)
0x0040068a <+134>:   pushl  -0x24(%ebp)
0x0040068d <+137>:   pushl  -0x20(%ebp)
0x00400690 <+140>:   pushl  -0x1c(%ebp)
0x00400693 <+143>:   lea    -0x18d0(%ebx),%eax
0x00400699 <+149>:   push   %eax
0x0040069a <+150>:   call   0x400430 <printf@plt>
0x0040069f <+155>:   add    $0x10,%esp
0x004006a2 <+158>:   sub    $0xc,%esp
0x004006a5 <+161>:   push   $0x0
0x004006a7 <+163>:   call   0x400460 <exit@plt>
```

```
0x00400604 <+0>:     lea    0x4(%esp),%ecx
0x00400608 <+4>:     and    $0xfffffff0,%esp
0x0040060b <+7>:     pushl  -0x4(%ecx)
0x0040060e <+10>:    push   %ebp
0x0040060f <+11>:    mov    %esp,%ebp
0x00400611 <+13>:    push   %esi
0x00400612 <+14>:    push   %ebx
0x00400613 <+15>:    push   %ecx
0x00400614 <+16>:    sub    $0x7c,%esp
0x00400617 <+19>:    call   0x4004e0 <__x86.get_pc_thunk.bx>
0x0040061c <+24>:    add    $0x19e4,%ebx
0x00400622 <+30>:    mov    %ecx,%esi
0x00400624 <+32>:    mov    0x4(%esi),%eax
0x00400627 <+35>:    add    $0x4,%eax
0x0040062a <+38>:    mov    (%eax),%eax
0x0040062c <+40>:    sub    $0xc,%esp
0x0040062f <+43>:    push   %eax
0x00400630 <+44>:    call   0x400480 <atoi@plt>
0x00400635 <+49>:    add    $0x10,%esp
0x00400638 <+52>:    mov    %eax,-0x1c(%ebp)
0x0040063b <+55>:    mov    0x4(%esi),%eax
0x0040063e <+58>:    add    $0x8,%eax
```

```
0x00400641 <+61>:    mov    (%eax),%eax
0x00400643 <+63>:    sub    $0xc,%esp
0x00400646 <+66>:    push   %eax
0x00400647 <+67>:    call   0x400480 <atoi@plt>
0x0040064c <+72>:    add    $0x10,%esp
0x0040064f <+75>:    mov    %eax,-0x20(%ebp)
0x00400652 <+78>:    sub    $0xc,%esp
0x00400655 <+81>:    lea    -0x88(%ebp),%eax
0x0040065b <+87>:    push   %eax
0x0040065c <+88>:    call   0x400440 <gets@plt>
0x00400661 <+93>:    add    $0x10,%esp
0x00400664 <+96>:    sub    $0xc,%esp
0x00400667 <+99>:    lea    -0x88(%ebp),%eax
0x0040066d <+105>:   push   %eax
0x0040066e <+106>:   call   0x400450 <puts@plt>
0x00400673 <+111>:   add    $0x10,%esp
0x00400676 <+114>:   sub    $0x8,%esp
0x00400679 <+117>:   pushl  -0x20(%ebp)
0x0040067c <+120>:   pushl  -0x1c(%ebp)
0x0040067f <+123>:   call   0x4005dd <add>
0x00400684 <+128>:   add    $0x10,%esp
0x00400687 <+131>:   mov    %eax,-0x24(%ebp)
0x0040068a <+134>:   pushl  -0x24(%ebp)
0x0040068d <+137>:   pushl  -0x20(%ebp)
0x00400690 <+140>:   pushl  -0x1c(%ebp)
0x00400693 <+143>:   lea    -0x18d0(%ebx),%eax
0x00400699 <+149>:   push   %eax
0x0040069a <+150>:   call   0x400430 <printf@plt>
0x0040069f <+155>:   add    $0x10,%esp
0x004006a2 <+158>:   sub    $0xc,%esp
0x004006a5 <+161>:   push   $0x0
```

```
// cprogram1.c //

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

int add (int x, int y)
{
     int result = 0;
     result = x + y;
     return result;
}

main (int argc, char **argv)
{
    int a = atoi(argv[1]);
    int b = atoi(argv[2]);
    int c;
    char buffer [100];

    gets(buffer);
    puts(buffer);

    c = add (a, b);

    printf("Sum of %d and %d = %d\n", a, b, c);
    exit (0);
}
```

Disassemble The Program: disassemble



```
int add (int x, int y)
{
     int result = 0;
     result = x + y;
     return result;
}
```

```
(gdb) disassemble add
Dump of assembler code for function add:
   0x004005dd <+0>:     push   %ebp
   0x004005de <+1>:     mov    %esp,%ebp
   0x004005e0 <+3>:     sub    $0x10,%esp
   0x004005e3 <+6>:     call   0x4006ac <__x86.get_pc_thunk.ax>
   0x004005e8 <+11>:    add    $0x1a18,%eax
   0x004005ed <+16>:    movl   $0x0,-0x4(%ebp)
   0x004005f4 <+23>:    mov    0x8(%ebp),%edx
   0x004005f7 <+26>:    mov    0xc(%ebp),%eax
   0x004005fa <+29>:    add    %edx,%eax
   0x004005fc <+31>:    mov    %eax,-0x4(%ebp)
   0x004005ff <+34>:    mov    -0x4(%ebp),%eax
   0x00400602 <+37>:    leave
   0x00400603 <+38>:    ret
End of assembler dump.
(gdb)
```

```
1     // cprogram1.c //
2
3       #include <stdio.h>
4       #include <stdlib.h>
5
6       int add (int x, int y)
7       {
8            int result = 0;
9            result = x + y;
10           return result;
(gdb) break 8
Breakpoint 1 at 0x4005ed: file cprogram1.c, line 8.
(gdb) run 10 20
Starting program: /cprogram1 10 20
Use the breakpoint
Use the breakpoint
```

- Why does eip =0x4005ed?

```
(gdb) disassemble add
Dump of assembler code for function add:
   0x004005dd <+0>:     push   %ebp
   0x004005de <+1>:     mov    %esp,%ebp
   0x004005e0 <+3>:     sub    $0x10,%esp
   0x004005e3 <+6>:     call   0x4006ac <__x86.get_pc_thunk.ax>
   0x004005e8 <+11>:    add    $0x1a18,%eax
   0x004005ed <+16>:    movl   $0x0,-0x4(%ebp)
   0x004005f4 <+23>:    mov    0x8(%ebp),%edx
   0x004005f7 <+26>:    mov    0xc(%ebp),%eax
   0x004005fa <+29>:    add    %edx,%eax
   0x004005fc <+31>:    mov    %eax,-0x4(%ebp)
   0x004005ff <+34>:    mov    -0x4(%ebp),%eax
   0x00400602 <+37>:    leave
   0x00400603 <+38>:    ret
End of assembler dump.
(gdb)
```

Putting It All Together

You should know:

- **What is a debugger**
- **Static Debugging**
- **Dynamic Debugging**
- **Symbol table**
- **Running gdb debugger**
- **Mange breakpoints**
- **Examine memory contents**
- **Display register values**
- **Disassemble c program**

Questions?

Week 4 Assignments

- **Learning Materials**

  1- Week 4 Presentation

Assignment

1- Complete “GDB commands” assignment by coming Sunday 11:59 PM.