

# Assignment 3

## Group Members

Shaurya Raj Verma	510519105
Shashank Srivastav	510519106
Rishita De	510519107
Sayak Rana	510519108
Abhiroop Mukherjee	510519109
Nikhil Joshi	510519110

## What is GNU debugger (GDB)?

GNU Debugger, also called "gdb" is a command line debugger primarily used in UNIX systems to debug programs of many languages like C, C++, etc.

It was written by Richard Stallman in 1986 as a part of this GNU System, and since then it has been widely used as a go-to debugger till today.

It provides a lot of features to help debug a program, which includes monitoring and modifying the values of internal variables at run time, calling functions independent of the actual program behavior, and much more.

## How to use GNU Debugger?

### Part 0: Installing debugger

"sudo apt install gdb"

### Part 1: Running the program using debugger

```
// Hello.c
#include <stdio.h>

int main(){
    int value = 3;
    for (int i = 0; i < 10; i++){
        if (i == value){
            printf("Target Reached, breaking...\n");
        }
    }
}
```

```
        break;
    }
    printf("Hello World\n");
}
}
```

Generally, we do “gcc hello.c -o hello”

But to generate GDB accessible program, we will have to use “-g” flag

- “-g” flag is used to signal gcc to also generate a **symbol table** for gdb to read

So “gcc hello.c -o hello -g”

Now we just have to do “gdb hello” to start the debugger

Now we are inside gdb, to just run the program, we can do the following

1. “r” to run the program
2. “r 1 2” to pass command-line arguments to the program
3. “r < file1” to feed a file for writing the outputs

## Part 2: Loading Symbol Table

1. Symbol table stores names of variables, functions and types defined within our program

## Part 3: Setting a Break Point

1. There are many ways to setup Break Points in the GNU GDB
2. Two majorly used ones include
  - a. Setting a Break Point at the start of a function():

```
break /*{function name}*/
```

- b. Setting a Break Point at a specific line:

```
break /*{line number}*/
```

```
(gdb) break 7
Breakpoint 1 at 0x804842e: file debug.c, line 7.
(gdb) run
Starting program: /home/user/ctest/debug

Breakpoint 1, main () at debug.c:7
7          int b = 1;
(gdb) info locals
a = 0
b = -1207963648
c = 134513787
d = -1208221696
(gdb)
```

This command will add a break point at the line number we specify. As soon as the execution reaches our line number, it will halt the execution process

3. After a Break Point is reached, we can view the state of every variable, function and the call stack up until that point.
4. The command “info locals” will show us the state of the local variables at our break point
5. The command “info break” will show us all the break points currently held within our program

```
(gdb) info break
Num      Type           Disp Enb Address      What
1        breakpoint      keep y   0x0804842e in main at debug.c:7
          breakpoint already hit 1 time
2        breakpoint      keep y   0x08048446 in main at debug.c:11
          breakpoint already hit 1 time
```

6. The command “continue” will resume the execution which was previously halted after reaching a break point
7. The command “delete /\*{breakpoint number}\*/ will delete a breakpoint

## Part 4: Listing variables

- The **info locals** command displays the local variable values in the current frame. You can select frames using the **frame**, **up** and **down**
- Note that the **info locals** command does not display the information about the function arguments.

- Use the **info args** command to list function arguments.

We will run the program, set a breakpoint in **func()** and use the **info locals** command to display the local variables in **main()**:

```
#include <stdio.h>

void func(int arg)
{
    printf("func(%d)\n", arg);
}

int main(int argc, char *argv[])
{
    int localVar1 = 1, localVar2 = 2;
    func(localVar1 + localVar2);
    return 0;
}
```

```
(gdb) b func
Breakpoint 1 at 0x80483ea: file test.cpp, line 5.
(gdb) r
Starting program: /home/testuser/test

Breakpoint 1, func (arg=3) at test.cpp:5
5 printf("func(%d) arg);
(gdb) backtrace
#0 func (arg=3) at test.cpp:5
#1 0x0804842a in main (argc=1, argv=0xbffff784) at test.cpp:11
(gdb) info locals
No locals.
(gdb) up
#1 0x0804842a in main (argc=1, argv=0xbffff784) at test.cpp:11
11 func(localVar1 + localVar2);
(gdb) info locals
localVar1 = 1
localVar2 = 2
(gdb) down
#0 func (arg=3) at test.cpp:5
5 printf("func(%d) arg);
(gdb) info args
arg = 3
(gdb)
```

## Part 5: Printing content of an array

## • Syntax:

**set print array on**

**set print array off**

**show print array**

## • Modes:

**on**

GDB will display the values of arrays in a simple one-line format (e.g. \$1 = {1, 2, 3}).

**off**

GDB will display the values of arrays using longer multi-line format.

- **Default mode:**

The default value for the **print array** setting is 'off'.

## • Remarks:

The **set print array** command can be used together with the **set print array-indexes** command to further customize the output of the array contents.

## Examples:

Below is a log of sample GDB session illustrating how **set print array** command affects the output of the **print** command:

```
(gdb) start
Temporary breakpoint 1 at 0x80483f3: file test.cpp, line 5.
Starting program: /home/bazis/test

Temporary breakpoint 1, main (argc=1, argv=0xbffff064) at test.cpp:5
5 int testArray[] = {1, 2, 3};
(gdb) next
6 return 0;
(gdb) print testArray
$1 = {1, 2, 3}
(gdb) show print array
Prettyprinting of arrays is off.
(gdb) set print array on
(gdb) print testArray
$2 = {1,
2,
3}
```

Example:

```
int main(){
    int *a;
    int b[3] = {1,2,3};
    a = b;

    int *c[3] = {a, b, 0};
    int **d = c;
    return 0;
}
```

While debugging the above code if we do:

```
(gdb) print b
$4 = {1, 2, 3}

(gdb) print a
$5 = (int *) 0x7fffffffefe0f0
```

Both works but in order to print a as an array:

```
(gdb) print (int []) *a
$7 = {1}
(gdb) print (int [3]) *a
$8 = {1, 2, 3}
```

and when we specify the size it gets better.

## Part 6: Printing Function Arguments

Inspecting a core dump and need to print function arguments values when only their types are known (no argument name symbols):

```
(gdb) frame 7
#7  0x00007f201a269e82 in f1(std::basic_string<char, std::char_traits<char>, std::allocator<char> >, unsigned
(gdb) info args
No symbol table info available.
(gdb) info f
Stack level 7, frame at 0x7f200ebf9e50:
 rip = 0x7f201a269e82
    in f1(std::basic_string<char, std::char_traits<char>, std::allocator<char> >, unsigned
called by frame at 0x7f200ebfa1c0, caller of frame at 0x7f200ebf9e00
Arglist at 0x7f200ebf9df8, args:
Locals at 0x7f200ebf9df8, Previous frame's sp is 0x7f200ebf9e50
Saved registers:
  rbx at 0x7f200ebf9e28, rbp at 0x7f200ebf9e30, r12 at 0x7f200ebf9e38, r13 at 0x7f200ebf9e40
```

- Specifically, we need to know what's in the first argument (std::string) and in the last one (std::string\*). The arglist and the *locals* in this frame both point to the same address.
- We have to look up the ABI for your platform and then use that to understand the argument-passing convention. This isn't always easy but it can be done. This will tell us how to find the arguments, whether they are in registers or memory; and then we can use casts to the appropriate types to print them.



- The casting may also be difficult if you don't have debuginfo. Though there is a trick: compile a dummy file with -g that has the types you need and then symbol-file it into gdb to get access to the types. This of course has caveats; we have to use the correct compiler and library versions, correct compiler target and ABI-changing flags, etc.
- It's really much, much, much better to plan ahead and always have debug info somewhere.

## Part 7: Next, Continue, Set commands

### Next - Is used to step through code

- If the instruction is a single line of code it displays it
- If the line is a function call, it executes the entire function call in one step

There is a variation of the call `next` which is `nexti (ni)`. According to the help pages of gdb below is its functionality as compared to `next (n)`

#### *next*

Step program, proceeding through subroutine calls.

Usage: `next [N]`

Unlike "step", if the current source line calls a subroutine, this command does not enter the subroutine, but instead steps over the call, in effect treating it as a single source line.

#### *nexti*

Step one instruction, but proceed through subroutine calls.

Usage: `nexti [N]`

Argument N means step N times (or till program stops for another reason).

Below is an example demonstrating the use of `next`

```
#include <iostream>
using namespace std;

void printString(string letters) {
    cout << "Printing the letters of string\n";
    for(auto letter : letters)
        cout << letter << ", ";
    cout << "\nEnd of string\n";
}
```



```

}

int main() {
    // a string variable
    string s = "Software engineering lab";

    // a function to print the string variable
    printString(s);
    return 0;
}

```

The output of this code is

```

riri@riri-VirtualBox:asgn 2$ ./next

Printing the letters of string
S, o, f, t, w, a, r, e, , e, n, g, i, n, e, e, r, i, n, g, , l, a, b,
End of string
riri@riri-VirtualBox:asgn 2$ 

```

For demonstration purposes, a break-point has been added at main and the program is then run.

```

riri@riri-VirtualBox:asgn 2$ gdb ./next -q
Reading symbols from ./next...
(gdb) break main
Breakpoint 1 at 0x13a5: file next.cpp, line 12.
(gdb) r
Starting program: /home/riri/Desktop/asgn 2/next

Breakpoint 1, main () at next.cpp:12
12      int main() {
(gdb) n
14          string s = "Software engineering lab";
(gdb) n
17          printString(s);
(gdb) n
Printing the letters of string
S, o, f, t, w, a, r, e, , e, n, g, i, n, e, e, r, i, n, g, , l, a, b,
End of string
18          return 0;
(gdb) 

```

After adding a break-point at main, we run and step through the code. It can be seen that when the code happens to be a single line, **next** simply displays the code. However when it hits the function call **printString(s)**, it executes the entire function

rather than stepping through it. The difference will be more visible after the demonstration of step 8.

**Continue - is used to resume normal execution of a program until it ends, crashes or a break-point is encountered.**

```
#include <iostream>
#include <array>
using namespace std;

void printFirstFive(const array<int, 10>& numbers) {
    for(int i = 0 ; i < 5 ; i++) {
        cout << "numbers[" << i << "] : " << numbers[i] << '\n';
    }
}

void printRest(const array<int, 10>& numbers) {
    for(int i = 5 ; i < 10 ; i++) {
        cout << "numbers[" << i << "] : " << numbers[i] << '\n';
    }
}

int main() {

    array<int, 10> arr = {3, 4, 6, 3, 2, 7, 2, 9, 6, 5};
    printFirstFive(arr);

    cout << "We have printed the first 5 numbers in the array\n";

    printRest(arr);
    cout << "We have printed the rest of the numbers of the array\n";
    return 0;
}
```

Using the program above, we see that after stepping through the code line by line, we can choose to run it until one of the conditions to stop normal execution are met.

Break-points have been set at functions *main* and *printRest*

```

riri@riri-VirtualBox:asgn 2$ gdb ./continue -q
Reading symbols from ./continue...
(gdb) break main
Breakpoint 1 at 0x1307: file continue.cpp, line 17.
(gdb) break printRest
Breakpoint 2 at 0x1278: file continue.cpp, line 11.
(gdb) r
Starting program: /home/riri/Desktop/asgn 2/continue

Breakpoint 1, main () at continue.cpp:17
17   int main() {
(gdb) s
19       array<int, 10> arr = {3, 4, 6, 3, 2, 7, 2, 9, 6, 5};
(gdb) s
20       printFirstFive(arr);
(gdb) s
printFirstFive (numbers=...) at continue.cpp:5
5   void printFirstFive(const array<int, 10>& numbers) {
(gdb) s
6       for(int i = 0 ; i < 5 ; i++) {
(gdb) s
7           cout << "numbers[" << i << " ] : " << numbers[i] << '\n';
(gdb) s
std::array<int, 10ul>::operator[] (this=0x7fffffffdf20, __n=140737488347216)
    at /usr/include/c++/9/array:189
189   operator[](size_type __n) const noexcept
(gdb) s
190   { return _AT_Type::_S_ref(_M_elems, __n); }
(gdb) s
std::__array_traits<int, 10ul>::_S_ref (__t=..., __n=140737488347216)
    at /usr/include/c++/9/array:55
55   _S_ref(const _Type& __t, std::size_t __n) noexcept
(gdb) s
56   { return const_cast<_Tp&>(__t[__n]); }
(gdb) s
std::array<int, 10ul>::operator[] (this=0x7fffffffdf30, __n=0) at /usr/include/c++/9/array:190
190   { return _AT_Type::_S_ref(_M_elems, __n); }
(gdb) s
numbers[0] : 3
printFirstFive (numbers=...) at continue.cpp:6
6   for(int i = 0 ; i < 5 ; i++) {
(gdb) 

```

After individually stepping through the code line by line, we get to printing the first number in the array which is 3. Following this I chose to resume normal execution rather than step through the code (Command : c).

```

(gdb) s
std::array<int, 10ul>::operator[] (this=0x7fffffffdf30, __n=0) at /usr/include/c++/9/array:190
190         { return _AT_Type::_S_ref(_M_elems, __n); }
(gdb) s
numbers[0] : 3
printFirstFive (numbers=...) at continue.cpp:6
6         for(int i = 0 ; i < 5 ; i++) {
(gdb) c
Continuing.
numbers[1] : 4
numbers[2] : 6
numbers[3] : 3
numbers[4] : 2
We have printed the first 5 numbers in the array

Breakpoint 2, printRest (numbers=...) at continue.cpp:11
11 void printRest(const array<int, 10>& numbers) {
(gdb) c
Continuing.
numbers[5] : 7
numbers[6] : 2
numbers[7] : 9
numbers[8] : 6
numbers[9] : 5
We have printed the rest of the numbers of the array
[Inferior 1 (process 3925) exited normally]
(gdb) 

```

The break-point at function *printRest* was met and normal execution stopped. Following this I again used *continue* to resume execution until termination of program.

## Set - is used to set the value of a variable in the program during runtime

The value of the variable is not changed in the code but is rather used only for evaluation purposes during runtime.

```

#include <iostream>
using namespace std;

int main() {
    int dividend = 84;
    int divisor = 2;

    if(divisor == 0)
        cout << "Division by 0 is not possible.\n";
    else if (divisor > dividend)
        cout << "Divisor " << divisor << " is greater then dividend " << dividend <<
        ".\n";
    else
        cout << dividend << "/" << divisor << " = " << dividend/divisor <<'\n';

    return 0;
}

```

Using `set` to set values for divisor to check resilience of code.

```
riri@riri-VirtualBox:asgn 2$ gdb ./set -q
Reading symbols from ./set...
(gdb) break main
Breakpoint 1 at 0x11c9: file set.cpp, line 4.
(gdb) r
Starting program: /home/riri/Desktop/asgn 2/set

Breakpoint 1, main () at set.cpp:4
4      int main() {
(gdb) s
5          int dividend = 84;
(gdb) s
6          int divisor = 2;
(gdb) s
8          if(divisor == 0)
(gdb) s
10         else if (divisor > dividend)
(gdb) set divisor = 100
(gdb) s
11             cout << "Divisor " << divisor << " is greater then dividend " << dividend << ".\n";
(gdb) s
Divisor 100 is greater then dividend 84.
16         return 0;
(gdb) □
```

The condition `divisor > dividend` was evaluated at run-time by changing the runtime value of variable `divisor`.

```
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/riri/Desktop/asgn 2/set

Breakpoint 1, main () at set.cpp:4
4      int main() {
(gdb) s
5          int dividend = 84;
(gdb) s
6          int divisor = 2;
(gdb) s
8          if(divisor == 0)
(gdb) set divisor = 0
(gdb) s
9             cout << "Division by 0 is not possible.\n";
(gdb) s
Division by 0 is not possible.
16         return 0;
(gdb) □
```

The condition `divisor == 0` evaluated to true by setting the value of `divisor` variable at runtime.

## Part 8: Single stepping into function

Single stepping into a function means to execute every line of the function body one by one rather than treating it as a single line and executing its body in one go as done by *next*.

Variation *stepi* is also available

According to the help pages

*stepi* (*si*)

Step one instruction exactly.

Usage: *stepi* [N]

Argument N means step N times (or till program stops for another reason).

*step* (*s*)

Step program until it reaches a different source line.

Usage: *step* [N]

Argument N means step N times (or till program stops for another reason).

It's usage is shown in the demonstration of *continue* above in STEP 7

## Part 9: Listing all breakpoints

*info breaks* is the command to list all break-points in the current file

Using the same example code as in Continue, break-points are added and then displayed. Two different ways of adding a break are used; one through using *functionName* and another by using *filename:lineNumber*.

```
riri@riri-VirtualBox:asgn 2$ gdb ./continue -q
Reading symbols from ./continue...
(gdb) break main
Breakpoint 1 at 0x1307: file continue.cpp, line 17.
(gdb) break continue.cpp:11
Breakpoint 2 at 0x1278: file continue.cpp, line 11.
(gdb) info break
Num      Type           Disp Enb Address                What
1        breakpoint     keep y   0x0000000000001307 in main() at continue.cpp:17
2        breakpoint     keep y   0x0000000000001278 in printRest(std::array<int, 10ul> const&)
                                     at continue.cpp:11
(gdb) □
```

## Part 10: Ignoring a breakpoint for N occurrence

*ignore [breakpointnumber] [x]* - ignores break-point numbered (this is done automatically by the system while setting a

breakpoint) breakpointnumber until x hits are registered under it.

```
#include <iostream>
#include <array>
using namespace std;

void print(const array<int, 10>& nums) {
    for(int i = 0 ; i < 10 ; i++) {
        cout << "nums [" << i << "]" : " << nums[i] << '\n';
    }
}

int main() {
    array<int, 10> nums = {3, 4, 5, 6, 2, 7, 3, 8, 3, 10};
    print(nums);
    return 0;
}
```

Setting a break-point at line 7 and running the code

```
riri@riri-VirtualBox:asgn 2$ gdb ./break -q
Reading symbols from ./break...
(gdb) break break.cpp:7
Breakpoint 1 at 0x1207: file break.cpp, line 7.
(gdb) info break
Num      Type           Disp Enb Address            What
1        breakpoint      keep y   0x0000000000001207 in print(std::array<int, 10ul> const&)
                                at break.cpp:7
(gdb) ignore 1 8
Will ignore next 8 crossings of breakpoint 1.
(gdb) r
Starting program: /home/riri/Desktop/asgn 2/break
nums [0] : 3
nums [1] : 4
nums [2] : 5
nums [3] : 6
nums [4] : 2
nums [5] : 7
nums [6] : 3
nums [7] : 8

Breakpoint 1, print (nums=...) at break.cpp:7
7      cout << "nums [" << i << "]" : " << nums[i] << '\n';
(gdb) c
Continuing.
nums [8] : 3

Breakpoint 1, print (nums=...) at break.cpp:7
7      cout << "nums [" << i << "]" : " << nums[i] << '\n';
(gdb) c
Continuing.
nums [9] : 10
[Inferior 1 (process 4926) exited normally]
(gdb) □
```



We can see that gdb ignored the first 8 hits of the breakpoint by not stopping normal execution. After the first 8 hits were ignored, it resumed normal function of breakpoint from iteration 8.

## Part 11: Enable/disable a breakpoint

Let's say we run the following program using gdb:

```
#include <stdio.h>
void fun2(int s)
{
    printf("The value you entered is : %d\n",s);
}
void fun1()
{
    int vari;
    printf("Enter any value of variable you want to print :");
    scanf("%d",&vari);
    fun2(vari);
}
int natural_no(int num)
{
    int i, sum = 0;
    // use for loop until the condition becomes false
    for (i = 1; i <= num; i++)
    {
        // adding the counter variable i to the sum value
        sum += i;
    }
    return sum;
}
int main()
{
    int num, total; // local variable
    printf("Enter a natural number : ");
    scanf("%d", &num); // take a natural number from the user
    total = natural_no(num); // call the function
    printf("Sum of first %d natural numbers are : %d\n", num, total);
    fun1();
    return 0;
}
```

This is basically a simple program where the user can get the sum of all natural numbers from 1 to the number entered and through the other functions the user gets

to see the number entered.

Now after going through Step -1 which says running a program through debugger ⇒

a. Introducing Breakpoints and **displaying break points** with command ⇒ `info`

`break`

```
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from prog...
(gdb) break main
Breakpoint 1 at 0x1248: file prog.c, line 25.
(gdb) break fun2
Breakpoint 2 at 0x1189: file prog.c, line 3.
(gdb) break fun1
Breakpoint 3 at 0x11b1: file prog.c, line 7.
(gdb) break prog.c:27
Breakpoint 4 at 0x1263: file prog.c, line 27.
(gdb) info break
Num      Type             Disp Enb Address                What
1        breakpoint      keep y   0x0000000000001248 in main at prog.c:25
2        breakpoint      keep y   0x0000000000001189 in fun2 at prog.c:3
3        breakpoint      keep y   0x00000000000011b1 in fun1 at prog.c:7
4        breakpoint      keep y   0x0000000000001263 in main at prog.c:27
```

b. Now, here we are **Disabling the Breakpoint 1** ⇒ `disable 1`

```
(gdb) run
Starting program: /home/sayakrana/swe_lab/prog

Breakpoint 1, main () at prog.c:25
25      {
(gdb) disable 1
(gdb) info break
Num      Type             Disp Enb Address                What
1        breakpoint      keep n   0x0000555555555248 in main at prog.c:25
          breakpoint already hit 1 time
2        breakpoint      keep y   0x0000555555555189 in fun2 at prog.c:3
3        breakpoint      keep y   0x00005555555551b1 in fun1 at prog.c:7
4        breakpoint      keep y   0x0000555555555263 in main at prog.c:27
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/sayakrana/swe_lab/prog

Breakpoint 4, main () at prog.c:27
27      printf("Enter a natural number : ");
(gdb) █
```

So, after disabling the breakpoint 1 when we do `info break` again we see 'n' under Enb which tells that breakpoint 1 is not enabled anymore. Also , again on running the program we see directly the program stops on Breakpoint 4 and not on Breakpoint 1.

c. Here we are **Enabling the Breakpoint 1** ⇒ `enable 1`

```
Breakpoint 4, main () at prog.c:27
27      printf("Enter a natural number : ");
(gdb) enable 1
(gdb) info break
Num      Type          Disp Enb Address                What
1        breakpoint     keep y   0x0000555555555248 in main at prog.c:25
2        breakpoint     keep y   0x0000555555555189 in fun2 at prog.c:3
3        breakpoint     keep y   0x00005555555551b1 in fun1 at prog.c:7
4        breakpoint     keep y   0x0000555555555263 in main at prog.c:27
breakpoint already hit 1 time
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/sayakrana/swe_lab/prog

Breakpoint 1, main () at prog.c:25
25      {
(gdb) █
```

So, after enabling the breakpoint 1 when we do `info break` again we see 'y' under Enb which tells that breakpoint 1 is enabled. Also , again on running the program we see directly the program stops on Breakpoint 1.

d. Lastly, we are running the whole program by pressing `c` (continue) when we encounter the Breakpoints.

```

(gdb) c
Continuing.

Breakpoint 4, main () at prog.c:27
27         printf("Enter a natural number : ");
(gdb) c
Continuing.
Enter a natural number : 30
Sum of first 30 natural numbers are : 465

Breakpoint 3, fun1 () at prog.c:7
7         {
(gdb) c
Continuing.
Enter any value of variable you want to print :5

Breakpoint 2, fun2 (s=32767) at prog.c:3
3         {
(gdb) c
Continuing.
The value you entered is : 5
[Inferior 1 (process 8439) exited normally]
(gdb) 

```

## Part 12: Break condition and Command

Let's say we run the following program using gdb:

```

#include <stdio.h>
void display(int n)
{
    printf("Displaying the numbers from 1 to %d\n",n);
    int i;
    for(i = 1;i<=n;i++)
    {
        printf("%d ",i);
    }
}
int main()
{
    int num; // local variable
    printf("Enter number upto which you want to print: ");
    scanf("%d", &num); // take a natural number from the user
    display(num);
    return 0;
}

```

This is basically a simple program where the user enters a number and gets all the numbers from 1 to the number entered.

Now after going through Step -1 which says running a program through debugger ⇒

a. **Introducing breakpoint** in prog1.c at line 8 with condition if i%2 == 0 ⇒

```
break prog1.c:8 if i%2==0
```

```
(gdb) break prog1.c:8 if i%2==0
Breakpoint 1 at 0x11b7: file prog1.c, line 8.
(gdb) info break
Num      Type             Disp Enb Address            What
1        breakpoint        keep y   0x00000000000011b7 in display at prog1.c:8
          stop only if i%2==0
(gdb)
```

So, here what this command does is that it whenever it gets a multiple of 2 (say even number) it causes a break or pause in the execution of the program.

b. Running the program and pressing **c** (continue) till we get the output.

```
(gdb) run
Starting program: /home/sayakrana/swe_lab/prog1
Enter number upto which you want to print: 8
Displaying the numbers from 1 to 8

Breakpoint 1, display (n=8) at prog1.c:8
8      printf("%d ",i);
(gdb) print i
$1 = 2
(gdb) continue
Continuing.

Breakpoint 1, display (n=8) at prog1.c:8
8      printf("%d ",i);
(gdb) print i
$2 = 4
(gdb) continue
Continuing.

Breakpoint 1, display (n=8) at prog1.c:8
8      printf("%d ",i);
(gdb) print i
$3 = 6
(gdb) continue
Continuing.

Breakpoint 1, display (n=8) at prog1.c:8
8      printf("%d ",i);
(gdb) print i
$4 = 8
(gdb) continue
Continuing.
1 2 3 4 5 6 7 8 [Inferior 1 (process 3466) exited normally]
(gdb)
```

So, on pressing run we get the breakpoints whenever i gets value which is a multiple of 2.

Thus if n = 8, we get four breakpoints say at 2 , 4, 6 and 8.

On pressing `print i` and then `c` (continue) we can get to see the respective values of i too which are respectively 2,4,6 and 8 as specified already.

## Part 13: Examining Stack Trace

Let's define a recursive function to see how the stack trace examination works in the gdb

```
// filename: q13.c
#include <stdio.h>

long long factorial(int n){
    if (n == 1)
        return 1;
    if (n == 5)
        printf("n = 5 reached for debugger\n");
    if (n == 3)
        printf("n = 3 reached for debugger\n");

    return (long long)n * factorial(n - 1);
}

int main(){
    int value = 10;
    printf("factorial of %d is %lld\n", value, factorial(value));
    return 0;
}
```

This is a standard factorial recursive function which can overshoot for large value of n

Let's compile and run it with gdb

1. compile the program "gcc q13.c -o q13 -g"
2. run the program in gdb: "gdb q13"
3. add breakpoints to the line 7 and 9 (the printf's in the factorial function): "b 7" and "b 9"
4. check if breakpoints are set or not: "info b"
5. run the program till a breakpoint occurs: "r"

- a. Now the program will stop at the first breakpoint (at line 7)
6. examine the stack trace: "bt"
  - a. Output from my side:

```
(gdb) r
Starting program: /home/abhiroop/gdbStuffs/q13
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, factorial (n=5) at q13.c:7
warning: Source file is more recent than executable.
7      int main(){
(gdb) bt
#0 factorial (n=5) at q13.c:7
#1 0x00005555555551c3 in factorial (n=6) at q13.c:11
#2 0x00005555555551c3 in factorial (n=7) at q13.c:11
#3 0x00005555555551c3 in factorial (n=8) at q13.c:11
#4 0x00005555555551c3 in factorial (n=9) at q13.c:11
#5 0x00005555555551c3 in factorial (n=10) at q13.c:11
#6 0x00005555555551ea in main () at q13.c:16
(gdb) █
```

- b. Here we can see the stack trace of the program in execution, the factorial function is called from main, then it is recursively called till the breakpoint at  $n = 5$
7. continue till next breakpoint: "c"
  - a. Now the program will stop again at breakpoint of line 9
8. examine the stack trace again: "bt"
  - a. output from my side:

```
Breakpoint 2, factorial (n=3) at q13.c:9
9      printf("factorial of %d is %lld\n", value, factorial(value));
(gdb) bt
#0 factorial (n=3) at q13.c:9
#1 0x00005555555551c3 in factorial (n=4) at q13.c:11
#2 0x00005555555551c3 in factorial (n=5) at q13.c:11
#3 0x00005555555551c3 in factorial (n=6) at q13.c:11
#4 0x00005555555551c3 in factorial (n=7) at q13.c:11
#5 0x00005555555551c3 in factorial (n=8) at q13.c:11
#6 0x00005555555551c3 in factorial (n=9) at q13.c:11
#7 0x00005555555551c3 in factorial (n=10) at q13.c:11
#8 0x00005555555551ea in main () at q13.c:16
(gdb) █
```

- b. Here we see that the previous stack trace grew from  $n = 5$  to  $n = 3$ , as expected



9. continue till the next breakpoint (there aren't any so the program will complete executing) : "c"
10. exit from gdb: "q"

## Part 14: Examining stack trace for multi-threaded program

consider the following simple program for this part

```
//filename q14.c
#include <stdlib.h>
#include <pthread.h>
#include <stdio.h>

#define NUM_THREADS 5

void *PrintHello(void *threadid) {
    long tid;
    tid = (long)threadid;
    printf("Hello World! Thread ID, %ld\n", tid);
    pthread_exit(NULL);
}

int main () {
    pthread_t threads[NUM_THREADS];
    int rc;
    int i;
    for( i = 0; i < NUM_THREADS; i++ ) {
        printf("main() : creating thread, %d\n", i);
        rc = pthread_create(&threads[i], NULL, PrintHello, (void *)i);
        if (rc) {
            printf("Error:unable to create thread, %d\n", rc);
            exit(-1);
        }
    }
    pthread_exit(NULL);
}
```

Here the main calls five threads and they print hello and exit

1. compile the program: "gcc -g q14.c -o q14 -pthread"
2. open program in gdb: "gdb q14"
3. add a breakpoint where we can stop the processes: "b 10"
4. start the program: "r"
  - a. we see the gdb notes that main() is creating threads

```
(gdb) r
Starting program: /home/abhiroop/gdbStuffs/q14
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
main() : creating thread, 0
[New Thread 0x7ffff7d8f640 (LWP 2543)]
main() : creating thread, 1
[New Thread 0x7ffff758e640 (LWP 2544)]
[Switching to Thread 0x7ffff7d8f640 (LWP 2543)]

Thread 2 "q14" hit Breakpoint 1, PrintHello (threadid=0x0) at q14.c:10
warning: Source file is more recent than executable.
10      printf("Hello World! Thread ID, %ld\n", tid);
(gdb) █
```

b. Now the program will stop when the threads are running

5. see the current thread: "thread"

```
(gdb) thread
[Current thread is 2 (Thread 0x7ffff7d8f640 (LWP 2543))]
(gdb) █
```

6. see the back trace of the current thread: "bt"

a. we see that the thread is not back tracing to main, it is back tracing to start\_thread implemented in the OS itself

```
(gdb) bt
#0 PrintHello (threadid=0x0) at q14.c:10
#1 0x00007ffff7e2a927 in start_thread (arg=<optimized out>) at pthread_create.c:435
#2 0x00007ffff7eba9e4 in clone () at ../sysdeps/unix/sysv/linux/x86_64/clone.S:100
```

7. see the back trace of all the processes: "thread apply all bt"

a. here we can see that thread 1 is from main making all the threads (was only able to make two threads before thread 2 stopped), and thread 3 has been made but hasn't started to execute PrintHello() just yet

```
(gdb) thread apply all bt

Thread 3 (Thread 0x7ffff758e640 (LWP 2544) "q14"):
#0  clone () at ../sysdeps/unix/sysv/linux/x86_64/clone.S:83
#1  0x00007ffff7e2a6b0 in ?? () at pthread_create.c:321 from /lib/x86_64-linux-gnu/libc.so.6
#2  0x00007ffff758e640 in ?? ()
#3  0x0000000000000000 in ?? ()

Thread 2 (Thread 0x7ffff7d8f640 (LWP 2543) "q14"):
#0  PrintHello (threadid=0x0) at q14.c:10
#1  0x00007ffff7e2a927 in start_thread (arg=<optimized out>) at pthread_create.c:435
#2  0x00007ffff7eba9e4 in clone () at ../sysdeps/unix/sysv/linux/x86_64/clone.S:100

Thread 1 (Thread 0x7ffff7fbb5c0 (LWP 2539) "q14"):
#0  clone () at ../sysdeps/unix/sysv/linux/x86_64/clone.S:83
#1  0x00007ffff7ebb7a1 in __GI___clone_internal (cl_args=cl_args@entry=0x7ffff7fdd50, func=func@entry=0x7ffff7e2a6b0 <start_thread>, arg=arg@entry=0x7ffff758e640) at ../sysdeps/unix/sysv/linux/clone-internal.c:72
#2  0x00007ffff7e2a5b5 in create_thread (pd=pd@entry=0x7ffff758e640, attr=attr@entry=0x7ffff7fde50, stopped_start=stopped_start@entry=0x7ffff7fde4e, stackaddr=stackaddr@entry=0x7ffff7d8e000, stacksize=<optimized out>, thread_ran=thread_ran@entry=0x7ffff7fde4f) at pthread_create.c:295
#3  0x00007ffff7e2b036 in __pthread_create_2_1 (newthread=<optimized out>, attr=<optimized out>, start_routine=<optimized out>, arg=<optimized out>) at pthread_create.c:799
#4  0x0000555555555252 in main () at q14.c:19
(gdb) █
```

8. now you can exit or keep running the program till execution ends

## Part 15: Core file debugging

A “core dump” is a snapshot of memory at the instant the program crashed, typically saved in a file called “core”. GDB can read the core dump and give you the line number of the crash, the arguments that were passed, and more.

Let’s write a simple program to generate a core dump and debug it.

```
#include<iostream>
using namespace std;

// function definition
int divide(int,int);

int main(){
    int x = 10 , y = 5;
    int xDivY = divide(x,y);
    cout << xDivY << "\n";

    x = 3; y = 0;
    // expecting error here !
    xDivY = divide(x,y);
    cout << xDivY << "\n";

    return 0;
}

// Takes two argument 'a' and 'b'
// Return 'a/b'
int divide(int a, int b){
    return a / b;
}
```

1. We will first compile this program using **g++** . Keep in mind to use **-g** option to debug with help of GDB. After that , we will load our executable file using gdb.

```
nickjosi@Nikhil:~/Desktop/study/6thSemester/Assignment/Software Engineering$ g++ -g coreDump.cpp -o coreDump
nickjosi@Nikhil:~/Desktop/study/6thSemester/Assignment/Software Engineering$ gdb coreDump
```

2. With **r** as the command, run the executable. Since there are not any breakpoints, it will run complete code.

```
(gdb) r
Starting program: /home/nickjosi/Desktop/study/6thSemester/Assignment/Software Engineering/coreDump

Program received signal SIGFPE, Arithmetic exception.
0x000055555555524e in divide (a=3, b=0) at coreDump.cpp:19
19         return a / b;
```

gdb is showing some information about core-dump. It is showing that our program has received the signal **Floating Point Error**. It is also showing **the line at which exception takes place**. It is on **line 19** , in function **divide** where arguments are **a = 3** and **b = 0** respectively. It is also listing line number 19, “**return a/b**”.

3. We can print out context ( code ) where exception (/core dump) occur using ‘**l**’ command.

```
(gdb) l
14
15         return 0;
16     }
17
18     int divide(int a, int b){
19         return a / b;
20     }
```

4. We can print the Stack trace using “**where**” command.

```
(gdb) where
#0  0x00005555555524e in divide (a=3, b=0)
    at coreDump.cpp:19
#1  0x000055555555212 in main () at coreDump.cpp:12
( ... )
```

This can be read as: The crash occurred in the function **divide** at **line 19** of **coreDump.cpp**. This, in turn, was called from the **function main at line 13** of **coreDump.cpp**.

5. We can use the **up** and the **down** command to move from the current level of stack trace up or down one level. Here, we are going from default level ‘0’ to one level up using **up**.

We are also using **list** to list the code near to command from where the divide function is called.

```
(gdb) up
#1  0x000055555555212 in main () at coreDump.cpp:12
12      xDivY = divide(x,y);
(gdb) list
7          int x = 10 , y = 5;
8          int xDivY = divide(x,y);
9          cout << xDivY << "/n";
10
11         x = 3; y = 0;
12         xDivY = divide(x,y);
13         cout << xDivY << "/n";
14
15         return 0;
16     }
```

6. We can print variable of current level of stack using **print** command.

```
(gdb) p x
$1 = 3
(gdb) p {x,y}
$2 = {3, 0}
```

## Part 16. Debugging of an already running assignment.

When a program is not started as a process yet, we can attach it with gdb and then run it using **r** command.

```
|  gdb program
```

But if the program is already running as a process then we have to perform following steps in order to attach gdb with it :

1. Find the process id (*pid*) with the help of **pidof** command:

```
pidof program
# Replace program with a file name or path to the program.
```

2. Attach GDB to this process:

```
gdb program -p pid
# Replace program with a file name or path to the program, replace pid with an actual
```

1. Code

```
coreDump.cpp X
coreDump.cpp > main()
4  int divide(int,int);
5
6  int main(){
7      int x=10, y=5;
8      int xDivY=divide(x,y);
9      cout<<xDivY<<"\n";
10
11     cout<<"Give x and y as input.\n";
12     cin>>x>>y;
13     xDivY=divide(x,y);
14     cout<<xDivY<<"/n";
15
16     return 0;
17 }
18
19 int divide(int a, int b){
20     return a/b;
21 }
```

2. Process start running

```
nickjosi@Nikhil:~/Desktop/study/6thSemester/Assignment/Software Engineering$ ./coreDump
2
Give x and y as input.
```

3. Finding Process id using **pidof** coreDump program running.

```
nickjosi@Nikhil:~/Desktop/study/6thSemester/Assignment/Software Engineering$ pidof coreDump
99427
```



4. Attaching gdb with program running with above mentioned process id.

```
nickjosi@Nikhil:~/Desktop/study/6thSemester/Assignment/Software Engineering$ sudo gdb "/home/nickjosi/Desktop/study/6thSemester/Assignment/Software Engineering/coreDump" -p 99427
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 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.
```

5. Now we can do debugging in our normal manner.

## Part 17: Watchpoint

Watchpoints are a special kind of breakpoint which is bound to a variable rather than line number

When the value of the variable being watched changes, the program is stopped

Consider a simple program

```
// filename: q17.c
#include <stdio.h>

int main(){
    for (int i = 0; i < 5; i++){
        if (i % 2 == 0)
            i++;
        printf("%d\n", i);
    }
    return 0;
}
```

Lets see the watch point in action

1. compile the code: "gcc -g q17.c -o q17"
2. open the program with gdb: "gdb q17"

3. add a breakpoint in main to stop the program just after the execution starts: "b main"
4. run the program to start execution of program: "r"
  - a. The program will stop just after the start of execution

```
(gdb) r
Starting program: /home/abhiroop/gdbStuffs/q17
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main () at q17.c:4
4      for (int i = 0; i < 5; i++){
```

5. add a watchpoint to i: "watch i"
6. Now keep running the program with "c", we observe that the program stops everytime i changes

```

(gdb) c
Continuing.

Hardware watchpoint 2: i

Old value = 0
New value = 1
main () at q17.c:7
7      printf("%d\n", i);
(gdb) c
Continuing.
1

Hardware watchpoint 2: i

Old value = 1
New value = 2
0x000055555555189 in main () at q17.c:4
4      for (int i = 0; i < 5; i++){
(gdb) c
Continuing.

Hardware watchpoint 2: i

Old value = 2
New value = 3
main () at q17.c:7
7      printf("%d\n", i);
(gdb) c
Continuing.
3

Hardware watchpoint 2: i

Old value = 3
New value = 4
0x000055555555189 in main () at q17.c:4
4      for (int i = 0; i < 5; i++){
(gdb) c
Continuing.

Hardware watchpoint 2: i

Old value = 4
New value = 5
main () at q17.c:7
7      printf("%d\n", i);
(gdb) c
Continuing.
5

Hardware watchpoint 2: i

Old value = 5
New value = 6

```