Subject : Software Engineering

Subject Code : CS3273

Section: Gx

Topic : Exploring GDB commands

Enrollment no.	Name
2020CSB007	ARATRIKA PAL
2020CSB009	MAYANK MANAVENDRA KUMAR
2020CSB010	GOURAV KUMAR SHAW
2020CSB011	TAMOGHNA ROY
2020CSB013	ABDUL KHAZMUDDIN
2020CSB014	SIDDARTHA JANA

1. Write a short note on GNU debugger(GDB) command in Linux.

Answer:

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 goto debugger till today.

GDB allows developers to inspect and manipulate the internal state of programs, including their memory and registers, to identify and diagnose issues that cause crashes, errors, and other unexpected behavior.

Part 1: Running a program

```c
#include <stdio.h>

int main() {
    int x = 1;
    printf("The value of x is %d\n", x);
    return 0;
}
```

- Generally, we do "gcc My_program.c -o My_program"
- 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 My_program.c -o My_program -g"
- Now we just have to do "gdb My_program" 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

- When we compile a program with debugging information using the −g flag, the compiler generates a symbol table for the program.
- The symbol table contains information about the program's source code, such as the names of functions and variables, their types, and their memory addresses. GDB uses this symbol table to provide more meaningful debugging information when we debug the program.

```
(gdb) file sample
Load new symbol table from "sample"? (y or n) y
Reading symbols from sample...
(No debugging symbols found in sample)
(gdb)
```

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}*/
```

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 out break point.

5. The command "`info break`" will show us all the break points currently held within our program.

6. The command "`continue`" will resume the execution which was previously halted after reaching a break point.

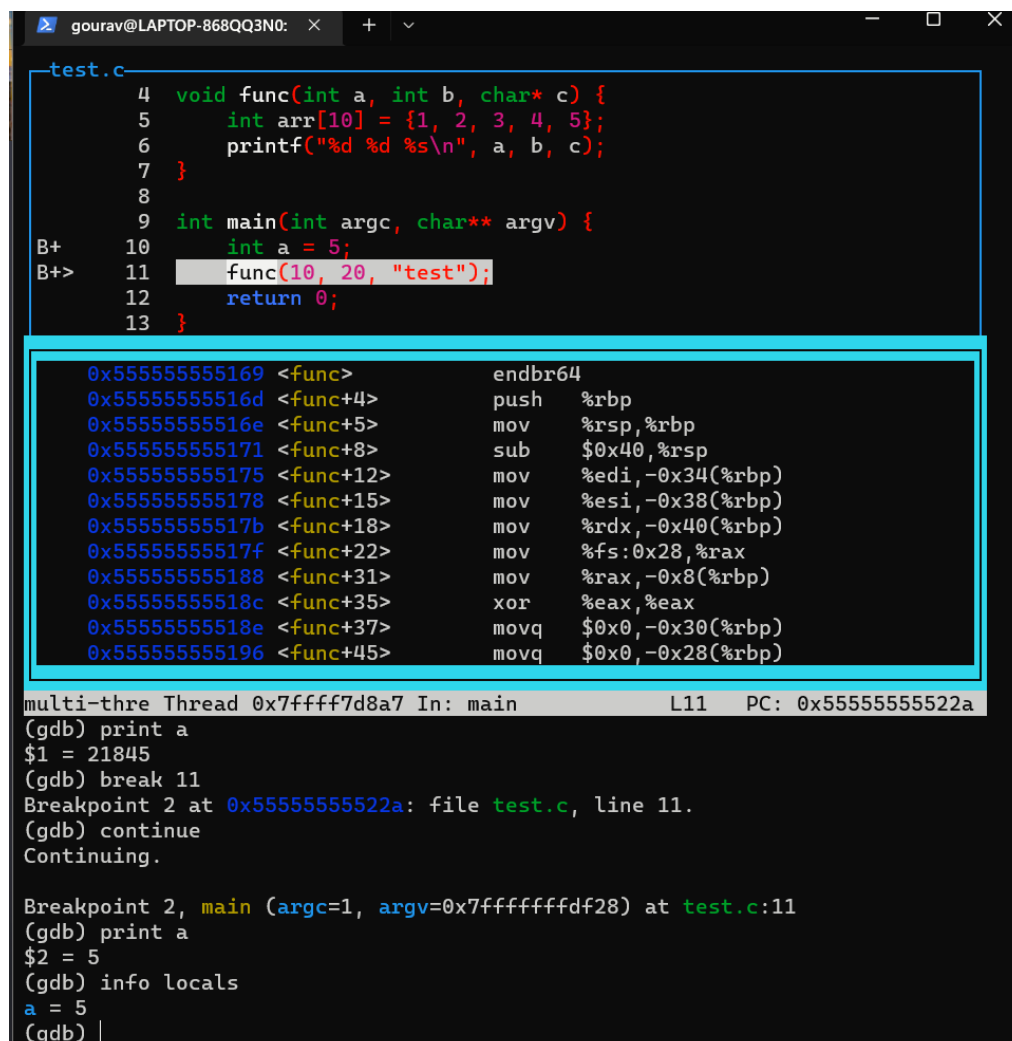7. The command "`delete /*{breakpoint number}*/` will delete a breakpoint

```
(gdb) break 5
Breakpoint 1 at 0x115c: file My_program.c, line 5.
(gdb) run
Starting program: /mnt/c/Users/Gourav Kumar Shaw/Desktop/6th_sem_main_assignments/software_eng_assignment/My_program.out
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main () at My_program.c:5
5           printf("The value of x is %d\n", x);
(gdb) info locals
x = 1
(gdb)
```

Part 4. Listing variables and examining their values.

Code :

```c
1   #include <stdio.h>
2
3
4   void func(int a, int b, char* c) {
5       int arr[10] = {1, 2, 3, 4, 5};
6       printf("%d %d %s\n", a, b, c);
7   }
8
9   int main(int argc, char** argv) {
10      int a = 5;
11      func(10, 20, "test");
12      return 0;
13  }
14
```

- **info locals**: The **info locals** command is used to display the values of local variables in the current scope. When you are stopped at a breakpoint or when your program is running in the debugger, we can use **info locals** to see the values of all local variables in the current function.

- In this example, **info locals** was used to display the value of the local variable a. Here **info locals** gives the output **a=5**.

- **info locals** will only show the values of local variables in the current scope. If we have nested scopes or if we are using function calls, we may need to use the up and down commands to change the current scope and display the values of local variables in other scopes.

- Note that the **info locals** command does not display the information about the function arguments

Part 5. Printing content of an array or contiguous memory

```
gourav@LAPTOP-868QQ3N0:    ×    +    ˅

┌─test.c─────────────────────────────────────────────────
│B+      4   void func(int a, int b, char* c) {
│        5       int arr[10] = {1, 2, 3, 4, 5};
│  >     6       printf("%d %d %s\n", a, b, c);
│        7   }
│        8
│        9   int main(int argc, char** argv) {
│B+     10       int a = 5;
│B+     11       func(10, 20, "test");
│       12       return 0;
│       13   }
│       14
│       15
│       16
│       17
│       18
│       19
│       20
│       21
│       22
│       23
│       24
│       25
│       26
│       27
└────────────────────────────────────────────────────────
multi-thre Thread 0x7ffff7d8a7 In: func

Breakpoint 3, func (a=10, b=20, c=0x55555555600e "test") at test.c:4
(gdb) n
(gdb) print arr
$2 = {1431650368, 21845, -134338468, 32767, 0, 0, -7831, 32767,
  -134471680, 32767}
(gdb) n
(gdb) print arr
$3 = {1, 2, 3, 4, 5, 0, 0, 0, 0, 0}
(gdb) x/5w arr
0x7fffffffddb0: 1        2        3        4
0x7fffffffddc0: 5
(gdb) |
```

- **x/5w arr:** In GDB, the x command is used to examine memory. The **/5w** in the **x/5w arr** command is a format specifier that tells GDB how to format the output.

- The **/5w** format specifier is used to display 5 memory words in hexadecimal format. A word is typically 4 bytes on most systems, so this command will display the values of 20 bytes of memory.

- The **arr** argument specifies the memory address to start displaying from, which is name of array. In this example, **x/5w arr** was used to display the values of the first 5 elements of the **arr** array. The output shows that the values of the elements are stored in consecutive memory locations, starting from the address :**0x7fffffffddb0.**

- If we gives the command **print arr** then it will gives all the array elements including 0.

- But if we want only particular elements to print then we have to use **x/nw arr** were n is number of element we want to print.

Part 6. Printing function arguments

```
┌─test.c──────────────────────────────────────
│B+      4    void func(int a, int b, char* c) {
│        5        int arr[10] = {1, 2, 3, 4, 5};
│  >     6        printf("%d %d %s\n", a, b, c);
│        7    }
│        8
│        9    int main(int argc, char** argv) {
│B+     10        int a = 5;
│B+     11        func(10, 20, "test");
│       12        return 0;
│       13    }
│       14
│       15
│       16
│       17
│       18
│       19
│       20
│       21
│       22
│       23
│       24
│       25
│       26
│       27
├─────────────────────────────────────────────
multi-thre Thread 0x7ffff7d8a7 In: func
0x7fffffffddb0: 1          2          3          4
0x7fffffffddc0: 5
(gdb) info args
a = 10
b = 20
c = 0x55555555600e "test"
(gdb) print a
$4 = 10
(gdb) print b
$5 = 20
(gdb) print c
$6 = 0x55555555600e "test"
(gdb) |
```

- **info args:** the **info args** command is used to display the arguments of the current function. When we are stopped at a breakpoint or when our program is running in the debugger.

- we can use **info args** to see the values of all arguments passed to the current function.

- **info args** will only show the values of arguments in the current function. If we are calling other functions, we need to set breakpoints in those functions and use **info args** to see the values of arguments passed to those functions.

- Here in our program if we give the command **info args** it gives the values of arguments **a=10, b=20 and c=0x55555555600e "test".**

Part 7. Next, Continue, Set command

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 it's 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).

```
 14       int main(int argc, char const *argv[])
 15
 16       {
B+  17         int n = atoi(argv[1]);
  > 18         int ans = sumOfn(n);
 19         printf("%d\n", ans);
 20         return 0;
 21       }            1; i <= n; i++)
 22
 23            sum += i;
 24
native process 56372 In: main
Breakpoprocess 4802 In: main 7: file def.c, line 17.
(gdb) print ans
$1 = 10
(gdb) continue
Continuing.
10
[Inferior 1 (process 56372) exited normally]
(gdb) run 4
Starting program: /home/hanau1/student/btech2020/mayankk/def 4

Breakpoint 1, main (argc=2, argv=0x7fffffffeaa8) at def.c:17
(gdb) next
(gdb) ▏
```

next: execute the next line of code (or a function without going into it). While at line 18 and giving the command `next`, line 19 is executed and not the function at line 5

```
14        int main(int argc, char const *argv[])
15
16        {
B+  17            int n = atoi(argv[1]);
18            int ans = sumOfn(n);
>   19            printf("%d\n", ans);
20            return 0;
21        }                   1; i <= n; i++)
22
23            sum += i;
24
native process 56372 In: main
Breakpoprocess 4802 In: main 7: file def.c, line 17.
$1 = 10
(gdb) continue
Continuing.
10
[Inferior 1 (process 56372) exited normally]
(gdb) run 4
Starting program: /home/hanau1/student/btech2020/mayankk/def 4

Breakpoint 1, main (argc=2, argv=0x7fffffffeaa8) at def.c:17
(gdb) next
(gdb) next
(gdb) 
```

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

```
     14        int main(int argc, char const *argv[])
     15
     16        {
B+   17            int n = atoi(argv[1]);
 >   18            int ans = sumOfn(n);
     19            printf("%d\n", ans);
     20            return 0;
     21        }
     22
     23
```

```
native process 18782 In: main
(gdb) run 4
Starting program: /home/hanau1/student/btech2020/mayankk/def 4
10
(gdb) break main
Breakpoint 1 at 0x5555555546c7: file def.c, line 17.
(gdb) run 4
Starting program: /home/hanau1/student/btech2020/mayankk/def 4

Breakpoint 1, main (argc=2, argv=0x7fffffffeaa8) at def.c:17
(gdb) next
(gdb) █
```

```
Breakpoint 1, main (argc=2, argv=0x7fffffffeaa8) at def.c:17
(gdb) next
(gdb) continue
Continuing.
10
[Inferior 1 (process 18782) exited normally]
(gdb) █
```

continue: continue normal execution, stopping at next breakpoints (if any).

<u>Set</u> - 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.
Set a debugger variable to a value.

```
[(gdb) run 5
Starting program: /home/hanau1/student/btech2020/mayankk/def 5
15
[Inferior 1 (process 39800) exited normally]
[(gdb) run 4
Starting program: /home/hanau1/student/btech2020/mayankk/def 4
10
[Inferior 1 (process 39835) exited normally]
```

run 5 gives value 15 and run 4 gives value 10.

```
Breakpoint 1, main (argc=2, argv=0x7fffffffeaa8) at def.c:17
(gdb) n
(gdb) print n
$1 = 5
(gdb) set variable n = 4
(gdb) print n
$2 = 4
(gdb) n
(gdb) n
10
```

Changing the value of n using set changes the value from 15 to 10.

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 it's 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).

step: stepping means executing just one more "step" of your program, where "step" may mean either one line of source code, or one machine instruction

```
14      int main(int argc, char const *argv[])
15
16      {
B+ 17           int n = atoi(argv[1]);
   > 18          int ans = sumOfn(n);
19              printf("%d\n", ans);
20              return 0;
21      }
22
23
```

```
native process 58992 In: main
(gdb) break main
Breakpoint 1 at 0x6c7: file def.c, line 17.
(gdb) run 5
Starting program: /home/hanau1/student/btech2020/mayankk/def 5

Breakpoint 1, main (argc=2, argv=0x7fffffffeaa8) at def.c:17
(gdb) next
(gdb) 
```

```
5       int sumOfn(int n){
   > 6           int sum = 0;
7               for (int i = 1; i <= n; i++)
8               {
9                   sum += i;
10              }
11              return sum;
12      }
13
14      int main(int argc, char const *argv[])
15
16      {
B+ 17           int n = atoi(argv[1]);
18              int ans = sumOfn(n);
19              printf("%d\n", ans);
20              return 0;
21      }
22
23
```

```
native process 58992 In: sumOfn
(gdb) break main
Breakpoint 1 at 0x6c7: file def.c, line 17.
(gdb) run 5
Starting program: /home/hanau1/student/btech2020/mayankk/def 5

Breakpoint 1, main (argc=2, argv=0x7fffffffeaa8) at def.c:17
(gdb) next
(gdb) step
sumOfn (n=5) at def.c:6
(gdb) 
```

Part 9: Listing all breakpoints

`info break` is the command to list all break-points in the current file.

```
     5          int sumOfn(int n){
b+   6              int sum = 0;
     7              for (int i = 1; i <= n; i++)
     8              {
     9                  sum += i;
    10              }
    11              return sum;
    12          }
    13
    14          int main(int argc, char const *argv[])
    15
    16          {
b+  17              int n = atoi(argv[1]);
    18              int ans = sumOfn(n);
    19              printf("%d\n", ans);
    20              return 0;
    21          }
    22
    23
```
```
exec No process In:
(gdb) run 5
Starting program: /home/hanau1/student/btech2020/mayankk/def 5
15
(gdb) break main
Breakpoint 1 at 0x5555555546c7: file def.c, line 17.
(gdb) break sumOfn
Breakpoint 2 at 0x555555554691: file def.c, line 6.
(gdb) info b
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x00005555555546c7 in main at def.c:17
2       breakpoint     keep y   0x0000555555554691 in sumOfn at def.c:6
(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.

```cpp
#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;
}
```

```
(gdb) break 7
Breakpoint 1 at 0x1203: file test.cpp, line 8.
(gdb) info break
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000000001203 in print(std::array<int, 10ul> const&) at test.cpp:8
(gdb) ignore 1 8
Will ignore next 8 crossings of breakpoint 1.
(gdb) r
Starting program: /mnt/c/Users/Gourav Kumar Shaw/Desktop/6th_sem_main_assignments/software_eng_assignment/test
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
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 test.cpp:8
8           cout << "nums [" << i << "] : " << nums[i] << '\n';
(gdb) c
Continuing.
nums [8] : 3

Breakpoint 1, print (nums=...) at test.cpp:8
8           cout << "nums [" << i << "] : " << nums[i] << '\n';
(gdb) c
Continuing.
nums [9] : 10
[Inferior 1 (process 1674) 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 break-point from iteration 8.

Part 11: Enable/disable a breakpoint

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

```c
#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`

```
 gourav    LAPTOP-868QQ3N0    ../software_eng_assignment    master ↻    gdb break_point
GNU gdb (Ubuntu 12.0.90-0ubuntu1) 12.0.90
Copyright (C) 2022 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.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from break_point...
(gdb) break 3
Breakpoint 1 at 0x1198: file break_point.cpp, line 4.
(gdb) break 8
Breakpoint 2 at 0x11cf: file break_point.cpp, line 9.
(gdb) break 11
Breakpoint 3 at 0x11fe: file break_point.cpp, line 11.
(gdb) break 17
Breakpoint 4 at 0x1231: file break_point.cpp, line 17.
(gdb) info break
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000000001198 in fun2(int) at break_point.cpp:4
2       breakpoint     keep y   0x00000000000011cf in fun1() at break_point.cpp:9
3       breakpoint     keep y   0x00000000000011fe in fun1() at break_point.cpp:11
4       breakpoint     keep y   0x0000000000001231 in natural_no(int) at break_point.cpp:17
(gdb)
```

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

```
(gdb) break 17
Breakpoint 4 at 0x1231: file break_point.cpp, line 17.
(gdb) info break
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000000001198 in fun2(int) at break_point.cpp:4
2       breakpoint     keep y   0x00000000000011cf in fun1() at break_point.cpp:9
3       breakpoint     keep y   0x00000000000011fe in fun1() at break_point.cpp:11
4       breakpoint     keep y   0x0000000000001231 in natural_no(int) at break_point.cpp:17
(gdb) disable 1
(gdb) info break
Num     Type           Disp Enb Address            What
1       breakpoint     keep n   0x0000000000001198 in fun2(int) at break_point.cpp:4
2       breakpoint     keep y   0x00000000000011cf in fun1() at break_point.cpp:9
3       breakpoint     keep y   0x00000000000011fe in fun1() at break_point.cpp:11
4       breakpoint     keep y   0x0000000000001231 in natural_no(int) at break_point.cpp:17
(gdb) run
Starting program: /mnt/c/Users/Gourav Kumar Shaw/Desktop/6th_sem_main_assignments/software_eng_assignment/break_point
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Enter a natural number : 7

Breakpoint 4, natural_no (num=7) at break_point.cpp:17
17          for (i = 1; i <= num; i++)
(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

So,

```
(gdb) enable 1
(gdb) info break
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000555555555198 in fun2(int) at break_point.cpp:4
2       breakpoint     keep y   0x00005555555551cf in fun1() at break_point.cpp:9
3       breakpoint     keep y   0x00005555555551fe in fun1() at break_point.cpp:11
4       breakpoint     keep y   0x0000555555555231 in natural_no(int) at break_point.cpp:17
        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: /mnt/c/Users/Gourav Kumar Shaw/Desktop/6th_sem_main_assignments/software_e
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Enter a natural number : 34

Breakpoint 4, natural_no (num=34) at break_point.cpp:17
17          for (i = 1; i <= num; i++)
(gdb) ▮
```

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

```
(gdb) c
Continuing.
Sum of first 34 natural numbers are : 595

Breakpoint 2, fun1 () at break_point.cpp:9
9          printf("Enter any value of variable you want to print :");
(gdb) c
Continuing.
Enter any value of variable you want to print :7

Breakpoint 3, fun1 () at break_point.cpp:11
11             fun2(vari);
(gdb)
```

Part 12: Break condition and Command

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

```c
#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.cpp 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

```
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

When our program has stopped, the first thing we need to know is where it stopped and how it got there.

Each time our program performs a function call, the information about where in your program the call was made from is saved in a block of data called a *stack frame*. The frame also contains the arguments of the call and the local variables of the function that was called. All the stack frames are allocated in a region of memory called the *call stack*.

When your program stops, the GDB commands for examining the stack allow us to see all of this information.

our test program

Here's a simple C program that declares a few variables and reads two strings from standard input. One of the strings is on the heap, and one is on the stack.

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

int main() {
    char stack_string[10] = "stack";
    int x = 10;
    char *heap_string;

    heap_string = malloc(50);

    printf("Enter a string for the stack: ");
    gets(stack_string);
    printf("Enter a string for the heap: ");
    gets(heap_string);
    printf("Stack string is: %s\n", stack_string);
    printf("Heap string is: %s\n", heap_string);
```

```
    printf("x is: %d\n", x);
}
```

step 0: compile the program.

We can compile it with `gcc -g -O0 test.c -o test`.

The `-g` flag compiles the program with debugging symbols, which is going to make it a lot easier to look at our variables.

`-O0` tells gcc to turn off optimizations which I did just to make sure our `x` variable didn't get optimized out.

step 1: start gdb

We can start gdb like this:

```
$ gdb ./test
```

It prints out some stuff about the GPL and then gives a prompt. Let's create a breakpoint on the `main` function.

```
(gdb) b main
Breakpoint 1 at 0x1171: file test.c, line 4.
```

Then we can run the program:

```
(gdb) run
Starting program: /home/bork/work/homepage/test

Breakpoint 1, main () at test.c:4
4       int main() {
```

Okay, great! The program is running and we can start looking at the stack

step 2: look at our variables' addresses

Let's start out by learning about our variables. Each of them has an address in memory, which we can print out like this:

```
(gdb) p &x
$3 = (int *) 0x7fffffffe27c
(gdb) p &heap_string
$2 = (char **) 0x7fffffffe280
(gdb) p &stack_string
$4 = (char (*)[10]) 0x7fffffffe28e
```

So if we look at the stack at those addresses, we should be able to see all of these variables!

concept: the stack pointer

We're going to need to use the stack pointer so I'll try to explain it really quickly.

There's an x86 register called ESP called the "stack pointer". Basically it's the address of the start of the stack for the current function. In gdb you can access it with $sp. When you call a new function or return from a function, the value of the stack pointer changes.

step 3: look at our variables on the stack at the beginning of `main`

First, let's look at the stack at the start of the `main` function. Here's the value of our stack pointer right now:

```
(gdb) p $sp


$7 = (void *) 0x7fffffffe270
```

So the stack for our current function starts at `0x7fffffffe270`.

Now let's use gdb to print out the first 40 words (aka 160 bytes) of memory after the start of the current function's stack. It's possible that some of this memory isn't part of the stack because I'm not totally sure how big the stack is here. But at least the beginning of this is part of the stack.

```
(gdb) x/40x $sp

0x7fffffffe270: 0x00000000  0x00000000  0x55555250  0x00005555
0x7fffffffe280: 0x00000000 0x00000000  0x55555070  0x00005555
0x7fffffffe290: 0xffffe390 0x00007fff  0x00000000  0x00000000
0x7fffffffe2a0: 0x00000000  0x00000000  0xf7df4b25  0x00007fff
0x7fffffffe2b0: 0xffffe398  0x00007fff  0xf7fca000  0x00000001
0x7fffffffe2c0: 0x55555169  0x00005555  0xffffe6f9  0x00007fff
0x7fffffffe2d0: 0x55555250  0x00005555  0x3cae816d  0x8acc2837
0x7fffffffe2e0: 0x55555070  0x00005555  0x00000000  0x00000000
0x7fffffffe2f0: 0x00000000  0x00000000  0x00000000  0x00000000
0x7fffffffe300: 0xf9ce816d  0x7533d7c8  0xa91a816d  0x7533c789
```

I've bolded approximately where the `stack_string`, `heap_string`, and `x` variables are and colour coded them:

- `x` is red and starts at `0x7fffffffe27c`
- `heap_string` is blue and starts at `0x7fffffffe280`
- `stack_string` is purple and starts at `0x7fffffffe28e`

Part 14. Examining stack trace for multi-threaded program

`dbx` can debug multithreaded applications that use either Solaris threads or POSIX threads. With `dbx`, we can examine stack traces of each thread, resume all threads, `step` or `next` a specific thread, and navigate between threads.

`dbx` recognizes a multithreaded program by detecting whether it utilizes `libthread.so`. The program uses `libthread.so` either by explicitly being compiled with `-lthread` or `-mt`, or implicitly by being compiled with `-lpthread`.

Part 15. Core file debugging

Core files are produced as a result of a fatal error in a program. A program generates a fatal error when it attempts an operation that violates the rules of the operating system. When this happens, Unix takes a snapshot of the program copy in memory and dumps that data into a core file. The core file debugging tools, such as dbx and gdb, are used to read the information in the core file and provide a stack trace of the processing at the time the error occurred.

The first step in collecting a stack trace is to identify the process that produced the core by running the following command:

```
cstat -n <core_file_name>
```

This command outputs the name of the program that failed. The core file name is a required parameter when running the `cstat` command.

`cstat`, `dbx`, and `gdb` send their output to the screen. Therefore, a script shell should first be started to collect the output. For example:

```
script core_output.txt
```

This command redirects the screen output to the `core_output.txt` file.

Use `Ctrl+d` to stop the script shell once all of the debugging output has been displayed.

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
   # 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 process id number from the pidof output.
   ```

Code

```cpp
#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";
        cout<<"Give x and y as input\n";
        cin>>x>>y;
        xDivY = divide(x,y);
        cout << xDivY << endl;
        return 0;
}
// Takes two argument 'a' and 'b'
// Return 'a/b'
int divide(int a, int b){
        return a / b;
}
```

1. Process start running

```
troy@troy-inspiron    ×    troy@troy-inspiron55
[troy@troy-inspiron5593 Desktop]$ g++ div.cpp -o div -g
[troy@troy-inspiron5593 Desktop]$ ./div
2/nGive x and y as input
```

2. Finding Process id using **pidof div** program runnning.

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

4. 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: w.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 w.c -o w"

2. open the program with gdb: "gdb w"

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

5. add a watchpoint to i: "watch i"


```
troy@troy-inspiron5593:~/Desktop                    +    ...    —    □    X

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from w...
(gdb) b main
Breakpoint 1 at 0x1141: file w.c, line 3.
(gdb) r
Starting program: /home/troy/Desktop/w

This GDB supports auto-downloading debuginfo from the following URLs:
https://debuginfod.archlinux.org
Enable debuginfod for this session? (y or [n])
Debuginfod has been disabled.
To make this setting permanent, add 'set debuginfod enabled off' to .g
dbinit.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/usr/lib/libthread_db.so.1".

Breakpoint 1, main () at w.c:3
3               for(int i=0;i<5;i++){
(gdb) |
```

6. Now keep running the program with "c", we observe that the program stops everytime i changes


```
troy@troy-inspiron5593:~/Desktop                    +    ...    —    □    X

3               for(int i=0;i<5;i++){
(gdb) watch i
Hardware watchpoint 2: i
(gdb) c
Continuing.

Hardware watchpoint 2: i

Old value = 32767
New value = 0
main () at w.c:3
3               for(int i=0;i<5;i++){
(gdb) c
Continuing.

Hardware watchpoint 2: i

Old value = 0
New value = 1
main () at w.c:6
6                       printf("%d\n",i);
(gdb) c
Continuing
```

```
Hardware watchpoint 2: i

Old value = 1
New value = 2
0x0000555555555175 in main () at w.c:3
3                for(int i=0;i<5;i++){
(gdb) c
Continuing.

Hardware watchpoint 2: i

Old value = 2
New value = 3
main () at w.c:6
6                        printf("%d\n",i);
(gdb) c
Continuing.
3

Hardware watchpoint 2: i

Old value = 3
New value = 4
0x0000555555555175 in main () at w.c:3
3                for(int i=0;i<5;i++){
(gdb) c
Continuing.

Hardware watchpoint 2: i

Old value = 4
New value = 5
main () at w.c:6
6                        printf("%d\n",i);
(gdb) c
Continuing.
5

Hardware watchpoint 2: i

Old value = 5
New value = 6
0x0000555555555175 in main () at w.c:3
3                for(int i=0;i<5;i++){
(gdb) c
Continuing.
```