Debuggers

Debuggers

- A debugger is a software tool or program used by developers and programmers to identify and resolve issues, errors, or bugs in computer programs or code.
- A debugger allows for the step-by-step execution of a program, providing visibility into the program's state and behavior at each step
- Following are few debuggers that are widely used:
  - GDB
  - LLDB
  - TotalView Debugger
  - Arm DDT / Allinea DDT
  - Visual Studio Debugger

Debuggers

- A debugger is a software tool or program used by developers and programmers to identify and resolve issues, errors, or bugs in computer programs or code.
- A debugger allows for the step-by-step execution of a program, providing visibility into the program's state and behavior at each step
- Following are few debuggers that are widely used:
  - GDB
  - LLDB
  - TotalView Debugger
  - Arm DDT / Allinea DDT
  - Visual Studio Debugger

Our focus will remain here

GDB (GNU Debugger)

The GNU Debugger, commonly known as GDB, is a widely used and powerful debugger for various programming languages, including C, C++, Ada, Objective-C, and more.

It is part of the GNU Project and is available on multiple platforms, such as Linux, macOS, and Windows.

GDB (GNU Debugger)

The GNU Debugger, commonly known as GDB, is a widely used and powerful debugger for various programming languages, including C, C++, Ada, Objective-C, and more.

It is part of the GNU Project and is available on multiple platforms, such as Linux, macOS, and Windows.

Features supported by GDB:

- Setting Breakpoints & Watchpoints
- Stepping through instructions
- Variable Inspection
- Stack trace inspection
- Core dump analysis

More Info here: https://www.sourceware.org/gdb/

Compilation

- A program that requires debugging needs to be compiled with the command line option ***-g***. This enables the addition of debugging symbols to the code which can be used by *gdb*.

```
# gcc        -g      my_program.c
```

Compilation

- A program that requires debugging needs to be compiled with the command line option ***-g***.
  This enables the addition of debugging symbols to the code which can be used by *gdb*.

```
# gcc        -g       my_program.c
```

- When debugging with optimization options, the compiler may delete or move code or variables.
  The debugger may not be able to reference such variables or set correct breakpoints.
  Hence, it is a always better to use ***-g*** in conjunction with ***-O0*** for compilation.
  [-O0 here disables optimization of code]

```
# gcc        -g       -O0          my_program.c
```

Starting and Exiting GDB

- To start gdb, you can simply issue the below command from your linux terminal.

```
# gdb                my_executable_program
(gdb)
```

There actually exists multiple ways in which you can start gdb depending upon the user requirements. The above mentioned way is one of the most common and simplest way.

Starting and Exiting GDB

- To start gdb, you can simply issue the below command from your linux terminal.

```
# gdb                my_executable_program
(gdb)
```

  There actually exists multiple ways in which you can start gdb depending upon the user requirements. The above mentioned way is one of the most common and simplest way.

- To exit from gdb prompt, you can use the below command.

```
(gdb) quit
#
```

GDB commands

- **help** [command]

  To obtain the list of features and operations supported by gdb

  Information for specific operations can also be explained by specifying the respective command

```
(gdb) help
(gdb) help break
```

GDB commands

- **help** [command]

  To obtain the list of features and operations supported by gdb
  Information for specific operations can also be explained by specifying the respective command

```
(gdb) help
(gdb) help break
```

- **file** *executable*

  This specifies the executable program which is intended to be debugged
  Program to be compiled with -g flag, otherwise, gdb won't be able to obtain clear information.

```
(gdb) file executable
```

GDB commands

- **run** [arg1 arg2 … argn]

  Executes the loaded executable program with the provided arguments.

```
(gdb) run  3  4  “abc”
```

GDB commands

- **run** [arg1 arg2 … argn]

  Executes the loaded executable program with the provided arguments.

```
(gdb) run  3  4  “abc”
```

- **info** [*about]*

  The *info* command lists information about the argument (about).
  Info. can be obtained for frames, stacks, registers, breakpoints, watchpoints and functions.

```
(gdb) info break
```

GDB commands

- **list** [linenumber]

  Prints out lines form the source code around the *linenumber*.
  Function names can also be used to display lines from the beginning of the function.

```
(gdb) list
```

GDB commands

- **list** [linenumber]

  Prints out lines form the source code around the *linenumber*.
  Function names can also be used to display lines from the beginning of the function.

```
(gdb) list
```

- **print** *expression*

  The print command is used to print the value of the provided *expression* which could be a variable name or a complex expression.

```
(gdb) print   variable_name
```

GDB commands

- **break** [[filename:]linenumber]

  Used to set a breakpoint in the execution of the program. Function names can also be used. The program execution will stop before the code at given line number is executed.

```
(gdb) break 25
```

GDB commands

- **break** [[filename:]linenumber]

  Used to set a breakpoint in the execution of the program. Function names can also be used. The program execution will stop before the code at given line number is executed.

```
(gdb) break 25
```

- **condition** *breakpoint_number condition*

  Condition command updates the breakpoint indicated by the given line number such that the execution stops only when the condition is true.

```
(gdb) condition    2    x >= 35
```

GDB commands

- **delete** *breakpoint_number*

  Delete removes the indicated breakpoint that has been previously set.
  The list of breakpoints / watchpoints can be obtained by using the *info* command.

```
(gdb) delete 25
```

GDB commands

- **delete** *breakpoint_number*

  Delete removes the indicated breakpoint that has been previously set.
  The list of breakpoints / watchpoints can be obtained by using the *info* command.

```
(gdb) delete 25
```

- **kill**

  Terminates the current debugging session

```
(gdb) kill
```

GDB commands

- **step**

  Steps through a single line of code. This command does step *into* function calls.

```
(gdb) step
```

GDB commands

- **step**

  Steps through a single line of code. This command does step *into* function calls.

```
(gdb) step
```

- **next**

  Steps through a single line of code. This command steps *over* function call.s

```
(gdb) next
```

GDB commands

- **continue**

  Resumes the execution of a stopped program.
  The program will be halted during the next breakpoint

```
(gdb) continue
```

GDB commands

- **continue**

  Resumes the execution of a stopped program.
  The program will be halted during the next breakpoint

```
(gdb) continue
```

- **backtrace**

  Backtrace command prints the stack trace of the program in execution.

```
(gdb) backtrace
```

Workout 1

A) Debug the listed program using the techniques learnt above.

B) Note which gdb commands are being used frequently.

C) List the commands which you find useful in the listed program.

D) Note difference between logical and programmatic errors. (Hint)

```
#include <stdio.h>

int main (int argc, char *argv[]) {

        int arr[10] = {10, 20, 30, 40, 50, 60, 70};
        int arr_len = 7;
        float sum = 0;

        for (int i = arr_len - 1; i >= 0; ++i) {
                sum += arr[i] / (i * 10);
        }

        printf("Sum = %f.\n", sum);
        return 0;
}
```

Workout 2

A) Debug the listed program using the techniques learnt above.

B) Note which gdb commands are being used frequently.

C) List the commands which you find useful in the listed program.

D) Note difference between logical and programmatic errors. (Hint)

```
#include <stdio.h>

void set_value(int v, int* ptr1, int* ptr2) {
        ptr1 = &v;
        *ptr2 = *ptr1;
}

int main (int argc, char *argv[]) {
        int* p1 = NULL;
        int* p2 = NULL;
        int value = 10;

        set_value(value, p1, p2);

        printf("Original value = %d.\n", value);
        printf("P1 ref.  value = %d.\n", *p1);
        printf("P2 ref.  value = %d.\n", *p2);

        return 0;
}
```

“Debugging is like being the detective in a crime movie where you are also the murderer.”

Questions?

Workout 3

A) Debug the listed program using the techniques learnt above.

B) Note which gdb commands are being used frequently.

C) List the commands which you find useful in the listed program.

D) Note difference between logical and programmatic errors. (Hint)

```
#include <stdio.h>

int main (int argc, char *argv[]) {

        int arr[5] = {1, 2, 3, 4, 5};
        int normal = 0;

        #pragma omp parallel for shared(arr) num_threads(5)
        for (int i = 4; i >= 0; --i) {
                arr[i] = arr[i] / i+1;
        }

        for (int i = 0; i < 5; ++i) {
                normal += arr[i];
        }

        printf("Normal value = %d.\n", normal);

        return 0;
}
```

Thanks!

Contact:

Vineet More
vineet.more.bfab@gmail.com
[Make sure to use HPCAP23 as the subject line for your queries]

LinkedIn Handle:
https://www.linkedin.com/in/vineet-more-c-programmer/