GDB – GNU Debugger

Table of Contents

2. Getting In and Out of GDB

2.1. Invoking GDB

```
gdb program [core] [--silent] [-help]
```

Attach GDB to a process:

```
gdb -p processId
```

Add directory to the path to search for source and script files:

```
-d directoryName
```

2.2. Quitting GDB

```
quit [expression]
```

If *expression* is given, its result is used as the error code.

An interrupt (Ctrl-C) does not exit GDB, but rather terminates the action of any GDB command that is in progress.

2.3. Shell Commands

Executing shell commands from GDB:

```
shell command
! command   // here the environment variable SHELL determines which shell to run
!command   // GDB uses the default shell
```

2.4. Logging Output

Show current values of logging settings:

```
show logging
```

Save the output of GDB commands to a file.

```
set logging [on | off]
```

Change name of log file. Default is gdb.txt

```
set logging file fileName
```

3. GDB Commands

We can abberivate GDB commands to the 1st few letters of the command name if that abbreviation is unambiguous.

3.1. Command Syntax

A *blank line* as input to GDB means to repeat the previous command. Certain commands will not repeat this way, these are commands whose unintentional repetition might cause trouble and which you are unlikely to want to repeat.

Any text from a **#** to the end of the line is a command. This is useful mainly in *command files*.

3.2. Command Settings

Many commands change their behavious according to command specific variables or settings. These settings can be changes with the set subcommands.
Ex, to change the limit of array elements to print

```
set print elements number-of-elements
```

3.3. Command Completion

On auto completion, if the number of possilbe completions is large, GDB will print as much of the list as it has collected, as well as a message indicating that the list may be truncated.

Show the current max limit:

```
show max-completions
```

Set completion limit:

```
set max-completions limit
```

Setting a value of *0* disables tab-completion.

3.5. Getting Help

We can use **help** command to display a short list of named classes of commands.
We can use **info** and **show** commands.

info is used to *inquire about the state of the program*. We can see the arguments passed to a function, list the registers currently in use, list of breakpoints. We can get complete list of sub-commands with help info.

show is used to *describe the state of the GDB* itself.

4. Running Programs Under GDB

4.1. Compiling for Debugging

To effectively debug a program we need to generate debugging information when compiling it. This debugging info is stored in the object file, it describes the type of each variable or function and the correspondence between source line numbers and addresses in executable code.
Using **`-g`** option when compining the compiler.

4.2. Starting your Program

Using run command to start program under GDB. We must 1st specify the program name with an argument to GDB, or by using the **`file`** or **`exec-file`** command.

`run` creates an inferior process and makes that process run the program. With **`run`** the program begins to execute immediately. The **`start`** command does the equivalent of setting a temporary breakpoint at the beginning of the main procedure and then invoking the **`run`** command.

4.3. Program's Arguments

Arguments to the program can be specified by the arguments of the **`run`** command. **`run`** with no arguments uses the same arguments used by the previous **`run`**, or those set by the **`set args`** command.

Specify the arguments to be used the next time program is run. If **`set args`** has no arguments, **`run`** executes program with no arguments.

```
set args
```

Show the arguments to give to the program

```
show args
```

4.4. Program's Environment

Environment consists of set of environment variables and their values. Environment variables conventially record such things as name, home directory, terminal type, search path for programs to run.

Add directory to the front of the PATH environment variable (the search path for executables) that will be passed to the program.

```
path directory
```

Display the list of search paths for executables:

```
show paths
```

Print the value of environment variable varname.

```
show environment [ varname ]
```

Set environment variable varname to value.

```
set environment varname [ =value ]
```

Remove variable varname from the environment.

```
unset environment varname
```

4.5. Program's working directory

Each time we start a program with **run** command, the inferior will be initialized with the current working directory specified by the **set cwd** command. If no directory is specified, the inferior will inherit GDB's current directory.

```
set cwd [ directory ]
```

Show the inferiors working directory

```
show cwd
```

Set GDB's working directory to directory.

```
Cd [ directory ]
```

Print GDB's working directory

```
pwd
```

4.6. Programs Input and Output

By default, the running program uses the GDB terminal for IO. We can redirect IO using shell redirection with **run** command.

```
run > outfile
```

Another way to specify IO is with **tty** command. It accepts a file name as argument, and causes this file to be the default for the future **run** commands.

```
tty /dev/ttyb
```

An explicit redirection in **run** overrides the **tty** command's effect on the IO device.

To show the current tty for the program being debugged.

```
show inferior-tty
set inferior-tty [ tty ]
```

Omitting *tty* restores the default behaviour, which is to use the same terminal as GDB.

4.7. Debugging Already-running Process

```
attach process-id
```

This command attaches to a running process. You must have permission to send the process a signal. The first thing GDB does after arranging to debug the specified process is to stop it.

When finished debugging, we can use the **detach** command to release it from GDB. Detaching the process continues its execution.

attach and **debug** does not repeat if we press **RET** again after executing the command.

4.8. Killing the Child Process

kill is used to kill the child process running under GDB.

4.9. Debugging Multiple Inferiors and Programs

GDB represents the state of each program execution with an object called an ***inferior***. Print a list of all inferiors currently being managed by GDB.

```
info inferiors
```

To switch focus between inferiors:

```
inferior infNo
```

We can get multiple executables into the debugging session:

```
add-inferior [ -copies n ] [ -exec executable ]
clone-inferior [ -copies n ] [ infNo ]
remove-inferior infNo
```

It is not possible to remove an currently running inferior with this command. Use the **`kill`** or **`detach`** command.

```
detach inferior infNo
kill inferior infNo
```

4.12. Setting Checkpoints

On certain OS, GDB can save snapshots of program's state, called ***checkpoint***, and come back to it later. Returning to a checkpoint effectively undoes everything that has happened in the program since the checkpoint was saved.

Save a snapshot of the program's current execution state.

```
checkpoint
```

List all the checkpoints

```
info checkpoints
```

Restore to the checkpoint

```
restart checkpointId
```

Delete checkpoint

```
delete checkpoint checkpointId
```

Each checkpoint will have a unique process id, and each will be different from the program's orignal *pid*.

5. Stopping and Continuing

Display information about the status of program, whether it is running or not, what process it is, and why it stopped.

```
info program
```

5.1. Breakpoints, Watchpoints, and Catchpoints

A **breakpoint** makes program stop whenever a certain point in the program is reached. We can add conditions to control in finer detail when program stops.

Watchpoint is a special breakpoint that stops program when the value of an expression changes. Sometimes called **data breakpoints**.

Catchpoint is a special breakpoint that stops the program when a certain kind of event occurs.

5.1.1. Setting Breakpoints

Set a breakpoint at the given *location*, which can specify a function name, a line number, or an address of an instruction. The breakpoint will stop before it executes any of the code in specified location. If it is called without arguments, it sets a breakpoint at the next instruction to be executed in the selected stack frame.

```
break [ location ]
```

Set a breakpoint with condition *cond*; program stop only if the *cond* is nonzero; true.

```
break ... if cond
```

Set a breakpoint enabled for only one stop. Breakpoint is automatically deleted after the first time the program stops.

```
tbreak args
```

Set breakpoint on all functions matching the regular expression *regex*.

```
rbreak regex
```

Print a table of all breakpoint.

```
info breakpoints [ list... ]
```

5.1.2. Setting Watchpoints

Watchpoints are used to stop execution whenever the value of an *expression* changes. Depending on the system, watchpoints may be implemented in *software* or *hardware*. GDB does software watchpointing by single-stepping the program and testing the variable's value each time, which is very slow than normal execution.

```
watch expression_or_variable
```

The `-location` argument tells GDB to instead watch the memory referred to by *expr*.

Print a table of watchpoints.

```
info watchpoints
```

5.1.6. Break Conditions

A breakpoint with a condition evaluates the expression each time the program reaches it, and program stops only if the condition is true. Break conditions can be specified when a breakpoint is set, by using the **`if`** argument with **`break`** command. They can also be changed at any time with the **`condition`** command.

```
break 9 if (i > 10)
condition 56 if( i > 0)
```

5.1.7. Breakpoint Command Lists

We can give any breakpoint a series of commands to execute when your program stops due to that breakpoint.

```
break 9 if( i > 0)
commands
print "Positive"
end
```

5.1.9. Saving Breakpoints

```
save breakpoints [filename]
breakpoint definitions can be loaded using the source command.
```

5.2. Continuing and Stepping

`continue` is used to resuming program execution until program completes normally. To resume execution at aribitary location, we can use **`return`** to go back to the calling function, or **`jump`** to go to an arbitary location in the program.

`step` is used to continue running program until it reaches a different source line.
`next` is used to continue to the next source line in the current stack frame.
`finish` is used to continue running until just after function in the selected stack frame returns. Print the value returned.
`until` is used to continue running until a source line past the current line, in the current stack frame, is reached. This is used to avoid single stepping through a loop more than once.

5.3. Skipping Over Functions and Files

`skip` *`function_name`*
This instructs GDB to never step into the function.

5.4. Signals

Signal is an asynchronous event that can happen in a program. The OS defines the possible kinds of signals, and gives each kind a name and a number. In UNIX, *SIGINT* is a signal for interrupt character, *SIGSEGV* is the signal for memory errors.

6. Running Program Backward

GDB's built-in record and replay has several limitations, e.g. no support for ***AVX instructions***. GDB has built-in support for reverse execution, activated with the **`record`** command.

`reverse-continue`
`reverse-next`
`reverse-step`

8. Examining the Stack

8.1. Stack Frames

Stack frame / *frame* is the data associated with one call to one funciton. The frame contains the argument given to the function, the function's local variable, & the address at which the function s being executing.

8.2. Backtraces

Backtrace is a summary of how program got where it is.
To print a backtrace of the entire stack

```
backtrace [-full]
```

8.3. Selecting a Frame

```
frame f
```

Information about frame

```
info frame
```

Move ***n*** frames up in the stack, *n* defaults to 1

```
up [n]
```

Move ***n*** frames down in the stack, *n* defaults to 1

```
down [n]
```

9. Examining Source Files

9.1. Printing Source Lines

`list` command is used to print 10 lines from source.

```
list [linenum | function]
```

9.2. Specifying a Location

Location can be specified using 3 differnet formats:

- Linespec
- Explicit
- Address

9.3. Editing Source Files

Source files can be editted using the **`edit`** command.

9.4. Searching through Source Files

```
forward-search regexp
reverse-search regexp
```

`forward-search` *`regexp`* check each line, starting with the one following the last line listed, for a match for *regexp*.

10. Examining Data

Data in a program can be examined with **`print`** command. The **`print`** command supports a number of options that allow overriding relevant global print settings as set by **`set print`** subcommands. Options like printing *address*, printing *array*, printing *array indexes*, printing *structured data.*

A more low-level way of examining data is with the **`x`** command. It examines data in memory at a specfied address and prints it in a specified format.

10.1. Expressions

`print` & many other GDB commands accept an expression & compute its value. GDB supports array constants in expression input by the user. The syntax is *{element, element,...}*.

`@` is binary operator for treating parts of memory as arrays.
`::` allows you to specify a variable in terms of the file / function where it is defined.
`{type} addr` refers to an object of type type stored at address addr in memory.

10.2. Ambiguous Expressions

When an ambuiguity that needs to be resolved is detected, the debugger has the *capability to display a menu of numbered choices* for each possibility, & then waits for the selection with the prompt '>'.

By default, if the command with which the expression is used allows more than one choice, then GDB automatically selects all possible choices.

10.3. Program Variables

Variables in expressions are understood in the selected stack frame; they must be either:
- global
- visible according to the scope rules of the programming

10.4. Artificial Arrays

We can reffer to a contiguous span of memory as an artifical array, using the binary operator **`@`**.

```
print *array@len
```

10.5. Output Formats

By default, GDB prints a value according to its data type. The simplest use of *output formats* is to say how to print a value already computed. This is done by starting the arguments of the **`print`** command with a *slash* and a *format letter*.

```
print num
66
print/x num
0x42
```

10.6. Examining Memory

We can use the command **`x`** to examine memory in any of several formats, independently of your program's data types.

`x/nfu addr`

n, *f*, & *u* are optional parameters that specify the *repeat count*, the *display format*, & the *unit size*.
Unit size can be b (bytes), h (halfword - 2 bytes), w (word – 4 bytes), g (giant word – 8 bytes).

10.7. Automatic Display

The expressions in automatic display list is executed each time the program stops.

`display expr`

10.8. Print Settings

Print index of each element when displaying arrays:

`set print array-indexes [on | off]`

show null characters in string array

`set print null-stop [on | off]`

Show content of objects:

`set print object [on | off]`

Show static members of class when printing object:

`set print static-members [on | off]`

Pretty print C++ virtual function tables:

`set print vtbl [on | off]`

10.9. Pretty Printing

GDB provides a mechanism to allow *pretty printing of values using python code*. It greatly simplifies the display of complex objects.

10.9.1. Pretty Printer Introduction

When GDB prints a value, it first sees if there is a pretty-printer ***registered*** for the value. If there is then GDB invokes the pretty-printer to print the value. The **`info pretty-printer`** command will list all the installed pretty-printers with their names.

10.9.3. Pretty-Printer Commands

pretty-printers can be enabled / disabled:

```
enable pretty-printer [object-regexp [name-regexp]]
disable pretty-printer [object-regexp [name-regexp]]
```

10.10. Value History

Values printed by the **`print`** command are saved in the GDB value history. This allows us to refer to them in other expressions. Values are kept until the symbol table is re-read or discarded.

$ referes to the last printed value.

Print last printed values *(default **n** is 10)* :
`show values [n]`

10.13. Registers

We can refer to machine register contents, in expressions, as variables with names starting with **$**. Registers names can be different on different machines, **`info registers`** cal list names.

10.16. OS Auxiliary Information

GDB provides interfaces to useful OS facilities that can help you debug your program. Some OS supply an auxilary vector to programs at startup. This is akin to the arguments and environments that you specify for a program, but contain a system-dependent variety of binary values that tell system libraries important details about the hardware, OS and process.

10.18. Copy Between Memory and a File

We can ues the commands **`dump`**, **`append`**, and **`restore`** to copy data between target memory and a file. The **`dump`** and **`append`** commands write data to a file, and the **`restore`** command reads data from a file back into the inferior's memory.

10.19. Producing Core Files

A core file records the memory image of a runing process and its process status (register values etc). Its primary use is post-mortem debugging of a program that crashed when it ran outside a debugger. Core file can be generated by

`generate-core-file [file]`
`gcore [file]`

10.22. Search Memory

Memory can be searched for a particular sequence of bytes with the **`find`** command.

11. Debugging Optimized Code

Almost all compilers support optimization. With optimization disabled, the compiler generates assembly code that corresponds directly to your source code, in a simplistic way. As the compiler applies more powerful optimizations, the generated assembly code diverges from your original source code. With help from debugging information generated by the compiler, GDB can map from the running program back to constructs from your original source.

13. Tracepoints

If the program's correctness depends on its real-time behavior, delays introduced by a debugger might cause the program to change its behavior drastically, or perhaps fail, even when the code itself is correct. *It is useful to be able to observe the program's behavior without interrupting it*.

Using GDB's **`trace`** and **`collect`** commands, you can specify locations in the program, called ***tracepoints***, and arbitrary expressions to evaluate when those tracepoints are reached. Later, using the **`tfind`** command, you can examine the values those expressions had when the program hit the tracepoints.

The expressions may also denote objects in memory - structures or arrays, for example - whose values GDB should record; while visiting a particular tracepoint, you may inspect those objects as if they were in memory at that moment.

15. Using GDB with Different Languages

15.2. Displaying the Language

`show language`
Display the current working language. This is the language you can use with commands such as **`print`** to build and compute expressions that may involve variables in your program.

15.4. Supported Languages

GDB supports
- C
- C++
- D
- Go
- Objective-C
- Fortran
- QpenCL C
- Pascal
- Rust
- assembly
- Modula-2
- Ada

Some GDB features may be used in expressions regardless of the language you use.

16. Examining the Symbol Table

The commands described in this chapter allow you to inquire about the symbols *(names of variables, functions and types)* defined in your program. This information is inherent in the text of your program and does not change as your program executes.

```
set print type methods
set print type methods on
set print type methods off
```

Normally, when GDB prints a class, it displays any methods declared in that class. You can control this behavior either by passing the appropriate flag to **ptype**, or using **set print type methods**. Specifying ***on*** will cause GDB to display the methods; this is the default. Specifying ***off*** will cause GDB to omit the methods.

`info address` *symbol*
Describe where the data for *symbol* is stored.

`info symbol` *addr*
Print the name of a symbol which is stored at the address ***addr***. If no symbol is stored exactly at ***addr***, GDB prints the nearest symbol and an offset from it

```
(gdb) info symbol 0x54320
_initialize_vx + 396 in section .text
```

`whatis` *[/flags] [arg]*
Print the data type of *arg*, which can be either an expression or a name of a data type. With no argument, print the data type of *$*, the last value in the value history. If *arg* is an expression, it is not actually evaluated, and any side-effecting operations inside it do not take place.

`ptype` *[/flags] [arg]*
ptype accepts the same arguments as **whatis**, but prints a detailed description of the type, instead of just the name of the type.

Contrary to **whatis**, **ptype** always unrolls any *typedefs* in its argument declaration, whether the argument is a variable, expression, or a data type. This means that ptype of a variable or an expression will not print literally its type as present in the source code - use **whatis** for that.

`info scope location`
List all the variables local to a particular scope. This command accepts a location argument - a function name, a source line, or an address preceded by a '*', and prints all the variables local to the scope defined by that location.

`info source`

```
(gdb) info source
Current source file is Cpp_Prog.cpp
Compilation directory is /home/user/Directory
Located in /home/user/Directory/Cpp_Prog.cpp
Contains 52 lines.
Source language is c++.
Producer is GNU C++14 7.5.0 -mtune=generic -march=x86-64 -ggdb -fstack-protector-strong.
Compiled with DWARF 2 debugging format.
Does not include preprocessor macro info.
```

Show information about the current source file - that is, the source file for the function containing the current point of execution:
- the name of the source file, and the directory containing it
- the directory it was compiled in
- its length, in lines

- which programming language it is written in
- if the debug information provides it, the program that compiled the file
- whether the executable includes debugging information for that file, and if so, what format the information is in (e.g., STABS, Dwarf 2, etc.)
- whether the debugging information includes information about preprocessor macros

```
maint info symtabs [ regexp ]
maint info psymtabs [ regexp ]
```

List the **struct symtab** or **struct partial_symtab** structures whose names match *regexp*. If regexp is not given, list them all. The output includes expressions which you can copy into a GDB debugging this one to examine a particular structure in more detail.

17. Altering Execution

Once you think you have found an error in your program, you might want to find out for certain whether correcting the apparent error would lead to correct results in the rest of the run. You can find the answer by experiment, using the GDB features for altering execution of the program.

17.1 Assignment to Variables

To alter the value of a variable, evaluate an assignment expression.

```
print x=4
```

stores the value *4* into the variable ***x***, and then prints the value of the assignment expression.

If you are not interested in seeing the value of the assignment, use the **`set`** command instead of the **`print`** command.

```
(gdb) set var width=47
```

17.2 Continuing at a Different Address

Ordinarily, when you continue your program, you do so at the place where it stopped, with the **`continue`** command. You can instead continue at an address of your own choosing, with the following commands:

```
jump location
j location
```

Resume execution at *location*. Execution stops again immediately if there is a breakpoint there.

On many systems, you can get much the same effect as the **`jump`** by storing a new value into the register **`$pc`**. The difference is that this does not start your program running; it only changes the address of where it will run when you continue. For example,

```
set $pc = 0x485
```

makes the next continue command or stepping command execute at address 0x485, rather than at the address where your program stopped.

The most common occasion to use the jump command is to back up - perhaps with more breakpoints set - over a portion of a program that has already executed, in order to examine its execution in more detail.

17.3 Giving your Program a Signal

```
signal signal
```

Resume execution where your program is stopped, but immediately give it the signal *`signal`*. The signal can be the name or the number of a signal. For example, on many systems signal 2 and signal SIGINT are both ways of sending an interrupt signal.

Alternatively, if signal is zero, continue execution without giving a signal. This is useful when your program stopped on account of a signal and would ordinarily see the signal when resumed with the continue command; ‘signal 0’ causes it to resume without a signal.

```
queue-signal signal
```

Queue signal to be delivered immediately to the current thread when execution of the thread resumes. The signal can be the name or the number of a signal.

17.4 Returning from a Function

```
return
return expression
```

You can cancel execution of a function call with the **`return`** command. If you give an expression argument, its value is used as the function's return value.

When you use **`return`**, GDB discards the selected stack frame (and all frames within it). You can think of this as making the discarded frame return prematurely. If you wish to specify a value to be returned, give that value as the argument to **`return`**.

17.5 Calling Program Functions

```
print expr
```

Evaluate the expression *expr* and display the resulting value. The expression may include calls to functions in the program being debugged. call *expr*

You can use this variant of the **`print`** command if you want to execute a function from your program that does not return anything, but without cluttering the output with void returned values that GDB will otherwise print. If the result is not void, it is printed and saved in the value history.

It is possible for the function you call via the print or call command to generate a signal. What happens in that case is controlled by the **`set unwindonsignal`** command.

17.6 Patching Programs

By default, GDB opens the file containing your program's executable code read-only. This prevents accidental alterations to machine code; but it also prevents you from intentionally patching your program's binary.

If you'd like to be able to patch the binary, you can specify that explicitly with the set write command. For example, you might want to turn on internal debugging flags, or even to make emergency repairs.

```
set write [off | on]
```

If you specify set write on, GDB opens executable and core files for both reading and writing; if you specify set write off (the default), GDB opens them read-only.

If you have already loaded a file, you must load it again (using the exec-file or core-file command) after changing set write, for your new setting to take effect.

18 GDB Files

18.1 Commands to Specify Files

You may want to specify executable and core dump file names. The usual way to do this is at start-up time, using the arguments to GDB's start-up commands. Occasionally it is necessary to change to a different file during a GDB session. Or you may run GDB and forget to specify a file you want to use.

`file` *`filename`*
Use *filename* as the program to be debugged. It is read for its symbols and for the contents of pure memory. It is also the program executed when you use the **`run`** command.

You can load unlinked object ***`.o`*** files into GDB using the **`file`** command. You will not be able to "run" an object file, but you can disassemble functions and inspect variables.

`file`
`file` with no argument makes GDB discard any information it has on both executable file and the symbol table.

`exec-file [ filename ]`
Specify that the program to be run is found in filename.

`core-file [filename]`
`core`
Specify the whereabouts of a core dump file to be used as the "contents of memory". Traditionally, core files contain only some parts of the address space of the process that generated them; GDB can access the executable file itself for other parts. core-file with no argument specifies that no core file is to be used.

`info files`
`info target`
info files and info target are synonymous; both print the current target, including the names of the executable and core dump files currently in use by GDB, and the files from which symbols were loaded. The command help target lists all possible targets rather than current ones.

18.3 Debugging Information in Separate Files

GDB allows you to put a program's debugging information in a file separate from the executable itself, in a way that allows GDB to find and load the debugging information automatically.

GDB supports two ways of specifying the separate debug info file:
> The executable contains a debug link that specifies the name of the separate debug info file. The separate debug file's name is usually executable.debug, where executable is the name of the corresponding executable file without leading directories.

> The executable contains a build ID, a unique bit string that is also present in the corresponding debug info file.

19 Specifying a Debugging Target

A target is the execution environment occupied by your program. Often, GDB runs in the same host environment as your program; in that case, the debugging target is specified as a side effect when you use the **`file`** or **`core`** commands. When you need more flexibility—for example, running GDB on a physically separate host, or controlling a standalone system over a serial port or a realtime system over a TCP/IP connection—you can use the target command to specify one of the target types configured for GDB.

It is possible to build GDB for several different target architectures. When GDB is built like that, you can choose one of the available architectures with the set architecture command.

```
set architecture arch
```

This command sets the current target architecture to arch. The value of arch can be "auto", in addition to one of the supported architectures.

```
set processor
processor
```

These are alias commands for, respectively, set architecture and show architecture.

19.3 Choosing Target Byte Order

Some types of processors, such as the MIPS, PowerPC, and Renesas SH, offer the ability to run either big-endian or little-endian byte orders. Usually the executable or symbol will include a bit to designate the endian-ness, and you will not need to worry about which to use. However, you may still find it useful to adjust GDB's idea of processor endian-ness manually.

```
set endian big
set endian little
set endian auto
show endian
```

25. GDB Text User Interface

The GDB Text User Interface (TUI) is a terminal interface which uses the curses library to show the source file, the assembly output, the program registers and GDB commands in separate text windows. The TUI mode is supported only on platforms where a suitable version of the curses library is available.

The TUI mode is enabled by default when you invoke GDB as **`gdb -tui`**.

25.1. TUI Overview

In TUI mode, GDB can display several text windows:

- command : This window is the GDB command window with the GDB prompt and the GDB output. The GDB input is still managed using readline.
- source : The source window shows the source file of the program. The current line and active breakpoints are displayed in this window.
- assembly : The assembly window shows the disassembly output of the program.
- register : This window shows the processor registers. Registers are highlighted when their values change.

The source and assembly windows show the current program position by highlighting the current line and marking it with a '>' marker.

The source, assembly and register windows are updated when the current thread changes, when the frame changes, or when the program counter changes.

These windows are not all visible at the same time. The command window is always visible. The others can be arranged in several layouts:
- source only,
- assembly only,
- source and assembly,
- source and registers, or
- assembly and registers.

These are the standard layouts, but other layouts can be defined.

25.2. TUI Key Bindings

The TUI installs several key bindings in the readline keymaps. The following key bindings are installed for both TUI mode and the GDB standard mode.

`Ctrl + x A`
Enter or leave the TUI mode. When leaving the TUI mode.

`Ctrl + x 1`
Use a TUI layout with only one window. The layout will either be 'source' or 'assembly'. When the TUI mode is not active, it will switch to the TUI mode.

`Ctrl + x 2`
Use a TUI layout with at least two windows. When the current layout already has two windows, the next layout with two windows is used.

`Ctrl + x o`
Change the active window. The TUI associates several key bindings with the active window. This command gives the focus to the next TUI window.

`Ctrl + x s`

Switch in and out of the TUI SingleKey mode that binds single keys to GDB commands.

25.4 TUI-specific Commands

The TUI has specific commands to control the text windows. These commands are always available, even when GDB is not in the TUI mode.

`tui [enable | disable]`
Activate / Disable TUI mode.

`info win`
List and give the size of all displayed windows.

`tui new-layout name window weight [window weight…]`
Create a new TUI layout. The new layout will be named name, and can be accessed using the **`layout`** command.

Each window parameter is either the name of a window to display, or a window description. The windows will be displayed from top to bottom in the order listed.

The names of the windows are the same as the ones given to the focus command; additional, the status window can be specified. Note that, because it is of fixed height, the weight assigned to the status window is of no importance. It is conventional to use '0' here.

A window description looks a bit like an invocation of **`tui new-layout`**, and is of the form **`{[-horizontal]window weight [window weight…]}`**.

Each weight is an integer. It is the weight of this window relative to all the other windows in the layout. These numbers are used to calculate how much of the screen is given to each window.

For example:
```
(gdb) tui new-layout example src 1 regs 1 status 0 cmd 1
```

Here, the new layout is called *'example'*. It shows the source and register windows, followed by the status window, and then finally the command window. The non-status windows all have the same weight, so the terminal will be split into three roughly equal sections.

Here is a more complex example, showing a horizontal layout:
```
(gdb) tui new-layout example {-horizontal src 1 asm 1} 2 status 0 cmd 1
```

This will result in side-by-side source and assembly windows; with the status and command window being beneath these, filling the entire width of the terminal. Because they have weight 2, the source and assembly windows will be twice the height of the command window.

`layout [next | prev | src | asm | split | regs | user_defined]`
Changes which TUI windows are displayed. The parameter controls which layout is shown. It can be either one of the built-in layout names, or the name of a layout defined by the user using tui new-layout.

`focus [next | prev | src | asm | regs | cmd]`
Changes which TUI window is currently active for scrolling.

```
winheight name +count
winheight name -count
```
Change the height of the window name by count lines. Positive counts increase the height, while negative counts decrease it. The name parameter can be one of src (the source window), cmd (the command window), asm (the disassembly window), or regs (the register display window).

25.5 TUI Configuration Variables

Several configuration variables control the appearance of TUI windows.

`set tui border-kind [space | ascii | acs]`
Select the border appearance for the source, assembly and register windows.

`set tui border-mode [normal | standout | reverse | half | half-standout | bold | bold-standout]`
`set tui active-border-mode [normal | standout | reverse | half | half-standout | bold | bold-standout]`
Select the display attributes for the borders of the inactive windows or the active window.

`set tui tab-width nchars`
Set the width of tab stops to be nchars characters. This setting affects the display of TAB characters in the source and assembly windows.