The GNU Debugger

A Partial Reference

Michael Marsh

The GNU Debugger

A Partial Reference

Michael Marsh

Contents

Introduction **1**

Running a Program **3**
Starting GDB . . . 3
Running a Program in GDB . . . 3
Synonyms . . . 4
Setting Breakpoints . . . 4
Synonyms . . . 4
Controlling Execution . . . 4
Synonyms . . . 5

Examining the Code **7**
Synonyms . . . 8

Examining the Stack **9**
Viewing the Backtrace . . . 9
Changing the Current Function View . . . 9
Inspecting Frame Information . . . 10
Synonyms . . . 10

Examining Variables and Memory **11**
Local Variables . . . 11
Synonyms . . . 12
Registers . . . 12
Synonyms . . . 13
Examining Memory . . . 13

Text User Interface **17**

Appendix: Quick Reference **21**

Introduction

The intended audience for this book is a student taking CMSC414 or a similar course who needs to use the GNU Debugger (GDB) to inspect a program's behavior. We make no attempt to thoroughly document GDB, only to provide a solid basis for a particular type of debugging. Noteworthy things we omit include:

- breakpoints beyond a simple `break` command
- tracepoints
- `find`
- `explore`

For more information, see http://www.gnu.org/software/gdb/documentation/ or the `help` command in GDB.

Running a Program

Starting GDB

First, to start GDB, there are three basic options. Let's say the program you are debugging is called `foo`:

1. `gdb foo`

   This loads the program `foo` into the debugger, with it ready to run.

2. `gdb foo core`

   This loads the program `foo` into the debugger, but gets the program's state from a *core* file. This is a snapshot of memory generated when the program crashes, and allows you to recreate the point at which the crash occurred without having to re-run the program. The file is called `core` because early computers had a drum core that served as its RAM. It used to be common, at least on Posix (and earlier UNIX) systems, for the system to dump a core file every time the program crashed. Now, this is typically disabled, but you can enable it by running:

   `ulimit -S -c unlimited`

   The `-S` sets the *soft* limit (which does not restrict further increases) for the resource. The `-c unlimited` sets the core file size to unlimited, which means it can be as large as it needs to in order to contain the full program memory, subject to available disk space. To reset this to the default behavior, run:

   `ulimit -S -c 0`

3. `gdb -p 1234`

   Instead of specifying the program to run, this attaches GDB to a running process with the process ID (PID) `1234`. This can be useful if you have a program that is taking longer than you expect or is otherwise behaving oddly.

Running a Program in GDB

To start a program, the simplest command is

`run`

This begins the program with no command-line arguments. If you want to provide arguments (since you cannot do this directly through the `gdb` command), you can provide them as arguments to `run`. Say we're running `ulimit` in GDB. We might start it with:

`run -S -c unlimited`

If you are trying to identify the cause of a segmentation fault (segfault, or SEGV), this might be the only thing you need to do from this chapter. Usually, you will want to do something from the following section before calling `run`.

Synonyms

- `run`, `r`

Setting Breakpoints

Unless you are examining a core file or simply running until a segmentation fault occurs, you will likely want to set a *breakpoint* before you call `run`. This is the point at which execution should stop and return control to the user (you).

When you specify the breakpoint, you will usually use a *linespec*, which specifies the line on which to break. The simplest forms of this are

- `break foo`: set a breakpoint at the first instruction of function `foo`
- `break 123`: set a breakpoint at line 123 of the current file

You can also specify the file in which to break, if you have a program with multiple files. For example:

- `break foo.c:123`: set a breakpoint at line 123 of the file `foo.c`

Another option is to break at a specific address. If you want to break at the instruction beginning 23 bytes from the beginning of the fuction `foo`, for example, you could call:

- `break *foo + 23`

There are many other options, including boolean conditionals, so that execution only breaks if a certain condition holds.

Synonyms

- `break`, `br`, `b`

Controlling Execution

Once execution reaches a breakpoint, you will frequently want to continue executing instructions while examining the program's status. When the source is available (that is, the code was compiled with debugging symbols and the source files can be found by GDB), you can execute line-by-line with either `next` or `step`. The difference is that `step` *descends* into functions, while `next` does not. As an example, consider the following code:

```
#include <stdio.h>

void g() {
   printf("in g()\n");
}

void f() {
   printf("in f()\n");
   g();
}

int main(int argc, char** argv) {
   printf("in main()\n");
   f();
   return 0;
}
```

Here is the behavior when we use `next`:

```
Breakpoint 1, f () at example.c:8
8       printf("in f()\n");
(gdb) next
in f()
9       g();
(gdb) next
in g()
```

Here is the behavior when we use `step`:

```
Breakpoint 1, f () at example.c:8
8       printf("in f()\n");
(gdb) step
0x00007f02e6687be8 in __GI__IO_puts (str=0x55787595c00b "in f()")
    at ./libio/ioputs.c:35
warning: 35 ./libio/ioputs.c: No such file or directory
(gdb) step
__strlen_evex () at ../sysdeps/x86_64/multiarch/strlen-evex-base.S:72
warning: 72 ../sysdeps/x86_64/multiarch/strlen-evex-base.S: No such file or directory
```

Both `next` and `step` take an optional argument N which steps N times instead of just once.

While `next` and `step` attempt to step by source code lines, we sometimes want to step by CPU instructions. For this, we can use `stepi`, which behaves like `step`, but by instructions, not source code lines.

If you want to resume execution until the program completes, produces an error, or reaches the next breakpoint, you can use the `continue` command.

Synonyms

- `next`, `n`
- `step`, `s`
- `stepi`, `si`
- `continue`, `c`, `fg`

Examining the Code

If you are debugging a program with symbols and source available, you can see the actual source code. The easiest way to do this is with the `list` command. Calling `list` multiple times will advance the section of the source code displayed.

You can also specify funtions, line numbers, and files to list. The format is essentially the same as that used for breakpoint linespecs.

You can also view the assembly instructions for the program, using `disassemble`. With no arguments, it will disassemble the current function. If we break in `f` and then run `disassemble`, we might see something like (for the AMD64 platform):

```
(gdb) disassemble
Dump of assembler code for function f:
   0x0000559f1a0c5163 <+0>: endbr64
   0x0000559f1a0c5167 <+4>: push   %rbp
   0x0000559f1a0c5168 <+5>: mov    %rsp,%rbp
=> 0x0000559f1a0c516b <+8>: lea    0xe99(%rip),%rax        # 0x559f1a0c600b
   0x0000559f1a0c5172 <+15>:    mov    %rax,%rdi
   0x0000559f1a0c5175 <+18>:    call   0x559f1a0c5050 <puts@plt>
   0x0000559f1a0c517a <+23>:    mov    $0x0,%eax
   0x0000559f1a0c517f <+28>:    call   0x559f1a0c5149 <g>
   0x0000559f1a0c5184 <+33>:    nop
   0x0000559f1a0c5185 <+34>:    pop    %rbp
   0x0000559f1a0c5186 <+35>:    ret
End of assembler dump.
```

The arrow indicates the next instruction that the CPU will execute. Let's take a closer look at this line:

```
=> 0x0000559f1a0c516b <+8>: lea    0xe99(%rip),%rax        # 0x559f1a0c600b
```

The first value we see is `0x0000559f1a0c516b`; this is the address of the instruction, and is in the code segment of the process. After this we see `<+8>`, which tells us that this instruction begins 8 bytes after the beginning of the function. Next is the actual instruction, as assembly, which is `lea 0xe99(%rip),%rax`. This is a "load effective address", which allows us to work with memory addresses (pointers). The comment at the end of the line tells us what the value will be: `%rip` at the time the instruction is executed plus `0xe99`. Since the instruction is at `0x0000559f1a0c516b`, once it is executing the instruction pointer will be advanced to `0x0000559f1a0c5172`. If we add `0xe99` to this, we get `0x559f1a0c600b`.

We can do more with `disassemble`, though. If we want to see the actual bytecode, we can add the "raw" option with `disassemble/r`:

```
(gdb) disassemble/r
Dump of assembler code for function f:
   0x0000559f1a0c5163 <+0>: f3 0f 1e fa         endbr64
   0x0000559f1a0c5167 <+4>: 55                  push   %rbp
   0x0000559f1a0c5168 <+5>: 48 89 e5            mov    %rsp,%rbp
```

```
=> 0x0000559f1a0c516b <+8>: 48 8d 05 99 0e 00 00    lea    0xe99(%rip),%rax        # 0x559f1a0c600b
   0x0000559f1a0c5172 <+15>:    48 89 c7            mov    %rax,%rdi
   0x0000559f1a0c5175 <+18>:    e8 d6 fe ff ff      call   0x559f1a0c5050 <puts@plt>
   0x0000559f1a0c517a <+23>:    b8 00 00 00 00      mov    $0x0,%eax
   0x0000559f1a0c517f <+28>:    e8 c5 ff ff ff      call   0x559f1a0c5149 <g>
   0x0000559f1a0c5184 <+33>:    90                  nop
   0x0000559f1a0c5185 <+34>:    5d                  pop    %rbp
   0x0000559f1a0c5186 <+35>:    c3                  ret
End of assembler dump.
```

Certain common instructions, like `push %rbp`, are single-byte instructions (`55`).

We can also see the corresponding source code, if available, with the "source" options `dissamble/s`:

```
(gdb) disassemble/s
Dump of assembler code for function f:
example.c:
7   void f() {
   0x0000559f1a0c5163 <+0>: endbr64
   0x0000559f1a0c5167 <+4>: push   %rbp
   0x0000559f1a0c5168 <+5>: mov    %rsp,%rbp

8       printf("in f()\n");
=> 0x0000559f1a0c516b <+8>: lea    0xe99(%rip),%rax        # 0x559f1a0c600b
   0x0000559f1a0c5172 <+15>:    mov    %rax,%rdi
   0x0000559f1a0c5175 <+18>:    call   0x559f1a0c5050 <puts@plt>

9       g();
   0x0000559f1a0c517a <+23>:    mov    $0x0,%eax
   0x0000559f1a0c517f <+28>:    call   0x559f1a0c5149 <g>

10  }
   0x0000559f1a0c5184 <+33>:    nop
   0x0000559f1a0c5185 <+34>:    pop    %rbp
   0x0000559f1a0c5186 <+35>:    ret
End of assembler dump.
```

Here we see lines of source code (including line numbers), followed by the instructions involved in executing them. You can even combine the raw and source options with `disassemble/rs`, which does what you would expect.

We will see another way to display assembly instructions later.

Synonyms

- `disassemble`, `disas`

Examining the Stack

Viewing the Backtrace

If we set a breakpoint in `g` and run our program, we can see the current stack with the `where` command:

```
(gdb) where
#0  g () at example.c:4
#1  0x000055fc5f7d1184 in f () at example.c:9
#2  0x000055fc5f7d11b3 in main (argc=1, argv=0x7fff3464e5f8) at example.c:14
```

The frame at `#0` is the current execution context, while `#1` is its caller, and so on. `backtrace` and `bt` are synonyms for `where`.

If all we are trying to do is track down a segmentation fault, we might not need to do anything other than run your program in GDB and use `where` to find the offending statement.

Changing the Current Function View

Depending on the specific instruction causing an error, you might need to look in the the calling frame. The `up` command lets you change the active frame (the function being examined) by switching to the current function's caller. You can call this multiple times. The reverse of this is `down`.

For example, if we run `up` at the previous breakpoint, we see

```
(gdb) up
#1  0x000055fc5f7d1184 in f () at example.c:9
9       g();
```

This tells us that we're now in frame `#1`, which is line 9 of `example.c`, in the function `f`, and we've called `g()`. Running it again gives us

```
(gdb) up
#2  0x000055fc5f7d11b3 in main (argc=1, argv=0x7fff3464e5f8) at example.c:14
14      f();
```

which is the point in `main` where we called `f()`.

Running `down` twice takes us back to `#0` (`g`):

```
(gdb) down
#1  0x000055fc5f7d1184 in f () at example.c:9
9       g();
(gdb) down
#0  g () at example.c:4
4       printf("in g()\n");
```

Inspecting Frame Information

Every function call has an associated stack frame, which holds function arguments, local variables, and any information we need to address these or unwind to the calling function's frame. We can see this information with `info frame`. Both the `info` command and the `frame` subcommand can be shortened to single characters, so `i frame`, `info f`, and `i f` all behave the same.

Here's what we see in `g`:

```
(gdb) info frame
Stack level 0, frame at 0x7fff3464e4b0:
 rip = 0x55fc5f7d1151 in g (example.c:4); saved rip = 0x55fc5f7d1184
 called by frame at 0x7fff3464e4c0
 source language c.
 Arglist at 0x7fff3464e4a0, args:
 Locals at 0x7fff3464e4a0, Previous frame's sp is 0x7fff3464e4b0
 Saved registers:
  rbp at 0x7fff3464e4a0, rip at 0x7fff3464e4a8
```

This tells us where the frame begins (`0x7fff3464e4b0`), and all valid stack addresses for this frame will have *lower* values than this (because the stack grows *downward*). It also tells us the current value of `rip`, the instruction pointer (`pc` on ARM platforms), which is the address of the next instruction for the CPU to execute, as well as the saved `rip` from the caller, which is the address of the next instruction in the caller after this function returns. On AMD64 (the most common 64-bit x86-family architecture), we store the base and instruction pointers for the caller in our frame. For ARM, these are typically only stored if this function is going to call another one.

`info frame` also takes an optional integer argument, which equates to the frame level as shown by `where`. That is, `info frame 0` will show us information about the stack frame of `g`, `info frame 1` will show us information about the stack frame of `f`, and `info frame 2` will show us information about the stack frame of `main`.

Synonyms

- `where`, `backtrace`, `bt`
- `info frame`, `i f`

Examining Variables and Memory

We are going to look at a different program now:

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

void f(int a) {
  int b;
  char c[] = "hello, world!";
  void* p = &b;

  printf("in f()\n");
  return;
}

int main(int argc, char** argv) {
  f(1);
  return 0;
}
```

Local Variables

When you are in a function, you can refer to local and global variables by name. To see all of the local variables in the format that gdb thinks makes the most sense, you can run `info locals`. For example:

```
(gdb) info locals
b = 0
c = "hello, world!"
p = 0x7ffe2d1aac7c
```

You can also view the function's arguments:

```
(gdb) info args
a = 1
```

For any of these symbols, you can print their values:

```
(gdb) print a
$5 = 1
(gdb) print b
$6 = 0
(gdb) print c
$7 = "hello, world!"
(gdb) print p
$8 = (void *) 0x7ffe2d1aac7c
```

The `print` command can also take expressions:

```
(gdb) print b
$9 = 0
(gdb) print b+2
$10 = 2
(gdb) print c[3]
$11 = 108 'l'
(gdb) print &b
$12 = (int *) 0x7fff5ad997cc
(gdb) print 1 + 5
$13 = 6
```

You can modify local variables and function arguments with `set variable`:

```
(gdb) set variable b = 3
(gdb) print b
$14 = 3
```

This can be very useful when you're trying to diagnose an issue, and need to trigger a particular branch that is otherwise very difficult to reach.

Synonyms

- `print`, `p`
- `set variable`, `set var`

Registers

Running programs make use of the CPU's registers for information that needs to be accessed particularly quickly. You can see all of the registers' current values with `info registers`:

```
(gdb) info registers
rax            0x7fff5ad997cc      140734717597644
rbx            0x7fff5ad99938      140734717598008
rcx            0x55d505bc7db8      94373412634040
rdx            0x7fff5ad99948      140734717598024
rsi            0x7fff5ad99938      140734717598008
rdi            0x1                 1
rbp            0x7fff5ad997f0      0x7fff5ad997f0
rsp            0x7fff5ad997b0      0x7fff5ad997b0
r8             0x0                 0
r9             0x7f4affbe3380      139960094962560
r10            0x7fff5ad99530      140734717596976
r11            0x203               515
r12            0x1                 1
r13            0x0                 0
r14            0x55d505bc7db8      94373412634040
r15            0x7f4affc16000      139960095170560
rip            0x55d505bc51ab      0x55d505bc51ab <f+66>
...
```

Note that for many of these, we see the register's value in hexadecimal and decimal. For addresses, both columns are in hexadecimal, and if the address corresponds to a known symbol, that too is included. For instance, `rip` (the instruction pointer) is displayed with `<f+66>`, which means 66 bytes after the beginning of the function `f`.

If we only want to see particular registers, we can specify these as additional arguments:

```
(gdb) info registers rax rbx
rax            0x7fff5ad997cc      140734717597644
rbx            0x7fff5ad99938      140734717598008
```

Registers can be used in expressions, such as when calling `print`, by preceding the register name with `$`:

```
(gdb) print $rip
$15 = (void (*)()) 0x55d505bc51ab <f+66>
```

We can also set register values:

```
(gdb) set $rip = f+67
(gdb) print $rip
$16 = (void (*)()) 0x55d505bc51ac <f+67>
(gdb) set $rip = f+66
(gdb) print $rip
$17 = (void (*)()) 0x55d505bc51ab <f+66>
```

Note that we don't require a subcommand for `set` in this case.

Synonyms

- `info registers`, `info reg`

Examining Memory

One extremely powerful command is `x`, which allows you to examine the contents of memory. It takes an address, which might be an expression, and an optional format string. Let's start with a simple example:

```
(gdb) x &a
0x7fff154b62ac: 0x00000001
(gdb) x &b
0x7fff154b62bc: 0x00000000
(gdb) x c
0x7fff154b62ca: 0x6c6c6568
(gdb) x p
0x7fff154b62bc: 0x00000000
```

In the first column, we see the address of the first byte being displayed on that line. What appears next is the contents of memory beginning at that address. Here, our default is to display one *word* (4 bytes) of memory as an integer in hexadecimal. Regardless of the endianness of the platform, this will be displayed in "normal reading" format, which is big-endian. For `c`, which is a string, this might look odd on a little-endian machine (as was used here). From the ASCII character table, `6c` is `l`, `65` is `e`, and `68` is `h`.

For expressions, we might do something like:

```
(gdb) x c+2
0x7fff154b62cc: 0x2c6f6c6c
```

Note that `0x7fff154b62cc` is `0x7fff154b62ca` plus 2.

Format strings are where `x` really shines. To specify a format, we follow the `x` with `/FMT`, where `FMT` is our format string. The display above is equivalent to:

```
(gdb) x/wx c
0x7fff154b62ca: 0x6c6c6568
(gdb) x/xw c
0x7fff154b62ca: 0x6c6c6568
```

How do we build a format string? There are three components, all of which are optional. If not specified, the prior value of the component is used. First, we have a *count*. This is the number of "things" we want to display, which will depend on the other components. Since we haven't specified this, we're getting the default of 1. After this are a *format letter* and a *size letter*, which can be provided in either order.

The format letters are:

- `t`: binary
- `o`: octal
- `d`: decimal
- `u`: unsigned decimal
- `x`: hexadecimal
- `z`: 0-padded hexadecimal
- `f`: floating-point
- `a`: address
- `i`: instruction
- `c`: character
- `s`: string

The size letters are:

- `b`: bytes
- `h`: half-word (2 bytes)
- `w`: word (4 bytes)
- `g`: giant (8 bytes)

Some things to note:

- For the address, instruction, character, and string formats, you should not supply a size.
- In most other cases, you probably want hexadecimal.

Let's take a look at a few of these, beginning with a string:

```
(gdb) x/s c
0x7fff154b62ca: "hello, world!"
(gdb) x/14c c
0x7fff154b62ca: 104 'h' 101 'e' 108 'l' 108 'l' 111 'o' 44 ',' 32 ' ' 119 'w'
0x7fff154b62d2: 111 'o' 114 'r' 108 'l' 100 'd' 33 '!' 0 '\000'
```

Here we see that the `s` format letter tells gdb to treat memory it sees as a null-terminated string, while `c` treats each ASCII character separately, displaying both the numeric value (in decimal) and the character or an octal code, depending on whether it's printable or not. We can also use these format letters with non-string data:

```
(gdb) x/s &b
0x7fff154b62bc: ""
(gdb) x/s &a
0x7fff154b62ac: "\001"
```

Obviously, in most cases this won't be useful.

Next, let's look at the instructions format letter:

```
(gdb) x/i $rip
=> 0x5638632241ab <f+66>:   lea    0xe52(%rip),%rax        # 0x563863225004
(gdb) x/i 0x5638632241ab
=> 0x5638632241ab <f+66>:   lea    0xe52(%rip),%rax        # 0x563863225004
(gdb) x/i f+66
=> 0x5638632241ab <f+66>:   lea    0xe52(%rip),%rax        # 0x563863225004
(gdb) disassemble f
Dump of assembler code for function f:
   0x0000563863224169 <+0>: endbr64
```

```
   0x000056386322416d <+4>: push   %rbp
   0x000056386322416e <+5>: mov    %rsp,%rbp
   0x0000563863224171 <+8>: sub    $0x40,%rsp
   0x0000563863224175 <+12>:    mov    %edi,-0x34(%rbp)
   0x0000563863224178 <+15>:    mov    %fs:0x28,%rax
   0x0000563863224181 <+24>:    mov    %rax,-0x8(%rbp)
   0x0000563863224185 <+28>:    xor    %eax,%eax
   0x0000563863224187 <+30>:    movabs $0x77202c6f6c6c6568,%rax
   0x0000563863224191 <+40>:    mov    %rax,-0x16(%rbp)
   0x0000563863224195 <+44>:    movabs $0x21646c726f7720,%rax
   0x000056386322419f <+54>:    mov    %rax,-0x10(%rbp)
   0x00005638632241a3 <+58>:    lea    -0x24(%rbp),%rax
   0x00005638632241a7 <+62>:    mov    %rax,-0x20(%rbp)
=> 0x00005638632241ab <+66>:    lea    0xe52(%rip),%rax        # 0x563863225004
(gdb) x/15i f
   0x563863224169 <f>:  endbr64
   0x56386322416d <f+4>:    push   %rbp
   0x56386322416e <f+5>:    mov    %rsp,%rbp
   0x563863224171 <f+8>:    sub    $0x40,%rsp
   0x563863224175 <f+12>:   mov    %edi,-0x34(%rbp)
   0x563863224178 <f+15>:   mov    %fs:0x28,%rax
   0x563863224181 <f+24>:   mov    %rax,-0x8(%rbp)
   0x563863224185 <f+28>:   xor    %eax,%eax
   0x563863224187 <f+30>:   movabs $0x77202c6f6c6c6568,%rax
   0x563863224191 <f+40>:   mov    %rax,-0x16(%rbp)
   0x563863224195 <f+44>:   movabs $0x21646c726f7720,%rax
   0x56386322419f <f+54>:   mov    %rax,-0x10(%rbp)
   0x5638632241a3 <f+58>:   lea    -0x24(%rbp),%rax
   0x5638632241a7 <f+62>:   mov    %rax,-0x20(%rbp)
=> 0x5638632241ab <f+66>:   lea    0xe52(%rip),%rax        # 0x563863225004
```

Here, we see that we can inspect the current instruction pointer, the literal address value, and the symbol `f` with a 66-byte offset. We also compare the output of disassembling `f` with requesting 15 instructions starting at `f`. For the last two, the formats are similar, but not identical, and we don't have the additional options that `disassemble` provides.

The `a` format letter takes 4 bytes (for a 32-bit architecture) or 8 bytes (for a 64-bit architecture), and displays it as a hexadecimal number with leading zeroes removed.

```
(gdb) x/a &p
0x7fff154b62c0: 0x7fff154b62bc
(gdb) p p
$24 = (void *) 0x7fff154b62bc
(gdb) p &p
$25 = (void **) 0x7fff154b62c0
```

If we display a region of memory that includes addresses within a known function, then that will be displayed:

```
(gdb) x/10a $rsp
0x7fff154b62a0: 0x7e00000006    0x100000000
0x7fff154b62b0: 0x0 0x0
0x7fff154b62c0: 0x7fff154b62bc  0x2c6f6c6c65680000
0x7fff154b62d0: 0x21646c726f7720    0x9158879107457c00
0x7fff154b62e0: 0x7fff154b6300  0x5638632241ee <main+29>
```

Here we see that there are two "addresses" shown per line (they might not correspond to actual addresses, this is simply how we are formatting them), which is 16 bytes. The address at `0x7fff154b62e8`, which is

`0x5638632241ee`, is the instruction at 29 bytes from the start of `main`. From `info frame`, we can verify that this is the location of the saved instruction pointer (that it, the return address for the function `f`).

The remaining things we'll look at are the `x` format letter combined with the size letters. In all cases, we will examine 64 bytes from `$rsp`, the top of the stack:

```
(gdb) x/64xb $rsp
0x7fff154b62a0: 0x06    0x00    0x00    0x00    0x7e    0x00    0x00    0x00
0x7fff154b62a8: 0x00    0x00    0x00    0x00    0x01    0x00    0x00    0x00
0x7fff154b62b0: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fff154b62b8: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fff154b62c0: 0xbc    0x62    0x4b    0x15    0xff    0x7f    0x00    0x00
0x7fff154b62c8: 0x00    0x00    0x68    0x65    0x6c    0x6c    0x6f    0x2c
0x7fff154b62d0: 0x20    0x77    0x6f    0x72    0x6c    0x64    0x21    0x00
0x7fff154b62d8: 0x00    0x7c    0x45    0x07    0x91    0x87    0x58    0x91
(gdb) x/32xh $rsp
0x7fff154b62a0: 0x0006  0x0000  0x007e  0x0000  0x0000  0x0000  0x0001  0x0000
0x7fff154b62b0: 0x0000  0x0000  0x0000  0x0000  0x0000  0x0000  0x0000  0x0000
0x7fff154b62c0: 0x62bc  0x154b  0x7fff  0x0000  0x0000  0x6568  0x6c6c  0x2c6f
0x7fff154b62d0: 0x7720  0x726f  0x646c  0x0021  0x7c00  0x0745  0x8791  0x9158
(gdb) x/16xw $rsp
0x7fff154b62a0: 0x00000006  0x0000007e  0x00000000  0x00000001
0x7fff154b62b0: 0x00000000  0x00000000  0x00000000  0x00000000
0x7fff154b62c0: 0x154b62bc  0x00007fff  0x65680000  0x2c6f6c6c
0x7fff154b62d0: 0x726f7720  0x0021646c  0x07457c00  0x91588791
(gdb) x/8xg $rsp
0x7fff154b62a0: 0x0000007e00000006  0x0000000100000000
0x7fff154b62b0: 0x0000000000000000  0x0000000000000000
0x7fff154b62c0: 0x00007fff154b62bc  0x2c6f6c6c65680000
0x7fff154b62d0: 0x0021646c726f7720  0x9158879107457c00
```

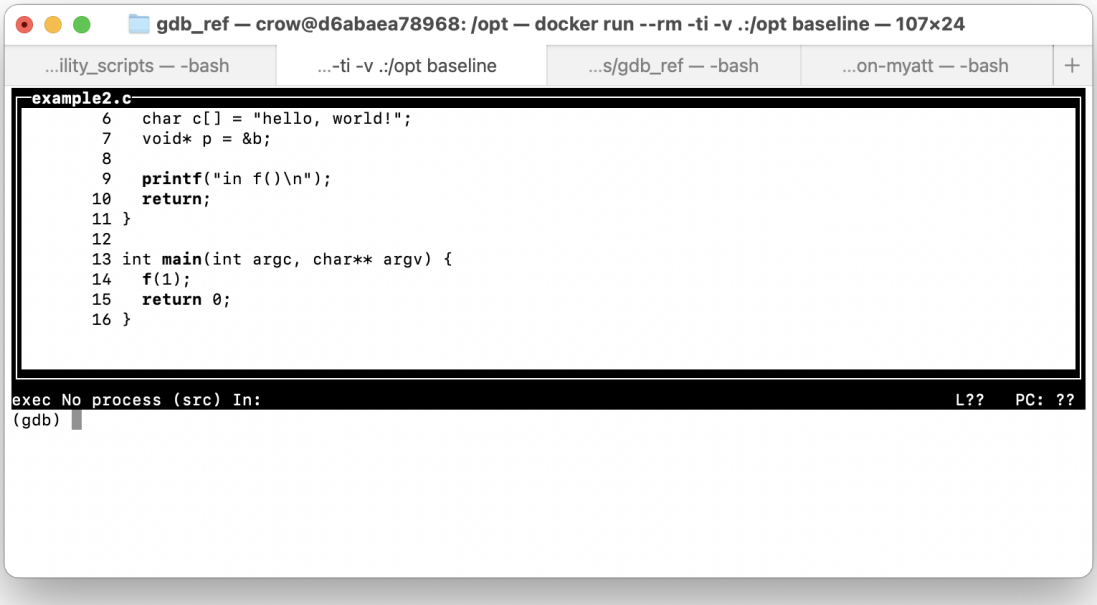
This is a little-endian system, which we can observe in more detail by repeating the exercise with only the 8 bytes beginning at `0x7fff154b62d0`:

```
(gdb) x/8xb 0x7fff154b62d0
0x7fff154b62d0: 0x20    0x77    0x6f    0x72    0x6c    0x64    0x21    0x00
(gdb) x/4xh 0x7fff154b62d0
0x7fff154b62d0: 0x7720  0x726f  0x646c  0x0021
(gdb) x/2xw 0x7fff154b62d0
0x7fff154b62d0: 0x726f7720  0x0021646c
(gdb) x/1xg 0x7fff154b62d0
0x7fff154b62d0: 0x0021646c726f7720
```

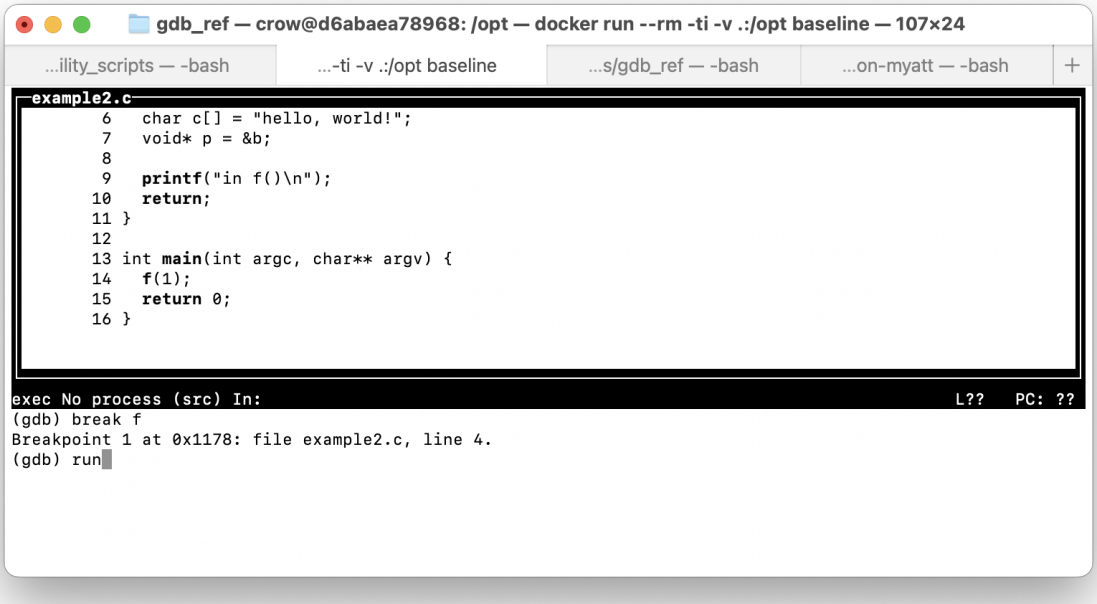
Note that there's no guarantee that variables will align on 8-byte boundaries, so you might not see the byte pattern you're looking for starting from any particular address. When in doubt, add fixed offsets from 1 to 7 to your starting address (eg, `$rsp+3`).

Text User Interface

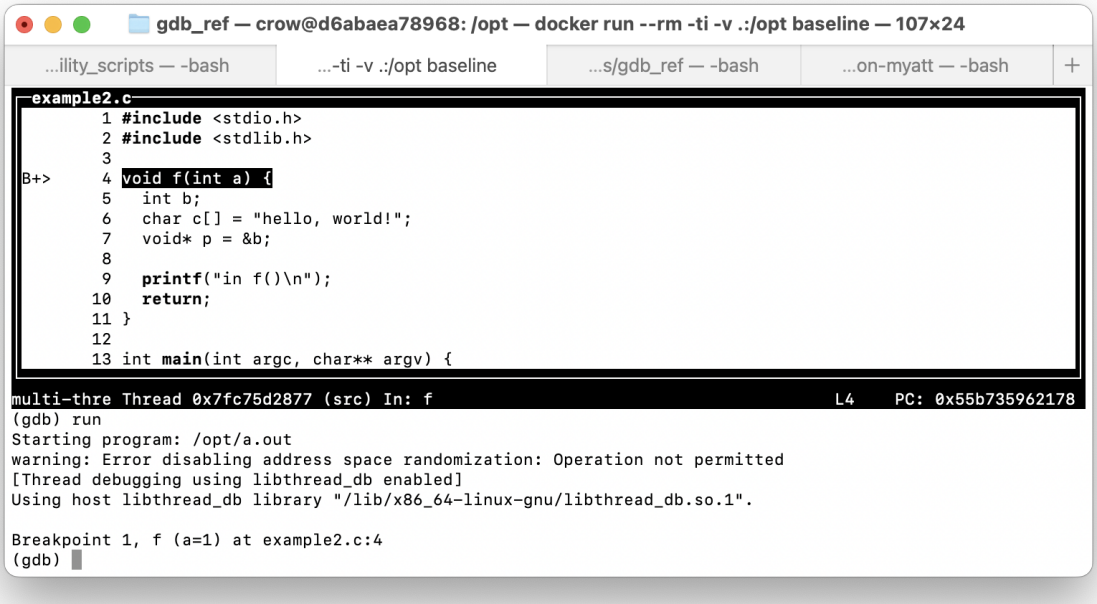
We do not cover it in detail, but you might find the *text user interface* useful. Here we show a few screen captures of this in action, with the command details provided in later chapters. You can enable it with the command `tui enable` (`tui disable` disables the text user interface), after which you will see something like:



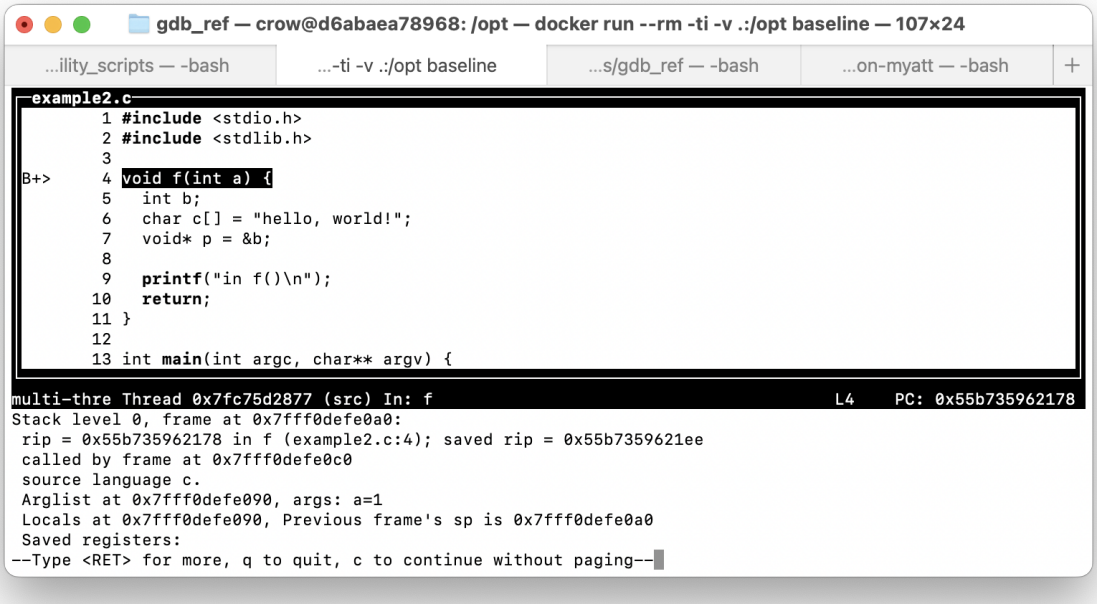
We can set a breakpoint and run the program as normal:



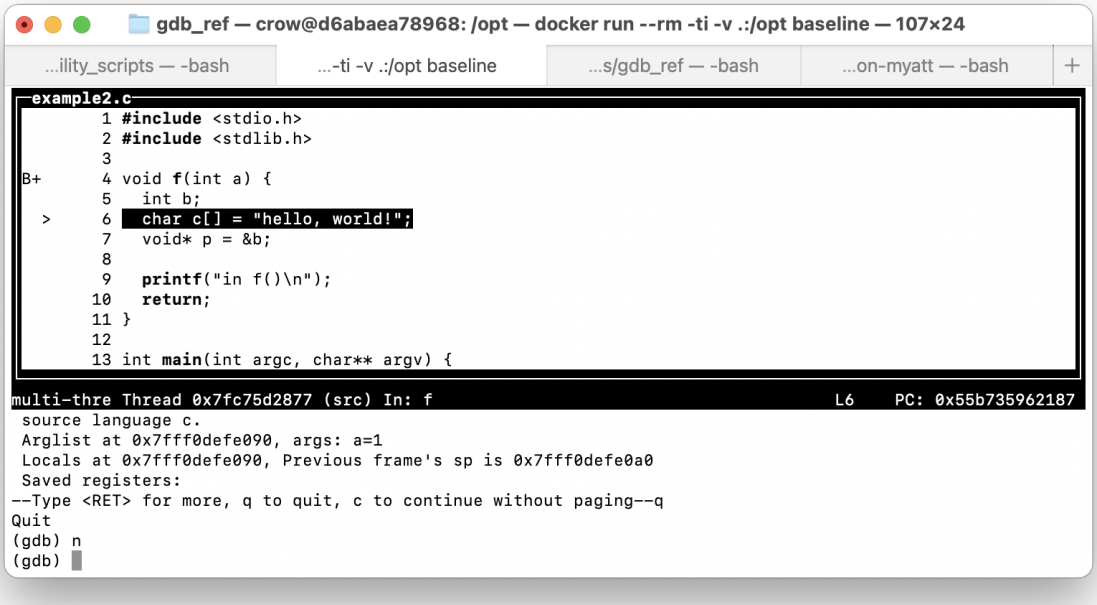
We now run the program, hitting the breakpoint at the start of the function `f`:



If we run `info frame`, we can confirm that the information displayed matches what we would expect:



If we run `next`, we can execute the current line and see the next one to be executed. The information displayed by the T U I is updated automatically:



There are other views that you can enable in the text user interface, and much more that it can do.

Appendix: Quick Reference

- Starting GDB
  - `gdb <program>`: debug the program `<program>`
  - `gdb <program> <core>`: debug the program `<program>` with core file `<core>`
  - `gdb -p <pid>`: debug the running program with PID `<pid>`
  - `ulimit -S -c unlimited`: allow the system to produce core files of unlimited size [shell command, not gdb]
  - `ulimit -S -c 0`: limit core file sizes to 0 bytes (ie, disable core dumps) [shell command, not gdb]
- Running a Program
  - `run`: run the program without command-line arguments
  - `run <arg1> [<arg2>...]`: run the program with command-line arguments
- Setting Breakpoints
  - `break <function>`: set a breakpoint at the first instruction of `<function>`
  - `break *<function> + <offset>`: set a breakpoint at the instruction `<offset>` bytes from the start of function `<function>`
  - `break <line>`: set a breakpoint at line `<line>`
  - `break <file>:<line>`: set a breakpoint at line `<line>` in source file `<file>`
- Controlling Execution
  - `next`: execute one source code statement in the current function
  - `next <num>`: run `next <num>` times
  - `step`: execute one source code statement, descending into other functions if appropriate
  - `step <num>`: run `step <num>` times
  - `stepi`: execute one instruction
  - `stepi <num>`: run `stepi <num>` times
  - `continue`: run until a breakpoint is encountered or the program terminates
- Examining the Code
  - `disassemble`: display the assembly code for the current function
  - `disassemble <function>`: display the assembly code for the function `<function>`
  - `disassemble/r`: like `disassemble`, but includes the raw bytes of each instruction
  - `disassemble/s`: like `disassemble`, but groups instructions with corresponding lines of source code, which are also displayed
  - `disassemble/rs`: combines `/r` and `/s` options
- Viewing the Backtrace
  - `where`, `bt`, `backtrace`: displays the program's stack frames, starting with the function containing the instruction pointer
- Changing the Current Function View
  - `up`: change the active stack frame to the current frame's caller
  - `down`: change the active stack frame to the current frame's callee
- Inspecting Frame Information
  - `info frame`: display stack frame information for the active stack frame
  - `info frame <num>`: display stack frame information for the frame `<num>`, which is numbered the same as in the backtrace
- Local Variables

  - `info locals`: display the current function's local variables, with values
  - `info args`: display the current function's arguments, with values
  - `print <expression>`: print the value of `<expression>`
  - `set variable <var> = <val>`: set the variable `<var>` to `<val>`
- Registers
  - `info registers`: display all registers and their values
  - `info registers <reg1> [<reg2> ...]`: display the specified registers and their values
  - `print $<reg>`: print the value of register `<reg>`
  - `set $<reg> = <val>`: set the value of register `<reg>` to `<val>`
- Examining Memory
  - `x <addr-expression>`: examine the memory specified by `<addr-expression>`
  - `x/<fmt> <addr-expression>`: examine the memory specified by `<addr-expression>` with the format string `<fmt>`
- Format Strings
  - `<N>`: repeat count of `<N>` items, where each item is determined by the format and size letters
  - format letter
    * `t`: binary
    * `o`: octal
    * `d`: decimal
    * `u`: unsigned decimal
    * `x`: hexadecimal
    * `z`: 0-padded hexadecimal
    * `f`: floating-point
    * `a`: address
    * `i`: instruction
    * `c`: character
    * `s`: string
  - size letter
    * `b`: bytes
    * `h`: half-word (2 bytes)
    * `w`: word (4 bytes)
    * `g`: giant (8 bytes)