gdb

Dr. Yihsiang Liow (August 11, 2023)

Contents

1 Introduction

A debugger or debugging tool is a computer program used to test and debug other programs (the "target" program). The main use of a debugger is to run the target program under controlled conditions that permit the programmer to track its execution and monitor changes in computer resources that may indicate malfunctioning code. Typical debugging facilities include the ability to run or halt the target program at specific points, display the contents of memory, CPU registers or storage devices (such as disk drives), and modify memory or register contents in order to enter selected test data that might be a cause of faulty program execution. – *Wikipedia*

The GNU Debugger (GDB) is a portable debugger that runs on many Unix-like systems and works for many programming languages, including Ada, Assembly, C, C++, D, Fortran, Go, Objective-C, OpenCL C, Modula-2, Pascal, Rust, and partially others. – *Wikipedia*

A debugger such as `gdb` is a tool that might help you in locating a bug. It allows you to step through your program one statement at a time and allows you to look at the values of your variables as you trace your code. Of course you can achieve the same thing by temporarily inserting print statements into your code; but after you are done you have to remove/comment those print statements. However `gdb` is more than just help with eliminating debug printing.

`gdb` is considered the de facto standard for Unix-based C/C++ debugger and is heavily used by C/C++ professionals. Many C/C++ IDE actually uses `gdb` (example: kdevelop).

`gdb` is a very powerful debugger. But it's a programming / software engineering tool. A software engineering tool will save you some time. It might help you get data, organize data, etc. But there's one thing it won't do. It won't think for you. Software engineering tools (including debuggers) don't solve problems. A software engineer solves problems.

2 Prerequisites

In terms of prerequisites (your background and the platform pre-requisites), I assume

- You know C++ as in you know CISS240, CISS245, CISS350. If you have CISS240, you can still benefit from studying this set of notes as long as you ignore exercises/comments on pointers and classes.
- You know how to use our fedora virtual machines. Specifically, I'm using our Fedora 31 virtual machine. But I'm using `gdb` which works on all unix-based OSs.
- You know basic linux commands. (See my `unix1` tutorial.)
- You know how to write text files using a text editor (in the virtual machine). I'll be using `emacs`. You can use whatever you want. However in a later section, I'll be showing you how to run `gdb` inside `emacs`. (See my `emacs` tutorial.)
- You know how to build executables. I'll be using g++ and `make`. (See my g++ and `make` tutorial.) But you don't really need to know g++ and `make` in detail.

3 Installation

To make sure you have `gdb`, run `gdb` in your bash shell and then quit by doing Ctrl-D. If `gdb` is not found, as root, you should install `gdb` by executing

```
dnf -y install gdb
```

In the next few sections on using `gdb`, `gdb` might give you a warning such as

```
Missing separate debuginfos, use: dnf debuginfo-install libasan-9.3.1-2 [... etc. ...]
```

That means `gdb` needs the libraries `libasan-9.3.1-2.fc31.x86_64`, etc. In that case, as root, execute the above command `gdb` gave you:

```
dnf debuginfo-install libasan-9.3.1-2 [... etc. ...]
```

4 Starting and quitting

First write `main.cpp`:

```
#include <iostream>

int main()
{
    std::cout << "hello world" << std::endl;
    return 0;
}
```

Now compile the above with the `-g` option:

```
[student@localhost test] g++ main.cpp -g -o main.exe
```

The executable binary file is `main.exe`. Now run `gdb` on `main.exe`:

```
[student@localhost test] gdb main.exe
```

You will see the `gdb` prompt where you can execute commands:

```
(gdb)
```

To quit `gdb` you type `q` at the `gdb` prompt.

If you ran `gdb` without loading the executable, you can load that inside `gdb` by doing:

```
(gdb) file main.exe
```

Try that now.

Exercise 4.1. Quit `gdb` and then run it with `main.exe` again. Is there any other way to quit? □

If you have not studied my makefile pdf, you should do that now. Here's a very simple `makefile`

```
# file: makefile
main.exe:
    g++ -g *.cpp -o main.exe
```

Note that you should compile with the `-g` option when using `gdb`.

To compile `main.exe` with the `makefile`, execute `make` in the bash shell. If you have not studied my `make.pdf`, you should do it now.

5 Listing

You can view your source in gdb. Here's one way of doing it. Type

```
(gdb) list
```

at the gdb prompt Try it now. You see that gdb will list your program. You can also do

```
(gdb) l
```

(that's the letter l and not the number 1!)

If you do `list` again, gdb will tell you that you at the end of your source file. So `list` list from the last line of your source file. By default verb!gdb! list 10 lines.

Another way is to specify a line number like this:

```
(gdb) l 3
```

gdb will list the source around line 3 of your program. You can specify a range like this:

```
(gdb) l 3, 5
```

That will list lines 3, 4, and 5.

Here are some other options:

```
l 3,        -- lists starting at line 3
l ,3        -- lists up to line 3
l +         -- lists after this point
l -         -- lists before this point
```

Now modify your program:

```
#include <iostream>

void f()
{
    std::cout << "f()" << std::endl;
}

void g()
{
    std::cout << "g()" << std::endl;
}

void h()
{
    std::cout << "h()" << std::endl;
}

int main()
```

```
{
    std::cout << "hello world" << std::endl;
    f();
    g();
    h();
    return 0;
}
```

Again compile your program with the -g option. Run `gdb` on the executable. And execute `l` several times. Notice that each time you run `l`, you get about 10 lines.

This time try to list your source around the function f like this:

```
(gdb) l f
```

Exercise 5.1. Can `gdb` list source near a class? □

If you have more than one source file and one of them is named `ABC.cpp` then you can specify the source file name in the list command like this:

```
(gdb) l ABC.cpp:3
```

This will list `ABC.cpp` starting around line 3. Or you can do this:

```
(gdb) l ABC.cpp:f
```

This will list `ABC.cpp` starting around function `f`.

6 Running a program

Write this `main.cpp`:

```
#include <iostream>

void g()
{
    std::cout << "A" << '\n';
    std::cout << "B" << '\n';
}

void f()
{
    std::cout << "a" << '\n';
    std::cout << "b" << '\n';
    g();
    std::cout << "c" << '\n';
    std::cout << "d" << '\n';
}

int main()
{
    std::cout << "1" << '\n';
    std::cout << "2" << '\n';
    f();
    std::cout << "3" << '\n';
    std::cout << "4" << '\n';
    return 0;
}
```

Run `gdb` with the the executable of the above source file. To run your executable within `gdb` type this at the `gdb` prompt:

```
(gdb) r
```

No surprises here. (`gdb` will also show you some extra output including loading of libraries.)

If you need to run your program with some input file, say `input.txt`, you do this in `gdb`:

```
(gdb) r < input.txt
```

Exercise 6.1. Verify I'm not lying: Write a program `main.exe` that accepts an integer and prints that integer. Write the input in a file named `input.txt`. Run `main.exe` in `gdb` using `input.txt` as input. □

7 Breakpoints, next, step, continue

Now let's set a breakpoint. (You'll see what it does in a minute.) Here's our `main.cpp`

```
1  #include <iostream>
2
3  void g()
4  {
5      std::cout << "A" << '\n';
6      std::cout << "B" << '\n';
7  }
8
9  void f()
10 {
11     std::cout << "a" << '\n';
12     std::cout << "b" << '\n';
13     g();
14     std::cout << "c" << '\n';
15     std::cout << "d" << '\n';
16 }
17
18 int main()
19 {
20     std::cout << "1" << '\n';
21     std::cout << "2" << '\n';
22     f();
23     std::cout << "3" << '\n';
24     std::cout << "4" << '\n';
25     return 0;
26 }
```

I'm going to put a breakpoint at line 20:

```
(gdb) b 20
```

Now run the program with the r command. The output looks like this:

```
Starting program: /home/student/shares/yliow/Documents/work/projects/gdb/test/main.exe

Breakpoint 1, main () at main.cpp:20
20            std::cout << "1" << '\n';
```

Aha! The point of a breakpoint is to stop the execution of the program. Why? Because you suspect there's a bug coming up and you want stop here and analyze the program carefully.

Also, note that your breakpoint is breakpoint 1. Each breakpoint is given an id.

You can continue the execution of your program in several ways: using continue, step, and next. First try to continue the execution:

```
(gdb) c
```

No surprises there. So continue will run the program to the end. If during the run the

program hits another breakpoint, it will temporarily pause execution again.

Next, run (with `r`) your program a second time. Again you hit the breakpoint at line 20. Now, to continue the execution of your program, use step:

```
(gdb) s
```

You'll get this:

```
std::operator<< <std::char_traits<char> > (__out=..., __s=0x40201d "1")
    at /usr/src/debug/gcc-9.3.1-2.fc31.x86_64/obj-x86_64-redhat-linux/x86_64-redhat-linux/libstdc++-v3/include/ostream:565
565           operator<<(basic_ostream<char, _Traits>& __out, const char* __s)
```

Do a few more steps with `s`.

What's happening? With step, `gdb` will execute the next instruction of your program. In the above example, the next few steps involved C++ code from printing (i.e., `operator<<`).

Finally let's try the next command. Run your program again (with `r`) and when you hit the breakpoint, do

```
(gdb) n
```

Do this a couple of times.

```
(gdb) n
1
21            std::cout << "2" << '\n';
(gdb) n
2
22            f();
(gdb) n
a
b
A
B
c
d
23            std::cout << "3" << '\n';
```

You notice that the program execution pauses at every statement of `main()`. However it executes through function `f()` without pausing. `n` also did not pause at code from `operator<<`.

So

- continue (i.e., `c`) will run your program to the end, unless of course it's stopped by a breakpoint.
- step (i.e., `s`) will run one statement at a time and pause.
- next (i.e., `n`) will execute the program, pausing only at statements of the function where you last hit a breakpoint.

Get it?

Instead of doing one step/next, you can do

```
(gdb) s 3
```

which will execute 3 steps. You can also do `n 3` to do next three times.

You can also set a breakpoint with a function name. Try this:

```
(gdb) b f
```

This will put a breakpoint at the first statement of the function `f()`. Run your program again and try it out. You can list all breakpoints by doing

```
info b
```

Exercise 7.1. Can you create a breakpoint at an inline function? □

Exercise 7.2. Write a program containing a class with a method. Verify that you can insert a breakpoint at the method. □

Now for disabling a breakpoint. You can do that using:

```
(gdb) disable 1
```

The "1" here refers to breakpoint 1 and not some line number. So

```
(gdb) info b
```

and look at the status of your breakpoints. Run your program again. You can enable a breakpoint after disabling it.

```
(gdb) enable 1
```

and do `info b` again. Do you see it's now active again? Run your program again. As you can see, disabling a breakpoint does not remove it. It's just not active.

If you want to remove a breakpoint you can do

```
(gdb) delete 1
```

Do `info b` to check.

If you are working with multiple files, you can include the name of the cpp file when you set breakpoints. For instance

```
(gdb) b ABC.cpp:10
```

sets a breakpoint at line 10 of `ABC.cpp`.

During the pause of a run, you can stop the run by performing a kill:

```
(gdb) k
```

Exercise 7.3. Verify that I'm not lying: Write a `main.cpp` with `main()` calling `f()` in `a.cpp`. `f()` print 1, 2, 3, 4, 5 with 5 separate statements. Run `gdb` on `main.exe`. In `gdb`, create a breakpoint at the statement in `f()` that prints 3. Run your program up to the breakpoint and finish the execution with `n`. □

Exercise 7.4. In `gdb`, run a program up to a breakpoint and then do a kill. □

Exercise 7.5. Try this: You can save your breakpoints by executing "`save breakpoints brk.txt`". When you restart your `gdb`, you recover your saved breakpoints by doing "`source brk.txt`".

8 Viewing program state

Why do we want to set breakpoints? Because we want to view the state of the program which is just a fancy way of saying we want to see the values of the variables in our program at the point when the program stops. This is useful in a debugging process: You set breakpoints just before the point where you suspect something is wrong, run the program up to that point, and view the values of the variables to see if values matches your expectation.

Modify your `main.cpp` as follows:

```
#include <iostream>

int main()
{
    int x = 0;
    x = 1;
    x = 2;
    x = 3;
    return 0;
}
```

Recompile your program and run `gdb` on the executable. Set a breakpoint at `main()`:

```
(gdb) b main
```

Next execute the program. When you hit the breakpoint, print the value of variable `x`:

```
p x
```

You will see that

```
(gdb) print x
$1 = 0
(gdb)
```

i.e., currently the value of `x` is `0`. Do next and then print the value of `x` again:

```
(gdb) n
1
7           x = 1;
```

Try next and print `x` a couple more times.

Note that you can only execute `p x` when `x` is in scope.

You can also print the value inside an array. For instance if `a` is an array (or a pointer), you can do

```
p a[1]
```

You can also print the whole array by doing

```
p a
```

To print the first 5 values of array `a`, you execute

```
p *a@5
```

You can also print the values of an object.

Exercise 8.1. Verify that you can print all the values of an array and the values of the array up to a certain index value. □

There are times when you know that something is wrong with the value of a single variable, say `x`, and you want to know the behavior of this variable. You can of course execute "`p x`" again and again. But there's an easier way to do this. You can get `gdb` to display the value of `x` at every breakpoint. You do that with this command:

```
(gdb) disp x
```

When you do `n`, the value of `x` is automatically printed. When you do `disp x`, you'll notice that when `gdb` prints `x`, it also prints a number next to `x`. That's the display id:

```
(gdb) disp x
1: x = 0
```

You can also see all displays using info:

```
(gdb) info disp
Auto-display expressions now in effect:
Num Enb Expression
1:   y  x
```

With that you can temporarily disable the display of `x`. For instance if the display id of `x` is `1`, to disable you do

```
(gdb) disable disp 1
```

To enable the display of `x` again, you do

```
(gdb) enable disp 1
```

You can list all the displays by doing

```
(gdb) info disp
```

You can remove a display using undisp with the id:

```
(gdb) undisp 1
```

Exercise 8.2. Using this program

```
int main()
{
    int x = 0; // set breakpoint here
    int y = 0;
    x = 1;
    y = 1;
    x = 2;
    y = 2;
    x = 3;
    y = 3;
    x = 4;
    y = 4;
    x = 5;
    y = 5;
    x = 6;
    y = 6;
    return 0;
}
```

Run this in `gdb`, set a breakpoint at line 3, run the program. At the breakpoint, print `x` and `y` (note: `y` is not declared yet). Then set up a display of `x`. Can you set up display of `y`? Do `n` two times. Set up display of `y`. Show all displays. Do `n` two times. Stop the display of `x`. Do `n` two times. Re-enable the display of `x`, and do `n` until the program ends. □

By the way, you can also print all the local variables by doing

```
(gdb) info locals
```

Exercise 8.3. Check if you can do the following:

(a) Can you print the address of a variable?
(b) If `p` is a pointer, can you print the value of `p` and the value that `p` points to?

□

Exercise 8.4. Check if you can do the following:

(a) Can you print an object?
(b) Can you print the value of a member variable of an object? What about private member variables?

□

9 Setting the value of a variable

You can set the value of a variable too. Here's our program again:

```
#include <iostream>

int main()
{
    int x = 1;
    std::cout << "1" << std::endl;
    std::cout << x << '\n';
    return 0;
}
```

Set a breakpoint at line 6. Run the program. At the breakpoint set the value of `x` to `999`:

```
(gdb) set variable x = 999
```

and issue the next command:

```
(gdb) n
```

You'll see that the next statement of the program prints 999.

Exercise 9.1. Create an example with an array. In `gdb` change the value of of the element of the array. □

10 Backtrace

A program execution usually weaves through lots of function calls. The backtrace command tells you where the program execution has gone through.

Modify our program as follows:

```
#include <iostream>

void foo2(int x)
{
    std::cout << "foo2" << std::endl;
    std::cout << 1 / x << std::endl;
}

void foo(int x)
{
    std::cout << "foo" << std::endl;
    --x;
    foo2(x);
}

int bar()
{
    std::cout << "bar" << std::endl;
    return 42;
}

double baz()
{
    std::cout << "baz" << std::endl;
    return 3.14;
}

int main()
{
    std::cout << "hello world" << std::endl;
    int x = 1;
    foo(x);
    x = x + 2;
    x = bar();
    x = baz();
    return 0;
}
```

First compile and run your program in your bash shell. You will get a program error. You do get an error message which might help. Next, let's get `gdb` to run it and see what happens. You get this error message.

```
Program received signal SIGFPE, Arithmetic exception.
0x00000000004007f4 in foo2 (x=0) at main.cpp:6
6            std::cout << 1 / x << std::endl;
```

Much better right? Notice that the error you get is an arithmetic exception and the signal SIGFPE is a signal sent by the CPU to the current running process to indicate this error.

Also, execute the following `gdb` command:

```
(gdb) bt
```

and you will see the function call stack, i.e., the functions starting at `main()` up to the function where the error occurs:

```
#0  0x00000000004007f4 in foo2 (x=0) at main.cpp:6
#1  0x0000000000400848 in foo (x=0) at main.cpp:13
#2  0x00000000004008d1 in main () at main.cpp:32
```

Each function uses a certain amount of memory – this is called a function frame (or function record). The function frames are organized as a stack (see CISS360).

The error message in `gdb` and the call stack will help you debug your program.

At this point, in `gdb`, you can go through the stack of function frames by move up and down through the function frames and see what causes the program crash. The program has crashed at `foo2`. You can print all the relevant variables in scope at this point and analyze what's happening.

```
(gdb) print x
$1 = 0
```

So the value of `x` at this point is `0`.

If you were expect `x` in `foo2` to be something else, you can move to the function frame before this function call and see why `0` was passed into the function. If you try to go `down`, you get this:

```
(gdb) down
Bottom (innermost) frame selected; you cannot go down.
```

Aha ... `down` is moving *forward* in the function call chain. So if you want the function frame of the function *before* `foo2`, you have to go `up`:

```
(gdb) up
#1  0x0000000000401207 in foo (x=0) at main.cpp:13
13           foo2(x);
```

So you are now at line 13 of function `foo` where you called `foo2(x)`. You can print the value of `x` that is currently in scope:

```
(gdb) print x
$1 = 0
```

(Of course in this case the value of `x` in `foo` is the same as the value of `x` in `foo2`.)

Get it? This is a pretty useful feature for debugging.

In case the stack is huge, you can do

```
(gdb) bt 2
```

to see the 2 function frames closest to the current point of execution, i.e., the previous 2. Doing

```
(gdb) bt -2
```

would be the opposite – those 2 frames further from the current point of execution, i.e., the first 2.

Exercise 10.1. As an execise, run `gdb` on the following program, moving up and down through the function frames and printing the variables which are in scope.

```
#include <iostream>

void foo2(int x)
{
    std::cout << "foo2" << std::endl;
    int a[10];
    std::cout << a[x] << '\n';
}

void foo(int x)
{
    std::cout << "foo" << std::endl;
    x *= 1000;
    foo2(x);
}

int bar()
{
    std::cout << "bar" << std::endl;
    return 42;
}

double baz()
{
    std::cout << "baz" << std::endl;
    return 3.14;
}

int main()
{
    std::cout << "hello world" << std::endl;
    int x = 1000;
    foo(x * 5);
    x = x + 2;
    x = bar();
    x = baz();
    return 0;
}
```

What is the signal in this case? Of course if you are a good programmer, you can see the error almost immediately. But pretend you don't see it. Practise moving forward and backward through the function frames and print the variables in scope.

11 Conditional breakpoints

You can set a breakpoint based on a condition. Here's the program to try out:

```
#include <iostream>

int main()
{
    for (int i = 0; i < 100; i++)
    {
        std::cout << i << std::endl;
    }
    std::cout << "end" << std::endl;
    return 0;
}
```

Compile it with the `-g` option and run `gdb` with the executable. At the `gdb` prompt, set a conditional breakpoint at line 7 when `i` is 95:

```
(gdb) b 7 if i == 95
```

Now run the program.

The above illustrate how to get to a point in the execution of the program quickly when that point is dependent on a variable and not the line number of the source.

Exercise 11.1. Can you use `<` instead of `==`? What else should you try? □

12 Watchpoint

Run `gdb` with this program:

```
#include <iostream>

int main()
{
    int x = 0; // breakpoint
    int y = 0;
    x = 1;
    y = 1;
    x = 2;
    y = 2;
    x = 3;
    y = 3;
    x = 4;
    y = 4;
    return 0;
}
```

Set a breakpoint at line 5 and execute `r`. When you reach the breakpoint, set a watch point on `x`:

```
watch x
```

Continue with `c` and you'll see this:

```
(gdb) c
Continuing.

Hardware watchpoint 2: x

Old value = 0
New value = 1
main () at main.cpp:8
8           y = 1;
```

You see that the program execution stop when `x` changes its value. (The line of code shown is line 8, but line 8 has not executed yet.) If you continue again you get

```
(gdb) c
Continuing.

Hardware watchpoint 2: x

Old value = 1
New value = 2
main () at main.cpp:10
10            y = 2;
```

Again you see that execution continues until the value of `x` changes.

This is obviously very helpful if you are watching a particular variable. If the value of `x` computed is incorrect, you can set a watch on `x` and see how `x` changes. You don't have to `p x` for every `n`.

Instead of watching how `x` changes (i.e., there's a write operation on `x`), you can also have a `rwatch`, i.e., a read watch, to see when the value of `x` was read. Run `gdb` with this

```
#include <iostream>

int main()
{
    int x = 0; // breakpoint
    int y = 0;
    x = 1;
    y = 1;
    x = 2;
    y = x; // value of x is read
    return 0;
}
```

Set a breakpoint at line 5. Execute `r`. At the breakpoint, create a read watch on `x`:

```
(gdb) rwatch x
```

Continue with `c` to get:

```
(gdb) c
Continuing.

Hardware read watchpoint 2: x

Value = 2
0x0000000000401170 in main () at main.cpp:10
10            y = x; // value of x is read
```

i.e., the program execution pauses when you reach line 10 where `x` is read.

For a read/write watch, you do

```
(gdb) awatch x
```

Like previous commands, you can execute `info watch` and see all watchpoints. Also, you do `info b`, you'll also see all breakpoints and all watchpoints.

```
(gdb) info b
Num     Type           Disp Enb Address            What
3       breakpoint     keep y   0x000000000040114a in main() at main.cpp:5
        breakpoint already hit 1 time
4       hw watchpoint  keep y                      x
```

You can `disable`, `enable`, `delete` a watchpoint using its id.

13 Recompiling

Of course you should know by now that during and after you analyzed your bug(s), you will need to change your source code, recompile, and retest. You can still recompile in your bash shell. When you re-run you program inside `gdb`, `gdb` will detect that your executable has changed and will reload it. But you can actually execute `make` in `gdb`.

Exercise 13.1. Verify I'm not lying: Write a simple C++ file. Compile it. Load it in `gdb`. Modify your C++ file and recompile it. Go back to `gdb` and do a `r`. Check that `gdb` reloads the executable. □

Exercise 13.2. Next, recompile (after modifying your C++ file) inside `gdb`. □

14 Emacs and gdb

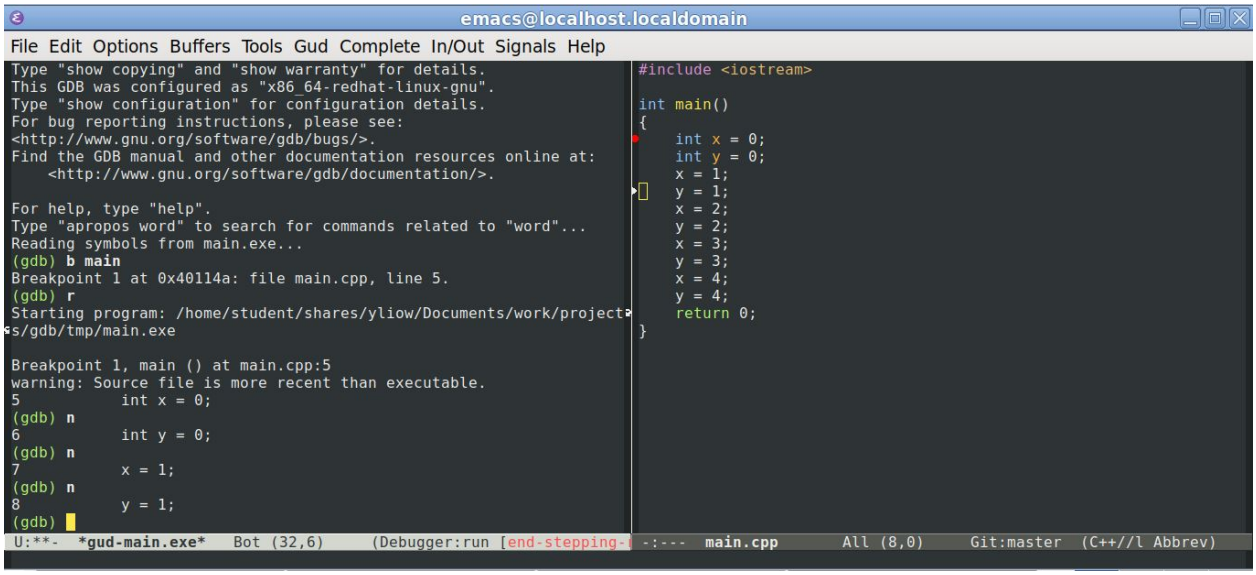
You can run `gdb` in `emacs`. Write a `main.cpp` (any would do). I assume you have the following very simple `makefile`:

```
main.exe: main.cpp
	g++ -g main.cpp -o main.exe
```

In the following remember that `C-x` means `Ctrl-c` and `M-x` means `Alt-x`.

Do the following:

- Run `emacs` in your bash shell with `emacs main.cpp &`. Maximize your emacs.
- In `emacs`, vertically split your frame (`C-x 3`).
- In `emacs`, in the left frame, do `M-x gdb`. `emacs` will ask if you want to run `gdb` on `main.cpp`. Change `main.cpp` to `main.exe`. You now have `gdb` running in your left frame and `emacs` editing `main.cpp` on your right frame.
- It's helpful to turn on line numbers in your right frame. Go to the `main.cpp` frame and do `M-x display-line-numbers-mode`.
- Go to the `gdb` frame. Set a breakpoint with `b main`, run with `r`, and do next `n` a couple of times. Emacs will show you your breakpoints in your `main.cpp` and also where `gdb` paused the execution.



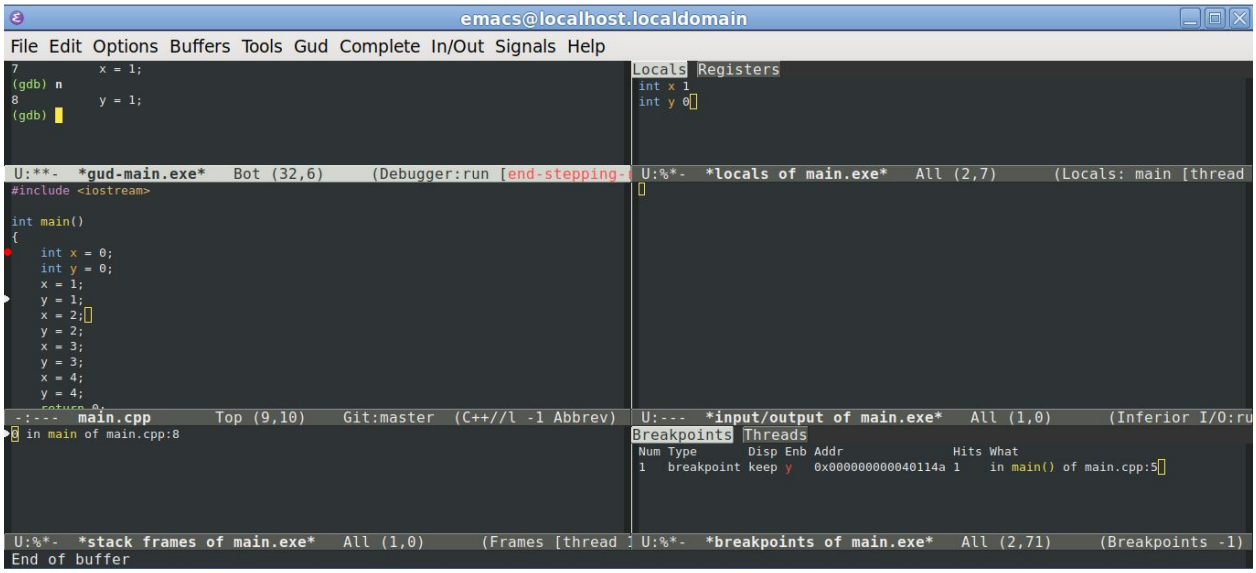
Nice right? By the way, click on the Gud button at the top and take a look at the pull down menu. (Gud = grand unified debugger.)

When you change your `main.cpp`, you want to recompile your `main.exe`. You can do `make` in `gdb` or in `emacs`. I'll do it in `emacs`. Do this:

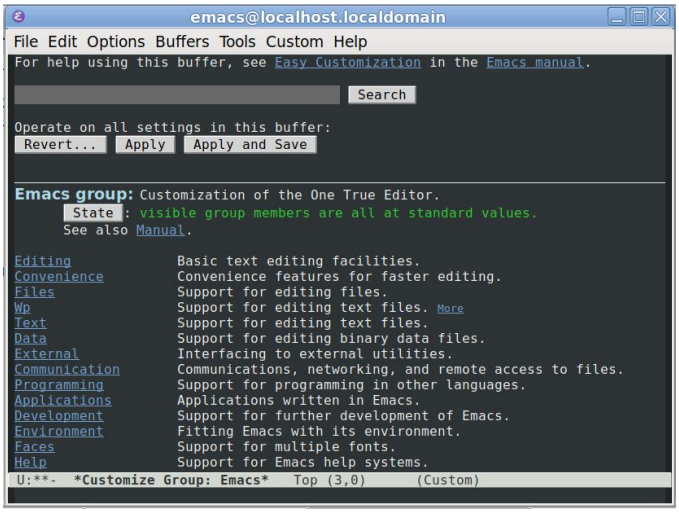
- In `emacs`, do `M-x compile`. `emacs` will ask if you want to run `make`. Press the enter key.
- `emacs` will most likely show you the results of g++, which will change one of your frames, i.e., either your `gdb` frame or your `main.cpp` frame will be hidden. You can always do `C-x-b` to switch buffer to get back to the previous setup.

Exercise 14.1. Try this: Run emacs. In `emacs` do `M-x gdb` and load your `main.exe`. Do a vertical split of the frame. Then do `r` in `gdb` and you'll see your source file is auto-loaded into one of the frames. □

You can also get emacs to give you a layout of common frames for `gdb` debugging. In emacs, do `M-x gdb-many-windows` and you'll get



You can customize your `emacs` so that you always have the `gdb-many-windows` layout whenever you run `gdb` in `emacs`. To do that, in `emacs` execute `M-x customize` and you'll get



In the search box type `gdb`. Then look for "Gdb Many Windows", click on it, and toggle it to turn on this option. You can also turn on "Gdb Show Main". This will show the C++ source file with the `main()` function in the source file frame. Then scroll to the top and do apply and save. This will change your emacs customization. Next time you run emacs and go `M-x gdb`, you'll get the `gdb-many-windows` layout.

(If you want to save your current `emacs` configuration before you mess around with it, your `emacs` configuration file is `/home/student/.emacs.d/init.el`.)

15 Summary

You can read more about the other commands using the `gdb` help command:

```
(gdb) h
```

There's a lot more to gdb. You can find more info on `gdb` on the web.

Here's a summary of the commands used in this tutorial.

```
file main.exe     load main.exe
q                 quit

r                 run from beginning to the end or to the first breakpoint
c                 continue: run to the next breakpoint
n                 next: run to beginning of next statement in current function
s                 step: run to the beginning of next statement
k                 kill

b 42              set breakpoint at line 42
b a.cpp:42        set breakpoint at line 42 of a.cpp
b f               set breakpoint at function f
disable b 1       disable breakpoint 1 (breakpoint is not removed)
enable b 1        enable breakpoint 1
info b            info on all breakpoints and watchpoints
delete 1          delete breakpoint 1
clear 23          delete breakpoint at line 23

p x               print value of variable x or all values of x if x is an array
p x[2]            print value at index 2 of array x
info locals       print all local variables

disp x            set automatic display of x
disable disp 1    disable disp 1
enable disp 1     enable disp 1
info disp         info on all displays
undispl 1         delete disp 1

watch x           set a (write) watchpoint on x
rwatch x          set a read watchpoint on x
awatch x          set a read and write watchpoint on x
info write        info on all watchpoints
disable 1         disable watchpoint 1
enable 1          enable watchpoint 1
delete 1          delete watchpoint 1

set variable x=1  set value of x to 1

bt                print backtrace
bt 2              previous 2 function frames in backtrace
bt -2             first 2 function frames in backtrace
up                previous function frame (forward)
down              next function frame (backward)

save b brk.txt    save breakpoints to brk.txt
source brk.txt    restore breakpoints from brk.txt

h                 help
```

Exercise 15.1. Try this: There's a GUI for `gdb` called ddd. Install it and try it out when you have a couple of minutes. □

Exercise 15.2. Try this: There's a text-based interface alternative for `gdb`. In your bash shell run `gdb -tui main.exe`. You can try it out for a couple of minutes. □

16 Exercises

Is the following correct? If it's not, fix it. Use `gdb` only if necessary.

```
#include <iostream>
#include <ctime>
#include <cstdlib>
#include <vector>

void randvector(std::vector< int > & v)
{
    for (auto && e: v) e = rand();
}


void swap(int & a, int & b)
{
    int t = a;
    a = b;
    b = t;
}


void bubblesort(std::vector< int > & v)
{
    size_t n = v.size();
    for (size_t i = n - 2; i >= 0; --i)
    {
        for (size_t j = 0; j <= i; ++j)
        {
            if (v[j] > v[j + 1])
            {
                swap(v[j], v[j + 1]);
            }
        }
    }
}


size_t binarysearch(const std::vector< int & v, int target)
{
    size_t left = 0;
    size_t right = v.size() - 1;
    while (left < right)
    {
        size_t mid = (right - left) / 2;
        if (x[mid] == target)
        {
            return mid;
        }
```

```
        else if (x[mid] < target)
        {
            right = mid;
        }
        else
        {
            left = mid;
        }
    }
}


int main()
{
    srand((unsigned int) time(NULL));
    std::vector< int > v(10000);
    randvector(v);
    bubblesort(v);
    int target;
    std::cin >> target;
    std::cout << binarysearch(v, target) << '\n';
    return 0;
}
```

17 Core dumps

A core dump consists of the recorded state of the working memory of a computer program at a specific time, generally when the program has crashed or otherwise terminated abnormally.

Run the following program in your bash shell:

```
#include <iostream>

int main()
{
    int x = 0;
    int y = 0;
    x = 1;
    y = 1;
    x = 2;
    y = 2;
    x = 3;
    y = 3;
    x = 4;
    y = 4;
    x = 1 / 0;

    return 0;
}
```

(Clearly there's a division-by-zero error, but pretend you didn't see that.) You'll get a core dump:

```
[student@localhost tmp]$ ./main.exe
Floating point exception (core dumped)
```

For Fedora 31, the core dump is stored somewhere else, not in your current directory. To get a copy of the core dump, do this:

```
[student@localhost tmp]$ coredumpctl -o core dump
```

and you'll get this

```
Hint: You are currently not seeing messages from other users and the system.
      Users in groups 'adm', 'systemd-journal', 'wheel' can see all messages.
      Pass -q to turn off this notice.
           PID: 14151 (main.exe)
           UID: 1000 (student)
           GID: 1000 (student)
        Signal: 8 (FPE)
     Timestamp: Thu 2023-08-10 10:54:23 EDT (38s ago)
  Command Line: ./main.exe
    Executable: /home/student/shares/yliow/Documents/work/projects/gdb/tmp/main.exe
 Control Group: /user.slice/user-1000.slice/session-2.scope
          Unit: session-2.scope
         Slice: user-1000.slice
       Session: 2
     Owner UID: 1000 (student)
       Boot ID: 7ac02c5b3b264d669b2c5299e1b823fa
```

```
    Machine ID: 8180067302a34256ab3a8de9a3f087da
      Hostname: localhost.localdomain
      cStorage: /var/lib/systemd/coredump/core.main\x2eexe.1000.7ac02c5b3b264d669b2c5299e1b823fa.14151.169167
9263000000.lz4
       Message: Process 14151 (main.exe) of user 1000 dumped core.

                Stack trace of thread 14151:
                #0  0x000000000040119b n/a (/home/student/shares/yliow/Documents/work/projects/gdb/tmp/main.e
xe)
                #1  0x00007fb187fda1a3 __libc_start_main (libc.so.6)
                #2  0x000000000040108e n/a (/home/student/shares/yliow/Documents/work/projects/gdb/tmp/main.e
xe)
More than one entry matches, ignoring rest.
```

You'll also have a `core` file:

```
[student@localhost tmp]$ ls core
core
```

Now run `gdb` like this:

```
[student@localhost tmp]$ gdb main.exe core
```

and you'll see

```
[student@localhost tmp]$ gdb main.exe core
GNU gdb (GDB) Fedora 8.3.50.20190824-30.fc31
... snipped ...
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from main.exe...
[New LWP 12548]
Core was generated by `./main.exe'.
Program terminated with signal SIGFPE, Arithmetic exception.
#0  0x000000000040119b in main () at main.cpp:15
15            x = 1/0;
```

In this case, the signal SIGFPE is a fatal arithmetic error. You can find information on linux signals if you google, including SIGFPE. In more complex cases, you want to print the values of the variables at this point. Your program is *not* running, but `gdb` can read the state of your program when it crashes through the information in the core dump. All you need to go now is to do the same as back in the section on backtrace: trace the program by analyzing the values of the variables in the function frames in the function call stack that lead up to the program crash.

The only difference between the scenario here and the example in the section on backtrace is that when analyzing your program using the core dump, your program is not running. We are analyzing a snapshot of a program execution (up to the crash) that was stored on harddrive.

Analyzing the core dump is useful. For instance if your program uses random numbers and

these random numbers are not stored by your program, then it would be very difficult to recreate the same scenario for debugging. Even if you have all the inputs stored somewhere, it might be very time consuming to recreate the same scenario by running the program in `gdb`.

Exercise 17.1. Generate a core dump from the following. Use `gdb` to analyze what happened. Fix the program. Of course if you are a strong programmers, you can figure out the bug very quickly and without `gdb`. Just pretend you don't see it. Walk through the function frames and print the values in scope before fixing the bug.

```
#include <iostream>
#include <string>

class SLNode
{
public:
    SLNode(int key, SLNode * next)
        : key_(key), next_(next)
    {}
    int key_;
    SLNode * next_;
};

std::ostream & operator<<(std::ostream & cout, const SLNode & node)
{
    cout << node.key_;
    return cout;
}

class SLList
{
public:
    SLList()
        : phead_(NULL), size_(0)
    {}
    // WARNING: No destructor, copy constructor, operator=
    void insert_head(int key)
    {
        phead_ = new SLNode(key, phead_);
        ++size_;
    }
    SLNode * phead_;
    size_t size_;
};

std::ostream & operator<<(std::ostream & cout, const SLList & list)
{
    cout << '{';
```

```
    std::string delim = "";
    SLNode * p = list.phead_;
    for (size_t i = 0; i <= list.size_; ++i)
    {
        cout << delim << (*p);
        delim  = ", ";
        p = p->next_;
    }
    cout << '}';
    return cout;
}

int main()
{
    SLList list;
    std::cout << list << '\n';
    for (int i = 42; i < 50; ++i)
    {
        list.insert_head(i);
        std::cout << list << '\n';
    }
    return 0;
}
```