What is GDB?

“GDB, the GNU Project debugger, allows you to see what is going on `inside' another program while it executes -- or what another program was doing at the moment it crashed. GDB can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act:

- Start your program, specifying anything that might affect its behavior.
- Make your program stop on specified conditions.
- Examine what has happened, when your program has stopped.
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.”
- Sourceware

How do you use GDB?

Basic usage

Running gdb requires a binary file of your program with debugging symbols. To get the debugging symbols, you need to compile your program with the “-g” flag. Otherwise, when debugging, function names, variables, and other things will not show up properly. You might see “??” instead of the expected names or variables.

Once you’ve compiled with the debugging symbols enabled, simply type “gdb <your_program>”. For example, if your program name isHelloWorld, you’d simply type, “gdb HelloWorld”.

Common useful commands

- run <arg1> <arg2> … <argN>
  - runs your program with any command line arguments.
- break <line number or function name>
  - adds a breakpoint to signal gdb to pause at a given line.
- print <variable_name>
  - prints the value held in a variable.
- next
  - step OVER this line of code. Just run this line of code.
- step
  - step INTO this line of code. Let me debug inside of the function on this line.
- backtrace
  - print the frame leading up to this spot in your code. Shows the function call that led to this line (includes function name and parameters).
- list
  - display lines of code from the source file.

Trivial example

Using the studentList.c, studentList.h, and makefile. Comments/explanations in red.

```
jmata@ubuntu:~/demo/linkedlist$ gdb studentList     Run the debugger
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from studentList...
(gdb) break main   We create a breakpoint starting at main()
Breakpoint 1 at 0x137d: file studentList.c, line 55.
(gdb) run  The code is run until we get to the breakpoint at main
Starting program: /home/jmata/demo/linkedlist/studentList

Breakpoint 1, main (argc=21845, argv=0x0) at studentList.c:55
55      int main(int argc, char** argv){
(gdb) list  We print out some lines centered at where our debugger is stopped at (line 55).
50          printf("%s\n", currNode->name);
51        }
52      }
53
54
55      int main(int argc, char** argv){
56        struct LinkedList* myList = init(myList);
57
58        myList->head = NULL;
59
(gdb)
```

```
(gdb) next
56        struct LinkedList* myList = init(myList);
(gdb) next  run init(myList) to initialize head/tail memory allocations and pointer updates. We will step OVER it as we will trust it to just work.
58        myList->head = NULL;
(gdb) next set myList->head as null (for demonstration purposes). This WILL cause a segfault
60        insert(myList, "James");
(gdb) step  step INTO the insert() function. We want to see what happens inside.
insert (self=0x555555555201 <init+56>,
    student=0x5555555550e0 <_start> "\363\017\036\372\061\355I\211\321^H\211\342H\203\344\360PTL\215\005\246\003") at stu
29      int insert(struct LinkedList* self, char* student){
(gdb) next
30        struct Node* currNode = self->head;
(gdb) next  Run the first line of the function. This just creates a local pointer.
33        while (currNode->next != self->tail &&
(gdb) print self  Print the value of "self", which is a pointer to a node struct object, so an address is printed.
$1 = (struct LinkedList *) 0x5555555592a0
(gdb) print self->head    Print the value of self->head, which should be a pointer to a node object. Notice it returns 0 (a very low memory
$2 = (struct Node *) 0x0 address which we cannot access with our program). This will cause a segfault if we ever dereference the head object.
(gdb)
```

```
(gdb) next
            We set currNode = self->head, so as soon as we try to dereference it, we segfault.
Program received signal SIGSEGV, Segmentation fault.
0x00005555555552b9 in insert (self=0x5555555592a0, student=0x555555556004 "James") at studentList.c:33
33        while (currNode->next != self->tail &&
```

Core Files

What is a core file?

"A *core file* or *core dump* is a file that records the memory image of a running process and its process status (register values etc.). Its primary use is post-mortem debugging of a program that crashed while it ran outside a debugger. A program that crashes automatically produces a core file, unless this feature is disabled by the user."

- Sourceware

A core file is generated when you have a segfault. Instead of rerunning your code line by line, you can run your core through gdb directly to start at the exact line where the segfault occurred. This is one of the more convenient ways to debug a segfault.

How do I use the core file to find my error?

After your program crashes, and you have verified the core file is generated (see below if it isn't), run gdb with your core file. The syntax is "gdb <file_name> <core_name>". You can then print the variables and run a backtrace to see what calls were made to get the program to the point of failure.

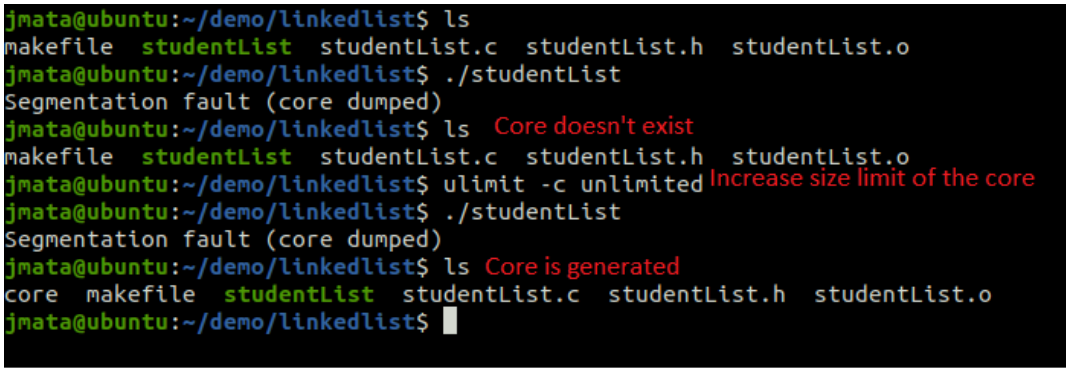
How do I generate a core (dump) file?

You need 2 things:

1. Enable debugging at compile time. When you compile, add the "-g" flag to generate symbols so your functions/variable names show up.
2. Check the maximum core size allowed on your current shell.

When your program crashes via a segfault and the core dump isn't automatically generated, your current shell process may be limiting the core dump size. You can check what the current allowed size is by running "ulimit -a" and looking for the line indicating core file size. If it's set to 0, then run "ulimit -c unlimited".

Now, just run your code again and verify the core file is in your program's directory.



Trivial example with the core file