How to Use gdb

Brad Vander Zanden

- gdb cheat sheet: contains quick summary of useful gdb commands.
- gdb does not run well on Macintoshes and may not run well on Window machines. I suggest that you use the hydra machines when you want to use gdb.

---

gdb is a GNU debugger that can be very helpful in finding problems with your programs. It allows you to do a number of useful things including:

1. controlling the execution of your program by placing breakpoints and single stepping your program.
2. printing the values of variables in your program, and
3. determining where a segmentation fault or bus error occurs in your program.

In CS102 you probably debugged your program by inserting print statements into your program. While that technique often proves helpful, gdb provides far more functionality and is worth learning. We will start you out with a simple tutorial that shows you some of its features and then show you how you can use it to help you with debugging.

In order to use gdb you need to compile your files with the -g option. For example:

```
gcc -g -c student1.c
gcc -o student1 student1.o
```

A Sample GDB Session

Start by copying the files in ~bvz/cs140/gdb to your directory and then typing **make**. `make` should compile the sample files in your directory into executables.

Now type:

```
./names_list
```

`names_list` prompts the user for five names and prepends the names to a list. The list starts with a sentinel node. The program then prints the names in the reverse order from which they were entered. In this case `names_list` has nothing

wrong with it. We simply want to use it to show you the various ways in which you can use gdb. After that we will show you how you can use gdb to debug programs.

To use gdb with `names_list`, type

```
gdb names_list
```

gdb is the command you type to invoke the gdb debugger and `names_list` is the name of the executable you wish to debug.

Listing the lines in a program

You can list the lines in your program using the `list` command, or `l` for short. `list` lists 10 lines of your program, centered about the next line to be executed. If you do it now, it will probably list the first 10 lines in your file since your program has not yet started execution. You can make it

```
l first_line_number, last_line_number
```

For example, typing `l 10,23` will cause gdb to list lines 10-23 of the current file.

Sometimes you will want to list the lines in a different file. For example, `names_list` consists of the two files `list.cpp` and `add_name.cpp`. The file that was listed defaults to `list.cpp` because that was the first file I gave to gcc. However, I can look at lines in `add_name.cpp` by prefixing the line numbers with add_name.cpp. For example:

```
l add_name.cpp:6,12
```

Executing a Program in GDB

You start running a program using gdb's `run` command or `r` for short. Try it now on the `names_list` program. Simply type `r` at the gdb prompt:

```
(gdb) r
```

`names_list` does not require any command line arguments but if you have a program that requires command line arguments, you list them after the `run` command. For example:

```
r 3 gradefile
```

If your program is seg faulting then the run command will probably suffice to start your debugging session. When the program seg faults gdb will tell you the file, the procedure, and the line number where the seg fault occurred. We will explore debugging later. For the time being we will concentrate on showing you how you can control the execution of a program using gdb.

Frequently you will want to stop the program at some point during its execution and start single-stepping it. You can cause the program to stop at a certain line or when a certain function is called by creating breakpoints. A breakpoint is created using the `break` command or `b` for short. It takes either a line number or a function as an argument. For example, each of the follow commands will set a breakpoint in the names_list program:

```
b 15                  // set a breakpoint at line 15 in the current file
b add_name_to_list    // set a breakpoint on the function add_name_to_list
b add_name.cpp:8      // set a breakpoint at line 8 in add_name.cpp
```

When you place a breakpoint on a function, such as `add_name_to_list`, gdb will stop the program any time that the function `add_name_to_list` is called, no matter where in the program `add_name_to_list` is called.

When a breakpoint is created gdb will assign it a number. If you want to later delete the breakpoint you can do so by typing `delete` and the breakpoint number. Alternatively you can type `clear` and either the line number or the function to which the breakpoint is attached. For example:

```
delete 1
clear add_name_to_list
clear add_name.cpp:8
```

These three commands will delete the three breakpoints created earlier.

If you forget what breakpoints you have set, you can type `info b` to print the breakpoints. For example:

```
(gdb) info b
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000000400b2b in main() at list.cpp:8
(gdb)
```

The `Disp` field is short for `Disposition` and indicates whether the breakpoint is marked to be disabled or deleted when it is next encountered. The `Enb` field is short for `Enabled` and indicates whether or not the breakpoint is currently enabled. You do not need to worry about either of these fields. The `What` field tells you where the breakpoint is set, and that information, plus the number of the breakpoint, are probably the two pieces of information you need.

Continuing from a Breakpoint

You can cause the program to continue executing and proceed to the next breakpoint by typing `continue`, or `c` for short. Try it out by re-instating the breakpoints you typed in previously, re-starting the program, and then typing `c` each time you reach a breakpoint. You will need to re-type the previous breakpoints if you deleted them.

Printing the Values of Variables

You can print the values of variables in gdb using the `print` command, or `p` for short. You print items just as you would reference them in a C program. However, unlike printf, you do not have to provide formatting information.

To try out the printing command first delete your previous breakpoints and place a breakpoint at line 21 in list.cpp. Now re-run your program from scratch and enter a name when prompted. The program will break at line 21:

```
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/bvz/courses/140/sp16/gdb/names_list
enter a name: brad

Breakpoint 1, main () at list.cpp:21
21              add_name_to_list(name_list, name);
(gdb)
```

Now let's print the contents of `name_list` and `name`:

```
(gdb) p name_list
$1 = (listNode *) 0x603010
(gdb) p name
$2 = "brad"
(gdb)
```

`name_list` is a pointer which points to memory starting at 0x603010 and `name` is the string that I just typed in at the prompt. We can see that "brad" was properly read into `name`. The memory address you see may be different than mine because your program may be allocated a different part of memory than mine was. Throughout this tutorial, you should expect your memory addresses to differ from the ones shown in this tutorial.

We can print the contents of the node pointed to by name_list by dereferencing the pointer:

```
(gdb) p *name_list
$3 = {name = "", next = 0x0}
```

Remember that our list has a sentinel node so everything is as it should be. The sentinel node's `next` field is the null pointer, thus making it an empty list.

If you want to examine all of the local variables in the function, you can type `info locals`:

```
(gdb) info locals
i = 0
name = "brad"
name_list = 0x603010
name_list_ptr = 0x400d10 <__libc_csu_init>
```

Note that `i` and `name_list_ptr` both have values, even though they have not been initialized. It would be nice if gdb would mark them as uninitialized, but you don't always get what you want in life. Since `i` and `name_list_ptr` are uninitialized, you can't count on them always having reasonable values. For example, when I tried to see what memory was pointed to by `name_list_ptr`, I got the following:

```
(gdb) p *name_list_ptr
$4 = {
  name = , next = 0x41d589495541f689}
```

In other words, `name_list_ptr` does not point to a legitimate memory object.

Let's continue with our program's execution. Type `continue`, or `c`, and enter another name:

```
(gdb) c
Continuing.
enter a name: nels

Breakpoint 1, main () at list.cpp:21
18              add_name_to_list(name_list, name);
(gdb)
```

At this point "brad" should have been added to the list but "nels" should not yet have been added to the list. However, "nels" should be contained in name. We can check all this out using the print command:

```
(gdb) p name
$16 = "nels"                        // "nels" is in name as it should be
(gdb) p *name_list
$5 = {name = "", next = 0x603090}  // the sentinel node points to something
(gdb) p *name_list->next           // and that something is "brad"
$6 = {name = "brad", next = 0x0}
(gdb)
```

Note that brad's node does not point to anything, which is correct. Also note how I was able to "chase" a pointer by typing

```
p *name_list->next
```

Let's try one more iteration of the program:

```
(gdb) c
Continuing.
enter a name: pat

Breakpoint 1, main () at list.cpp:21
18              add_name_to_list(name_list, name);
(gdb) p *name_list->next
$7 = {name = "nels", next = 0x603090}      // the first node is "nels"
(gdb) p *name_list->next->next
$8 = {name = "brad", next = 0x0}           // the second node is "brad"
(gdb) p name
$9 = "pat"                                 // "pat" is in name
(gdb)
```

Notice that I can get arbitrarily far in the list by typing a series of `next's`.

Single Stepping Through a Program

Once you have gotten a program to stop you will often want to single step through the program, executing one statement at a time, rather than continuing to the next breakpoint. gdb provides you with two commands for doing so:

1. `step` (`s` for short): step executes each statement and, if it encounters a function call, it will step into the function, thus allowing you to follow the flow-of-control into subroutines.

2. `next` (`n` for short): next also executes each statement but if it encounters a function call it will execute the function as an atomic statement. In other words, it will execute all the statements in the function and in any functions that that function might call. It will seem as though you typed `continue` with a breakpoint set at the next statement. The one exception to this statement is that if there is a breakpoint nested in the function, then gdb will break when it reaches that breakpoint.

To see the difference between the two types of commands, re-run your program from scratch by typing `r`. It should break after it prompts you for a name and you enter one:

```
(gdb) r
Starting program: /home/bvz/courses/140/sp16/gdb/names_list
enter a name: brad

Breakpoint 1, main () at list.cpp:21
21              add_name_to_list(name_list, name);
```

First try using the `step` command:

```
(gdb) s
add_name_to_list (names=0x603010, name_to_add="brad") at add_name.cpp:7
7           listNode *new_node = new listNode;
(gdb)
```

We have stepped into the function `add_name_to_list` and are about to execute the first statement. Before executing this statement try printing `new_node`. Since it has not been initialized, it will have a random value. When I printed the value of `new_node` I got the following result:

```
(gdb) p new_node
$9 = (listNode *) 0x400a30 <_start>
```

Now execute the first statement in `add_name_to_list` by typing `s`:

```
listNode::listNode (this=0x603090) at list.h:5
5       class listNode {
```

What's happening? gdb is executing the first statement in `add_name_to_list`, which

is line 7:

```
listNode *new_node = new listNode;
```

Note the call to `new`, which creates a new `listNode`. When a new `listNode` is created, its constructor gets called. This call to the constructor is what is being printed by gdb when you see:

```
listNode::listNode (this=0x603090) at list.h:5
```

The `this` variable is an implicit parameter passed to every constructor that gives the address of the newly allocated object.

Type `s` again:

```
(gdb) s
add_name_to_list (names=0x603010, name_to_add="brad") at add_name.cpp:9
9           new_node->name = name_to_add;
```

Since the constructor for `listNode` is empty, we immediately step out of the constructor and back into `add_name_to_list` at its next executable statement, which is line 9.

Try printing out the new value of `new_node` and the contents of the memory pointed to by `new_node`:

```
(gdb) p new_node
$10 = (listNode *) 0x603090
(gdb) p *new_node
$11 = {name = "", next = 0x0}
(gdb)
```

Notice that `new_node` has the same memory address as the `this` variable in `listNode`'s constructor. That's because `0x603090` is the memory address of the object allocated by new. It was the memory address passed to the constructor and also assigned to `new_node`. Also note that the contents of `new_node` have reasonable values. string objects are automatically initialized to the empty string but it is purely luck that `next` is a null pointer as opposed to a random memory address. It's possible that when you run your own program, that `next` will point to something random.

Here is a helpful shortcut you can use while single-stepping. Type `s` to execute line 9. Now hit return a couple times until you reach statement 12 with the empty right curly brace, }. Notice that it single-steps for you. gdb remembers the last command that you entered and if you hit return, it will repeat that command. This feature is handy when you want to single-step through a number of statements.

You should now be at line 12, which is the end of the function. Let us make sure that the new node was prepended to the list and that it has the appropriate memory contents:

```
(gdb) p *names                    // print the sentinel node for the names list
$1 = {name = "", next = 0x603090} // it points to our new node
(gdb) p *names->next              // now look at the first node in the list
$2 = {name = "brad", next = 0x0}
```

Everything looks fine. You might wonder why we indirectly checked the contents of the new node by typing `p *names->next` rather than by typing `p new_node`. To see why, try to print new_node:

```
(gdb) p new_node
No symbol "new_node" in current context.
```

This seems odd. Why is `new_node` not defined? The reason is that we have exited the scope of the block defined by the initial { in the function. When we exited this block, `new_node` went out of scope. We are still in the function, albeit at its end, so we can still see the function's parameters. That's why we used the `names` pointer to check everything.

Let's continue by typing `c`

```
(gdb) c
Continuing.
enter a name: nels

Breakpoint 1, main () at list.cpp:21
21              add_name_to_list(name_list, name);
(gdb)
```

This time we will single step by using `next` rather than `step`:

```
(gdb) n
18          for (i = 0; i < NUM_NAMES; i++) {
(gdb)
```

Notice that we have executed `add_name_to_list` without stepping into it and have now gone to the top of the loop. Let's make sure everything worked before we continue:

```
(gdb) p *name_list                // print the sentinel node so we can find the first
                                  node in the list
$3 = {name = "", next = 0x6030e0}
(gdb) p *name_list->next          // the first node should be the new node
$4 = {name = "nels", next =
0x603090}                         // it is
(gdb) p *name_list->next->next    // now check the second node
$5 = {name = "brad", next = 0x0}  // it's also ok
```

Note that the `next` field for the new node points to the previous node we prepended. If you do not believe me, check the memory address, which is 0x603090, with the memory address that was given to new_node the first time we visited `add_name_to_list`. They are the same.

Instead of typing continue, try typing `n` a couple more times until cin gets executed:

```
(gdb) n
19              printf("enter a name: ");
(gdb) n
20              cin >> name;
(gdb) n
enter a name: sue
```

First, note that the prompt string is not printed until you execute the cin statement. That is because C++ buffers output until 1) a newline character is encountered, or 2) the output exceeds a certain length, or 3) something occurs that forces the output to be printed, such as a cin statement.

Also note that if you are using an older version of gdb, that you could get an error message when you entered "sue". The problem with the older versions of gdb is that they will only give THE FIRST CHARACTER to cin and then they will try to interpret the rest of the character string. If you encounter this problem, you can circumvent it by placing a breakpoint **after** the statements which read keyboard input and "continue" through them. Doing so will allow all the keyboard input to be read in before control is returned to you. Notice that I cleverly told you to place your breakpoint at line 21, which is the first statement after the cin.

Determining Your Location in a Program

You can find where you are in a program using the `backtrace` command or `bt` for short. The backtrace command will show you the current stack of functions that are active. For example, re-create the breakpoint at line 10 of `add_name` and then run the program until it reaches the breakpoint:

```
(gdb) b add_name.cpp:10
Breakpoint 2 at 0x400ccb: file add_name.cpp, line 10.

(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/bvz/courses/140/sp16/gdb/names_list

enter a name: brad

Breakpoint 1, main () at list.cpp:21
21              add_name_to_list(name_list, name);
(gdb) c
Continuing.

Breakpoint 2, add_name_to_list (names=0x603010, name_to_add="brad")
    at add_name.cpp:10
10          new_node->next = names->next;
(gdb)
```

Now type `backtrace`, or `bt` for short:

```
(gdb) bt
#0  add_name_to_list (names=0x603010, name_to_add="brad") at add_name.cpp:10
#1  0x0000000000400b98 in main () at list.cpp:21
(gdb)
```

It tells you that you are currently in `add_name_to_list` at line 10 in `add_name.cpp` and that `add_name_to_list` was called from `main` at line 21 in `list.cpp`. Note that the backtrace command also tells you the values of the arguments passed to each active function. `backtrace` is typically the first command you will type when debugging a seg fault or bus error because you will want some idea of where you are in the program.

You can use the `up` and `down` commands to move around in the stack. These commands are necessary if you want to print a variable that is local to another function. For example, if I try to print `main's name` variable while I'm stopped at the current breakpoint, I should get the following error message:

```
(gdb) p name
No symbol "name" in current context.
(gdb)
```

although you might also see something like:

```
(gdb) p name
$7 = '\000'
```

which is nonsense. However, if I type "`frame 1`" I will be moved up to the stack record for main. I can then print the value of `name`:

```
#1  0x0000000000400b98 in main () at list.cpp:21
21              add_name_to_list(name_list, name);
(gdb) p name
$8 = "brad"
```

Debugging Programs

Now that you've learned the basics of gdb, let's see how you can use it to debug a program. Quit gdb and then re-enter it using the program `segfault1`:

```
(gdb) q
The program is running.  Exit anyway? (y or n) y
cetus3> gdb segfault1

(gdb) r
Starting program: /home/bvz/courses/140/sp16/gdb/segfault1

Program received signal SIGSEGV, Segmentation fault.
0x0000000000400604 in main () at segfault1.cpp:6
6           *number = 10;
```

gdb tells us that we received a segfault at line 6 in segfault1.cpp. It also shows

us that the statement being executed was `*number = 10`. The fact that we are trying to assign to a dereferenced pointer should give us a strong clue that the pointer does not point to a legitimate piece of memory. We can test our hypothesis by checking the memory address pointed to by `number`:

```
(gdb) p number
$1 = (int *) 0x0
(gdb)
```

Sure enough, `number` is a null pointer, which means that we have not allocated memory for it using `malloc`. Try changing `segfault1.c` so that it calls the new operator to create a new int object and assigns it to `number` before trying to assign 10 to it. Re-compile and re-run the program. It should work and print out 10.

Debugging a Program with Array Overflow

In CS102 and perhaps even in CS140 you have encountered problems with stepping past the end of an array and your program seg faulting. gdb can help you determine where the program is seg faulting and by doing some investigating, it might help you determine the source of the error.

Try running `bad_names_list`. It has an array big enough to hold 3 names. Try entering 4 names:

```
hydra4.eecs.utk.edu> gdb bad_names_list

(gdb) r
Starting program: /home/bvz/courses/140/sp16/gdb/bad_names_list

enter a name: brad
enter a name: peggy
enter a name: sue
enter a name: jack

Program received signal SIGSEGV, Segmentation fault.
0x00007ffff7b8fd75 in ?? () from /lib64/libstdc++.so.6
```

What went wrong? The first thing we should do is try to find what line was executing in our program when the program crashed. Note that the program seg faulted in a function called `/lib64/libstdc++.so.6`. This is clearly not a function that we wrote. Use the backtrace command to print the stack:

```
(gdb) bt
#0  0x00007ffff7b8fd75 in ?? () from /lib64/libstdc++.so.6
#1  0x00007ffff7b9103e in std::string::assign(std::string const&) ()
   from /lib64/libstdc++.so.6
#2  0x0000000000400c06 in main () at list1.cpp:17
```

We can see that we were executing line 17 in `list1.cpp` when the program crashed. Let's print this line:

```
(gdb) l list1.cpp:17
12          int num_names = 0;
13          int i;
14
15          printf("enter a name: ");
16          while (cin >> name) {
17              name_list[num_names] = name;
18              printf("enter a name: ");
19              num_names++;
20          }
21
```

We see that line 17 attempted to assign `name` to our `name_list` array. In these situations, it's best to print the names of all of our "suspects", which are all of our variables in the statement: `name_list`, `num_names`, and `name`:

```
(gdb) frame 2
#2  0x0000000000400c06 in main () at list1.cpp:17
17              name_list[num_names] = name;
(gdb) info locals
name = "jack"
name_list = {"brad", "peggy", "sue"}
num_names = 3
i = 0
```

Notice that I first had to type `frame 2` to get to the frame containing my main function. Then I typed `info locals` because it was easier than individually printing my three variables.

The resulting print-out may or may not lead you to the error. In this case the problem is that the `name_list` array can hold only three names, and I'm trying to assign a fourth name to it at index location 3 (the value of `num_names`). Since this location is past the end of the array, the code seg faults.

Debugging an Infinite Loop

gdb can also be useful in situations where the program does not crash, but instead has a logic error that causes unexpected output or unexpected behavior, such as an infinite loop. I have created a new file named `add_name1.cpp` that contains a logic error which introduces an infinite loop and compiled it with `list.cpp` into an executable called `bad_names_list1`. Like `names_list` it is supposed to read 5 names and then print them out in reverse order. However, when it reaches the fifth name, it instead goes into an infinite loop printing out the last name that we entered.

First try running the program and entering 5 names. You can type `Ctrl-D` to stop the program's infinite execution:

```
enter a name: joe
enter a name: frank
enter a name: nancy
```

```
enter a name: sue
enter a name: yifan
yifan
yifan
yifan
yifan
yifan
...
```

Let's run the program in gdb. In order to use gdb effectively in such situations, we need to form a hypothesis about what is wrong with the program. Since the program enters an infinite loop when it starts printing the list, a natural hypothesis is that something is wrong with our list of names. If we look in `list.cpp`, we find that the for loop for printing the names starts at line 24, so let's put a breakpoint at that line and then run the program until we reach that line:

```
hydra4> gdb bad_names_list1
(gdb) b 24
Breakpoint 1 at 0x400ba2: file list.cpp, line 24.

(gdb) r
Starting program: /home/bvz/courses/140/sp16/gdb/bad_names_list1

enter a name: joe
enter a name: frank
enter a name: nancy
enter a name: sue
enter a name: yifan

Breakpoint 1, main () at list.cpp:24
24          for (name_list_ptr = name_list->next;
```

Now let's start printing our list manually using gdb. We will start by printing the sentinel node:

```
(gdb) p *name_list
$2 = {name = "", next = 0x6031d0}
```

That looks fine--the name field is empty and `next` appears to be pointing to something. Now let's print the first node in the list:

```
(gdb) p *name_list->next
$3 = {name = "yifan", next = 0x6031d0}
```

The name field looks right--it's "yifan". Remember that we are prepending names to the list, so "yifan" should be the first node in the list. If you're paying close attention to what's been printed thus far, you may notice something funny about the `next` field. However, if you don't see anything wrong yet, then try printing the second node:

```
(gdb) p *name_list->next->next
$4 = {name = "yifan", next = 0x6031d0}
```

Hmm, that doesn't look right. The second node also appears to be "yifan". Maybe now is the time to look more closely at that `next` field. Look at the value of "yifan"'s next field and the value of the sentinel node's next field. Notice that

they point to the same memory address and this memory address must be yifan's node. In other words, yifan's node points back to itself rather than pointing to sue's node.

At this point we have confirmed our hypothesis that the list has been improperly constructed. We should also be able to understand why the for loop is an infinite loop that always prints yifan. At each step, the list pointer is being made to point to yifan, and hence the for loop never exits and the loop body always prints the string "yifan".

You should now suspect that there is a problem in the function `add_name_to_list`, which is located in `add_name1.cpp`. If you look at the code in that function, I hope you can figure out what is wrong by manually inspecting the code. However, suppose that you cannot do so. You can use gdb to help you figure out what's wrong.

To do so, you'll need to re-start the program and put a breakpoint on the `add_name_to_list` function:

```
(gdb) b add_name_to_list
Breakpoint 1 at 0x400c9f: file add_name1.cpp, line 7.

(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/bvz/courses/140/sp16/gdb/bad_names_list1

enter a name: joe

Breakpoint 1, add_name_to_list (names=0x603010, name_to_add="joe")
    at add_name1.cpp:7
7           listNode *new_node = new listNode;
(gdb)
```

The problem seems to be in how we link the list, so let's use the `next` command to skip ahead until we reach the first linking statement, which is statement 10. I've also printed `new_node` and the contents of `new_node` when I reach statement 10, just to make sure everything looks okay (it does):

```
(gdb) n
9           new_node->name = name_to_add;
(gdb) n
10          names->next = new_node;
(gdb) p new_node
$0 = (listNode *) 0x603090
(gdb) p *new_node
$1 = {name = "joe", next = 0x0}
```

If we inspect statement 10, we can see that it is making the list's sentinel node point to joe's node. That's fine--we want to prepend joe to the front of the list. Let's execute statement 10 and make sure it worked by printing the first node of the list:

```
(gdb) n
11          new_node->next = names->next;
(gdb) print *names->next
$2 = {name = "joe", next = 0x0}
```

So far, so good. Let's move on to the next linking statement, which is statement 11. statement 11 is supposed to make joe point to the next node in the list. Since the list was previously empty, joe should point to the sentinel node, which we will check by printing the contents of `new_node->next`:

```
(gdb) n
12      }
(gdb) print *new_node->next
No symbol "new_node" in current context.
```

ARGHHHHHHH! I hate it when gdb does this. We have reached the end of the function and gdb has already de-allocated the `new_node` variable. You have two choices here. First you can add a useless statement, such as a printf statement, to the end of the `add_name_to_list` function so that the function does not end at line 12. That way `new_node` will still be available for you to examine. However, I don't want to waste my time adding a new statement to `add_name_to_list` and re-compiling it.

I have a better idea. I know that joe is the first node on the list and that joe should point to the sentinel node. So the sentinel node should be the "second" node on the list (I put "second" in parentheses because the list only has one value and the sentinel node marks the end of the list). Therefore I can print the second node of the list as follows:

```
(gdb) print *names->next->next
$3 = {name = "joe", next = 0x603090}
```

Hmm. That does not look too good. The second node is "joe", not the sentinel node. Hopefully at this point we flash back to our earlier exploration with yifan and realize that joe must be pointing to himself. If you don't believe me, you can re-run the program, and before you execute statement 11, print the address of joe's object (you can do that by printing `new_node`). You will then find for certain that joe is pointing to himself.

You now know that statement 11 is not working properly. At this point gdb cannot help you any longer. It has given you all the information it can provide and you will have to puzzle out the problem from here. Hopefully you can. If you can't figure it out, the lab TA will cover the solution in lab.

A Final Debugging Tip

When something goes wrong in a program, it often helps to try to come up with the simplest possible set of inputs that can reproduce the problem. In `bad_names_list1` you had to enter 5 names before discovering that something is

wrong. Since the program expects 5 names, you can't make the input any shorter. However, you might realize that you could change the constant `NUM_NAMES` in `list.cpp` so that it is 1 rather than 5 and then see if the problem occurs with 1 input. Indeed it does, and that could help simplify your debugging task. In this case the problem was so simple that the simplification would not really help. However, in lab I see many of you start your testing by using Dr. Plank's test cases, rather than constructing your own simple examples and using the test editors that he provides you with. Only after testing your simple examples and getting your program to work with them should you move on to his test cases. When one of his test cases fails, you can often make your debugging job easier if you can simplify his test file by removing one or more commands. This is what I frequently do when I help you with one of his test cases. I keep removing commands until I have the simplest possible file that causes the same error. Once I have this simpler file, I start trying to find the source of the problem. The simpler file means that far fewer statements might be responsible for the bug and makes it easier for me to find your error.

Additional Help

You can locate additional information about gdb in three ways:

1. using the `help` command in gdb
2. typing `man gdb` in your unix shell
3. surfing the internet