

CS 351 Fall 2009

Lab 2 : GDB & Data structures with C

Lab Objectives

After completing this lab, you should have:

- A basic understanding of how to use `gdb` for debugging
- A better understanding of pointer manipulation and memory management in C
- A working singly-linked list implementation

1 Introduction

Pointer manipulation and memory management are two aspects of programming with C that frequently drive programmers to fits of rage. We'll be doing plenty of both this semester, so if you haven't had much experience with either, it's a good idea to get in some practice.

Working with data structures is traditionally prescribed as good practice for pointer/memory management novices, so in this lab you will be fleshing out a partial singly linked list implementation. Along the way, you'll hopefully get a sense of the "C way" of doing things, and a feel for proper pointer usage and memory management.

When you run into difficulties during your various programming assignments, as you likely will (and frequently!), you will have need of a good debugger. `gdb`, the gnu debugger, provides a rudimentary but powerful interface to a full featured C debugger. The first part of this lab will run you through some exercises with `gdb`.

2 Hand Out Instructions

Before working on this lab, be sure to update the lab repository you cloned in lab 1. You can do this with the following command (issued in your working tree):

```
git pull
```

If you run into issues with the pull, you might need to first commit your changes (and enter a commit message):

```
git commit -a
```

In this lab you'll be working under the `labs/gdblab` subdirectory of your repository. Before proceeding, the first thing you need to do is edit the `TEAMINFO` file there and enter your name and e-mail address. This is how I'll know who your submission is from, so **do it now!**

3 Part I: The GNU Debugger

We'll start with an examination of the GNU debugger (`gdb`), a powerful command line debugger available to you on the course server and on virtually all Unix-like systems. `gdb` will prove indispensable in many of our labs, and, as we'll be using output from `gdb` along with `gdb`-based syntax in future lectures, it behooves you to learn how to use it well.

3.1 `gdb` commands

The following table lists the `gdb` commands that we will be using most frequently. A `gdb` reference card has been posted on the course website's "Resources" section that you can refer to for other commands (and their options).

Command	Description
<code>run [args]</code>	Run the program (with optional arguments)
<code>print expr</code>	Evaluate and print the value of <code>expr</code>
<code>display expr</code>	Display the value of <code>expr</code> every time the debugger stops (e.g., after a step, or at a breakpoint)
<code>break location</code>	Set a breakpoint at <code>location</code> , which can be a line number, function, offset, or explicit address
<code>continue</code>	Continue execution of a program
<code>next/nexti</code>	Execute the next instruction (<code>next</code>) or the next machine instruction (<code>nexti</code>), skipping over function calls
<code>step/stepi</code>	Execute the next instruction (<code>step</code>) or the next machine instruction (<code>stepi</code>), stepping into function calls
<code>until location</code>	Execute the program until <code>location</code>
<code>backtrace</code>	Print out the current program stack trace
<code>x expr</code>	Examine the contents of memory at the address <code>expr</code>
<code>quit</code>	Quit <code>gdb</code> .

Most of the commands also have shortcuts that can be used in place of their full names. You will likely want to use 'p' for `print`, 'b' for `break`, 'c' for `continue`, 'n' and 'ni' in place of `next` and `nexti`, 's' and 'si' in place of `step` and `stepi`, and 'bt' instead of `backtrace`.

In addition to these commands, `gdb` also has an extensive online help facility, which you can access with the `help` command, followed by an optional topic name. The command `help` on its own will list the classes of commands for which help is available — you should explore this list and the other commands available to you after walking through the tutorials in this lab. The `info` command is another handy one for obtaining status information from `gdb` — we'll mention some of its uses in due course.

The `gdb` implementation also has a basic history mechanism, which we can use to navigate through previously issued commands using the up/down arrow keys. Another handy trick: hitting enter without typing anything will automatically execute the previous command.

3.2 Tutorial 1

For this tutorial, you'll want to `cd` into the `gdb/fact` directory, where you'll find the source code for a recursive factorial program in `fact.c`, the compiled binaries `fact-g.out`, and `fact.out`, along with the Makefile that was used to build them. You can recompile these programs at any time with

the `make` command. Note that `fact-g.out` was built by invoking `gcc` with the `-g` flag, which causes the compiler to store debugging information in the object file such as the source code itself, line numbers, variables, etc., in the binary. `fact.out` was compiled without debugging information.

Note that the empty boxes beneath selected questions are for you to take notes – you will *not* need to submit your answers to these questions for grading!

3.2.1 Basics: Breakpoints, Stepping, and Stack traces

Run the debugger on the “`fact-g.out`” by typing `gdb ./fact-g.out` at the Linux command line. This will take you to the `gdb` command prompt:

```
(gdb)
```

Issuing the `run` command at this point will start the program and run it to completion. What gets printed?

In order to halt execution and start the debugging process, we need to set a breakpoint. You can do this with either a filename/line-number combination, or by using labels (e.g., function names) in your program. Set a breakpoint on the `main` function with the command “`b main`”. Now try running the program again — it should stop at the first line of `main`. After halting execution, we can now start issuing commands for debugging purposes.

Try entering the following commands, and observe their output:

```
(gdb) info break
(gdb) list
(gdb) p i
(gdb) x i
(gdb) x /d &i
(gdb) bt
(gdb) p argv
(gdb) p argv[0]
(gdb) p (void *)argv[0]
(gdb) x argv[0]
(gdb) x /s (void *)argv[0]
(gdb) p fact
(gdb) x /i fact
(gdb) p fact(5)
(gdb) x fact(5)
```

After running the program with the breakpoint set, `gdb` halts execution at the first line of the `main` function. Does the first statement get executed? How can you tell?

Note that the argument for the `p` (`print`) and `x` commands can be any valid C expression. How do these two commands make use of the evaluated expressions differently?

Let's now step through the program. Using the `s` (`step`) command, continue stepping until you are in the first line of the `fact` function — you should see the following output:

```
(gdb) s fact (x=10) at fact.c:4 4 if (x <= 1) (gdb)
```

Since we know that the `fact` function makes a number of recursive calls, we can simplify our debugging by setting a breakpoint at the start of the function (`b fact`), and jump to successive calls with the `c` (`continue`) command. Do this a few times, and print out the stack trace again (`bt`). What useful information is contained in the stack trace?

We're don't really want to walk through the rest of the execution trace, as this program is not too interesting, but it's useful at this point to highlight another capability of `gdb` — we can use it to modify the program at runtime as we wish. In the middle of the set of recursive calls to `fact`, we can force the current function to return with a specified value, using the `return val` command. The `finish` command will tell `gdb` to ignore all breakpoints and simply execute to completion on the next `continue`. `quit` will exit `gdb`.

3.2.2 GDB scripts

`gdb` can also be used with a script file containing a list of commands to be executed in order by `gdb`, once it is started. A script file for working with the `fact` program might look as follows:

```
break fact
run
additional commands to be run
finish
c
quit
```

Given that we've saved our script to a file named "fact.gdb", then, we would invoke `gdb` on our program by entering the following command at the Linux prompt:

```
gdb -x fact.gdb ./fact-g.out
```

Exercise 1

Create a `gdb` script file to be used with the `fact-g.out` executable which will cause the program to terminate and print out the value 720. You can do this in any number of different ways, but your solution must allow the `fact` function to call itself recursively at least twice. Base your script on the listing above and save it as `fact.gdb` in the `gdb/fact` directory.

3.3 Tutorial 2

Now, we'll try debugging the object file compiled without debugging information. Start up the session with `gdb ./fact.out`. As is often the case, you'll want to start by setting a breakpoint, and running the program. Set a breakpoint at `main`, and use `run` to get things underway.

Now try viewing the instructions with the `list` command, and try stepping through the program as we did before with `step`. What happens? Why?



You can restart execution by re-entering `run` at any point, so do that now — this will bring you back to the first breakpoint you set at `main`. To view the assembly code for the function, you can use the command `disass main` — try this now (`disass` by itself will also work, as `gdb` will default to disassembling the current function). The `bt` command will still give you information about the current stack frame, but it will not be quite as verbose about it as before (with embedded debugging information). For more details about the current procedure frame, use the command `info frame`.

Using the commands just discussed, determine where the program counter points directly after execution to the breakpoint on `main` (recall, the PC in IA32 is register `eip`). Does it point to the first instruction in the function? How does `gdb` seem to interpret a breakpoint set to a function label?

Expressions used with the `p` and `x` commands can also involve registers — instead of using the `%` prefix, though, `gdb` requires that register names be prefaced with a `$` character. Try issuing the following commands:

```
(gdb) p $eip
(gdb) p /x *(int *)$eip
(gdb) x /3b $eip
(gdb) x /i $eip
```

What are the assembly and machine language (hex) renditions of the code that `%eip` currently points to?

When stepping through assembly code, `gdb` does not automatically print contextual information for us, so it is useful to make it do this. We can use the `display` command to have `gdb` display a particular expression each time it “stops” execution (for a breakpoint or after a step). The following command forces `gdb` to print out the next instruction each stop:

```
(gdb) display /i $eip
```

Test this out by stepping through a few assembly instructions using the `stepi` (abbrev. `si`) command. Play with this a bit — remember that you can view the contents of registers with `p reg`. To see the contents of all registers, you can use `info reg` (this will print out the contents of the segment and flag registers as well, which we haven’t yet discussed).

Now try setting a breakpoint on the `fact` function and use `c` to continue execution, as before. Take note the of the address (and offset from the `fact` function label) that this takes us to. What would you guess constitutes the “setup” portion of the `fact` assembly code?

If you wish to set a breakpoint at the “real” starting address of the `fact` function, you can set a breakpoint at an explicit address using the form `break *address` — note that the ‘*’ is necessary, or `gdb` will assume that you are specifying a line number. The same syntax (`*address`) can be used to provide an explicit address to the `clear` and `until` commands. The following session shows how I set up and run to a breakpoint on `fact` (the address of `fact` in your session may be different):

```
(gdb) display /i $eip
(gdb) break *0x08048384
Breakpoint 1 at 0x08048384
(gdb) run
Starting program: /home/lee/.../fact/fact.out

Breakpoint 1, 0x08048384 in fact ()
1: x/i $eip 0x08048384 <fact>: push %ebp
```

To clear a breakpoint, use the `clear` command — it’s argument should match the one given to `break` to set the breakpoint:

```
(gdb) clear *0x08048384
Deleted breakpoint 1
```

Although you may not have looked specifically at IA32 procedure call conventions before, you should be able to guess a good deal by just looking at the disassembled `fact` function. Which assembly language instructions perform the decrement and recursive call to `fact` (i.e., `fact(x-1)`)?



3.3.1 Exercise 2

Now go back and make sure that the `gdb` script you wrote for Exercise 1 works for this executable as well. If you put `-g` dependent `gdb` commands in your script, you may have to tweak it a little.

3.4 Fun with Linked Lists

Another neat thing about `gdb` is that it stores the results of all the `p` commands we issue so that we can reference them later on. You’ve likely already noticed these history value names, which are prefaced by a `$`, and whose names correspond to the count of `p` commands that have been carried out to that point. We can either reference these history values directly (e.g., with names `$1`, `$2`, etc.), or we can use some shortcuts provided by `gdb` — `$` on its own refers to the most recent history value, and `$$n` refers to the n^{th} value before the last.

The following session shows a number of `p` commands, and usage of history values:

```

(gdb) p $eip
$1 = (void *) 0x8048534
(gdb) p /x *(int *)$
$2 = 0xb8f0e483
(gdb) p /d $
$3 = -1192172413
(gdb) x /i $1
0x8048534 <main+6>:      and    $0xffffffff0,%esp

```

Armed with this knowledge, let us now load up the “list-g.out” binary into `gdb`. This particular executable was compiled with the ‘-g’ flag, so the command `list main`, will print out the `main` function’s source code.

Although we already know the contents of the list being constructed in `main`, we’ll see how to navigate manually through the data structure in `gdb`.¹ Most graphical debuggers can automatically do this for us, but it’s a useful skill to master. After setting a breakpoint at `main`, go ahead and step (you will want to use `n` here instead of `s` to avoid stepping into all the calls to `cons`) past the statement that creates all the list nodes.

Since `gdb` fully supports the use of C constructs and variables in the running program, getting at the items in the list is not so difficult — the following partial session should get us started:

```

(gdb) print l
$2 = 0x8049820
(gdb) x /s l->value
0x8049830:      "What"
(gdb) p l->next
$3 = (struct node *) 0x8049800

```

Notice the use of C’s ‘->’ operator in the `p` and `x` commands. It should be clear that if we continue following this pattern of usage, we can navigate through the rest of the list. But it turns out that `gdb` provides a `while` facility just for things like this — the following session demonstrates its usage:

```

(gdb) set $foo = l
(gdb) while ($foo)
  >x /s $foo->value
  >set $foo = $foo->next
  >end
0x8049830:      "What"
0x8049810:      "is"
0x80497f0:      "this"
0x80497d0:      "list?!"

```

Note that `while` stops looping when `$foo` (the conditional expression, in this case) is zero. `$foo` is an example of a “convenience” variable in `gdb` — we set the value of `$foo` (and simultaneously “declare” it) prior to entering the `while` construct, and retrieve its value just as we would a

¹Before proceeding, be sure to have read through the prescribed chapters in K&R — we haven’t covered C structs and typedefs yet in lecture, but you need them for this next exercise!

register or history value. The rules for naming convenience variables follow those for typical language identifiers, and they can not clash with reserved names (e.g., registers, history values). The `show convenience` command can be used to get information about convenience variables you've defined.

If we expect to need to iterate through lists often, it might pay to define our own user defined `gdb` command that does just that, using the `while` construct defined above:

```
(gdb) define iterate
Type commands for definition of "iterate".
End with a line saying just "end".
>set $n = $arg0
>while ($n)
  >x /s $n->value
  >set $n = $n->next
>end
>end
```

In the definition for `iterate`, we were able to get at the argument provided to the user defined command using the `$arg0` variable — 10 such variables (`$arg0-$arg9`) are available in each user defined command.

We can now use `iterate` just as any other builtin `gdb` command:

```
(gdb) iterate 1
0x8049830:      "What"
0x8049810:      "is"
0x80497f0:      "this"
0x80497d0:      "list?!"
```

These examples should make it clear that `gdb` truly is a full-featured debugger and code browser — indeed, we've barely scratched the surface of what it can do. We end, then, with a final exercise:

3.4.1 Exercise 3

In the same directory as the last binary we `gdb`'d, you'll find the "secret_llist.out" executable. The source code from which it was compiled is identical to that of "llist-g.out", except for the contents of the list constructed in `main`. To make things more interesting, it was also compiled without the `-g` flag. Your goal is to come up with a `gdb` script that will automatically traverse and print the contents of the list. Save your script in a file named `llist.gdb` in the `gdb/llist` directory.

HINT: There is not too much to this, actually. The first thing you'll need to find is the address of the "first" node in the list. You'll want to take a look at the few statements that precede the call to the `free_list` function to find this address. You'll also need to figure out how to get at the elements in a particular node — since you can't do this using the `'->'` operator anymore (`gdb` no longer knows about the struct), you'll have to manually find a way to get at the addresses.

4 Part II: A Singly-Linked List Implementation in C

As you've already examined linked lists in detail in your prerequisite data structures course, we won't bother with a drawn out discussion of their virtues and shortcomings. We also won't go into the specifics of the algorithms for their manipulation. We will, however, jump right into the skeleton code provided to you for a singly-linked list implementation in C. You can find the file `l1ist.c` in the `l1ist` directory off of the main `lab1` root. Note that this is *not* the same `l1ist.c` file that was provided for the `gdb` section of this lab!

We start with the type definitions:

```
#define EMPTY_LIST NULL

typedef struct node {
    int data;
    struct node *next;
} l1node, *l1nodep;
```

`l1node` is our linked list node type, which consists of a `data` field that we will fix as an `int`, and a `next` field, which simply points to another list node. `l1nodep` will be used to designate pointers to linked list nodes. Fittingly, `NULL` will be used for empty lists.

A simple example illustrating the traversal of a list using our defined types follows — a straightforward implementation of the `length` function:

```
int length (l1nodep n) {
    int count = 0;
    for (; n != EMPTY_LIST; n = n->next, count++);
    return count;
}
```

A more interesting function is `push`, which treats the list as a stack, pushing a new value onto the front of the list, and moving the head to the newly inserted node.

```
void push (l1nodep *head, int data) {
    l1nodep n = (l1nodep) malloc(sizeof(l1node));
    n->data = data;
    n->next = *head;
    *head = n;
}
```

Note that in order to allow for modification of the list in the caller, a pointer to the head pointer, `l1nodep *head`, is passed. In this way, the last assignment, `*head = n`, can be used to proper effect.

Your job will be to implement the remaining empty functions found in `l1ist.c`, according to descriptions found in their accompanying comments. The function prototypes are duplicated below for reference:

```
/* Create a node with the new data, and append it to the end of the list */
void append (l1nodep *head, int data);

/* Returns a count of the number of times the data appears in the list */
int count_of (l1nodep head, int data);
```

```

/* Deallocates all the memory used by the given list, and sets its head to
 * NULL */
void freelist (llnodep *head);

/* The complementary function to push --- remove the head node of the list
 * and return its data */
int pop (llnodep *head);

/* Reverse the list by rearranging next pointers and moving the head
 * pointer (your solution should be iterative) */
void reverse (llnodep *head);

/* Sorts the given list in increasing order (the easier route to doing
 * this does not involve modifying any pointers) */
void sort (llnodep *head);

```

When you are ready to test your implementation, build an executable by running the `make` command – this will generate the binary named `l1ist` in the same directory, which will run the provided `main` function.

5 Submission

When you have finished with all your testing, simply run the following command (and enter the requested information) in the `labs/gdblab` directory to turn in your work. Make sure you’ve filled out the `TEAMINFO` file!

```
make handin
```

This will package your files and automatically send them to the `handin` server. If your `handin` is successful the `handin` script will print a message out to that effect and give you a submission timestamp. If the `handin` is unsuccessful, you’ll usually get back an error that indicates why – fix the problem and try again (or notify me if you can’t figure out why it’s not working).