**Synopsis -** In this lab you will be introduced to one of the most powerful tools in a software developer’s repertoire, the *debugger*. We will use the [GNU Debugger (GDB)](https://www.gnu.org/software/gdb/), an industrial strength command line debugger from the open source software community. Commit to the use of the debugger in all programming tasks in the course. Effective debugging is an essential hard skill in computer science.

*Debuggers* are useful for visualizing and reasoning about the active data in a program. An automatic instinct to immediately open a debugging session when a bug appears is a useful habit to develop. For now, we will focus on the basics.

**Linked list drawing -** Before any coding, students should produce a drawing based on their inspection of the provided linked list code framework. Looking strictly at the program output and source code. Drawings should include the same annotation and be similar to the class slides on linked lists, e.g. shows head and tail pointers, nodes are represented with individual node’s content: item, next and prev pointers.

**Lab 2 readings -** this lab has an assigned supplemental reading from zyBooks on memory management, the Big Three: destructor, copy constructor, and overloaded assignment operator,and other material related to this lab. You can access the reading for lab 2 here:

* [16.7 Memory leaks](https://learn.zybooks.com/zybook/UCRCS010CRusichSummerSessionB2021%20/chapter/16/section/7)
* [16.8 Destructors](https://learn.zybooks.com/zybook/UCRCS010CRusichSummerSessionB2021%20/chapter/16/section/8)
* 1[6.9 Copy constructors](https://learn.zybooks.com/zybook/UCRCS010CRusichSummerSessionB2021%20/chapter/16/section/9)
* [16.10 Copy assignment operator](https://learn.zybooks.com/zybook/UCRCS010CRusichSummerSessionB2021%20/chapter/16/section/10)

## Getting Started with GDB

Download the ***lab2\_list.tgz*** linked list archive from the Google Drive lab2 folder. Upload the file into a lab2 folder in your Cloud9 environment in AWS. Uncompress the framework, to reveal the following **source files: main.cc list.cc, list.h, node.h.** Before you compile and run the code, visually inspect the main function for a few minutes. The time should be just long enough to understand all code and its operation in the main function.

Next, draw on paper what you believe the two lists (first\_list and second\_list) and all of their elements look like after each operation is performed on them in the main. It is sufficient to have a single drawing showing each list's cumulative state after the main function is executed.

**Install GDB :** See [instructions](#_heading=h.fuvtc3jon0ud) at back of this document.

Next, compile the code with the following command:

$ g++ -o lab2.out -g -W -Wall -Werror -std=c++11 \*.cc

The **-g** flag will allow us to use gdb’s functionality, without it gdb is very limited on your compiled code. The **-W -Wall -Werror -std=c++11** flags print out more error messages when errors occur. The “\*” is a wildcard, so \*.cc will include all .cc source code files in the current directory. We could list them out manually, but \*.cc is much shorter. In Lab 1 our C++ source files were .cpp, but in Lab 2 we are using .cc, which is an alternative file extension for C++ source files.

Run the program. What happens? Why did it happen? Nothing is wrong with main.cc.

$ ./lab2.out

Now that we know there is an issue, we can find the cause by running the program through the debugger. You will also use the debugger to help you fill in your drawings with specific information. This information should include the value of a node, the memory address of a node, the memory address of its pointers, and the memory addresses of pertinent list items, i.e. head etc. Start gdb on your program:

$ gdb lab2.out

Then at the GDB prompt. i.e. (gdb) enter “run”:

*(gdb) run*

A GDB session is running. See “(gdb)” to the left of the command line now. Any commands will be directly sent to gdb, not our usual bash shell that we use to compile, navigate directories, and run other programs.

Let’s look at the **main.cc** in **Cloud9**. Open the main.cc file in edit mode. Record the line numbers in the main where operations of interest occur. For example, the assignment of one list variable to another list instance variable is of consequence. Specifically, write down the four line numbers for:

* when both lists are done being initially populated with push\_front(),
* when second\_list is set equal to first\_list,
* after the pair of push\_front()’s have been called,
* after the pair of push\_back()’s have been called.

We will be adding breakpoints directly after these events have occurred. Add a breakpoint for all these line numbers now. For example, if we are interested in the line:

34. second\_list() = first\_list();

Then we will add a breakpoint at line 35, because if we added a breakpoint at line 34, we will break at line 34 before it actually executes, so second\_list() will still be its original value during the break. In the first terminal window with gdb running add the breakpoint:

(gdb) break main.cc:35

After all of your breakpoints have been set, we can run the program. Do not be alarmed if the breakpoints occur at different spots then the ones you entered. The code will not break on empty lines and will skip ahead to the next non-empty, non-comment line of code.

(gdb) run

Now your code will execute until it hits the first breakpoint. The code will not continue until you use a command such as ***continue, step over, or step into***.

Before doing so, print out some information first. Check the current state of any variable during the break:

*(gdb) p first\_list*

*(gdb) p first\_list->head*

*(gdb) p first\_list->head->value*

*(gbd) p first\_list->head->next->value*

Experiment and make modifications to get the information needed to finish your drawing for all lists and nodes. Remember to fill in the memory addresses, not just the individual node values.

When you are finished with all the information at this breakpoint, proceed to the next one and collect all information relevant to the lists of that breakpoint. You will need to use either a continue, step over, or step into command. Use the GDB reference to find out how to use these commands!

Note that all three gdb commands have different behavior. A continue command will take you to the next breakpoint in the flow of execution. A step over will go to the very next line of code, even if it isn’t a breakpoint, but it won’t enter the function it steps on. A step into will do the same thing, but it will enter any functions it steps on. You may use multiple step over and step into commands to navigate line by line. This is a very slow process so adding breakpoints is recommended for most uses.

Did the outputted behavior of the program match your initial drawing? You should find that the lists do not behave in the way you originally expected from a simple visual inspection of the main function. *To remedy this, you will implement the missing parts for the list program. When done, verify using the debugger that your list class functions correctly.*

Resources

* **Shallow v. deep copy**:
  + [www.fredosaurus.com/notes-cpp/oop-condestructors/shallowdeepcopy.html](http://www.fredosaurus.com/notes-cpp/oop-condestructors/shallowdeepcopy.html)
* **Destructor**:
  + [www.fredosaurus.com/notes-cpp/oop-condestructors/destructors.html](http://www.fredosaurus.com/notes-cpp/oop-condestructors/destructors.html)
* **Copy constructor**:
  + [www.fredosaurus.com/notes-cpp/oop-condestructors/copyconstructors.html](http://www.fredosaurus.com/notes-cpp/oop-condestructors/copyconstructors.html)
  + <https://www.codeproject.com/tips/78946/c-copy-constructor-in-depth>
* **Overloaded assignment operator**:
  + [www.fredosaurus.com/notes-cpp/oop-overloading/overloadassign.html](http://www.fredosaurus.com/notes-cpp/oop-overloading/overloadassign.html)
* **Review GDB\_Quick\_Reference.pdf:**
  + <http://users.ece.utexas.edu/~adnan/gdb-refcard.pdf>

Reading the following sections.  
- Essential Commands

- Breakpoints and Watchpoints  
- Program Stack  
- Execution Control

* **GDB user manuals**
  + <http://www.yolinux.com/TUTORIALS/GDB-Commands.html>
  + <http://davis.lbl.gov/Manuals/GDB/gdb_9.html>

**Demo:** Demo your working code for, and show your completed drawings to your TA. You can demo your solution on Tue. June 29th in class for credit.

**Submission:** Submit your source code and drawing in pdf format as lab2.zip via Canvas.You must submit this by end of Monday June 28th.

**Rubric:**  
25 pts Initial Drawing (By visual inspection of main.cc)  
25 pts Augmented Drawing (Using GDB to fill in information)  
50 pts fix list class, add appropriate functionality to list class

### GDB: Instructions for installation:

Run the following commands in your lab 2 folder.

(1) Remove several other large packages that aren't used by the course. The following command frees up 734 MB of space:

$ sudo yum -y remove java-11-amazon-corretto-headless golang-bin docker

(2) Then, install the necessary gdb packages.

$ sudo debuginfo-install libgcc-7.3.1-12.amzn2.x86\_64 libstdc++-7.3.1-12.amzn2.x86\_64

(3) Type and run the following:

$ sudo debuginfo-install libgcc-7.3.1-13.amzn2.x86\_64 libstdc++-7.3.1-13.amzn2.x86\_64

Notes for (3): When you are prompted to, type y, then return (see below)

Total download size: 334 M

Installed size: 1.5 G

Is this ok [y/d/N]: