

# CS 455 Lab 12: C++ Debugger / Linked lists

Fall 2021 [Bono]

---

## Goals and Background

This lab will give you some practice with linked lists, along the way introducing you to a symbolic debugger to use with g++-compiled programs on Linux. You will need pencil and paper to complete the lab (as well as a computer :-).

The linked list aspect of the lab is to give you practice with reasoning about and writing linked list code, by dealing with some buggy linked list code. One focus of this lab is not debugging per se, but being able to predict a program's behavior from looking at the code; this will help you write correct code.

The lab instructions assume you will be using the gdb debugger. For easy reference while doing the lab, you may want to print out the help-sheet called "Getting started with gdb" that's linked in the next section.

**Note:** some of the functions here use PRE/POST style of function comments. You are already familiar with preconditions. A postcondition describes what is true after the function executes (that is, if the precondition is met before the call, the postcondition will be true after the call). Sometimes the postcondition needs to refer to the updated value of a reference parameter as well as its original value. To distinguish these two, we'll use ' to denote the updated one. E.g., in Exercise 3 below, the append function postcondition comment says:

```
// POST: list' is same as list, but with item appended
```

list refers to the value of the list *before* the call, and list' refers to the value of the list *after* the call.

## Reading and reference material

- [Getting started with gdb](#). This is also linked in the Documentation section of the web page. We'll refer to this here as the debugging handout.
- cplusplus.com tutorial on [Pointers](#) and [Dynamic memory](#), [Structures](#).
- Thurs 11/18 lecture on Pointers and Linked Lists

## Exercise 1 (1 checkoff point)

Take a look at the source code file `factors.cpp`.

`factors.cpp` is a program for calculating all of the factors of a number. The program does not quite work as is. Since you've seen it before, it will be easy to debug it: the goal here is to use gdb.

Compile and run the `factors.cpp` program. To make this easier for you, we've created a Makefile for you with rules for compiling all the code for this lab (we'll be learning more about make and Makefiles soon). To compile factors type

```
make factors
```

at the shell. This makes an executable called `factors`.

---

**More about make and Makefiles.** We will often put our compile commands in a Makefile, especially when we start compiling multi-file programs in C++, because the compile steps can get complicated. So, here we run

make, instead of g++ directly. make uses the rules in the Makefile to decide what needs to be done. You can look in the Makefile we're using here to see what command it runs (look for the line that says factors:)

---

Use gdb to find out where the program crashed. See the [debugging handout](#) for how to start up gdb, and for how to use it in this kind of situation. Use gdb and program examination to find the cause of the error, and fix it. The mistake causing the run-time error can be corrected by changing only *one* line.

Note: when the program crashes it may say "core dumped"; this means a huge file, called core containing the contents of memory at the time of the crash is now in your directory. In dire circumstances you could examine this file to help you debug, but usually you do not need it, so delete it. (Note: it looks like in Vocareum, the file doesn't actually get created.)

Recompile your new version and run it. (Hint: You can abort a program by typing ctrl-c. Use gdb again to help find the remaining error: You can abort the program from inside gdb to see where it is executing, then you can look at the values of variables. See the debugging handout for more details.

For check-off show the TA your working working factors program running in gdb. In particular show them how you set a breakpoint, run to the breakpoint, and then single step through the program for a few iterations, displaying the values of key variables.

## Exercise 2 (1 checkoff point)

Note: the beginning of this and the next exercise involves your stepping away from the computer and using pencil and paper. The best approach is probably for you and your partner to each draw the diagram independently, and then get together to reconcile the results.

```
// BUGGY sumList
// PRE: list is a well-formed list
// POST: returns sum of elements in list
int sumList (Node *list)
{
    Node *p = list;
    int sum = 0;

    while (p!=NULL) {
        sum += p->data;
        p->next = p;
    }
    return sum;
}
```

Draw a box-and-pointer diagram showing the state of sumList on entry with a non-empty list, and after each of several iterations of the loop.

The testlist.cpp program is a small interactive program to test several linked list routines, including sumList. You can compile it using the command `make testlist`. Note: don't be surprised that the "b"uild function of testlist builds the list in reverse order from the order the values are entered.

Compile and run the testlist program to test sumList and verify your appraisal of what it currently does. The command `make testlist` will work to compile it. Use gdb to verify your box-and-pointer diagram for the buggy sumList function. HINT: if you print a pointer in gdb you get its value, which is an address (in hex). To see what a pointer, p, points to, you can print \*p. If two pointers point to the same object, you will know this because they will have the same address.

For checkoff show the t.a. your diagram, describe what happened in gdb, and show them what the fix was.

### Exercise 3 (1 checkoff point)

```
// BUGGY append routine
// PRE: list is a well-formed list
// POST: list' is same as list, but with item appended
void append (Node * &list, int item)
{
    Node *p = list;

    if (p == NULL) {
        insertFront(list, item);
        return;
    }

    while (p->next != NULL) {
        p = p->next;
    }

    p = new Node(item);
}
```

On paper create a box-and-pointer diagram for append of an empty list, and append of a non-empty list.

Compile and run the program to test append and verify your appraisal of what it currently does. Use gdb if necessary to get further information.

Fix the problem, and create a box-and-pointer diagram for the corrected version showing the values of the variables at the end of the function.

For checkoff, show the TA your two diagrams, and the corrected version of the program.

### Exercise 4 (1 checkoff point)

Complete the implementation of the function printEveryOther (skeleton of body and call already appear in testlist.cpp).

Your implementation must only do about  $n/2$  iterations of the loop, where  $n$  is the length of the list. Your code must work for both even- and odd-length lists. Here's the skeleton:

```
// printEveryOther: prints 1st, 3rd, 5th, ... elements in a list
// PRE: list is a well-formed linked list
void printEveryOther(Node * list)
{
    while (list != NULL) {
        cout << list->data << " ";

        // YOU COMPLETE THE BODY OF THE LOOP
    }
    cout << endl;
}
```

---

### Checkoff for DEN students

In the README describe the behavior of each of the buggy functions (i.e., before you fixed them). Upload photos of your handwritten box-and-pointer diagrams into Vocareum (you should name them so it's obvious what they are). Submit your factors.cpp and testlist.cpp with all the work for the lab completed.

When you click the Submit button, it will be looking for and compiling (for source code) the files README, factors.cpp, and testlist.cpp.

---

*This lab is adapted from one written by Mike Clancy.*

---