CMPS 12M

Introduction to Data Structures

Lab Assignment 5

Objective

The purpose of this assignment is to get practice debugging C code and learn how linked lists work in C.

Overview

You are given a buggy implementation of a linked list in C (list.c). Your task is to fix the bugs so that the program works correctly. You'll start by practicing using Valgrind and GDB on a simpler file (malloc.c). Files for this lab can be found in:

`/afs/cats.ucsc.edu/users/r/nwhitehe/cmps12/lab5/`

Linked Lists

For this lab, the buggy program attempts to create a singly-linked list of integers. Each node in the list is made up of a "struct node" and contains an integer and a pointer to the next node. A global "head" variable keeps track of the head of the list, initially NULL to indicate the empty list.

Possible Problems in C

Reading or writing to a NULL pointer is always a bug.

Variables in C can be declared separately from being initialized. Uninitialized variables are variables which are declared but whose value is used before being assigned to. These values are implementation defined and can cause severe problems. Any use of an initialized variable is a bug.

Memory leaks occur when memory is allocated but not freed when no longer needed. C does not have a garbage collector so the memory will be wasted. The operating system usually cleans up after processes terminate to prevent memory leaks in one program from permanently wasting memory. For short running programs memory leaks may not have much consequence, but for longer running programs they are serious bugs. Good programming practice is to never leak memory; in this assignment any memory leak is an error.

A dangling pointer is a pointer value used by a program that points to memory that was previously valid but has since been freed. Any access through a dangling pointer is a serious memory bug. Returning addresses of stack-allocated variables from functions is guaranteed to cause dangling pointers because the stack-allocated variable is freed as the function returns.

Valgrind

Valgrind is a powerful memory checking tool. It is simple to use. Simply compile your program with debugging symbols, then run "valgrind program". You will see extra debugging information that Valgrind provides, warning you about various types of memory issues such as uninitialized memory accesses, illegal reads and writes to memory, and possible memory leaks.

To try it out, look at malloc.c (available in the lab directory).

```
#include <stdio.h>
#include <stdlib.h>

struct node {
    int value;
    struct node *next;
};

int main(int argc, char *argv[]) {
    struct node *ptr = malloc(sizeof(struct node));
    ptr = malloc(sizeof(struct node));
    ptr->value = 19;
    ptr->next = NULL;
    printf("%p -> node{ %d, %p }\n", ptr, ptr->value, ptr->next);
    free(ptr);
    return EXIT_SUCCESS;
}
```

Try compiling the program with gcc and running it:

```
$ gcc -Wall -g -std=c99 malloc.c -o malloc
$ ./malloc
0xa82030 -> node{ 19, (nil) }
```

The "-g" option turn on debugging information, which improves the messages that Valgrind generates and makes debugging with GDB possible.

Now run the executable in Valgrind:

```
$ valgrind ./malloc
==18537== Memcheck, a memory error detector
==18537== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==18537== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright
info
...
```

For more information you can turn up the verbosity of Valgrind:

```
$ valgrind --leak-check=full ./malloc
==18761== Memcheck, a memory error detector
...
```

You should be able to see an obvious memory problem with the malloc.c file.

GDB

GDB is the Linux command-line debugger. Similarly to Valgrind, you invoke it on a version of your program that has been compiled with the "-g" option. Unlike Valgrind, GDB does not automatically run your program when started. Instead it takes you to a command line that lets you issue commands to GDB. You can set breakpoints, examine variables, single step your program, or run your program.

```
$ gcc -Wall -g -std=c99 malloc.c -o malloc
$ gdb ./malloc
GNU gdb (Ubuntu 7.7-0ubuntu3.1) 7.7
Copyright (C) 2014 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"
...
(gdb)
```

From the command line of gdb you can issue commands such as "run", "quit", "next".

Here is a series of commands to try while debugging malloc.

```
break main        Set a breakpoint in the main function
run               Start the program
print ptr         Show the value of a variable
print *ptr        Show the value of an expression (this is a bad memory access)
next              Single step to next line in current function
print ptr
print *ptr
next
next
next
print ptr
print *ptr
next
next
next
next              __libc_start_main is the startup code called by the OS
quit
```

Here is another run through:

```
break 12          Set breakpoint on line 12
run
print ptr->value
print ptr->next
list              Show source lines near point of execution
bt                Show stack trace (not very interesting)
```

Task

Now work on fixing list.c. Your fixed version should perform the intended actions of the original. Namely, it should:

1. Print out an empty list ""
2. Insert 42 at the start
3. Insert 17 and 10 at the start
4. Insert 18 at the start
5. Search through and remove the 10 element
6. Print out the resulting list "18 17 42"
7. Deallocate all memory properly

Keep a copy of the original list.c file. As you fix bugs, in the file BUGS record which line numbers you are updating from the original list.c, what bug existed, and a general idea of how you fixed it.

Your final solution should not have any uninitialized variables, dangling pointers, or memory leaks. Running valgrind on your program should show no errors or warnings.

What to Turn In

All files you turn in for every assignment and lab should begin with a comment block that includes your name, CruzID, class, date, filename, short description of the file's role in the assignment, and any special instructions related to the file. Also create a file called README. The README file should have the normal comment block, then list all the files being submitted (including itself) along with any special notes to the graders. If you do pair programming, both names and CruzIDs should appear as author and in the README file.

For this lab, submit the following files:

```
README
BUGS
Makefile
list.c
```

To submit, use the submit command.

```
submit cmps12b-nojw.f14 lab5 ...
```

Your makefile should have a default target that builds an executable called "list", along with phony targets "clean" that removes compiled object files, "spotless" that cleans up all built files, and "test" which runs the executable in valgrind.