CSE 4714 / 6714 – Theory and Implementation of Programming Languages
Lab 4 – Debugging and Pointers

Introduction

Pointers add an additional layer of complexity to your programs and significantly increase the challenge of debugging your programs. This lab activity is meant to give you some insight into how pointers work and also into how you might debug problems created by pointers.

In general, there are two ways that you can debug problems created by pointers--either use the debugger or use cout statements. The debugger is most helpful when your program seg faults because it can identify the line number where the seg fault occurred and this often gives you a good starting point for finding your bug. cout statements are most helpful when a variable has an unexpected value and you are trying to determine how that variable was assigned that value. The reason that cout statements are better in this situation is that you can use them to continuously divide the area of the program where the problem might lie in half. Every time you insert a cout statement you will know whether the problem lies before the cout statement or after the cout statement. Hence you can eliminate the portion of the program where you know the problem does not lie. This search procedure is much like binary search and you can hopefully locate the problem with $\log_2 n$ probes, where `n` is the number of lines in your program.

In contrast, if you try to use the debugger to find where a variable changes value, you are effectively using linear search, because you generally execute the program one statement at a time, which is much like linear search. Linear search takes much longer to find something than binary search, which is why you are better off using cout statements to try to find where in the code a variable has been assigned an unexpected value.

There are a number of problems that can arise with pointers, including:

1. Failure to initialize a pointer before using it: This problem can either cause your program to seg fault if the pointer's value is 0, or can cause you to access an unexpected portion of memory if the pointer's value is some random address in memory. The former problem is easier to deal with because your program will immediately crash with a segmentation violation. The latter problem is harder to deal with because your program will keep running but will be using a bad value in its computations. Here is an example of an uninitialized pointer:

   ```
   int *ptr;

   cout << "*ptr = " << *ptr << endl;
   ```

   The debugger is a good tool to use to locate the location of a bug when your program seg faults since it will give you the line number where the seg fault occurred.

2. Dangling pointers: A dangling pointer occurs when you have a pointer to a piece of memory that gets de-allocated by the `delete` command. When you later try to access the value of this memory, you may get garbage because the memory allocator may have already given this piece of memory to another pointer and this other pointer may have already changed the value of the memory to an unexpected value. Here is an example of the dangling pointer problem:

```
int *ptr1, *ptr2, *ptr3;

ptr1 = new int;
ptr2 = ptr1;
*ptr2 = 30;

delete ptr1;  // ptr2 still points to ptr1's memory
ptr3 = new int;  // new might give ptr3 the memory that used to
                 // belong to ptr1
*ptr3 = 50;

// You would expect this print statement to print 30 because
// that is the value you assigned to ptr2's memory. However,
// when you deleted ptr1, you returned ptr2's memory to the
// memory allocator and the memory allocator may have given
// this memory to ptr3. Hence it is possible that the following
// statement will print 50 rather than 30.
cout << "*ptr2 = " << *ptr2 << endl;
```

Dangling pointers are difficult to locate because your program will not crash when you try to use them. They simply will have an unexpected value.

Dangling pointers can also cause unexpected issues when you assign to them. Continuing the previous example, suppose your program executes this statement much later in the program:

```
*ptr2 = 100;
```

If `ptr3` was assigned ptr1's old memory, then this assignment statement will unexpectedly change the value of `*ptr3` to 100. You may then perform some computation and be surprised to find that `*ptr3` is 100. You can look endlessly in your code to try to find the place where you changed `*ptr3` to 100 and you will never find it, because the culprit is the assignment of 100 to the dangling pointer in `ptr2`.

Dangling pointers are among the toughest problems you will ever debug and it is tough to use the debugger to debug them. Typically, your best strategy is to insert cout statements into your program to try to pinpoint the place in your code where `*ptr3` becomes 100.

3. Memory leaks caused by failing to de-allocate a memory location when no variable points to it any longer. Here is a common problem:

```
int *ptr = new int;   // assign first memory block to ptr

ptr = new int;        // assign second memory block to ptr
*ptr = 10;
```

   Students think they have to initialize `ptr` when they declare it and so they assign it a block of memory. However, in their code they immediately assign it a second block of memory and never de-allocate the first piece of memory. This causes a memory leak because the first piece of memory is now inaccessible to the memory allocator and it will never be able to re-use this memory. In long running programs, your program may eventually run out of memory and terminate if you have too many memory leaks.

   The solution to the above problem is not to allocate the first memory block. There was no need to do so because you initialized `ptr` to a memory block before you tried to assign a value to it. Hence you should have written:

```
int *ptr;

ptr = new int;
*ptr = 10;
```

   Memory leaks are hard to locate because they do not cause a program to crash if the computation is correct, unless the program runs for a very long time. There are tools like valgrind that can help you detect memory leaks.

4. Making a pointer point to the wrong object. This bug often occurs once you start working with linked data structures and is something that the debugger can help you with.

A Simple Pointer to a Stack Variable

We are first going to look at a couple programs that help show the difference between pointers to named "stack" variables and pointers to unnamed "heap" variables or objects. Stack variables are local variables and parameters in a function that are allocated in a stack frame. They always have a name, such as "int x;". By contrast unnamed "heap" variables are allocated from the heap using the new operator. For example, "new int".

Take a look at pointer1.cpp:

```
int main() {
    int number;
    int *ptr;

    // cout << "value of ptr and *ptr before initializing ptr" << endl;
    // cout << "ptr = " << ptr << endl;
    // cout << "*ptr = " << *ptr << endl << endl;

    number = 10;
    ptr = &number;
    cout << "number = " << number << endl;
    cout << "address of number = " << &number << endl;
    cout << "ptr = " << ptr << endl;
    cout << "*ptr = " << *ptr << endl;

    // Note that no memory was dynamically allocated

    return(EXIT_SUCCESS);
}
```

This program makes `ptr` point to a stack variable named `number`. Try compiling and running[+] this program. Note the difference between the value of `ptr`, which is a memory address and is printed in hexadecimal notation, and the value of `*ptr`, which is the value at the memory address to which `ptr` points. Also note that the memory address to which `ptr` points is `number's` memory address, and hence `*ptr` prints the value of `number`.

Your turn

Now let's see what happens when you try to reference `*ptr` before you assign `ptr` a legitimate memory address. Uncomment lines 8-10.

```
    int *ptr;

    cout << "value of ptr and *ptr before initializing ptr" << endl;
    cout << "ptr = " << ptr << endl;
    cout << "*ptr = " << *ptr << endl << endl;
```

+ There are several programs (files with main functions) in this one directory. Therefore, there is not a makefile for the programs. You can run each program in Visual Studio Code using the **Run > Start Debugging** menu option.

Now compile and run the modified program. Your program will do one of two things depending on the previous value stored in `ptr`:

1. Seg fault: If `ptr` contains an invalid memory address, such as 0x0, then your program will seg fault.
2. Print garbage: If `ptr` prints to a valid memory address in your program's memory space, then you will get random values printed, which we call "garbage".

Take a screenshot to show which your program does: seg fault or print garbage.

A simple pointer to a heap variable

We are going to re-run our experiment except now we will be using a pointer to a heap variable. Take a look at pointer2.cpp:

```
int main() {

        int *ptr;

        // cout << "ptr before assigning it a block of memory" << endl;
        // cout << "ptr = " << ptr << endl;
        // cout << "*ptr = " << *ptr << endl << endl;

        ptr = new int;
        cout << "ptr after assigning it a block of memory but before" << endl
                << "assigning the block of memory a value" << endl;
        cout << "ptr = " << ptr << endl;
        cout << "*ptr = " << *ptr << endl << endl;

        *ptr = 30;
        cout << "ptr after assigning it a block of memory and after" << endl
               << "assigning the block of memory a value" << endl;
        cout << "ptr = " << ptr << endl;
        cout << "*ptr = " << *ptr << endl << endl;

        delete ptr;
        ptr = nullptr;

        return(EXIT_SUCCESS);
}
```

This program makes `ptr` point to a heap-allocated block of memory. We will often refer to this block of memory as an **unnamed variable** or **unnamed heap variable**. We print out the value of `*ptr` before and after we assign the unnamed variable a value. Before we assign it a value, it has garbage (i.e., a random value). Afterwards it has the value 30.

Take note of the memory address of this unnamed variable. Note that it's memory address is quite different from the memory address of the stack variable in the previous section. This is because stack memory and heap memory come from very different locations in your program's memory space. Typically stack memory comes from the end of your program's memory space (hence the larger memory address) and heap memory comes from the beginning of your program's memory space (hence the smaller memory address).

Your turn

First, uncomment lines 8-10 and see what happens when you try to reference `*ptr` before you assign `ptr` a legitimate memory address. Take a screenshot to document your finding.

Now we are going to introduce a memory leak into your program and see what it looks like. Modify `program2.cpp` as follows. I have highlighted the new code in blue:

```
ptr = new int;   // new line number 18
*ptr = 30;
cout << "ptr after assigning it a block of memory and after" << endl
       << "assigning the block of memory a value" << endl;
cout << "ptr = " << ptr << endl;
cout << "*ptr = " << *ptr << endl << endl;
```

Compile and run this program. Note that the initial unnamed variable referenced by `ptr` is different than the second unnamed variable referenced by `ptr` – their memory addresses are different. This program has a memory leak because this program never returns the first unnamed variable to the memory allocator using delete.

valgrind is a tool that can be used to detect memory leaks in programs. First install valgrind at the linux command prompt:

```
sudo apt install valgrind
```

If the name of your executable program is `pointer2`, you would run valgrind as follows:

```
valgrind ./pointer2
```

Here is the output when I run valgrind on the modified pointer2.cpp program:

```
==2786== LEAK SUMMARY:
==2786==    definitely lost: 4 bytes in 1 blocks
==2786==    indirectly lost: 0 bytes in 0 blocks
==2786==      possibly lost: 0 bytes in 0 blocks
==2786==    still reachable: 0 bytes in 0 blocks
==2786==         suppressed: 0 bytes in 0 blocks
==2786== Rerun with --leak-check=full to see details of leaked memory
```

The results shown in the blue box are the lines that you want to pay attention to. The lost 4 bytes are the result of the program asking for space for 2 integers (`ptr = new int` on lines 12 and 18) and only returning the space allocated for 1 of the integers (line 25).

Remove the line you added that created the memory leak. Take a screenshot showing that your pointer2.cpp program no longer has a memory leak.

Another experiment with valgrind

Run the Project Part 3 Example parser for the Arithmetic Language with valgrind.

```
valgrind ./parse.exe
```

You should see something similar to:

```
==5046== LEAK SUMMARY:
==5046==    definitely lost: 0 bytes in 0 blocks
==5046==    indirectly lost: 0 bytes in 0 blocks
==5046==      possibly lost: 0 bytes in 0 blocks
==5046==    still reachable: 16,458 bytes in 3 blocks
==5046==         suppressed: 0 bytes in 0 blocks
==5046== Rerun with --leak-check=full to see details of leaked memory
```

Create a memory leak within the program. This can be accomplished by removing the call to delete the root of the parse tree in the `driver.cpp` file. Or a memory leak can be created by modifying one of the destructors in the classes that represent the parse tree nodes. Or a memory leak can be created by doubling up on the calls to allocate parse tree nodes in `parse.h`:

```
ExprNode* newExprNode = new ExprNode;
newExprNode = new ExprNode;
```

Choose one of the preceding methods and cause the program to leak memory. Take a screenshot showing the memory leak using valgrind. For example, when I remove the call to delete the root of the parse tree, this is the result:

```
==6353== LEAK SUMMARY:
==6353==    definitely lost: 56 bytes in 1 blocks
==6353==    indirectly lost: 376 bytes in 14 blocks
==6353==      possibly lost: 0 bytes in 0 blocks
==6353==    still reachable: 16,458 bytes in 3 blocks
==6353==         suppressed: 0 bytes in 0 blocks
==6353== Rerun with --leak-check=full to see details of leaked memory
```

After documenting the memory leak, revert the changes to the example code so that it does not leak memory.

Deliverable

Submit a report showing that you have performed this lab. The first page should be a title page with your name, the course number and name, the date, and a title for the report. The remaining pages should consist of the requested screenshots. The screenshots should be labelled and must be in the order given in the lab assignment. Each screenshot should be of a single screen. Do NOT show all of the monitors in a screenshot if you have multiple monitors. Screenshots that are not readable will not receive credit.

Grading

Report contains all requested items	20pts
Labelled screenshots	80pts

References

Original lab exercise written by Brad Vander Zanden
http://web.eecs.utk.edu/~bvanderz/cs140/debugging/lab7/