Debugging with Valgrind

Valgrind is a programming tool designed to help programmers (that's us!) find and fix memory mishandling in their code. On this page we will cover allocating and freeing memory, understanding Valgrind output, and fixing errors that are detected using Valgrind. Other guides have been provided on LearningSuite for debugging and understanding test driver output which you may find useful.

Managing Memory

Although at times it can be difficult to implement, the basic idea of memory management is actually pretty straightforward! Somewhere in the depths of your computer there is space set aside to temporarily store the variables and objects that you create and use while programming. Having a deeper understanding of how memory is handled (including understanding the heap and the stack) is very useful when programming. If it's been a while, it would be a good idea to review the difference between the heap and the stack before continuing on in this guide (youtube is a great resource for this).

Using "new"

Ok, so on to the good stuff. When we want to *dynamically allocate* an object we use the keyword "new" to do so. For example, if I had a class named "LinkedList" and I wanted to create a pointer to a dynamically allocated LinkedList I would say,

```
    LinkedList* myPtr = new LinkedList();
```

where the () after LinkedList contains the arguments to my LinkedList constructor. When I execute this statement a new space will be set aside in memory and the information contained in my LinkedList will be stored there.

Using "delete"

When I am done with any object that I create using "new" I need to let the computer know that I am no longer using that space in memory and that other information can be stored there. For example, if I was done with my LinkedList from above I could say

```
    delete myPtr;
```

and the space in memory where my LinkedList had been stored would be considered open for new writes.

There are two things to watch out for here. The first is that I can reassign myPtr at any time to point at another LinkedList object. This is very useful in practice, but it is dangerous because it means that I can lose track of the LinkedList that I created above. For example, if I said

```
LinkedList* myPtr = new LinkedList();

myPtr = NULL;
```

I would lose track of my LinkedList when I set the pointer to NULL. Once I've lost my LinkedList, there is no way for me to get it back. Thus, it is important to always have at least one pointer pointing to any object that I create using "new".

The second thing to remember is that calling "delete" on an object will not actually delete anything. The keyword "delete" does not change any of the information stored in memory, it merely lets the computer know that the memory reserved for the object I created can now be overwritten. For example, if I said

```
Node* myPtr = new Node(7); //creates a new Node with a value of 7

delete myPtr;

cout << myPtr->value << endl;
```

I would actually see the value "7" printed to the terminal. When I called "delete" on myPtr, I did not change the values stored in memory. I only gave my permission for the memory to be overwritten if the space was needed later. Note that if I had used a getter function like myPtr->getValue() instead of accessing the value directly my program would have crashed because the space in memory where my Node is stored is no longer considered a Node object.

Understanding Valgrind Errors

In order to prevent losing objects and accessing invalid data, Valgrind keeps track of which locations in memory are available and which have been invalidly accessed. Valgrind output can be very intimidating at first, but if you understand what it is trying to say it really can be very useful for finding and fixing bugs in your code.

```
Invalid read/write of size X
```

An invalid read occurs when you attempt to read the value of an object that has been deleted, or when you read a value from memory that you should not have access to (like going off the end of an array). In the example above where we created a Node with a value of 7 the last line

```
cout << myPtr->value << endl;
```

would cause an invalid read because in order to print the value of the Node we first need to *read* the value from the Node that we had previously deleted.

An invalid write most commonly occurs when we attempt to *write* to an object that has already been deleted. For example, if I said,

```
Node* myPtr = new Node(7);

delete myPtr;

myPtr->value = 6;
```

I would get an invalid write on the last line because I tried to write a new value (6) into the Node that I had previously deleted.

Invalid reads and writes can be fixed by making sure to set any unused pointer to NULL and being careful about the memory locations that you access in your code.

```
Conditional jump or move depends on uninitialised value(s)
```

Uninitialized value errors are reported when your program uses an uninitialized value to perform an operation (jump and move refer to assembly instructions). This can happen in if or while statements, but it can also occur at other times when you access a value that has not been initialized. For example, if I said

```
int x;

printf("x = %d\n\r", x);
```

I would get a conditional jump error because I tried to print the value of x before I gave it a value. A common source of this error is failure to initialize data members of your classes **in the class constructor**. This error can be solved by initializing all values (including pointers).

```
Invalid free()
```

Invalid frees / invalid deletes occur when you attempt to call "delete" on any space in memory that cannot be freed. The most common source of this error is deleting the same object twice. For example, if I said,

```
Node* myPtr = new Node(7);

delete myPtr;

delete myPtr;
```

I would get an invalid free error. This error can be fixed by being careful with your objects and making sure to delete them only once.

```
Mismatched free() / delete / delete []
```

Mismatched frees and deletes occur when you do not use the proper form of "delete" to delete an object. For example, if I make a new Node I would free the memory by saying

```
    delete myNodePtr;
```

whereas if I made a new pointer to an array of integers I would free the memory by saying

```
    delete[] myArrayPtr;
```

This error can be fixed by using the proper form of delete to delete the object that you create. As a general rule use "delete[]" to delete dynamically allocated arrays, and use "delete" for anything else.

```
8 bytes in 1 blocks are definitely lost in loss record 1 of 14
```

Memory leaks are caused when memory is allocated for an object but never freed. This happens when we create a new object and then never delete it. For example, if I wrote a main function that said,

```
    int main() {

    Node* myPtr = new Node(7);

    return 0;

    }
```

I would get a "memory leak" and Valgrind would report the number of bytes lost. This can be solved by deleting every "new" object that we create before our program terminates. A common source of this error is failing to delete allocated objects when the class destructor is called.

There are other errors and messages that Valgrind can print, but these are by far the most common. If you have any questions about what your error messages mean feel free to come in and talk to the TAs. See you around!