

LAB #4

Practice the Big Three and More File I/O

Remember, if you need to get Lab #3 graded, you need to show your lab to the TAs within 10 minutes of getting to lab, and you and your partner will not receive lab credit if you do not get checked off before leaving each lab. Once you have a zero on a lab, then it cannot be changed because we have no way of know if you were there or not!!!

Reminder: All of our labs involve paired programming. You do not have to keep the same partner for each lab, but you **MUST** work with someone in each lab!!! First, find a partner for this lab. It can be the same partner from the previous lab or a different partner.

You are going to continue to work on Assignment #2 in this lab. You will define the “Big Three” for the movie class, see if you have a memory leak, and update a data file.

(5 pts) Define the Big Three

Since your movie class has a dynamic array of strings, then you need to define a destructor, copy constructor, and assignment operator overload. You will work on your movie.h and movie.cpp files in this lab. You need to declare your class in the .h with preprocessor safe guards. Define a default, non-default, and big three for the movie class.

```
class movie {
public:
    movie();
    movie(int); //This will set num_cast and create cast array
    //Define any other constructors and the Big Three
private:
    string name;
    int stars;
    int num_cast;
    string *cast; //This is a dynamic array of strings
    string rating;
    int copies;
};
```

Implement the constructors, destructors, and member functions in the movie.cpp. In a main.cpp file, let's learn about a tool to test for memory leaks. In your driver, main.cpp, make a call to a function that creates a local movie object using the non-default constructor that sets num_cast and creates a dynamic array of num_cast strings for the movie. This will test the functionality of the **destructor**. For example:

```
void fun() {
    movie m(9);
}
int main() {
    fun();
    return 0;
}
```

We will learn about a utility in Linux that allows us to view how much memory we have allocated and deallocated (freed) during a run of your program. You can run your program through this utility called `valgrind`. For example, type **valgrind prog_name** at the prompt.

```
% valgrind movie.exe
```

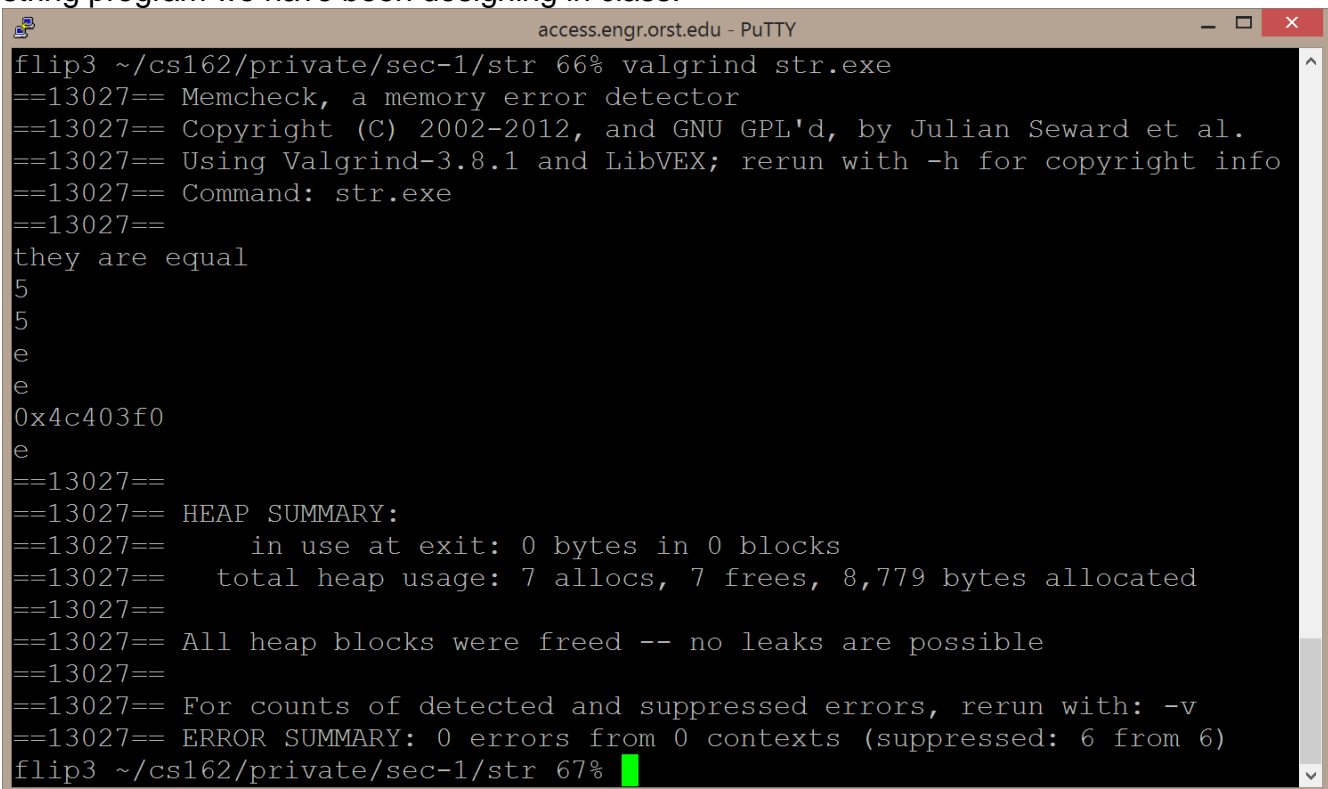
Now, you can add passing a movie object by value to see if the **copy constructor** allocates the correct memory and the **destructor** deallocates the memory when it goes out of scope.

```
void fun(movie main_mov) {
    movie m(9);
}
int main() {
    movie m(4);
    fun(m);
    return 0;
}
```

Run this through `valgrind` to show that this functionality is working correctly. Lastly, let's see if the **assignment operator overload** is deallocating/reallocating correctly by assigning the movie parameter to the local movie in the function.

```
void fun(movie main_mov) {
    movie m(9);
    m = main_mov;
}
```

Run this through `valgrind` to **show the TAs that the “Big Three” functionality is working correctly**. For instance, your output should look similar to the output below, which is using the string program we have been designing in class:



```
access.engr.orst.edu - PuTTY
flip3 ~/cs162/private/sec-1/str 66% valgrind str.exe
==13027== Memcheck, a memory error detector
==13027== Copyright (C) 2002-2012, and GNU GPL'd, by Julian Seward et al.
==13027== Using Valgrind-3.8.1 and LibVEX; rerun with -h for copyright info
==13027== Command: str.exe
==13027==
they are equal
5
5
e
e
0x4c403f0
e
==13027==
==13027== HEAP SUMMARY:
==13027==    in use at exit: 0 bytes in 0 blocks
==13027==   total heap usage: 7 allocs, 7 frees, 8,779 bytes allocated
==13027==
==13027== All heap blocks were freed -- no leaks are possible
==13027==
==13027== For counts of detected and suppressed errors, rerun with: -v
==13027== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 6 from 6)
flip3 ~/cs162/private/sec-1/str 67%
```

(5 pts) Update movies rented

Your file, `netflix.dat`, should have the following information separated by the pipe delimiter. For example, here are the categories in the file:

```
#|Title|Stars|NumCast|Members|Rating|Copies|Rented
```

Pre-Populate the data file with the following movies:

```
1|Computer Science 162|5|2|Jennifer Parham-Mocello|Students|PG|2|0
2|Computer Science 261|5|2|Ron Metoyer|Students|PG|2|0
3|Computer Science 271|5|2|Kevin McGrath|Students|PG|2|0
```

Now, we want to rent the “Computer Science 261” movie to see what the next class is going to look like. This should retain all the information in the current file and only update the movies rented for “Computer Science 261” to 1. For example, `netflix.dat` should look like the example below, after running your program.

```
1|Computer Science 162|5|2|Jennifer Parham-Mocello|Students|PG|2|0
2|Computer Science 261|5|2|Ron Metoyer|Students|PG|2|1
3|Computer Science 271|5|2|Kevin McGrath|Students|PG|2|0
```

Make sure you look over the example from class of finding the right title. You need to modify this so that it goes until the end of file, `eof`, rather than just two movies in the outer loop. For example:

```
while(!iofile.eof()) { //is the character read or viewed the eof
    //code to go through the movie data is here

    iofile.peek(); //Don't read data, look if there is another movie
}
```

To modify data in the data file, open a different file, **`netflix2.dat`**, for overwriting, and add all the un-modified and modified data to this file. Look at the `rename()` function that is part of the C `stdio.h` to **move the `netflix2.dat` to the `netflix.dat` file.**

If you finish the lab early, work on the “Big Three” for the netflix class!