Some takeaways from Assignment 1

I am putting down some things I noticed in various assignments, with the hope that all of you will read through this and understand it… Nothing like learning from other's mistakes ☺

- Sanjeev

1. No need to check a pointer for null before deleting it.

In other words, just saying `delete ptr1` or `delete[] ptr2` (whichever delete is appropriate) is fine.

2. `Circle Circle::operator + ( const Circle & other )`

You should declare this operator overload as a const since it does not modify the `this` object.

3. If you want to copy from two different sources, and the source you choose depends on some condition, you can combine the code as follows to make it smaller and simpler:

```
// can throw exception if out of mem.

mName = new char[MaxCircleNameSize];

const char * source = (name != NULL) ? name : "Some default name";

strncpy_s( mName, MaxCircleNameSize, source, _TRUNCATE );
```

4. `const Circle Circle::operator ++ (int)`

There is no need to return a `const`, since you are returning a temporary object created on the stack inside the member operator. So just leave the return type as `Circle`.

5. return type of `getName` should be declared as `const char *` instead of `char *` . Otherwise one can have the following code:

```
Circle c1 (1,10,6,”child”);
char * ptr = c1.getName();

ptr[0] = ‘C’;        // I was able to modify the name,
                     // which I really should not be able to do
```

But if it is declared as follows, then I cannot modify the name:

```
const char * getName() const;

char * ptr = c1.getName(); // This will give a compile error.

const char * ptr = c1.getName();  // This will compile fine.

ptr[0] = ‘C’;        // This will give a compile error,
                     // and I cannot modify the name
```

6. operator + should not change the `this` object.

```
Circle Circle::operator+(const Circle & c2)
{
      mRadius = mRadius + c2.mRadius;
      return *this;
}
```

Given above implementation of operator +, look at the code below:

```
// c1, c2 and c3 are Circle objects.
```

```
c1 = c2 + c3;
```

I don't expect c2 to change as a result of this. But with the above implementation, c2 will change. So the right thing to do would be something like:

```
Circle Circle::operator+(const Circle & c2)
{
        return Circle( mRadius, c2.getRadius(), other args );
}
```

7. If you allocate memory, and then assign something to that pointer, you just lost the memory you allocated (resulting in a memory leak):

```
char * newName = new char[kMaxSize];
newName = "newCircle";     // this is a shallow copy, and memory allocated
                           // above is lost.
```

8. In the code below, pre and post increment are not doing anything different.

```
const Circle & Circle::operator++()
{
        mRadius++;
        return *this;
}

const Circle Circle::operator++(int)
{
        mRadius++;
        return Circle(mRadius, mXCoord, mYCoord, mName);
}
```

For post increment, you need to copy the circle object **before** the increment is done, then return that copy. (See solution).

Also, there is no need to return a const reference or a const object. (See solution and also point 4 above).

9. If you have name allocation and initialization code in a separate function, say, InitName, then make sure that your data member mName gets initialized even if the name argument passed in is NULL.

   ```
   void InitializeName( const char * src )
   {
          If (src != NULL)
          {
                 // allocate memory and copy from src
          }
   }
   ```

   In this case, data member `mName` should likely be initialized before in the initialization list of the constructor, and then `InitializeName` will only change it if `src` is non NULL.

10.

Look at the constructor and destructor below:

```
Circle::Circle( int radius, int xCoord, int yCoord,
                      const char * name )
:mRadius( radius ), mXCoord( xCoord ), mYCoord( yCoord )
{
    if (name)
    {
        std::string tmpString(name);
        mName = strdup(tmpString.c_str());
    }
  else
  {
        mName = "";
    }
}

Circle::~Circle()
{
       free( mName );
}
```

I can see two issues here:

a. There is no need to create `tmpString` object, as *name* can be passed in to `strdup` directly, so no need to pay for construction and destruction of *tmpString*.

```
mName = strdup( name );
```

b. If name is NULL, then the code does: `mName = "";`

   This means that if the name passed in to the constructor is NULL, then the data member mName is made to point to the string literal "".

   In the destructor, `free( mName )` is called. This is a problem.

   Any string literal is read only memory, and you are not allowed to modify its contents or delete that memory.

   In the case where name passed in is NULL, mName can be initialized to NULL. So, you could initialize mName to NULL in the initializer list, and only have the `if (name)` block in the constructor.

11. I see two issues in the code below:
    a. **Performance**: Look at comment regarding calls to strlen.
    b. **Array access out of bounds**. See comments below.

```
Circle::Circle(int radius, int xCoord = 0, int yCoord = 0, const char * name = NULL )
: mRadius(radius), mXCoord(xCoord), mYCoord(yCoord)
{

        if(name) {

                // sq  Performance issue:
                //     strlen is an O(N) operation.
                //     It is being called 3 times below, thats not good, you
                //     should stores its return value in a local variable.

                mName = new char[strlen(name)];

                strncpy(mName, name, strlen(name));

                mName[strlen(name)] = '\0';      // sq  BUG: Array overflow. You are
                                                  // accessing mName array out of bounds.

                            // Should have allocated strlen(name) + 1 bytes.

        } else {

                mName = nullptr;                  // sq Should initialize to nullptr in
                                                  //    initialization list
  }
```

12.
The assignment operator below passes other by value as the developer wants to use swap to achieve assignment.
However, the self assignment guard at the beginning is not needed in this case, because when you pass other by value, it's a copy of what was used in the assignment. As a result, the address of that copy cannot be equal to this object.

```
        Circle& Circle::operator=(Circle other)
        {
          if (&other == this)      ← Check not needed
                return *this;
        …
        }
```

Lets say the code to invoke above assignment operator is:

```
        Circle c1(1, 5, "test");
        c1 = c1;
```

The above is translated to something like:

```
        c1.operator = (copy of c1 because the parameter is passed by value).
```

Hence, the address of copy will not be equal to c1

13.
Two problems in the code below:

1. It deletes mName before doing any memory allocation.
    a. See Operators slide deck and/or class recording on why that can cause trouble.

2. If parameter `name` is nullptr, then `mName` is not assigned a new value.
   It means `mName` continues to point to the memory that was just deleted.
   As a result, in the destructor, when `delete[] mName` is done again, it will cause undefined behavior (typically a crash).
   So, in such a situation where you delete memory pointed to by a data member, and may not assign memory to it after that, then you can assign nullptr to it to be safe.

```
Circle::CloneString(const char* name)
{
    delete[] mName;

    if (name!= nullptr)
    {
            mName = new char[ …];
            copy to mName;
    }
}
```

14.
sizeof does NOT give you length of a string.

Name is declared as const char * name

```
if (name)
{
    mName = new char[sizeof(name)];
    … code to copy string into mName
}
```

The code above will allocate enough bytes to hold a pointer, because sizeof returns the size (in bytes) of the passed in object / type.
But the intent was to allocate enough memory to copy the string pointed to by name into mName. So this is an error.