Winter 2020 CS 32

Project 2 FAQ

1. **Should I insert a new item into the linked list at the front or the back?**

   Imagine doing it one way. Now imagine doing it the other way. Does either way cause a violation of the specification? No? Then it's just an implementation detail of your choosing.

2. **I'm thinking of implementing a private member function that returns a pointer to a nested class/struct. How do I do that?**

   Consider this:

```
class M
{
  ...
  struct N
  {
    ...
  };
  N* f();
  ...
};

N* M::f()     // Error!  Compiler doesn't recognize N.
{
  ...
}

M::N* M::f()  // Oh, M's N!  This will compile.
{
  ...
}
```

   There's a C++ language rule that says roughly that the return type part of the definition of a member function of a class M isn't in the scope of M, so a name from M (like N) needs to be qualified (like M::N). Oddly enough, the parameter list *is* in the scope of M, so if M declared a member function `void g(N* n)`, the implementation could begin `void M::g(N* n)` if you wish (or `void M::g(M::N* n)`, of course).

3. **Does this compile? If so, what is its output?**

```
void f(const int& x, int& y)
{
    y = x + 1;
}

int main()
{
    int k = 0;
    f(k, k);
    cout << k << endl;
}
```

   Yes, it compiles, and the output is `1`. The declaration `const int& x;` says that *through the name `x`*, the function is *not* allowed to modify the object `x` refers to, so an assignment like `x = 42` would not compile. The declaration `int& y;` says that *through the name `y`*, the function *is* allowed to modify the

object `y` refers to. So in the call `f(k, k)`, the function can modify `k` if it does so through the name `y`.

4. **How can I test for memory leaks?**

   That's a hard problem in general. If we limit it to checking whether we leak any linked list nodes, where each linked list node contains an object of the ValueType, say, of the Map, then we can do it this way: Declare the value type of the Map to be a special class we'll write, one that will keep track of *all* creations and deletions of objects of that special type. We can do this by instrumenting every constructor and the destructor of that special class. Then we run our tests and see if the number of destructions equals the number of constructions, as shown in some sample code.

   Running our g32 command also helps, since we set it up to invoke g++ with compiler options that insert code that detects most memory leaks that occur during execution.