Programming Assignment

The goal of this assignment is to implement map from the previous assignment as a C++ class.

Part 1: restructure code in C++ style – hide Node definition inside Map class (nested classes). Make sure you understand access privelidges (basically none – neither outer, nor inner class has any special rights in accessing data in inner (correspondingly outer) classes. Note that unlike in C version of this assignemnt there is no need to worry that client sees Node, since it's private.

Part 2: implement Big Four: default ctor, copy ctor, assignment operator, and destructor. Remember to perform deep copy in copy ctor and assignment operator, do not duplicate code (create a helper function).

Part 3: to allow client to traverse the map, we need something similar to this code from the previous assignment:

```
void print_forwards(Node_handle pRoot) {
    Node_handle b = first(pRoot);
    while ( b  ) {
        printf ("%d -> %d;  ",getkey(b),getvalue(b));
        b = increment(b);
    }
    printf("\n");
}
```

remember that opaque pointer `Node_handle` main goal was to provide pointer like behavior, while not letting the client to perform certain operations. In C++ this is done using **Proxy** classes. Proxy class has a look and feel of protected type, while it also implements logic that prevents certain operations. We will discuss full-blown Proxies later, and for this assignment you'll create a

```
class Map_iterator {
    private:
        Node* p_node;
    public:
        Map_iterator(Node* p);
        Map_iterator& operator++();
        Map_iterator operator++(int);
        int& operator*();
        bool operator!=(const Map_iterator& rhs);
        bool operator==(const Map_iterator& rhs);
        friend class Map;
};
```

which is a proxy of a **pointer to Node** (same way as Node_handle before). Notice that `Map_iterator` implements "typical" pointer operations `p++, ++p, *p,` etc. `Map_iterator` allows client to modify data (`*p=100;`), but not keys.

Notice that the only data member of `Map_iterator` is `Node* p_node`, which allows iterator to access data in the corresponding node. Since `p_node` was not created by a constructor of `Map_iterator, Map_iterator` DOES NOT OWN `p_node.` This is referred to as non-owning semantics and such classes, usually do not require deep copy constructor and assignment operator.

To allow constant Map traversal we also need `Map_iterator_const.` While working on this part try to answer the following question – why cannot we use `"const Map_iterator"?`

In order to traverse Map objects we'll need 2 special iterator, so that the loop knows where to start and when to stop. Let's review how this is achieved with C++ arrays:

```
for (type* p = array;  p != array+size; ++p) ...
```

so that the initial value of `p` is pointing to the first element of the array (since `p = &array[0]`), and the last value is `array+size` which is pointing to one past the last element of the array.

The initial value for looping is an iterator to the first element in the map which is returned by `begin() method.` The "one past last" is more complex – since unlike arrays there is no "next" pointer, we'll need a special iterator:

```
static Map_iterator             end_it;
static Map_iterator_const const_end_it;
```

which are returned by

```
Map_iterator       end();
Map_iterator_const end() const;
```

correspondingly. Now we can do

```
CS225::Map::iterator b = map.begin(), e = map.end();
for ( ; b!=e; ++b) {
    std::cout << *b << std::endl;
}
```

To make our Map to feel like **"associative array"**, we need a simple index operator

```
int& operator[](int key);
```

which

1) return a non-const reference to a value if `key` exists
2) creates an element with default value 0 and `key`, otherwise

Note that there is no const version of index operator because of the 2).

Notes:

- make sure you understand the structure of the code and choices for private/public access
- read my comments
- compare to `std::map`
- deletion may be implemented using either `delete` or `delete_proper` from the previous assignment (master output uses `delete_proper`), I'll provide output using `delete` if requested. It is strongly suggested though that you try to implement the former – it is a very reasonable interview question, plus it may save you time in CS280.
- Make sure that code that I commented out and marked as "should not compile" **does not compile.** You will loose points if it does.
- Look at my `logging.h, logging_macros.h` (possibly at `cycle.h` which is not mine, but also of some interest). Make sure you understand how to use it. There is a small issue with cycle.h and -pedantic, the former requires "long long" which is not ANSI, thus pedantic compilation fails. Use another GCC target `gcc0_nopedantic` which skips pedantic, but remember that I will compile your code with `gcc0`, so comment out logging:
  `//#define FUNCTION_LOG "LOG"`

Note:
when updating your code to C++ you we'll substitute **malloc** with **new** remember to remove **if** check for NULL pointer, instead of returning a NUL pointer new will through, so **if** is useless. You are not required to add exception handling at this point.

To submit,
map.h
map.cpp

Code should be commented using doxygen style, file/method headers, etc.