Goals

Implement the copy constructor and assignment operator (ie. =) for your hash list. Implement a basic iterator and accompanying functions for your hash_list. See hash_list.h for full details.

Submission instructions

Upload your hash_list.h and hash_list.cpp to Gradescope. Any tests that you pass will be in green, any that you fail will be in red, and for any failed test the output from that run (ie. everything that was printed to the terminal) will be displayed in a box under the tests name

Restrictions

Same as in the previous lab

Copy constructor

Lets imagine we have a class like the following

```
class node
{

    node(int elem)
    {
        _elem = elem;
    }

    int _elem;
};
```

If we have a node we want to copy we could do the following

```
node x(3);

node y(x); -> create y as a copy of x
```

The question then becomes "does this work, and if so, how?". The answer to the first part of that question is yes, it does work, and the answer to the second part is what the rest of this handout will explain.

This odd looking line of code invokes the copy constructor of the node class.

```
node y(x);
```

As its name implies, the copy constructor is used to copy objects. It takes an instance of an object as input and creates a copy of that object. It turns out that the C++ compiler is very nice and will provide a built in copy constructor if you don't provide one. The built in implementation does a bitwise copy of each member of the class. So in our example above we would end up with

```
x._elem = 3    -> because we set this when we called node x(3);

y._elem = 3    -> because the built in copy constructor copies x.elem
   into y.elem
```

Now this seems very convenient. We wanted y to be a copy of x after all, and that's exactly what we got. But the built in implementation can also be very dangerous. Let's imagine we had the following class instead

```
class dangerous_node
{

    dangerous_node(int n)
    {
        ptr = new int;
        *ptr = n;
    }

    ~dangerous_node()
    {
        delete(ptr);  -> Ensures that we can't leak memory
    }

    int* ptr;
};
```

Now lets imagine we copy a dangerous_node the same way we did above

```
{
    dangerous_node x(100);

    dangerous_node y(x);
}
```

If you write code like this your program will probably crash. Let's explore why.

```
{
    dangerous_node x(100);

    dangerous_node y(x); -> At this point we copy each member of x into
       y. So we end up with y.ptr == x.ptr
}
```

Once we leave the scope that contains x and y we call their destructors

```
{
    dangerous_node x(100);

    dangerous_node y = x;
} -> At this point we call ~x() and ~y()
```

When we call ~x() all is good. We call delete() on x.ptr, so we have no memory leaks. But when we call ~y() we end up with a large problem. Remember, the built in copy constructor does a bitwise copy of each member. So y.ptr points to the same piece of memory that x.ptr pointed do. And we already deleted that memory when we called ~x(). So when we call ~y() and call delete() on y.ptr we're attempting to delete memory that's already been deleted. That's undefined behavior, so your program will probably crash.

So in the earlier case with the nodes class we liked what the compiler did for us, but we didn't like what the compiler did for us in the dangerous_node case.

The underlying reason that we need to change the implementation of the copy constructor goes back to what we mean when we say "copy". In the case of the dangerous_node the thing that we've interested in copying isn't x.ptr but rather the memory that x.ptr points to. The pointer is really just a way to access the memory we've allocated. And because of that when we copy x we don't want to copy the contents of x.ptr, we want to copy the memory that x.ptr points to. This means we can't rely on the built in copy constructor, since the built in constructor isn't aware that x.ptr points to allocated memory. In this respect what we want when we copy x is the following

```
y.ptr = new int;     -> Allocate a new chunk of memory

*y.ptr = *x.ptr;     -> Copy data pointed to by x.ptr into memory pointed
   to by y.ptr
```

By the way, when you have the situation we did above, where you have a chunk of memory and want to "copy" it by allocating new memory and copying data from the old memory to the new memory, that's called a deep copy. Presumably the reason is because you're not copying the surface level part (ie. the pointer to the memory) but rather are going "deep" by following the pointer and copying the memory the pointer points to.

The opposite of a "deep" copy is a "shallow" copy. That's where you just do a bitwise copy of each member, ie. what the compiler does for you. This is fine for our node class but usually wont't work for classes that manage memory, such as our dangerous_node class

Assignment operator

One of the werid quirks with the above examples is the way that we constructed new objects. If we had asked you to write code that would assign x to y you probably would have done this

```
node y;
y = x;
```

Instead of

```
node y(x);
```

The reason for us choosing to use the copy constructor above is because the = operator is a little more complicated than the copy constructor, so it's easier to explain the copy constructor and introduce the = operator later. Let's illustrate why = is a more complicated with an example from our dangerous_node.

Imagine we want to implement the = operator for our dangerous_node class above. An initial implementation might look like this

```
dangerous_node& operator=(const dangerous_node& other)
{
    this.ptr = new int;          -> Allocate new memory for the deep copy
    *this.ptr = *other.ptr;      -> Deep copy data
}
```

This code looks perfect. It avoid exactly the issues that we talked about in the copy constructor. It allocates new memory and copies data from the old memory into the new memory. And yet this implementation will leak memory all over the place. Here's how

```
dangerous_node y(100);     -> Allocate memory for integer 100 and have y
   point to it

dangerous_node x(200);     -> Allocate memory for integer 200 and have x
   point to it

y = x;                     -> Call = operator for y
```

When we do y = x we end up calling the = operator code from above. And the problem is here

```
dangerous_node& operator=(const dangerous_node& other)
{
    this.ptr = new int -> this.ptr already points to the memory we
       allocated earlier. When we reassign this.ptr we leak that memory
       and can't delete it
    *this.ptr = *other.ptr;
}
```

And this reveals the difference between the copy constructor and the = operator. The copy constructor is responsible for creating a new object from an existing one. That new object has no previous state to clean up; it is simply brought into existence. But the = operator doesn't do this. The = operator takes an existing object and converts its contents to the contents of a different object. That means that we also need to clean up that object's state before we reassign it. So the = really does 2 things

```
1. Clean up the state of the object (ie. destruct it)
2. Copy state from the other object into the current object -> The copy
   constructor only does this
```

note: although I use the word destruct to describe the first step of what the destructor does you shouldn't call the destructor in your = operator.

So our final implementation of our dangerous_node = operator would look like this

```
dangerous_node& operator=(const dangerous_node& other)
{
    delete(this.ptr); -> Here we're cleaning up the old state
    this.ptr = new int;
    *this.ptr = *other.ptr;
}
```

There are two last topics you should be aware of. The first is the copy and swap idiom, which is way to use the copy constructor to implement the = operator (see how much duplicate code exists between our copy constructor and = operator implementations?) and the second is the rule of 3, which says that if you implement the copy constructor, = operator, or destructor, you should probably implement all 3.

std::pair<K,V> explanation

If you want to return multiple values from a function in python, you can just return a tuple. In C++ you can either use a tuple (which functions similarly to how python tuples function) or you can return a pair (which is basically a tuple that has exactly two values).

For example, if you wanted your function to return an integer and a float you could do

```
std::pair<int, float> foo() // <int, float> tells the compiler the types
   you're returning
{
    int x = 3;
    float y = 4;
    return {3,4};   // putting the values inside of {} creates a pair
}
```

Two of the members of std::pair are `first` and `second`. These are how you access the values that are inside of the pair. Using the `foo()` function from above we would access x and y by

```
std::pair<int, float> returned_pair = foo();

int new_x = returned_pair.first;        // Get x out of the pair
```

```
float new_y = returned_pair.second;     // Get y out of the pair
```

Iterator discussion

When implementing the iterator functions we left it unspecified what the iterator should point to when you create a list. This is because we're only defining the iterator behavior after reset_iter() is called. This means that before we use the iterator to iterate over your lists we will call reset_iter().

An additional issue you might think of is what happens if someone is iterating over your list and calls remove() on the node that the iterator is pointing to? We've decided to call this undefined behavior. If someone is iterating over the list and they call remove(), the iterator is invalid. If they want to keep iterating over the list after calling remove() they need to reset the iterator and start from the beginning again.

We've also decided it's undefined behavior to change the value of the key using get_iter_value(). This is because changing the key means you're able to get around the invariant that each key in the hash_list is unique.

We've given an example of how to use the iterator in main.cpp