Module IN3013/INM173 – Object Oriented Programming in C++

Solutions to Exercise Sheet 10

1. The recommended uses of exceptions:

   (a) clean up (eg close open files) and re-throw (or exit with error):

```
try {
        // do something that might fail
} catch (...) {
        // clean up
        throw;
}
```

   (b) attempt to rectify original cause and ensure that try block is re-executed:

```
bool finished;
do {
        finished = true;
        try {
                // do something that might fail
        } catch (SomeError) {
                // attempt to fix problem
                finished = false;
        }
} while (! finished)
```

   (c) attempt to achieve same effect as the try block in some other way:

```
try {
        // do something that might fail
} catch (...) {
        // do something else
}
```

2. easy stuff

3. The following version will destroy the table and name automatically:

```
class Node {
        auto_ptr<Table> table;
        auto_ptr<Name> name;
```

```
public:
        Node(const char *n) :
                table(new Table()), name(new Name(n)) {}

        // ...
};
```

Thanks to the overloading of the pointer operators, there is no need to change anything else.

The default copy constructor will copy the `auto_ptr`s, and they have safe copy constructors. So there is no need to write an user-defined copy constructor. The same goes for the assignment operator.

4. First we need an auxiliary class to hold the pointer and a count of references to it.

```
template <class T> class smart_ptr_aux {
        T *ptr;
        int count;      // reference count

public:
        smart_ptr_aux(T *p) : ptr(p), count(1) {}

        T *get_ptr() const { return ptr; }

        static void increment(smart_ptr_aux<T> *p) {
                p->count++;
        }

        static void decrement(smart_ptr_aux<T> *p) {
                p->count--;
                if (p->count == 0) {
                        delete p->ptr;
                        delete p;
                }
        }
};
```

The functions `increment` and `decrement` adjust the count, deleting the object pointed to and this auxiliary object when the count falls to zero. The `decrement` function must be `static` in order to delete the object `p`. The `increment` function is made `static` for consistency.

Now the smart pointer class itself uses the above methods:

```
template <class T> class smart_ptr {
        smart_ptr_aux<T> *aux;

public:
        smart_ptr(T *p) : aux(new smart_ptr_aux<T>(p)) {}

        smart_ptr(smart_ptr<T> & p) : aux(p.aux) {
                smart_ptr_aux<T>::increment(aux);
        }

        smart_ptr& operator=(smart_ptr<T> & p) {
                if (&p != this) {
                        smart_ptr_aux<T>::decrement(aux);
                        aux = p.aux;
                        smart_ptr_aux<T>::increment(aux);
                }
                return *this;
        }

        // make it look like a pointer
        T & operator*() { return *(aux->get_ptr()); }
        T * operator->() { return aux->get_ptr(); }

        ~smart_ptr() {
                smart_ptr_aux<T>::decrement(aux);
        }
};
```

5. It never deletes data unsafely, but sometimes fails to delete garbage. If one were to destroy the last reference to a doubly linked list, the list elements would still be referring to each other, and so would not be deleted. In general, the reference counting scheme fails to reclaim cyclic data structures.