lab03a.cpp: defining class operators

In lab01c.cpp, you defined a class `myclass` and tried to apply your generic algorithm `maxrange` to it. The attempt failed, because the generic algorithm required the objects to which it was applied to support the operator `>`, and `myclass` did not do this.

Add external friend functions `operator ==` and `operator <` to the `myclass` class in lab01c.cpp. See the example on page 65 for a demonstration of how friend operators may be defined. Have your operators determine whether a given instance of `myclass` is less than (or equal to) another purely on the basis of the `age` field. Pass both parameters to these operators by constant reference, for maximal efficiency and compatibility; e.g. `const myclass & arg1`. The `const` guarantees that the operator will not try to change the value of the parameter, and the reference saves the computer from a possibly lengthy copy operation. Then make sure your version of `maxrange` uses the `>` operator, not `<`; if it does not, rewrite it so it does. Try to compile the program; it should still fail, since `operator >` is still not defined.

Add the header file `<utility>` to your list of includes. This is the header file where the templates described in §3.2 are defined. Adding `using namespace rel_ops;` after the `using namespace std;` line. With these additions, the “missing” four operators become available to your program. Recompile, and test your program to be sure it works as expected.

Refine `myclass` one more time by adding friend operators for iostream insertion and extraction (operator `<` and `operator >>`). See the sample program on page 65 for an example of defining `operator <<`, but pass the `myclass` parameter by `const` reference. `Operator >>` is coded similarly, except the `myclass` must not be `const` (why?), and it uses `istream` instead of `ostream`, of course. Stream insertion should format a `myclass` object as a pair of integers in parentheses, first the `id` and then the `age`. Stream extraction should simply read two integers for the `id` and `age` fields, with no parentheses or comma. Modify your driver program so that, instead of statically initializing the array of `myclass` objects, it reads `myclass` data from `cin`, reprints the input, and then prints the value of the largest object as found by `maxrange`. Note that you will have to add a default constructor for `myclass` because C++ uses the default constructor when building an array. The default constructor could initialize `id` and `age` to anything; say to 0. Print a listing and a sample run to turn in.

lab03b.cpp: using function objects

The operator `<` that you defined in lab03a.cpp allows `myclass` to be used with any STL algorithms that require ordering capability. For instance, you could sort your array of `myclass` objects, using the STL `sort` algorithm, declared in the header file `<algorithm>`

```cpp
sort ( myarray, myarray+myarraysize );
```

Add this capability to your program, and print out the sorted array using your overloaded stream insertion operator. Note that the array has been sorted by `age`, since that’s how the operator `<` works.

But what if you want to sort by `id` instead, without having to change your definition of operators? No problem, because there is a form of the `sort` algorithm with a third parameter, with which you can specify a comparison to be used for the sort, as a function object. A function object is a class which implements `operator ()`, the function call operator. Add such an object to your code, where `operator ()` takes two (constant reference) `myclass` parameters and returns a `bool` result representing whether the first parameter’s `id` field is less than the second’s. Call your object `myclasslt`, and derive it from `binary_function < myclass, myclass, bool >` (defined in the header file `<functional>`) to give it a lot of useful self-
description tags. Begin your definition of `myclasslt` as follows:

```cpp
class myclasslt: public binary_function<myclass, myclass, bool>
```

(If you make `myclasslt` a struct, so access is by default `public`, you can leave off the keyword `public` in this line, but it’s probably better just to get in the habit of using explicitly public inheritance.)

Add a line `friend class myclasslt;` inside `myclass`, so that `myclasslt` has access to `myclass`’ private data. Declare the `operator()` member function to be `const`, guaranteeing that it will not modify any instance variables of class `myclasslt`. The syntax for this is:

```cpp
bool operator() ( const myclass & arg1, const myclass & arg2 ) const
```

Then, following the original sort, sort it again using an instance of your new function object as a third parameter. The easiest way to get such an instantiation is to create a temporary, unnamed object by calling the default constructor of `myclasslt`:

```cpp
sort ( myarray, myarray+myarraysize, myclasslt() );
```

Once again, print a listing and a sample run to turn in.

**lab03c.cpp: functional adapters and algebra**

If the ordering relation you wish to use for sorting can be easily derived from an existing operator, it will usually be easier to create the required function object from existing templates in the STL than to write one by hand. For instance, to explicitly sort your array of `myclass` objects by `age`, you might use the STL template `less<myclass>` as the third parameter of `sort`. This template simply turns an existing operator `<` into a function object which does the same thing.

If the ordering relation you need is derivable from an existing function object, you may follow a different but similar approach. To get an `descending` sort based on `myclasslt` without having to write a new function object, you could use the adapter `not2( myclasslt() )`. This adapter reverses the sense of a binary (hence, the “2”) function object, thereby causing the reversal in sorting order. Similarly, since `less<myclass>` is a binary function object, `not2(less<myclass>())` would cause the sort to go in decreasing order by `age`.

Add sorts to the end of `lab03b.cpp` to sort in descending order by `age` and again by `id`, without writing new function objects. Print a listing and a sample run.