So, calling `swap()` instead of swapping the values directly might result in substantial performance improvements. You should always offer a specialization of `swap()` for your own types if doing so has performance advantages.

4.5 Supplementary Comparison Operators

Four template functions define the comparison operators `!=`, `>`, `<=`, and `>=` by calling the operators `==` and `<`. These functions are defined in `<utility>` as follows:

```
namespace std {
    namespace rel_ops {
        template <class T>
        inline bool operator!= (const T& x, const T& y) {
            return !(x == y);
        }

        template <class T>
        inline bool operator> (const T& x, const T& y) {
            return y < x;
        }

        template <class T>
        inline bool operator<= (const T& x, const T& y) {
            return !(y < x);
        }

        template <class T>
        inline bool operator>= (const T& x, const T& y) {
            return !(x < y);
        }
    }
}
```

To use them, you only need to define operators `<` and `==`. By using namespace `std::rel_ops`, the other comparison operators are defined automatically. For example:

```
#include <utility>

class X {
    ...
  public:
```

```
  private:
    int theValue;     // the value to add
  public:
    // constructor initializes the value to add
    AddValue(int v) : theValue(v) {
    }

    // the ''function call'' for the element adds the value
    void operator() (int& elem) const {
        elem += theValue;
    }
};

int main()
{
    list<int> coll;

    // insert elements from 1 to 9
    for (int i=1; i<=9; ++i) {
        coll.push_back(i);
    }

    PRINT_ELEMENTS(coll,"initialized:                ");

    // add value 10 to each element
    for_each (coll.begin(), coll.end(),    // range
              AddValue(10));               // operation

    PRINT_ELEMENTS(coll,"after adding 10:            ");

    // add value of first element to each element
    for_each (coll.begin(), coll.end(),    // range
              AddValue(*coll.begin()));    // operation

    PRINT_ELEMENTS(coll,"after adding first element: ");
}
```

After the initialization, the collection contains the values 1 to 9:

```
initialized:                1 2 3 4 5 6 7 8 9
```

To use this generic function you simply must pass the container the old key and the new key. For example:

```
std::map<std::string,float> coll;
...
MyLib::replace_key(coll,"old key","new key");
```

It works the same way for multimaps.

Note that maps provide a more convenient way to modify the key of an element. Instead of calling `replace_key()`, you can simply write the following:

```
// insert new element with value of old element
coll["new_key"] = coll["old_key"];
// remove old element
coll.erase("old_key");
```

See Section 6.6.3, page 205, for details about the use of the subscript operator with maps.

Inserting and Removing Elements

Table 6.31 shows the operations provided for maps and multimaps to insert and remove elements.

Operation	Effect
`c.insert(elem)`	Inserts a copy of `elem` and returns the position of the new element, and for maps, whether it succeeded
`c.insert(pos,elem)`	Inserts a copy of `elem` and returns the position of the new element and whether it succeeded (`pos` is used as a hint pointing to where the insert should start the search)
`c.insert(beg,end)`	Inserts a copy of all elements of the range [`beg`,`end`) (returns nothing)
`c.erase(elem)`	Removes all elements with value `elem` and returns the number of removed elements
`c.erase(pos)`	Removes the element at iterator position `pos` (returns nothing)
`c.erase(beg,end)`	Removes all elements of the range [`beg`,`end`) (returns nothing)
`c.clear()`	Removes all elements (makes the container empty)

Table 6.31. Insert and Remove Operations of Maps and Multimaps

The remarks on page 182 regarding sets and multisets apply here. In particular, the return types of these operations have the same differences as they do for sets and multisets. However, note that the elements here are key/value pairs. So, the use is getting a bit more complicated.

To insert a key/value pair, you must keep in mind that inside maps and multimaps the key is considered to be constant. You either must provide the correct type or you need to provide implicit or explicit type conversions. There are three different ways to pass a value into a map: