Click on the banner to return to the user guide home page.

Comparators

The associative container-based and the sorted sequence-based collection classes maintain order internally. This ordering is based on a comparison object, an instance of a comparator class you must supply when instantiating the template. A comparator must contain a `const` member `operator()`, the function-call operator, which takes two potential elements of the collection class as arguments and returns a Boolean value. The returned value should be `true` if the first argument must precede the second within the collection class, and `false` otherwise. Often, it is easiest to use one of the function-object classes provided by the Standard C++ Library in the header file `<functional>`. In particular, use `less<T>` to maintain elements in increasing order, or `greater<T>` to maintain them in decreasing order. For example:

```
#include <functional>
#include <rw/tvset.h>
#include <rw/rwdate.h>

RWTValSet<int, less<int> > mySet1;
RWTValSet<RWDate, greater<RWDate> > mySet2;
```

Here `mySet1` is a set of integers kept in increasing order, while `mySet2` is a set of dates held in decreasing order; that is, from the most recent to the oldest. You can use these comparators from the Standard C++ Library as long as the expression `(x < y)` for the case of `less<T>`, or `(x > y)` for the case of `greater<T>`, are valid expressions that induce a total ordering on objects of type `T`.

More on Total Ordering

As noted above, the comparator must induce a total ordering on the type of the items in the collection class. This means that the function-call operator of the comparator must satisfy the following two conditions[15], assuming that `comp` is the comparison object and `x`, `y`, and `z` are potential elements of the collection class, not necessarily distinct:

```
I.  Exactly one of the following statements is true:
    a) comp(x,y) is true  and comp(y,x) is false
    b) comp(x,y) is false and comp(y,x) is true
    c) comp(x,y) is false and comp(y,x) is false
      (or, in other words: not both comp(x,y) and comp(y,x)
        are true)
II.  If comp(x,y) and comp(y,z) are true, then so is
     comp(x,z) (transitivity).
```

The truth of `I.a` implies that `x` must precede `y` within the collection class, while `I.b` says that `y` must precede `x`. More interesting is `I.c`. If this statement is true, we say that `x` and `y` are equivalent, and it doesn't matter in what order they occur within the collection class. This is the notion of equality that prevails for the templates that take a comparator as a parameter. For example, when the member function `contains(T item)` of an associative container-based template tests to see if the collection class contains an element equivalent to `item`, it is really looking for an element `x` in the collection class such that `comp(x,item)` and `comp(item,x)` are both `false`. It is important to realize that the == operator is not used. Don't worry if at first it seems counter-intuitive that so much negativity can give rise to equivalence_you are not alone! You'll soon be comfortable with this flexible way of ensuring that everything has its proper place within the collection class.

Comparators are generally quite simple in their implementation. Take for example:

```
class IncreasingLength {
public:
    bool operator()(const RWCString& x, const RWCString& y)
    { return x.length() < y.length(); }
};
RWTValSet<RWCString,IncreasingLength> mySet;
```

Here `mySet` maintains a collection of strings, ordered from shortest to longest by the length of those strings. You can verify that an instance of the comparator satisfies the given requirements for total ordering. In the next example, `mySet2` maintains a collection class of integers in decreasing order:

```
class DecreasingInt {
public:
    bool operator()(int x, int y)
    { return x > y; }
};
RWTValSet<int, DecreasingInt> mySet2;
```

Although the sense of the comparison may seem backwards when you first look at it, the comparator says that `x` should precede `y` within the collection class if `x` is greater than `y`; hence, you have a decreasing sequence. Finally, let's look at a bad comparator:

```
// DON'T DO THIS:
class BadCompare {
public:
    bool operator()(int x, int y)
    { return x <= y; }    // OH-OH! Not a total ordering relation
};
RWSetVal<int, BadCompare> mySet3;  // ILLEGAL COMPARATOR!
```

To determine why it's bad, consider an instance `badcomp` of `BadCompare.` Note that when using the value `7` for both `x` and `y`, none of the three statements `I.a, I.b,` or `I.c` is `true`, which violates the first rule of a total ordering relation.[16]

Previous Contents Next