Data Structures Using C++ 2E

Chapter 13

Standard Template Library (STL) II

Objectives

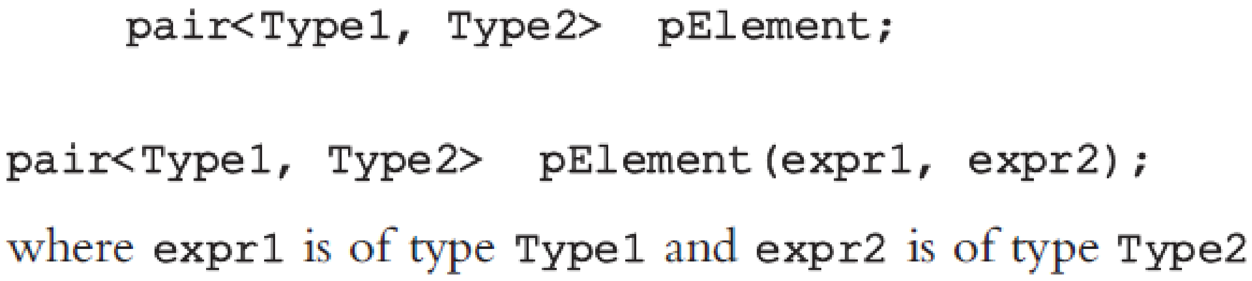
- Learn more about the Standard Template Library (STL)
- Become familiar with associative containers
- Explore how associative containers are used to manipulate data in a program
- Learn about various generic algorithms

`Class pair`

- Allows two values to be combined into a single unit
  - Treated as one unit
  - Functions using `class pair`
    - Return two values
- Using the `class pair` in a program
  - `class pair` definition contained in header file `utility`
  - Include statement: `#include <utility>`

`Class pair` (cont’d.)

- `class pair` constructors
  - Default constructor
  - Constructor with two parameters
- Type `pair` object
  - Two `public` data members: `first`, `second`
- See Example 13-1



```
pair<Type1, Type2>  pElement;
```

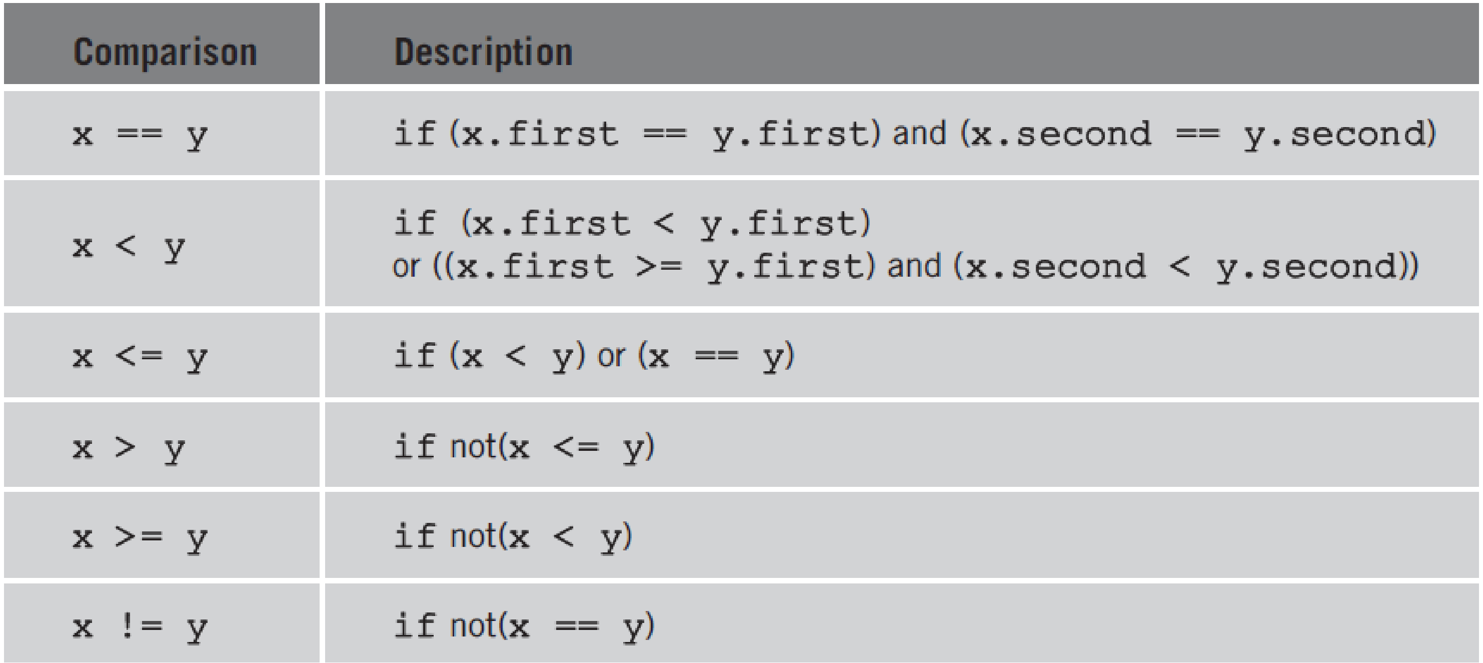
```
pair<Type1, Type2>  pElement(expr1, expr2);
```

where `expr1` is of type `Type1` and `expr2` is of type `Type2`

Comparing Objects of Type `pair`

- Relational operators
  - Overloaded for `class pair`

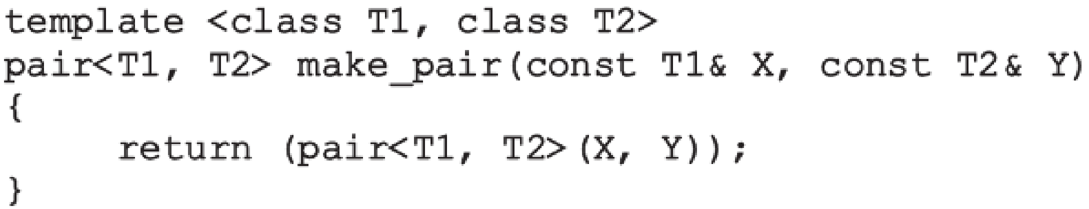
TABLE 13-1 Relational operators for the `class pair`



Comparison	Description
`x == y`	`if (x.first == y.first)` and `(x.second == y.second)`
`x < y`	`if (x.first < y.first)` or `((x.first >= y.first)` and `(x.second < y.second))`
`x <= y`	`if (x < y)` or `(x == y)`
`x > y`	`if` not`(x <= y)`
`x >= y`	`if` not`(x < y)`
`x != y`	`if` not`(x == y)`

Type `pair` and Function `make_pair`

- Header file `utility`
  - Contains definition of function template `make_pair`
- Create pairs without explicitly specifying type `pair`
  - With function `make_pair`
- Function template `make_pair`
  - Value returning function
    - Returns a value of type `pair`



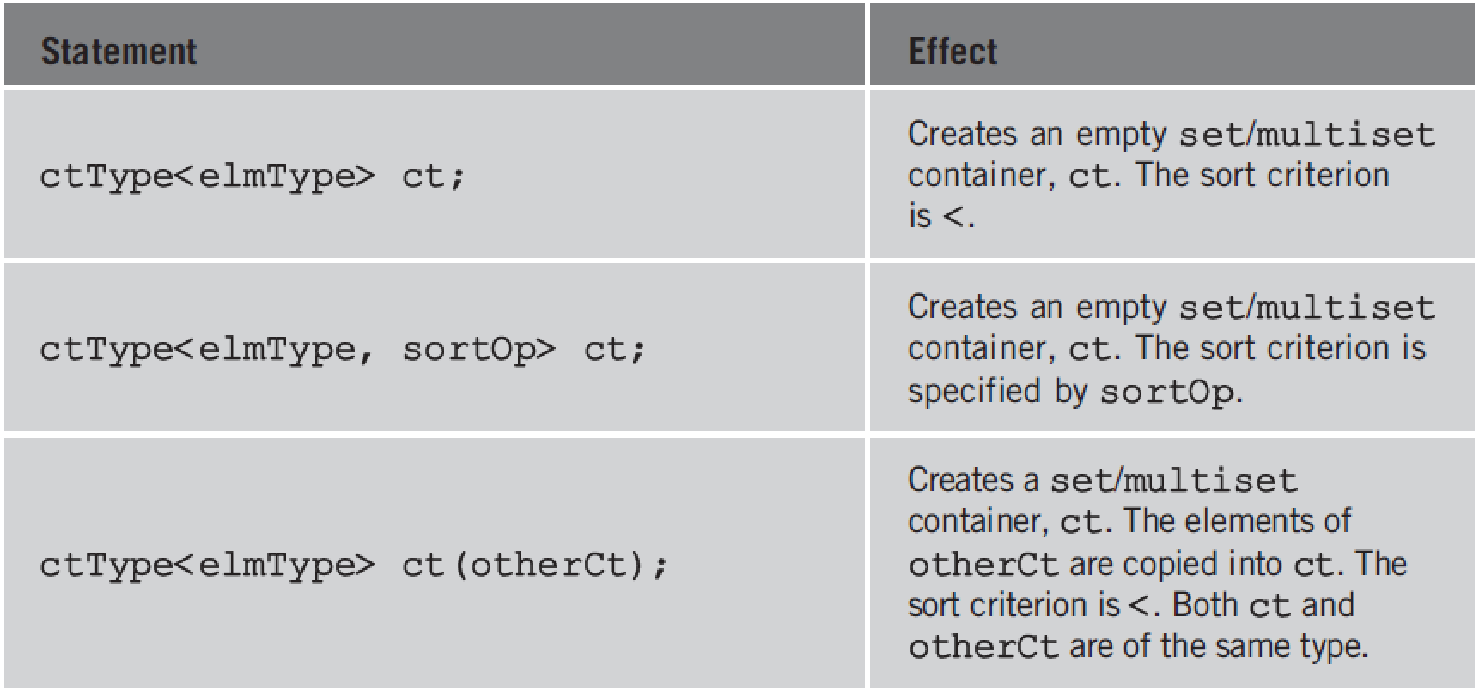
```
template <class T1, class T2>
pair<T1, T2> make_pair(const T1& X, const T2& Y)
{
     return (pair<T1, T2>(X, Y));
}
```

Associative Containers

- Elements automatically sorted
  - According to some ordering criteria
  - Default ordering criterion
    - Relational operator < (less than)
    - Users can specify own ordering criterion
- New element inserted at the proper place
- Binary search tree
  - Convenient and fast way to implement data structure
- Four predefined associative containers

Associative Containers: `set` and `multiset`

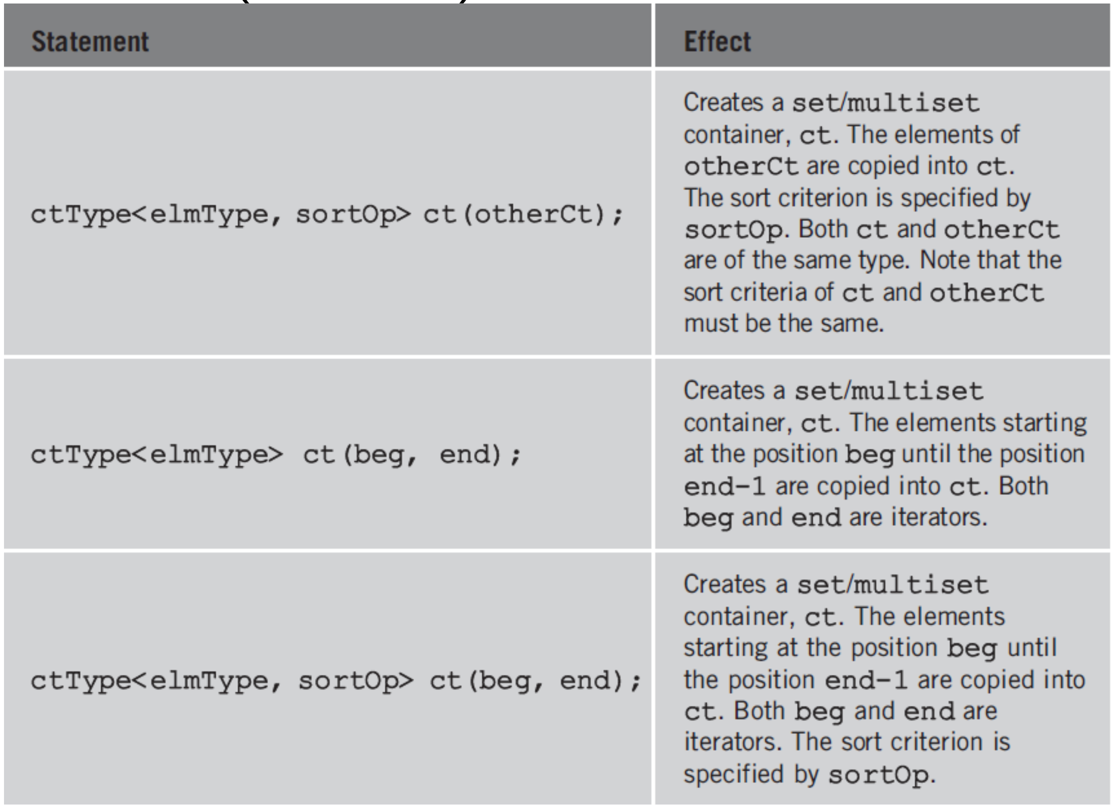
TABLE 13-2 Various ways to declare a `set`/`multiset` container



Statement	Effect
`ctType<elmType> ct;`	Creates an empty `set/multiset` container, `ct`. The sort criterion is <.
`ctType<elmType, sortOp> ct;`	Creates an empty `set/multiset` container, `ct`. The sort criterion is specified by `sortOp`.
`ctType<elmType> ct(otherCt);`	Creates a `set/multiset` container, `ct`. The elements of `otherCt` are copied into `ct`. The sort criterion is <. Both `ct` and `otherCt` are of the same type.

Associative Containers: `set` and `multiset` (cont'd.)

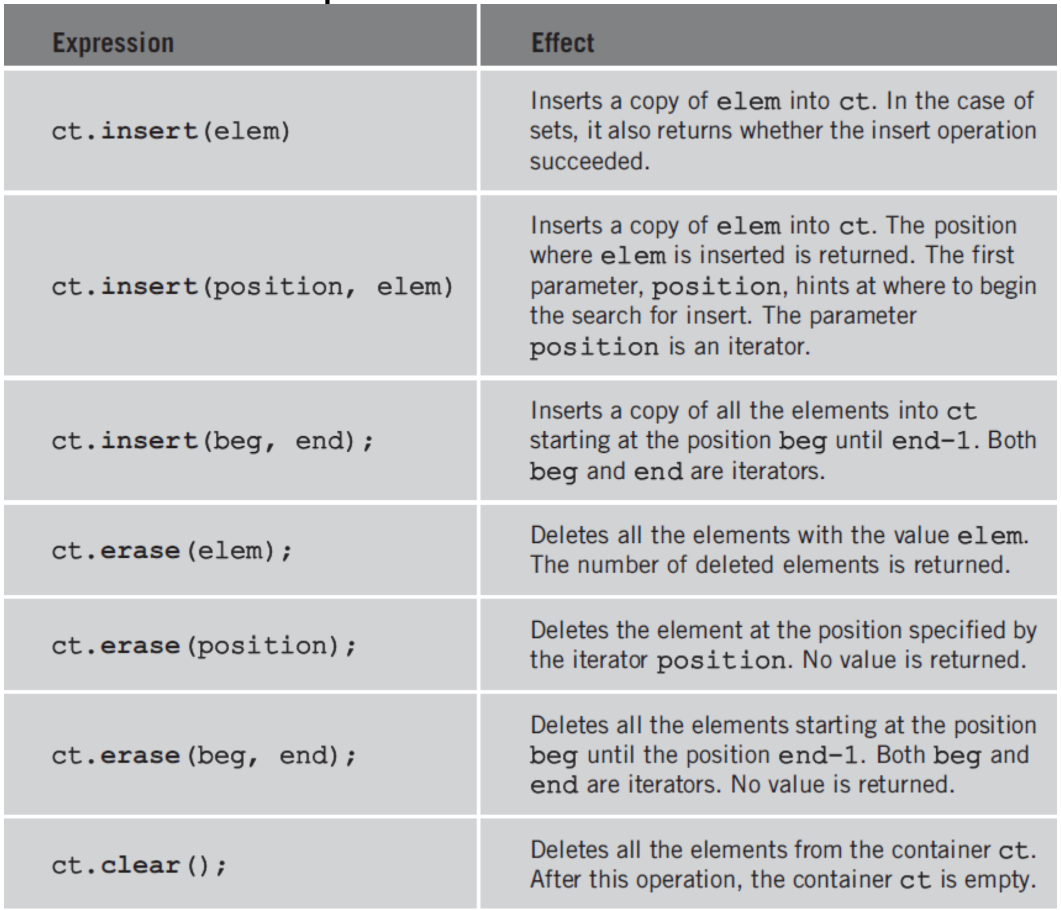
TABLE 13-2 Various ways to declare a `set`/`multiset` container (continued)



Statement	Effect
`ctType<elmType, sortOp> ct(otherCt);`	Creates a `set/multiset` container, `ct`. The elements of `otherCt` are copied into `ct`. The sort criterion is specified by `sortOp`. Both `ct` and `otherCt` are of the same type. Note that the sort criteria of `ct` and `otherCt` must be the same.
`ctType<elmType> ct(beg, end);`	Creates a `set/multiset` container, `ct`. The elements starting at the position `beg` until the position `end-1` are copied into `ct`. Both `beg` and `end` are iterators.
`ctType<elmType, sortOp> ct(beg, end);`	Creates a `set/multiset` container, `ct`. The elements starting at the position `beg` until the position `end-1` are copied into `ct`. Both `beg` and `end` are iterators. The sort criterion is specified by `sortOp`.

Associative Containers: `set` and `multiset` (cont'd.)

TABLE 13-3 Operations to insert or delete elements from a set



Expression	Effect
`ct.insert(elem)`	Inserts a copy of `elem` into `ct`. In the case of sets, it also returns whether the insert operation succeeded.
`ct.insert(position, elem)`	Inserts a copy of `elem` into `ct`. The position where `elem` is inserted is returned. The first parameter, `position`, hints at where to begin the search for insert. The parameter `position` is an iterator.
`ct.insert(beg, end);`	Inserts a copy of all the elements into `ct` starting at the position `beg` until `end-1`. Both `beg` and `end` are iterators.
`ct.erase(elem);`	Deletes all the elements with the value `elem`. The number of deleted elements is returned.
`ct.erase(position);`	Deletes the element at the position specified by the iterator `position`. No value is returned.
`ct.erase(beg, end);`	Deletes all the elements starting at the position `beg` until the position `end-1`. Both `beg` and `end` are iterators. No value is returned.
`ct.clear();`	Deletes all the elements from the container `ct`. After this operation, the container `ct` is empty.

Associative Containers: `map` and `multimap`

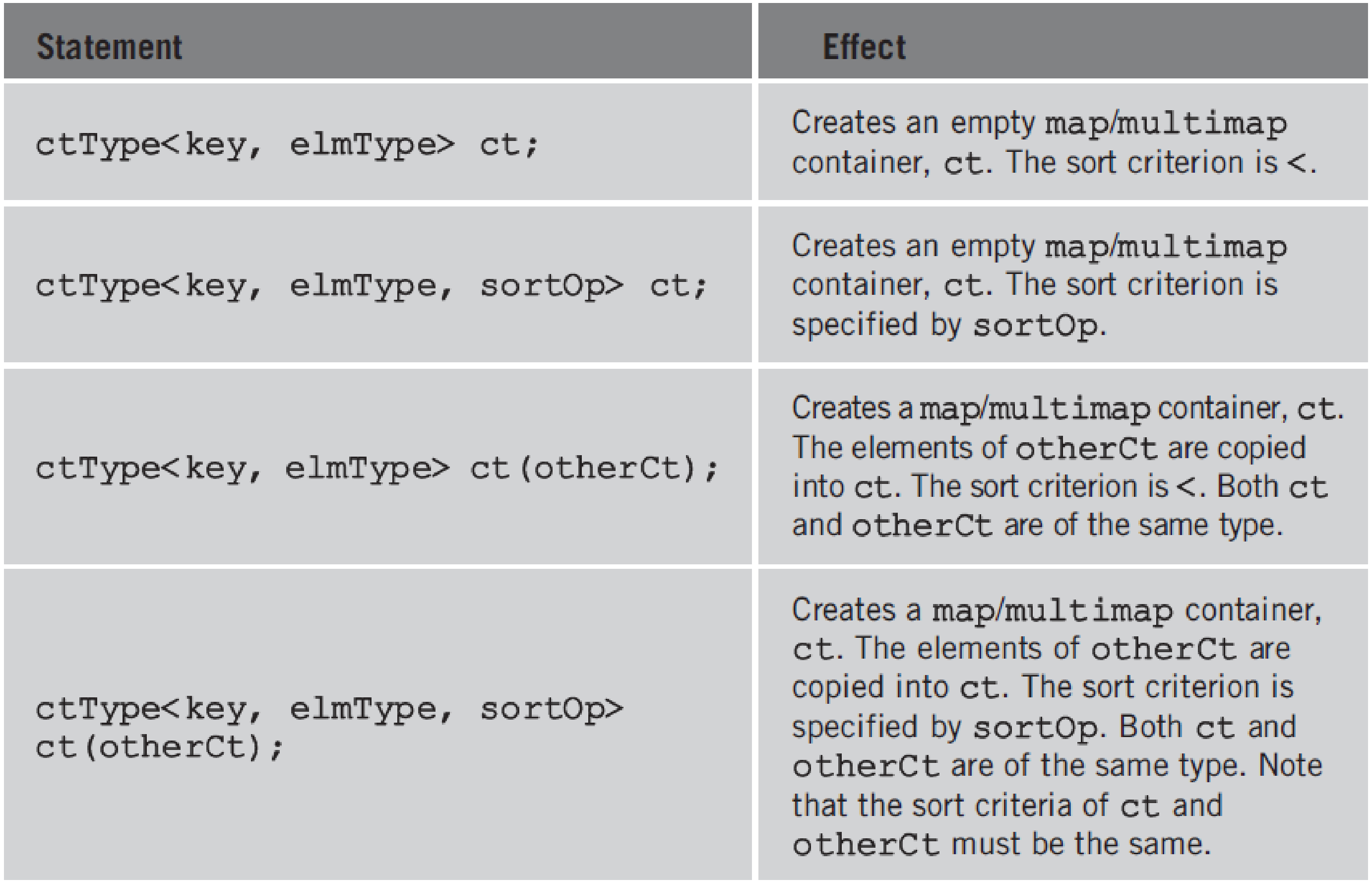
- Manage elements in the form key/value
- Sorting elements
  - Automatically according to sort criteria applied on key
    - Default sorting criterion: relational operator < (less than)
  - User can specify sorting criteria
- User-defined data types and relational operators
  - Must be properly overloaded

Associative Containers: `map` and `multimap` (cont'd.)

- Difference between `map` and `multimap`
  - Container `multimap` allows duplicates
  - Container `map` does not
- Class name defining container `map`: map
- Class name defining container `multimap`: multimap
- Use include statement: `#include <map>`

Associative Containers: `map` and `multimap` (cont'd.)

TABLE 13-4 Various ways to declare a `map`/`multimap` container

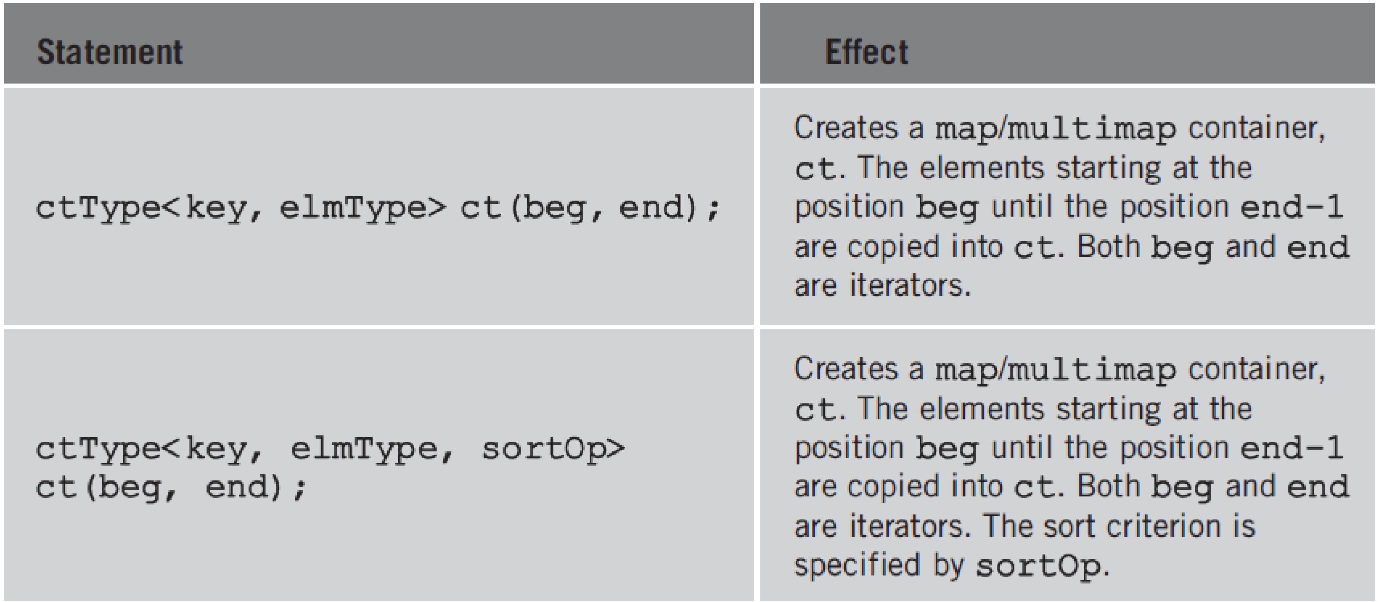


Statement	Effect
`ctType<key, elmType> ct;`	Creates an empty `map/multimap` container, `ct`. The sort criterion is <.
`ctType<key, elmType, sortOp> ct;`	Creates an empty `map/multimap` container, `ct`. The sort criterion is specified by `sortOp`.
`ctType<key, elmType> ct(otherCt);`	Creates a `map/multimap` container, `ct`. The elements of `otherCt` are copied into `ct`. The sort criterion is <. Both `ct` and `otherCt` are of the same type.

`ctType<key, elmType, sortOp>`
`ct(otherCt);`
	Creates a `map/multimap` container, `ct`. The elements of `otherCt` are copied into `ct`. The sort criterion is specified by `sortOp`. Both `ct` and `otherCt` are of the same type. Note that the sort criteria of `ct` and `otherCt` must be the same.

Associative Containers: `map` and `multimap` (cont'd.)

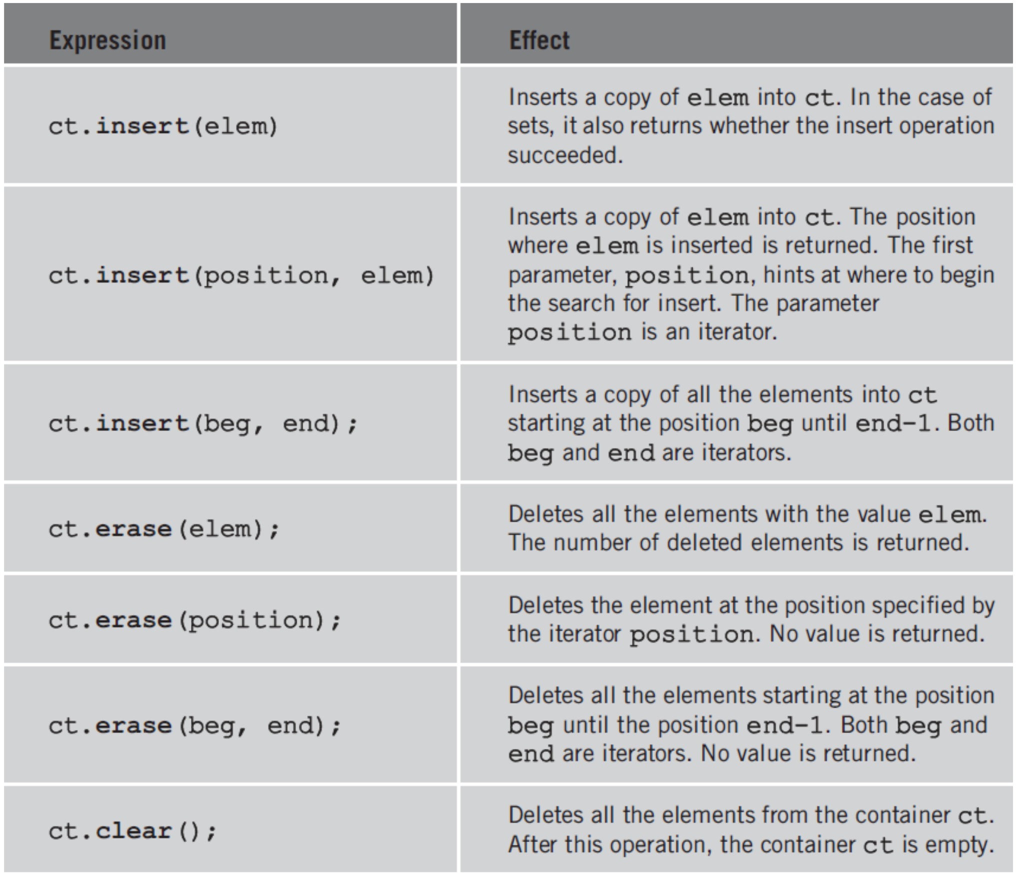
TABLE 13-4 Various ways to declare a `map`/`multimap` container (continued)



Statement	Effect
`ctType<key, elmType> ct(beg, end);`	Creates a `map`/`multimap` container, `ct`. The elements starting at the position `beg` until the position `end-1` are copied into `ct`. Both `beg` and `end` are iterators.
`ctType<key, elmType, sortOp> ct(beg, end);`	Creates a `map`/`multimap` container, `ct`. The elements starting at the position `beg` until the position `end-1` are copied into `ct`. Both `beg` and `end` are iterators. The sort criterion is specified by `sortOp`.

Associative Containers: `map` and `multimap` (cont'd.)

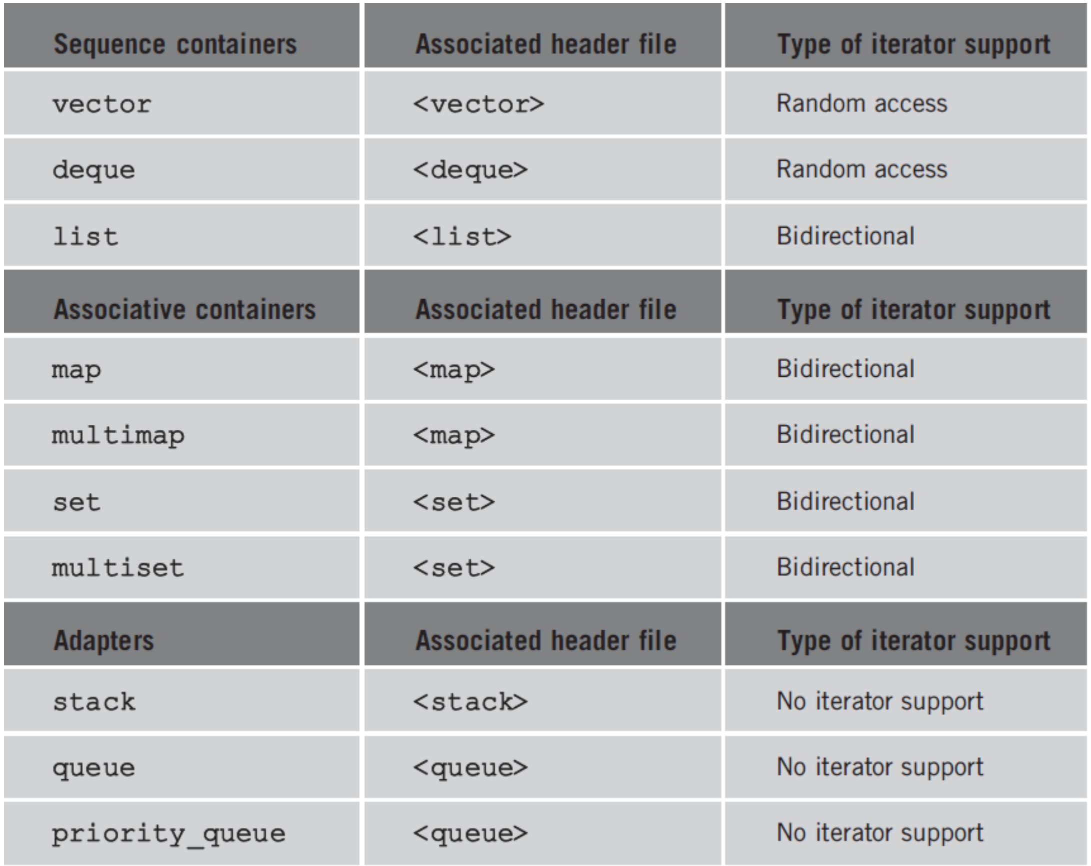
TABLE 13-5 Operations to insert or delete elements from a `map` or `multimap`



Expression	Effect
`ct.insert(elem)`	Inserts a copy of `elem` into `ct`. In the case of sets, it also returns whether the insert operation succeeded.
`ct.insert(position, elem)`	Inserts a copy of `elem` into `ct`. The position where `elem` is inserted is returned. The first parameter, `position`, hints at where to begin the search for insert. The parameter `position` is an iterator.
`ct.insert(beg, end);`	Inserts a copy of all the elements into `ct` starting at the position `beg` until `end-1`. Both `beg` and `end` are iterators.
`ct.erase(elem);`	Deletes all the elements with the value `elem`. The number of deleted elements is returned.
`ct.erase(position);`	Deletes the element at the position specified by the iterator `position`. No value is returned.
`ct.erase(beg, end);`	Deletes all the elements starting at the position `beg` until the position `end-1`. Both `beg` and `end` are iterators. No value is returned.
`ct.clear();`	Deletes all the elements from the container `ct`. After this operation, the container `ct` is empty.

Containers, Associated Header Files, and Iterator Support

TABLE 13-6 Containers, their associated header files, and the type of iterator supported by each container



Sequence containers	Associated header file	Type of iterator support
vector	<vector>	Random access
deque	<deque>	Random access
list	<list>	Bidirectional
Associative containers	**Associated header file**	**Type of iterator support**
map	<map>	Bidirectional
multimap	<map>	Bidirectional
set	<set>	Bidirectional
multiset	<set>	Bidirectional
Adapters	**Associated header file**	**Type of iterator support**
stack	<stack>	No iterator support
queue	<queue>	No iterator support
priority_queue	<queue>	No iterator support

Algorithms

- Some operations specific to a container
  - Provided as part of container definition
- Generic algorithms
  - Common to all containers
    - Contained in header file algorithm
  - Examples
    - Find
    - Sort
    - Merge

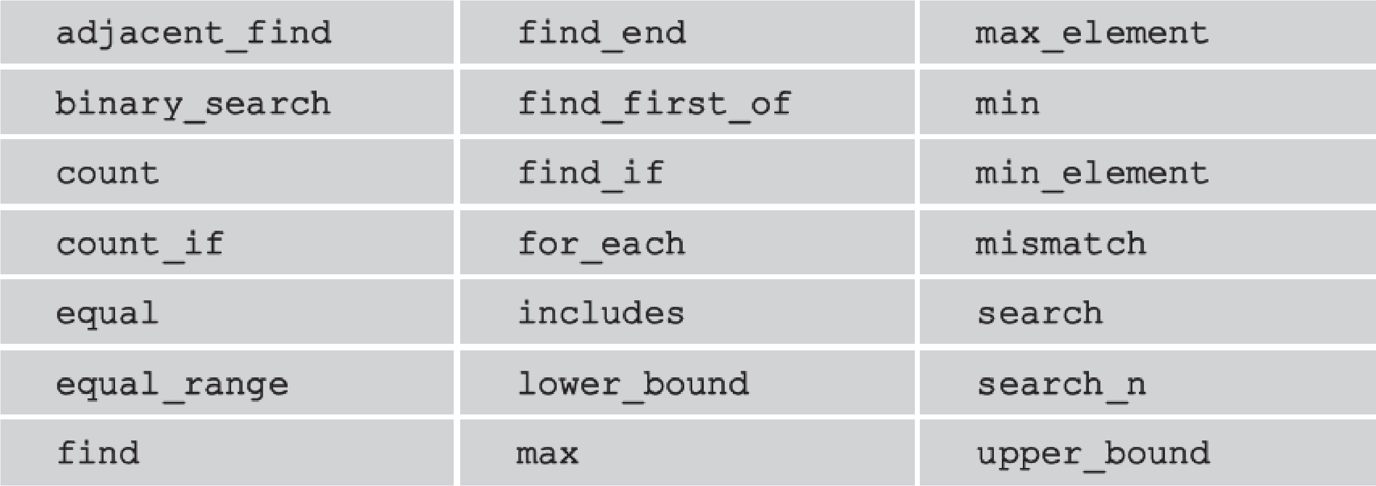
STL Algorithm Classification

- Algorithms may be tied to a specific container
  - Members of a specific class
  - Examples: `clear`, `sort`, `merge`
- Generic algorithms
  - Applied in a variety of situations
- STL generic algorithm classifications
  - Nonmodifying algorithms
  - Modifying algorithms
  - Numeric algorithms
  - Heap algorithms

Nonmodifying Algorithms

- Do not modify container elements
- Investigate the elements

TABLE 13-7 Nonmodifying algorithms



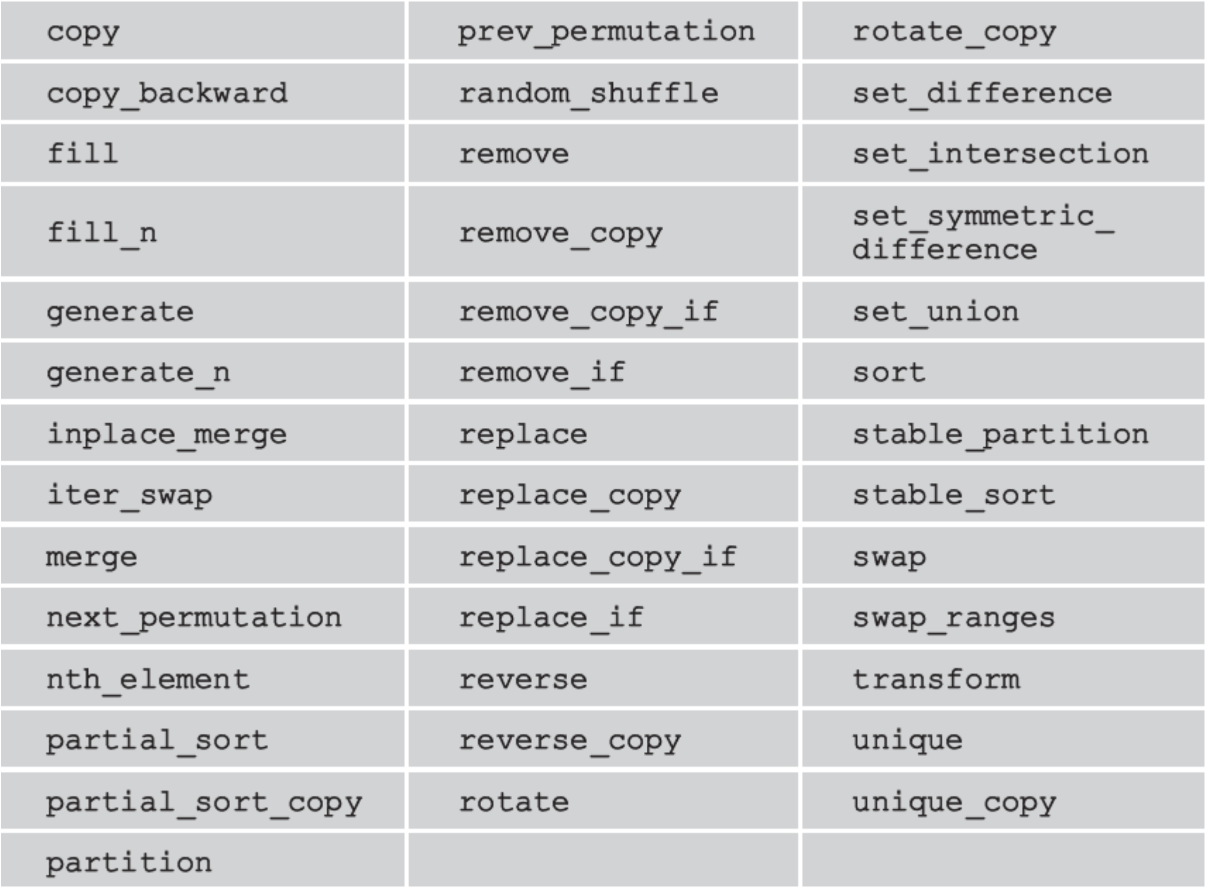
adjacent_find	find_end	max_element
binary_search	find_first_of	min
count	find_if	min_element
count_if	for_each	mismatch
equal	includes	search
equal_range	lower_bound	search_n
find	max	upper_bound

Modifying Algorithms

- Modify container elements by
  - Rearranging, removing, changing element values
- Mutating algorithms
  - Modifying algorithms that change element order
    - Not element values
  - Examples
    - `next_permutation`, `partition`, `prev_permutation`, `random_shuffle`, `reverse`, `reverse_copy`, `rotate`, `rotate_copy`, `stable_partition`

Modifying Algorithms (cont'd.)

TABLE 13-8 Modifying algorithms

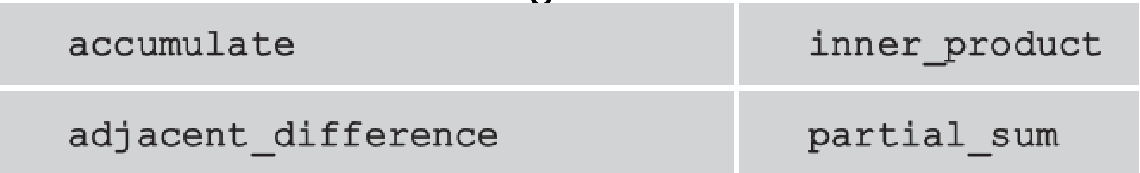


copy	prev_permutation	rotate_copy
copy_backward	random_shuffle	set_difference
fill	remove	set_intersection
fill_n	remove_copy	set_symmetric_difference
generate	remove_copy_if	set_union
generate_n	remove_if	sort
inplace_merge	replace	stable_partition
iter_swap	replace_copy	stable_sort
merge	replace_copy_if	swap
next_permutation	replace_if	swap_ranges
nth_element	reverse	transform
partial_sort	reverse_copy	unique
partial_sort_copy	rotate	unique_copy
partition		

Numeric Algorithms

- Designed to perform numeric calculations container elements

TABLE 13-9 Numeric algorithms

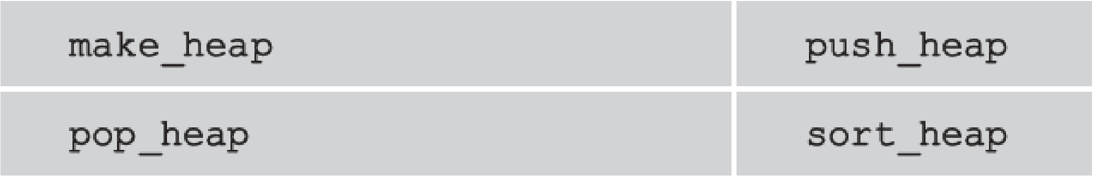


accumulate	inner_product
adjacent_difference	partial_sum

Heap Algorithms

- Based on heapsort algorithm operation

TABLE 13-10 Heap algorithms



`make_heap`	`push_heap`
`pop_heap`	`sort_heap`

Function Objects

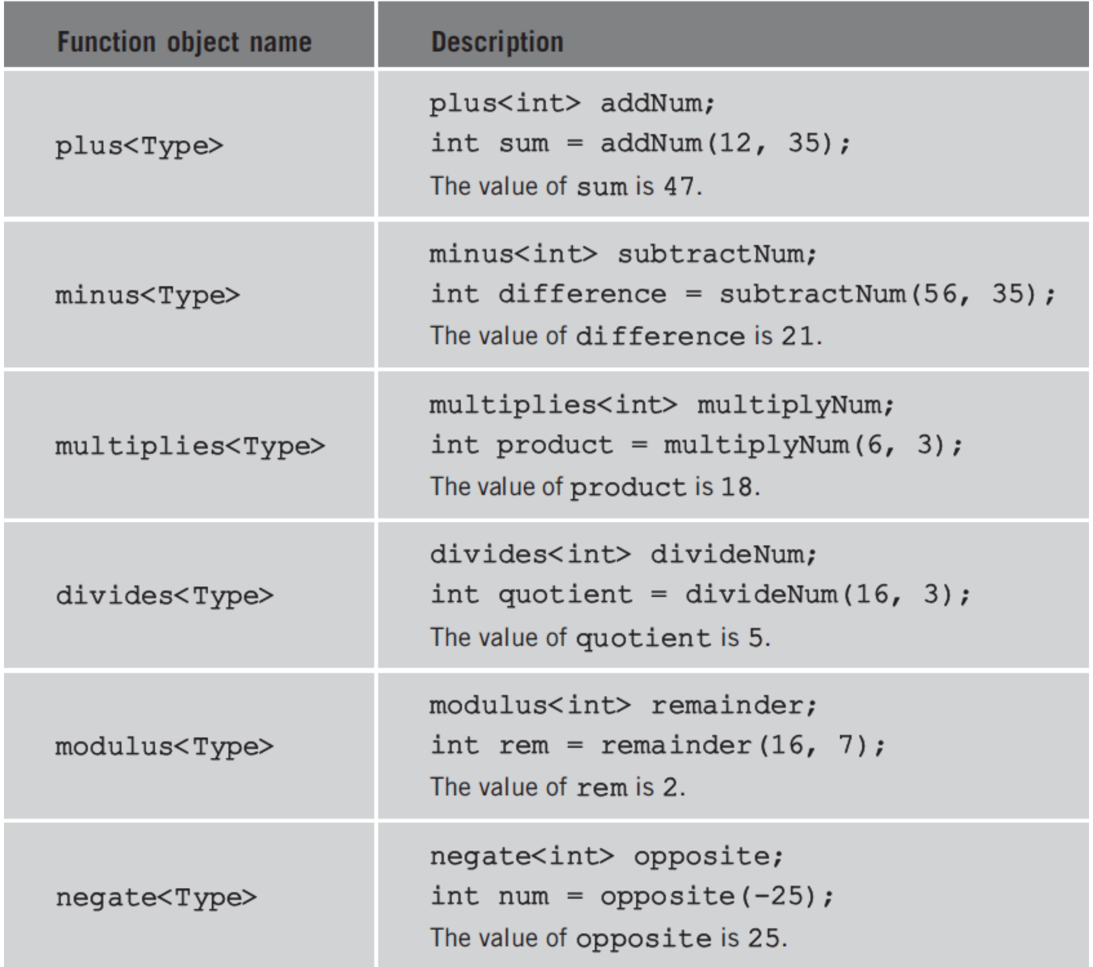
- Generic algorithm flexibility
  - STL provides two forms of an algorithm
    - Using function overloading
- First algorithm form
  - Uses natural operation to accomplish goal
- Second algorithm form
  - User specifies criteria

Function Objects (cont'd.)

- Function object
  - Contains a function
    - Treated as a function using function call operator, ()
  - Class template
    - Overloads the function call operator, ()
  - STL allows creation of own function objects
    - STL provides arithmetic, relational, logical function objects
- STL's function objects
  - Contained in header file `functional`

Function Objects (cont'd.)

TABLE 13-11 Arithmetic STL function objects



Function object name	Description

`plus<Type>`	`plus<int> addNum;`
	`int sum = addNum(12, 35);`
	The value of `sum` is 47.

`minus<Type>`	`minus<int> subtractNum;`
	`int difference = subtractNum(56, 35);`
	The value of `difference` is 21.

`multiplies<Type>`	`multiplies<int> multiplyNum;`
	`int product = multiplyNum(6, 3);`
	The value of `product` is 18.

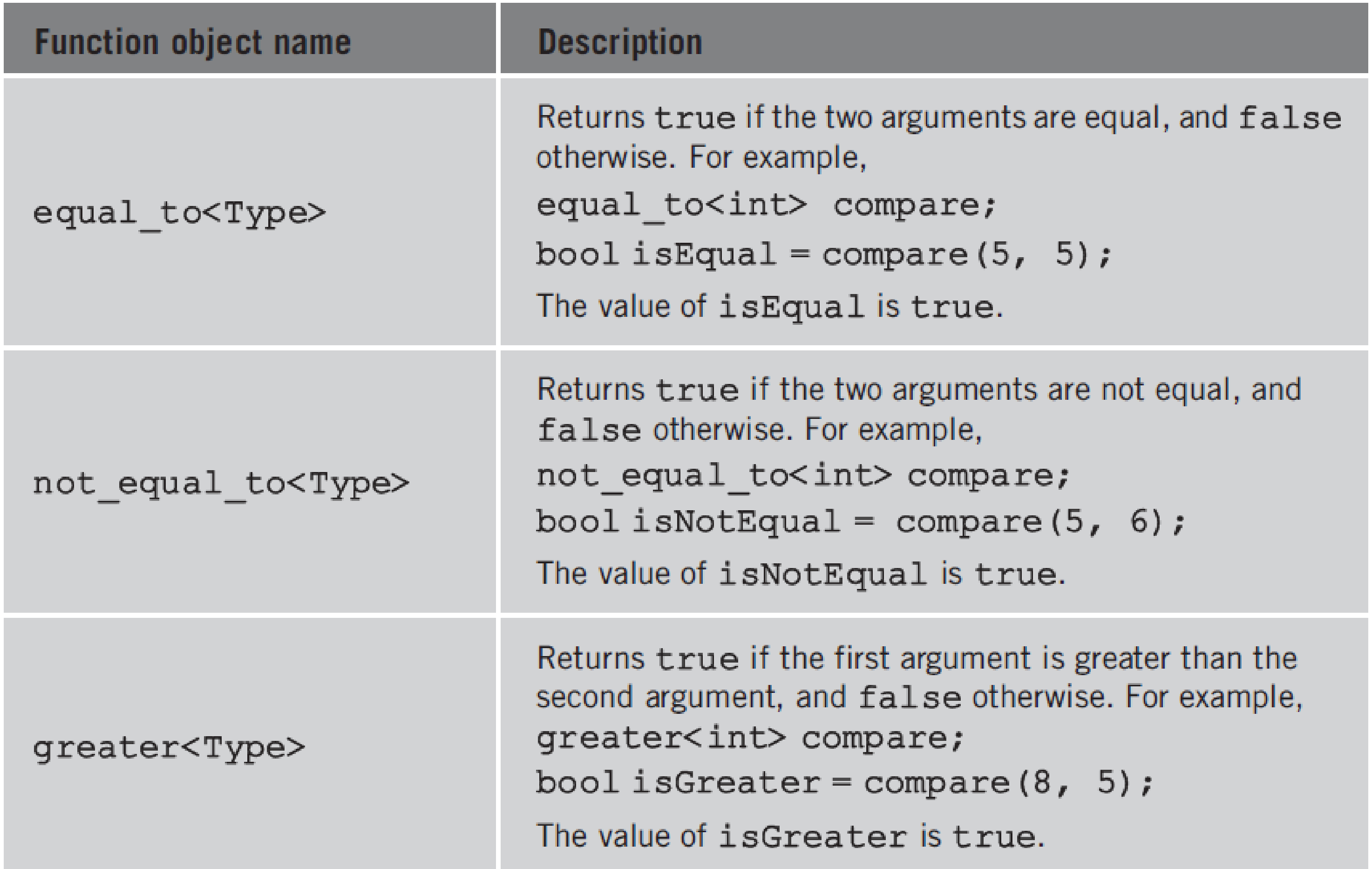
`divides<Type>`	`divides<int> divideNum;`
	`int quotient = divideNum(16, 3);`
	The value of `quotient` is 5.

`modulus<Type>`	`modulus<int> remainder;`
	`int rem = remainder(16, 7);`
	The value of `rem` is 2.

`negate<Type>`	`negate<int> opposite;`
	`int num = opposite(-25);`
	The value of `opposite` is 25.

Function Objects (cont'd.)

TABLE 13-12 Relational STL function objects



Function object name	Description

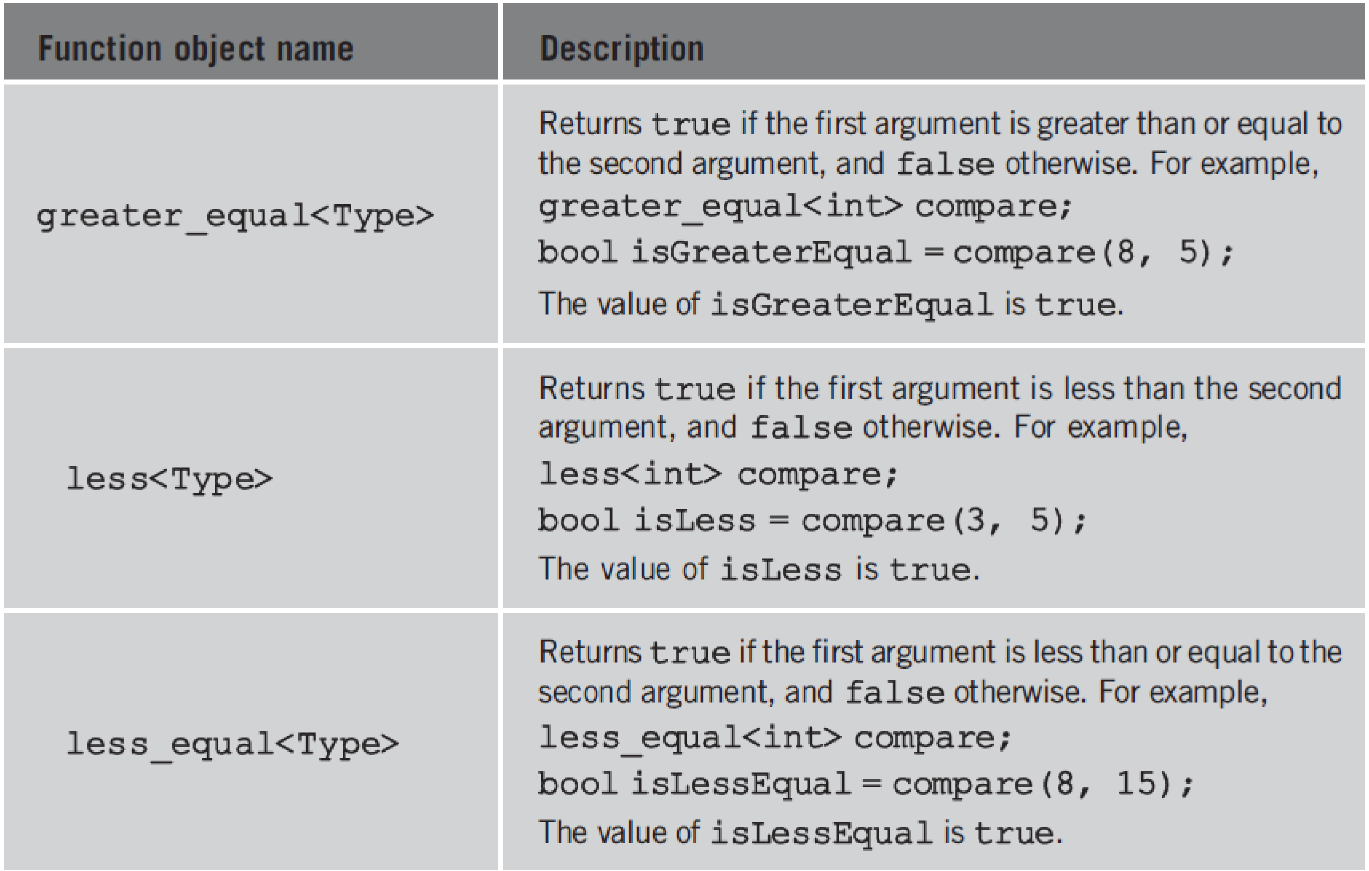
`equal_to<Type>`	Returns `true` if the two arguments are equal, and `false` otherwise. For example,
	`equal_to<int> compare;`
	`bool isEqual = compare(5, 5);`
	The value of `isEqual` is `true`.

`not_equal_to<Type>`	Returns `true` if the two arguments are not equal, and `false` otherwise. For example,
	`not_equal_to<int> compare;`
	`bool isNotEqual = compare(5, 6);`
	The value of `isNotEqual` is `true`.

`greater<Type>`	Returns `true` if the first argument is greater than the second argument, and `false` otherwise. For example,
	`greater<int> compare;`
	`bool isGreater = compare(8, 5);`
	The value of `isGreater` is `true`.

Function Objects (cont'd.)

TABLE 13-12 Relational STL function objects (continued)



Function object name	Description

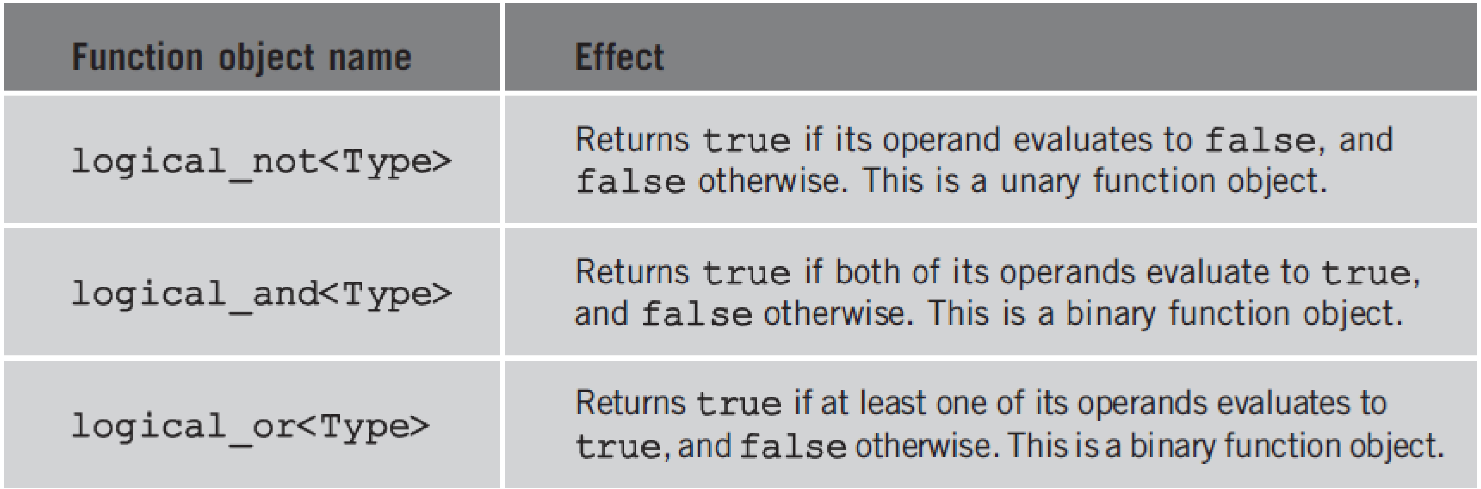
`greater_equal<Type>`	Returns `true` if the first argument is greater than or equal to the second argument, and `false` otherwise. For example,
	`greater_equal<int> compare;`
	`bool isGreaterEqual = compare(8, 5);`
	The value of `isGreaterEqual` is `true`.

`less<Type>`	Returns `true` if the first argument is less than the second argument, and `false` otherwise. For example,
	`less<int> compare;`
	`bool isLess = compare(3, 5);`
	The value of `isLess` is `true`.

`less_equal<Type>`	Returns `true` if the first argument is less than or equal to the second argument, and `false` otherwise. For example,
	`less_equal<int> compare;`
	`bool isLessEqual = compare(8, 15);`
	The value of `isLessEqual` is `true`.

Function Objects (cont'd.)

TABLE 13-13 Logical STL function objects



Function object name	Effect
`logical_not<Type>`	Returns `true` if its operand evaluates to `false`, and `false` otherwise. This is a unary function object.
`logical_and<Type>`	Returns `true` if both of its operands evaluate to `true`, and `false` otherwise. This is a binary function object.
`logical_or<Type>`	Returns `true` if at least one of its operands evaluates to `true`, and `false` otherwise. This is a binary function object.

Predicates

- Special types of function objects
  - Return Boolean values
- Unary predicates
  - Check a specific property for a single argument
- Binary predicates
  - Check a specific property for a pair of (two) arguments

Predicates (cont’d.)

- Typical use
  - Specifying searching, sorting criterion
- In STL
  - Always return same result for same value
- Functions modifying their internal states
  - Cannot be considered predicates

Predicates (cont’d.)

- Insert iterator
  - STL provides three insert iterators
    - To insert elements at destination
- Class `vector`
  - Does not support the `push_front` operation
    - Cannot be used for a vector container

Predicates (cont'd.)

- `back_inserter`
  - Uses the `push_back` operation of the container in place of the assignment operator
- `front_inserter`
  - Uses the `push_front` operation of the container in place of the assignment operator
- `inserter`
  - Uses the container's `insert` operation in place of the assignment operator

STL Algorithms

- Many STL algorithms available
- Section coverage
  - Function prototypes
  - Brief description of what the algorithm does
  - Program showing how to use algorithm
- Section conventions
  - In the function prototypes
    - Parameter types indicate for which type of container the algorithm is applicable
  - Abbreviations used

STL Algorithms (cont'd.)

- Functions `fill` and `fill_n`
  - Function `fill`
    - Fills a container with elements
  - Function `fill_n`
    - Fills in the next *n* elements
- Functions `generate` and `generate_n`
  - Both generate elements and fill a sequence
- Functions `find`, `find_if`, `find_end`, and `find_first_of`
  - All are used to find the elements in a given range

STL Algorithms (cont'd.)

- Functions `remove`, `remove_if`, `remove_copy`, and `remove_copy_if`
  - Function `remove`
    - Removes certain elements from a sequence
  - Function `remove_if`
    - Removes elements from a sequence
    - Using some criterion

STL Algorithms (cont'd.)

- Functions `remove`, `remove_if`, `remove_copy`, and `remove_copy_if` (cont'd.)
  - Function `remove_copy`
    - Copies the elements in a sequence into another sequence
    - By excluding certain elements from the first sequence
  - Function `remove_copy_if`
    - Copies elements in a sequence into another sequence
    - By excluding certain elements, using some criterion, from the first sequence

STL Algorithms (cont'd.)

- Functions `replace`, `replace_if`, `replace_copy`, and `replace_copy_if`
  - Function `replace`
    - Replaces all the occurrences, within a given range, of a given element with a new value
  - Function `replace_if`
    - Replaces the values of the elements, within a given range, satisfying certain criteria with a new value

STL Algorithms (cont'd.)

- Functions `replace`, `replace_if`, `replace_copy`, and `replace_copy_if` (cont'd.)
  - Function `replace_copy`
    - Combination of `replace` and `copy`
  - Function `replace_copy_if`
    - Combination of `replace_if` and `copy`

STL Algorithms (cont'd.)

- Functions `swap`, `iter_swap`, and `swap_ranges`
  - Used to swap elements
- Functions `search`, `search_n`, `sort`, and `binary_search`
  - Used to search elements

STL Algorithms (cont'd.)

- Functions `adjacent_find`, `merge`, and `inplace_merge`
  - Function `adjacent_find`
    - Finds the first occurrence of consecutive elements satisfying a certain criterion
  - Algorithm `merge`
    - Merges two sorted lists
  - Algorithm `inplace_merge`
    - Combines two sorted, consecutive sequences

STL Algorithms (cont'd.)

- Functions `reverse`, `reverse_copy`, `rotate`, and `rotate_copy`
  - Algorithm `reverse`
    - Reverses the order of the elements in a given range
  - Algorithm `reverse_copy`
    - Reverses the elements in a given range while copying into a destination range
    - Source not modified

STL Algorithms (cont'd.)

- Functions `reverse`, `reverse_copy`, `rotate`, and `rotate_copy` (cont'd.)
  - Algorithm `rotate`
    - Rotates the elements in a given range
  - Algorithm `rotate_copy`
    - Copies the elements of the source at the destination in a rotated order

STL Algorithms (cont'd.)

- Functions `count`, `count_if`, `max_element`, `min_element`, and `random_shuffle`
  - Algorithm `count`
    - Counts occurrences of a given value in a given range
  - Algorithm `count_if`
    - Counts occurrences of a given value in a given range satisfying a certain criterion
  - Algorithm `max_element`
    - Determines the largest element in a given range

STL Algorithms (cont'd.)

- Functions `count`, `count_if`, `max_element`, `min_element`, and `random_shuffle` (cont'd.)
  - Algorithm `min_element`
    - Determines the smallest element in a given range
  - Algorithm `random_shuffle`
    - Used to randomly order the elements in a given range

STL Algorithms (cont'd.)

- Functions `for_each` and `transform`
  - Algorithm `for_each`
    - Used to access and process each element in a given range by applying a function, which is passed as a parameter
  - Function `transform`
    - Creates a sequence of elements by applying certain operations to each element in a given range

STL Algorithms (cont'd.)

- Functions `includes`, `set_intersection`, `set_union`, `set_difference`, and `set_symmetric_difference`
  - Algorithm `includes`
    - Determines whether the elements of one range appear in another range
  - Algorithm `set_intersection`
    - Finds the elements that are common to two ranges of elements
  - Algorithm `set_union`
    - Finds the elements that are contained in two ranges of elements

STL Algorithms (cont'd.)

- Functions `includes`, `set_intersection`, `set_union`, `set_difference`, and `set_symmetric_difference` (cont'd.)
  - Algorithm `set_difference`
    - Finds the elements in one range of elements that do not appear in another range of elements
  - Given two ranges of elements, the algorithm `set_symmetric_difference`
    - Determines the elements that are in the first range but not the second range, or the elements that are in the second range but not the first range

STL Algorithms (cont'd.)

- Functions `accumulate`, `adjacent_difference`, `inner_product`, and `partial_sum`
  - All are numerical functions
    - Manipulate numeric data

Summary

- This chapter discussed
  - The Standard Template Library (STL)
  - Associative containers
  - Operations on associative containers
  - Function and algorithms on associative containers
  - Examples of the use of function and algorithms