Matrix Example

6.10 The following program declares two matrices of integers. The first matrix is filled with a series of integral values computed in the nested loops, and the second matrix is derived from the first. Several of the **Matrix**<*Type*> overloaded operators are used, and the resulting matrices are printed.

```
1   #include <COOL/Matrix.h>                          // COOL Matrix class

2   DECLARE Matrix<int>                               // Declare a matrix of integers
3   IMPLEMENT Matrix<int>                             // Implement matrix of integers

4   int main (void) {
5     Matrix<int> mc1(3,4), mc2(3,4);                 // Two 3x4 matrices of integers
6     for (int i = 0; i < 3; i++)                     // For each row in matrix
7       for (int j = 0; j < 4; j++)                   // For each column in matrix
8         mc1.put(i,j,(i+2)*(j+3));                   // Assign element value
9     mc2 = mc1 + 5;                                  // Copy matrix with added value
10    mc1 = mc1 + mc2;                                // Add the matrices together
11    cout << mc1 << "\n" << mc2 << "\n";             // Output the starting matrices
12    exit (0);                                       // Exit with OK status
13  }
```

Line 1 includes the COOL `Matrix.h` class header file. Lines 2 and 3 declare and implement the `Matrix<int>` class. Line 5 declares two `Matrix<int>` variables, each of which have three rows and four columns. Lines 6 through 8 generate a series of integral values that are copied into the elements of the first matrix. Line 9 uses the overloaded addition and assignment operators for the **Matrix**<*Type*> class and computes the value of the second matrix. Line 10 uses the overloaded addition operator to add the two matrices together. Line 11 uses the overloaded output operator to display the contents of each matrix. Finally, the program ends with a valid exit code on line 12.

The following shows the output from the program:

```
17 21 25 29
23 29 35 41
29 37 45 53

11 13 15 17
14 17 20 23
17 21 25 29
```

Matrix<*Type*>& **operator=** (**const Matrix**<*Type*>& *m*);
Overloads the assignment operator for the **Matrix**<*Type*> class and assigns one **Matrix**<*Type*> object to have the value of another by duplicating the size and element values.

Matrix<*Type*>& **operator+=** (**const Matrix**<*Type*>& *m*);
Overloads the addition–with–assignment operator to provide matrix addition for the **Matrix**<*Type*> class. The source is modified to contain the result. If the matrices are of a different size, an **Error** exception is raised.

Matrix<*Type*>& **operator+=** (**const** *Type& value*);
Overloads the addition-with-assignment operator to provide scalar addition for the **Matrix**<*Type*> class. The source is modified to contain the result. If the matrices are of a different size, an **Error** exception is raised.

Matrix<*Type*>& **operator*=** (**const Matrix**<*Type*>& *m*);
Overloads the multiplication-with-assignment operator to provide matrix multiplication for the **Matrix**<*Type*> class. The source is modified to contain the result. If the matrices are of a different size, an **Error** exception is raised.

Matrix<*Type*>& **operator*=** (**const** *Type& value*);
Overloads the multiplication-with-assignment operator to provide scalar multiplication for the **Matrix**<*Type*> class. The source is modified to contain the result

Boolean operator== (**const Matrix**<*Type*>& *m*) **const**;
Overloads the equality operator for the **Matrix**<*Type*> class. This function returns **TRUE** if the matrices have the same number of elements with the same values; otherwise, this function returns **FALSE**.

inline Boolean operator!= (**const Matrix**<*Type*>& *m*) **const**;
Overloads the inequality operator for the **Matrix**<*Type*> class. This function returns **TRUE** if the matrices have a different number of elements or different values.

inline void put (**unsigned int** *row*, **unsigned int** *col*, *Type value*);
Assigns *value* to the element at the specified *row* and *col*. If the row or column specification is out of range, an **Error** exception is raised.

inline int rows () **const**;
Returns the number of rows in the matrix.

inline void set_compare (*Matrix_Compare* = **NULL**);
Updates the compare function for this class of matrix. *Matrix_Compare* is a function of type **Boolean** (**Function*)(**const** *Type&*, **const** *Type&*). If no argument is provided, the **operator==** for the type over which the class is parameterized is used.

Friend Functions:

friend ostream& operator<< (**ostream&** *os*, **const Matrix**<*Type*>& *m*);
Overloads the output operator for a reference to a **Matrix**<*Type*> object *m* to provide a formatted output capability.

inline friend ostream& operator<< (**ostream&** *os*, **const Matrix**<*Type*>* *m*);
Overloads the output operator for a pointer to a **Matrix**<*Type*> object *m* to provide a formatted output capability.

Name: **Matrix**<*Type*> — A parameterized matrix class

Synopsis: **#include** <COOL/Matrix.h>

Base Classes: **Matrix, Generic**

Friend Classes: None

Constructors:

Matrix<*Type*> (**unsigned int** *row*, **unsigned int** *col*);
Allocates enough storage for a matrix of a specific type with the specified number of rows and columns.

Matrix<*Type*> (**unsigned int** *row*, **unsigned int** *col*, **int** *init_num*, ...);
Allocates enough storage for a matrix of the specified type and size. The third argument *init_num* indicates the number of optional initialization values. Matrix elements are initialized in row-major order.

Matrix<*Type*> (**unsigned int** *row*, **unsigned int** *col*, **const** *Type& value*);
Allocates enough storage for a matrix of a specific type with the specified number of rows and columns. In addition, each element of the matrix is initialized to *value*.

Matrix<*Type*> (**const Matrix**<*Type*>& *m*);
Duplicates the size and value of a **Matrix**<*Type*> object *m*.

Member Functions:

inline int columns () **const**;
Returns the number of columns in the matrix.

void fill (**const** *Type& value*);
Sets all elements in the matrix to *value*.

inline Type get (**unsigned int** *row*, **unsigned int** *col*) **const**;
Returns the value of the element at the indicated row and column. If the row or column specification is out of range, an **Error** exception is raised.

Matrix<*Type*> **operator+** (**const Matrix**<*Type*>& *m*) **const**;
Overloads the addition operator to provide matrix addition for the **Matrix**<*Type*> class. A new matrix is returned as the result. If the matrices are of a different size, an **Error** exception is raised.

Matrix<*Type*> **operator+** (**const** *Type& value*) **const**;
Overloads the addition operator to provide scalar addition for the **Matrix**<*Type*> class. A new matrix is returned as the result.

Matrix<*Type*> **operator*** (**const Matrix**<*Type*>& *m*) **const**;
Overloads the multiplication operator to provide matrix multiplication for the **Matrix**<*Type*> class. A new matrix is returned as the result. If the matrices are of a different size, an **Error** exception is raised.

Matrix<*Type*> **operator*** (**const** *Type& value*) **const**;
Overloads the multiplication operator to provide scalar multiplication for the **Matrix**<*Type*> class. A new matrix is returned as the result.

Matrix<*Type*>& **operator=** (**const** *Type& value*);
Overloads the assignment operator for the **Matrix**<*Type*> class and assigns all elements of a matrix to *value*.

inline friend ostream& operator<< (*ostream& os*, **const Queue**<*Type*>* *q*);
Overloads the output operator for a pointer to a **Queue**<*Type*> object to provide a formatted output capability.

Queue Example

6.8 The following program declares a queue of doubles. Random floating-point values are added to the queue in a loop. The elements added are then output. Next, a loop iterates through the elements of the queue by using the current position functionality. If any random number added to the queue is below some arbitrary tolerance, it is removed. Finally, the remaining elements are printed.

```
1   #include <COOL/Queue.h>                    // COOL Queue class
2   #include <COOL/Random.h>                   // COOL Random number class

3   DECLARE Queue<double>;                     // Declare a queue of doubles
4   IMPLEMENT Queue<double>;                   // Implement a queue of doubles

5   int main (void) {
6     Queue<double> q1;                        // Create empty queue
7     Random r (SIMPLE, 1, 3.0, 9.0);          // Simple random generator
8     for (int i = 0; i < 5; i++)              // Put five random numbers
9                 q1.put (r.next ());          // into the queue
10    cout << q1;                              // Output queue elements
11    for (q1.reset(); q1.next(); )            // For each element in queue
12                if (q1.value() < 4.5)        // If less than tolerance
13                          q1.remove ();      // Remove from queue
14    cout << "\n" << q1;                      // Output queue elements
15    exit (0);                                // Exit with OK status
16  }
```

Lines 1 through 4 define the `Queue<double>` class, and line 6 declares an instance of this class. Line 7 declares a random number generator whose values are guaranteed to be within the range 3.0 to 9.0 inclusive (see Section 3, Number Classes, for a discussion about the **Random** class). Lines 8 and 9 contain a simple loop that adds five random numbers to the queue. Line 10 uses **operator<<** for the **Queue** class to output the element values. Lines 11 through 13 use the current position functions to iterate through the elements, removing any entry below an arbitrary tolerance. Finally, the remaining elements are output.

The following shows the output from a sample run of the program:

```
<First in> 6.08322 4.05445 4.85191 6.2072 8.68577 <Last in>
<First in> 6.08322 4.85191 6.2072 8.68577 <Last in>
```

Matrix Class

6.9 The **Matrix**<*Type*> class implements two-dimensional arithmetic matrices for a user-specified numeric data type. Using the parameterized types facility of C++, it is possible, for example, for the user to create a matrix of rational numbers by parameterizing the **Matrix** class over the **Rational** class (see Section 3, Number Classes, for a discussion regarding the **Rational** class). The only requirement for the type is that it support the basic arithmetic operators. Note that unlike the other sequence classes, the **Matrix**<*Type*> class is fixed-size only (that is, it will not grow once the size has been specified to the constructor).

Boolean remove ();
Removes the element at the current position. This function returns **FALSE** if the current position is invalid; otherwise, this function sets the current position to the element immediately following the element removed (if not at end of queue) and returns **TRUE**. If the current position is at the last element before removing, this function invalidates the current position and returns **TRUE** after removing the element.

Boolean remove (**const** *Type& value*);
Searches for *value* and, if found, this function removes and sets the current position to the element immediately following the element removed, and then it returns **TRUE**. If *value* is found but at the end of the queue, this function invalidates the current position and returns **TRUE**. If the element is not found, this function returns **FALSE**.

inline void reset ();
Invalidates the current position.

void resize (**long** *number*);
Resizes the queue for at least *number* of elements. If a growth ratio has been selected and it satisfies the resize request, the queue is grown by this ratio, the current position is invalidated, and **TRUE** is returned. Otherwise, this function returns **FALSE**. If the size specified is zero or negative, an **Error** exception is raised.

inline void set_alloc_size (**int** *size*);
Updates the allocation growth size to be used when the growth ratio is zero. Default allocation growth size is 100 bytes. If the size specified is negative, an **Error** exception is raised.

inline void set_compare (*Queue_Compare* = **NULL**);
Updates the compare function for this class of queue. *Queue_Compare* is a function of type **Boolean** (**Function*)(**const** *Type&*, **const** *Type&*). If no argument is provided, the **operator==** for the type over which the class is parameterized is used.

inline void set_growth_ratio (**float** *ratio*);
Updates the growth ratio for this instance of a queue to *ratio*. When a queue needs to grow, the current size is multiplied by the ratio to determine the new size. If *ratio* is negative, an **Error** exception is raised.

Boolean unget (**const** *Type& value*);
Puts *value* onto the front of the queue. If required and not prohibited, this function grows the queue, puts the first-in item on the queue, and returns **TRUE**. Otherwise, this function returns **FALSE**. If there are no elements in the queue, an **Error** exception is raised.

Type& unput ();
Removes and returns a reference to the last-in item on the queue.

inline Type& value ();
Returns a reference to the element at the current position. If the current position is invalid, an **Error** exception is raised.

Friend Functions:

friend ostream& operator<< (*ostream& os*, **const Queue**<*Type*>& *q*);
Overloads the output operator for a reference to a **Queue**<*Type*> object to provide a formatted output capability.

inline Queue_state& current_position ();
Returns a reference to the state information associated with the current position. This function should be used with the **Iterator**<*Type*> class to save and restore the current position, thus facilitating multiple iterators over an instance of queue.

Boolean find (**const** *Type& value*);
Searches the queue for *value*. If *value* is found, this function sets the current position and returns **TRUE**; otherwise, this function resets the current position and returns **FALSE**.

Type& get ();
Removes and returns a reference to the first-in item on the queue. If there are no elements in the queue, an **Error** exception is raised.

inline Boolean is_empty () **const**;
Returns **TRUE** if there are no items in the queue. Otherwise, this function returns **FALSE**.

inline long length () **const**;
Returns the number of elements in the queue.

inline Type& look ();
Returns the first-in item on the queue. If there are no elements in the queue, an **Error** exception is raised.

Boolean next ();
Advances the current position to the next element in the queue and returns **TRUE**. If the current position is invalid, this function sets the current position to the first element and returns **TRUE**. If the current position is the last element of the queue, this function invalidates the current position and returns **FALSE**.

Queue<*Type*>*&* **operator=** (**const Queue**<*Type*>*& q*);
Overloads the assignment operator for the **Queue** class and assigns *q* to the queue object by duplicating the size and element values. This function invalidates the current position.

Boolean operator== (**const Queue**<*Type*>*& q*) **const**;
Overloads the equality operator for the **Queue** class. This function returns **TRUE** if the queues have an equal number of elements with the same values; otherwise, this function returns **FALSE**.

inline Boolean operator!= (**const Queue**<*Type*>*& q*) **const**;
Overloads the inequality operator for the **Queue** class. This function returns **TRUE** if the queues have an unequal number of elements or unequal values.

Boolean prev ();
Moves the current position pointer to the previous element in the queue and returns **TRUE**. If the current position is invalid, this function moves to the last element and returns **TRUE**. If the current position is the first element in the queue, this function invalidates the current position and returns **FALSE**.

Boolean put (**const** *Type& value*);
Puts *value* onto the back of the queue, making it the last-in item. If required and not prohibited, this function grows the queue, puts the new last-in item on the queue, and returns **TRUE**. Otherwise, this function returns **FALSE**.

Queue Class

6.7 The **Queue**<*Type*> class implements a conventional first-in, first-out data structure that holds a user-specified data type. All memory management and initialization is encapsulated and performed by the class constructors and member functions. Queue objects can be either static-sized or dynamic. Queues are, by default, dynamic in nature. A static-sized queue object is selected by setting the growth allocation size to zero or by passing in a pointer to a block of user-supplied storage to the constructor. If a queue is of static size and an operation is performed that requires more storage, an **Error** exception is raised.

The **Queue**<*Type*> class implements the notion of a current position. This is useful for iterating through the elements of a queue. The current position is maintained in a data member of type **Queue_state** and is set or reset by all member functions affecting elements in the class. Member functions are provided to reset the current position, move to the next and previous elements, find an element, and get the value at the current position. The **Iterator**<*Type*> class provides a mechanism to save and restore the state associated with the current position, thus allowing the programmer to use multiple iterators over the same instance of a queue.

The **Queue**<*Type*> class allows the programmer to add and/or remove items from either end of the queue. In addition, the current position and iterator functions allow the programmer to examine other entries in the middle of the queue and remove or change them. This would be useful in implementing a prioritized queue where the entries may need to be rearranged at times.

Name: **Queue**<*Type*> — A dynamic, parameterized queue

Synopsis: **#include** <COOL/Queue.h>

Base Classes: **Queue, Generic**

Friend Classes: None

Constructors:

Queue<*Type*> ();
Creates an empty queue of the specified type.

Queue<*Type*> (**unsigned long** *number*);
Allocates enough storage for a queue of a specific type to hold *number* of elements specified by the argument.

Queue<*Type*> (**const Queue**<*Type*>& *q*);
Duplicates the size and value of a queue object *q*.

Queue<*Type*> (**void*** *storage*, **unsigned long** *number*);
Creates a static-sized queue object for *number* of elements whose storage *storage* is provided by the user. If an object of this type attempts to grow dynamically or the programmer invokes the **resize** member function, an Error exception is raised.

Member Functions:

inline long capacity () **const**;
Returns the maximum number of elements the stack can contain.

void clear ();
Sets the number of items in the queue to zero. This function invalidates the current position.

Stack Example

6.6 The following program declares a stack capable of initially holding 10 integers. Integer values are pushed onto the stack in a loop. Notice that since more than 10 elements are pushed onto the stack, it must grow automatically and add storage capacity as necessary to hold the extra elements. Finally, these elements are then popped from the stack and printed.

```
1   #include <COOL/Stack.h>                    // COOL Stack class

2   DECLARE Stack<int>;                        // Declare stack of integers
3   IMPLEMENT Stack<int>;                      // Implement stack of integers

4   int main (void) {
5     Stack<int> s1(3);                        // Declare stack of integers
6     for (int i = 1; i <= 5; i++)             // In a small loop, push "n"
7       s1.pushn (i,i);                        // copies of an integer value
8     for (i = 0; i < 5; i++) {                // In another similar loop up to
9       for (int j = 0; j < s1.top(); j++)     // the top element value, get
10        cout << s1.pop();                    // a value from stack and print
11      cout << "\n";                          // Output a newline and repeat
12    }
13    exit (0);                                // Exit with OK status
14  }
```

Lines 1 through 3 define the **Stack** class. Line 5 defines a stack of integers with initial storage for 10 elements. Lines 6 and 7 loop from one through five and push the current loop number on the stack. Thus, the first time through the loop, one element whose value is one is pushed. The second time, two elements whose values are two are pushed, and so on. Because more than 10 elements are added to the stack, an automatic resize is performed by the stack object to accommodate more elements. Because no user-specified growth factor was given, enough storage is allocated to hold 100 elements. Lines 8 through 11 contain nested loops that read the top value on the stack, loop that many times, pop off a value, and print it. After each inner loop completes, a newline character is printed.

The following shows the output from the program:

```
55555
4444
333
22
1
```

inline void set_growth_ratio (float *ratio*);
Updates the growth ratio for this instance of a stack to *ratio*. When a stack needs to grow, the current size is multiplied by the ratio to determine the new size. If *ratio* is negative, an **Error** exception is raised.

inline long set_length (**long** *number*);
Specifies the number of elements in a stack to allow random access via the overloaded **operator[]** member function. If *number* is larger than the storage allocated, this function truncates *number* to the largest value that the allocated size will support. This function returns the new element count. If *number* is negative, an **Error** exception is raised.

inline Type& top ();
Returns (without removing) a reference to the top element of the stack. If the number of elements (that is, length) has been set to a zero-relative index greater than the size of the stack, an **Error** exception is raised.

Friend Functions:

friend ostream& operator<< (*ostream& os*, **const Stack**<*Type*>& *stk*);
Overloads the output operator for a reference to a **Stack**<*Type*> object to provide a formatted output capability.

inline friend ostream& operator<< (*ostream& os*, **const Stack**<*Type*>* *stk*);
Overloads the output operator for a pointer to a **Stack**<*Type*> object to provide a formatted output capability.

Boolean operator== (**const Stack**<*Type*>& *stk*) **const**;
Overloads the equality operator for the **Stack**<*Type*> class. Returns **TRUE** if the stacks have an equal number of elements with the same values; otherwise this function returns **FALSE**.

inline Boolean operator!= (**const Stack**<*Type*>& *stk*) **const**;
Overloads the inequality operator for the **Stack**<*Type*> class. This function returns **TRUE** if the stacks have a unequal number of elements or unequal values; otherwise this function returns **FALSE**.

inline Type& operator[] (**unsigned long** *number*);
Overloads the index operator for the **Stack**<*Type*> class. This function returns a reference to the element of the stack that is *number* of elements from the top of the stack. If *number* is greater than the size of the stack, an **Error** exception is raised.

inline Type& pop ();
Removes and returns a reference to the top element on the stack. If the number of elements (that is, length) has been set to a zero-relative index greater than the size of the stack, an **Error** exception is raised.

Type& popn (**long** *n*);
Pops *n* elements off the stack, returning a reference to the last one popped off the stack. With an argument of zero, this function returns the top item of the stack without removing it. If the number of elements to pop is negative, an **Error** exception is raised.

long position (**const** *Type& value)* **const**;
Searches the stack for *value*. If *value* is found, this function returns the zero-relative index, from the top of the stack, of that element; otherwise, this function returns –1.

inline Boolean push (**const** *Type& value*);
Pushes *value* onto the top of a stack. If required and not prohibited, this function grows the stack object. This function returns **TRUE** if successful; otherwise, this function returns **FALSE**. If the stack is prohibited from growing dynamically, an **Error** exception is raised.

Boolean pushn (**const** *Type& value*, *long n*);
Pushes *n* items onto the top of the stack, all of which have the specified *value*. When *n* is zero, this function replaces the top item on the stack with *value*. If required and not prohibited, this function grows the stack object. This function returns **TRUE** if successful; otherwise, this function returns **FALSE**. If the stack is prohibited from growing dynamically, an **Error** exception is raised.

void resize (**long** *number*);
Resizes the stack for at least *number* of elements. If a growth ratio has been selected and it satisfies the resize request, the stack is grown by this ratio. If the stack is prohibited from dynamically growing, an **Error** exception is raised.

inline void set_alloc_size (**int** *size*);
Updates the allocation growth size to be used when the growth ratio is zero. Default allocation growth size is 100 bytes. Setting the allocation growth size to zero results in a static-sized object. If *size* is zero or negative, an **Error** exception is raised.

inline void set_compare (*Stack_Compare* = **NULL**);
Sets the compare function for this class of stack. *Stack_Compare* is a user-defined function of type **Boolean** (**Function*)(**const** *Type&*, **const** *Type&*). If no such function is provided, the **operator==** for the type over which the class is parameterized is used.

Stack Class

6.5 The **Stack**<*Type*> class implements a conventional first-in, last-out data structure that holds a user-specified data type. All memory management and initialization is encapsulated and performed by the class constructors and member functions. Stack objects can be either static-sized or dynamic. Stacks are, by default, dynamic in nature. A static-sized stack object is selected by setting the growth allocation size to zero or by passing in a pointer to a block of user-supplied storage to the constructor. If a stack is of static size and an operation is performed that requires more storage, an **Error** exception is raised.

Name: **Stack**<*Type*> — A dynamic, parameterized stack

Synopsis: **#include** <COOL/Stack.h>

Base Classes: **Stack**, **Generic**

Friend Classes: None

Constructors:

Stack<*Type*> ();
Creates an empty stack of the specified type.

Stack<*Type*> (**unsigned long** *number*);
Allocates enough storage for a stack of a specific type to hold *number* of elements.

Stack<*Type*> (**const Stack**<*Type*>& *stk*);
Duplicates the size and value of a stack object *stk*.

Stack<*Type*> (**void*** *storage*, **unsigned long** *number*);
Creates a static-sized stack object for *number* of elements whose storage *storage* is provided by the user. If an object of this type attempts to grow dynamically or the programmer invokes the **resize** member function, an Error exception is raised.

Member Functions:

inline long capacity () **const**;
Returns the maximum number of elements the stack can contain.

inline void clear ();
Sets the number of elements in the stack to zero.

Boolean find (**const** *Type& value*);
Searches the stack for *value*. If *value* is found, this function returns **TRUE**; otherwise, this function returns **FALSE**.

inline Boolean is_empty () **const**;
Returns **TRUE** if the stack has no elements; otherwise, this function returns **FALSE**.

inline long length () **const**;
Returns the number of elements in the stack.

Stack<*Type*>& **operator=** (**const Stack**<*Type*>& *stk*);
Overloads the assignment operator for the **Stack**<*Type*> class and assigns *stk* to the stack object by duplicating the size and element values. If the stack object is prohibited from dynamically growing, an **Error** exception is raised.

Vector Example

6.4 The following program declares a vector of five strings whose contents are initialized with state names. The element values are first printed by using **operator<<**, then sorted in reverse alphabetical order, and finally printed by iterating through the vector by using the current position functions.

```
1   #include <COOL/String.h>                    // COOL String class
2   #include <COOL/Vector.h>                    // COOL Vector class

3   DECLARE Vector<String>;                     // Declare vector of strings
4   IMPLEMENT Vector<String>;                   // Implement vector of strings

5   Boolean my_compare (const String& s1, const String& s2) {
6     return ((s1 <= s2) ? FALSE : TRUE);       // Reverse alphabetize
7   }

8   int main (void) {
9     Vector<String> v1(5);                     // Declare vector of strings
10    v1.push ("Texas");                        // Add "Texas"
11    v1.push ("Alaska");                       // Add "Alaska"
12    v1.push ("New York");                     // Add "New York"
13    v1.push ("Alabama");                      // Add "Alabama"
14    v1.push ("North Dakota");                 // Add "North Dakota"
15    cout << v1 << "\n";                       // Output the vector
16    v1.sort (my_compare);                     // Reverse sort the vector
17    for (v1.reset(); v1.next(); )             // For each element
18      cout << v1.value() << "\n";             // Output the value
19    exit (0);                                 // Exit with OK status
20    }
```

Lines 1 through 4 define the **Vector** and **String** classes. Lines 5 through 7 declare a simple sort function that reverses the lexical comparison test performed by **operator<=** in the **String** class. Line 9 defines a vector of strings with initial storage for five elements. Lines 10 through 14 push five literal character strings into the vector. Line 15 uses **operator<<** for the **Vector** class to output the element values. Line 16 sorts the vector according to the predicate function provided. Finally, lines 17 and 18 use the current position functions to iterate through the elements, printing each one.

The following shows the output for the program:

```
Texas Alaska New York Alabama North Dakota
Texas
North Dakota
New York
Alaska
Alabama
```

inline void set_compare (*Vector_Compare* = **NULL**);
Updates the compare function for this class of vector. *Vector_Compare* is a function of type **Boolean** (**Function*)(**const** *Type&*, **const** *Type&*). If no argument is provided, the **operator==** for the type over which the vector is parameterized is used..

inline void set_growth_ratio (**float** *ratio*);
Updates the growth ratio for this instance of a vector to the specified value. When a vector needs to grow, the current size is multiplied by the ratio to determine the new size. If *ratio* is negative, an **Error** exception is raised.

inline long set_length (*long*);
Specifies the number of elements in a vector to allow random access via the overloaded **operator[]** member function. If the number requested is larger than the storage allocated, this function truncates to the largest value that the allocated size will support. This function returns the updated number of elements. If the length is negative, an **Error** exception is raised.

void sort (*Predicate p*);
Sorts the elements of a vector by using the supplied predicate for determining the collating sequence and invalidates the current position. *Predicate* is a user-defined function of type **int** (**Function*) (**const** *Type&*, **const** *Type&*) that returns –1 if the first argument should precede the second, zero if they are equal, and 1 if the first argument should follow the second.

inline Type& value ();
Returns a reference to the element at the current position. If the current position is invalid, an **Error** exception is raised.

Friend Functions:

friend ostream& operator<< (*ostream& os*, **const Vector**<*Type*>& *vec*);
Overloads the output operator for a reference to a **Vector**<*Type*> object to provide a formatted output capability.

friend ostream& operator<< (*ostream& os*, **const Vector**<*Type*>* *vec*);
Overloads the output operator for a pointer to a **Vector**<*Type*> object to provide a formatted output capability.

inline Boolean put (**const** *Type& value*, **long** *n*)
Replaces the *n*th (zero-relative) element in the object with *value*. This function returns **TRUE** if the *n*th element exists; otherwise, this function returns **FALSE**. If the index is negative, an **Error** exception is raised.

Type remove ();
Removes and returns the element at the current position. This function sets the current position to the element immediately following the element removed. If the element is found but at the end of the vector, this function invalidates the current position.

Boolean remove (**const** *Type& value*);
Searches for *value* and, if found, this function removes and sets the current position to the element immediately following the element removed, and then it returns **TRUE**. If *value* is found but at the end of the vector, this function invalidates the current position and returns **TRUE**. If *value* is not found, this function returns **FALSE**.

Boolean remove_duplicates ();
Removes any duplicate elements from the vector and invalidates the current position. This function returns **TRUE** if any elements were removed; otherwise, this function returns **FALSE**.

Boolean replace (**const** *Type& value1*, **const** *Type& value2*);
Replaces the first occurrence of *value2* with *value1*. If *value2* is found, this function returns **TRUE**; otherwise, this function returns **FALSE**.

Boolean replace_all (**const** *Type& value1*, **const** *Type& value2*);
Replaces all occurrences of *value2* with *value1* and sets the current position to the last replaced element. This function returns **TRUE** if any elements were replaced; otherwise, this function returns **FALSE**.

inline void reset ();
Invalidates the current position.

void resize (**unsigned long** *size*);
Resizes the vector for at least *size* number of elements and invalidates the current position. If a growth ratio has been selected and it satisfies the resize request, the vector is grown by this ratio. If the new size is negative, an **Error** exception is raised.

void reverse ();
Reverses the order of elements in a vector and invalidates the current position.

Boolean search (**const Vector**<*Type*>& *vec*, **long** *start*=0, **long** *end*=–1);
Searches within the specified range (*start* inclusive, *end* exclusive) of a vector object for a sequence *vec*. If *end* is equal to minus one (the default), all elements from *start* to the end of the vector are filled. If the sequence is found, this function sets the current position in the destination vector to the start of the matched sequence and returns **TRUE**; otherwise, this function returns **FALSE**.

inline void set_alloc_size (**int** *size*);
Updates the allocation growth size for all instances of the class to be used when the growth ratio is zero. Default allocation growth size is 100 bytes. Setting the allocation growth size to zero results in a static-sized object. If the size specified is negative, an **Error** exception is raised.

Boolean operator== (**const Vector**<*Type*>*& vec*) **const**;
Overloads the equality operator for the **Vector**<*Type*> class and returns **TRUE** if the vector object has the same number of elements with the same values as *vec*; otherwise, this function returns **FALSE**.

inline Boolean operator!= (**const Vector**<*Type*>*& vec*) **const**;
Overloads the inequality operator for the **Vector**<*Type*> class and returns **TRUE** if the vector object does not have the same number of elements or the same values as *vec*; otherwise, this function returns **FALSE**.

inline Type& operator[] (**unsigned long** *index*) **const**;
Overloads the brackets operator for the **Vector**<*Type*> class and returns a reference to an individual element from the vector at the zero-relative *index* specified. If *index* is invalid or out of range, an **Error** exception is raised. You should be careful when using **operator[]** because an index is out of range if it is greater than the number of elements in the vector. If random access to all allocated space is desired, first use the **set_length()** function.

Type& pop ();
Returns a reference to the last element in the vector and invalidates the current position.

inline long position () **const**;
Returns the current position as a zero-relative index into the vector that can be used with the overloaded **operator[]**.

inline long position (**const** *Type& value*) **const**;
Searches the vector for *value*. If the element is found, this function updates the current position and returns the zero-relative index of the element; otherwise, this function invalidates the current position and returns –1.

Boolean prepend (**const Vector**<*Type*>*& vec*);
Inserts the elements of one vector *vec* at the beginning of a the vector object. The current position is set to the new position of the first element of the old destination vector. If required and not prohibited, this function grows the destination vector and returns **TRUE**; otherwise, this function returns **FALSE**.

inline Boolean prev ();
Moves the current position pointer to the previous element in the vector and returns **TRUE**. If the current position is invalid, this function moves to the last element and returns **TRUE**. If the current position is the first element in the vector, this function invalidates the current position and returns **FALSE**.

Boolean push (**const** *Type& value*);
Adds *value* to the end of a vector. If required and not prohibited, this function grows the vector object. This function returns **TRUE** if successful; otherwise, this function returns **FALSE**. This function sets the current position to point to the new element added.

Boolean push_new (**const** *Type& value*);
Adds *value* to the end of a vector if it is not already in the vector. If required and not prohibited, this function grows the vector object. This function returns **TRUE** if successful; otherwise, this function returns **FALSE**. This function sets the current position to point to the new element added.

Boolean insert_after (**const** *Type& value*);
Inserts (not replaces) the element *value* after the current position and does not change the current position. If the current position is invalid, this function returns **FALSE**. If required and not prohibited, this function grows the target vector and returns **TRUE**; otherwise, this function returns **FALSE**.

Boolean insert_after (**const** *Type& value*, *long index*);
Inserts (not replaces) the element *value* after the specified zero-relative *index* and updates the current position to the specified index. If the index is out of range, this function returns **FALSE**. If required and not prohibited, this function grows the target vector and returns **TRUE**; otherwise, this function returns **FALSE**.

Boolean insert_before (**const** *Type& value*);
Inserts (not replaces) the element *value* before the current position and advances the current position one element, thus leaving it pointing at the same element. If the current position is invalid**,** this function returns **FALSE**. If required and not prohibited, this function grows the target vector and returns **TRUE**; otherwise, this function returns **FALSE**.

Boolean insert_before (**const** *Type& value*, *long index*);
Inserts (not replaces) the element *value* before the specified zero-relative *index* and updates the current position to the specified index. If the index is out of range, this function returns **FALSE**. If required and not prohibited, this function grows the target vector and returns **TRUE**; otherwise, this function returns **FALSE**.

inline Boolean is_empty ();
Returns **TRUE** if the vector contains no entries; otherwise, returns **FALSE**.

inline long length () **const**;
Returns the number of elements in the vector.

void merge (**const Vector**<*Type*>& *vec*, *Predicate p*);
Merges the elements of one vector into another by using the supplied predicate for determining the collating sequence. The current position in the destination vector is invalidated. *Predicate* is a user-defined function of type **int** (**Function*) (**const** *Type&*, **const** *Type&*) that returns –1 if the first argument should precede the second, zero if they are equal, and 1 if the first argument should follow the second.

inline Boolean next ();
Advances the current position to the next element in the vector and returns **TRUE**. If the current position is invalid, this function sets the current position to the first element and returns **TRUE**. If the current position is the last element of the vector, this function invalidates the current position and returns **FALSE**.

Vector<*Type*>& **operator=** (**const** *Type& value*);
Overloads the assignment operator for the **Vector**<*Type*> class and assigns all elements *value*. If dynamic growth of the vector is prohibited, an **Error** exception is raised. If there is enough room, the current position is invalidated and a reference to the vector is returned.

Vector<*Type*>& **operator=** (**const Vector**<*Type*>*& vec*);
Overloads the assignment operator for the **Vector**<*Type*> class and assigns *vec* to the vector object, duplicating the size and element values. The current position in the destination vector is invalidated. A reference to the vector object is returned.

Vector<*Type*> (**unsigned long** *number*, **int** *init_num*, ...);
Allocates enough storage for a vector of a specific type to hold *number* elements. The second argument *init_num* specifies the number of optional initialization values provided for consecutive elements of the vector. Any remaining elements are not initialized.

Vector<*Type*> (**Vector**<*Type*>*& vec*);
Duplicates the size and value of another vector object *vec*. Element values are copied by **operator=** for the type specified.

Vector<*Type*> (**void*** *storeage*, **unsigned long** *number*);
Creates a static-sized vector object for *number* elements whose storage *storeage* is provided by the user. If a vector object created in this manner attempts to grow dynamically or the resize member function is invoked, an **Error** exception is raised.

Member Functions:

Boolean append (**const Vector**<*Type*>*& vec*);
Adds the elements of vector *vec* to the end of a vector object. The current position in the vector object is set to the position of the last element of *vec*. If required and not prohibited, this function grows the destination vector and returns **TRUE**; otherwise, this function returns **FALSE**.

inline long capacity () **const**;
Returns the maximum number of elements the vector can contain without growing.

void clear ()
Removes all elements in the object and invalidates the current position.

void copy (**const Vector**<*Type*>*& vec*, **unsigned long** *start* = 0,
long *end* = –1);
Copies the specified range (*start* inclusive and *end* exclusive) from the source vector *vec* to the vector object. The destination vector will grow if necessary and if allowed. The current position is set to the last element copied into the destination. If *end* is equal to minus one (the default), all elements from *start* to the end of the vector are copied. If the *start* or *end* or both indexes are invalid or the vector needs to grow dynamically and this is prohibited, an **Error** exception is raised.

inline Vector_state& current_position ();
Returns a reference to the state information associated with the current position. This function should be used with the **Iterator**<*Type*> class to save and restore the current position, thus facilitating multiple iterators over an instance of vector.

void fill (**const** *Type& value*, **unsigned long** *start* = 0, **long** *end* = –1);
Sets all elements within the specified range (*start* inclusive and *end* exclusive) to *value* and invalidates the current position. If *end* is equal to minus one (the default), all elements from *start* to the end of the vector are filled. If the *start* or *end* or both indexes are invalid, an **Error** exception is raised.

Boolean find (**const** *Type& value*, **unsigned long** *start* = 0);
Searches the vector for *value* beginning at the specified *start* index. If the element is found, this function sets the current position and returns **TRUE**; otherwise, this function invalidates the current position and returns **FALSE**.

inline *Type&* **get** (**int** *n*)
Returns a reference to the *n*th (zero-relative) element in the object. This function sets the current position to this element. If the index is negative or out of range, an **Error** exception is raised.

Vector Class

6.3 The **Vector**<*Type*> class implements one-dimensional vectors of a user-specified type. All memory management and initialization is encapsulated and performed by the class constructors and member functions. Vector objects can be either static-sized or dynamic. Vectors are, by default, dynamic in nature. A static-sized vector object is selected by setting the growth allocation size to zero or by passing in a pointer to a block of user-supplied storage to the constructor. If a vector is of static size and an operation is performed that requires more storage, an **Error** exception is raised.

The **Vector**<*Type*> class implements the notion of a current position. This is useful for iterating through the elements of a vector. The current position is maintained in a data member of type **Vector_state** and is set or reset by all member functions affecting elements in the class. Member functions are provided to reset the current position, move to the next and previous elements, find an element, and get the value at the current position. The **Iterator**<*Type*> class provides a mechanism to save and restore the state associated with the current position, thus allowing the programmer to use multiple iterators over the same instance of a vector.

The **Vector**<*Type*> class follows conventional object-oriented programming techniques and encapsulates the actual data elements from the user. The advantage of this approach is that the class can automatically manage memory, maintain the element count, and be aware of any changes made to the vector. The user of the class facilitates this operation by using the **insert**, **push**, **pop**, and **remove** member functions and their variants. However, the **Vector**<*Type*> class also overloads the **operator[]** and allows the user to access a specific element directly. This is done partly for efficiency and partly for compatibility with past usage.

The drawback of this approach, however, is that the object may not always know its current state. For example, a newly declared vector object has no elements. Each use of **push** to add an element will increment the element count by one. However, elements added at random locations via the **operator[]** will not be counted. A user may get unexpected results by mixing these approaches. For this reason, the **set_length**() member function allows the user to manually set the element count before use of **operator[]** for random-access write operations.

Name: **Vector**<*Type*> — A dynamic, parameterized vector class

Synopsis: **#include** <COOL/Vector.h>

Base Classes: **Vector, Generic**

Friend Classes: None

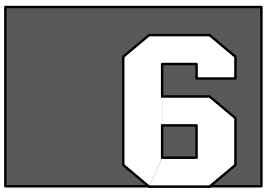
Constructors:

Vector<*Type*> ();
Creates an empty vector of the specified type.

Vector<*Type*> (**unsigned long** *number*);
Allocates enough storage for a vector of a specific type to hold *number* elements. Elements are not initialized.

Vector<*Type*> (**unsigned long** *number*, **const** *Type& value*);
Allocates enough storage for a vector of a specific type to hold *number* elements, each of which is initialized with *value*.

ORDERED SEQUENCE CLASSES



Introduction

6.1 The ordered sequence classes are a collection of basic data structures that implement sequential access data structures as parameterized classes, thus allowing the user to customize a generic template to create a specific user-defined class. The following classes are discussed in this section:

- **Vector**<*Type*>
- **Stack**<*Type*>
- **Queue**<*Type*>
- **Matrix**<*Type*>

The **Vector**<*Type*> class implements dynamic, one-dimensional vectors supporting such functions as insert, delete, replace, search, reverse, print, and sort. The **Stack**<*Type*> class implements dynamic stacks with the functions push, pop, find, position, and empty. The **Queue**<*Type*> class implements a dynamic, circular buffer queue with support for get, unget, put, and unput to access elements at either end of the queue. The **Matrix**<*Type*> class implements static-sized, two-dimensional matrices with support for the basic arithmetic operations. The **Vector**<*Type*> and **Queue**<*Type*> classes support the notion of a current position. See Section 5, Parameterized Templates, for more information regarding the current position mechanism and the **Iterator**<*Type*> class.

In order to achieve successful compilation and usage, certain operations must be supported by any user-specified type over which a sequence class is parameterized. The member functions **operator=**, **operator<**, **operator>**, **operator==**, and **operator<<** for both pointer and reference must be overloaded for any class object used as the type. In addition, the **Matrix**<*Type*> class requires the supplied type to support **operator+**, **operator–**, **operator/**, and **operator***. Note that built-in types already have these functions defined.

NOTE: The ordered sequence classes use **operator=** of the parameterized type when copying elements. You should be careful when parameterizing an ordered sequence class over a pointer to a type, since the default pointer assignment operator usually copies the pointer, not the value pointed at.

Requirements

6.2 This section discusses the parameterized ordered sequence container classes. It assumes that you have read and understood Section 5, Parameterized Templates. In addition, no attempt is made to discuss the concepts and algorithms for the data structures discussed. You should refer to a general data structures or computer science text for this information.

Printed on: Wed Apr 18 07:06:39 1990

Last saved on: Tue Apr 17 14:01:07 1990

Document: s6

For: skc