1 Background Information

Suppose your research topic involves operations with n-dimensional vectors of the form $a = (a_0, a_1, \cdots, a_{n-1})$ with n elements a_k with $0 \leq k < n$ and $n \geq 1$.

To implement the operations, you have written a Java class named **Vec** equipped with operations named **add** , **subtract**, **multiply**, **read**, **write**, etc. Although you are very satisfied with the results of your computations, you are not too happy about coding the expressions in your computations. For example, to code the expression $2 * a/3 + a - b/4$ you would code something like this:

```
int n = 100; // cin >> n;

Vec a = new Vec(n); // create a Vec
a.read(); // initialize a's elements

Vec b = new Vec(n); // create another Vec
b.read(); // initialize b's elements
```

Here is one way to implement $2 * a/3 + a - b/4$:

```
Vec temp1 = a.multiply(2);
Vec temp2 = temp1.divide(3);
Vec temp3 = temp2.add(a);
Vec temp4 = b.divide(4);
Vec result = temp3.subtract(temp4);
result.output();
```

Here is another way to implement $2 * a/3 + a - b/4$:

```
Vec result = a.multiply(2).divide(3).add(a).subtract(b.divide(4));
```

As you can see, it is just not obvious that either of the above two implementations computes $2 * a/3 + a - b/4$. Thus, the more expressions you need to implement, the more time you'll spend on verification of your implementation.

Ideally, you would like to be able to express the expression $2 * a/3 + a - b/4$ like this:

```
Vec result = 2 * a / 3 + a - b / 4;
```

or like this, putting it all together:

```
int n = 100; // cin >> n;
Vec a(n), b(n);
cin >> a >> b;
Vec result = 2 * a / 3 + a - b / 4;
cout << result;
```

2 Operator Overloading

2.1 Problem Statement

Design and implement a **Vec** class that represents n-dimensional vectors of the form $a = (a_0, a_1, \cdots, a_{n-1})$ with n **int** elements a_k with $0 \leq k < n$ and $n \geq 1$.

2.2 Notation

Let a $= (a_0, a_1, \cdots, a_{n-1})$, b $= (b_0, b_1, \cdots, b_{n-1})$, c $= (c_0, c_1, \cdots, c_{n-1})$ be n-dimensional vectors, x $= (x_0, x_1, \cdots, x_{m-1})$ an m-dimensional vector, and v and integer value, with $m \geq 1$ and $n \geq 1$.

2.3 Common Vec Operations

Operation	Definition
+a	$(a_0, a_1, \cdots, a_{n-1})$
–a	$(-a_0, -a_1, \cdots, -a_{n-1})$
a + b	$(a_0 + b_0, a_1 + b_1, \cdots, a_{n-1} + b_{n-1})$
a – b	$(a_0 - b_0, a_1 - b_1, \cdots, a_{n-1} - b_{n-1})$
a += b	a = a + b
a –= b	a = a – b
a += v	$(a_0 + v, a_1 + v, \cdots, a_{n-1} + v)$
a –= v	$(a_0 - v, a_1 - v, \cdots, a_{n-1} - v)$
a *= v	$(a_0 * v, a_1 * v, \cdots, a_{n-1} * v)$
a /= v	$(a_0/v, a_1/v, \cdots, a_{n-1}/v)$
a %= v	$(a_0\%v, a_1\%v, \cdots, a_{n-1}\%v)$

a++	Increments **a** by 1, and returns a copy of the original **a**.
a−−	Decrements **a** by 1, and returns a copy of the original **a**.
++a	Increments **a** by 1 and returns the resulting **a**.
−−a	Decrements **a** by 1 and returns the resulting **a**.
a == x	Evaluates to true if $n = m$, and $a_k == x_k$, for all $k = 0, 1, \cdots, n-1$
a < x	Let i be the first index, if any, such that $a_i \neq x_i$. If i exists, then **a < x** evaluates to the value of $a_i < x_i$; otherwise, **a < x** evaluates to the value of $n < m$.
a != x	!(a == x)
a <= x	(a < x) or a == x
a > x	!(a <= x)
a >= x	!(a < x)

2.4 Vec's C++ Representaion

Vec.h

```
/**
* Class Vec models a container object that uses an array to store
* the integer data elements.
* The purpose of this class is to demonstrate operator overloading in C++.
*
* @author  S. Ghaderpanah
* @version 1.0
* @since   June 4, 2018
*/
#ifndef Vec_H
#define Vec_H

#include <iostream>
#include<initializer_list>

class Vec {
private:
    int *store;// pointer to storage for elements of this Vec
    int sz; // size of this Vec
public:
    int size() const;    // Returns the size of the Vec
    Vec reverse() const;    // reverses the Vec elements
```

Default + Normal Constructors — Vec.h (Continued)

```
public:
    // 2 constructors: default + conversion
    explicit Vec(int = 5);

    // normal constructor
    // initializes this Vec of n elements to val
    Vec(int n, const int &val);

    // conversion constructor
    // initializes this Vec of n elements to the supplied initializer_list<int>
    Vec(std::initializer_list<int> initial_list);

    // Normal constructor
    // initializes this Vec of n elements to the elements of a raw C-array
    Vec(const int A[], int n);
```

The Big Three — Vec.h (Continued)

```
38      /*
39      The Big Three, a rule of thumb in C++: If a class defines at least one
40      of copy constructor, assignment operator, and destructor, it should
41      probably explicitly define all three.
42      */
43
44      // copy constructor
45      Vec(const Vec &source);
46
47      // assignment operator
48      Vec & operator=(const Vec &rhs);
49
50      // destructor
51      virtual ~Vec();
```

The Big Five — Vec.h (Continued)

```
52
53      // Since C++11:
54      // The Big Five = The Big Three + move ctor + move assignment operator
55
56      // move ctor
57      Vec(Vec &&source) noexcept; // will not thow exceptions
58
59      // move assignment
60      Vec & operator=(Vec &&rhs) noexcept;  // will not thow exceptions
```

The subscript operator [ ] — Vec.h (Continued)

```
61      int& operator[](int i);  // subscript operator overload: read and write
62      const int& operator[](int i) const; // subscript operator overload: read-only
```

Question Can the **return** type in the above read-only version of the subscript operator be just **int&** without the **const**?

Answer Yes, the return type can be **int&**.

```
int& operator[](int i) const; // subscript operator overload: read-only
```

But that would defeat the *const-ness* of the invoking object:

```
const Vec cv; // that's OK, cv is a const Vec
cv[0] = 10;   // OK too! Equivalent to cv.operator[](0) = 10;
```

That seems like a contradiction to me! if object **cv** is meant to be a **const Vec**, why allow **cv[0] = 10**?

Keeping its promise, the read-only version of **operator[]** does not modify **cv** during the call **cv.operator[](0)**, but it returns a reference to a variable inside **cv**, namely, **store[0]**, which is set to 10.

Using **const int&** as the return type for the **const** version of the **operator[]** member function, we arrange for **cv** to remain truly constant both inside and outside of that function.

Compound Arithmetic Assignment Operator Overloads — Vec.h (Continued)

```
63      Vec& operator+=(const Vec& rhs); // Vec += Vec
64      Vec& operator-=(const Vec& rhs); // Vec -= Vec
65
66      Vec& operator+=(int val); // Vec += val,  add val to every element
67      Vec& operator-=(int val); // Vec -= val,  subtract val from every element
68      Vec& operator*=(int val); // Vec *= val,  multiply every element by val
69      Vec& operator/=(int val); // Vec /= val,  divide every element by val
70      Vec& operator%=(int val); // Vec %= val,  modulus every element by val
```

Implementation Tip: Implement these operator before the corresponding operators in the following simple arithmetic operator group:

Simple Arithmetic Operator Overloads — Vec.h (Continued)

```
71      friend Vec operator+(const Vec& lhs, const Vec& rhs); // Vec + Vec
72      friend Vec operator-(const Vec& lhs, const Vec& rhs); // Vec - Vec;
73
74      friend Vec operator+(const Vec& lhs, const int& val); // Vec + int
75      friend Vec operator-(const Vec& lhs, const int& val); // Vec - int
76      friend Vec operator*(const Vec& lhs, const int& val); // Vec * int
77      friend Vec operator/(const Vec& lhs, const int& val); // Vec / int
78      friend Vec operator%(const Vec& lhs, const int& val); // Vec % int
```

Note: The above simple arithmetic operators may also be implemented as member functions. Conventionally, however, they are implemented as non-member functions because they do not modify their operands.

Simple Arithmetic Operator Overloads — Vec.h (Continued)

```
79      friend Vec operator+(const int& val, const Vec& rhs); // int + Vec
80      friend Vec operator-(const int& val, const Vec& rhs); // int - Vec
81      friend Vec operator*(const int& val, const Vec& rhs); // int * Vec
82      friend Vec operator/(const int& val, const Vec& rhs); // int / Vec
83      friend Vec operator%(const int& val, const Vec& rhs); // int % Vec
```

Note: The above simple arithmetic scalar operators cannot be implemented as member functions, because their left-hand-side operand is not a **Vec**.

Unary Operator Overloads — Vec.h (Continued)

```
84
85      Vec operator+() const; // unary +
86      Vec operator-() const; // unary -
87
88      Vec& operator++(); // unary prefix increment
89      Vec& operator--(); // unary prefix decrement
90
91      Vec operator++(int); // unary postfix increment
92      Vec operator--(int); // unary postfix decrement
```

Relational Operator Overloads — Vec.h (Continued)

```
94      // we plan to implement these two directly:
95      friend bool operator==(const Vec& lhs, const Vec& rhs); // Vec == Vec
96      friend bool operator<(const Vec& lhs, const Vec& rhs); // Vec < Vec
97
98     // we plan to implement these four using the above two:
99      friend bool operator>(const Vec& lhs, const Vec& rhs); // Vec > Vec
100     friend bool operator!=(const Vec& lhs, const Vec& rhs); // Vec != Vec
101     friend bool operator<=(const Vec& lhs, const Vec& rhs); // Vec <= Vec
102     friend bool operator>=(const Vec& lhs, const Vec& rhs); // Vec >= Vec
```

Note 1: The above relational operators may also be implemented as member functions. Conventionally, however, they are implemented as non-member functions because they do not modify their operands.

Note 2: For the purpose of demonstration, we will use the same 'relational' rules as those used when c-strings are compared.

Note 3: The bottom 4 relational operator overloads are implicitly defined in terms the the top two if we use **using namespace std::rel_ops;** from <**utility**>; here, however, we plan to directly implement them for your reference.

Extraction (input) and Insersion (output) operator overloads — Vec.h (Continued)

```
103
104     friend istream& operator>>(istream &sin, Vec &target);
105     friend ostream& operator<<(ostream &sout, const Vec &source);
106 };
107 #endif
```

3 Implementaion

Vec.cpp

```
#include<iostream>
#include<ostream>
#include <stdexcept>
#include <utility>      // std::rel_ops
using std::out_of_range;
using std::ostream;
using std::istream;
using std::cout;
using std::endl;
#include "Vec.h"
```

Default + Normal Constructors (1/4) Vec.cpp (Continued)

```
// default + conversion constructors
// see prototype for the default value for n
Vec::Vec(int n) {
    if (n <= 0) throw out_of_range("Vec size cannot be zero or negative");
    sz = n;
    store = new int[n];
}
```

Default + Normal Constructors (2/4) Vec.cpp (Continued)

```

// normal constructor
// initializes this Vec of n elements to val
Vec::Vec(int n, const int &val) {
    if (n <= 0) throw out_of_range("Vec size cannot be zero or negative");
    sz = n;
    store = new int[n];
    for (int i = 0; i < n; ++i) {
        store[i] = val;
    }
}
```

Default + Normal Constructors (3/4) — Vec.cpp (Continued)

```
29
30 // conversion constructor
31 // initializes this Vec of n elements to the supplied initializer_list<int>
32 Vec::Vec(std::initializer_list<int> initial_list)
33 {
34     sz = initial_list.size();
35     store = new int[sz];
36     int count = 0;
37     for (int x : initial_list) {
38         store[count] = x;
39         count++;
40     }
41 }
```

Default + Normal Constructors (4/4) — Vec.cpp (Continued)

```
42
43 // Normal constructor
44 // initializes this Vec of n elements to the elements of a raw C-array
45 Vec::Vec(const int A[], int n) {
46     if (n <= 0) throw out_of_range("Vec size cannot be zero or negative");
47     sz = n;
48     store = new int[n];
49     for (int i = 0; i < n; ++i) {
50         store[i] = A[i];
51     }
52 }
```

The Big Three, a rule of thumb in C++: If a class defines at least one of copy constructor, assignment operator, and destructor, it should probably explicitly define all three.

The Big Three (1/3) — Vec.cpp (Continued)

```
109 // Copy constructor
110 Vec::Vec(const Vec &source) {
111     sz = source.size();
112     store = new int[source.size()];
113     for (int i = 0; i < source.size(); ++i)
114     {
115         // since we have overloaded operator[] we might as well use it.
116         (*this)[i] = source[i]; // or this->store[i] = source.store[i];
117     }
118 }
```

Note that the parentheses in **(*this)[i]** are mandatory because precedence of operator **[]** is higher than precedence of **operator *****, so ***this[i]** is interpreted as ***(this[i])**, where **this[i]** makes no sense. Note, however, that the operators **()**, **[]**, -> and **.** have equal precedence and operate

from left to right.

The Big Three (2/3) — Vec.cpp (Continued)

```
119
120 // assignment operator overload
121 Vec& Vec::operator=(const Vec &rhs) {
122     if (this != &rhs) {
123         // no storage allocation if lhs and rhs Vecs each have the same size
124         if (size() != rhs.size()) {
125             delete[] store;
126             sz = rhs.size();
127             store = new int[rhs.size()];
128         }
129         for (int i = 0; i < rhs.size(); ++i)
130             store[i] = rhs[i];
131     }
132     return *this;
133 }
```

The Big Three (3/3) — Vec.cpp (Continued)

```
134
135 // destructor
136 Vec::~Vec() {
137     delete[] store;
138 }
```

The Big Five (1/2) — Vec.cpp (Continued)

```
139 // Since C++11:
140 // The Big Five = The Big Three + move ctor + move assignment operator
141 // The move constructor + move assignment make C++ faster and more efficient.
142
143 // move constructor
144 // note that source is not const (why?)
145 // steals all resources in source
146 Vec::Vec(Vec &&source) noexcept : store(source.store), sz(source.sz)
147 {
148     // leave source in a state in which it is safe to run the destructor
149     source.store = nullptr;
150     source.sz = 0;
151 }
```

The Big Five (2/2) — Vec.cpp (Continued)

```
152
153 // move assignment
154 Vec & Vec::operator=(Vec &&rhs) noexcept // note that rhs is not const (why?)
155 {
156     if (this != &rhs) // direct check for self-assignment
157     {
158         delete[] store; // free existing storage
159         this->store = rhs.store;  // steal storage from rhs
160         sz = rhs.sz;
161
162         // leave rhs in a state in which it is safe to run the destructor
163         rhs.store = nullptr;
164         rhs.sz = 0;
165     }
166     return *this;
167 }
```

The subscript operator [ ] — Vec.cpp (Continued)

```
168
169 // subscript operator overload: read and write.
170 int& Vec::operator[](int i) {
171    if (i < 0 || i >= size())
172    {
173       throw out_of_range("Vec index is out of range");
174    }
175    return store[i];
176 }
177
178 // subscript operator overload: read only.
179 const int& Vec::operator[](int i) const {
180    if (i < 0 || i >= size())
181    {
182       throw out_of_range("Vec index is out of range");
183    }
184    return store[i];
185 }
```

Compound Arithmetic Assignment Operator Overloads (1/7) Vec.cpp (Continued)

```
186
187 Vec & Vec::operator+=(const Vec & rhs)
188 {
189     if (this->size() != rhs.size())
190     {
191         throw std::invalid_argument("cannot add two Vecs of different sizes");
192         // where the class std::invalid_argument comes from <stdexcept>
193         // invalid_argument --> logic_error --> exception
194     }
195     for (int i = 0; i < rhs.size(); ++i) { store[i] += rhs[i]; }
196     return *this;
197 }
```

Compound Arithmetic Assignment Operator Overloads (2/7) Vec.cpp (Continued)

```
198
199 Vec & Vec::operator-=(const Vec & rhs)
200 {
201     if (this->size() != rhs.size())
202     {
203         throw std::invalid_argument("cannot add/subtract two Vecs of different sizes");
204     }
205     for (int i = 0; i < rhs.size(); ++i) { store[i] -= rhs[i]; }
206     return *this;
207 }
```

Compound Arithmetic Assignment Operator Overloads (3/7) Vec.cpp (Continued)

```
208
209 // Adds val to every element in this Vec
210 Vec & Vec::operator+=(const int & val)
211 {
212     for (int i = 0; i < this->size(); ++i) { store[i] += val; }
213     return *this;
214 }
```

Compound Arithmetic Assignment Operator Overloads (4/7) Vec.cpp (Continued)

```
215
216 // Substracts val from every element in this Vec
217 Vec & Vec::operator-=(const int & val)
218 {
219     for (int i = 0; i < this->size(); ++i) { store[i] -= val; }
220     return *this;
221 }
```

Compound Arithmetic Assignment Operator Overloads (5/7) — Vec.cpp (Continued)

```
222
223 Vec & Vec::operator*=(const int & val)
224 {
225     for (int i = 0; i < this->size(); ++i) { store[i] *= val; }
226     return *this;
227 }
```

Compound Arithmetic Assignment Operator Overloads (6/7) — Vec.cpp (Continued)

```
228
229 Vec & Vec::operator/=(const int & val)
230 {
231     for (int i = 0; i < this->size(); ++i) { store[i] /= val; }
232     return *this;
233 }
```

Compound Arithmetic Assignment Operator Overloads (7/7) — Vec.cpp (Continued)

```
234
235 Vec & Vec::operator%=(const int & val)
236 {
237     for (int i = 0; i < this->size(); ++i) { store[i] %= val; }
238     return *this;
239 }
```

Simple Arithmetic Operator Overloads (1/7) — Vec.cpp (Continued)

```
240
241 // returns lhs + rhs
242 Vec operator+(const Vec & lhs, const Vec & rhs)
243 {
244     Vec temp(lhs);
245     temp += rhs;  // using operator+=(rhs) defined above
246     return temp;
247 }
```

Simple Arithmetic Operator Overloads (2/7) — Vec.cpp (Continued)

```
248
249 // returns lhs - rhs
250 Vec operator-(const Vec & lhs, const Vec & rhs)
251 {
252     Vec temp(lhs);
253     temp -= rhs; // using operator-=(rhs) defined above
254     return temp;
255 }
```

Simple Arithmetic Operator Overloads (3/7) — Vec.cpp (Continued)

```
256
257 // returns Vec + int
258 Vec operator+(const Vec & lhs, const int & val)
259 {
260     Vec temp(lhs);
261     temp += val; // using operator+=(val) defined above
262     return temp;
263 }
```

Simple Arithmetic Operator Overloads (4/7) — Vec.cpp (Continued)

```
264
265 Vec operator-(const Vec & lhs, const int & val)
266 {
267     Vec temp(lhs);
268     temp -= val;  // using operator-=(val) defined above
269     return temp;
270 }
```

Simple Arithmetic Operator Overloads (5/7) — Vec.cpp (Continued)

```
271
272 Vec operator*(const Vec & lhs, const int & val)
273 {
274     Vec temp(lhs);  // using operator*=(val) defined above
275     temp *= val;
276     return temp;
277 }
```

Simple Arithmetic Operator Overloads (6/7) — Vec.cpp (Continued)

```
278
279 Vec operator/(const Vec & lhs, const int & val)
280 {
281     Vec temp(lhs);
282     temp /= val;  // using operator/=(val) defined above
283     return temp;
284 }
```

Simple Arithmetic Operator Overloads (7/7) — Vec.cpp (Continued)

```
285
286 Vec operator%(const Vec & lhs, const int & val)
287 {
288     Vec temp(lhs);
289     temp %= val;  // using operator%=(val) defined above
290     return temp;
291 }
```

Simple Arithmetic Operator Overloads (1/5) — Vec.cpp (Continued)

```
292
293 Vec operator+(const int & val, const Vec & rhs)
294 {
295     return rhs + val;
296 }
```

Simple Arithmetic Operator Overloads (2/5) — Vec.cpp (Continued)

```
297
298 // return int + Vec
299 Vec operator-(const int & val, const Vec & rhs)
300 {
301     return rhs - val;
302 }
```

Simple Arithmetic Operator Overloads (3/5) — Vec.cpp (Continued)

```
303
304 Vec operator*(const int & val, const Vec & rhs)
305 {
306     return rhs * val;
307 }
```

Simple Arithmetic Operator Overloads (4/5) — Vec.cpp (Continued)

```
308
309 Vec operator/(const int & val, const Vec & rhs)
310 {
311     return rhs / val;
312 }
```

Simple Arithmetic Operator Overloads (5/5) — Vec.cpp (Continued)

```
313
314 Vec operator%(const int & val, const Vec & rhs)
315 {
316     return rhs % val;
317 }
```

Unary Operator Overloads (1/6) — Vec.cpp (Continued)

```
318 Vec Vec::operator+() const
319 {
320    return *this;
321 }
```

Unary Operator Overloads (2/6) — Vec.cpp (Continued)

```
322
323 Vec Vec::operator-() const
324 {
325    for (int i = 0; i < this->size(); ++i) { store[i] = -store[i]; }
326    return *this;
327 }
```

Unary Operator Overloads (3/6) — Vec.cpp (Continued)

```
328
329 Vec& Vec::operator++()
330 {
331    *this += 1;
332    return *this;
333 }
```

Unary Operator Overloads (4/6) — Vec.cpp (Continued)

```
334 Vec& Vec::operator--()
335 {
336    return (*this -= 1);
337 }
```

Unary Operator Overloads (5/6) — Vec.cpp (Continued)

```
338
339 Vec Vec::operator++(int)
340 {
341     Vec temp(*this);
342     *this += 1;
343     return temp;
344 }
```

Unary Operator Overloads (6/6) — Vec.cpp (Continued)

```
345
346 Vec Vec::operator--(int)
347 {
348     Vec temp(*this);
349     *this -= 1;
350     return temp;
351 }
```

Relational Operator Overloads (1/6) — Vec.cpp (Continued)

```
352 // here we consider two Vecs equal  if they have the same
353 // lengths and the equal corresponding elements
354 bool operator==(const Vec & lhs, const Vec & rhs)
355 {
356     if (lhs.size() != rhs.size())
357     {
358         return false;
359     }
360
361     for (int i = 0; i < lhs.size(); ++i)
362     {
363         if (lhs[i] != rhs[i]) // using operator[]
364         {
365             return false;
366         }
367     }
368     return true;
369 }
```

Relational Operator Overloads (2/6) — Vec.cpp (Continued)

```
370
371 // For the purpose of demonstration, we will use the same 'relational'
372 // rules as those used when c-strings are compared.
373 bool operator<(const Vec & lhs, const Vec & rhs)
374 {
375     // compute length of the smaller Vec
376     int min_length = lhs.size() < rhs.size() ? lhs.size() : rhs.size();
377
378     // compare existing elements for 'less than' and 'greater than' notions
379     for (int i = 0; i < min_length; ++i)
380     {
381         if (lhs[i] < rhs[i]) // using operator[]
382         {
383             return true;
384         }
385         else if (lhs[i] > rhs[i])
386         {
387             return false;
388         }
389     }
390     // all elements are equal, so their lengths will decide the outcome
391     return lhs.size() < rhs.size();
392 }
```

Relational Operator Overloads (3/6) — Vec.cpp (Continued)

```
393 // the remaining operators !=, <=, >, >= are now defined in terms of the two == and < abo
394
395 bool operator>(const Vec & lhs, const Vec & rhs)
396 {
397     return  rhs < lhs;
398 }
```

Relational Operator Overloads (4/6) — Vec.cpp (Continued)

```
399
400 bool operator!=(const Vec & lhs, const Vec & rhs)
401 {
402     return !(lhs == rhs);
403 }
```

Relational Operator Overloads (5/6) — Vec.cpp (Continued)

```
404
405 bool operator<=(const Vec & lhs, const Vec & rhs)
406 {
407     return !(rhs < lhs);
408 }
```

Relational Operator Overloads (6/6) — Vec.cpp (Continued)

```
409
410 bool operator>=(const Vec & lhs, const Vec & rhs)
411 {
412     return !(lhs < rhs);
413 }
```

Insersion (output) operator overloads — Vec.cpp (Continued)

```
414
415 // insertion operator overload for Vec objects
416 ostream& operator<<(ostream &sout, const Vec &source) {
417    sout << '(' << source[0];
418    for (int i = 1; i < source.size(); ++i)
419       sout << ", " << source[i];
420    sout << ')';
421    return sout;
422 }
```

Extraction (input) operator overloads — Vec.cpp (Continued)

```
423
424 // extraction operator for Vec objects
425 // note that the supplied target object must be non-const (why?)
426 istream& operator>>(istream &sin, Vec &target) {
427    cout << "Enter values for all " << target.size()
428         << " vector elements (integers)" << endl;
429    for (int i = 0; i < target.size(); ++i)
430    {
431       std::cout << '?';
432       sin >> target[i];
433    }
434    return sin;
435 }
```

size() Vec.cpp (Continued)

```
436
437 // Returns the size of this Vec
438 int Vec::size() const { return sz; }
```

reverse() Vec.cpp (Continued)

```
439
440 // reverse the elements of this Vec
441 Vec Vec::reverse() const
442 {
443     Vec temp(*this); // make a temp copy of *this
444                      // now reverse temp
445     int left = 0;
446     int right = temp.size() - 1;
447     while (left < right)
448     {
449         int t = temp.store[left];
450         temp.store[left] = temp.store[right];
451         temp.store[right] = t;
452         ++left;
453         --right;
454     }
455     return temp;
456 }
```

4 A Vec Test Driver

A Vec Test Driver — Vec_Driver.cpp

```cpp
#include<iostream>
#include<iomanip>
using namespace std;

#include "Vec.h"

int main()
{
    try
    {
        Vec a;
        cin >> a; // will prompt for and reads 10 inegers
        cout << a << endl;

        Vec b{ 1,2,3,4,5,6,7,8,9 };
        cout << left << setw(20) << "b: " << b << endl;

        cout << left << setw(20) << " b.reverse(): " << b.reverse() << endl;
        Vec c = b.reverse();
        cout << left << setw(20) << "c = b reversed: " << c << endl;

        // add 1 to every element of c
        c += 1;
        cout << left << setw(20) << "c += 1: " << c << endl;

        // add 1 to every element of c
        c = c + 1;
        cout << left << setw(20) << "c = c + 1: " << c << endl;

        // copy b+c to d
        Vec d = b + c;
        cout << left << setw(20) << "d = b + c: " << d << endl;

        // add -2 to every element of d
        d -= 2;
        cout << left << setw(20) << "d -= 2: " << d << endl;

        // copy b-c to e
        Vec e = b - c;
        cout << left << setw(20) << "e = b - c: " << e << endl;

        // multiply every element of e by -2
        e = e * (-2);
        cout << left << setw(20) << "e = e * (-2): " << e << endl;

        // multiply every element of e by -2
        e = 2 * e;
        cout << left << setw(20) << "e = 2 * e: " << e << endl;
```

A Vec Test Driver — Vec_Driver.cpp (Continued)

```
458
459         // devide every element of e by 4
460         e = e / 4;
461         cout << left << setw(20) << "e = e / 4: " << e << endl;
462
463         // demo initializer-list
464         Vec f = { 1, 2, 3 }, g = { 2, 3 };
465         cout << "f: " << f << endl;
466         cout << "g: " << g << endl;
467
468 cout << f << " is" << ((f < g) ? "" : " not") << " less than " << g << endl;
469 cout << f << " is" << ((f <= g) ? "" : " not") << " less than or equal to " << g << endl;
470 cout << f << " is" << ((f > g) ? "" : " not") << " greater than " << g << endl;
471 cout << f << " is" << ((f >= g) ? "" : " not") << " greater than or equal to" << g << endl
472 cout << f << " and " << g << " are " << ((f == g) ? "" : " not") << " equal" << endl;
473 cout << f << " and " << g << " are " << ((f != g) ? "" : " not") << " unequal" << endl;
474
475         Vec h{ 1,2,3 }, hh{ 2,3,4 }, hhh{ 3,4,5 };
476
477         Vec r = hh;
478         r = r++;
479         cout << "r: " << r << endl;
480         if (r != hh)
481         {
482             throw logic_error("error: operator++(int)");
483         }
484         cout << "r: " << r << endl;
485
486         Vec s = hh;
487         s = s--;
488         cout << "s: " << s << endl;
489         if (s != hh)
490         {
491             throw logic_error("error: operator--(int)");
492         }
493         cout << "s: " << s << endl;
494
495         Vec rr = hh;
496         ++rr;
497         cout << "rr: " << rr << endl;
498         if (rr != hhh)
499         {
500             throw logic_error("error: operator++");
501         }
502         cout << "rr: " << rr << endl;
503
504         Vec ss = hh;
505         --ss;
506         cout << "ss: " << ss << endl;
507         if (ss != h)
508         {
509             throw logic_error("error: operator--");
510         }
511         cout << "ss: " << ss << endl;
512     }
```

A Vec Test Driver — Vec_Driver.cpp (Continued)

```
513
514     catch (const std::out_of_range& oor) {
515         std::cerr << "Out of Range error: " << oor.what() << '\n';
516     }
517     catch (std::bad_alloc& ba)
518     {
519         std::cerr << "bad_alloc caught: " << ba.what() << '\n';
520     }
521     catch (std::logic_error& le)
522     {
523         std::cerr << "logic error : " << le.what() << '\n';
524     }
525     catch (...)
526     {
527         std::cerr << "catch-all error  " << '\n';
528     }
529     return 0;
530 }
```