assign
Syntax:

```
#include <vector>
void assign( size_type num, const TYPE& val );
void assign( input_iterator start, input_iterator end );
```

The assign() function either gives the current vector the values from *start* to *end*, or gives it *num* copies of *val*.

This function will destroy the previous contents of the vector.

For example, the following code uses assign() to put 10 copies of the integer 42 into a vector:

```
vector<int> v;
v.assign( 10, 42 );
for( int i = 0; i < v.size(); i++ ) {
  cout << v[i] << " ";
}
cout << endl;
```

The above code displays the following output:

```
42 42 42 42 42 42 42 42 42 42
```

The next example shows how assign() can be used to copy one vector to another:

```
vector<int> v1;
for( int i = 0; i < 10; i++ ) {
  v1.push_back( i );
}

vector<int> v2;
v2.assign( v1.begin(), v1.end() );

for( int i = 0; i < v2.size(); i++ ) {
  cout << v2[i] << " ";
}
cout << endl;
```

When run, the above code displays the following output:

```
0 1 2 3 4 5 6 7 8 9
```

Related topics:
(C++ Strings) assign
insert
push_back
(C++ Lists) push_front

---

at
Syntax:

```
#include <vector>
TYPE& at( size_type loc );
const TYPE& at( size_type loc ) const;
```

The at() function returns a reference to the element in the vector at index *loc*. The at() function is safer

than the [] operator, because it won't let you reference items outside the bounds of the vector.

For example, consider the following code:

```
vector<int> v( 5, 1 );
for( int i = 0; i < 10; i++ ) {
  cout << "Element " << i << " is " << v[i] << endl;
}
```

This code overrunns the end of the vector, producing potentially dangerous results. The following code would be much safer:

```
vector<int> v( 5, 1 );
for( int i = 0; i < 10; i++ ) {
  cout << "Element " << i << " is " << v.at(i) << endl;
}
```

Instead of attempting to read garbage values from memory, the at() function will realize that it is about to overrun the vector and will throw an exception.

Related topics:
Vector operators

---

back
Syntax:

```
#include <vector>
TYPE& back();
const TYPE& back() const;
```

The back() function returns a reference to the last element in the vector.

For example:

```
vector<int> v;
for( int i = 0; i < 5; i++ ) {
  v.push_back(i);
}
cout << "The first element is " << v.front()
     << " and the last element is " << v.back() << endl;
```

This code produces the following output:

```
The first element is 0 and the last element is 4
```

The back() function runs in constant time.

Related topics:
front
pop_back

---

begin
Syntax:

```
#include <vector>
iterator begin();
const_iterator begin() const;
```

The function begin() returns an iterator to the first element of the vector. begin() should run in constant time.

For example, the following code uses begin() to initialize an iterator that is used to traverse a list:

```
   // Create a list of characters
   list<char> charList;
   for( int i=0; i < 10; i++ ) {
     charList.push_front( i + 65 );
   }
   // Display the list
   list<char>::iterator theIterator;
   for( theIterator = charList.begin(); theIterator != charList.end(); theIterator++
) {
     cout << *theIterator;
   }
```

Related topics:
end
rbegin
rend

---

capacity
Syntax:

```
  #include <vector>
  size_type capacity() const;
```

The capacity() function returns the number of elements that the vector can hold before it will need to allocate more space.

For example, the following code uses two different methods to set the capacity of two vectors. One method passes an argument to the constructor that suggests an initial size, the other method calls the reserve function to achieve a similar goal:

```
vector<int> v1(10);
cout << "The capacity of v1 is " << v1.capacity() << endl;
vector<int> v2;
v2.reserve(20);
cout << "The capacity of v2 is " << v2.capacity() << endl;
```

When run, the above code produces the following output:

```
The capacity of v1 is 10
The capacity of v2 is 20
```

C++ containers are designed to grow in size dynamically. This frees the programmer from having to worry about storing an arbitrary number of elements in a container. However, sometimes the programmer can improve the performance of her program by giving hints to the compiler about the size of the containers that the program will use. These hints come in the form of the reserve() function and the constructor used in the above example, which tell the compiler how large the container is expected to get.

The capacity() function runs in constant time.

Related topics:
reserve
resize
size

---

clear
Syntax:

```
#include <vector>
void clear();
```

The function clear() deletes all of the elements in the vector. clear() runs in linear time.

Related topics:
erase

---

empty
Syntax:

```
#include <vector>
bool empty() const;
```

The empty() function returns true if the vector has no elements, false otherwise.

For example, the following code uses empty() as the stopping condition on a (C/C++ Keywords) while loop to clear a vector and display its contents in reverse order:

```
vector<int> v;
for( int i = 0; i < 5; i++ ) {
  v.push_back(i);
}
while( !v.empty() ) {
  cout << v.back() << endl;
  v.pop_back();
}
```

Related topics:
size

---

end
Syntax:

```
#include <vector>
iterator end();
const_iterator end() const;
```

The end() function returns an iterator just past the end of the vector.

Note that before you can access the last element of the vector using an iterator that you get from a call to end(), you'll have to decrement the iterator first. This is because end() doesn't point to the end of the vector; it points **just past the end of the vector**.

For example, in the following code, the first "cout" statement will display garbage, whereas the second statement will actually display the last element of the vector:

```
vector<int> v1;
v1.push_back( 0 );
v1.push_back( 1 );
v1.push_back( 2 );
v1.push_back( 3 );

int bad_val = *(v1.end());
cout << "bad_val is " << bad_val << endl;
```

```
  int good_val = *(v1.end() - 1);
  cout << "good_val is " << good_val << endl;
```

The next example shows how begin() and end() can be used to iterate through all of the members of a vector:

```
vector<int> v1( 5, 789
 ); vector<int>::iterator it; for( it = v1.begin(); it !=
 v1.end(); it++ ) { cout << *it << endl; }
```

The iterator is initialized with a call to begin(). After the body of the loop has been executed, the iterator is incremented and tested to see if it is equal to the result of calling end(). Since end() returns an iterator pointing to an element just after the last element of the vector, the loop will only stop once all of the elements of the vector have been displayed.

end() runs in constant time.

Related topics:
begin
rbegin
rend

---

erase
Syntax:

```
  #include <vector>
  iterator erase( iterator loc );
  iterator erase( iterator start, iterator end );
```

The erase() function either deletes the element at location *loc*, or deletes the elements between *start* and *end* (including *start* but not including *end*). The return value is the element after the last element erased.

The first version of erase (the version that deletes a single element at location *loc*) runs in constant time for lists and linear time for vectors, dequeues, and strings. The multiple-element version of erase always takes linear time.

For example:

```
// Create a vector, load it with the first ten characters of the alphabet
vector<char> alphaVector;
for( int i=0; i < 10; i++ ) {
  alphaVector.push_back( i + 65 );
}
int size = alphaVector.size();
vector<char>::iterator startIterator;
vector<char>::iterator tempIterator;
for( int i=0; i < size; i++ ) {
  startIterator = alphaVector.begin();
  alphaVector.erase( startIterator );
  // Display the vector
  for( tempIterator = alphaVector.begin(); tempIterator != alphaVector.end();
tempIterator++ ) {
    cout << *tempIterator;
  }
  cout << endl;
}
```

That code would display the following output:

```
BCDEFGHIJ
CDEFGHIJ
DEFGHIJ
EFGHIJ
FGHIJ
GHIJ
HIJ
IJ
J
```

In the next example, erase() is called with two iterators to delete a range of elements from a vector:

```
// create a vector, load it with the first ten characters of the alphabet
vector<char> alphaVector;
for( int i=0; i < 10; i++ ) {
  alphaVector.push_back( i + 65 );
}
// display the complete vector
for( int i = 0; i < alphaVector.size(); i++ ) {
  cout << alphaVector[i];
}
cout << endl;

// use erase to remove all but the first two and last three elements
// of the vector
alphaVector.erase( alphaVector.begin()+2, alphaVector.end()-3 );
// display the modified vector
for( int i = 0; i < alphaVector.size(); i++ ) {
  cout << alphaVector[i];
}
cout << endl;
```

When run, the above code displays:

```
ABCDEFGHIJ
ABHIJ
```

Related topics:
clear
insert
pop_back
(C++ Lists) pop_front
(C++ Lists) remove
(C++ Lists) remove_if

---

front
Syntax:

```
#include <vector>
TYPE& front();
const TYPE& front() const;
```

The front() function returns a reference to the first element of the vector, and runs in constant time.

Related topics:
back
(C++ Lists) pop_front
(C++ Lists) push_front

---

insert

Syntax:

```
  #include <vector>
  iterator insert( iterator loc, const TYPE& val );
  void insert( iterator loc, size_type num, const TYPE& val );
  template<TYPE> void insert( iterator loc, input_iterator start, input_iterator end
);
```

The insert() function either:

- inserts *val* before *loc*, returning an iterator to the element inserted,
- inserts *num* copies of *val* before *loc*, or
- inserts the elements from *start* to *end* before *loc*.

Note that inserting elements into a vector can be relatively time-intensive, since the underlying data structure for a vector is an array. In order to insert data into an array, you might need to displace a lot of the elements of that array, and this can take linear time. If you are planning on doing a lot of insertions into your vector and you care about speed, you might be better off using a container that has a linked list as its underlying data structure (such as a List or a Deque).

For example, the following code uses the insert() function to splice four copies of the character 'C' into a vector of characters:

```
// Create a vector, load it with the first 10 characters of the alphabet
vector<char> alphaVector;
for( int i=0; i < 10; i++ ) {
  alphaVector.push_back( i + 65 );
}

// Insert four C's into the vector
vector<char>::iterator theIterator = alphaVector.begin();
alphaVector.insert( theIterator, 4, 'C' );

// Display the vector
for( theIterator = alphaVector.begin(); theIterator != alphaVector.end();
theIterator++ )    {
  cout << *theIterator;
}
```

This code would display:

```
CCCCABCDEFGHIJ
```

Here is another example of the insert() function. In this code, insert() is used to append the contents of one vector onto the end of another:

```
 vector<int> v1;
 v1.push_back( 0 );
 v1.push_back( 1 );
 v1.push_back( 2 );
 v1.push_back( 3 );

 vector<int> v2;
 v2.push_back( 5 );
 v2.push_back( 6 );
 v2.push_back( 7 );
 v2.push_back( 8 );

 cout << "Before, v2 is: ";
 for( int i = 0; i < v2.size(); i++ ) {
   cout << v2[i] << " ";
 }
```

```
cout << endl;

v2.insert( v2.end(), v1.begin(), v1.end() );

cout << "After, v2 is: ";
for( int i = 0; i < v2.size(); i++ ) {
  cout << v2[i] << " ";
}
cout << endl;
```

When run, this code displays:

```
Before, v2 is: 5 6 7 8
After, v2 is: 5 6 7 8 0 1 2 3
```

Related topics:
assign
erase
push_back
(C++ Lists) merge
(C++ Lists) push_front
(C++ Lists) splice

---

max_size
Syntax:

```
#include <vector>
size_type max_size() const;
```

The max_size() function returns the maximum number of elements that the vector can hold. The max_size() function should not be confused with the size() or capacity() functions, which return the number of elements currently in the vector and the the number of elements that the vector will be able to hold before more memory will have to be allocated, respectively.

Related topics:
size

---

pop_back
Syntax:

```
#include <vector>
void pop_back();
```

The pop_back() function removes the last element of the vector.

pop_back() runs in constant time.

Related topics:
back
erase
(C++ Lists) pop_front
push_back

---

push_back
Syntax:

```
#include <vector>
void push_back( const TYPE& val );
```

The push_back() function appends *val* to the end of the vector.

For example, the following code puts 10 integers into a list:

```
list<int> the_list;
for( int i = 0; i < 10; i++ )
  the_list.push_back( i );
```

When displayed, the resulting list would look like this:

```
0 1 2 3 4 5 6 7 8 9
```

push_back() runs in constant time.

Related topics:
assign
insert
pop_back
(C++ Lists) push_front

---

rbegin
Syntax:

```
#include <vector>
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
```

The rbegin() function returns a reverse_iterator to the end of the current vector.

rbegin() runs in constant time.

Related topics:
begin
end
rend

---

rend
Syntax:

```
#include <vector>
reverse_iterator rend();
const_reverse_iterator rend() const;
```

The function rend() returns a reverse_iterator to the beginning of the current vector.

rend() runs in constant time.

Related topics:
begin
end
rbegin

---

reserve
Syntax:

```
#include <vector>
void reserve( size_type size );
```

The reserve() function sets the capacity of the vector to at least *size*.

reserve() runs in linear time.

Related topics:
capacity

---

resize
Syntax:

```
#include <vector>
void resize( size_type num, const TYPE& val = TYPE() );
```

The function resize() changes the size of the vector to *size*. If *val* is specified then any newly-created elements will be initialized to have a value of *val*.

This function runs in linear time.

Related topics:
Vector constructors & destructors
capacity
size

---

size
Syntax:

```
#include <vector>
size_type size() const;
```

The size() function returns the number of elements in the current vector.

Related topics:
capacity
empty
(C++ Strings) length
max_size
resize

---

swap
Syntax:

```
#include <vector>
void swap( const container& from );
```

The swap() function exchanges the elements of the current vector with those of *from*. This function operates in constant time.

For example, the following code uses the swap() function to exchange the values of two strings:

```
  string first( "This comes first" );
  string second( "And this is second" );
  first.swap( second );
  cout << first << endl;
  cout << second << endl;
```

The above code displays:

```
  And this is second
  This comes first
```

Related topics:
(C++ Lists) splice

Vector constructors
Syntax:

```
#include <vector>
vector();
vector( const vector& c );
vector( size_type num, const TYPE& val = TYPE() );
vector( input_iterator start, input_iterator end );
~vector();
```

The default vector constructor takes no arguments, creates a new instance of that vector.

The second constructor is a default copy constructor that can be used to create a new vector that is a copy of the given vector *c*.

The third constructor creates a vector with space for *num* objects. If *val* is specified, each of those objects will be given that value. For example, the following code creates a vector consisting of five copies of the integer 42:

```
vector<int> v1( 5, 42 );
```

The last constructor creates a vector that is initialized to contain the elements between *start* and *end*. For example:

```
// create a vector of random integers
cout << "original vector: ";
vector<int> v;
for( int i = 0; i < 10; i++ ) {
  int num = (int) rand() % 10;
  cout << num << " ";
  v.push_back( num );
}
cout << endl;

// find the first element of v that is even
vector<int>::iterator iter1 = v.begin();
while( iter1 != v.end() && *iter1 % 2 != 0 ) {
  iter1++;
}

// find the last element of v that is even
vector<int>::iterator iter2 = v.end();
do {
  iter2--;
} while( iter2 != v.begin() && *iter2 % 2 != 0 );

// only proceed if we find both numbers
if( iter1 != v.end() && iter2 != v.begin() ) {
```

```
  cout << "first even number: " << *iter1 << ", last even number: " << *iter2 <<
endl;

  cout << "new vector: ";
  vector<int> v2( iter1, iter2 );
  for( int i = 0; i < v2.size(); i++ ) {
    cout << v2[i] << " ";
  }
  cout << endl;
}
```

When run, this code displays the following output:

```
original vector: 1 9 7 9 2 7 2 1 9 8
first even number: 2, last even number: 8
new vector: 2 7 2 1 9
```

All of these constructors run in linear time except the first, which runs in constant time.

The default destructor is called when the vector should be destroyed.

Vector operators
Syntax:

```
#include <vector>
TYPE& operator[]( size_type index );
const TYPE& operator[]( size_type index ) const;
vector operator=(const vector& c2);
bool operator==(const vector& c1, const vector& c2);
bool operator!=(const vector& c1, const vector& c2);
bool operator<(const vector& c1, const vector& c2);
bool operator>(const vector& c1, const vector& c2);
bool operator<=(const vector& c1, const vector& c2);
bool operator>=(const vector& c1, const vector& c2);
```

All of the C++ containers can be compared and assigned with the standard comparison operators: ==, !=, <=, >=, <, >, and =. Individual elements of a vector can be examined with the [] operator.

Performing a comparison or assigning one vector to another takes linear time. The [] operator runs in constant time.

Two vectors are equal if:

1. Their size is the same, and
2. Each member in location i in one vector is equal to the the member in location i in the other vector.

Comparisons among vectors are done lexicographically.

For example, the following code uses the [] operator to access all of the elements of a vector:

```
vector<int> v( 5, 1 );
for( int i = 0; i < v.size(); i++ ) {
  cout << "Element " << i << " is " << v[i] << endl;
}
```

Related topics:
at
