1. Given a vector of integers, \( v \), write a short code segment that uses **iterators** to add every second entry in the vector. For example, when \( v \) is

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
1 & 16 & 4 & -3 & 2 & 76 & 9 & 3 & 6
\end{array}
\]

the segment should add \( 1 + 4 + 2 + 9 + 6 \) to get 22. Store the result in a variable called \( \text{sum} \). Repeat the problem using array indices instead of iterators.

**Iterator Solution:**

```
int sum=0;
vector<int>::iterator it = v.begin();
while ( it != v.end() )
{
    sum += *it;
    ++ it;
    if ( it != v.end() ) ++it;
}
```

**Indexing Solution:**

```
int sum=0;
unsigned int i=0;
while ( i < v.size() )
{
    sum += v[i];
    i += 2;
}
```

2. What is the output of the following code fragment?

```cpp
void foo( int x, int & y )
{
    int t=x;
    x = y;
    y = t;
}
```
cout << x << " " << y << "\n";
}

int main()
{
    int a = 6;
    int b = 3;
    foo( a, b );
    cout << a << " " << b << "\n";
}

Solution

3 6
6 6

3. Given a struct:

struct bar {
    int x;
    double y;
    list<string> z;
};

Write a function that rearranges a vector of bar’s so that they are ordered by decreasing value of x, and by increasing y for bar’s with equal values of x. Also, make sure that each bar has its list z in increasing order.

Solution:

bool compare_bar( const bar& one, const bar& two )
{
    return one.x > two.x
        || ( one.x == two.x && one.y < two.y );
}

void sort_bar( vector<bar>& v )
{
    sort( v.begin(), v.end(), compare_bar );
}
for ( unsigned int i=0; i<v.size(); ++i )
    v[i].z.sort();
}

4. Write a code segment that copies the contents of a string into a list of char in reverse order.

Solution:

    // assume the string is s and is initialized
    list<char> result;
    for ( int i=0; i<s.size(); ++i )
        result.push_front( s[i] );

5. Write a function that takes a list of doubles and copies its values into two new lists of doubles, one containing only the negative numbers from the original list, and the other containing the non-negative numbers from the original list. For example, if the original list contains the values

-1.3, 5.2, 8.7, 0.0, -4.5, 7.8, -9.1, 3.5, 6.6

then the resulting list containing the negative numbers should contain

-1.3, -4.5, -9.1

and the resulting list containing the positive numbers should contain

5.2, 8.7, 0.0, 7.8, 3.5, 6.6

Start this problem by writing the function prototype as you think it should appear (hint: there should be three parameters) and then write the code.

Solution

void copy_pos_neg( const list<double>& original,
                   list<double>& negatives,
                   list<double>& positives )
{
    negatives.clear(); positives.clear();
    for ( list<double>::const_iterator i = original.begin();

i != original.end(); ++ i )
{
    if ( *i < 0 )
        negatives.push_back( *i );
    else
        positives.push_back( *i );
}

6. Write a code segment that removes all occurrences of the letter 'c' from a string. Consider both uppercase 'C' and lowercase 'c'. For example, the string

    Chocolate

would become the string

    hoolate

Start by not using any generic functions and then repeat the problem using remove and erase.

**Without Generic Functions:** This version does what remove might do by copying over the 'C' and the 'c' and then erasing the extra locations at the end.

    // assume the string is s
    unsigned int i=0, j=0;
    while ( j != s.size() )
    {
        if ( s[j] != 'C' && s[j] != 'c')
        {
            s[i] = s[j]; // copy from location j to i
            ++i; ++j; // increment both
        }
        else
            ++j; // increment just j, thereby skipping the 'C' or 'c'
    }
    s.erase( s.begin()+i, s.end() ); // remove everything from i to the end

**With Generic Functions:**
// assume the string is s
s.erase( remove( s.begin(), s.end(), 'C' ), s.end() );
s.erase( remove( s.begin(), s.end(), 'c' ), s.end() );

// actually, the above solution can be reduced to one line:
s.erase( remove( s.begin(),
    remove( s.begin(), s.end(), 'c' ), 'C' ), s.end() );

7. Write a function that rearranges a list of doubles so that all the negative values come before all the non-negative values AND the order of the negative values is preserved AND the order of the positive values is preserved. For example, if the list contains

-1.3, 5.2, 8.7, 0.0, -4.5, 7.8, -9.1, 3.5, 6.6

Then the modified list should contain

-1.3, -4.5, -9.1, 5.2, 8.7, 0.0, 7.8, 3.5, 6.6

This is perhaps a little more challenging than some of the other problems, but still instructive.

**Solution:**
Here’s an “in-place” solution which does not use an extra list. Whenever a non-negative number is seen it is removed and placed on the back of the list. Negative numbers are skipped. The trick to ensuring this works is using a separate counter to ensure that all items in the list are tested exactly once. Otherwise, some will be examined multiple times.

```cpp
void rearrange( list<double>& dbl )
{
    unsigned int sz = dbl.size();
    unsigned int i;
    list<double>::iterator itr;
    for ( i=0; i<sz; ++i )
    {
      if ( *itr < 0 )
        ++ itr;
```
else
{
    dbl.push_back( *itr );
    itr = dbl.erase( itr );
}
}

Here's a second solution which uses an extra list and two passes through the list. The first pass puts the negative numbers on the front and the second pass puts the positive numbers on the back.

void
rearrange( list<double>& dbl )
{
    list<double> temp;
    list<double>::iterator i;
    for ( i = dbl.begin(); i != dbl.end(); ++i )
    if ( *i < 0 )
        temp.push_back( *i );
    for ( i = dbl.begin(); i != dbl.end(); ++i )
    if ( *i >= 0 )
        temp.push_back( *i );
    dbl = temp;
}