Top Ten Things I Love About STL

Miro Jurišić <meeroh@meeroh.org>

June 19, 2003

allocators

- Despite their limitations, STL allocators can be immensely useful
- Take over memory management while retaining all the benefits of STL containers and iterators
- Don't attempt writing allocators before reading Effective STL

stream exceptions

- By default, streams enter an error state when something goes wrong
- You may prefer to have an exception throw
- Use `basic_ios::exceptions`

```
ofstream        output ("myfile");
output.exceptions (
    ofstream::badbit |
    ofstream::eofbit |
    ofstream::failbit
);
```

```
try {
    output << stuff;
} catch (ios_base::failure& e) {
    // Handle error;
}
```

numeric_limits

- You want `INT_MAX` equivalent for user-defined types
- You have it: `numeric_limits`
- Works for all numeric types (floating point and integer)

```
typedef signed long long            MyGoofyType;

cout << "MyGoofyType goes from" <<
    numeric_limits <MyGoofyType>::min () << " to " <<
    numeric_limits <MyGoofyType>::max () <<
    " which means it can hold all numbers with  " <<
    numeric_limits <MyGoofyType>::digits10 <<
    " decimal digits." << endl;
```

iterator::operator[]

- All random access iterators have `operator []`
- It makes them behave like arrays (as in C)
- Note: `rend () [0]` is the same as `end () [-1]`

```
deque <MyClass>     d;
cout << "Fifth element from the end is " << d.end () [-5] << endl;
```

swap

- swap interchanges two objects
- It's easy to write an exception-safe swap
- It's easy to then write an exception-safe assignment operator
- swap lets you force a vector release its allocated memory

```
Class&
Class::operator = (
    const Class&        inOriginal)
{
    Class   copy (inOriginal);
    swap (*this, copy);
    return *this;
}
```

```
namespace std {
    swap (
        Class&          inLeft,
        Class&          inRight)
    {
        swap (inLeft.m1, inRight.m1);
        swap (inLeft.m2, inRight.m2);
        // etc
    }
}
```

```
vector <int>        v;

// Do something to grow the vector

v.clear ();                     // Not required to release buffer
                                // (and usually doesn't)
swap (v, vector <int> ());  // Will release buffer
```

nth_element, partial_sort and partition

- partial_sort sorts elements close to the beginning of a range without sorting the whole range
- partition moves all elements satisying a criterion to the beginning of a range
- nth_element moves an element into its sorted position and all the rest on the appropriate side of that element
- Avoid fully sorting containers when you just need a partial sort

```
vector <int>    data;

// Fill with: 13, 8, 5, 3, 2, 1, 1
```

```
partial_sort (data.begin (), data.begin () + 4, data.end ());

// 1, 1, 2, 3, [5, 8, 13 in some order]

partition (
    data.begin (),
    data.end (),
    bind2nd (
        greater <int> (),
        2
    )
);

// [3, 5, 8, 13 in some order], [1, 1, 2 in some order]

nth_element (data.begin (), data.begin () + 4, data.end ());

// [1, 1, 2, 3 in some order], 5, [8, 13 in some order]
```

accumulate

- accumulate is excellent for gathering data about a container

```
bool IsVowel (
    char            inChar)
{
    switch (toupper (inChar)) {
        case 'A': case 'E': case 'I':
        case 'O': case 'U': case 'Y':
            return true;

        default:
            return false;
    }
}
```

```
int AccumulateVowelCount (
    int             inAccumInput,
    const string&   inString)
{
    return inAccumInput + count_if (
        inString.begin (),
        inString.end (),
        ptr_fun (&IsVowel)
    );
}
```

```
// count all vowels in an an array of strings
vector <string>     strings;
cout << "Total vowel count is " <<
    accumulate (
        strings.begin (),
        strings.end (),
        0,
        AccumulateVowelCount
    ) << endl;
```

transform

- Apply unary or binary transformations to whole containers or parts thereof

```
vector <string>                 strings;
vector <string::size_type>      lengths;

transform (
    strings.begin (),
    strings.end (),
    lengths.begin (),
    mem_fun_ref (&string::size)
);
```

```
vector <int>        v;
deque <int>         d;
int*                a = new int [v.size ()];

transform (
    v.begin (),
    v.end (),
    d.begin (),
    a,
    multiplies <int> ()
);
```

istreambuf_iterator

- Avoid overhead of formatted input and output
- Use `istreambuf_iterator` for unformatted I/O

```
transform (
    istreambuf_iterator <char> (cin.rdbuf ()),
    istreambuf_iterator <char> (),
    ostreambuf_iterator <char> (cout.rdbuf ()),
    ptr_fun (&toupper)
);
```

boost

- boost is a high-quality C++ library
- boost covers a wide range of applications, including smart poitners, regular expression, graph theory, numerics, parsing, and much, much more
- boost is very well documented and extensively peer-reviewed
- boost is a must if you are serious about using STL
- Significant chunks of boost will be part of STL in the future