Top Ten Things I Hate About STL

Miro Jurišić <meeroh@meeroh.org>

June 19, 2003

Unstable Iterators

- STL iterators are invalidated more often than you'd want
- STL iterator invalidation rules are complex

```
vector <int> v;
vector <int>::iterator i = v.begin ();
v.insert (i++, 1);      // bad
i= v.begin ();
v.insert (i, 1);
v.insert (i, 2);        // bad
```

```
i = v.begin ();
while (i != v.end ()) {
    v.erase (i);        // bad
}
```

- Familiarize yourself with iterator invalidation rules
- Use a debugging version of STL (CW 8+)

Unchecked Iterators

- Using an invalid iterator has undefined behavior
- No bounds checking

```
vector <int> v;
vector <int>::iterator i = v.end ();
*i = 0;                    // bad
```

- Use a debugging version of STL (CW 8+)

`wstring`

- `wstring` sounds like it's useful if you want Unicode support
- It's not
- Uses per-character copies and lexicographic comparisons
- Has no knowledge of Unicode decompositions
- `wchar_t` size platform-specific

- Use CFString until someone writes a good C++ Unicode string class

Thread safety

- C++ standard has no mention of threads
- Thread safety is up you and and your vendor
- Thread safety of containers is not an easy problem
- Locking granularity depends on usage scenarios
- Every possible solution is unacceptable in many valid scenarios

- Familiarize yourself with STL thread safety gurantees
- Familiarize yourself with your vendors' guarantees
- Familiarize yourself with third-party thread libraries (`boost::threads`)

vector <bool>

- `vector <bool>` isn't an STL container
- Does not obey STL container semantics (`&v[0]` doesn't compile)

- Don't use it
- Use `bitset`, `deque <bool>`, `boost::dynamic_bitset`

remove()

- remove() doesn't
- It only moves elements to the beginning

```
vector <int>        v;
v.push_back (1);
v.push_back (2);
v.push_back (3);                        // v.size () == 3
```

```
#ifdef WRONG
remove (v.begin (), v.end (), 2);       // v.size () == 3!
#else
erase (
    remove (v.begin (), v.end (), 3),
    v.end ()
);
#endif
```

- Remember that `remove()` doesn't, and call `erase()`

auto_ptr

- You can't store an `auto_ptr` in a container
- Doesn't work on `new []`
- `auto_ptr` is no magic dust

```
void f (Class* s1, Class* s2);

// Not exception safe, could leak!
f (new Class (), new Class ());

void f (auto_ptr <Class> s1, auto_ptr <Class> s2);

// Not exception safe, could leak!
f (auto_ptr <Class> (new Class ()), auto_ptr <Class> (new Class ()));
```

- Familiarize yourself with other smart pointers
  (`boost::[shared|scoped]_[ptr|array]`)

ptr_fun

- What is `ptr_fun` for anyway?
- But my code compiles without it!

```
bool IsEven (int x);

vector <int> v;

find_if (v.begin (), v.end (), &IsEven);                    // ok
find_if (v.begin (), v.end (), ptr_fun (&IsEven));          // ok

find_if (v.begin (), v.end (), not1 (&IsEven));             // error
find_if (v.begin (), v.end (), not1 (ptr_fun (&IsEven)));   // ok
```

- It is easier to consistently use `ptr_fun`
- Using `ptr_fun` will not break your code
- Using `ptr_fun` may make your code faster

reverse_iterator

- Some insertions and deletions require forward iterators
- Conversion of reverse iterators to forward iterators can be confusing

```
vector <int>::iterator              fi;
vector <int>::reverse_iterator      ri = fi;

v.insert (fi, x);           // Is the same as
v.insert (ri.base (), x);

v.erase (fi);               // Is the same as
v.erase ((++ri).base ()));  // !
```

- Don't use reverse iterators if you don't have to
- When you have to convert to a forward iterator, be careful to adjust as necessary

Error Messages

- STL error messages are long
- STL messages are complex
- STL messages are hard to decypher

- Beg your compiler vendors to improve the compiler
- Use STLFilt
- Practice