The Library Extensions Technical Report

Matt Austern

August 9, 2005

Outline

- History
- Some highlights of TR1
- The future

The C++ Standard

- J16 founded in 1989, WG21 in 1991
- Standard finalized in late 1997, published as ISO/IEC 14882:1998
- ~300 pages for core, library
- Complete implementations: around 2001

Post-standard work

- Two meetings a year instead of three
- 1998-2003: maintenance mode
- Technical Corrigendum 1: ISO/IEC 14882:2003
- Starting to work on “C++0x” now

Sample defect report

8. Locale::global lacks guarantee

Section: 22.1.1.5 [lib.locale.statics] **Status**: TC
Submitter: Matt Austern **Date**: 24 Dec 1997

It appears there's an important guarantee missing from clause 22. We're told that invoking locale::global(L) sets the C locale if L has a name. However, we're not told whether or not invoking setlocale(s) sets the global C++ locale.

Why library extension?

- Stability more important for core
  - Getting changes into compilers is hard
  - No such thing as a “pure” extension
  - Core language is mostly good enough
- C++ library is too small
  - A fraction the size of Java or Python
  - Little high-level application support

The Python standard library

- Containers and strings (regexps)
- I/O (serialization, sockets, mmap)
- Threading
- DBM and Berkeley DB
- FTP, HTTP, POP3 and IMAP
- SGML, XML, and HTML
- MIME and base64
- Image and audio manipulation
- …

The C++ standard library

- Low-level language support
- Everything in the C standard library
- Strings, iostreams, and locales
- Valarray and complex numbers
- The STL

Extending the library: a timeline

- Santa Cruz, 1998: BOOST
- Copenhagen, 2001: TR project started
- Santa Cruz, 2002: first proposals accepted
- Kona, 2003: last proposals accepted
- Lillehammer, 2005: technical work finished

ISO/IEC DTR 19768

Doc No: N1836=05-0096
Date: 2005-06-24
Reply to: Matt Austern
austern@google.com

Draft Technical Report on C++ Library Extensions

6 Containers [tr.cont]

1 This clause describes components that C++ programs may use to organize collections of information.

2 The following subclauses describe tuples, fixed size arrays, and unordered associated containers, as summarized in Table 19.

Table 19: Container library summary

Subclause	Header(s)
6.1 Tuple types	`<tuple>` `<utility>`
6.2 Fixed size array	`<array>`

6.3 Unordered associative containers	`<functional>`
	`<unordered_set>`
	`<unordered_map>`

6.1 Tuple types [tr.tuple]

1 6.1 describes the tuple library that provides a tuple type as the class template `tuple` that can be instantiated with any number of arguments. An implementation can set an upper limit for the number of arguments. The minimum value for this implementation quantity is defined in Annex A. Each template argument specifies the type of an element in the `tuple`. Consequently, tuples are heterogeneous, fixed-size collections of values.

6.1.1 Header `<tuple>` synopsis [tr.tuple.synopsis]

```
namespace std {
namespace tr1 {
  // [6.1.3] Class template tuple
  template <class T1 = unspecified,
            class T2 = unspecified,
            ...,
            class TM = unspecified> class tuple;

  // [6.1.3.2] Tuple creation functions
  const unspecified ignore;

  template<class T1, class T2, ..., class TN>
    tuple<V1, V2, ..., VN> make_tuple(const T1&, const T2& , ..., const TN&);
```

What's in TR1

- Low-level utilities
  - Smart pointers
- Improved STL function objects
- Type traits
- Numerics
  - Random numbers
- Containers
  - Tuples
  - Hash tables
- Regular expressions
- C99

Smart pointers

- Techniques for managing resource lifetime
  - Delete by hand
  - RAII
  - Garbage collection
- auto_ptr
  - Useful for simple scopes
  - Useless for copying or shared ownership

std::tr1::shared_ptr

Object is destroyed when last pointer disappears

```
struct A {
  A() { cout << "Create" << endl; }
  A(const A&) { cout << "Copy" << endl; }
  ~A() { cout << "Destroy" << endl; }
};

int main() {
  shared_ptr<A> p1(new A);
  shared_ptr<A> p2 = p1;
  assert (p1 != NULL && p2 != NULL && p1 == p2);
}
```

Example: mix STL and OOP

```
class Object {
public:
  virtual ~Object();
  static Object* make(const string& name);
private:
  Object(const Object&);
  void operator=(const Object&);
...
};

vector<Object> v;                          // wrong
vector<tr1::shared_ptr<Object> > v;        // ok
```

Problem: return multiple values

Option 1: pointer arguments

```
void fn(int in,
        int* out1, int* out2, int* out3);
```

Option 2: std::pair

```
pair<Iter, bool> insert(const Value&);
```

Option 3: ad hoc structure

```
div_t div(int num, int denom);
```

Better solution: std::tr1::tuple

Function declaration:

```
tuple<int, int, int> fn(int in);
```

Don't even need temporary for return value!

```
int x, y, z;
tie(x, y, z) = fn(72);
```

Other tuple features

- Tuple construction:

```
tuple<int, int, float> t
 = make_tuple(1, 4, 9);
```

- Ignore some return values:

```
tie(x, tr1::ignore, y) = fn(27);
```

- Element access:

```
int n = get<1>(t);
```

Random numbers

Shuffle a vector:

```
random_shuffle(v.begin(), v.end());
```

But what if you want to use a seed?

```
my_rng rng(my_seed);
random_shuffle(v.begin(), v.end(), rng);
```

Standard library has no class to use for this.

Shuffling with TR1

```
variate_generator<minstd_rand,
                  uniform_int<int> >
  gen(minstd_rand(seed),
      uniform_int<int>());

random_shuffle(v.begin(), v.end(), gen);
```

- Can “mix and match” engines and distributions.

Other distributions

- Example: random delay of 5m ± δ

```
normal_distribution<double> g(300., 5.);
minstd_rand eng;
…
double delay = g(eng);
```

Hash tables

- Fast lookup by key
- Better performance than trees for many applications
- First proposed for the STL in 1995
- Shipped by SGI, GNU, Dinkumware, Metrowerks,…

Hash tables (example)

```
int main()
{
  typedef unordered_map<string, unsigned long> Map;
  Map colors;

  colors["black"] = 0xff0000ul;
  colors["red"]   = 0xff0000ul;
  colors["green"] = 0x00ff00ul;
  colors["blue"]  = 0x0000fful;
  colors["white"] = 0xfffffful;

  for (Map::iterator i = colors.begin();
       i != colors.end();
       ++i)
  cout << i->first << " -> " << i->second << endl;
}
```

Regular expressions

- Similar to Python model
  - `regex` object to represent a pattern
  - Algorithms: `regex_match`, `regex_search`, `regex_replace`.
  - `match_results` (`smatch` or cmatch) for details.
- Based on ECMAscript
- Many options, including alternate syntaxes.

Example: find matching lines

```
bool
do_grep(const string& exp, istream& in, ostream& out)
{
  regex r(exp);
  bool found_any = false;
  string line;

  while (getline(in, line))
    if (regex_search(line, r)) {
      found_any = true;
      out << line;
    }

  return found_any;
}
```

Finding submatches

```
const string datestring = "10/31/2004";
…
const regex r("(\\d+)/(\\d+)/(\\d+)");
smatch fields;
if (!regex_match(datestring, fields, r))
  throw runtime_error(”invalid date");

const string month = fields[1];
const string day = fields[2];
const string year = fields[3];
```

Search and replace

```
const string date = "10/31/2004";
…
const regex r("(\\d+)/(\\d+)/(\\d+)");
const string date2
    = regex_replace(date, r, "$2/$1/$3");
```

Iterating through all matches

```
vector<string>
simple_tokenizer(const string& s) {
  const regex pat("[a-zA-Z]+");
  sregex_token_iterator first(s.begin(), s.end(),
                              pat);
  sregex_token_iterator last;
  return vector<string>(first, last);
}
```

What I left out

- Better function object support
  - Function and reference wrappers
  - bind and mem_fn
- Fixed-size arrays: `array<int, 10>`
- Special functions: J_ν, ζ, etc.
- C99 ("it just works")
- Type traits

The future: C++0x and TR2

- C++0x comes *before* TR2
- October '05: cutoff for new C++0x proposals
- October '06: cutoff for TR2 proposals
- April '07: cutoff for C++0x "cleanup" proposals

- **If you have ideas for new C++ libraries, now would be a good time.**