# An Introduction to TR1 and Boost

## A Shotgun-Firehose Approach

**Scott Meyers, Ph.D.**
Software Development Consultant

smeyers@aristeia.com
http://www.aristeia.com/

Voice: 503/638-6028
Fax: 503/638-6614



**Last Revised: 5/7/07**

## Presentation Overview

- TR1 and Boost
- Selective Survey of TR1 Contents.
  - 7 of 14 libraries.
- TR2
- Selective Survey of Boost Beyond TR1.
  - 7 of 69+ libraries.
- Further Reading

## What is TR1?

- Standard C++ Committee Library "Technical Report 1."
- The basis for new library functionality to be in standard C++ 200x (C++0x).
- TR1 functionality is in namespace std::tr1.
- TR1-like functionality in C++0x will probably be in std.
  - Such functionality may not be identical to that in TR1.
    - C++0x will have language features TR1 itself can't rely on.





## TR1 Summary

| New Functionality | Summary |
|---|---|
| Reference Wrapper | Objects that act like references |
| Smart Pointers | Reference-counting smart pointers |
| Getting Function Object Return Types | Useful for template programming |
| Enhanced Member Pointer Adapter | 2nd-generation mem_fun/mem_fun_ref |
| Enhanced Binder | 2nd-generation bind1st/bind2nd |
| Generalized Functors | Generalization of function pointers |
| Type Traits | Compile-time type reflection |
| Random Numbers | Supports customizable distributions |
| Mathematical Special Functions | Laguerre polynomials, beta function, etc. |
| Tuples | Generalization of pair |
| Fixed Size Array | Like vector, but no dynamic allocation |
| Hash Tables | Hash table-based set/multiset/map/multimap |
| Regular Expressions | Generalized regex searches/replacements |
| C99 Compatibility | 64-bit ints, <cstdint>, new format specs, etc. |

# TR1 Itself

TR1 is a *specification*:

- Aimed at implementers, not users.
- Lacks background, motivation, rationale for functionality it specifies.
- Doesn't stand on its own.
  - E.g., assumes information in the C++ Standard.





# Understanding TR1

To *understand* the functionality in TR1:

- Consult the Further Reading.
- Look at the extension proposals.
  - Links are available at Scott Meyers' TR1 Information web page, http://www.aristeia.com/EC3E/TR1_info.html.

| EC++ Page | *Effective C++, Third Edition* Name | TR1 Name | Proposal Document |
|---|---|---|---|
| 265 | Smart Pointers | Smart Pointers | n1450 |
| 265 | tr1::function | Polymorphic Function Wrappers | n1402 |
| 266 | tr1::bind | Function Object Binders | n1455 |
| 266 | Hash Tables | Unordered Associative Containers | n1456 |
| 266 | Regular Expressions | Regular Expressions | n1429 |
| 266 | Tuples | Tuple Types | n1403 (PDF) |
| 267 | tr1::array | Fixed Size Array | n1479 |
| 267 | tr1::mem_fn | Function Template mem_fn | n1432 |
| 267 | tr1::reference_wrapper | Reference Wrappers | n1453 |
| 267 | Random Number Generation | Random Number Generation | n1452 |
| 267 | Mathematical Special Functions | Mathematical Special Functions | n1422 |
| 267 | C99 Compatibility Extensions | C Compatibility | n1568 |
| 267 | Type Traits | Metaprogramming and Type Traits | n1424 |
| 267 | tr1::result_of | Function Return Types | n1454 |

## What is Boost?

- A volunteer organization and a web site (boost.org).
- A place to try out prospective C++ library enhancements.
- A repository for C++ libraries that are
  - Open-source
  - Portable
  - Peer-reviewed
  - Available under a "non-viral" license.





## Boost and TR1

Boost motivated and implements most of TR1:

- 10 of 14 libraries in TR1 are modeled on Boost libraries.
  - Boost libraries are executable, TR1 isn't.
    - Compilers currently ship with little TR1 support.
- Boost is committed to developing a full TR1 implementation.
- Boost also offers a portable TR1 façade:
  - Uses native TR1 functionality when available.
  - Uses Boost functionality otherwise (if it exists).

Other full or partial TR1 implementations are also available:

- Dinkumware
- Gnu, Metrowerks

## TR1 vs. Boost

Boost ≠ TR1:

- Boost offers much more functionality than in TR1.
  - 69 libraries in current release (version 1.33.1 – December 2005)
    - Several libraries have been accepted since then.
- Currently, not all TR1 functionality is available at Boost.
- TR1 functionality may differ from "equivalent" Boost libraries:
  - Examples (for Boost 1.33.1):
    - tr1::is_base_of<T, T> yields true, boost::is_base_of<T, T> yields false.
    - tr1::array<T> allows size 0, boost:array<T> doesn't.
  - Boost will bring its libraries into conformance with TR1.
- Other TR1 implementations may differ from those at Boost.
  - TR1 specifies *interfaces*, not implementations.





## TR1/Boost Summary

- TR1 is a specification for new standard library functionality.
- Boost is the premier repository of open-source, portable, peer-reviewed C++ libraries.
- Much TR1 functionality is available from Boost and others.
- Boost offers many non-TR1 libraries, too.

## A Selective Survey of TR1





## TR1 Smart Pointers

Motivation:

- Smart pointers simplify resource management.
  - E.g., prevention of leaks when exceptions are thrown.
- auto_ptr supports only exclusive ownership:
  - This leads to surprising copy semantics.
    - auto_ptrs can't be stored in containers.
- A standard shared-ownership smart pointer is needed:
  - Should offer "normal" copy semantics.
    - Hence may be stored in containers.
  - Many versions have been created and deployed.
    - Typically based on reference counting.

# TR1 Smart Pointers

- Declared in <memory>.
- tr1::shared_ptr is a reference-counting smart pointer.
- Pointed-to resources are released when the ref. count (RC) → 0.

```
{
  std::tr1::shared_ptr<Widget> p1(new Widget);   // RC = 1
  std::tr1::shared_ptr<Widget> p2(p1);           // RC = 2
  …
  p1->doThis();                                   // use p1 and p2
  if (p2) p2->doThat();                           // like normal ptrs
  …
  p2.reset();                                     // make p2 null;
  …                                               // RC = 1
}                                                 // RC = 0; Widget
                                                  // deleted
```





# tr1::shared_ptr Constructors

- Default, copy, from raw pointer.
- From compatible auto_ptr, tr1::shared_ptr, or tr1::weak_ptr.
- From this:
  - It's a raw pointer, but other shared_ptrs might already exist!

```
std::tr1::shared_ptr<ISomething>
Widget::get()
{                                                      // dangerous!
  return std::tr1::shared_ptr<ISomething>(this);       // could create a
}                                                      // new ref count!

std::tr1::shared_ptr<ISomething>
Widget::get()
{                                                      // okay, no
  return                                               // chance
    std::tr1::shared_ptr<ISomething>(shared_from_this()); // of a
}                                                      // new RC
```

## Some tr1::shared_ptr Features

- Access to underlying raw pointer:
  - Useful for communicating with legacy APIs.

```
void oldAPI(Widget *pWidget);
std::tr1::shared_ptr<Widget> spw(new Widget);
oldAPI(spw.get());
```

- Access to reference count:

```
if (spw.unique()) …                     // always efficient
std::size_t refs = spw.use_count();     // may be inefficient
```





## Some tr1::shared_ptr Features

- Operators:
  - static_pointer_cast, dynamic_pointer_cast, const_pointer_cast

```
void someFunc(std::tr1::shared_ptr<Widget> spw)
{
  using namespace std::tr1;
  if (shared_ptr<Gadget> spg =
        dynamic_pointer_cast<Gadget>(spw)) {
  …                                   // spw really points to a Gadget
  }
}
```

  - Relationals: ==, !=, <
  - Output: <<

```
std::tr1::shared_ptr<Widget> spw;
…
std::cout << spw;
```

## Support for Incomplete Types

Unlike auto_ptr, shared_ptr supports incomplete types:

```
class Widget;                       // incomplete type
std::auto_ptr<Widget> ap;           // error!
std::tr1::shared_ptr<Widget> sp;    // fine
```

tr1::shared_ptr thus allows common coupling-reduction strategies.

- E.g., pimpl.





## Support for Inheritance Conversions

auto_ptr fails to support some inheritance-based conversions that shared_ptr offers:

```
class Base { … };
class Derived: public Base { … };

std::auto_ptr<Derived> create();    // func. returning auto_ptr<Der>
void use(std::auto_ptr<Base>);      // func. taking auto_ptr<Base>

use(create());                      // error!  won't compile


std::tr1::shared_ptr<Derived> create();     // same code, but with
void use(std::tr1::shared_ptr<Base>);       // shared_ptr

use(create());                              // fine
```

Note: the auto_ptr-based code (erroneously) compiles on some platforms.

## Custom Deleters

By default, tr1::shared_ptrs use delete to release resources, but this can be overridden:

```
Widget* getWidget();                // API to acquire/release a
void releaseWidget(Widget*);        // resource

{
  std::tr1::shared_ptr<Widget> pw(getWidget(), releaseWidget);
  …
}                                   // releaseWidget called
```

The default deleter is a function invoking delete.

- Out of the box, the cross-DLL delete problem goes away!

Deleters are really *releasers* (as above):

- E.g., a deleter could release a lock.





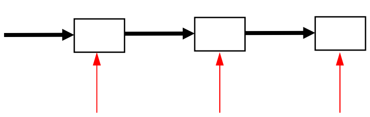
## Weak Pointers

tr1::weak_ptrs are like raw pointers, but they know when they dangle:

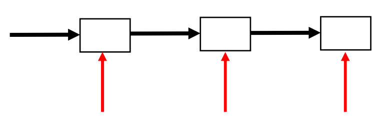
- When a resource's RC → 0, its tr1::weak_ptrs *expire.*
  - ➡ The tr1::shared_ptr releasing a resource expires all tr1::weak_ptrs:

```
std::tr1::shared_ptr<Widget> spw(new Widget);    // RC = 1
std::tr1::weak_ptr<Widget> wpw(spw);             // RC remains 1
…
if (!wpw.expired()) …                            // if RC >= 1 …
```

- Useful for "observing" data structures managed by others.



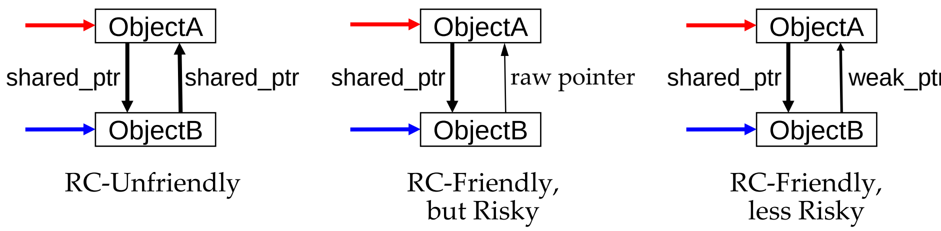
Pointer observers -- Risky



tr1::weak_ptr observers – Less Risky

## Weak Pointers

- Also to facilitate cyclic structures that would otherwise foil RC:
  - Consider reassigning the red pointer, then later the blue one.








## Weak Pointers

tr1::weak_ptrs aren't really smart pointers!

- No dereferencing operators (no operator-> or operator*).
- No implicit nullness test (conversion to something boolish).

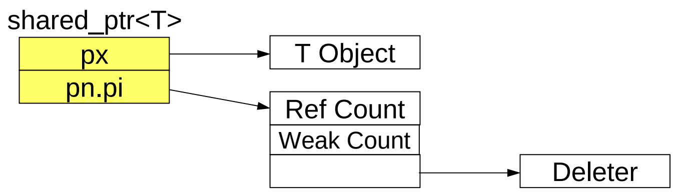
To use a tr1::weak_ptr as a pointer, create a tr1::shared_ptr from it:

```
std::tr1::weak_ptr<Widget> wpw(spw);
wpw->doSomething();                         // risky and won't compile
if (!wpw.expired()) wpw->doSomething();     // won't compile

std::tr1::shared_ptr<Widget> pw1(wpw);      // create tr1::shared_ptr;
                                            // throws if wpw's expired
pw1->doSomething();                         // fine (if pw1 constructed)

std::tr1::shared_ptr<Widget> pw2(wpw.lock()); // pw2 is null if wpw's
                                              // expired
if (pw2) pw2->doSomething();                  // fine
```

## Cost of tr1::shared_ptrs

Sample implementation (Boost 1.33.1):

- 2 words in size (pointer to object, pointer to RC).




- Uses dynamically allocated memory for the RC.
- Resource release (i.e., deletion) via a virtual function call ⇒ vtbls.
- Incurs cost for weak_ptr count even if no weak_ptrs are used.
  - ➡ Increases cost of RC manipulations in thread-safe implementations.
    - ◆ The next implementation should avoid this cost.




## TR1 Smart Pointers Summary

- tr1::shared_ptrs use reference counting to manage resource lifetimes.
- They support incomplete types, inheritance-based conversions, custom deleters, and C++-style casts.
- tr1::weak_ptrs are useful for detecting dangling pointers and for breaking smart pointer cycles.
- TR1 smart pointers incur a size and space cost compared to std::auto_ptrs or built-in pointers.

## TR1 Hash Tables

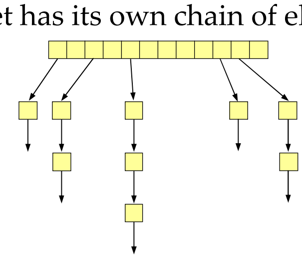
Motivation:

- They're an "obvious omission" in the library.
  - ➡ They're widely useful.
  - ➡ They were omitted from C++98 only due to lack of time.




## TR1 Hash Tables

- Declared in <unordered_set> and <unordered_map>.
  - ➡ Default hashing functionality declared in <functional>.
- Designed to not conflict with pre-TR1 implementations.
  - ➡ Multiple vendors offer hash_set, hash_map, hash_multiset, hash_multimap.
    - ◆ Interfaces vary – hence the need for standardization.
    - ◆ TR1 uses the names unordered_set, unordered_map, etc.
  - ➡ Compatible with hash_* interfaces where possible.
- Each bucket has its own chain of elements:



*Conceptual diagram!*
*Implementations vary!*

- Bucket count can change dynamically.

## Containers' Characteristics

- The usual members exist:
  - iterator/const_iterator and other typedefs.
  - begin/end, insert/erase, size, swap, get_allocator, etc.
- Also 3 associative container functions: find, count, equal_range.
  - lower_bound/upper_bound are absent.
- unordered_map/unordered_multimap offer operator[].
- Relationals are *not* supported: no <, <=, ==, !=, >=, >
  - Indeterminate ordering makes these too expensive.
- Only forward iteration is provided.
  - No reverse_iterators, no rbegin/rend.
  - Some implementations offer these, anyway.





## TR1 Hash Table Parameters

Hashing and equality-checking types are template parameters:

```
template<class Value,                          // copied out of TR1
         class Hash = hash<Value>,
         class Pred = std::equal_to<Value>,
         class Alloc = std::allocator<Value> >
class unordered_set { … };

template<class Key,
         class T,
         class Hash = hash<Key>,
         class Pred = std::equal_to<Key>,
         class Alloc = std::allocator<std::pair<const Key, T> > >
class unordered_map { … };
```

## Hashing Functions

Defaults are provided for built-in and string types:

```
class Widget { … };
std::tr1::unordered_set<int> si;                    // all use default hash
std::tr1::unordered_multiset<double> md;           // func for shown types
std::tr1::unordered_map<std::wstring, int>mwi;
std::tr1::unordered_multimap<Widget*, std::string> mmwps;
```

To override a default or hash a UDT, use a custom functor:

```
struct IntHasher: public std::unary_function<int, std::size_t> {
  std::size_t operator()(int i) const { … };
};
std::tr1::unordered_map<int, std::string, IntHasher> mis;

struct WidgetHasher: public std::unary_function<Widget, std::size_t> {
  std::size_t operator()(const Widget& w) const { … };
};
std::tr1::unordered_set<Widget, WidgetHasher> sw;
```





## Operations for Bucket Count and Load Factor

Constructors allow a floor on bucket count (B) to be specified:

```
std::tr1::unordered_set<int> s;        // B chosen by implementation
std::tr1::unordered_set<int> s(53);    // B >= 53.  (Other ctor forms
                                        // support bucket floor, too.)
```

A table's *load factor (z)* is the average number of elements/bucket:

- z = *container*.size()/B.
- z can be queried, and a ceiling for it can be "hinted" (requested):

```
float z = s.load_factor();              // get current load factor
s.max_load_factor(0.75f);               // request ceiling for z;
                                        // future insertions may
                                        // increase B so that z <= .75,
                                        // then rehash s to use new B
float z_max = s.max_load_factor();      // get current z_max (defaults to 1)
```

Because max_load_factor(*z*) is only a request, it's possible that *container*.load_factor() > *container*.max_load_factor().

## Rehashing

Explicit rehashing can also change the bucket count and load factor:

```
std::size_t n = computeNewB();
s.rehash(n);                         // reorganize s so that B >= n
                                     // and s.size()/B <= z_max
```

Rehashing (implicitly or explicitly) invalidates iterators.

- But not pointers or references.




## Iterating Over Bucket Contents

Useful for e.g., monitoring performance of hashing functions.

```
using namespace std;
tr1::unordered_set<string> s;
…
size_t numBuckets = s.bucket_count();          // # buckets
for (int b = 0; b < numBuckets; ++b) {
  cout << "Bucket " << b << " has "
       << s.bucket_size(b) << " elements: ";   // # elems in bucket b
  copy(s.begin(b), s.end(b),                   // iters for bucket b
       ostream_iterator<string>(cout, ' '));
}
```

## TR1 Hash Tables Summary

- Unordered containers based on hash tables with open hashing.
- Only forward iteration is supported.
- Maximum load factor can be dynamically altered.
- There is support for iterating over individual buckets.





## TR1 Regular Expressions

Motivation:

- Regular expression (RE) functionality is widely useful.
- Many programming languages and tools support it.
- C RE libraries support only char-based strings.
  - ➡ C++ should support wchar_t strings and std::strings, too.

## TR1 Regular Expressions

- Declared in <regex>.
- RE objects modeled on string objects:
  - Support char, wchar_t, locales.
- Supports multiple RE syntaxes:
  - Modified ECMAScript is the default

```
std::tr1::regex capStartRegex("[A-Z][[:alnum:]]*");  // alnum substr.
                                                      // starting with a
                                                      // capital letter

std::tr1::regex SSNRegex("\\d{3}-\\d{2}-\\d{4}");    // looks like a SSN
                                                      // (ddd-dd-dddd)
```

  - Alternatives: POSIX Basic, POSIX Extended, awk, grep, egrep
- Offers control over state machine behavior:

```
std::tr1::regex filenameRegex(          // regex for some
  "\\w+\\.((txt)|(dat)|(log))",         // .txt, .dat, and .log files;
  std::tr1::regex::icase |              // ignore case during search;
  std::tr1::regex::optimize             // match speed more important
);                                      // than regex ctor speed
```





## Fundamental Functionality

- regex_match: Does the RE match the complete string?
- regex_search: Does the RE occur in the string?
- regex_replace: Replace text matching RE with other text.
  - Replacement isn't in-place: new text is returned.

Matches are held in match_results objects. Iteration is supported:

- regex_iterator: Iterate over match_results objects for a string.
- regex_token_iterator: Iterate over matched strings in a string.

These are templates. You normally use named instantiations:

- For strings: smatch/sregex_iterator/sregex_token_iterator
- For wstrings: wsmatch/wsregex_iterator/wsregex_token_iterator
- For char*s: cmatch/cregex_iterator/cregex_token_iterator
- For wchar_t*s: wcmatch/wcregex_iterator/wcregex_token_iterator

## Examples

Does text look like an SSN?

```
const std::tr1::regex SSNRegex("\\d{3}-\\d{2}-\\d{4}");

bool looksLikeSSN(const std::string& text)
{
  return std::tr1::regex_match(text, SSNRegex);
}
```

Does text contain a substring that looks like an SSN?

```
bool mayContainSSN(const std::string& text)
{
  return std::tr1::regex_search(text, SSNRegex);
}
```





## Examples

Collect all substrings that look like SSNs:

```
void possibleSSNs1(const std::string& text, std::list<std::string>& results)
{
  std::string::const_iterator b(text.begin()), e(text.end());
  std::tr1::smatch match;

  while (std::tr1::regex_search(b, e, match, SSNRegex)) {
    results.push_back(match.str());
    b = match[0].second;
  }
}
```

Or:

```
void possibleSSNs2(const std::string& text, std::list<std::string>& results)
{
  std::tr1::sregex_token_iterator b(text.begin(), text.end(), SSNRegex);
  std::tr1::sregex_token_iterator e;

  std::copy(b, e, std::back_inserter(results));
}
```

## Examples

Replace all substrings that look like SSNs with dashes:

- Note use of "the swap trick" to effect in-place replacement.

```
void dashifySSNs(std::string& text)
{
  const std::string dashes("-----------");
  std::tr1::regex_replace(text, SSNRegex, dashes).swap(text);
}
int main()
{
  using namespace std;
  string data("123-45-6789x777-77-7777abc");
  cout << data;                // 123-45-6789x777-77-7777abc
  dashifySSNs(data);
  cout << data;                // -----------x-----------abc
}
```





## TR1 Regular Expressions Summary

- Several RE syntaxes and string representations are supported.
- Search functions are regex_match and regex_search.
- regex_replace performs global search and replace; the result is a new string.
- Iteration over matches is supported.

## TR1 Tuples

Motivation:

- pair should be generalized.
- Tuple utility demonstrated by other languages.





## TR1 Tuples

- tr1::tuple declared in <tuple>, helper templates in <utility>.
- Offers fixed-size heterogeneous containers:
  - Implementations may limit sizes to as little as 10.
  - Fixed-size ⇒ no dynamic memory ⇒ no allocator.

```
class Name { ... };
class Address { … };
class Date { … };

std::tr1::tuple<Name, Address, Date>        // function to return
employeeInfo(unsigned employeeID);          // employee name,
…                                           // address, hire date

unsigned eid;
…
std::tr1::tuple<Name, Address, Date>        // initialize tuple with
  info(employeeInfo(eid));                  // value from
                                            // employeeInfo
```

## Using get

Tuple elements are accessed via tr1::get:

- Takes a compile-time index; indices start at 0:

```
Name empName(get<0>(info));
Address empAddr(get<1>(info));
Date empHDate(get<2>(info));
```

- A compile-time index!
  - tr1::get is a template, and the index is a *template argument*.

```
int nameIdx = 0;
Name empName(get<nameIdx>(info));                    // error!
```

  - for loops over tr1::tuples aren't possible.





## Using tie

tr1::tie can perform the work of multiple tr1::gets:

```
std::tr1::tie(empName, empAddr, empHDate) =     // assign to all 3
  employeeInfo(eid);                            // variables
```

tr1::ignore can be used within tr1::tie to get only selected elements:

```
std::tr1::tie(empName, empAddr, ignore) =     // assign only name
  employeeInfo(eid);                          // and address

std::tr1::tie(ignore, empAddr, ignore) =      // assign address only.
  employeeInfo(eid);                          // (Here, using get
                                              // would be easier.)
```

## Using make_tuple

tr1::make_tuple is a generalization of make_pair:

```
class Employee {
public:
  Name name() const;
  Address address() const;
  Date hireDate() const;
  …
};
Employee findByID(unsigned eid);


std::tr1::tuple<Name, Address, Date>
employeeInfo(unsigned employeeID)
{
  Employee e(findByID(employeeID));
  return tr1::make_tuple(e.name(), e.address(), e.hireDate());
}
```





## Tuple Reflection

There's support for compile-time reflection:

```
template<typename Tuple>
void someFunc(Tuple t)
{
  std::size_t numElems =                              // # elems
    std::tr1::tuple_size<Tuple>::value;               // in Tuple
  typedef
    typename std::tr1::tuple_element<0, Tuple>::type  // type of
    FirstType;                                        // 1st elem
  …
}
```

## Other tr1::tuple Functionality

The usual STL container relationals (<, <=, ==, !=, >=, >):

- == and != tests use elementwise ==
- Other relational tests are lexicographical using only <:
  - ➡ Values are considered equal if they're equivalent (based on <)

pair<T1, T2> can often be used as a tr1::tuple<T1, T2>:

- A 2-element tr1::tuple can be created or assigned from a compatible pair.
- get<0> and get<1> both work on pairs.
- So do tuple_size and tuple_element.



## TR1 Tuples Summary

- Tuples are a generalization of std::pair.
- Element access is via compile-time index using get or via tie.
- Compile-time reflection is supported. It works on std::pairs, too.

## TR1 Fixed-Size Arrays

Motivation:

- Need an STL container with array-like performance:
  - Arrays aren't STL containers:
    - No begin, end, etc.
    - They don't know their size.
    - They decay into pointers.
  - vector imposes overhead:
    - Dynamic memory allocation.





## TR1 Fixed-Size Arrays

- Declared in <array>.
- Offers the expected container members:
  - iterator/const_iterator/reverse_iterator and other typedefs
  - begin/end, rbegin/rend, empty, size, relational operators, etc.
    - But swap runs in *linear* – not constant – time.
- Also vectoresque members: operator[], at, front, back
- Contents are layout-compatible with C arrays.
  - Get a pointer to elements via data (as with string):

```
std::tr1::array<int, 5> arr;              // create tr1::array
…
int *pElements = arr.data();              // get pointer to elements
```

## TR1 Fixed-Size Arrays

Because tr1::arrays are fixed-size,

- No insert, push_back, erase, clear, etc.
- No dynamic memory allocation.
  - Hence no allocator.





## tr1::arrays are Aggregates

A tr1::array is an *aggregate*, so it can be brace-initialized.

- If initializer has too few values, zeros are added at end.
- If initializer has too many values, compilation fails.

```
std::tr1::array<int, 5> arr1 = { 1, 2, 3, 4, 5 };
std::tr1::array<short, 5> arr2 = {10, 20, 30 };       // last 2 values
                                                       // are init'd to 0
std::tr1::array<float, 1> arr3 = { 1, 2, 3, 4, 5 };   // error!  won't
                                                       // compile
```

## tr1::arrays are Aggregates

Because tr1::array is an aggregate:

- All members are public!
- Only default and copy construction is supported.
  - ➡ These constructors are compiler-generated.
  - ➡ Range construction is unavailable:

```
std::vector<int> v;
…
std::tr1::array<int, 10>              // error!  tr1::array supports only
  arr(v.begin(), v.begin()+10);       // default and copy construction
```





## tr1::arrays as Tuples

tr1::array<T, n> can sometimes be treated like tr1::tuple<T, T, …, T>:

```
class Widget { … };
const int arraySz = 10;
typedef std::tr1::array<Widget, arraySz> WidgetArray;

WidgetArray arr;                    // 10 Widgets are default-constructed

…

std::size_t numElements = std::tr1::tuple_size<WidgetArray>::value;

std::size_t elemSize =
  sizeof(std::tr1::tuple_element<0, WidgetArray>::type);

const Widget& value = std::tr1::get<0>(arr);

std::cout << "arr has " << numElements
          << " elements, each of size " << elemSize
          << ".  The first element's value is " << value
          << '\n';
```

## tr1::array vs. vector

- tr1::array is fixed-size, vector is dynamically sized.
- tr1::array uses no dynamic memory, vector does.
- tr1::array can be brace-initialized, vector can't.
- tr1::array::swap is linear-time and may throw, vector::swap is constant-time and can't throw.
- tr1::array can be treated like a tuple, vector can't.

Scott Meyers, Software Development Consultant
http://www.aristeia.com/


Page 55

## tr1::array vs. C Arrays

- tr1::array objects know their size, C arrays don't
- tr1::array allows 0 elements, C arrays don't
- tr1::array requires an explicit size, C arrays can deduce it from their initializer
- tr1:array supports assignment, C arrays don't
- tr1::array can be treated like a tuple, C arrays can't

**Given tr1::array, vector, and string, there is little reason to use C-style arrays any longer.**

Scott Meyers, Software Development Consultant
http://www.aristeia.com/


Page 56

## TR1 Fixed-Size Arrays Summary

- tr1::array objects are STLified C arrays.
- They support brace-initialization, but not range initialization.
- They support some tuple operations.
- Given tr1::array, std::vector, and std::string, there is little reason to use C-style arrays.





## TR1 Generalized Functors

Motivation:

- "Normal" (C-like) function pointers are rigid:
  - Exact parameter/return types and ex. specs. must be specified.
  - Can't point to nonstatic member functions.
  - Can't point to function objects.
- Member function pointers are rigid:
  - Exact parameter/return types and ex. specs. must be specified.
  - Can't point to non-member or static member functions.
  - Can't point to function objects.
- Useful to be able to refer to ***any callable entity compatible with a given calling interface***.
  - Convenient for developers.
    - Especially for callback interfaces.
  - Can help limit code bloat from template instantiations.

## Callable Entities

Something that can be called using C function call syntax:

- Functions, function pointers, function references:

```
void f(int x);                      // function
void (*fp)(int) = f;                // function pointer
void (&fr)(int) = *fp;              // function reference
int val;
…
f(val);                             // invoke any of above
fp(val);                            // using C call syntax
fr(val);
```





## Callable Entities

- Objects implicitly convertible to one of those:

```
class Widget {
public:
  typedef void (*FuncPtr)(int);
  operator FuncPtr() const;     // conversion to function ptr
  …
};
Widget w;                       // object with conversion to func ptr
int val;
…
w(val);                         // invoke (w.operator FuncPtr())()
                                // using C call syntax
```

## Callable Entities

- Function objects:

```
class Gadget {
public:
  void operator()(int);     // function call operator
  …
};
Gadget g;                   // object supporting operator()
int val;
…
g(val);                     // invoke w.operator() using C call syntax
```

Note that non-static member functions are not callable entities.

- They're not invoked using C function call syntax.





## Specifying a Calling Interface

Function pointer parameters can be declared 2 ways:

```
void f( void (*fp)(int) );          // f takes a fcn pointer fp
void f( void fp(int) );             // same thing
```

Parameter names may be omitted:

```
void g( int x );                    // g takes an int
void g( int );                      // ditto, but no param name
```

Thus:

```
void f( void (int) );               // same f as above, but no
                                    // param name
```

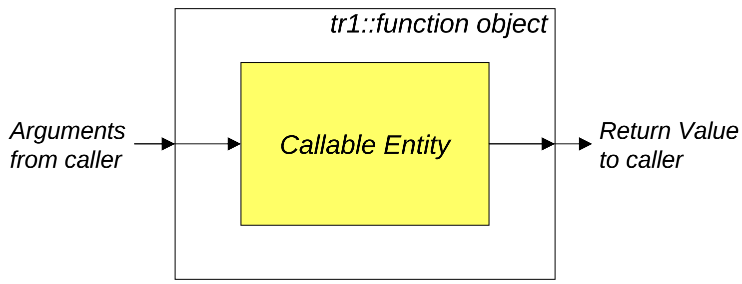
tr1::function types use this syntax to specify their calling interface:

```
void f (std::tr1::function<void (int)> func); // func refers to any callable
                                              // entity supporting this
                                              // interface
```

## Compatible Signatures

A callable entity is *compatible* with a tr1::function object if:

- The tr1::function object's parameter types can be converted to the entity's parameter types.
- The entity's return type can be converted to that of the tr1::function object.






## tr1::function Callback Example

A Button class supporting click-event callbacks:

- The callback parameter indicates a down- or up-click.

```
class Button: public SomeGUIFrameworkBaseClass {
public:
  …
  typedef std::tr1::function<void(short)> CallbackType;
  void setCallback(CallbackType cb)
  {
    clickHandler = cb;                 // assign tr1::function object
  }
  virtual void onClick(int upOrDown)   // invoked by GUI framework
  {
    clickHandler(upOrDown);            // "call" tr1::function object
  }
private:
  CallbackType clickHandler;
};
```

## tr1::function Callback Example

```
void mouseClickHandler(int eventType);    // non-member function
class MouseHandler {
public:
  …
  static int clicked(short upOrDown);     // static member function
};
void (*clicker)(int) = mouseClickHandler; // function pointer
Button b;
…
b.setCallback(mouseClickHandler);         // pass non-member
b.setCallback(MouseHandler::clicked);     // pass static member
b.setCallback(clicker);                   // pass function ptr
```

Note the (compatible) type mismatches:

- mouseClickHandler and clicker take int, not short
- MouseHandler::clicked returns int, not void





## tr1::function Callback Example

```
class MouseClickCallback {                  // class generating
public:                                     // function objects
  void operator()(short upOrDown) const;
};
Button b;
…
MouseClickCallback mccb;
b.setCallback(mccb);                        // pass function object
```

## Other tr1::function Characteristics

- Declared in <functional>
- Supports nullness testing:

```
std::tr1::function<signature> f;
…
if (f) …                              // fine
if (f == 0) …                         // also fine
```

- Disallows equality and inequality testing
  - ➡ Nontrivial to determine whether two tr1::function objects refer to equal callable entities.

```
std::tr1::function<signature> f1, f2;
…
if (f1 == f2) …                       // error!
if (f1 != f2) …                       // error!
```





## TR1 Generalized Functors Summary

- tr1::function objects are generalizations of function pointers.
- Can refer to any callable entity with a compatible signature.
- Especially useful for callback interfaces.
- Explicitly disallow tests for equality or inequality.

## TR1 Binder

Motivation:

- bind1st and bind2nd are constrained:
  - Can bind only first or second arguments.
  - Can bind only one argument at a time.
  - Can't bind functions with reference parameters.
  - Require adaptable function objects.
    - Often necessitates ptr_fun, mem_fun, and mem_fun_ref.





## TR1 Binder

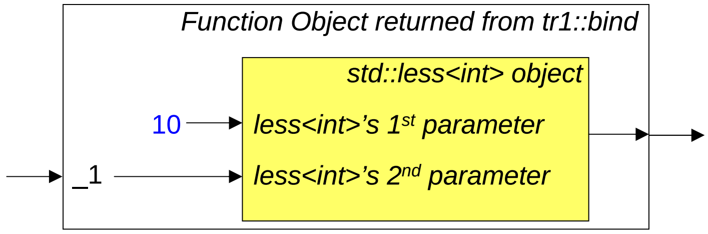
- Declared in <functional>.
- Produces a function object from:
  - A *callable object*:
    - A callable entity (e.g., function pointer or function object).
    - A member pointer.
  - A specification of which arguments are to be bound.

```
functionObject std::tr1::bind( callableObject,
                               1stArgBinding,
                               2ndArgBinding,
                               …
                               nthArgBinding);
```

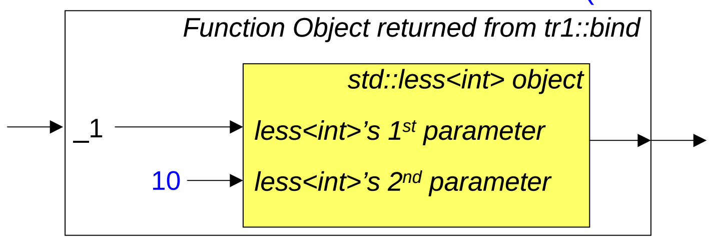
  - *Placeholders* allow mapping from arguments for tr1::bind's return value to callable object arguments.
    - _*n* specifies the *n*th argument passed to the function object returned by tr1::bind.

## Simple tr1::bind Examples

```
std::tr1::bind(std::less<int>(), 10, _1)    // same as
                                            // std::bind1st(std::less<int>(), 10)
```




```
std::tr1::bind(std::less<int>(), _1, 10)    // same as
                                            // std::bind2nd(std::less<int>(), 10)
```




Scott Meyers, Software Development Consultant
http://www.aristeia.com/

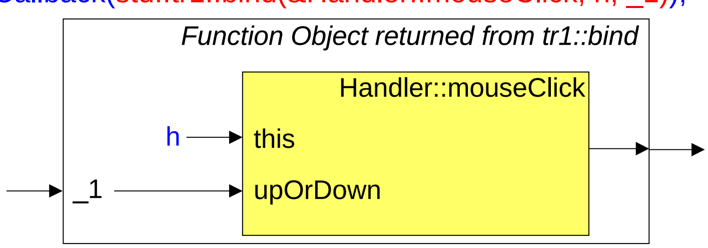
Page 71

## Binding Non-Static Member Functions

For non-static member functions, this comes from the first argument:

- Just like for bind1st and bind2nd.

```
class Handler {
public:
  void mouseClick(int upOrDown);
};

Handler h;
Button b;                          // from the tr1::function example
…
b.setCallback(std::tr1::bind(&Handler::mouseClick, h, _1));
```




Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Page 72

## Binding Beyond the 2nd Argument

Binding beyond the 2nd argument is easy:

```
class Point {
public:
  …
  void translate(int deltaX, int deltaY);
};

std::vector<Point> vp;
…

std::for_each(                                  // translate points
  vp.begin(), vp.end(),                         // in vp by (10, 20);
  std::tr1::bind(&Point::translate, _1, 10, 20) // note that deltaY
);                                              // is 3rd arg
```





## tr1::bind and Adapters

Unlike bind1st and bind2nd, tr1::bind needs no adapters:

```
class Point {
public:
  …
  void rotate(int degrees);
};

std::vector<Point> vp;
…

// without tr1::bind
std::for_each(vp.begin(), vp.end(),
              std::bind2nd(std::mem_fun_ref(&Point::rotate), 10));

// with tr1::bind
std::for_each(vp.begin(), vp.end(),
              std::tr1::bind(&Point::rotate, _1, 10));
```

## tr1::bind and tr1::function

tr1::bind's result is often stored in a tr1::function object:

```
class Button: public SomeGUIFrameworkBaseClass {    // from
public:                                             // tr1::function
  typedef std::tr1::function<void(short)> CallbackType;   // discussion

  void setCallback(CallbackType cb)
  { clickHandler = cb; }
  …

private:
  CallbackType clickHandler;
};

class Handler {                                     // from
public:                                             // earlier
  void mouseClick(int upOrDown);
};

Handler h;
Button b;
…
b.setCallback(std::tr1::bind(&Handler::mouseClick, h, _1));
```

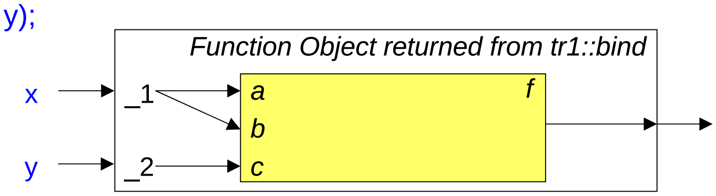




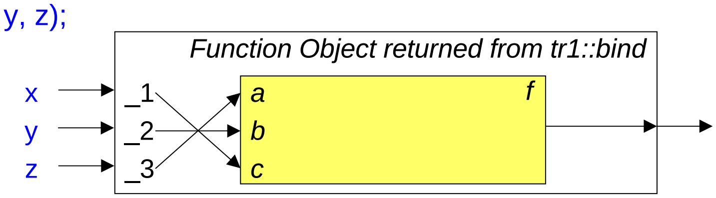
## Fun With tr1::bind

tr1::bind allows reordering and duplicating arguments:

```
void f(int a, int b, int c);
int x, y, z;
…
std::tr1::function<void(int, int)> f1 = std::tr1::bind(f, _1, _1, _2);
f1(x, y);
```




```
std::tr1::function<void(int, int, int)> f2 = std::tr1::bind(f, _3, _2, _1);
f2(x, y, z);
```



## TR1 Binder Summary

- tr1::bind generalizes bind1st and bind2nd.
- Has no need for ptr_fun, mem_fun, mem_fun_ref, or tr1::mem_fn.
- Results are often stored in tr1::function objects.





## Other TR1 Functionality

| New Functionality | Summary |
|---|---|
| Reference Wrapper | Objects that act like references |
| Smart Pointers | Reference-counting smart pointers |
| Getting Function Object Return Types | Useful for template programming |
| Enhanced Member Pointer Adapter | 2nd-generation mem_fun/mem_fun_ref |
| Enhanced Binder | 2nd-generation bind1st/bind2nd |
| Generalized Functors | Generalization of function pointers |
| Type Traits | Compile-time type reflection |
| Random Numbers | Supports customizable distributions |
| Mathematical Special Functions | Laguerre polynomials, beta function, etc. |
| Tuples | Generalization of pair |
| Fixed Size Array | Like vector, but no dynamic allocation |
| Hash Tables | Hash table-based set/multiset/map/multimap |
| Regular Expressions | Generalized regex searches/replacements |
| C99 Compatibility | 64-bit ints, <cstdint>, new format specs, etc. |

## TR2

Work is underway on TR2.

- May not be completed until *after* C++0x is finalized.

Example proposals:

- Boost Filesystem library.
- Boost Any library.
- Boost Lexical Conversion library.
- Boost Networking library.
- Boost String Algorithms library.
- Enhanced diagnostics library.
  - Based on Boost work for the Filesystem and Networking libraries.





## TR2

To get an idea of what's been proposed:

- Visit http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/
  and http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/
  and http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/
- Browse the papers aimed at the Library subgroup:
  - Not all such papers are proposals for new features.
  - Many TR2 proposals have "TR2" in the title.

| WG21 Number | J16 Number | Title | Author | Document Date | Mailing Date | Previous Version | Subgroup |
|---|---|---|---|---|---|---|---|
| SD-1 | 06-0000 | 2006 J16/WG21 document list | Clark Nelson | 2006-02-27 | 2006-02 | | |
| SD-2 | 06-0001 | ISO WG21 and INCITS J16 membership list | Clark Nelson | 2006-02-24 | 2006-02 | | |
| SD-5 | 03-0029 | WG21 and J16 (C++) Joint Mailing and Meeting Information | Herb Sutter | 2003-04-17 | 2006-02 | | |
| N1932 | 06-0002 | Random Number Generation in C++0X: A Comprehensive Proposal | W. Brown, M. Fischler, J. Kowalkowski, M. Paterno | 2006-02-23 | 2006-02 | | Library |
| N1933 | 06-0003 | Improvements to TR1's Facility for Random Number Generation | W. Brown, M. Fischler, J. Kowalkowski, M. Paterno | 2006-02-23 | 2006-02 | | Library |
| N1934 | 06-0004 | Filesystem Library Proposal for TR2 (Revision 2) | Beman Dawes | 2006-02-22 | 2006-02 | N1889=05-0149 | Library |
| N1935 | 06-0005 | C++ Standard Core Language Active Issues, Revision 40 | William M. Miller | 2006-02-24 | 2006-02 | N1929=05-0189 | Core |
| N1936 | 06-0006 | C++ Standard Core Language Defect Reports, Revision 40 | William M. Miller | 2006-02-24 | 2006-02 | N1930=05-0190 | Core |
| N1937 | 06-0007 | C++ Standard Core Language Closed Issues, Revision 40 | William M. Miller | 2006-02-24 | 2006-02 | N1931=05-0191 | Core |
| N1938 | 06-0008 | Lookup Issues in Destructor and Pseudo-Destructor References | William M. Miller | 2006-02-27 | 2006-02 | | Core |
| N1939 | 06-0009 | Any Library Proposal for TR2 | B. Dawes, K. Henney | 2006-02-24 | 2006-02 | | Library |

## TR1 Summary

- Specifies 14 new kinds of library functionality.
- Functionality is available from Boost and others.
- tr1::shared_ptr performs reference-counting, supports custom deleters, and cooperates with tr1::weak_ptr.
- Unordered containers are based on hash tables.
- Regular expressions support many syntaxes and string types, search/replace functionality, and iteration over matches.
- tr1::tuple is a generalization of std::pair.
- tr1::array is an STLified C array.
- tr1::function is a generalization of function pointers.
- tr1::bind is a generalization of std::bind1st and std::bind2nd.
- Work is underway on TR2.





## A Selective Survey of Boost Beyond TR1

## Boost Smart Pointers Library

- 2 TR1 smart pointers: shared_ptr, weak_ptr
  - Both support incomplete types.
- Boost offers 4 other resource-managing smart pointers:
  - shared_array: RC pointer to an array.
  - scoped_ptr: Like std::auto_ptr, but not copyable.
  - scoped_array: Like scoped_ptr, but for an array.
  - intrusive_ptr: RC pointer to an object where an RC is already available.
  - Only intrusive_ptr supports incomplete types.
- A headers-only library: *PtrName*.hpp, e.g., shared_ptr.hpp.
  - Or use smart_ptr.hpp to get them all.





## scoped_ptr and scoped_array

```
class Widget { … };

Widget* makeWidgetObject() { return new Widget; }      // factory
Widget* makeWidgetArray() { return new Widget[n]; }    // functions

{
  boost::scoped_ptr<Widget> pw(makeWidgetObject());
  boost::scoped_array<Widget> pa(makeWidgetArray());

  …                                    // neither pw nor pa is copied

}                                      // Widget array is deleted,
                                       // then Widget object
```

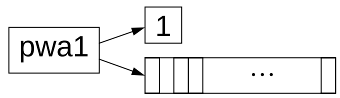
Unlike tr1:: (or boost::) shared_ptr, neither supports custom deleters.

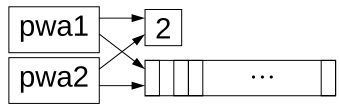
- scoped_ptr uses delete, scoped_array uses delete[].

Generally, prefer scoped_ptr<vector<T> > to a scoped_array<T>.

## shared_array

```
void doSomething()
{
  ...
  boost::shared_array<Widget>
    pwa1(makeWidgetArray());
  ...                                    // use pwa1
  boost::shared_array<Widget>
    pwa2(pwa1);
  ...
  pwa1 = pwa2;                           // nothing changes
}                                        // Widget array is deleted
```




Differences from tr1:: (or boost::) shared_ptr:

- No operator*, operator->, or pointer type conversions.
- No custom deleters; delete[] is always used.

Generally, prefer shared_ptr<vector<T> > to a shared_array<T>.





## intrusive_ptr

Clients define RC functions that intrusive_ptr calls:

```
void intrusive_ptr_add_ref(T *pObj);     // increment RC for *pObj
void intrusive_ptr_release(T *pObj);     // decrement RC;
                                         // if (RC == 0) release *pObj
```

Use of intrusive_ptr itself is straightforward:

```
{
  boost::intrusive_ptr<Widget> p1(new Widget);   // call …_add_ref
  boost::intrusive_ptr<Widget> p2(p1);           // call …_add_ref
   …
  p1->doThis();                        // use p1 and p2
  if (p2) p2->doThat();                // like normal ptrs
  …
  p2 = 0;                              // make p2 null;
  …                                    // call …_release
}                                      // p2 and p1 destroyed;
                                       // call …_release (for p1)
```

## intrusive_ptr

With MT, RC manipulation typically requires synchronization.

- This may need to be programmed manually, e.g.,

```
void intrusive_ptr_add_ref(Widget *pw)
{
  lock *pw's RC;                    // often in ctor of local obj
  increment *pw's RC;
  release lock;                     // often in dtor of local obj
}

void intrusive_ptr_release(Widget *pw)
{
  lock *pw's RC;
  decrement *pw's RC;
  if (RC == 0) release *pw;
  release lock;
}
```

Scott Meyers, Software Development Consultant
http://www.aristeia.com/



Page 87

## intrusive_ptr

Consider two hierarchies with different RC implementations:

- Widget has AddRef/Release; deletion is automatic when RC == 0.
  - ➨ Think COM…
- Gadget has incRC/decRC; clients delete when RC == 0.
- Assume AddRef/Release/incRC/decRC are MT-safe in an MT environment.

```
void intrusive_ptr_add_ref(Widget *pw)
{ pw->AddRef(); }

void intrusive_ptr_release(Widget *pw)
{ pw->Release(); }

void intrusive_ptr_add_ref(Gadget *pg)
{ pg->incRC(); }

void intrusive_ptr_release(Gadget *pg)
{ if (pg->decRC() == 0) delete pg; }
```

Scott Meyers, Software Development Consultant
http://www.aristeia.com/



Page 88

## Boost Smart Pointer Comparison

| | shared_ptr | shared_array | scoped_ptr | scoped_array | weak_ptr | intrusive_ptr |
|---|---|---|---|---|---|---|
| Bytes (32 bit system) | 8 | 8 | 4 | 4 | 8 | 4 |
| Supports incomplete types? | **Y** | N | N | N | **Y** | **Y** |
| Supports Derived ⇒ Base conversions? | **Y** | N | N | N | N | **Y** |
| Supports ->/*? | **Y** | N | **Y** | N | N | **Y** |
| Supports [ ]? | N | **Y** | N | **Y** | N | N |
| Allows copying? | **Y** | **Y** | N | N | **Y** | **Y** |
| Usable as bool? | **Y** | **Y** | **Y** | **Y** | N | **Y** |
| Supports *_pointer_casts? | **Y** | N | N | N | N | **Y** |
| Allocates mem? | **Y** | **Y** | N | N | N | N |
| Custom deleters? | **Y** | N | N | N | NA | NA |
| Create vtbl? | **Y** | N | N | N | N | N |
| Thread-safe RC manipulation? | In MT programs | In MT programs | NA | NA | In MT programs | NA |





## Boost Smart Pointers Summary

- Non-TR1 smart pointers are shared_array, scoped_ptr, scoped_array, and intrusive_ptr.
- Little use for scoped_array and shared_array:
  - ➡ scoped_ptr<vector<T> > often preferable to scoped_array<T>.
  - ➡ shared_ptr<vector<T> > often preferable to shared_array<T>.
- intrusive_ptr supports incomplete types, but requires manual synchronization in threaded environments.
- Different smart pointers have different pros and cons.

## Aside: Boost Pointer Container Library

An alternative to containers of raw or smart pointers are *pointer containers*:

- Containers of owned pointers.
  - Contained pointers deleted when the container is destructed.
- Boost.PointerContainer offers:
  - ptr_vector, ptr_deque, ptr_list, ptr_array.
  - ptr_set, ptr_multiset, ptr_map, ptr_multimap.

Advantages:

- Less error-prone than containers of raw pointers.
- Smaller/faster than containers of tr1::shared_ptrs.

Consult Boost.PointerContainer documentation for more info.





## Boost Static Assert Library

- Defines a single macro: BOOST_STATIC_ASSERT(*condition*)
  - If condition is false, compilation fails:

```
BOOST_STATIC_ASSERT(sizeof(long) == sizeof(void*));
```

- A header-only library: static_assert.hpp.
  - Type traits – useful in static assertions – are in type_traits/*.hpp.
- Incurs no runtime penalty: no code, no data.
  - Inserts a typedef into the current scope.
    - Typedef name incorporates the __LINE__ macro.
- Previews funtionality to be in C++0x.
  - The syntax will be static_assert(*condition*, "*message*");
    - If condition is false, a diagnostic with *message* will be issued.

```
static_assert(sizeof(long) == sizeof(void*),
              "sizeof(long) != sizeof(void*) on this platform");
```

## Example Static Assertions

BOOST_STATIC_ASSERT valid at namespace/function/class scope:

```
// posible namespace scope assertion: pointers must have 64 bits
BOOST_STATIC_ASSERT(CHAR_BIT * sizeof(void*) == 64);

// possible function (template) scope assertion:
//    parameters must be iterators pointing to PODs
template<typename IterT>
void algForPODSOnly(IterT begin, IterT end)
{
  typedef typename std::iterator_traits<IterT>::value_type value_type;
  BOOST_STATIC_ASSERT(boost::is_pod<value_type>::value);
  …
}
```

- Aside: is_pod is also part of TR1.





## Example Static Assertions

```
// possible class (template) scope assertion:  T must be in Widget hierarchy
template<typename T>
class ClassForWidgetsOnly {
  BOOST_STATIC_ASSERT((boost::is_base_of<Widget, T>::value));
  …
};
```

Aside: is_base_of is also part of TR1, but with different semantics:

- std::tr1::is_base_of<T, T> is true.
- boost::is_base_of<T, T> is false.
  - ➡ This will change in the next Boost release (beyond 1.33.1).

To get TR1 semantics in Boost now:

```
template<typename T>
class ClassForWidgetsOnly {
  BOOST_STATIC_ASSERT((boost::is_same<T, Widget>::value ||
                       boost::is_base_of<Widget, T>::value));
  …
};
```

## Static Assertions in Boost.MPL

Boost's Metaprogramming Library (MPL) also supports static assertions:

- Resulting diagnostic messages are often clearer.
- Diagnostic messages may be customized.
  - ➡ Useful for library authors communicating with library users.
- There are syntactic quirks compared to BOOST_STATIC_ASSERT:

```
BOOST_MPL_ASSERT_RELATION(sizeof(long), ==, sizeof(void*));
BOOST_MPL_ASSERT((boost::is_pod<value_type>::value));
```

Try them both, see which you prefer.

- ➡ Header for MPL assertions is mpl/assert.hpp.





## Boost Static Assertions Summary

- BOOST_STATIC_ASSERT supports compile-time assertions at namespace, class, and function scope.
- Static assertions often use TR1's type traits.
- MPL assertions yield better diagnostics but have syntactic quirks.
- static_assert will be part of C++0x.

## Boost Lambda Library

- Generates anonymous function objects on the fly.
  - ➡ The general term for such functions is *lambda functions*.

```
using namespace boost::lambda;

std::vector<int> vi;
int low, high;
…
std::vector<int>::iterator firstWithinLimits =
  std::find_if(vi.begin(), vi.end(), low <= _1 && _1 <= high);

std::for_each(vi.rbegin(), vi.rend(), std::cout << _1 << ' ');

std::for_each(vi.begin(), vi.end(), _1 += low);

std::vector<std::tr1::shared_ptr<std::string> > vps;
…
std::sort(vps.begin(), vps.end(), *_2 < *_1);
```

- A headers-only library: lambda/*.hpp.



## Lambda Syntax Surprises

Lambda gives great demo and is truly useful, but icebergs abound.

- Non-operator function calls must use bind – *Lambda* bind.
  - ➡ Lambda bind ≠ Boost bind (which ≅ TR1's bind)!

```
void f(int x);

std::for_each(vi.begin(), vi.end(), f(_1));            // error!
std::for_each(vi.begin(), vi.end(), bind(f, _1));      // okay
std::for_each(vi.begin(), vi.end(), bind(&f, _1));     // also okay

std::sort(vps.begin(), vps.end(), _2->size() < _1->size());   // error!
std::sort(vps.begin(), vps.end(),
          bind(&std::string::size, *_2) <                     // okay
            bind(&std::string::size, *_1));
```

## Lambda Syntax Surprises

Binding non-const member functions needs more syntax.

- By default, objects inside bound objects are stored by ref-to-*const*.

```
class Widget {
public:
  …
  void rotate(int degrees);
};

std::list<Widget> lw;
…
std::for_each(lw.begin(), lw.end(),
              bind(&Widget::rotate, _1, 90));         // error!

std::for_each(lw.begin(), lw.end(),
              bind(&Widget::rotate, var(_1), 90));    // okay
```





## Lambda Syntax Surprises

Binding isn't the only place where surprises crop up:

```
std::for_each(
  vi.rbegin(), vi.rend(),
  std::cout << "value is " << _1 << '\n'              // prints "value is"
);                                                    // only once

std::for_each(
  vi.rbegin(), vi.rend(),
  std::cout << constant("value is ") << _1 << '\n'    // prints "value is"
);                                                    // once per element
```

## More Lambda Syntax

Lambda offers replacements for many C++ features, e.g.:

- new and delete expressions (new_ptr, delete_ptr).
- Conditionals (if_then, while_loop, switch_statement, etc.)
- Exceptions (try_catch, catch_exception, throw_exception)

Example (adapted from the Lambda documentation):

```
std::for_each(
  vi.begin(), vi.end(),
  try_catch(
    bind(f, _1),                                   // f may throw
    catch_exception<f_exception>(
      std::cout << constant("Exception: argument = ") << _1
    ),
    catch_all(
      ( std::cout << constant("Unknown"), rethrow() )
    )
  )
);
```





## Lambda Functions in C++0x

C++0x is likely to offer native support for Lambda functions.

- As a language feature, not a library.
  - ➡ The result should have fewer “gotchas” than Boost’s library.

## Boost Lambda Summary

- Boost.Lambda allows in-place creation of function object types.
- Syntax is natural for operator-based expressions, less so for function calls, etc.
- C++0x will probably offer native support for lambda functions.





## Boost File System Library

- A portable way to work with paths, files, and directories.
- A library that requires building.
  - Headers are filesystem/*.hpp.
- Example:

```
void printFilesInDirTree(const std::string& dirName)
{
  using namespace std;
  namespace fs = boost::filesystem;

  fs::path dir(dirName, fs::native);              // name is in OS fmt
  if (!fs::exists(dir) || !fs::is_directory(dir)) {
    // have error message give full native pathname
    cerr << system_complete(dir).native_directory_string()
         << ": No such directory\n.";
    return;
  }
```

*Continued on next slide…*

## File System Example

```
  typedef vector<fs::path> PathVec;              // to save dirs for
  PathVec dirs;                                  // later traversal

  fs::directory_iterator di(dir);
  fs::directory_iterator end;
  while (di != end) {
    if (fs::is_directory(*di)) dirs.push_back(*di);  // save for later
    else if (!fs::symbolic_link_exists(*di)) {       // ignore symlinks
      cout << di->native_file_string() << '\n';      // print filename in
    }                                                // native format
    ++di;
  }

  // show subdirectories
  for(PathVec::const_iterator i = dirs.begin(); i != dirs.end(); ++i)
    printFilesInDirTree(i->native_directory_string());
}
```





## Once More – With Lambda!

```
  typedef vector<fs::path> PathVec;        // to save dirs for
  PathVec dirs;                            // later traversal

  for_each(
    fs::directory_iterator(dir), fs::directory_iterator(),
    if_then_else(bind(&fs::is_directory, _1),
                 bind(&PathVec::push_back, var(dirs), _1),
                 if_then(!bind(&fs::symbolic_link_exists, _1),
                         cout << bind(&fs::path::native_file_string, _1)
                              << '\n')));

  // show subdirectories
  for_each(dirs.begin(), dirs.end(),
           bind(&printFilesInDirTree,
                bind(&fs::path::native_directory_string, _1)));
}
```

## File System

- Other functionality:
  - Creating, removing, renaming, copying files/directories.
  - Checking portability of file/directory names.
  - std::fstreams taking boost::filesystem::path objects as names (instead of const char*s).
- Race conditions in file systems are common:

```
assert(exists(path) == exists(path));      // typically may fail
                                           // due to concurrent
                                           // directory manipulations
```

  - This complicates directory iteration.
  - Code using Boost.FileSystem must be prepared for exceptions.





## Boost File System Summary

- Boost.FileSystem offers a platform-independent file system API.
- Supports files, directories, symbolic links, and iteration over directory contents.

## Boost Conversions Libraries

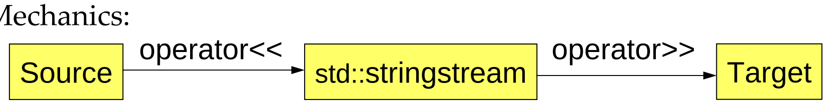
- Special "casts" (really function templates):
  - lexical_cast: Converts objects to/from textual representations
  - numeric_cast: Throws if the target can't represent the source value.
  - polymorphic_cast: Like dynamic_cast for pointers, but throws if conversion fails.
  - polymorphic_downcast: In debug builds, dynamic_cast with assert that result is non-null; in release builds, just a static_cast.
- Header-only libraries:
  - cast.hpp:
    - polymorphic_cast and polymorphic_downcast.
  - lexical_cast.hpp:
    - lexical_cast.
  - numeric/conversion/cast.hpp:
    - numeric_cast.





## lexical_cast

Mechanics:




- Source types must support operator<< for ostreams.
- Target types must support operator>> for istreams.

Example conversions to text:

```
using boost::lexical_cast;

int i = 46;
std::string s1(lexical_cast<std::string>(i));        // s1 == "46"

float f = 3.14159;
std::string s2(lexical_cast<std::string>(f));        // s2 == "3.14159"

Widget w;
std::string s3(lexical_cast<std::string>(w));        // s3 == streamed
                                                     // representation
                                                     // of w
```

## lexical_cast

Example conversions from text:

```
using boost::lexical_cast;

std::string s1("46");
int i = lexical_cast<int>(s1);                    // i == 46

std::string s2("3.14159");
float f = lexical_cast<float>(s2);                // f == 3.14159

std::string s3( … );                              // assume s3 has textual
                                                  // rep for a Widget

Widget w(lexical_cast<Widget>(s3));               // w == value of Widget
                                                  // stored in s3
```





## lexical_cast

- Conversion to/from string/wstring is most common, but all types supporting operator<</operator>> work.
- Cost caveat: std::stringstream/wstringstream allocates/deallocates memory.
- Failure during stringstream writing/reading ⇒ exception.
- Whitespace around non-string values should be trimmed.
  - The Boost String Algorithms library makes this easy:

```
std::string s1("27");
int i1 = lexical_cast<int>(s1);             // fine

std::string s2("9 ");
int i2a = lexical_cast<int>(s2);            // throws
boost::trim(s2);                            // from String Algo. Lib.
int i2b = lexical_cast<int>(s2);            // fine

std::string s3(" 8.2");
float fa = lexical_cast<float>(s3);         // throws
boost::trim(s3);
float fb = lexical_cast<float>(s3);         // fine
```

## numeric_cast

- Casts to integral types. Throws if the value can't be preserved:

```
using boost::numeric_cast;
void f(long l, double d, unsigned u)
{
  int i1 = numeric_cast<int>(l);      // all throw if the source
  int i2 = numeric_cast<int>(d);      // value won't fit in an int
  int i3 = numeric_cast<int>(u);
  …
}
```

- Can be configured to support user-defined types:

```
class Int_24 { … };                  // 24-bit int; must specialize
                                     // std::numeric_limits, etc.
int i;
…
Int_24 myInt = numeric_cast<Int_24>(i);    // throws if i's value
                                           // won't fit in myInt
```

- Handles *conversions* only, not arithmetic overflow/underflow:

```
int i = numeric_cast<int>(INT_MAX * 2);    // has no effect
```





## polymorphic_cast and polymorphic_downcast

polymorphic_cast:

- Applicable only to pointers.
- Throws if the cast fails (like dynamic_cast with references).
  - Avoids "forgot to check for null" bugs.

polymorphic_downcast:

- dynamic_cast + assert(*result* != 0) in debug builds, static_cast in production builds.
  - Good when runtime speed is paramount.
    - And your testing is thorough :-)
  - Useful when RTTI is disabled in release mode.
    - And your testing is thorough :-)
- Valid only where static_cast is:
  - No cross casting in MI hierarchies.
  - No downcasting from virtual bases.

## Boost Conversions Summary

- lexical_cast converts objects to/from textual representations
- numeric_cast ensures that the target can hold the source value.
- polymorphic_cast does dynamic_cast on a pointer, throwing if conversion fails.
- polymorphic_downcast is static_cast in release builds, dynamic_cast asserting a non-null result in debug builds.




## Boost Format Library

- Type-safe printf-like formatting with support for UDTs:

  *stream* << format(*formatString*) % $arg_1$ % $arg_2$ % … % $arg_n$;
- Examples:

```
using namespace std;
using boost::format;

printf(            "Name, age = %s, %d\n",     name,   age);
cout << format( "Name, age = %s, %d\n") % name % age;

printf(            "Name, age = %-20.18s %3d\n",    name,   age);
cout << format( "Name, age = %-20.18s %3d\n") % name % age;

printf(            "%10d%% %4.2f %E %+08i\n",    i,   f,   d,   s);
cout << format( "%10d%% %4.2f %E %+08i\n") % i % f % d % s;
```

- A header-only library: format.hpp.
  - ➡ Declarations only are in format_fwd.hpp.

## Format

- Format strings largely – not completely – compatible with printf.
  - E.g.,"%05.3d" displays more leading 0s using format.
  - E.g., * in format strings (e.g., "%*d") is unsupported.
- Supports more options than standard C's printf. Examples:
  - Positional argument references, e.g., "%2$d" ⇒ arg 2 as int.
  - Omitted type specifiers, e.g., "%|5|" ⇒ width ≥ 5 for this arg.
  - Positional refs w/o type specs, e.g., "%3%" ⇒ third arg.
  - New alignment specifiers, e.g., "%=11g" ⇒ center value in field.

```
cout << format("%|2$=8| %1%")     // print age centered in field
          % name                  // of width ≥ 8, then name
          % age;
```





## Format

- Supports UDTs:

```
class Widget { … };
std::ostream& operator<<(std::ostream&, const Widget&);
…
Widget w;
…
std::cout << format("Widget value = %1%") % w;
```

- Slower than direct uses of operator<< and printf:
  - Everything's converted to a std::string for output.
  - Reordering support may require multiple allocations.
  - Documentation suggests 5x slower than printf not surprising.

## Boost Format Summary

- Offers type-safe printf-like formatting.
- Supports both built-in and user-defined types.
- Behavior mostly printf-compatible, but some differences exist.
- Offers some formatting capabilities not available with printf.
- Incurs a nontrivial performance penalty.





## Boost Variant Library

- Discriminated unions supporting arbitrary types:
  - Real unions don't know what type they hold.
  - Real unions support only PODs.
- A header-only library: variant.hpp.
  - Functionality subsets are available via more specific headers.
- Type-safe:
  - No "type punning:" only the type stored can be read.
  - Never empty: a valid object is always held.
- "Smart" about initialization and assignment:

```
using boost::variant;

variant<int, std::string> v(27);   // holds an int with value 27

v = "Boost";                       // holds the string "Boost"; note
                                   // const char* ⇒ string conversion

v = 2.13;                          // destructs string, holds int with
                                   // value 2; note double ⇒ int conv.
```

## Variant

- Uses heap memory only for heterogeneous assignments:

```
variant<Widget, Gadget> v1(Widget());       // v1 holds a Widget
variant<Widget, Gadget> v2(Gadget());       // v2 holds a Gadget
v1 = v2;                                    // destroy v1's Widget, copy
                                            // construct v2's Gadget into v1
```

  - ➡ Before assignment, v1's value copied to heap.
    - Assignment succeeds ⇒ heap memory released.
    - Assignment throws ⇒ v1's value unchanged.
    - If known that copy ction can't throw ⇒ no buffer allocated.
- Supports output streaming when all contained types do:

```
std::cout << v1 << v2;                      // fine as long as Widget and
                                            // Gadget support operator<<
```





## get and variants

get is dynamic_cast for variants (with a different syntax):

```
typedef variant<Widget, int, std::string> WiSVariant;
WiSVariant v;
…
int *pi = get<int>(&v);      // ptr form: returns null if v doesn't hold int
int& i = get<int>(v);        // ref form: throws if v doesn't hold int
```

This makes type discovery possible (if you know all possible types):

```
void workOnVariant(WiSVariant& v)
{
  if (Widget *pw = get<Widget>(&v)) {          // v holds a Widget?
    use pw…
  } else if (int *pi = get<int>(&v)) {         // an int?
    use pi…
  } else {
    std::string& s = get<std::string>(v);      // must be a string...
    use s…
  }
}
```

## Visitation Support

Cascading type tests are gross:

- Tricky: derived types must be tested before bases.
- Must be manually updated when the list of types changes.
- If a possible type isn't checked for, there's no warning.





## Visitation Support

A better approach: Create a visitor…

```
class WiSVisitor: public static_visitor<void> {
public:
  void operator()(Widget& w)                          // visitors have one
  { process a WiSVariant holding a Widget; }          // op() per type

  void operator()(int i)
  { process a WiSVariant holding an int; }

  void operator()(std::string& s)
  { process a WiSVariant holding a string; }
};
```

…and apply it to the variant:

```
WiSVisitor visitor;

apply_visitor(visitor, v);            // invoke WiSVisitor::operator()
                                      // on whatever v holds
```

- apply_visitor compiles only if visitor(T) is valid for each T in v.

## Visitation Support

Templatize operator() to create generic visitors:

```
struct ContainedSize: static_visitor<std::size_t> {
  template<typename T>
  std::size_t operator()(const T&) const { return sizeof(T); }
};

template<typename Var>
std::size_t contained_size(const Var& v)
{
  return boost::apply_visitor(ContainedSize(), v);
}
...
std::cout << contained_size(v);            // print sizeof(T) for
                                           // whatever T v holds
```

Boost.Variant also supports binary visitation:

apply_visitor(*BinaryVisitor*, *variant*, *variant*);





## Boost Variant Summary

- Boost.Variant offers type-safe discriminated unions that can hold any type.
- get offers dynamic_cast functionality for variants.
- Visitation support makes it possible to avoid cascading get tests.

## Boost Summary

- Non-TR1 smart pointers are intrusive_ptr, scoped_ptr, scoped_array, and shared_array.
  - Pointer containers are also available.
- BOOST_STATIC_ASSERT and MPL assertions support compile-time assertions.
- Conversions are lexical_cast, numeric_cast, polymorphic_cast, and polymorphic_downcast.
- Lambda lets you create function object types on the fly.
- File System allows portable manipulation of files, directories, etc.
- Format provides type-safe, extensible, printf-like formatting.
- Variant offers type-safe discriminated unions.





## Further Reading

*Many cited print publications are also available online.*

**TR1 Itself:**

- *Draft Technical Report on C++ Library Extensions*, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1836.pdf.
  - Ignore “Draft” – this is final.
- *Library Extension Technical Report -- Issues List, Revision 10*, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1837.pdf.
  - 144 pages of issues raised about TR1!
    - Most have already been resolved.

# Further Reading

**Overviews of TR1:**

- *The C++ Standard Library Extensions*, Pete Becker, Addison-Wesley, 2007, ISBN 0-321-41299-0.
  - A comprehensive reference for TR1.
- "The Technical Report on C++ Library Extensions," Matthew H. Austern, *Dr. Dobb's Journal*, June 2005.
- "The Technical Report on C++ Library Extensions (TR1)," Matt Austern, November 2005, http://www.devconnections.com/updates/LasVegasFall%5F05/CPP%5FConnections/CLT304.pdf.
  - Slides from a conference presentation.
- "The New C++ Not-So-Standard Library," Pete Becker, *C/C++ Users Journal*, June 2005.
- *Scott Meyers' TR1 Information Page*, http://www.aristeia.com/EC3E/TR1_info.html.
  - Includes links to proposal documents.





# Further Reading

**Smart Pointers (shared_ptr and weak_ptr):**

- "Bad Pointers," Pete Becker, *C/C++ Users Journal*, September 2005.
- "More Bad Pointers," Pete Becker, *C/C++ Users Journal*, October 2005.
- "Weak Pointers," Pete Becker, *C/C++ Users Journal*, November 2005.
- *Effective C++, Third Edition*, Scott Meyers, Addison-Wesley, 2005.
  - Describes tr1::shared_ptr and uses it throughout the book.
  - The TOC is attached.
- *Smart Pointer Timings*, http://www.boost.org/libs/smart_ptr/smarttests.htm.
  - Shows comparative performance of 5 possible implementations.

## Further Reading

**Pointer containers:**

- *Boost.PointerContainer Documentation*, Thorsten Ottosen, http://www.boost.org/libs/ptr_container/.
- "Pointer Containers," Thorsten Ottosen, *Dr. Dobbs Journal*, October 2005.





## Further Reading

**Hash Tables (unordered_*):**

- "STL and TR1: Part III," Pete Becker, *C/C++ Users Journal*, February 2006.
- "Hash Tables for the Standard Library," Matt Austern, *C/C++ Users Journal Experts Forum*, April 2002, http://www.cuj.com/documents/s=7984/cujcexp2004austern/.
- *Effective STL*, Scott Meyers, Addison-Wesley, 2001.
  - ➡ Item 25 is devoted to the pre-TR1 hash_* containers.
  - ➡ The TOC is attached.

## Further Reading

**Regular Expressions (regex):**

- *Boost.RegEx Documentation*, http://www.boost.org/libs/regex/doc/index.html.
  - Describes Boost's – not TR1's – RE library, but the two are similar.
- "Regular Expressions," Pete Becker, *Dr. Dobb's Journal*, May 2006.

**Tuples:**

- "The Header <tuple>," Pete Becker, *C/C++ Users Journal*, July 2005.

**Fixed-Size Arrays (array):**

- "STL and TR1: Part II," Pete Becker, *C/C++ Users Journal*, January 2006.





## Further Reading

**Generalized Function Pointers (function):**

- "Generalized Function Pointers," Herb Sutter, *C/C++ Users Journal*, August 2003.
- "Generalizing Observer," Herb Sutter, *C/C++ Users Journal Experts Forum*, September 2003.
- *Effective C++, Third Edition*, Scott Meyers, Addison-Wesley, 2005.
  - Item 35 explains and demonstrates use of tr1::function.
  - The TOC is attached.

**Reference Wrappers (ref):**

- "Reference Wrapper," Danny Kalev, http://www.informit.com/guides/content.asp?g=cplusplus&seqNum=217.

## Further Reading

**TR2:**

- *Library Extension TR2 Call for Proposals*, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1810.html.
- *List of C++ Standards Committee Papers*, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/.
  - Useful for browsing for library proposals.





## Further Reading

**Boost:**

- *The Boost Web Site*, boost.org.
  - Note pubs list at http://www.boost.org/more/bibliography.html.
- *Beyond the C++ Standard Library: An Introduction to Boost*, Björn Karlsson, Addison-Wesley, 2006, ISBN 0-321-13354-4.
  - Covers Boost versions of 5 TR1 components + 4 libs in this talk.
- *C++ Cookbook*, D. Ryan Stephens *et al.*, O'Reilly, 2005, ISBN 0-596-00761-2.
  - Includes sample uses of several Boost libraries.
- "C++ Type traits," John Maddock and Steve Cleary, *Dr. Dobb's Journal*, October 2000.
- *The Boost Graph Library*, Jeremy Siek *et al.*, Addison-Wesley, 2002, ISBN 0-201-72914-8.
- *C++ Template Metaprogramming*, David Abrahams and Aleksey Gurtovoy, Addison-Wesley, 2004, ISBN 0-321-22725-5.

# Further Reading

**static_assert in C++0x:**

- *Proposal to Add Static Assertions to the Core Language (Revision 3)*, Robert Klarer *et al.*, October 20, 2004, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1720.html.

**Lambda functions in C++0x:**

- *A proposal to add lambda functions to the C++ standard*, Valentin Samko, February 23, 2006, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1958.pdf.
- *Lambda expressions and closures for C++*, Jeremiah Willcock *et al.*, February 26, 2006, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1968.pdf.





# Further Reading

**Arithmetic underflow/overflow:**

- *Integer Handling with the C++ SafeInt Class*, David LeBlanc, January 7, 2004, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/secure01142004.asp.
- *Another Look at the SafeInt Class*, David LeBlanc, May 2, 2005, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/secure05052005.asp.

## Please Note

Scott Meyers offers consulting services in all aspects of the design and implementation of C++ software systems. For details, visit:

http://www.aristeia.com/

Scott also offers a mailing list to keep you up to date on his professional activities. Read about the mailing list at:

http://www.aristeia.com/MailingList/





## Contents



Reprinted from *Effective C++, Third Edition: 55 Specific Ways to Improve Your Programs and Designs*, by Scott Meyers, Addison-Wesley Publishing Company, 2005.

Reprinted from *Effective C++, Third Edition: 55 Specific Ways to Improve Your Programs and Designs*, by Scott Meyers, Addison-Wesley Publishing Company, 2005.

## Contents



Reprinted from *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*, by Scott Meyers, Addison-Wesley Publishing Company, 2001.

Reprinted from *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*, by Scott Meyers, Addison-Wesley Publishing Company, 2001.