Boost

Akim Demaille akim@lrde.epita.fr

June, 1st 2015

(2016-11-16 09:53:54 +0100 121bea3)

boost C++ LIBRARIES

Programming in C11

Boost

Boost

Boost

> *[...]one of the most highly regarded and expertly designed C++ library projects in the world.*
>
> — [Sutter and Alexandrescu, 2005]

> *Item 55: Familiarize yourself with Boost.*
>
> — [Meyers, 2005]

> *The obvious solution for most programmers is to use a library that provides an elegant and efficient platform independent to needed services. Examples are Boost*[...]
>
> — [Stroustrup, 2003]

A Meta Library

- Algorithms
- Broken Compiler Workarounds
- Concurrent Programming
- Containers
- Correctness and Testing
- Data Structures
- Domain Specific
- Function Objects and Higher-order Programming
- Generic Programming
- Image Processing
- Input/Output
- Inter-language Support
- Iterators
- Language Features Emulation
- Math and Numerics
- Memory
- Parsing
- Patterns and Idioms
- Preprocessor Metaprogramming
- Programming Interfaces
- State Machines
- String and Text Processing
- System
- Template Metaprogramming
- Miscellaneous

Boost 1.58.0: 131 libraries

Accumulators, Algorithm, Align, Any, Array, Asio, Assert, Assign, Atomic, Bimap, Bind, Call Traits, Chrono, Circular Buffer, Compatibility, Compressed Pair, Concept Check, Config, Container, Context, Conversion, Core, Coroutine, CRC, Date Time, Dynamic Bitset, Enable If, Endian, Exception, Filesystem, Flyweight, Foreach, Format, Function, Function Types, Functional, Functional/Factory, Functional/Forward, Functional/Hash, Functional/Overloaded Function, Fusion, Geometry, GIL, Graph, Heap, ICL, Identity Type, In Place Factory, Typed In Place Factory, Integer, Interprocess, Interval, Intrusive, IO State Savers, Iostreams, Iterator, Lambda, Lexical Cast, Local Function, Locale, Lockfree, Log, Math, Math Common Factor, Math Octonion, Math Quaternion, Math/Special Functions, Math/Statistical Distributions, Member Function, Meta State Machine, Min-Max, Move, MPI, MPL, Multi-Array, Multi-Index, Multiprecision, Numeric Conversion, Odeint, Operators, Optional, Parameter, Phoenix, Pointer Container, Polygon, Pool, Predef, Preprocessor, Program Options, Property Map, Property Tree, Proto, Python, Random, Range, Ratio, Rational, Ref, Regex, Result Of, Scope Exit, Serialization, Signals2, Smart Ptr, Sort, Spirit, Statechart, Static Assert, String Algo, Swap, System, Test, Thread, ThrowException, Timer, Tokenizer, Tribool, TTI, Tuple, Type Erasure, Type Index, Type Traits, Typeof, uBLAS, Units, Unordered, Utility, Uuid, Value Initialized, Variant, Wave, Xpressive

Boost 1.58.0: 131 libraries

Accumulators, Algorithm, Align, Any, Array, Asio, Assert, Assign, Atomic, Bimap, Bind, Call Traits, Chrono, Circular Buffer, Compatibility, Compressed Pair, Concept Check, Config, Container, Context, Conversion, Core, Coroutine, CRC, Date Time, Dynamic Bitset, Enable If, Endian, Exception, Filesystem, Flyweight, Foreach, Format, Function, Function Types, Functional, Functional/Factory, Functional/Forward, Functional/Hash, Functional/Overloaded Function, Fusion, Geometry, GIL, Graph, Heap, ICL, Identity Type, In Place Factory, Typed In Place Factory, Integer, Interprocess, Interval, Intrusive, IO State Savers, Iostreams, Iterator, Lambda, Lexical Cast, Local Function, Locale, Lockfree, Log, Math, Math Common Factor, Math Octonion, Math Quaternion, Math/Special Functions, Math/Statistical Distributions, Member Function, Meta State Machine, Min-Max, Move, MPI, MPL, Multi-Array, Multi-Index, Multiprecision, Numeric Conversion, Odeint, Operators, Optional, Parameter, Phoenix, Pointer Container, Polygon, Pool, Predef, Preprocessor, Program Options, Property Map, Property Tree, Proto, Python, Random, Range, Ratio, Rational, Ref, Regex, Result Of, Scope Exit, Serialization, Signals2, Smart Ptr, Sort, Spirit, Statechart, Static Assert, String Algo, Swap, System, Test, Thread, ThrowException, Timer, Tokenizer, Tribool, TTI, Tuple, Type Erasure, Type Index, Type Traits, Typeof, uBLAS, Units, Unordered, Utility, Uuid, Value Initialized, Variant, Wave, Xpressive

Navigating in Boost

- Home page
  `http://www.boost.org`
- Library list
  `http://www.boost.org/doc/libs/`
- Special announces
  `http://www.boost.org/users/news/`
- Finally moving to Git!
  `https://github.com/boost-lib`
- Papers about Boost Components
  `http://www.boost.org/users/bibliography.html`

Boost Software License - Version 1.0 - August 17th, 2003

Permission is hereby granted, free of charge, to any person or organization obtaining a copy of the software and accompanying documentation covered by this license (the "Software") to use, reproduce, display, distribute, execute, and transmit the Software, and to prepare derivative works of the Software, and to permit third-parties to whom the Software is furnished to do so, all subject to the following:

The copyright notices in the Software and this entire statement, including the above license grant, this restriction and the following disclaimer, must be included in all copies of the Software, in whole or in part, and all derivative works of the Software, unless such copies or derivative works are solely in the form of machine-executable object code generated by a source language processor.

The software is provided “as is”, without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose, title and non-infringement. In no event shall the copyright holders or anyone distributing the software be liable for any damages or other liability, whether in contract, tort or otherwise, arising from, out of or in connection with the software or the use or other dealings in the software.

Boost Software License - Version 1.0 - August 17th, 2003

Permission is hereby granted, free of charge, to any person or organization obtaining a copy of the software and accompanying documentation covered by this license (the "Software") to use, reproduce, display, distribute, execute, and transmit the Software, and to prepare derivative works of the Software, and to permit third-parties to whom the Software is furnished to do so, all subject to the following:

The copyright notices in the Software and this entire statement, including the above license grant, this restriction and the following disclaimer, must be included in all copies of the Software, in whole or in part, and all derivative works of the Software, unless such copies or derivative works are solely in the form of machine-executable object code generated by a source language processor.

The software is provided “as is”, without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose, title and non-infringement. In no event shall the copyright holders or anyone distributing the software be liable for any damages or other liability, whether in contract, tort or otherwise, arising from, out of or in connection with the software or the use or other dealings in the software.

Boost 1.53's Primary Test Compilers

Linux	• GCC: 4.1.2, 4.2.4, 4.4.4, 4.5.3, 4.6.3, 4.7.2 • GCC, C++11 mode: 4.4.4, 4.5.3, 4.6.3, 4.7.2 • Intel: 11.1, 12.1 • LLVM Clang: 2.8 • LLVM Clang, with libc++: 3.2
OS X	• GCC: 4.4.7 • GCC, C++11 mode: 4.4.4 • Intel: 11.1, 12.0
Windows	• Visual C++: 9.0, 10.0
FreeBSD	• GCC: 4.2.1, 32 and 64 bit

Boost 1.53's Additional Test Compilers

Linux
- Cray: 4.6.1
- Clang: from subversion
- LLVM Clang, with libc++: 3.2
- GCC: 4.2.4, 4.4.4, 4.5.3, 4.6.3, 4.7.1
- GCC, C++11 mode: 4.4.4, 4.5.3, 4.6.3, 4.7.1, 4.7.2
- pgCC: 11.9
- Intel: 10.1, 11.1, 12.1
- Intel, C++11 mode: 13.0.1

OS X
- Clang: from subversion
- Clang, C++11 mode: from subversion
- Intel: 11.1, 12.0
- GCC: 4.4.7
- GCC, C++11 mode: 4.4.4

Windows
- Visual C++: 10.0, 11.0
- Visual C++ with STLport: 9.0
- Visual C++, Windows Mobile 5, with STLport: 9.0

AIX
- IBM XL C/C++ Enterprise Edition: V12.1.0.1

Most Libraries are Header-Only

Exceptions:

- Boost.Filesystem
- Boost.GraphParallel
- Boost.IOStreams
- Boost.MPI
- Boost.ProgramOptions
- Boost.Python
- Boost.Regex
- Boost.Serialization
- Boost.Signals
- Boost.System
- Boost.Thread
- Boost.Wave

Programming in NIH

TR1

C++ Technical Report 1 [ISO/IEC, 2006]

> *C++ Technical Report 1 (TR1) is the common name for ISO/IEC TR 19768, C++ Library Extensions, which was a document proposing additions to the C++ standard library for the C++03 language standard. [...] most of its proposals became part of the current official standard, C++ 11. [...] Compilers needed not include the TR1 components to be conforming [...]. However, most of it was available from Boost, and several compiler/library distributors implemented all or part of the components.*
>
> — [WC++_Technical_Report_1]

Boost.TR1

- Implements TR1
- Actually a thin adapter to other Boost Libraries
- Boost header

```
#include <boost/tr1/tuple.hpp>
```

- or standard header

```
#include <tuple>
```

- but standard interface

```
std::tr1::tuple<int, std::string>
  t = std::tr1::make_tuple(10, "hello");
```

General Utilities

Boost.Ref

Boost.Ref

- Pass references where copies are taken
- E.g., `<functional>`

Boost.Ref

```
#include <boost/bind.hpp>
#include <iostream>
#include <vector>
#include <algorithm>

void add(int i, int j, std::ostream& os)
{
  os << i + j << '\n';
}

int main()
{
  std::vector<int> v;
  v.push_back(1);
  v.push_back(3);
  v.push_back(2);

  std::for_each(v.begin(), v.end(),
                boost::bind(add, 10, _1, boost::ref(std::cout)));
}
```

Boost.SmartPointers

Smart Pointers in C++ 98: auto_ptr

```
#include <iostream>
#include <memory>

int main(int argc, char **argv)
{
  int *i = new int;
  std::auto_ptr<int> x(i);
  std::auto_ptr<int> y;

  y = x;

  std::cout << x.get() << '\n'; // 0 (well, NULL).
  std::cout << y.get() << '\n'; // &i.
}
```

- cannot be put in standard containers
- cannot deal with C and arrays (calls `delete`)

Boost.SmartPointers

- RAII for memory
- C++ 98 provides `auto_ptr`
- C++ 11 deprecates `auto_ptr`
- Different types:
  - `scoped_ptr` Simple sole ownership of single objects. Noncopyable.
  - `scoped_array` Simple sole ownership of arrays. Noncopyable.
  - `shared_ptr` Object ownership shared among multiple pointers.
  - `shared_array` Array ownership shared among multiple pointers.
  - `weak_ptr` Non-owning observers of an object owned by `shared_ptr`.
  - `intrusive_ptr` Shared ownership of objects with an embedded reference count.
  - Pointer containers. Syntactic and performance improvements.
- `unique_ptr` C++ 11's improved version of `scoped_ptr` and `scoped_array`.

Boost.SmartPointers

- RAII for memory
- C++ 98 provides `auto_ptr`
- C++ 11 deprecates `auto_ptr`
- Different types:
  - `scoped_ptr` Simple sole ownership of single objects. Noncopyable.
  - `scoped_array` Simple sole ownership of arrays. Noncopyable.
  - `shared_ptr` Object ownership shared among multiple pointers.
  - `shared_array` Array ownership shared among multiple pointers.
  - `weak_ptr` Non-owning observers of an object owned by `shared_ptr`.
  - `intrusive_ptr` Shared ownership of objects with an embedded reference count.
  - Pointer containers. Syntactic and performance improvements.
- `unique_ptr` C++ 11's improved version of `scoped_ptr` and `scoped_array`.

Boost.SmartPointers

- RAII for memory
- C++ 98 provides `auto_ptr`
- C++ 11 deprecates `auto_ptr`
- Different types:
  - `scoped_ptr` Simple sole ownership of single objects. Noncopyable.
  - `scoped_array` Simple sole ownership of arrays. Noncopyable.
  - `shared_ptr` Object ownership shared among multiple pointers.
  - `shared_array` Array ownership shared among multiple pointers.
  - `weak_ptr` Non-owning observers of an object owned by `shared_ptr`.
  - `intrusive_ptr` Shared ownership of objects with an embedded reference count.
  - Pointer containers. Syntactic and performance improvements.
- `unique_ptr` C++ 11's improved version of `scoped_ptr` and `scoped_array`.

Boost.SmartPointers

- RAII for memory
- C++ 98 provides `auto_ptr`
- C++ 11 deprecates `auto_ptr`
- Different types:
  - `scoped_ptr` Simple sole ownership of single objects. Noncopyable.
  - `scoped_array` Simple sole ownership of arrays. Noncopyable.
  - `shared_ptr` Object ownership shared among multiple pointers.
  - `shared_array` Array ownership shared among multiple pointers.
  - `weak_ptr` Non-owning observers of an object owned by `shared_ptr`.
  - `intrusive_ptr` Shared ownership of objects with an embedded reference count.
  - Pointer containers. Syntactic and performance improvements.
- `unique_ptr` C++ 11's improved version of `scoped_ptr` and `scoped_array`.

Boost.SmartPointers

- RAII for memory
- C++ 98 provides `auto_ptr`
- C++ 11 deprecates `auto_ptr`
- Different types:
  - `scoped_ptr` Simple sole ownership of single objects. Noncopyable.
  - `scoped_array` Simple sole ownership of arrays. Noncopyable.
  - `shared_ptr` Object ownership shared among multiple pointers.
  - `shared_array` Array ownership shared among multiple pointers.
  - `weak_ptr` Non-owning observers of an object owned by `shared_ptr`.
  - `intrusive_ptr` Shared ownership of objects with an embedded reference count.
  - Pointer containers. Syntactic and performance improvements.
- `unique_ptr` C++ 11's improved version of `scoped_ptr` and `scoped_array`.

scoped_ptr and scoped_array

```cpp
#include <boost/scoped_ptr.hpp>

int main()
{
  auto i = boost::scoped_ptr<int>{new int};
  *i = 1;
  *i.get() = 2;
  i.reset(new int);
}
```

```cpp
#include <boost/scoped_array.hpp>

int main()
{
  auto i = boost::scoped_array<int>{new int[2]};
  *i.get() = 1;
  i[1] = 2;
  i.reset(new int[3]);
}
```

scoped_ptr vs. unique_ptr

- unique_ptr supports move semantics
- unique_ptr supports customized deleter

```
namespace std
{
  template <class T, class Deleter = default_delete<T>>
  class unique_ptr;

  template <class T, class Deleter>
  class unique_ptr<T[], Deleter>;
}
```

shared_ptr

```cpp
#include <boost/shared_ptr.hpp>

int main()
{
  boost::shared_ptr<int> i1(new int(1));
  boost::shared_ptr<int> i2(i1);
  i1.reset(new int(2));
}
```

```cpp
#include <boost/shared_ptr.hpp>
#include <vector>

int main()
{
  std::vector<boost::shared_ptr<int> > v;
  v.push_back(boost::shared_ptr<int>(new int(1)));
  v.push_back(boost::shared_ptr<int>(new int(2)));
}
```

C++ 11: make_shared: More Than Just Sugar

```
void* operator new(size_t s) {
  auto res = malloc(s);
  std::cerr << "malloc(" << s << ") = " << res << std::endl;
  return res;
}
void operator delete(void* p) {
  std::cerr << "free(" << p << ")" << std::endl;
  free(p);
}
int main() {
  auto sp1 = std::shared_ptr<int>(new int(51));
  auto sp2 = std::make_shared<int>(42);
}
```

```
malloc(4) = 0x7fdce8c000e0
malloc(24) = 0x7fdce8c03ac0
malloc(32) = 0x7fdce8c03ae0
free(0x7fdce8c03ae0)
free(0x7fdce8c000e0)
free(0x7fdce8c03ac0)
```

shared_array

```
#include <boost/shared_array.hpp>
#include <iostream>

int main()
{
  boost::shared_array<int> i1(new int[2]);
  boost::shared_array<int> i2(i1);
  i1[0] = 1;
  std::cout << i2[0] << '\n';
}
```

Strong and Weak pointers

- Strong pointers guarantee their own validity
  - You own the object being pointed at; you create it and destroy it
  - You do not have defined behavior if the object doesn't exist
  - You need to enforce that the object exists
- Weak pointers guarantee *knowing* their own validity
  - You access it, but it's not yours
  - You have defined behavior if the object doesn't exist
  - It never throws an exception.


[2036182]

intrusive_ptr: Efficiency, or Binding to Existing API

```
class RefCounted
{
public:
  friend void intrusive_ptr_add_ref(RefCounted* p) {
    ++p->references;
  }
  friend void intrusive_ptr_release(RefCounted* p) {
    if (--p->references == 0)
      delete p;
  }

  RefCounted() : references(0) {}

private:
  size_t references;
};
```

Pointer Containers

```cpp
#include <boost/shared_ptr.hpp>
#include <vector>

int main()
{
  std::vector<boost::shared_ptr<int> > v;
  v.push_back(boost::shared_ptr<int>(new int(1)));
  v.push_back(boost::shared_ptr<int>(new int(2)));
}
```

```cpp
#include <boost/ptr_container/ptr_vector.hpp>

int main()
{
  boost::ptr_vector<int> v;
  v.push_back(new int(1));
  v.push_back(new int(2));
}
```

Pointer Containers

- Less syntactic clutter
- More efficiency
- But a sole owner: the container
- `boost::ptr_`*`container`*
  `vector`, `deque`, `list`, `set`, `map`, `unordered_set`, `unordered_map`

Boost.Regex

Boost.Regex

```
// \A(\d{3,4})[- ]?(\d{4})[- ]?(\d{4})[- ]?(\d{4})\z.
const boost::regex e
  ("\\A(\\d{3,4})[- ]?(\\d{4})[- ]?(\\d{4})[- ]?(\\d{4})\\z");

std::string machine_readable_card_number(const std::string s)
{
  return regex_replace(s, e, "\\1\\2\\3\\4");
}

std::string human_readable_card_number(const std::string s)
{
  return regex_replace(s, e, "\\1-\\2-\\3-\\4");
}
```

C++ to HTML Pretty-Printer: main

```
int main(int argc, const char** argv)
{
  try
    {
      for (int i = 1; i < argc; ++i)
        process(argv[i]);
    }
  catch (const std::exception& s)
    {
      std::cerr << "exception caught: " << s.what() << std::endl;
      return 1;
    }
  catch (...)
    {
      std::cerr << "unknown exception caught" << std::endl;
      return 1;
    }
  return 0;
}
```

C++ to HTML Pretty-Printer: process

```
void
process(const std::string& fn)
{
  std::cout << "Processing file " << fn << std::endl;
  std::string in = load_file(fn);
  std::string out_name = fn + ".html";
  std::ofstream os(out_name.c_str());

  // strip '<' and '>' first by outputting to a temporary string
  // stream
  std::string t = subst(in,
                        "(<)|(>)|(&)|\\r",
                        "(?1&lt;)(?2&gt;)(?3&amp;)");

  // then output to final output stream adding syntax highlighting:
  os << header_text
     << subst(t, expression_text, format_string)
     << footer_text;
}
```

C++ to HTML Pretty-Printer: load_file

```
std::string
load_file(const std::string& fn)
{
  std::ifstream is(fn.c_str());
  if (is.bad())
    throw std::runtime_error(fn + ": cannot open");
  std::string res;
  res.reserve(is.rdbuf()->in_avail());
  char c;
  while (is.get(c))
    {
      if (res.capacity() == res.size())
        res.reserve(2 * res.capacity());
      res.append(1, c);
    }
  return res;
}
```

C++ to HTML Pretty-Printer: subst

```
std::string
subst(const std::string& in,
      const std::string& pattern, const std::string& replacement)
{
  std::ostringstream os(std::ios::out | std::ios::binary);
  std::ostream_iterator<char> oi(os);
  boost::regex re(pattern);
  boost::regex_replace(oi, in.begin(), in.end(),
                       re, replacement,
                       boost::match_default | boost::format_all);
  return os.str();
}
```

C++ to HTML Pretty-Printer: Main Pattern

```
const char* expression_text =
  // comment: index 2
  "(//[^\\n]*|/\\*.*?\\*/)|"
  // keywords: index 5
  "\\<(asm|auto|bool|break|case|catch|cdecl|char|class|const|const_cast"
  "|continue|default|delete|do|double|dynamic_cast|else|enum|explicit"
  "|extern|false|float|for|friend|goto|if|inline|int|long|mutable"
  "|namespace|new|operator|pascal|private|protected|public|register"
  "|reinterpret_cast|return|short|signed|sizeof|static|static_cast"
  "|struct|switch|template|this|throw|true|try|typedef|typeid|typename"
  "|union|unsigned|using|virtual|void|volatile|wchar_t|while)\\>"
```

```
const char* format_string =
  "(?1<font color=\"#008040\">$&</font>)"
  "(?2<I><font color=\"#000080\">$&</font></I>)"
  "(?3<font color=\"#0000A0\">$&</font>)"
  "(?4<font color=\"#0000FF\">$&</font>)"
  "(?5<B>$&</B>)";
```

C++ to HTML Pretty-Printer: Main Pattern

```
// preprocessor directives: index 1
"(^[[:blank:]]*"
   "#"
   "(?:[^\\\\\\n]"
     "|\\\\\[^\\n[:punct:][:word:]]*[\\n[:punct:][:word:]]"
   ")*"
 ")|"
// string literals: index 4
"('(?:[^\\\\']|\\\\.)*'|\"(?:[^\\\\\"]|\\\\.)*\")|"
```

C++ to HTML Pretty-Printer: Main Pattern

```
  // preprocessor directives: index 1
  "(^[[:blank:]]*"
     "#"
     "(?:[^\\\\\\n]"
       "|\\\\\[^\\n[:punct:][:word:]]*[\\n[:punct:][:word:]]"
     ")*"
   ")|"
  // string literals: index 4
  "('(?:[^\\\\']|\\\\.)*'|\"(?:[^\\\\\"]|\\\\.)*\")|"
```

```
(^\s*
  #
  (?:[^\\\n]
    |\\[^\w\n[:punct:]]*[\w\n[:punct:]]
  )*
)



/('(?:[^\\']|\\.)*'|"(?:[^\\"]|\\.)*")/
```

C++ to HTML Pretty-Printer: Main Pattern

```
// literals: index 3
"\\<([+-]?"
    "(?:(?:0x[[:xdigit:]]+)"
      "|(?:(?:[[:digit:]]*\\.)?[[:digit:]]+(?:[eE][+-]?[[:digit:]]+)?)"
    ")"
    "u?(?:(?:int(?:8|16|32|64))|L)?"
")\\>|"
```

Numerical

Boost.Random

Boost.Random

```
// Produces randomness out of thin air.
boost::random::mt19937 rng;

// Distribution that maps to 1..6.
boost::random::uniform_int_distribution<> six(1,6);

// Roll a die.
int x = six(rng);
```

Boost.Functional/Hash

Boost.Functional/Hash

```cpp
#include <boost/functional/hash.hpp>

int main()
{
  boost::hash<std::string> string_hash;
  std::size_t h = string_hash("Hash me");
}
```

```cpp
template <class Container>
std::vector<std::size_t> hashes(Container const& x)
{
  std::vector<std::size_t> hashes;
  std::transform(x.begin(), x.end(), std::insert_iterator(hashes),
                 boost::hash<typename Container::value_type>());
  return hashes;
}
```

Boost.Functional/Hash

```
std::unordered_multiset<int, boost::hash<int>>  set_of_ints;

std::unordered_set<std::pair<int, int>, boost::hash<std::pair<int, int>>
        set_of_pairs;

std::unordered_map<int, std::string, boost::hash<int>> map_int_to_string;
```

Boost.Complex

Boost.Complex

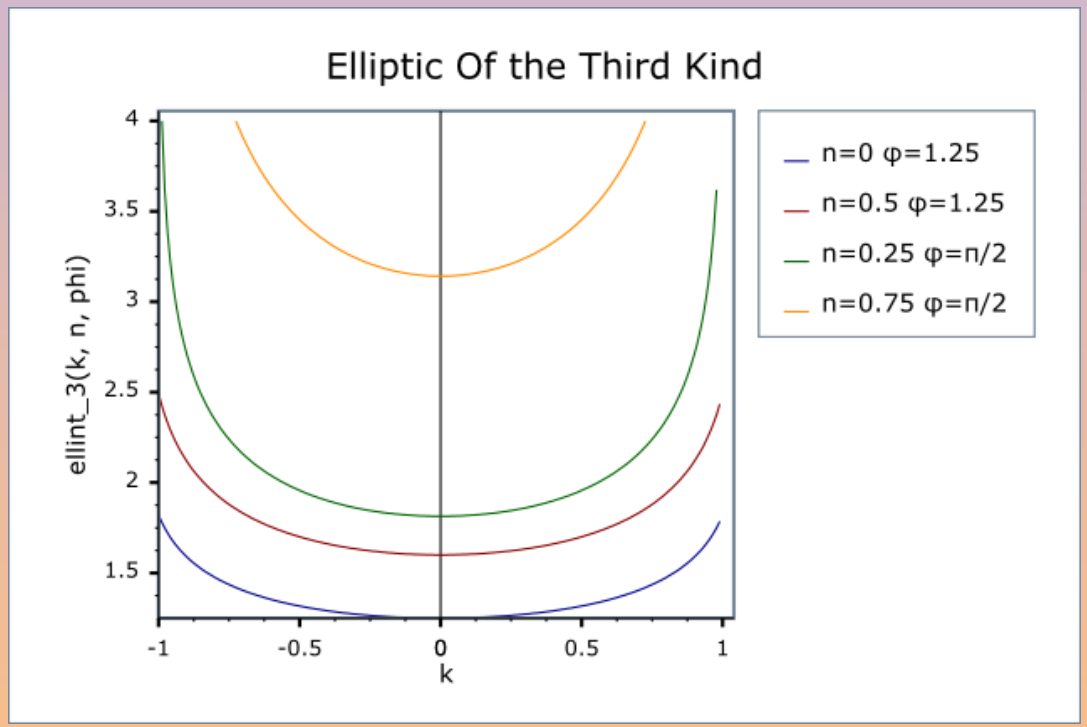
```
template <typename Real1, typename Real2>
std::complex <PROMOTE(Real1, Real2)>
   pow(const std::complex<Real1>& x, const std::complex<Real2>& y);


// sin^{-1}(z) = -i log(iz + sqrt(1 - z^2))
template <typename Real>
std::complex<real> asin(const std::complex<Real>& z);
```

Boost.Math/SpecialFunctions

Boost.Math/SpecialFunctions

- associated Laguerre polynomials
- associated Legendre functions
- beta function
- (complete) elliptic integral of the first, second and third kinds
- confluent hypergeometric functions
- regular modified cylindrical Bessel functions
- cylindrical Bessel functions (of the first kind)
- irregular modified cylindrical Bessel functions
- cylindrical Neumann functions
- cylindrical (incomplete) elliptic integral of the first, second and third kinds
- Bessel functions (of the second kind)
- exponential integral
- Hermite polynomials
- hypergeometric functions
- Laguerre polynomials
- Legendre polynomials
- Riemann zeta function
- spherical Bessel functions (of the first kind)
- spherical associated Legendre functions
- spherical Neumann functions
- spherical Bessel functions (of the second kind)

```
// [5.2.1.14] (incomplete) elliptic integral of the third kind:
double ellint_3(double k, double n, double phi);
float ellint_3f(float k, float n, float phi);
long double ellint_3l(long double k, long double n, long double phi);
```



$$\Pi(n, \varphi, k) = \int_0^{\varphi} \frac{\mathrm{d}\theta}{(1 - n\sin^2\theta)\sqrt{(1 - k^2\sin^2\theta)}}$$

$$\Pi(n, \varphi, k) = \int_0^{\varphi} \frac{\mathrm{d}\theta}{(1 - n\sin^2\theta)\sqrt{(1 - k^2\sin^2\theta)}}$$

Using Carlson symmetric forms of elliptic integrals [WCarlson_symmetric_form].

$$\begin{aligned}
\Pi(n, \varphi, k) &= \sin\varphi R_F(\cos^2\varphi, 1 - k^2\sin^2\varphi, 1) \\
&\quad + \frac{n}{3}\sin^3\varphi R_J(\cos^2\varphi, 1 - k^2\sin^2\varphi, 1, 1 - n\sin\varphi) \\
R_F(x, y, z) &= \frac{1}{2}\int_0^{\infty} \frac{\mathrm{d}t}{\sqrt{(t + x)(t + y)(t + z)}} \\
R_J(x, y, z, p) &= \frac{3}{2}\int_0^{\infty} \frac{\mathrm{d}t}{(t + p)\sqrt{(t + x)(t + y)(t + z)}}
\end{aligned}$$

Function Objects

Boost.Utility

boost::result_of: C++ 11

```
struct X
{
          int& operator()(int);
    int const& operator()(int) const;

         char& operator()(char&);
   char const* operator()(char const&);
};

int main()
{
#define CHECK(In, Out)                                                  \
  static_assert(boost::is_same<result_of<In>::type, Out>::value, "fail")

  using boost::result_of;
  CHECK(       X(int)          , int &);
  CHECK(const X(int)          , int const &);
  CHECK(       X(char&)        , char &);
  CHECK(       X(char const&), char const *);
}
```

boost::result_of: C++ 03 Compatibility Layer

```
struct X
{
          int& operator()(int);
    int const& operator()(int) const;

         char& operator()(char&);
   char const* operator()(char const&);

#ifndef BOOST_RESULT_OF_USE_DECLTYPE
   template<typename T> struct result;
#endif
};

#ifndef BOOST_RESULT_OF_USE_DECLTYPE
template<>struct X::result<      X(int)>  { typedef int& type; };
template<>struct X::result<const X(int)>  { typedef const int& type; };
template<>struct X::result<      X(char&)>{ typedef char& type; };
template<>struct X::result<X(char const&)>{ typedef char const* type; };
#endif
```

Boost.Bind

boost::mem_fn

An upgrade of std::mem_fun and std::mem_fun_ref (e.g., variadic, unified for ref/ptr/shared).

```
struct X
{
  void f();
};

void g(std::vector<X> & v)
  std::for_each(v.begin(), v.end(), boost::mem_fn(&X::f));
}

void h(std::vector<X *> const & v) {
  std::for_each(v.begin(), v.end(), boost::mem_fn(&X::f));
}

void k(std::vector<boost::shared_ptr<X> > const & v) {
  std::for_each(v.begin(), v.end(), boost::mem_fn(&X::f));
}
```

boost::bind

```
int f(int a, int b)        { return a + b; }
int g(int a, int b, int c) { return a + b + c; }

assert
{
  bind(f, 1, 2)()    == f(1, 2);
  bind(g, 1, 2, 3)() == g(1, 2, 3);

  // Placeholders.
  int x = 42, y = 51, z = 69;
  bind2nd(std::ptr_fun(f), 5)(x) = f(x, 5); // C++98
  bind(f, _1, 5)(x) == f(x, 5);             // TR1

  bind(g, _1, 9, _1)(x) ==  g(x, 9, x);
  bind(g, _3, _3, _3)(x, y, z) == g(z, z, z);

  int i = 5;
  bind(f, ref(i), _1);
  bind(f, cref(42), _1);
}
```

boost::bind

- Works for all sorts of function kinds
- Convenient support for Boolean operators

```
std::remove_if(first, last,
               !bind(&X::visible, _1));

std::find_if(first, last,
             bind(&X::name, _1) == "Peter");

std::find_if(first, last,
             bind(&X::name, _1) == "Paul"
             || bind(&X::name, _1) == "Peter");
             // Could be bind(&X::name, _1) == _2

std::sort(first, last,
          bind(&X::name, _1) < bind(&X::name, _2));
```

Boost.Function

boost::function

- Generalized callbacks.
- Works for all sorts of function kinds.

boost::function

```cpp
#include <boost/function.hpp>
#include <iostream>
#include <cstdlib> // atoi
#include <cstring> // strlen

int main()
{
  boost::function<int (const char* s)> f;
  // Or, for Peter Jackson powered compilers:
  //    boost::function1<int, const char*> f;
  try { f("1609"); }
  catch (boost::bad_function_call& ex)
  { std::cerr << ex.what() << '\n'; }

  f = std::atoi;
  std::cout << f("1609") << '\n';
  f = std::strlen;
  std::cout << f("1609") << '\n';
}
```

boost::function: Member Functions

```
#include <boost/function.hpp>
#include <iostream>

struct world
{
  void hello(std::ostream& os)
  {
    os << "Hello, world!" << '\n';
  }
};

int main()
{
  // "this" is the first argument (a pointer).
  boost::function<void (world*, std::ostream&)> f = &world::hello;
  world w;
  f(&w, boost::ref(std::cout));
}
```

Boost.TypeTraits

Type traits

> *Internationalizing the Standard C++ Library required inventing some novel techniques, one of which is the unexpectedly useful traits — it radically simplifies the interface to class templates instantiable on native C++ types.*
>
> — [Myers, 1995]

> *Think of a trait as a small object whose main purpose is to carry information used by another object or algorithm to determine "policy" or "implementation details".*
>
> — Bjarne Stroustrup

std::numeric_limits

```
#include <iostream>
#include <limits>

int main ()
{
  using namespace std;
  cout << boolalpha
       << "Minimum value: " << numeric_limits<int>::min() << '\n'
       << "Maximum value: " << numeric_limits<int>::max() << '\n'
       << "Is signed:     " << numeric_limits<int>::is_signed << '\n'
       << "Non-sign bits: " << numeric_limits<int>::digits << '\n'
       << "Has infinity:  " << numeric_limits<int>::has_infinity << '\n';
}
```

Digression: Why is numeric_limits<T>::min a Function

- C++ (03 and 11) forbids `static const` members with floating point:

```
struct math
{
  static const float pi = 3.14;
};
```

```
pi.cc:3:27: error: floating-point literal cannot appear
                   in a constant-expression
   static const float pi = 3.14;
                           ^
pi.cc:3:27: warning: ISO C++ forbids initialization of member
                     constant 'math::pi' of non-integral type
                     'const float' [-Wpedantic]
```

- separating definition from declaration would break its status of "constant"
- for consistency between integral and floating point types, traits than can return floating point values are defined as functions
- C++ 11 makes then `constexpr`

<type_traits> I

4.5.1 primary type categories:
`is_void`, `is_integral`, `is_floating_point`, `is_array`, `is_pointer`, `is_reference`, `is_member_object_pointer`, `is_member_function_pointer`, `is_enum`, `is_union`, `is_class`, `is_function`

4.5.2 composite type categories:
`is_arithmetic`, `is_fundamental`, `is_object`, `is_scalar`, `is_compound`, `is_member_pointer`

4.5.3 type properties:
`is_const`, `is_volatile`, `is_pod`, `is_empty`, `is_polymorphic`, `is_abstract`, `has_trivial_constructor`, `has_trivial_copy`, `has_trivial_assign`, `has_trivial_destructor`, `has_nothrow_constructor`, `has_nothrow_copy`, `has_nothrow_assign`, `has_virtual_destructor`, `is_signed`, `is_unsigned`, `alignment_of`, `rank`, `extent`

<type_traits> II

4.6 type relations:
`is_same`, `is_base_of`, `is_convertible`

4.7.1 const-volatile modifications:
`remove_const`, `remove_volatile`, `remove_cv`, `add_const`, `add_volatile`, `add_cv`

4.7.2 reference modifications:
`remove_reference`, `add_reference`

4.7.3 array modifications:
`remove_extent`, `remove_all_extents`

4.7.4 pointer modifications:
`remove_pointer`, `add_pointer`

Boost.EnableIf (C++ 11)

```
#include <type_traits>

// A is enabled via a template parameter
template <typename T, typename Enable = void>
class A; // undefined

template <typename T>
class A<T,
        typename std::enable_if<std::is_floating_point<T>::value>::type>
{};

int main()
{
  A<double> d;
  A<int>    i;
}
```

```
error: aggregate 'A<int> i' has incomplete type and cannot be defined
   A<int>    i; // compile-time error
             ^
```

Boost.StaticAssert (C++ 11)

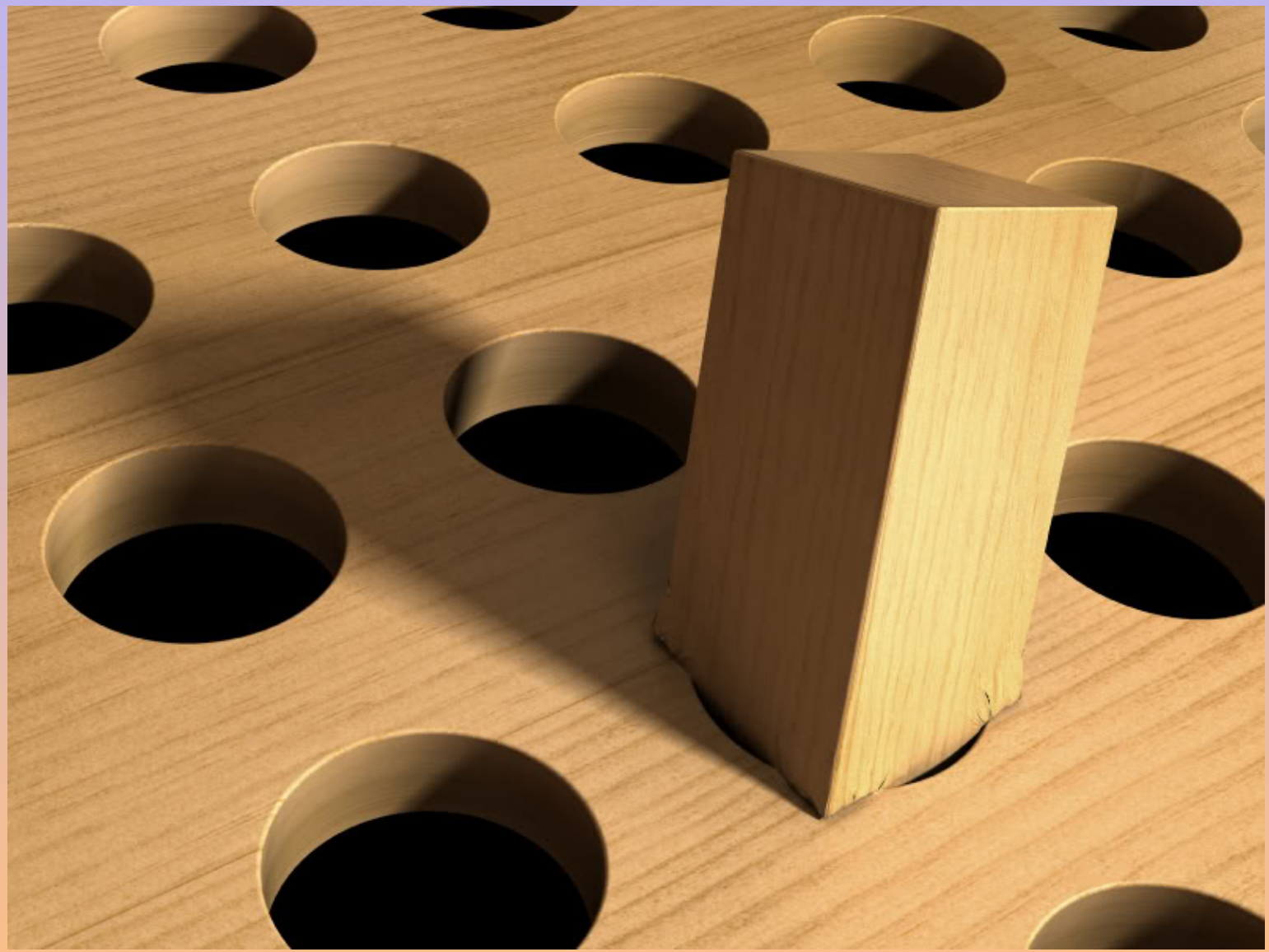
```
#include <type_traits>

template <typename T>
class A
{
  static_assert(std::is_floating_point<T>::value,
                "a float type is needed for A");
};

int main()
{
  A<double> d; // OK
  A<int>    i; // compile-time error
}
```

```
In instantiation of 'class A<int>':
14:13:   required from here
7:3: error: static assertion failed: a float type is needed for A
   static_assert(std::is_floating_point<T>::value,
   ^
```

enable_if vs. static_assert

- prefer `static_assert` to catch errors
- use `enable_if` to dispatch

SFINAE: Substitution Failure Is Not An Error



SFINAE

```
struct Test
{
  typedef int foo;
};

template <typename T> void f(typename T::foo) {} // Definition #1

template <typename T> void f(T) {}               // Definition #2

int main()
{
  f<Test>(10); // Call #1.
  f<int>(10);  // Call #2. Without error.
}
```

SFINAE

```
template <int I>
void foo(char(*)[I % 2 == 0 ? 1 : -1] = nullptr)
{
  std::cout << "even\n";
}

template <int I>
void foo(char(*)[I % 2 == 1 ? 1 : -1] = nullptr)
{
  std::cout << "odd\n";
}

int main()
{
  foo<42>();
  foo<51>();
}
```

SFINAE

<type_traits>: Optimizing by Hand

```
template<typename I1, typename I2>
inline I2 copy(I1 first, I1 last, I2 out)
{
  // We can copy with memcpy if T has a trivial assignment operator,
  // and if the iterator arguments are actually pointers (this last
  // requirement we detect with overload resolution):
  typedef typename std::iterator_traits<I1>::value_type value_type;
  return detail::copy_imp(first, last, out,
                          boost::has_trivial_assign<value_type>());
}
```

<type_traits>: Optimizing by Hand

```
namespace detail
{
  template<typename I1, typename I2, bool b>
  I2 copy_imp(I1 first, I1 last, I2 out,
              const boost::integral_constant<bool, b>&)
  {
    for (/* empty */; first != last; ++out, ++first)
      *out = *first;
    return out;
  }

  template<typename T>
  T* copy_imp(const T* first, const T* last, T* out,
              const boost::true_type&)
  {
    memmove(out, first, (last - first) * sizeof(T));
    return out + (last - first);
  }
}
```

<type_traits>: Optimizing by Hand

	`char`	`int`
1000 Conventional copies	8.07ms	8.02ms
1000 Optimized copies	0.99ms	2.52ms

Containers

Boost.Tuple

Boost.Tuple [Schäling, 2011]

```
#include <boost/tuple/tuple.hpp>
#include <boost/tuple/tuple_io.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::tuple<std::string, std::string> person;
  person p("Boris", "Schaeling");
  std::cout << p << '\n';

  person q = std::make_pair("Boris", "Baillie");
  std::cout << (p.first == q.first) << '\n';
}
```

Boost.Tuple: get

```
#include <boost/tuple/tuple.hpp>
#include <boost/tuple/tuple_io.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::tuple<std::string, std::string, int> person;
  person p("Boris", "Schaeling", 43);
  std::cout << p << '\n';

  std::cout << p.get<0>() << '\n';
  std::cout << boost::get<0>(p) << '\n';

  p.get<1>() = "Becker";
  std::cout << p << '\n';
}
```

Boost.Tuple: tie

```cpp
#include <boost/tuple/tuple.hpp>
#include <string>
#include <iostream>

boost::tuple<std::string, int> func()
{
  return boost::make_tuple("Error message", 2009);
}

int main()
{
  std::string errmsg;
  int errcode;

  boost::make_tuple(boost::ref(errmsg), boost::ref(errcode)) = func();
  std::cout << errmsg << ": " << errcode << '\n';
}
```

```cpp
  boost::tie(errmsg, errcode) = func();
```

Boost.Array

Boost.Array

Fixed-size, statically allocated arrays.

```
#include <boost/array.hpp>
#include <iostream>

int main()
{
  boost::array<int,4> a = {{ 1, 2, 3 }};
  boost::array<int,4> b = a;
  std::cout
    << a.front() << '\n'
    << a.size() << '\n'
    << a[2] << '\n'
    << (a < b) << '\n';
  std::swap(a, b);
}
```

Boost.Unordered

Boost.Unordered

```
template
<
  class Key,
  class Hash = boost::hash<Key>,
  class Pred = std::equal_to<Key>,
  class Alloc = std::allocator<Key>
>
class unordered_set;

template
<
  class Key,
  class Hash = boost::hash<Key>,
  class Pred = std::equal_to<Key>,
  class Alloc = std::allocator<Key>
>
class unordered_multiset;
```

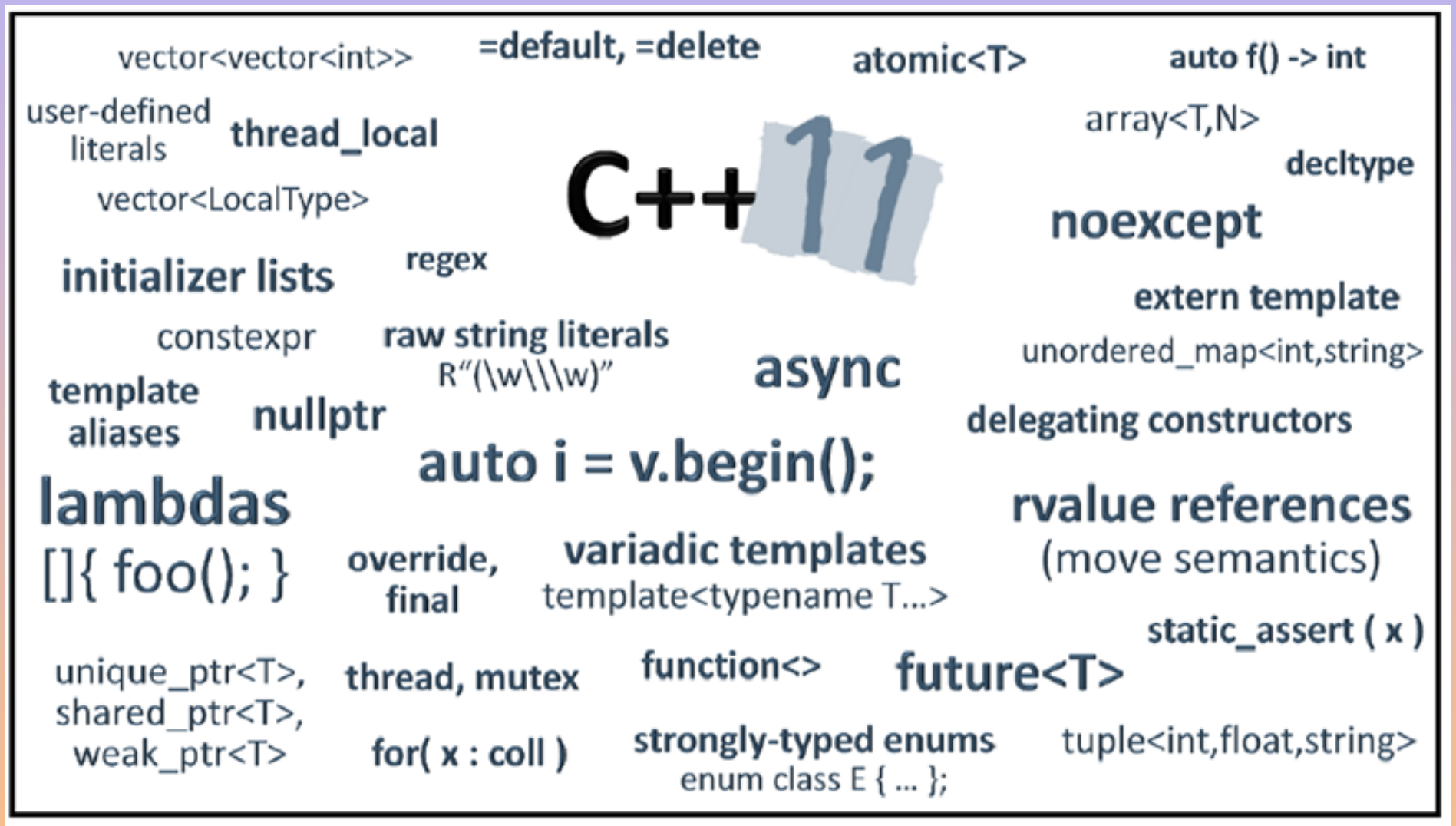
```
template
<
  class Key, class Mapped,
  class Hash = boost::hash<Key>,
  class Pred = std::equal_to<Key>,
  class Alloc = std::allocator<Key>
>
class unordered_map;

template
<
  class Key, class Mapped,
  class Hash = boost::hash<Key>,
  class Pred = std::equal_to<Key>,
  class Alloc = std::allocator<Key>
>
class unordered_multimap;
```

Boost.Unordered

```
typedef boost::unordered_map<std::string, int> map;
map x;
x["one"] = 1;
x["two"] = 2;
x["three"] = 3;

assert
{
  x.at("one") == 1;
  x.find("missing") == x.end();
}

BOOST_FOREACH(map::value_type i, x)
  std::cout << i.first << "," << i.second << "\n";
```

C++ 11

C++ 11


vector<vector<int>>
=default, =delete
atomic<T>
auto f() -> int
user-defined literals
thread_local
array<T,N>
decltype
C++11
vector<LocalType>
noexcept
initializer lists
regex
extern template
constexpr
raw string literals
R"(\w\\\w)"
async
unordered_map<int,string>
template aliases
nullptr
delegating constructors
auto i = v.begin();
lambdas
[]{ foo(); }
rvalue references
(move semantics)
override, final
variadic templates
template<typename T...>
static_assert (x)
unique_ptr<T>, shared_ptr<T>, weak_ptr<T>
thread, mutex
function<>
future<T>
for(x : coll)
strongly-typed enums
enum class E { ... };
tuple<int,float,string>

General Utilities

Boost.Foreach

Boost.Foreach: No Measurable Difference (at '-O3')

```
int main() {
  std::vector<int> v(1000000, 42);
  int sum = 0;
  for (int n = 0; n < 10000; ++n)
    for (std::vector<int>::const_iterator i = v.begin(), end = v.end();
         i != end; ++i)
      sum += *i;
  return sum % 100;
}
```

```
int main() {
  std::vector<int> v(1000000, 42);
  int sum = 0;
  for (int n = 0; n < 10000; ++n)
    for (int i: v)
      sum += i;
  return sum % 100;
}
```

```
#include <boost/foreach.hpp>
int main() {
  std::vector<int> v(1000000, 42);
  int sum = 0;
  for (int n = 0; n < 10000; ++n)
    BOOST_FOREACH (int i, v)
      sum += i;
  return sum % 100;
}
```

Boost.Foreach: No Measurable Difference (at '-O3')

```
int main() {
  std::vector<int> v(1000000, 42);
  int sum = 0;
  for (int n = 0; n < 10000; ++n)
    for (std::vector<int>::const_iterator i = v.begin(), end = v.end();
         i != end; ++i)
      sum += *i;
  return sum % 100;
}
```

```
int main() {
  std::vector<int> v(1000000, 42);
  int sum = 0;
  for (int n = 0; n < 10000; ++n)
    for (int i: v)
      sum += i;
  return sum % 100;
}
```

```
#include <boost/foreach.hpp>
int main() {
  std::vector<int> v(1000000, 42);
  int sum = 0;
  for (int n = 0; n < 10000; ++n)
    BOOST_FOREACH (int i, v)
      sum += i;
  return sum % 100;
}
```

Boost.Foreach: No Measurable Difference (at '-O3')

```cpp
int main() {
  std::vector<int> v(1000000, 42);
  int sum = 0;
  for (int n = 0; n < 10000; ++n)
    for (std::vector<int>::const_iterator i = v.begin(), end = v.end();
         i != end; ++i)
      sum += *i;
  return sum % 100;
}
```

```cpp
int main() {
  std::vector<int> v(1000000, 42);
  int sum = 0;
  for (int n = 0; n < 10000; ++n)
    for (int i: v)
      sum += i;
  return sum % 100;
}
```

```cpp
#include <boost/foreach.hpp>
int main() {
  std::vector<int> v(1000000, 42);
  int sum = 0;
  for (int n = 0; n < 10000; ++n)
    BOOST_FOREACH (int i, v)
      sum += i;
  return sum % 100;
}
```

Boost.StaticAssert

Boost.StaticAssert

```
namespace ultra_high_precision_trigonometry
{
  BOOST_STATIC_ASSERT_MSG(42 != 51,
                          "this is a PIII, it won't do");
}
```

Boost.StaticAssert

```
#include <limits>
#include <boost/static_assert.hpp>

template <class UInt>
class number
{
  BOOST_STATIC_ASSERT(    std::numeric_limits<UInt>::is_specialized
                       &&  std::numeric_limits<UInt>::is_integer
                       && !std::numeric_limits<UInt>::is_signed
                       && 16 <= std::numeric_limits<UInt>::digits);
public:
  // ...
};
```

Boost.ValueInitialized

Boost.ValueInitialized

```
#include <iostream>
#define ECHO(S) std::cerr << #S ": " << (S) << '\n'
#include <boost/utility/value_init.hpp>

void foo() { int x = 42; ECHO(x); }
void bar() { int y;      ECHO(y); }

int main() { foo();  bar(); }
```

```
x: 42
y: 42
```

Boost.ValueInitialized

How can you ensure proper initialization of a variable?

```
T1 var1;          // DefaultConstructible, fails for int.
T2 var2 = 0;      // Numeric (or designed to support it).
T3 var3 = {};     // Aggregates.
T4 var4 = T4();   // CopyConstructible.
```

C++ 11 unifies to `T var{...}` and `T var = {...}`.

Boost.ValueInitialized

```
#include <iostream>
#define ECHO(S) std::cerr << #S ": " << (S) << '\n'
#include <boost/utility/value_init.hpp>

void foo() {
  int x[2] = { 42, 51 };
  ECHO(x[0] * 100 + x[1]);
}

void bar() {
  boost::value_initialized<int[2]> x;
  ECHO(y[0] * 100 + y[1]);
}

int main() { foo(); bar(); }
```

```
x[0] * 100 + x[1]: 4251
y[0] * 100 + y[1]: 0
```

Boost.(Typed)?InPlaceFactory

Boost.(Typed)?InPlaceFactory

Suppose we have a class

```
struct X
{
  X(int, std::string);
};
```

and a container for it that supports an empty state:

```
struct C
{
  C() : x_(nullptr) {}
  ~C() { delete x_; }
  X* x_;
};
```

Boost.(Typed)?InPlaceFactory

A container supporting an empty state typically requires its contents to be CopyConstructible:

```
struct C
{
  C() : x_(nullptr) {}
  C(X const& v) : x_(new X(v)) {}
  ~C() { delete x_; }
  X* x_;
};

int main()
{
  // Temporary object created.
  C c(X(123,"hello"));
}
```

Boost.(Typed)?InPlaceFactory

```
struct C
{
  C() : x_(nullptr) {}
  C(X const& v) : x_(new X(v)) {}
  C(int a0, std::string a1) : x_(new X(a0, a1)) {}
  ~C() { delete x_; }
  X* x_;
};

int main()
{
  // Wrapped object constructed in-place.
  // No temporary created.
  C c(123,"hello");
}
```

Poor maintainability. . .

Boost.(Typed)?InPlaceFactory

```
struct C
{
  template<class InPlaceFactory>
  C(InPlaceFactory const& aFactory) : x_((X*)new char[sizeof(X)])
  {
    aFactory.template apply<X>(x_);
  }

  ~C() {
    x_->X::~X();
    delete[] x_;
  }

  X* x_;
};

int main() {
  C c(boost::in_place(123, "hello"));
}
```

C++ 11: Perfect Forwarding

```
struct C
{
  template <typename... Args>
  C(Args&&... args)
    : x_{new X{std::forward<Args>(args)...}}
  {}

  ~C() { delete x_; }

  X* x_;
};
```

C++ 11: Perfect Forwarding

```
struct C
{
  template <typename... Args>
  C(Args&&... args)
    : x_{new X{std::forward<Args>(args)...}}
  {}

  ~C() { delete x_; }

  X* x_;
};
```

```
struct X {
  X (int, std::string) {}
  X (int, int) {}
  X(const X&) = delete;
  X() = delete;
  X& operator=(const X&) = delete;
};
```

C++ 11: Perfect Forwarding

```
struct C
{
  template <typename... Args>
  C(Args&&... args)
    : x_{new X{std::forward<Args>(args)...}}
  {}

  ~C() { delete x_; }

  X* x_;
};
```

```
struct X {
  X (int, std::string) {}
  X (int, int) {}
  X(const X&) = delete;
  X() = delete;
  X& operator=(const X&) = delete;
};
```

```
void c()
{
  C c1{42, "51"}, c2{42, 51};
}
```

C++ 11: Perfect Forwarding

```
struct D
{
  template <typename... Args>
  D(Args&&... args)
    : x_{std::make_shared<X>(std::forward<Args>(args)...)}
  {}

  std::shared_ptr<X> x_;
};
void d()
{
  D d1{42, "51"}, d2{42, 51};
}
```

Boost.Move

Boost.Move

```
template <class T> void swap(T& a, T& b)
{
   T tmp(a);    // now we have two copies of a
   a = b;       // now we have two copies of b
   b = tmp;     // now we have two copies of tmp (aka a)
}
```

Imagine the cost for containers (e.g., vector)!

```
template <class T>
void swap(T& a, T& b)
{
   T tmp(std::move(a));
   a = std::move(b);
   b = std::move(tmp);
}
```

```
template <class T>
void swap(T& a, T& b)
{
   T tmp(boost::move(a));
   a = boost::move(b);
   b = boost::move(tmp);
}
```

Boost.Move

```
template <class T> void swap(T& a, T& b)
{
   T tmp(a);    // now we have two copies of a
   a = b;       // now we have two copies of b
   b = tmp;     // now we have two copies of tmp (aka a)
}
```

Imagine the cost for containers (e.g., `vector`)!

```
template <class T>
void swap(T& a, T& b)
{
   T tmp(std::move(a));
   a = std::move(b);
   b = std::move(tmp);
}
```

```
template <class T>
void swap(T& a, T& b)
{
   T tmp(boost::move(a));
   a = boost::move(b);
   b = boost::move(tmp);
}
```

C++ 11: Move Semantics

```
template <class T> class clone_ptr { T* ptr;
public:
  explicit clone_ptr(T* p = 0) : ptr(p) {} // construction
  ~clone_ptr() { delete ptr; }             // destruction
```

```
// copy semantics
clone_ptr(const clone_ptr& p)
  : ptr(p.ptr ? p.ptr->clone() : 0)
{}

clone_ptr&
operator=(const clone_ptr& p)
{
  if (this != &p) {
    T *p = p.ptr ? p.ptr->clone() : 0;
    delete ptr;
    ptr = p;
  }
  return *this;
}
```

```
// move semantics
clone_ptr(clone_ptr&& p)
  : ptr(p.ptr)
{ p.ptr = 0; }

clone_ptr&
operator=(clone_ptr&& p)
{
  if (this != &p) {
    std::swap(ptr, p.ptr);
    delete p.ptr;
    p.ptr = 0;
  }
  return *this;
}
```

C++ 11: Move Semantics

```
template <class T> class clone_ptr { T* ptr;
public:
  explicit clone_ptr(T* p = 0) : ptr(p) {} // construction
  ~clone_ptr() { delete ptr; }             // destruction
```

```
// copy semantics
clone_ptr(const clone_ptr& p)
  : ptr(p.ptr ? p.ptr->clone() : 0)
{}

clone_ptr&
operator=(const clone_ptr& p)
{
  if (this != &p) {
    T *p = p.ptr ? p.ptr->clone() : 0;
    delete ptr;
    ptr = p;
  }
  return *this;
}
```

```
// move semantics
clone_ptr(clone_ptr&& p)
  : ptr(p.ptr)
{ p.ptr = 0; }

clone_ptr&
operator=(clone_ptr&& p)
{
  if (this != &p) {
    std::swap(ptr, p.ptr);
    delete p.ptr;
    p.ptr = 0;
  }
  return *this;
}
```

Boost.Move

```
template <class T>
class clone_ptr
{
  // Mark this class copyable and movable
  BOOST_COPYABLE_AND_MOVABLE(clone_ptr)
  T* ptr;

public:
  // Construction
  explicit clone_ptr(T* p = 0)
    : ptr(p)
  {}

  // Destruction
  ~clone_ptr()
  {
    delete ptr;
  }
```

There is also BOOST_MOVABLE_BUT_NOT_COPYABLE

Boost.Move

```
// Copy semantics...
clone_ptr(const clone_ptr& p)
  : ptr(p.ptr ? p.ptr->clone() : 0)
{}

clone_ptr& operator=(BOOST_COPY_ASSIGN_REF(clone_ptr) p)
{
  if (this != &p)
  {
    T *t = p.ptr ? p.ptr->clone() : 0;
    delete ptr;
    ptr = t;
  }
  return *this;
}
```

Boost.Move

```
// Move semantics...
clone_ptr(BOOST_RV_REF(clone_ptr) p)
  : ptr(p.ptr)
{ p.ptr = 0; }

clone_ptr& operator=(BOOST_RV_REF(clone_ptr) p)
{
  if (this != &p)
  {
    delete ptr;
    ptr = p.ptr;
    p.ptr = 0;
  }
  return *this;
}
```

Boost.TypeOf

Boost.TypeOf

`make_pair` helps keeping values simple.

```
std::pair<int, double>(5, 3.14159);
```

```
std::make_pair(5, 3.14159);
```

But if you need to a variable. . .

```
std::pair<int, double> p(5, 3.14159);
```

```
std::pair<int, double> p = std::make_pair(5, 3.14159);
```

Boost.TypeOf: _1 > 15 && _2 < 20

```
lambda_functor<
    lambda_functor_base<
        logical_action<and_action>,
        tuple<
            lambda_functor<
                lambda_functor_base<
                    relational_action<greater_action>,
                    tuple<
                        lambda_functor<placeholder<1> >,
                        int const > > >,
            lambda_functor<
                lambda_functor_base<
                    relational_action<less_action>,
                    tuple<
                        lambda_functor<placeholder<2> >,
                        int const > > >
        >
    >
>
```

Boost.TypeOf

```
auto f = _1 > 15 && _2 < 20;
```

```
BOOST_TYPEOF(_1 > 15 && _2 < 20) f = _1 > 15 && _2 < 20;
```

Boost.Algorithm

Boost.Algorithm

```
// c = { 0, 1, 2, 3, 14, 15 }

bool isOdd (int i) { return i % 2 == 1; }
bool lessThan10 (int i) { return i < 10; }

using boost::algorithm;
assert
{
  !all_of(c, isOdd);
  !all_of(c.begin(), c.end(), lessThan10);
  all_of(c.begin(), c.begin() + 3, lessThan10);
  all_of(c.end(), c.end(), isOdd);
  !all_of_equal(c, 3);
  all_of_equal(c.begin() + 3, c.begin() + 4, 3);
  all_of_equal(c.begin(), c.begin(), 99);
}
```

all, any, none, one

Boost.Algorithm

```
clamp(v, low, high);

boost::tie(min, max) = minmax(v);
boost::tie(argmin, argmax) = minmax_element(v);

is_sorted(v);
is_sorted(v.begin(), v.end());
is_sorted_until(v.begin(), v.end(), std::less<int>());
```

is(_strictly)?_(de|in)creasing

Function Objects

Boost.LocalFunction

Boost.LocalFunction

```
int main() {
  int sum = 0, factor = 10;

  auto add = [factor, &sum](int num) {
      sum += factor * num;
  };

  add(1);                                    // Call the lambda.
  int nums[] = {2, 3};
  std::for_each(nums, nums + 2, add);        // Pass it to an algorithm.

  assert(sum == 60);
}
```

Using C++ 11's lambdas.

Boost.LocalFunction

```
int main() {
  int sum = 0, factor = 10;

  void BOOST_LOCAL_FUNCTION(const bind factor, bind& sum, int num) {
      sum += factor * num;
  } BOOST_LOCAL_FUNCTION_NAME(add)

  add(1);                                    // Call the local function.
  int nums[] = {2, 3};
  std::for_each(nums, nums + 2, add);        // Pass it to an algorithm.

  assert(sum == 60);
}
```

Basically the same performances.

Boost.LocalFunction

Using GCC's Statement Expressions

```
int val = 2;
int nums[] = {1, 2, 3};
int* end = nums + 3;

int* i = std::find_if(nums, end,
  GCC_LAMBDA(const bind val, int num,
             return bool) {
    return num == val;
  } GCC_LAMBDA_END
);
```

```
int val = 2;
int nums[] = {1, 2, 3};
int* end = nums + 3;

int* i = std::find_if(nums, end,
  [val](int num)
    -> bool {
    return num == val;
  }
);
```

Boost.Lambda

Boost.Lambda

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

void print(int i)
{
  std::cout << i << std::endl;
}

int main()
{
  std::vector<int> v(3, 42);
  std::for_each
    (v.begin(), v.end(),
     print);
}
```

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

#include <boost/lambda/lambda.hpp>
using namespace boost::lambda;



int main()
{
  std::vector<int> v(3, 51);
  std::for_each
    (v.begin(), v.end(),
     std::cout << _1 << "\n");
}
```

Boost.Lambda

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

void print(int i)
{
  std::cout << i << std::endl;
}

int main()
{
  std::vector<int> v(3, 42);
  std::for_each
    (v.begin(), v.end(),
     print);
}
```

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

#include <boost/lambda/lambda.hpp>
using namespace boost::lambda;



int main()
{
  std::vector<int> v(3, 51);
  std::for_each
    (v.begin(), v.end(),
     std::cout << _1 << "\n");
}
```

Boost.Lambda

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

void print(int i)
{
  std::cout << i << std::endl;
}

int main()
{
  std::vector<int> v(3, 42);
  std::for_each
    (v.begin(), v.end(),
     print);
}
```

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

#include <boost/lambda/lambda.hpp>
using namespace boost::lambda;



int main()
{
  std::vector<int> v(3, 51);
  std::for_each
    (v.begin(), v.end(),
     std::cout << _1 << "\n");
}
```

Can you spot the tiny difference between the two?

Boost.Lambda

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

void print(int i)
{
  std::cout << i << std::endl;
}

int main()
{
  std::vector<int> v(3, 42);
  std::for_each
    (v.begin(), v.end(),
     print);
}
```

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

#include <boost/lambda/lambda.hpp>
using namespace boost::lambda;



int main()
{
  std::vector<int> v(3, 51);
  std::for_each
    (v.begin(), v.end(),
     std::cout << _1 << "\n");
}
```

Can you spot the tiny difference between the two?

You must not use `std::endl`

Boost.Lambda: Pitfalls (wc: l177 L570 w1114 c20095)

```
foo.cc: In function 'int main()':
foo.cc:15:22: error: no match for 'operator<<' (operand types are 'const boost::lambda::lambda_functor<boost::lambda::
    lambda_functor_base<boost::lambda::bitwise_action<boost::lambda::leftshift_action>, boost::tuples::tuple<std::
    basic_ostream<char>&, boost::lambda::lambda_functor<boost::lambda::placeholder<1> >, boost::tuples::null_type, boost
    ::tuples::null_type, boost::tuples::null_type, boost::tuples::null_type, boost::tuples::null_type, boost::tuples::
    null_type, boost::tuples::null_type, boost::tuples::null_type> > >' and '<unresolved overloaded function type>')
      std::cout << _1 << std::endl);
                   ^
foo.cc:15:22: note: candidates are:
In file included from /opt/local/include/boost/lambda/lambda.hpp:26:0,
                 from foo.cc:5:
/opt/local/include/boost/lambda/detail/operators.hpp:114:1: note: template<class Arg, class B> const boost::lambda::
    lambda_functor<boost::lambda::lambda_functor_base<boost::lambda::bitwise_action<boost::lambda::leftshift_action>,
    boost::tuples::tuple<boost::lambda::lambda_functor<T>, typename boost::lambda::const_copy_argument<const B>::type> >
    > boost::lambda::operator<<(const boost::lambda::lambda_functor<T>&, const B&)
 BOOST_LAMBDA_BE(operator<<, bitwise_action<leftshift_action>, const A, const B,
 ^
/opt/local/include/boost/lambda/detail/operators.hpp:114:1: note:   template argument deduction/substitution failed:
foo.cc:15:30: note:   couldn't deduce template parameter 'B'
      std::cout << _1 << std::endl);
                             ^
In file included from /opt/local/include/boost/lambda/lambda.hpp:26:0,
                 from foo.cc:5:
/opt/local/include/boost/lambda/detail/operators.hpp:114:1: note: template<class A, class Arg> const boost::lambda::
    lambda_functor<boost::lambda::lambda_functor_base<boost::lambda::bitwise_action<boost::lambda::leftshift_action>,
    boost::tuples::tuple<typename boost::lambda::const_copy_argument<const A>::type, boost::lambda::lambda_functor<Arg> >
     > > boost::lambda::operator<<(const A&, const boost::lambda::lambda_functor<Arg>&)
 BOOST_LAMBDA_BE(operator<<, bitwise_action<leftshift_action>, const A, const B,
 ^
/opt/local/include/boost/lambda/detail/operators.hpp:114:1: note:   template argument deduction/substitution failed:
foo.cc:15:30: note:   couldn't deduce template parameter 'Arg'
      std::cout << _1 << std::endl);
                             ^
```

Boost.Lambda

```
#include <boost/lambda/lambda.hpp>
#include <boost/lambda/if.hpp>
#include <iostream>
#include <vector>
#include <algorithm>

int main()
{
  using namespace boost::lambda;
  std::vector<int> v(3, 42);
  std::for_each(v.begin(), v.end(),
                if_then(_1 > 1,
                        std::cout << _1 << "\n"));
}
```

Wrappers for loops, exceptions etc.

Summary

Boost vs. TR1

Array	`std::array`
Bind	`std::bind`
Enable If	`std::enable_if`
Function	`std::function`
Member Function	`std::mem_fn`
Random	`<random>`
Ref	`std::ref`, `std::cref`
Regex	`<regex>`
Result Of	`std::result_of`
Smart Ptr	`std::unique_ptr`, `std::shared_ptr`, `std::weak_ptr`
Swap	`std::swap`
Tuple	`std::tuple`
Type Traits	`<type_traits>`
Unordered	`<unordered_set>`, `<unordered_map>`

Boost vs. C++ 11

Foreach	Range-based for
Functional/Forward	Perfect forwarding (rvalue ref, variadic templates, `std::forward`)
In Place Factory	Perfect forwarding
Lambda	Lambda expression
Local function	Lambda expression
Min-Max	`std::minmax`, `std::minmax_element`
Move	Rvalue references
Ratio	`std::ratio`
Static Assert	`static_assert`
Thread	`<thread>`, `<mutex>`, etc.
Typeof	`auto`, `decltype`
Value initialized	List-initialization


[8851670: Relevant Boost features vs C++ 11]

Bibliography

Bibliography I


ISO/IEC (2006).
Draft technical report on C++ library extensions.
Technical Report N1836, ISO/IEC.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1836.pdf.


Meyers, S. (2005).
Effective C++: 55 Specific Ways to Improve Your Programs and Designs (3rd Edition).
Addison-Wesley Professional.


Myers, N. C. (1995).
Traits: a new and useful template technique.
C++ Report, 7(5):32–35.
http://www.cantrip.org/traits.html.

Bibliography II


Schäling, B. (2011).
The Boost C++ Libraries.
XML Press.
http://en.highscore.de/cpp/boost/.


Stroustrup, B. (2003).
Abstraction, libraries, and efficiency in C++.
Dr. Dobb's Journal China, 1(1).


Sutter, H. and Alexandrescu, A. (2005).
C++ Coding Standards: 101 Rules, Guidelines, And Best Practices.
The C++ In-Depth Series. Addison-Wesley.

Boost

Questions?

Questions?