aerix consulting

C++11 Library Design

Lessons from Boost and the Standard Library

Goals of This Talk

~~C++11 Gotchas!~~

Goals of This Talk

~~Tips and Tricks!~~

Goals of This Talk

~~Useless and Unnecessary TMP Heroics!~~

Goals of This Talk

Interface Design
Best Practices

Talk Overview

I. Function Interface Design
II. Class Design
III. "Module" Design

I. Function Interface Design

"Is my function … ?"

- … easy to call correctly?
- … hard to call incorrectly?
- … efficient to call?
  - …with minimal copying?
  - …with minimal aliasing?
  - …without unnecessary resource allocation?
- … easily composable with other functions?
- … usable in higher-order constructs?

Function Interfaces

What’s the best way of getting data into and out of a function?

Passing and Returning in C++98

Category	C++98 Recommendation
Input	
small	Pass by value
large	Pass by const ref
Output	
small	Return by value
large	Pass by (non-const) ref
Input/Output	Pass by non-const ref

How does C++11 change this picture?

MOVE SEMANTICS

Input Argument Categories

Read-only: value is only ever read from, never modified or stored

Sink: value is consumed, stored, or mutated locally

```
std::ostream& operator<<(std::ostream&, Task const &);
```

Task only read from

```
struct TaskQueue {
    void Enqueue(Task const &);
};
```

Task saved somewhere

Input Argument Categories

Read-only: value is only ever read from, never modified or stored

```
std::ostream& operator<<(std::ostream&, Task const &);
```

Guideline 1: Continue taking *read-only* value by const ref (except small ones)

“Sink” Input Arguments, Take 1

Goal: Avoid unnecessary copies, allow temporaries to be moved in.

```
struct TaskQueue {
  void Enqueue(Task const &);
  void Enqueue(Task &&);
};
```

Handles lvalues

Handles rvalues

```
Task MakeTask();

Task t;
TaskQueue q;

q.Enqueue(t);          // copies
q.Enqueue(MakeTask()); // moves
```

Programmer Heaven?

What if the function takes more than 1 sink argument?

```
struct TaskQueue {
  void Enqueue(Task const &, Task const &);
  void Enqueue(Task const &, Task &&);
  void Enqueue(Task &&, Task const &);
  void Enqueue(Task &&, Task &&);
};
```

"This isn't heaven. This sucks."

Sink Input Arguments, Take 2

Guideline 2: Take sink arguments *by value*

```
struct TaskQueue {
  void Enqueue(Task);
};
```

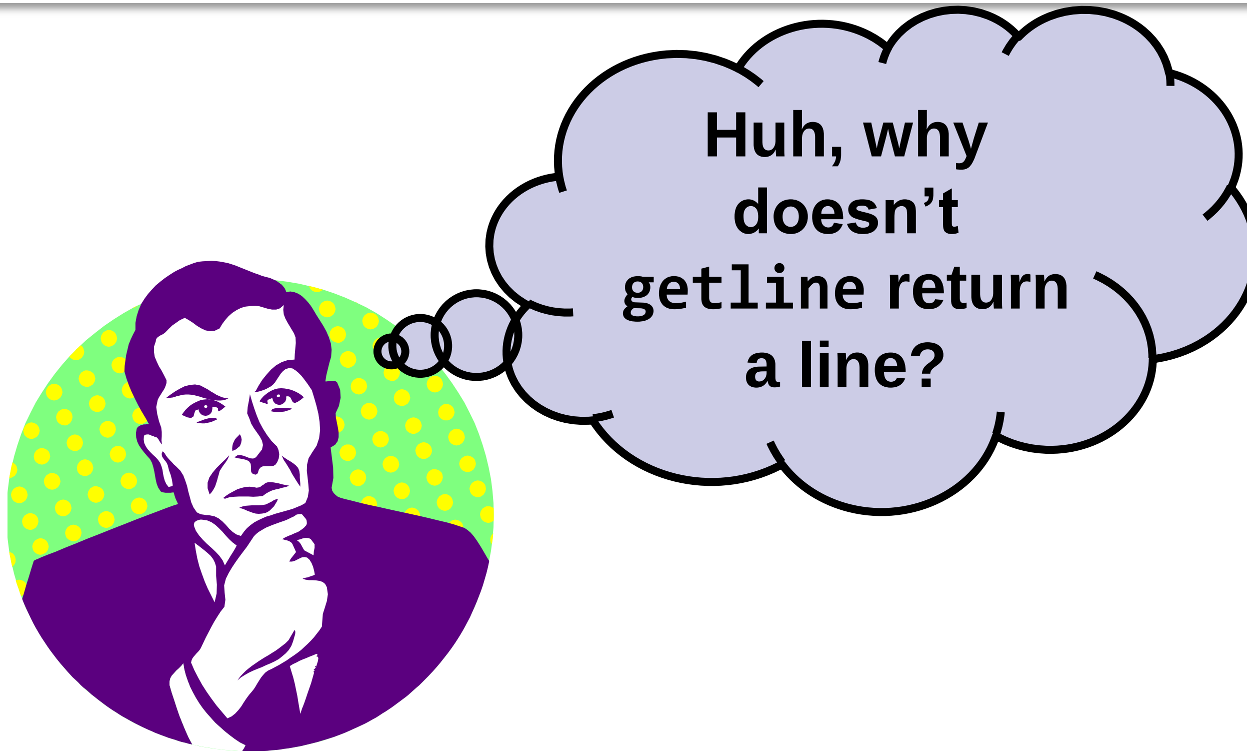
```
Task MakeTask();

Task t;
TaskQueue q;

q.Enqueue(t);             // copies
q.Enqueue(MakeTask()); // moves
```

Passing and Returning in C++11

Category	C++11 Recommendation
Input	
small & "sink"	Pass by value
all others	Pass by const ref
Output	Return by value
Input/Output	Pass by non-const ref (?)

Example: getline

```
std::istream & getline(std::istream &, std::string &);
```



Example: getline

```
std::istream & getline(std::istream &, std::string &);
```

```
std::string line;
if(std::getline(std::cin, line))
    use_line(line);
```

Must declare a string on a separate line

Can't immediately use the result

Example: getline, Improved?

```
std::string getline(std::istream &);
```

```
// Isn't this nicer?
use_line(getline(std::cin));
```

Example: getline

```
std::istream & getline(std::istream &, std::string &);
```

```
int main() {
    std::string line;
    while(std::getline(std::cin, line)) {
        use_line(line);
    }
}
```

Repeated calls to `getline` should reuse memory!

getline: Observation

```
std::istream & getline(std::istream &, std::string &);
```

This is NOT an out parameter!

Example: getline for C++11

```
lines_range getlines(std::istream &);
```

Fetches lines lazily, on demand

`std::string` data member gets reused

```
for(std::string const& line : getlines(std::cin))
    use_line(line);
```

[“Out Parameters, Move Semantics, and Stateful Algorithms”](http://ericniebler.com/2013/10/13/out-parameters-vs-move-semantics/)
http://ericniebler.com/2013/10/13/out-parameters-vs-move-semantics/

Input / Output Parameters

They indicate an algorithm is *stateful*

- *E.g.* current state, cache, precomputed data, buffers, etc.

Guideline 3: Encapsulate an algorithm's state in an object that implements the algorithm.

Examples: lines_range, Boost's boyer_moore

Passing and Returning in C++11

Category	C++11 Recommendation
Input	
small & "sink"	Pass by value
all others	Pass by const ref
Output	Return by value
Input/Output	Use a stateful algorithm object (*)

(*) Initial state is a **sink** argument to the constructor

Whither && 

OK, One Gotcha!

```
template< class Queue, class Task >
void Enqueue( Queue & q, Task const & t )
{
    q.Enqueue( t );
}
template< class Queue, class Task >
void Enqueue( Queue & q, Task && t )
{
    q.Enqueue( std::move( t ) );
}
```

Const ref here

Rvalue ref here

```
TaskQueue q;
Task t = MakeTask();

Enqueue( q, t );
```

Which overload?

If you don't know why this code is broken, seriously reconsider trying to do something clever with rvalue references!

"Fear rvalue refs like one might fear God. They are powerful and good, but the fewer demands placed on them, the better."

— Me

Perfect Forwarding Pattern

Uses [variadic] templates and rvalue refs in a specific pattern:

Argument is of form T&& where T is deduced

```
template< class Fun, class ...Args >
auto invoke( Fun && fun, Args && ... args )
{
    return std::forward<Fun>(fun)(std::forward<Args>(args)...);
}
```

C++14 only

Argument is used with `std::forward<T>(t)`

II. Class design

Designing classes for C++11

Class Design in C++11

How to design a class in C++11…

- … that makes best use of C++11
- … that plays well with C++11
  - language features
    - Copy, assign, move, range-based for, etc.
    - Composes well with other types
    - Can be used anywhere (heap, stack, static storage, in constant expressions, etc.)
  - library features
    - Well-behaved in generic algorithms
    - Well-behaved in containers

"Can my type be…?"

…copied and assigned?

…efficiently passed and returned?

…efficiently inserted into a vector?

…sorted?

…used in a map? An unordered_map?

…iterated over (if it's a collection)?

…streamed?

…used to declare global constants?

Regular Types

- What are they?
  - Basically, `int`-like types.
  - Copyable, default constructable, assignable, equality-comparable, swappable, order-able
- Why do we care?
  - They let us reason mathematically
  - The STL containers and algorithms assume regularity in many places

- How do they differ in C++03 and C++11?

C++98 Regular Type

```
class Regular {
    Regular();
    Regular(Regular const &);
    ~Regular(); // throw()
    Regular & operator=(Regular const &);
    friend bool operator==(Regular const &, Regular const &);
    friend bool operator!=(Regular const &, Regular const &);
    friend bool operator<(Regular const &, Regular const &);
    friend void swap(Regular &, Regular &); // throw()
};
```

Or specialize std::less

T a = b; assert(a==b);
T a; a = b; ⇔ T a = b;
T a = c; T b = c; a = d; assert(b==c);
T a = c; T b = c; zap(a); assert(b==c && a!=b);

"Fundamentals of Generic Programming", J. Dehnert, A. Stepanov, http://www.stepanovpapers.com/DeSt98.pdf

C++11 Regular Type

```
class RegularCxx11 {
    RegularCxx11();
    RegularCxx11(RegularCxx11 const &);
    RegularCxx11(RegularCxx11 &&) noexcept;
   ~RegularCxx11();
    RegularCxx11 & operator=(RegularCxx11 const &);
    RegularCxx11 & operator=(RegularCxx11 &&) noexcept;
    friend bool operator==(RegularCxx11 const &, RegularCxx11 const &);
    friend bool operator!=(RegularCxx11 const &, RegularCxx11 const &);
    friend bool operator<(RegularCxx11 const &, RegularCxx11 const &);
    friend void swap(RegularCxx11 &, RegularCxx11 &); // throw()
};
namespace std {
   template<> struct hash<RegularCxx11>;
}
```

"What is a 'Regular Type' in the context of move semantics?" S. Parent, stackoverflow.com, Dec 2012 http://stackoverflow.com/a/14000046/195873

C++11 Class Design

Guideline 4: Make your types regular (if possible)

Guideline 5: Make your types' move operations `noexcept` (if possible)

Statically Check Your Classes

Q: Is my type Regular?

A: Check it at compile time!

```
template<typename T>
struct is_regular
  : std::integral_constant< bool,
        std::is_default_constructible<T>::value &&
        std::is_copy_constructible<T>::value &&
        std::is_move_constructible<T>::value &&
        std::is_copy_assignable<T>::value &&
        std::is_move_assignable<T>::value >
{};
```

```
struct T {};
static_assert(is_regular<T>::value, "huh?");
```

equality_comparable

```
namespace detail
{
    template<typename T>
    std::false_type check_equality_comparable(T const & t, long);

    template<typename T>
    auto check_equality_comparable(T const & t, int)
        -> typename std::is_convertible<decltype( t == t ), bool>::type;
}

template<typename T>
struct is_equality_comparable
  : decltype(detail::check_equality_comparable(std::declval<T const &>(), 1))
{};
```

A Very Moving Example

Imagine a `unique_ptr` that guarantees its pointer is non-null:

```
template<class T>
class non_null_unique_ptr
{
    T* ptr_;
public:
    non_null_unique_ptr() : ptr_(new T{}) {}
    non_null_unique_ptr(T* p) : ptr_(p) { assert(p); }
    T* get() const { return ptr_; }
    non_null_unique_ptr(non_null_unique_ptr &&) noexcept; // ???
    // etc...
};
```

What does non_null_unique_ptr's move c'tor do?

A Very Moving Example

Class invariant of `non_null_unique_ptr`:

```
ptr.get() != nullptr
```

What does the move c'tor do?

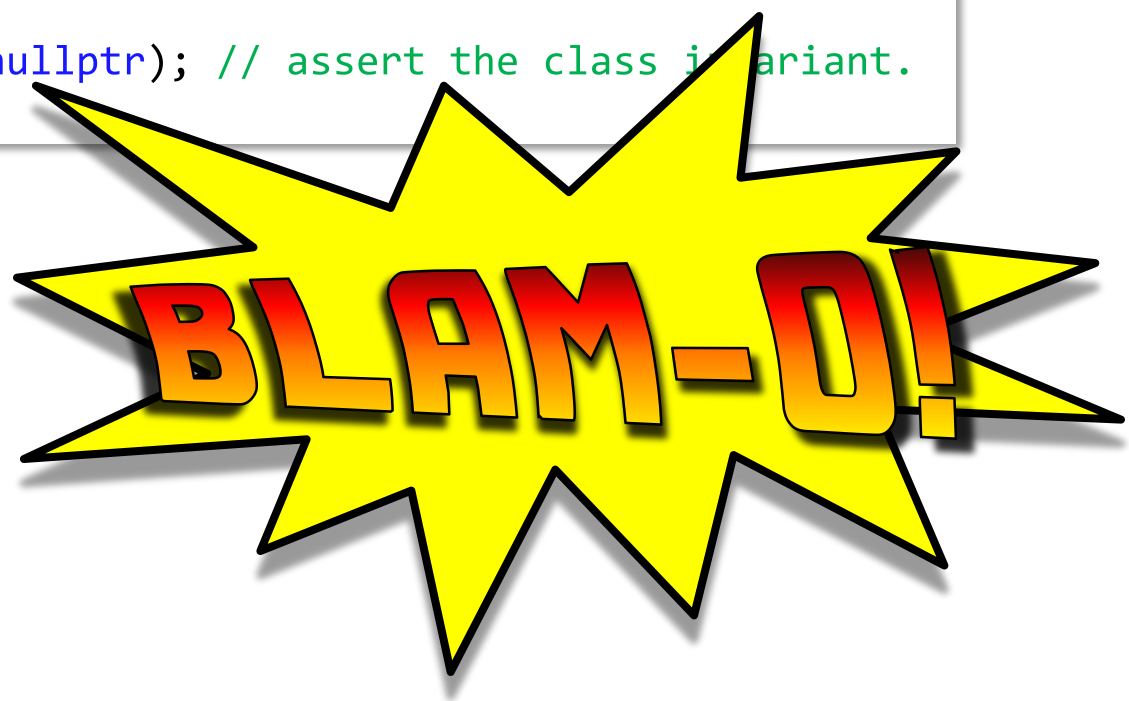
```
// Move constructor
non_null_unique_ptr(non_null_unique_ptr&& other) noexcept
  : ptr_(other.ptr_)
{
    other.ptr_ = nullptr;
}
```

Is this OK???

A Very Moving Example

Consider this code:

```
non_null_unique_ptr<int> pint{ new int(42) };
non_null_unique_ptr<int> pint2{ std::move( pint ) };

assert(pint.get() != nullptr); // assert the class i[illegible]ariant.
```



A Very Moving Example

Moved-from objects must be in a **valid but unspecified** state

A Very Moving Example

Q: Is this a better move constructor?

```
non_null_unique_ptr(non_null_unique_ptr&& other)
  : ptr_(new T(*other.ptr_))
{
    std::swap(ptr_, other.ptr_);
}
```

A: No:

- It's no different than a copy constructor!
- It can't be `noexcept` (non-ideal, but not a deal-breaker, *per se*)

A Very Moving Conclusion

Either:

1. `non_null_unique_ptr` doesn't have a natural move constructor, *or*
2. `non_null_unique_ptr` just doesn't make any sense.

Movable Types Summary

Guideline 6: The moved-from state must be part of a class's invariant.

Guideline 7: If Guideline 6 doesn't make sense, the type isn't movable.

Corollary: Every movable type must have a cheap(er)-to-construct, *valid* default state.

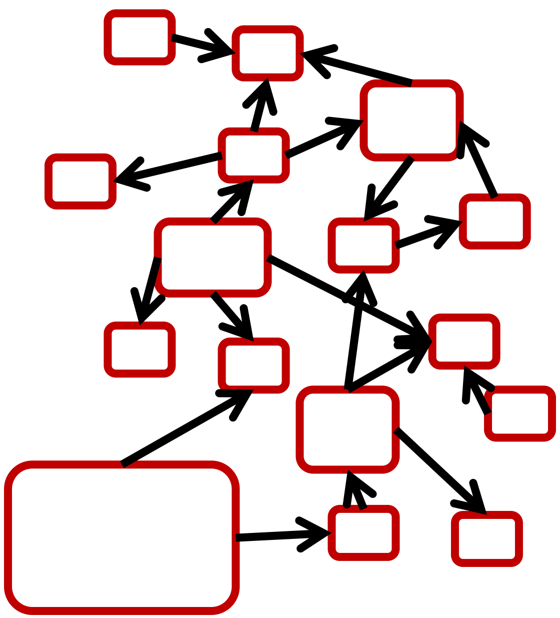
Further discussion can be found here:
http://lists.boost.org/Archives/boost/2013/01/200057.php

III. Modules

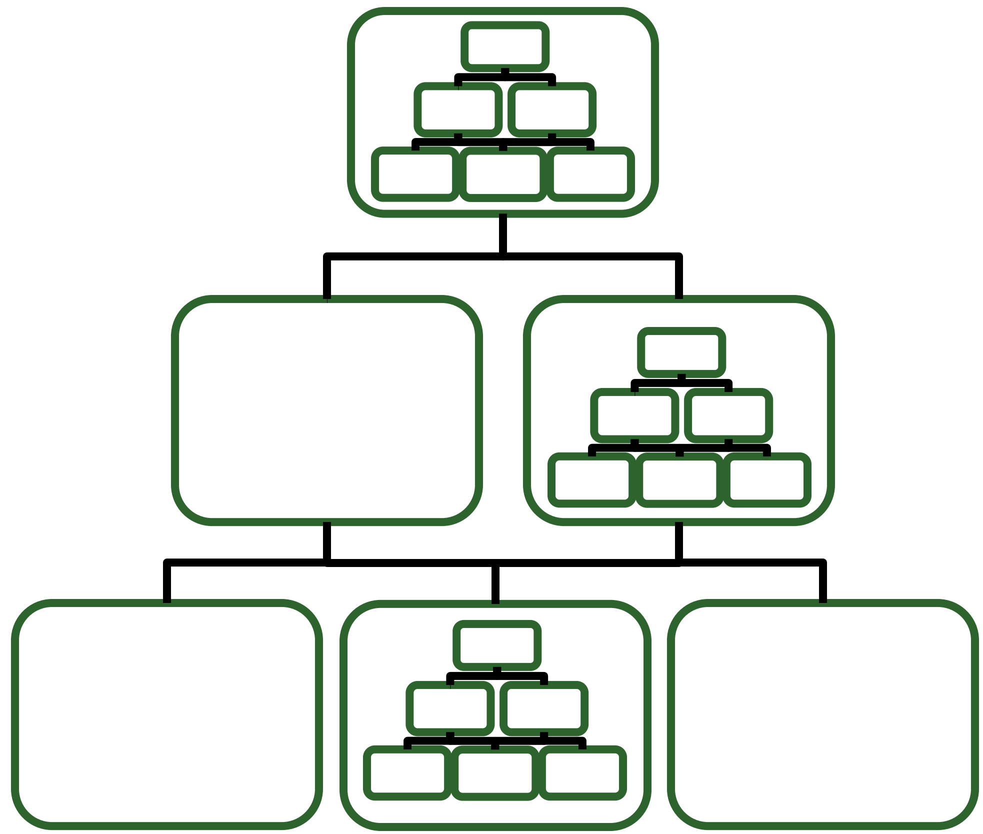
Library Design in the Large

Modules: Good and Bad

Bad



Good



Large-Scale C++11

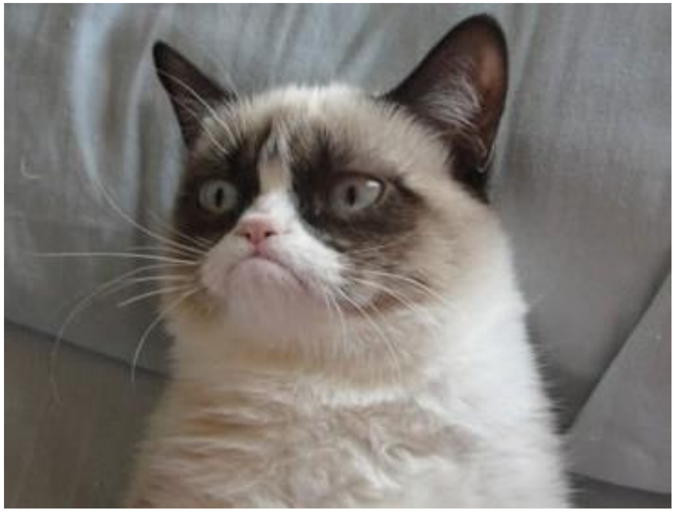
In C++11, what support is there for…

- … enforcing acyclic, hierarchical physical component dependencies?
- … decomposing large components into smaller ones?
- … achieving extensibility of components?
- … versioning (source & binary) components?

Large-scale C++11: The Bad News

- No proper modules support
- No support for dynamically loaded libraries
- No explicit support for interface or implementation versioning

…so no solution for
fragile base class



Evolving A Library

New library version with interface-breaking changes

```
namespace lib
{
  struct foo { /*...*/ };

  void bar(foo);

  template< class T >
  struct traits
  { /*...*/ };
}
```

```
namespace lib
{
  struct base {
    virtual ~base() {}
  };

  struct foo : base { /*...*/ };

  int bar(foo, int = 42);
  double bar(foo, double);

  template< class T >
  struct traits
  { /*...*/ };
}
```

New class layout

New argument/return

New overload

Evolving A Library, Take 1

New library version with interface-breaking changes

```
namespace lib
{
  // … old interface
}
```

```
namespace lib
{
  namespace v1
  {
    // … old interface
  }
}
```

```
namespace lib
{
  namespace v2
  {
    // … new interface
  }
  using namespace v2;
}
```

What's wrong with this picture?

Evolving A Library, Take 1

New library version with interface-breaking changes

```
namespace lib
{
  // … old interface
}
```

```
namespace lib
{
  namespace v1
  {
    // … old interface
  }
}
```

```
namespace lib
{
  namespace v2
  {
    // … new interface
  }
  using namespace v2;
}
```

A new namespace breaks binary compatibility

Can't specialize `lib::v2`'s templates in `lib` namespace

Evolving A Library, Take 1

```
namespace lib
{
  namespace v2
  {
    template< class T >
    struct traits
    { /*...*/ };
  }
  using namespace v2;
}
```

```
struct Mine
{};

namespace lib
{
  template<>
  struct traits< Mine >
  { /*...*/ };
}
```

ERROR! Can't specialize `lib::v2`'s templates in `lib` namespace

Evolving A Library, Take 2

New library version with interface-breaking changes

```
namespace lib
{
  // … old interface
}
```

```
namespace lib
{
  namespace v1
  {
    // … old interface
  }
}
```

```
namespace lib
{
  inline namespace v2
  {
    // … new interface
  }
}
```

Evolving A Library, Take 1

```
namespace lib
{
  inline namespace v2
  {
    template< class T >
    struct traits
    { /*...*/ };
  }
}
```

```
struct Mine
{};

namespace lib
{
  template<>
  struct traits< Mine >
  { /*...*/ };
}
```

OK!

Versioning: The Silver (In)Lining

Guideline 8: Put all interface elements in a versioning namespace *from day one*

Guideline 9: Make the current version namespace `inline`

Preventing Name Hijacking

Name Hijacking: Unintentional ADL finds the wrong overload

```
namespace rng
{
    template< class Iter >
    struct range
    {
        Iter begin_, end_;
    };

    template< class Iter >
    Iter begin( range< Iter > const & rng )
    {
        return rng.begin_;
    }
    template< class Iter >
    Iter end( range< Iter > const & rng )
    {
        return rng.end_;
    }
}
```

```
rng::range<int*> rng;

for( int i : rng )
{
    std::cout << i << std::endl;
}
```

Preventing Name Hijacking

Name Hijacking: Unintentional ADL finds the wrong overload

```
namespace tasks
{
    // Begin anything that looks like
    // a task.
    template< class TaskLike >
    void begin( TaskLike && t )
    {
        t.Begin();
    }

    struct Task
    {
        void Begin()
        { /*...*/ }
    };
};
```

```
rng::range<tasks::Task*> rng;

for( tasks::Task t : rng )
{
    t.Begin();
}
```

```
$ /usr/local/clang-trunk/bin/clang++ -c -O0 -std=gnu++11
  main.cpp -o main.o
main.cpp:43:23: error: cannot use type 'void' as an iterator
    for(tasks::Task t : p2) {}
                      ^
main.cpp:30:10: note: selected 'begin' template [with
Task = rng::range<tasks::Task *> &] with iterator type 'void'
    void begin( Task  && t )
         ^
```

Preventing Name Hijacking

Solution 1: Use a non-inline ADL-blocking namespace

```
namespace tasks
{
    // Begin anything that looks like
    // a task.
    template< class TaskLike >
    void begin( TaskLike && t )
    {
        t.Begin();
    }

    namespace block_adl_
    {
        struct Task
        {
            void Begin()
            { /*...*/ }
        };
    }
    using block_adl_::Task;
};
```

```
rng::range<tasks::Task*> rng;

for( tasks::Task t : rng )
{
    t.Begin();
}
```

Put type definitions in an ADL-blocking namespace.

Preventing Name Hijacking

Solution 2: Use global function objects instead of free functions

```
namespace tasks
{
    // Begin anything that looks like
    // a task.
    constexpr struct begin_fn
    {
       template< class TaskLike >
       void operator()( TaskLike && t ) const
       {
           t.Begin();
       }
    } begin {};

    struct Task
    {
        void Begin()
        { /*...*/ }
    };
};
```

```
rng::range<tasks::Task*> rng;

for( tasks::Task t : rng )
{
    t.Begin();
}
```

The begin object cannot ever be found by ADL

C++14 Variable Templates!

```
template<typename T>
struct lexical_cast_fn
{
    template<typename U>
    T operator()(U const &u) const
    {
        //...
    }
};

template<typename T>
constexpr lexical_cast_fn<T> lexical_cast{};

int main()
{
    lexical_cast<int>("42");
}
```

C++14 only

Ode To Function Objects

- They are never found by ADL (yay!)
- If phase 1 lookup finds an object instead of a function, ADL is disabled (yay!)
- They are first class objects
  - Easy to bind
  - Easy to pass to higher-order functions like `std::accumulate`

Preventing Name Hijacking

Guideline 10: Put type definitions in an ADL-blocking (non-inline!) namespaces and export then with a using declaration, *or…*

Guideline 11: Prefer global `constexpr` function objects over named free functions (except for documented customization points)

C++17

We need your contribution

Write a proposal!

Libraries We *Desperately* Need

File System	Boost, SG3
Databases	SOCI, SG11
Networking	SG4
Higher-Level Protocols	c++netlib
Unicode	☹
XML	☹
Ranges	SG9, *me!*
Graphics!	SG13
Concurrency	SG1

IO/Formatting	☹
Process	POCO
Date/time	Boost
Serialization	Boost
Trees	☹
Compression	POCO, Boost
Parsing	Boost
Linear Alg	☹
Crypto	POCO
…etc	

Getting Involved

- Get to know your friendly neighborhood C++ Standardization Committee:
  - http://isocpp.org/std/
  - http://www.open-std.org/jtc1/sc22/wg21/
- Participate in a Study Group:
  - https://groups.google.com/a/isocpp.org/forum/#!forumsearch/
- Get to know Boost.org:
  - http://www.boost.org
- Take a library, port to C++1[14], propose it!

Thank you

Questions?