Emulating Concepts with C++0x

Andrew Sutton

KENT STATE UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE

Software Development Laboratory
<SDML>

KENT STATE UNIVERSITY

<SDML>

Perspective Perspective

- Research interests
  - Software maintenance, evolution
  - Program comprehension
  - Library design
- PhD dissertation topic
  - Maintenance, evolution of gen. libs.
  - Tools, techniques for concepts

Research Revisited

- Rephrase gen. prog/gen libs in engineering context
  - Identifying emergent patterns in gen. lib construction [Holeman'09]
  - Role of concepts in architecture of gen. libs.
- Concepts central to these discussions
- Trying to be a user…

KENT STATE UNIVERSITY

<SDML>

Stop Gap Concepts

- How do we provide concepts…
  - Without compiler, preprocessors, metacompilers?
  - With minimal impact on existing code and practice?
- Emulation via library, idioms
  - Supports experimentation, experience

KENT STATE UNIVERSITY
<SDML>

From Idioms to Concepts

- Idioms used in GP w/C++
  - Template metaprogramming
  - Traits classes
  - Tag dispatch
  - Constrained polymorphism (SFINAE)
- Concept ≈ metafunction + trait
- Constraint ≈ SFINAE enabled
- Concept overloading ≈ tag dispatch

KENT STATE UNIVERSITY

<SDML>

Emulation Requirements

- Support “concept-like” usage
- Approximate concept features
- Be amenable to reverse engineering
- Allow experimentation with concept systems, generic libraries
- Support transformation from C++ syntax [future]

KENT STATE UNIVERSITY

<SDML>

Emulated Features (1)

- Defining concepts, requirements
  - Automatic, explicit
- Requiring operations
- Requiring, deducing type names
- Defining concept maps
- Concept Checking
  - Assert, overloading

KENT STATE UNIVERSITY

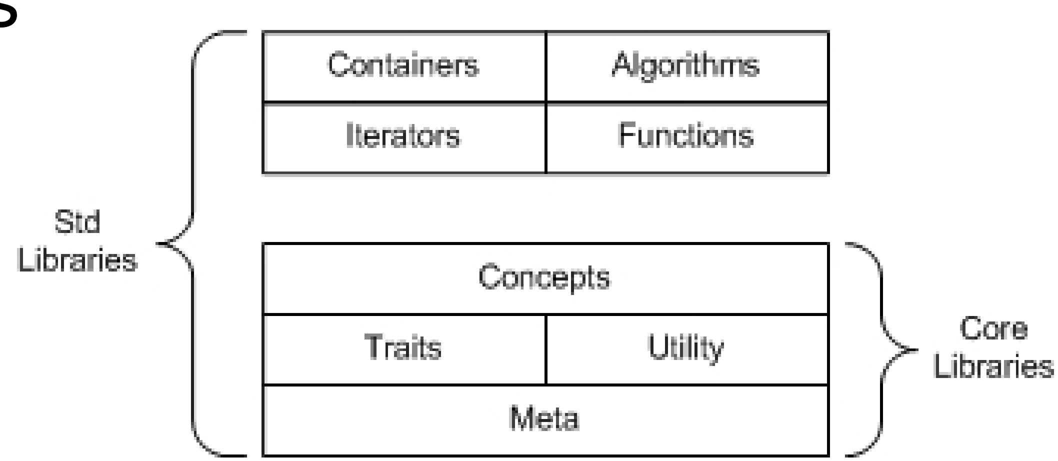
<SDML>

Emulated Features (2)

- Default overloads (provisions)
- Axioms
- Improved error messages (kind of)
- Archetypes (work in progress)

Origin C++0x Libraries

- Sandbox for C++0x experiments
- Core Libs
  - Metaprogramming, traits, concepts
- Std Libs
  - Functions, iterators, containers, algorithms



KENT STATE UNIVERSITY

<SDML>

Experiments

- Experimental concept systems
  - Concepts from n2914
  - Elements of Programming [Stepanov'09]
- Results
  - Replicate problems from WG21 pubs
  - Effective for describing semantics
  - Problems with semantics in syntax
  - Guidelines for concept design

KENT STATE UNIVERSITY

<SDML>

Afterthoughts, Questions?

- Template metaprogramming is idiomatic, abusive of notation
  - Resists comprehension, static analysis
- Do concepts deprecate template metaprogramming?
- Help concepts with lightweight, compile-time reflection?

<SDML>

Designing Concepts

KENT STATE UNIVERSITY

<SDML>

Concept Design Issues

- Aggregation of requirements
- Casual modeling
- Syntactic differentiation
- Axiomatic Concepts

KENT STATE UNIVERSITY

<SDML>

Requirement Aggregation

- Refinement complicates concept
  - Multiple, orthogonal hierarchies
  - Combinatoric explosion in number of refined concepts
- Example: Iterators
  - Traversal requirements
  - Read/Write requirements

KENT STATE UNIVERSITY

<SDML>

Casual Modeling

- A type “accidentally” models a concept without intent
  - Problem with automatic concepts
  - Concepts differentiated semantically
- Examples:
  - InputIter casually models FwdIter
  - Container casually models Range

KENT STATE UNIVERSITY

<SDML>

Casual Modeling Problem

- Automatic concepts only evaluate syntax, not semantics
  - Can lead to subtle semantic errors
- Solutions
  - Syntactic differentiation within the same concept hierarchy
  - Explicit concepts

KENT STATE UNIVERSITY

<SDML>

Syntactic Differentiation

- Disambiguate concepts that differ by semantic (axiomatic) requirements
- Example:
  - `operator++` for Input, Fwd Iterators… same syntax, different semantics
- Solution?
  - InputIterator—rename `++` to `next`

KENT STATE UNIVERSITY

<SDML>

Axiomatic Concepts

- Isolate non-checkable properties in explicit concepts
  - `MultipassIterator<X>`
    - Aggregate requirement: X is an Iterator
    - Multipass axiom
- Fwd Iterator aggregates requirement on Multipass

KENT STATE
UNIVERSITY

<SDML>

Axiomatic Concepts

- Explicit, axiomatic concepts are viral
- Type provider must affirm Multipass
  - FwdIterator is still automatically checked

KENT STATE UNIVERSITY

<SDML>

Problems and Stuff

KENT STATE UNIVERSITY

<SDML>

Emulation Problems

- Library rooted in idiomatic structures
- Preprocessor could be used…
- Fragile type traits
  - Variadic templates, forwarding seem to cause false negatives
  - Private members break traits
  - Existence of operators (., ->)

Kent State University

<SDML>

Compiler Issues?

- More compiler support for traits
  - Visibility, lifetime, virtuality?
- Strict requirements
- Injected type names
- Unbound type names

KENT STATE UNIVERSITY

<SDML>

Visibility Checks

- Private members break type traits
  - E.g., `has_constructor<T, Args...>`
    1. Look up constructor
    2. Check visibility
    3. Private? Compiler error!
- Solution?
  - More metaprogramming (ugh)
  - Compiler support?

Strict Requirements

- Traits based on SFINAE traps
  - Effectively implements checks on valid expressions, not pseudo-signatures
- Given a requirement
  - `result_type operator+(T, int)`
- Currently, this will match
  - `operator+(T, char)`
- Strict checks should cause failure…

KENT STATE UNIVERSITY

<SDML>

Injected Associated Types

- Shorthand notation for requirements injects type names
  - `template<Iterator Iter>` allows use of `Iter::reference`?
- Syntax “injects” associated types into template parameters
- Not easily approximated

KENT STATE UNIVERSITY

<SDML>

Kinds of Typenames

- *Deduced*—unconstrained, appears only as the result of an operation
- *Adapted*—specified with default, specialized by concept map
- *Unbound*—unspecified or undeduced typename within constraint

KENT STATE UNIVERSITY

<SDML>

Examples

Ex: Automatic Concept `std::Callable`

```
auto concept Callable<typename F,
                      typename... Args> {
  typename result_type;
  result_type operator()(F&&, Args...);
}
```

KENT STATE UNIVERSITY

<SDML>

Ex: Automatic Concept **`origin::Callable`**

```
template <typename F, typename... Args>
struct Callable {
  typedef call_result<F, Args...>::type
    result_type;
  typedef has_call<F, Args...>::type check;
  struct assertion {
    ~assertion() {
      static_assert(check::value,
                    “failed Callable”);
    }
  };
};
```

Ex: Concept Checking

std::find_if

```
template <typename Iter, typename Pred>
  requires InputIterator<Iter> &&
           Predicate<Pred, ...>
Iter find_if(Iter f, Iter l, Pred p) {
  ...
}
```

KENT STATE UNIVERSITY

<SDML>

Ex: Concept Checking **`origin::find_if`**

```
template <
  typename Iter, typename Pred,
  typename = typename concept_assert<
    InputIterator<Iter>,
    Predicate<Pred, ...>
  >::type>
Iter find_if(Iter f, Iter l, Pred p) {
  ...
}
```

Ex: Explicit Concepts

MultipassIterator

```
concept MultipassIterator<typename X> {
  axiom Multipass(X x, X y) {
    (x == y) => (*x == &y) && (++x == ++y);
  }
}

template <typename T>
concept_map MultipassIterator<T*> {
  // ...
};
```

KENT STATE UNIVERSITY

<SDML>

Ex: Explicit Concept **origin::MultipassIterator**

```
template <typename X>
struct MultipassIterator {
  typedef True<false>::check check;
  typedef True<false>::assertion assertion;
};

template <typename T>
struct MultipassIterator<T*> {
  typedef True<true>::check check;
  typedef True<true>::assertion assertion;
};
```

Ex: Axioms

MultipassIterator

```
namespace MultipassIterator_ {
  template <typename X>
  void Multipass(X x, X Y) {
    if(x == y) {
      assert((*x == *y) && (++x == ++y))
    }
  }
}
```

<SDML>

Ex: Refinement, Aggregation

std::Semiregular

```
concept Semiregular<typename T>
  : CopyConstructible<T>, CopyAssignable<T>
{
  requires SameType<
    CopyAssignable<T>::result_type, T&
  >;
}
```

KENT STATE UNIVERSITY

<SDML>

Ex: Refinement, Aggregation

origin::Semiregular

```
template <typename T>
struct Semiregular
  : CopyConstructible<T>, CopyAssignable<T>
{
  typedef typename concept_check<
    CopyConstructible<T>, CopyAssignable<T>,
    SameType<
      CopyAssignable<T>::result_type, T&
    >
  >::type check;
}
```

Ex: Associated Types

Graph

```
concept Graph<typename G> {
  typename vertex_desc = G::vertex_desc
}

template <typename G>
  requires Graph<G>
void bfs(G const& g) {
  typename Graph<G>::vertex_desc s =
    begin(g.vertices());
}
```

KENT STATE UNIVERSITY

<SDML>

Ex: Associated Types

Graph

```
template <typename G>
struct Graph<typename G> {
  typedef typename vertex_desc_<G>::type
    vertex_desc;
  typedef typename has_vertex_desc_<G>::type
    check;
}
```

Ex: Associated Types

Graph

```
template <
  typename G, requires(Graph<G>)>
void bfs(G const& g) {
  typename Graph<G>::vertex_desc s =
    begin(g.vertices());
}
```

Ex: Default Overloads `std::EqualityComparable`

```
concept EqualityComparable<typename X> {
  bool operator==(X const&, X const&);
  bool operator!=(X const& x, X const& y) {
    return !(x == y);
  }
}
```

Ex: Default Overloads

origin::EqualityComparable

```
template <typename T>
struct EqualityComparable<typename T> {
  // ...
}
namespace EqualityComparable_ {
  template <typename T>
  bool operator!=(T const& x, T const& y) {
    return !(x == y);
  }
};
```

KENT STATE UNIVERSITY

<SDML>

Ex: Using Default Overloads **origin::equal_to**

```
template <
  typename T,
  requires(EqualityComparable<T>)>
struct not_equal_to {
  bool operator()(T const& x,
                  T const& y) const
  {
    using namespace EqualityComparable_;
    return x != y;
  }
};
```

KENT STATE UNIVERSITY

<SDML>

Ex: Concept Overloading
`std::distance`

```
template <typename Iter>
  requires InputIterator<Iter>
int distance(Iter f, Iter l) { ... }

template <typename Iter>
  requires RandomAccessIterator<Iter>
int distance(Iter f, Iter l) { ... }
```

Ex: Concept Overloading `origin::distance`

```
template <
  typename Iter,
  requires(InputIterator<Iter>)>
int distance(Iter f, Iter l,
  typename concept_enable<
    InputIterator<Iter>,
    Not<RandomAccessIterator<Iter>>
  >::type* = nullptr)
{ ... }
```

<SDML>

Ex: Concept Overloading
origin::distance

```
template <
  typename Iter,
  requires(InputIterator<Iter>)>
int distance(Iter f, Iter l,
  typename concept_enable<
    RandomAccessIterator<Iter>
  >::type* = nullptr)
{ ... }
```

Kent State University

<SDML>

Origin.Traits: SFINAE Trap

`call`

```
template <typename F, typename... Args>
auto call(F&& f, Args&&... args)
  -> decltype(f(args...));

lookup_failure call(...);
```

Origin.Traits
`call_result`

```
template <typename F, typename... Args>
struct call_result {
  typedef decltype(
    call(value<F>(), value<Args>()...)
  ) type;
};
```

KENT STATE UNIVERSITY

<SDML>

Origin.Traits
`is_callable`

```
template <typename F, typename... Args>
struct is_callable
  : lookup_succeeded<
      typename call_result<F, Args...>::type
  >::type
{ };
```