

# Simulating Type Classes in C++

Kyle Ross  
kyle@cs.chalmers.se

Chalmers University of Technology  
29 September 2004

# Simulating Type Classes in C++

J.Järvi, J.Willcock, and A.Lumsdaine. "Concept-Controlled Polymorphism". In F.Pfennig and Y.Smaragdakis, editors, *Generative Programming and Component Engineering*, volume 2830 of LNCS, pages 228--244, September 2003. Springer Verlag.

# Type Classes

"Type classes may be thought of as a kind of bounded quantifier, limiting the types that a type variable may instantiate to ... Type classes also may be thought of as a kind of abstract data type. Each type class specifies a collection of functions and their types but not how they are to be implemented." - Wadler & Blott, 1988

"Type classes were first introduced to Haskell to control ad-hoc polymorphism ... Type classes are closely related to concepts in generic programming. Like concepts, type classes encapsulate requirements for a type or for a set of types collectively." - Järvi et al., 2003

# Type Classes

"Type classes may be thought of as a kind of bounded quantifier, limiting the types that a type variable may instantiate to ... Type classes also may be thought of as a kind of abstract data type. Each type class specifies a collection of functions and their types but not how they are to be implemented." - Wadler & Blott, 1988

"Type classes were first introduced to Haskell to control ad-hoc polymorphism ... Type classes are closely related to concepts in generic programming. Like concepts, type classes encapsulate requirements for a type or for a set of types collectively." - Järvi et al., 2003

```
class Eq a where
  (==) :: a -> a -> Bool
  ...

reflexive_equality x = x == x
(inferred to have type reflexive_equality :: forall a. (Eq a) => a -> Bool)

reflexive_equality True
reflexive_equality "hello"
reflexive_equality 3
(all return True)
```

# Assumptions

Haskell = Haskell98 + Glasgow extensions

C++ code assumes `using namespace std;` and `using namespace boost;`

C++ code relies on the presence of appropriate `#include` directives

# C++ Template Mechanism

```
template <typename T> int foo (T t1, T t2)
{ return t1 + t2; }

foo (1, 2);
instantiates foo<int> :: (T, T) -> int [T = int]
returns 3
```

# C++ Template Mechanism

```
template <typename T> int foo (T t1, T t2)
{ return t1 + t2; }
```

```
foo (1, 2);
instantiates foo<int> :: (T, T) -> string [T = int]
returns 3
```

```
template <int n> int moo()
{ return n; }
```

```
moo<5> ();
instantiates moo<5> :: () -> int
returns 5
```

# C++ Template Mechanism

```
template <typename T> int foo (T t1, T t2)
{ return t1 + t2; }
```

```
foo (1, 2);
instantiates foo<int> :: (T, T) -> string [T = int]
returns 3
```

```
template <int n> int moo()
{ return n; }
```

```
moo<5> ();
instantiates moo<5> :: () -> int
returns 5
```

```
template <bool c> int boo()
{ if (c) return 3;
  return 5 ;
}
```

```
boo<true> ();
instantiates boo<true> :: () -> int
returns 3
```

# C++ Template Mechanism

```
template <typename T> int foo (T t1, T t2)
{ return t1 + t2; }
```

```
foo (1, 2);
instantiates foo<int> :: (T, T) -> string [T = int]
returns 3
```

```
template <int n> int moo()
{ return n; }
```

```
moo<5> ();
instantiates moo<5> :: () -> int
returns 5
```

```
template <bool c> int boo()
{ if (c) return 3;
  return 5 ;
}
```

```
boo<true> ();
instantiates boo<true> :: () -> int
returns 3
```

```
bool x;
boo<x> ();
causes a compile-time error
because template arguments must be known statically
"error: non-constant `x' cannot be used as template argument" (g++)
```

# Limitations on Template Parameters

```
type, int, bool valid  
template <typename T> void foo () { body }  
template <int n> void foo () { body }  
template <bool c> void foo () { body }
```

# Limitations on Template Parameters

type, int, bool valid

```
template <typename T> void foo () { body }
```

```
template <int n> void foo () { body }
```

```
template <bool c> void foo () { body }
```

anything else is an error

```
template <double d> void foo () { body }
```

causes a compile-time error

```
"error: `double' is not a valid type for a template constant" (g++)
```

# Template Specialisation

```
primary (unspecialised) function template foo
template <typename T> string foo (T)
{ return "primary"; }
```

```
int specialisation function template foo
template <> string foo<int> (int)
{ return "int specialisation"; }
```

# Template Specialisation

```
primary (unspecialised) function template foo
template <typename T> string foo (T)
{ return "primary"; }
```

```
int specialisation function template foo
template <> string foo<int> (int)
{ return "int specialisation"; }
```

```
foo <bool>
foo <double>
foo <my_type>
all refer to the primary template
```

```
foo <int>
refers to the specialisation
```

```
foo (true);
instantiates foo<T> :: T -> string [T = bool]
returns "primary"
```

```
foo (3);
instantiates foo<int> :: int -> string
returns "int specialisation"
```

# (Run-Time) Conditionals with Templates

```
if n != 3 then return false
template <int n> int foo ()
{ return 0; }
```

```
if n == 3 then return true
template <> int foo<3> ()
{ return 1; }
```

```
foo<2> ();
instantiates foo<n> :: () -> int [n = 2]
returns 0 AT RUN TIME
```

```
foo<3> ();
instantiates foo<3> :: () -> int
returns 1 AT RUN TIME
```

# Static Conditionals with Templates

```
meta_if_int :: bool -> int -> int -> int (general true case)
template <bool c, int t, int f> struct meta_if_int
{ enum { ret = t }; };
```

```
meta_if_int (specialised false case)
template <int t, int f> struct meta_if_int<false, t, f>
{ enum { ret = f }; };
```

```
meta_if_int<2 == 3, 1, 0>::ret;
returns 0 AT COMPILE TIME
```

```
meta_if_int<3 == 3, 1, 0>::ret;
returns 1 AT COMPILE TIME
```

# Static Conditionals with Templates

```
meta_if_int :: bool -> int -> int -> int (general true case)
template <bool c, int t, int f> struct meta_if_int
{ enum { ret = t }; };
```

```
meta_if_int (specialised false case)
template <int t, int f> struct meta_if_int<false, t, f>
{ enum { ret = f }; };
```

```
meta_if_int<2 == 3, 1, 0>::ret;
returns 0 AT COMPILE TIME
```

```
meta_if_int<3 == 3, 1, 0>::ret;
returns 1 AT COMPILE TIME
```

```
meta_if_type :: bool -> * -> * -> * (general true case)
template <bool c, typename T, typename F> struct meta_if_type
{ typedef T ret; };
```

```
meta_if_type (specialised false case)
template <typename T, typename F> struct meta_if_type<false, T, F>
{ typedef F ret; };
```

```
meta_if_type<1 != 2, int, bool>::ret x;
x is declared as an int because 1!=2 evaluates to true
```

# Recursive Computation with Templates

```
factorial :: int -> int (recursion case)
template <int n> struct factorial
{ enum { ret = n * factorial<n - 1>::ret }; };
```

```
factorial (base (termination) case)
template <> struct factorial<0>
{ enum { ret = 1 }; };
```

```
factorial<7>::ret;
(statically) computes 7!
```

# Recursive Computation with Templates

```
factorial :: int -> int (recursion case)
template <int n> struct factorial
{ enum { ret = n * factorial<n - 1>::ret }; };
```

```
factorial (base (termination) case)
template <> struct factorial<0>
{ enum { ret = 1 }; };
```

```
factorial<7>::ret;
(statically) computes 7!
```

"When the compiler sees `factorial<7>::ret`, it instantiates the structure template `factorial<n>` for `n=7`. This involves initialising the enumerator `ret` with `7*factorial<6>::ret`. At this point, the compiler also has to instantiate `factorial<6>::ret`. The latter, of course, will require instantiating `factorial<n>::ret` for `n=5`, `n=4`, `n=3`, `n=2`, `n=1`, and `n=0`. The last initialisation matches the template specialisation for `n=0`, wherein `ret` is directly initialised to 1. This template specialisation terminates the recursion." - Czarnecki & Eisenecker, 2000

# The `enable_if` Mechanism

```
general (true) case defines type "type", defaulting to "void"  
template <bool B, typename T = void> struct enable_if_c  
{ typedef T type; };
```

```
specialised false clause defines no "type" member  
template <typename T> struct enable_if_c<false, T>  
{};
```

# The enable\_if Mechanism

```
general (true) case defines type "type", defaulting to "void"  
template <bool B, typename T = void> struct enable_if_c  
{ typedef T type; };
```

```
specialised false clause defines no "type" member  
template <typename T> struct enable_if_c<false, T>  
{};
```

```
template <typename T> typename enable_if_c<true, T>::type id (T t)  
{ return t; }  
id<T> :: T -> enable_if_c<true, T>::type
```

```
id (3);  
causes instantiation of id<T> :: T -> enable_if_c<true, T>::type [T = int]
```

# The enable\_if Mechanism

general (true) case defines type "type", defaulting to "void"  
template <bool B, typename T = void> struct enable\_if\_c  
{ typedef T type; };

specialised false clause defines no "type" member  
template <typename T> struct enable\_if\_c<false, T>  
{};

template <typename T> typename enable\_if\_c<true, T>::type id (T t)  
{ return t; }  
id<T> :: T -> enable\_if\_c<true, T>::type

id (3);  
causes instantiation of id<T> :: T -> enable\_if\_c<true, T>::type [T = int]

template <typename T> typename enable\_if\_c<false, T>::type nope (T t)  
{ return t; }  
un-callable because enable\_if\_c<false, T>::type is undefined for any type T

nope (3);  
tries to instantiate nope<T> :: T -> enable\_if\_c<false, T>::type [T = int]  
this fails because enable\_if\_c<false, int>::type is undefined  
produces a compile-time error  
"error: no matching function for call to `nope(int)'" (g++)

# "Substitution Failure is Not an Error"

"principle of SFINAE"

"If an invalid argument type or return type is formed during the instantiation of a function template, the instantiation is removed from the overload resolution set [the set of 'valid functions'] instead of causing a compilation error." - Järvi et al., 2003

# The Show Type Class

C++:

```
template <typename A, typename enable
    = void> struct Show_traits
{ static const bool conforms =
    false; };
```

...

Haskell:

```
class Show a where
    ...
```

# The Show Type Class

C++:

```
template <typename A, typename enable
    = void> struct Show_traits
{ static const bool conforms =
    false; };
```

```
template <typename A> struct Show
{ BOOST_STATIC_ASSERT
    (Show_traits<A>::conforms);
  static string show (const A& t)
  { return Show_traits<T>::show (t);
  }
};
```

Haskell:

```
class Show a where
```

```
    show :: a -> String
(simplified quite a bit)
```

# Using Show

## C++:

```
template <> struct Show_traits<bool>
{ static const bool conforms = true;
  static string show (bool t)
  { return t ? "True" : "False"; }
} ;
```

```
template <> struct Show_traits<int>
{ static const bool conforms = true;
  static string show (int t)
  { stringstream ss;
    ss << t;
    return ss.str ();
  }
};
```

```
template <typename T> typename
  enable_if_c<Show_traits<T>::conforms, void>::type print (const T&
  t)
{ cout << Show<T>::show(t) <<
  endl; }
```

## Haskell:

```
instance Show Bool where
  show True = "True"
  show False = "False"
```

```
instance Show Int where
  show n = Int -> String primitive
```

```
print x = putStrLn (show x)
(print :: forall a. (Show a) => a ->
  IO () inferred)
```

# Trying to Abuse Show

```
Show<double>::show (3.14159);
```

causes a compile-time error  
because double is not a member of the Show type class

```
"In instantiation of `Show<double>`:  
error: invalid application of `sizeof' to an incomplete type  
In static member function `static std::string  
    Show<T>::show(const T&) [with T = double]':  
error: 'struct Show_traits<double, void>' has no member named 'show'" (g++)
```

# models and DECLARE\_CONFORMS

```
if T models Show then list<T> does (instance Show T => Show [T])
template <typename T> struct Show_traits<list<T>, typename
    enable_if_c<Show_traits<T>::conforms, void>::type>
{ definition };

template <typename T, template <typename U> class C> struct models
{ typedef typename enable_if_c<C<T>::conforms, void>::type conforms; };

if T models Show then list<T> does (instance Show T => Show [T])
template <typename T> struct Show_traits<list<T>, typename models<T,
    Show_traits>::conforms>
{ definition };

yes, I conform!
#define DECLARE_CONFORMS static const bool conforms = true

no, I don't conform!
#define DENY_CONFORMS static const bool conforms = false
```

# Showing Lists

## C++:

```
template <typename T> struct
    Show_traits<list<T>, typename
        models<T, Show_traits>::conforms>
{ DECLARE_CONFORMS;
  static string show (list<T> t)
  { if (t.empty ())
      return "[]";
    string result = "[" +
    Show<T>::show (t.front ());
    for (typename
        list<T>::const_iterator i = ++
        (t.begin ()); i != t.end (); ++i)
        result += ("," + Show<T>::show
        (*i));
    return result + "];"
  }
};
```

## Haskell:

```
instance Show a => Show [a] where
  show x = "[" ++ (show_helper x) ++
    "]"
  where
  show_helper [] = ""
  show_helper [x] = show x
  show_helper (x:xs) = (show x) ++
    "," ++
    (show_helper xs)
```

# Concepts in Generic Programming

"Many commonly-used operations on sequences ... can be performed with algorithms that depend for their correct and efficient operation only on a few basic operations. By expressing algorithms in terms of these basic access operations ... we permit a single expression of the algorithm to be used with any concrete representation of the container." - Musser & Stepanov, 1993

"In generic programming the term **concept** is used to mean a set of abstractions (typically types) whose membership is defined by a set of requirements. The expression of requirements in a concept is referred to as a **concept description**. Concept requirements may be semantic as well as syntactic." - Willcock et al., 2004

# EqualityComparable Concept

## Description:

A type is `EqualityComparable` if objects of that type can be compared for equality using `operator==`, and if `operator==` is an equivalence relation.

## Valid expressions:

Equality: `x == y` returns (a type convertible to) `bool`

Inequality: `x != y` returns (a type convertible to) `bool`

## Semantics:

`&x == &y` implies `x == y` (identity)

`x == x` (reflexivity)

`x == y` implies `y == x` (symmetry)

`x == y` and `y == z` implies `x == z` (transitivity)

`x != y` is equivalent to `!(x == y)`

(SGI STL documentation)

# EqualityComparable Type Class

C++:

```
template <typename T, typename enable = void>
    struct EC_traits
{ DENY_CONFORMS; };
```

...

Haskell:

```
class EC ec where
    ...
```

# EqualityComparable Type Class

C++:

```
template <typename T, typename enable = void>
  struct EC_traits
{ DENY_CONFORMS; };

template <typename T> struct EC
{ BOOST_STATIC_ASSERT
  (EC_traits<T>::conforms);
  static bool equality (const T& t1, const T&
    t2)
  { return EC_traits<T>::equality (t1, t2); }
  static bool inequality (const T& t1, const
    T& t2)
  { return EC_traits<T>::inequality (t1, t2);
  }
};
```

Haskell:

```
class EC ec where
  equality :: ec -> ec -> Bool
  inequality :: ec -> ec -> Bool
  (yes, this is Eq with funny names)
```

# EqualityComparable Type Class

C++:

```
template <typename T, typename enable = void>
    struct EC_traits
{ DENY_CONFORMS; };
```

```
template <typename T> struct EC
{ BOOST_STATIC_ASSERT
  (EC_traits<T>::conforms);
  static bool equality (const T& t1, const T&
    t2)
  { return EC_traits<T>::equality (t1, t2); }
  static bool inequality (const T& t1, const
    T& t2)
  { return EC_traits<T>::inequality (t1, t2);
  }
};
```

```
template <typename T> struct EC_defaults
{ static bool inequality (const T& t1, const
  T& t2)
  { return !EC<T>::equality (t1, t2); }
};
```

Haskell:

```
class EC ec where
```

```
    equality :: ec -> ec -> Bool
```

```
    inequality :: ec -> ec -> Bool
```

(yes, this is Eq with funny names)

```
inequality x y = not (equality x y)
```

# EqualityComparable Type Class

C++:

```
template <typename T, typename enable = void>
  struct EC_traits
  { DENY_CONFORMS; };
```

```
template <typename T> struct EC
{ BOOST_STATIC_ASSERT
  (EC_traits<T>::conforms);
  static bool equality (const T& t1, const T&
    t2)
  { return EC_traits<T>::equality (t1, t2); }
  static bool inequality (const T& t1, const
    T& t2)
  { return EC_traits<T>::inequality (t1, t2); }
};
```

```
template <typename T> struct EC_defaults
{ static bool inequality (const T& t1, const
  T& t2)
  { return !EC<T>::equality (t1, t2); }
};
```

```
template <> struct EC_traits<bool> : public
  EC_defaults<bool>
{ DECLARE_CONFORMS;
  static bool equality (bool t1, bool t2)
  { return t1 == t2; }
};
```

Haskell:

```
class EC ec where
```

```
  equality :: ec -> ec -> Bool
  inequality :: ec -> ec -> Bool
  (yes, this is Eq with funny names)
```

```
inequality x y = not (equality x y)
```

```
instance EC Bool where
  equality x y = x == y
```

# EqualityComparable Type Class

## C++:

```
template <typename T> struct EC_defaults
{ static bool equality (const T& t1, const T&
  t2)
  { return !EC<T>::inequality (t1, t2); }
  static bool inequality (const T& t1, const
    T& t2)
    { return !EC<T>::equality (t1, t2); }
};
```

```
template <> struct EC_traits<bool> : public
  EC_defaults<bool>
{ DECLARE_CONFORMS;
  static bool equality (bool t1, bool t2)
  { return t1 == t2; }
};
```

## Haskell:

```
class EC ec where
  equality :: ec -> ec -> Bool
  inequality :: ec -> ec -> Bool
  equality x y = not (inequality x y)
  inequality x y = not (equality x y)

instance EC Bool where
  equality x y = x == y
```

# Ord Type Class

```
Haskell:  
class (Eq a) => Ord a where  
    less :: a -> a -> Bool  
(simplified quite a bit)
```

# Ord Type Class

C++:

```
template <typename T, typename
    enable = void> struct Ord_traits
{ DENY_CONFORMS; };
```

```
template <typename T> struct Ord
{ BOOST_STATIC_ASSERT
  (EC_traits<T>::conforms);
  BOOST_STATIC_ASSERT
  (Ord_traits<T>::conforms);
  static bool less (const T& t1,
    const T& t2)
  { return Ord_traits<T>::less (t1,
    t2); }
};
```

Haskell:

```
class (Eq a) => Ord a where
  less :: a -> a -> Bool
(simplified quite a bit)
```

# An Algebraic Data Type: RoseTree

C++:

```
enum RoseTreeConstructor {Leaf, Branch};

struct adt_exception { definition };

template <typename T> struct RoseTree
{ RoseTree (RoseTreeConstructor mc)
  :my_constructor (mc) {}

  bool match(RoseTreeConstructor c) const
  {return my_constructor == c;}

  T& value_Leaf ()
  { if (!match (Leaf))
    throw adt_exception message;
    return leaf_value;
  }

  list<RoseTree<T> >& value_Branch ()
  { if (!match (Branch))
    throw adt_exception message;
    return branch_value;
  }

protected:
  RoseTreeConstructor my_constructor;
  T leaf_value;
  list<RoseTree<T> > branch_value;
};
```

Haskell:

```
data RoseTree a = Leaf a
                | Branch [RoseTree a]
```

# Coerce Type Class

(from Wadler 1988)

## C++:

```
template <typename T, typename U,  
         typename enable = void> struct  
    Coerce_traits  
{ DENY_CONFORMS; };  
  
template <typename T, typename U>  
    struct Coerce  
{ BOOST_STATIC_ASSERT  
    ((Coerce_traits<T, U>::conforms));  
    static U coerce (const T& t)  
    { return Coerce_traits<T,  
    U>::coerce (t); }  
};
```

(continued on next slide)

## Haskell:

```
class Coerce a b where  
    coerce :: a -> b  
  
instance Coerce (RoseTree a) [a]  
    where  
        coerce (Leaf x) = [x]  
        coerce (Branch []) = []  
        coerce (Branch (xs)) = (concat  
        (map coerce xs))  
  
instance Coerce [a] (RoseTree a)  
    where  
        coerce [] = Branch []  
        coerce [x] = Leaf x  
        coerce (x:xs) = Branch [Leaf x,  
        (coerce xs)]
```

# Coerce Type Class (continued)

C++ (continued)

```
template <typename T> struct Coerce_traits<RoseTree<T>, list<T> >
{ DECLARE_CONFORMS;
  static list<T> coerce (const RoseTree<T>& t)
  { list<T> result;
    if (t.match (Leaf)) result.push_back (t.value_Leaf ());
    if (t.match (Branch))
    { list<RoseTree<T> > children = t.value_Branch ();
      for (typename list<RoseTree<T> >::const_iterator i = children.begin (); i !=
children.end (); ++i)
        { list<T> child_result = Coerce_traits<RoseTree<T>, list<T> >::coerce (*i);
          copy (child_result.begin (), child_result.end (), back_inserter (result));}
    }
  return result;};};

template <typename T> struct Coerce_traits<list<T>, RoseTree<T> >
{ DECLARE_CONFORMS ;
  static RoseTree<T> coerce (const list<T>& t)
  { if (t.empty ()) return RoseTree<T> (Branch);
    list<T> temp = t ;
    return helper (temp) ;}

protected: static RoseTree<T> helper (list<T>& t)
  { if (t.size () == 1)
    { RoseTree<T> result (Leaf) ; result.value_Leaf () = t.front () ; return result;
    }
    RoseTree<T> leaf_node (Leaf), parent_node (Branch) ;
    leaf_node.value_Leaf () = t.front () ; t.pop_front () ;
    parent_node.value_Branch ().push_back (leaf_node) ; parent_node.value_Branch ().
push_back (helper (t)) ;
    return parent_node ;};};
```

# Function Name Re-Use

```
template <typename T, typename enable = void> struct A_traits  
{ DENY_CONFORMS; };
```

```
template <typename T> struct A  
{ BOOST_STATIC_ASSERT (A_traits<T>::conforms);  
  static bool foo (T t)  
  { return A_traits<T>::foo (t); } };
```

```
template <typename T, typename enable = void> struct B_traits  
{ DENY_CONFORMS; };
```

```
template <typename T> struct B  
{ BOOST_STATIC_ASSERT (B_traits<T>::conforms);  
  static bool foo (T t)  
  { return B_traits<T>::foo (t); } };
```

```
template <> struct A_traits<int>  
{ DECLARE_CONFORMS ;  
  static bool foo (int)  
  { return false; } };
```

```
template <> struct B_traits<int>  
{ DECLARE_CONFORMS ;  
  static bool foo (int)  
  { return false; } };
```

```
A<int>::foo (3) ;  
B<int>::foo (3) ;
```

# Evaluation

- from Järvi et al., 2003:
  - conformance largely un-checked
  - generic functions must explicitly list type class requirements
  - multiple type classes can require functions with the same name
  - conditions for membership more flexible than in Haskell
- my observations:
  - generic programming needs concept checking (and this is a step in that direction)
  - it's a neat trick, but it feels like a huge hack, and the syntax is too cumbersome to be usable in real programs

# References

- "Concept-Controlled Polymorphism" - J.Järvi et al., 2003
- "How to Make Ad-Hoc Polymorphism Less Ad-Hoc" - P.Wadler & S.Blott, 1988
- "Algorithm-Oriented Generic Libraries" - D.R.Musser & A.A.Stepanov, 1993
- "A Formalisation of Concepts for generic Programming" - J.Willcock et al., 2004
- "Generative Programming: Methods, Tools, and Applications" - K.Czarnecki & U.Eisenecker, 2000
- "Haskell 98 Language and Libraries: The Revised Report" - S.P.Jones et al., 2002
- The SGI Standard Template Library - <http://www.sgi.com/tech/stl>
- "International Standard, Programming Languages - C++" - ISO/IEC:14882, 1998