Types, classes and concepts.

++it Italian C++ Community

Meetup

28 Giugno 2014, Milano

Nicola Bonelli :: nicola@pfq.io

Outline

- Rationale, Haskell types and type classes
- Type classes in C++?
- From Concepts to Concepts lite (C++14)
  - Accu 2013 Sutton constraints
- Type classes in C++
  - type class
  - type class instance
  - type class constraints

Haskell types and type classes

- Haskell is a pure functional language with support of parametric polymorphism and type classes.

- Parametric polymorphism advantages:
    - more intuitive that subtyping (no relationship among types)
    - efficient: static polymorphism
        - existential quantification enables runtime polymorphism (~ C++ inheritance + virtual fun).

- Type classes are collection of functions that represents the interface to which certain types must adhere.
    - New and existing types can be added as ***instances*** of type classes.

Typeclass: a simple example

Class declaration

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
    x == y = not (x /= y)
    x /= y = not (x == y)
```

Data declaration

```
data Maybe a = Nothing | Just a
```

Instance declaration

```
instance (Eq a) => Eq (Maybe a) where
    Nothing == Nothing = True
    Just x  == Just y  = x == y
    _ == _   = False
```

What about C++?

- Parametric polymorphism is enabled by template (generic programming)
- Overloading is fundamental
  - semantically equivalent operations with the same name
- Abuse of inheritance
- Functions are already first-class citizen (high-order functions).
  - pointer to functions, lambdas and std::function
  - C++ functors (not to confuse with functors from category theory)

Type classes in C++?

- ADL, ad-hoc polymorphism (overloading) and template are not enough!
- Functions need a support to restrict the types they take, to improve the quality of error messages.
  - very long errors with no a clear location.
- Constraining types by annotations is error prone.
  - comments in source codes are not a viable solution!

Concepts!?

- C++ Concepts were proposed as an extension (to be included into C++11) a construct that specify what a type must provide
    - to codify formal properties of types (also semantically).
    - to improve compiler diagnostics.

- In addition, *concept map* is a mechanism to bind types to concepts.
    - Types can then be converted into an interface that generic functions can utilize.

- Concepts were dropped due to some concerns.
    - Mainly because they were “not ready” for C++11...

Concepts Lite!

- Simplified implementation as **template constraints** for functions and classes template (in C++14 Technical specification).
  - Semantic check and concept map are not implemented.

- Used to check template arguments at the point of use.

- No runtime overhead.

Constraints

- A constraint is a **constexpr** function template
  - Can use type traits, call other constexpr functions
- Constraints check *syntactic requirements*
  - Is this expression valid for objects of type T?
  - Is the result type of an expression convertible to U?

Constraining template arguments

Constrain template arguments with predicates:

```
template<Sortable C>
void sort(C& container);
```

Constraints are just constexpr function templates:

```
template<typename T>
constexpr bool Sortable() {
    return ...;
}
```

Concepts Lite

- Concept lite are intended to:
  - specify properties and constraints of generic types.
  - extend overloading (most constrained overload wins).
  - select the most suitable class specialization.
  - … enable_if is somehow deprecated.

- They do not represent the C++ version of type classes per se.
  - constraint are only half of the problem.
    - *concept map* will be not available, Stroustrup says.

Type class recipe ?!?!

To implement a type class we need:

- static_assert or **concepts lite**
- constraint: constexpr predicate
- definition of the typeclass
- some SFINAE tricks
- a pinch of template metaprogramming

Typeclass: multiset #1

```
template <typename ...Ts> struct multiset
{
	static constexpr size_t size = sizeof...(Ts);
};

template <typename Set1, typename Set2> struct concat;

template <typename Set1, typename Set2>
using concat_t = typename concat<Set1, Set2>::type;

template <typename ...Ts, typename ...Vs>
struct concat<multiset<Ts...>, multiset<Vs...>>{
	using type = multiset<Ts..., Vs...>;
};
```

Typeclass: multiset #2

```
template <typename T, typename Set> struct erase;

template <typename T, typename V, typename ...Ts>
struct erase<V, set<T, Ts...>>
{
	using type = concat_t<set<T>, erase_t<V, set<Ts...>> >;
};
template <typename T, typename ...Ts>
struct erase<T, set<T, Ts...>> {
	 using type = set<Ts...>;
};
template <typename T>
struct erase<T, set<>> {
	 using type = set<>;
};
```

Typeclass: multiset #3

```
template <typename Set1, typename Set2> struct subtract;

template <typename ...Ts>
struct subtract< set<Ts...>, set<> > {
        using type = set<Ts...>;
};
template <typename V, typename ...Ts, typename ...Vs>
struct subtract< set<Ts...>, set<V, Vs...> > {
      using type = subtract_t< erase_t<V, set<Ts...>>, set<Vs...> >;
};
```

```
template <typename Set1, typename Set2>
constexpr bool equal_set()
{ return (Set1::size == Set2::size) && std::is_same<subtract_t<Set1, Set2>, set<> >::value;}
```

C++ type class

A typeclass is defined as a multiset of function types:

```
template <typename ...Fs>
using typeclass = type::multiset<Fs...>;
```

User typeclass declaration:

```
template <typename T>
struct Show
{
    static std::string show(T const &);
    static std::string showList(std::list<T> const &);

    using type = typeclass
                 <
                   decltype(show),
                   decltype(showList)
                 >;
};
```

C++ type class instance

The typeclass_instance is defined as follow:

```
template <template <typename> class Class, typename T> struct typeclass_instance
{
    using type = typeclass<>;        // default instance is an empty typeclass.
};
```

The user is required to define the typeclass instance as a specialization for a given class and type:

```
template <>
struct typeclass_instance<Show, Test>
{
    using type = typeclass <
            decltype(show),
            decltype(showList)
        >;
};
```

C++ type class instance (example)

Example:

```
struct Test { };

std::string show(Test const &)
{ return "Test"; }
std::string showList(std::list<Test> const &)
{ return "[Test]"; }


template <>
struct typeclass_instance<Show, Test>
{
    using type = typeclass <
            decltype(show),
            decltype(showList)
        >;
};
```

C++ type class constraint #1

Lastly we need the constraint that ensure a type is indeed an instance of a certain typeclass:

```
namespace details
{
    has_function_(show);
    has_function_(showList);
}

template <typename T>
constexpr bool ShowInstance()
{
  return  type::equal_set< typename typeclass_instance<Show,Ty>::type,
                typename Show<Ty>::type>() &&
             details::has_function_show<T>::value &&
             details::has_function_showList<std::list<T>>::value;
};
```

C++ type class constraint #2

Or alternatively:

```
template <typename T>
constexpr bool ShowInstance()
{
    static_assert(!(sizeof...(Fs) <  Class<Ty>::type::size), "instance declaration: incomplete interface");
    static_assert(!(sizeof...(Fs) >  Class<Ty>::type::size), "instance declaration: too many method");
    static_assert(type::equal_set<typename typeclass_instance<Show,Ty>::type, typename Show<Ty>::type >(),
              "instance declaration: function(s) mismatch");
    return details::has_function_show<T>::value &&
           details::has_function_showList<std::list<T>>::value;
};
```

C++ type class SFINAE

Where the has_function_ is a macro that generate a proper SFINAE test to check the existence of a certain function (fun). A compiler intrinsic would be very welcome!

```
#define has_function_(fun) \
template <typename __Type> \
struct has_function_ ## fun \
{ \
   using yes = char[1]; \
   using no  = char[2]; \
   \
   template <typename __Type2> static yes& check(typename std::decay<decltype(fun(std::declval<__Type2>()))>::type *); \
   template <typename __Type2> static no& check(...); \
   \
   static constexpr bool value = sizeof(check<__Type>(0)) == sizeof(yes); \
};
```

Compiler errors #1

Incomplete instance:

```
using type = typeclass_instance
    <
        Show, Test,
        decltype(show)
    >;
```

```
In file included from /root/GitHub/meetup-milano-2014/cpp_typeclass/code/typeclass.cpp:5:
/root/GitHub/meetup-milano-2014/cpp_typeclass/code/./hdr/typeclass.hpp:14:5: error: static_assert failed "instance declaration: incomplete interface"
      static_assert( !(sizeof...(Fs) <  Class<Ty>::type::size ), "instance declaration: incomplete interface");
```

Compiler errors #2

Bad signatures:

```
using type = typeclass_instance
    <
        Show, Test,
        decltype(show),
        decltype(badShowList),
    >;
```

```
In file included from /root/GitHub/meetup-milano-2014/cpp_typeclass/code/typeclass.cpp:5:
/root/GitHub/meetup-milano-2014/cpp_typeclass/code/./hdr/typeclass.hpp:16:5: error: static_assert failed
"instance declaration: function(s) mismatch"
    static_assert( type::equal_set< type::multiset<Fs...>, typename Cl<Ty>::type >(), "TypeClass:
instance error");
```

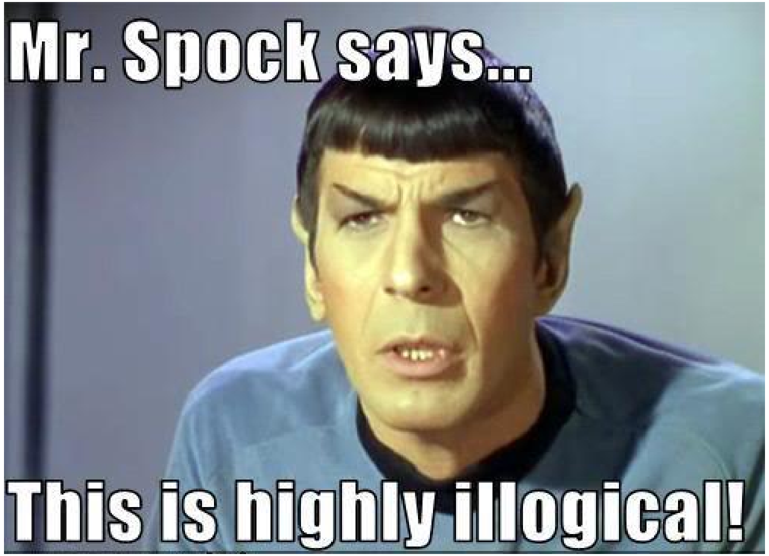
Compiler errors #3

Constraining a function argument type:

```
template <ShowInstance T>
void print(T const &elem)
{
     std::cout << show (elem) << std::endl;
}
```

```
error: no matching call to ‘print(NewType &)’
note: candidate is ‘print(T& elem)’
note: where T = NewType
note: template constraints not satisfied
note: ‘T’ is not a/an ‘ShowInstance’ type
```

```
template <typename T>
void print(T const &elem)
{
     static_assert(ShowInstace<T>(),
          “T not instance of Show”);
     std::cout << show (elem) << std::endl;
}
```


Mr. Spock says...
This is highly illogical!