C++: Type Traits

Miro Jurišić
meeroh@meeroh.org

Introduction

- Classes which encapsulate properties of types
- Is this an integer type? An arithmetic type? A pointer type?...
- Closely related to partial specialization

Example: Trivial partial specialization for pointers

```
template <typename T>
struct Example {
    static const bool isPointer = false;
};

template <typename T>
struct Example <T*> {
    static const bool isPointer = true;
};

Example <int>::isPointer;   // false
Example <int*>::isPointer;  // true
```

Example: Partial specialization for int

```
template <typename T>
class Array {
    private:
        size_t  mCount;
        T*      mData;

    public:
        Array (
            const Array <T>&    inOriginal);
};
```

```
template <typename T>
Array <T>::Array (
    const Array <T>&        inOriginal):
    mCount (inOriginal.mCount),
    mData (new T [mCount])
{
    for (
        size_t i = 0;
        i < mCount;
        ++i
    ) {
        mData [i] = inOriginal.mData [i];
    }
}
```

```
template <>
class Array <int> {
    private:
        int*    mData;
        size_t  mCount;

    public:
        Array (
            const Array <int>&  inOriginal);
};

template <>
Array <int>::Array (
    const Array <int>&      inOriginal):
    mCount (inOriginal.mCount),
    mData (new int [mCount])
{
    memcpy (mData, inOriginal.mData, mCount * sizeof (int));
}
```

Taking partial specializations further

- Partial specializations are useful
- Partial specializations permit optimizations
- Partial specializations help avoid code bloat
- Multiple partial specializations are a pain to write
- How can we make multiple partial specialiations easier to write?

Traits classes

```
namespace detail {
    template <typename T>
    struct is_interesting {
        static const bool value = false;
    };

    template <>
    struct is_interesting <int> {
        static const bool value = true;
    };

    template <>
    struct is_interesting <float> {
        static const bool value = true;
    };
```

```
    template <typename T, bool interesting>
    struct function_selector {
        static void function_impl (
            const vector <T>&       inVector)
        {
            // Do boring stuff
        }
    };

    template <typename T>
    struct function_selector <T,true> {
        void function_impl (
            const vector <T>&       inVector)
        {
            // Do interesting stuff
        }
    };
}
```

```
template <typename T>
void function (
    const vector <T>&           inVector)
{
    detail::function_selector <
     T,
     detail::is_interesting <T>::value
    >::function_impl (inVector);
}
```

Traits in STL

- None yet. A proposal is being reviewed.

Traits in boost

- Type traits proposed for STL are implemented in boost
- Except for the ones that require compiler support
- `#include <boost/type_traits.hpp>`

List of traits

- Primary categories: is_void, is_integral, is_float, is_array, is_pointer, is_reference, is_member_pointer, is_enum, is_union, is_class, is_function
- Composite categories: is_arithmetic, is_fundamental, is_object, is_scalar, is_compound
- Type properties: is_const, is_volatile, is_POD, is_empty, is_polymorphic, has_trivial_constructor, has_trivial_copy, has_trivial_assign, has_trivial_destructor, has_nothrow_construct, has_nothrow_copy, has_nothrow_assign
- Copiler support required for is_class, is_union, and is_polmorphic

Type relationships

- is_same, is_convertible, is_base_and_derived

Type modifications

- const-volatile modiciations: remove_const, remove_volatile, remove_cv, add_const, add_volatile, add_cv
- reference modiciations: remove_reference, add_reference
- array modifications: remove_bounds
- pointer modifications: remove_pointer, add_pointer

Example: debug_cast

```
namespace detail {
    template <bool IsPointer>
    struct debug_cast_impl {
    };

    template <>
    struct debug_cast_impl <true> {
        template <typename Result, typename Source>
        static Result cast (Source* inSource);
    };

    template <>
    struct debug_cast_impl <false> {
        template <typename Result, typename Source>
        static Result cast (Source& inSource);
    };
```

```
// Specialization for pointers
template <>
template <typename Result, typename Source>
inline Result
debug_cast_impl <true>::cast <Result, Source> (Source* inSource)
{
    assert (static_cast <Result> (inSource) == dynamic_cast <Result> (inSource));
    return static_cast <Result> (inSource);
}
```

```
    template <>
    template <typename Result, typename Source>
    inline Result
    debug_cast_impl <false>::cast <Result, Source> (Source& inSource)
    {
        #if Mollie_Target_Build_Debug
        try {
            assert (
             &static_cast <Result> (inSource) ==
             &dynamic_cast <Result> (inSource)
            );
        } catch (...) {
            // convert exceptions to assertions to prevent them
            // from being "handled"
            assert_str (std::string ("Cast failed in ") + __PRETTY_FUNCTION__);
        }
        #endif
        return static_cast<Result>(inSource);
    }
}
```

```
template <typename Result, typename Source>
inline Result debug_cast (Source& inSource)
{
    // Call the appropriate implementation
    return detail::debug_cast_impl <
        boost::is_pointer <Source>::value
    >::cast <
        Result,
        boost::remove_pointer <Source>::type
    > (inSource);
}
```