Traits Go Mainstream

(They're not just for library developers any more: a case study)

Leor Zolman
BD Software
www.bdsoft.com // Updated site!
leor@bdsoft.com

September, 2015

Traits Began Stealthy

- Before C++11, code employing type traits (available from boost and tr1) was mostly buried in library implementations
  - Clients of those libs had no real need to understand traits-based implementation techniques
- The introduction of rvalue references into Modern C++, and especially *universal* references, changes that
  - This talk strives to illustrate why

2

Review: Function Overload Resolution w/r/t Templates

- When the compiler encounters a function call, the call is resolved in three basic steps

  Step 1: If there is a *non-template* function in scope with the "exact" signature, that's the function that is used

  Step 2: If there is a function *template* that can be specialized to represent a valid function with that exact signature, that specialization is used

  Step 3: Finally, non-template functions with conversions, if any, are considered

For example…

3

Function Overload Resolution

```
void f(float x, float y)
{
	cout << "In f(float, float): x = " << x << endl;
}

template<typename T>
void f(const T &x, const T &y)
{
	cout << "In f<T>: x = " << x << endl;
}

int main()
{
	f(2.2f, 3.3f);		// f(float, float)
	f(10, 20);		// template
	f(2.2f, 3.3);		// f(float, float)
}
```

4

Universal References and Greed

- When a universal reference appears as a function template parameter, it “hijacks” all corresponding parameters in other overloads of that template
  - But only for other *templates* that overload it
  - Non-template functions still have first dibs

5

Universal References and Greed

```
template<typename T>               // universal reference
void f(T &&x);                     // (greedy)

template<typename T>               // regular reference
void f(const T&x);                 // (not called here)

void f(int x);                     // non-template

int main()
{
      string s("foo");

      f(10);                       // calls f(int)
      f('c');                      // calls f<>(T &&)
      f(2.2);                      // calls f<>(T &&)
      f(s);                        // calls f<>(T &&)
      f(10u);                      // calls f<>(T &&)
}
```

6

Example with Simple Class

- Consider a trivial class `Widget` encapsulating an `int` and providing an inserter:

```
class Widget {
public:
	Widget(int n) : val_(n) {}
	friend ostream &operator<<(
				ostream &os, const Widget &w)
	{
		return cout <<
			"Widget with value: " << w.val_;
	}
private:
	int val_;
};
```

7

Example with Simple Class

```
template<typename T>	// useVal designed to accept all
void useVal(T &&val)	// flavors of Widgets (rvalues,
{			// (rvalues, etc.) most efficiently:
	cout << "Using T&& = " << val << endl;
}

void useVal(long long lval) // useVal designed to accept
{			    // “all int types” ??
	cout << "Using long long = " << lval << endl;
}

int main()
{
	Widget w{10};

	useVal(w);
	useVal(Widget(5));
	useVal(10LL);
	useVal(10);
}
```

Output:

```
Using T&& = Widget with value: 10
Using T&& = Widget with value: 5
Using long long = 10
Using T&& = 10       ← Oops!
```

8

Ways to Avoid the Problem

- Don't overload the function
  - Choose a different name for each function
  - Unfortunately, won't work for constructors
- Replace the universal reference template with pass by reference-to-`const` or by value
  - Inefficient when function needs to forward the parameter to another function, because extra copies end up needing to be constructed and destroyed
- Sometimes, you just need the universal reference version along with other overloads

9

Tag Dispatch to the Rescue

```
template<typename T>      // useVal for all types, dispatches
void useVal(T &&val)      // to "helper" Impl functions
{
    useValImpl(forward<T>(val), is_same<
       typename remove_reference<T>::type, Widget>());
}

template<typename T>      // useValImpl for Widgets
void useValImpl(T&& val, true_type)
{
       cout << "Using T&& (a Widget) = " << val << endl;
}

template<typename T>      // useValImpl for non-Widgets
void useValImpl(T&& val, false_type)
{
       cout << "Using T&& (NOT a Widget) = " << val << endl;
}
```

10

Tag Dispatch to the Rescue

```
int main()
{
        Widget w{10};

        useVal(w);
        useVal(Widget(5));
        useVal(10LL);
        useVal(10);
}
```

```
                  Output:
Using T&& (a Widget): Widget with value: 10
Using T&& (a Widget): Widget with value: 5
Using T&& (NOT a Widget): 10
Using T&& (NOT a Widget): 10
```

11

C++14 Traits Aliases

- In C++14, the use of type traits is a bit cleaner thanks to a set of "clean-up" aliases:

```
// C++11:
template<typename T>
void useVal(T &&val)
{
     useValImpl(forward<T>(val), is_same<
        typename remove_reference<T>::type, Widget>());
}

// C++14:
template<typename T>
void useVal(T &&val)
{
        useValImpl(forward<T>(val),
                is_same<remove_reference_t<T>, Widget>());
}
```

12

Limitations of Tag Dispatch

- Tag dispatch is limited to decisions based on tags
- Helper templates are needed to express one variant of functionality for each possible tag
- Another way to make overloading decisions is to leverage SFINAE (Substitution Failure Is Not An Error)…

13

SFINAE

- Say we overloaded functions named `crunch` to work with both primtive types and UDTs:

```
// UDT Widget, and a crunch() function template that only
// works for class types with certain characteristics:

class Widget {
public:
	using result_type = double;
	double process() const { return value + 2.236; }
private:
	double value = 3.14;
};

template<typename T>    // crunch function template for UDTs
typename T::result_type crunch(const T& t)
{
	return t.process();
}
```

14

SFINAE

- When resolving the call `crunch(10)` below, the compiler must consider specializations of the template…even though the instantiated code would be illegal! This is SFINAE in action.

```
// A non-template crunch function for integral types:
int crunch(unsigned long long x) { return (x >> 2) + 1; }

// Test driver:
int main()
{                    // CONSIDERS the template; rejects it.
    cout << "crunch(10) = " << crunch(10) << endl;

    Widget w;
    cout << "crunch(w) = " << crunch(w) << endl;
}
```

15

`enable_if`

- This curious little metaprogramming component may be used to "turn on" or "turn off" function template overloads based on logical conditions (typically built of type traits)
- The usage is:

```
enable_if<boolean-expression,
               type>::type
```

- This evaluates to the type *`type`* if the expression is true, or an *illegal construct* (by design) otherwise
- That "illegal construct" triggers SFINAE, and the entire function template is ignored!

16

enable_if Applied

```
// crunch function template overload for class types
// containing a nested typedef named result_type:
template<typename T>
typename enable_if<is_class<T>::value,
                   typename T::result_type>::type
crunch(const T& t)
{
       return t.process();
}

// overload for integral types:
template<typename T>
typename enable_if<is_integral<T>::value, T>::type
crunch(T x) { return (x << 2) + 1; }
```

17

Simpler Example: Enable Function Template Only for Integral Types

```
// limited to being specialized on integral types
template<typename T>
void processInt(T t, typename
      enable_if<is_integral<T>::value, T>::type = 0)
{
      cout << "t is integral: " << t << endl;
}

int main()
{
      short i, *pi = &i;
      processInt(10);          // OK
      processInt(i);           // OK
      processInt(15.5);        // ERROR
      processInt(&i);          // ERROR
}
```

18

An Implementation of `enable_if` (just because it's cool…)

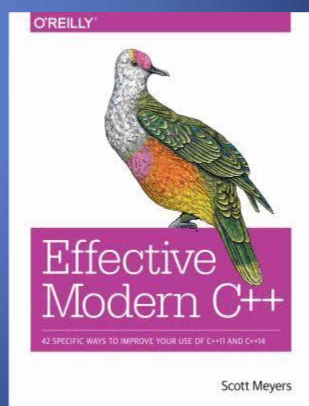
```
// Default case, when bool is false,
// yields an empty class (no members):
template<bool, typename T = void>
struct enable_if {};

// Special case, when bool is true,
// yields class with a member type
// that is a typedef for T:
template<typename T>
struct enable_if<true, T>
{
     typedef T type;
};
```

19

For More Information

- This talk was inspired by Item 27 of *Effective Modern C++* (by Scott Meyers):
  - "Familiarize yourself with alternatives to overloading on universal references"



20