Modern C++ Interfaces

Complexity, Emergent Simplicity, SFINAE, and Second Order Properties of Types

Stephen C. Dewhurst
stevedewhurst.com

About Steve Dewhurst

Steve Dewhurst is the cofounder and president of Semantics Consulting, Inc. He is the author of the books *C++ Common Knowledge* and *C++ Gotchas*, and the co-author of *Programming in C++*. He has written numerous technical articles on C++ programming techniques and compiler design. Steve served on both the ANSI/ISO C++ standardization committee and the ANSI/IEEE Pascal standardization committee.

Steve has consulted for projects in areas such as compiler design, embedded telecommunications, e-commerce, and derivative securities trading. He was programming track chair of *Embedded Systems*, a Visiting Scientist at CERT and a Visiting Professor of Computer Science at Jackson State University.

Steve was a contributing editor for *The C/C++ User's Journal*, an editorial board member for *The C++ Report*, and a cofounder and editorial board member of *The C++ Journal*. 
Outline

- Some years ago, Policy-Based Design techniques devolved implementation decisions to users of interfaces.
- More recently, interfaces seem to prefer to move such decisions away from users of interfaces to their implementers.
- Lately, there seems to be a great increase in use of SFINAE-based techniques in tandem with Modern C++. Why?
  - Increased complexity implies need for more nuanced interfaces.
  - Increased interface complexity implies that we are now embedding not just our experience in implementations, we're embedding our judgement in our interfaces.
  - New language features and libraries make it feasible.

Outline

✓ Hypothesis: We've hit a cusp such that C++ is complex enough that it's use is actually becoming simpler due to the necessity of using
  - convention,
  - idiom,
  - embedded experience,
  - and “Do What I Mean” interfaces.
Wishful Thinking...

- Recently, our code has evolved in the direction of relieving the user from, well, knowing much of anything.
- We've gone from comments...

```cpp
// DO NOT EVEN THINK OF PASSING AN ARRAY OF COMPLEX TYPES TO THIS FUNCTION!!!
template <typename T>
T *copy_it(T const *src, size_t n) {
    ~~~
}
```

Totalitarianism...

- ...to enforcing our will for their own good...

```cpp
template <typename T>
T *copy_it(T const *src, size_t n) {
    static_assert(
        is_trivially_copyable<T>::value,
        "array type must be memcpy-able"
    );
    ~~~
}
```
Embedding Experience

- ...to embedding our design experience directly in self-maintaining code.

```cpp
#include <cassert>
#include <cstddef>
#include <stdexcept>

template <typename T>
inline T *copy_array(const T *s, size_t n) {
    size_t const amt = sizeof(T) * n;
    T *d = static_cast<T*>(::operator new(amt));
    if (is_trivially_copyable<T>::value)
        d = static_cast<T*>(memcpy(d, s, amt));
    else if (has_nothrow_copy_constructor<T>::value)
        for (size_t i = 0; i != n; ++i)
            new (&d[i]) T(s[i]);
    else ...
}
```

Embedding Experience in C++17

- Moving faster than is typical, this idiom has made its way into the C++ standard.

```cpp
#include <cassert>
#include <cstddef>
#include <stdexcept>

template <typename T>
inline T *copy_array(const T *s, size_t n) {
    size_t const amt = sizeof(T) * n;
    T *d = static_cast<T*>(::operator new(amt));
    if constexpr (is_trivially_copyable<T>::value)
        d = static_cast<T*>(memcpy(d, s, amt));
    else if constexpr (has_nothrow_copy_constructor<T>::value)
        for (size_t i = 0; i != n; ++i)
            new (&d[i]) T(s[i]);
    else ...
}
```
Embedding Judgment

- We've simplified maintenance and use of implementations by embedding our experience.
- As implementations become more complex, some of that complexity inevitably leaks out into interfaces.
- As a result, designers have been embedding their judgement into interfaces.
- This has the effect of simplifying use of the interface, even if the actual interface is more complex due to its inflection by the nuanced implementation.

Language Changes That Impelled

- Increasing complexity in stating what your intentions are:
  - Preferential treatment of initializer-list arguments in overload resolution
  - Greedy universal references
  - Need to extend functionality in a backward-compatible way
  - Increasingly fine-grain distinguishability in overloaded function templates
- None of these individually caused the shift, but the language complexity reached a tipping point, where designers could no longer trust that their interfaces would allow the compiler and user to interpret an interface in the same way.

✓ To be clear: Increased language complexity is not an advantage in itself. However, it leads to greater expressiveness than would a less complex language. Simplicity is an emergent property.
Language Changes That Enabled

- Templated using declarations
- Default template arguments for function templates
- constexpr
- `<type_traits>`, in particular those aspects that require participation by the compiler.
- ...and some assistance from variadic templates.

SFINAE is Simple

- "Substitution Failure Is Not An Error" in template argument deduction.
- That is, if argument deduction finds at least one match, the failed matches aren't errors, as in:

  ```cpp
  template <typename T> void f(T);
  template <typename T> void f(T *);
  ```

  ```cpp
  f(1729); // no error, specializes first f
  ```

- The call `f(1729)` can match `f(T)`, but not `f(T *)`.
- The failure to match `f(T *)` is not an error.
- If `f(T)` were not present, it would be an error.
SFNAE in C++03 Was a Pain in the Neck

- Unlike a constraint implemented with a static assertion, SFNAE must be applied to an interface, before a decision is made.
- In the template parameter list,

  \[
  \text{template } \langle \text{typename } T \rangle \\
  \text{void munge_shape}(T \text{ const } &a) \\
  \{
  \}
  \]

  in the return type, in the argument list.

  or in the argument list.

  It's too late here, although we can static assert.

- In C++03, function templates could not have default template parameters.
- This typically left us to apply SFNAE to return types and argument lists. With unfortunate syntactic results.

SFNAE in Modern C++

- The augmented language makes it necessary to ask more compile-time questions.
  - We have more choices, and with great power comes great responsibility.
  - Happily, the augmented language provides facilities to help us to ask the questions.
    - One major piece: the fully-standard `<type_traits>` header file provides a collection of useful predicates (some of which are compiler intrinsics) and a syntactic model on which to build more complex predicates.
Default Function Template Arguments

- In C++11, function templates may have default template arguments.
  - This permits syntactic improvement because we no longer have to hide a constraint within some other facet of the declaration.

```cpp
template <
    typename T,
    typename = enable_if_t<is_base_of<Shape, T>::value>
>
void munge_shape(T const &a) { ~~~ }
```

- Now substitution will fail if it can't determine the type of the default template parameter.

Template Typedef

- In syntactic situations like this, use of using is of use:

```cpp
template <typename T>
using IsShape = typename enable_if<is_base_of<Shape, T>::value>::type;
```

- Our snobby function template is now fairly readable:

```cpp
template <typename T, typename = IsShape<T>>
void munge_shape(T const &a);
```
A Constructor Overload Issue

- Let's look at a sporadic problem with constructor overloading:

```cpp
template <typename T>
class Heap {
public:
    //
    Heap(size_t n, T const &v);
    template <typename In>
    Heap(In b, In e);  // range init
    //
};
```

Constructor Overload Code Smell

- Interference by the range initialization member template may give surprising results:

  ```cpp```
  Heap<int> h (5, 0);  // range initialization!
  ```cpp```

- The member template is a better match than the non-template two-argument constructor.

- Why?
  - The template is an exact match; `In` is deduced to be `int`.
  - The non-template requires a conversion on the first argument from `int` to `size_t`.

- I intended that constructor for input iterators only! Do what I mean!
Syntactic Difficulties

- Older template metaprogramming features of the standard library can be syntactically challenging:

  ```cpp
  is_same< // is this a random access STL iterator?
    typename iterator_traits<Iter>::iterator_category,
    random_access_iterator_tag
  >::value
  ```

- The expression uses long identifiers.
- It also requires explicit use of the keyword `typename` to identify the nested name `iterator_category` as a type.
- A “template typedef” alias can simplify the syntax...

Simplifying With “Template Typedef”

- For example, these alias templates can categorize iterators:

  ```cpp
  template <typename It>
  using Category = typename iterator_traits<It>::iterator_category;

  template <typename It>
  using is_exactly_rand
    = is_same<Category<It>, random_access_iterator_tag>;
  ```
Simplifying With Alias Declarations

- This alias template can determine if an iterator is an STL input iterator:

```cpp
template <typename It>
using is_in = is_true<
  is_exactly_in<It>::value || is_for<It>::value
>;
```

- The `is_true` template is non-standard.
- One last syntactic cleanup:

```cpp
template <typename It>
using IsIn
  = typename enable_if<is_in<It>::value>::type;
```

Disabling the Constructor with SFINAE

```cpp
template <typename T>
class Heap {
public:
  ~~~
  template <typename In, typename = IsIn<In> >
  Heap(In b, In e);
};
```

- Here, the required constraint is that `In` be an input iterator.
  ✓ *That's what I meant!*
Greedy Universal Members

- Universal references are very accommodating:

  ```
  template<typename T>
  class X {
    public:
      void operation(T const &); // #1: lvalue version
      void operation(T &&);      // #2: rvalue version
      template<typename S>
      void operation(S &&);      // #3: universal version
  };
  ```

- They often provide somewhat surprising better matches than functions without universal reference arguments.

Similar in Decay

- The `std::decay` type trait models the conversions and decay that occur when passing by value.
- We can use mutual decay to decide whether two types are “pretty much” the same:

  ```
  template<typename S, typename T>
  using similar = is_same<decay_t<S>, decay_t<T>>;

  template<typename S, typename T>
  using NotSimilar = enable_if_t<!similar<S, T>::value>;
  ```
Limiting Greediness

- Now we can use SFINAE to limit the use of the universal version of operation to types that are “not similar to” the type used to specialize X:

```cpp
template <typename T>
class X {
public:
    void operation(T const &); // #1: lvalue version
    void operation(T &&);       // #2: rvalue version
    template <typename S,
               typename = NotSimilar<S, T>>
    void operation(S &&);       // #3: universal version
      // Do What I Mean:
};
      // OK as long as S is not
      // "similar" to T
```

Self-Identification for SFINAE

- SFINAE for interface design is so effective, that some types are designed to facilitate it by making complex properties easy to determine.
- For example, complete specializations of standard function objects identify themselves as “transparent.”

```cpp
template <>
struct less<void> {
    template <class T, class U>
    constexpr auto operator()(T &&t, U &&u) const {
        return std::forward<T>(t)
            < std::forward<U>(u);
    }
    using is_transparent = void;
};
```
SFINAE, Again

- Standard set has members that are considered only if the set’s comparator is transparent:

```cpp
template <typename T, typename Comp = Comp::is_transparent>
class set {
public:
  iter lower_bound(const T &key);
  template <typename Key = Comp::is_transparent>
    iter lower_bound(const Key &key);
};
```

- Effectively, the interface to set is modified based on self-identified properties of its comparator.

Self-Identification

- For another example, consider a scoped enum that has been tricked up to act like a container of enumerators:*

```cpp
enum class bits {
  begin = 0x01,
  one = begin, two = 0x02, three = 0x04,
  four = 0x08, five = 0x10, six = 0x20, seven = 0x40,
  end = 0x80,
  is_enum_container
};
```

```cpp
template <typename E>
using IsEnumContainer = std::enable_if_t<sizeof(E::is_enum_container) >;
```

* Thanks to Dan Saks for the example.
Volunteering

- Only enums that self-identify as enum containers have container-like operations on their enumerators:

```cpp
template <
    typename T,
    T(&next)(T) = next_enum,
    T(&prev)(T) = prev_enum,
    typename = IsEnumContainer<T>
>
class enum_iterator {
    ~~~
};
```

Predicate Composition

- Compile time predicates like those in `<type_traits>` are often composed to test complex type properties.
- We can simplify the composition through use of a variadic template parameter pack:

```cpp
template <template <typename...> class... Preds>
struct Compose;
```
Using Composed Predicates

- We can use composition like this:

```
using Happy = Compose<is_class, is_transparent, is_big>;

static_assert(Happy::eval<T>(), "Unhappy, I am.");

template <typename T>
using IsHappy = enable_if_t<Happy::eval<T>>;

template <typename T, typename = IsHappy<T>>
void pursuit_of_happyness() { ~~~ }
```

Traditional First/Rest Implementation

```
template <template <typename...> class First,
    template <typename...> class... Rest>
struct Compose<First, Rest...> {
    template <typename T>
    static constexpr auto eval() {
        return First<T>::value &&
            Compose<Rest...>::template eval<T>();
    }
};

template <>
struct Compose<> {
    template <typename>
    static constexpr auto eval() {
        return true;
    }
};
```
Simpler Non-Recursive Implementation

- A C++14 constexpr function can simplify the implementation:

```cpp
template <template <typename...> class... Preds>
struct Compose {
    template <typename T>
    static constexpr auto eval() {
        auto results = { Preds<T>::value... };
        auto result = true;
        for (auto el : results)
            result &= el;
        return result;
    }
};
```

- ...and a C++17 fold operation could simplify even further.

Dealing With Complex Constraints

- We've seen a number of reasonably complex constraints so far.
- In such situations, it can help to have a framework available to automate away some of the complexity.
- Luckily, C++ has a rich collection of idioms to deal with complexity.
- We'll reuse some of these traditional idioms to write a framework:
  - Represent a compile-time data structure as a complex, nested type.
  - Use “expression template” operators to generate the complex type.
  - We'll write a constraint expression template language and parser that can handle the usual and, or, xor, and not operators.
Template Trees

- Rather than use a simple linear template predicate list, we'll use a template predicate tree structure.
  - Represent a compile-time data structure as a complex, nested type.
  - Use “expression template” operators to generate the complex type.
- We'll write a constraint expression template language and parser that can handle the usual and, or, xor, and not operators.

Abstract Syntax Trees

- A type predicate expression like

\[ \text{pred1} \& (\text{pred2} \mid \neg \text{pred3}) \]

- Should generate a parse tree like

```
&
\text{pred1}
\mid
\text{pred2}
\mid
\neg
\text{pred3}
```

where the leaves of the AST are templates.
Idiomatic Blast From The Past

- Actually, we don't really want a parse tree, *per se*, but a (compile time) type that contains the information from the parse tree, similar to the use of a type list to represent a linear sequence of types.
- The leaves of the expression tree are values of the form

  \[ \text{template <typename> class Pred; // a type predicate} \]

- For example, most of the predicates in `<type_traits>` qualify.
- We'll employ a compile-time-only version of the venerable Expression Template idiom in the implementation.
- Here’s the root type of the AST that will come in handy later:

  \[ \text{struct E {}; // every node type is an E of some sort} \]

And/Or...

- We'll implement binary operators like this:

  \[ \text{template <typename P1, typename P2> struct And : E \{ \text{\begin{verbatim}template <typename T> static constexpr bool eval() \{ return P1::template eval<T>(); & P2::template eval<T>(); \} \}} \text{template <typename P1, typename P2> struct Or : E \{ \text{\begin{verbatim}~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~\~}\end{verbatim}\}} \]
&/|...

- For clarity and convenience, we'll use an infix operator interface to generate the type.

```cpp
template <typename P1, typename P2> constexpr And<P1, P2> operator &(P1, P2) { return And<P1, P2>(); }
```

- Note that we're interested entirely in the (compile time) return type of the function rather than the (runtime) return value.

✓ *Note the value of leveraging function template argument deduction to perform compile-time type algebra.*

!...

- Unary operators are even easier:

```cpp
template <typename P>
struct Not : E {
    template <typename T>
    static constexpr bool eval() {
        return !P::template eval<T>();
    }
};

template <typename P>
constexpr Not<P> operator !(P) { return Not<P>(); }
```
Leaves

- The leaves in our compile time AST are unary type predicates.

```cpp
template <template <typename> class Pred> struct Id : E {
    template <typename T>
    static constexpr bool eval() {
        return Pred<T>::value;
    }
};

template <template <typename> class Pred> constexpr Id<Pred> pred()
    { return Id<Pred>(); }
```

<Type_traits>

- It's convenient to provide versions of standard unary type traits as leaves:

```cpp
constexpr auto isTriviallyCopyable = pred<std::is_trivially_copyable>();
constexpr auto isStandardLayout = pred<std::is_standard_layout>();
constexpr auto isPod = pred<std::is_pod>();
constexpr auto isLiteralType = pred<std::is_literal_type>();
// ad infinitum...
```
Constructing Complex Predicates

- We perform a compile time traversal of the type representation of the AST with a type argument:

```cpp
constexpr auto my_needs // build an AST
= isClass & (isPod | !isPolymorphic) ^ isShape;

constexpr auto your_needs // build another
= isClass & hasVirtualDestructor
  ^ !isNothrowCopyAssignable;
```

Compile Time Evaluation

- We can evaluate an AST directly:

```cpp
my_needs.eval<T>();
```

- ...but a little syntactic sugar is always in good taste:

```cpp
template <typename T, typename AST> // get a bool
constexpr bool constraint(AST)
{ return AST().template eval<T>(); }

template <typename T, typename AST> // get a type...maybe using Constraint =
std::enable_if_t<constraint<T>(AST())>;
```
Using the Predicate

- Sometimes we need a Boolean constraint:

  ```cpp
  static_assert(constraint<T>(my_needs),
                "My idiosyncratic needs are unmet.");
  ```

- Sometimes we're in SFINAE mode:

  ```cpp
  template<typename Me, typename You, 
            typename = Constraint<Me, my_needs>,
            typename = Constraint<You, your_needs>>
  void oy_vey(Me &&me, You &&you) { ~~~ }
  ```

That's Not What I Meant!

- Unfortunately, this implementation—intended to simplify our use of SFINAE—causes sporadic compilation errors.
- The overloaded operators are too accommodating.

  ```cpp
  template<typename P1, typename P2>
  constexpr And<P1, P2> operator &(P1, P2) 
  { return And<P1, P2>(); }
  ```

- This overload will be considered for any & that accepts at least one class argument...
  ✓ ...which is not what I meant.
What I Mean Is...

- We'll call in SFINAE to rescue our SFINAE toolkit:

  template <typename... Ps>
  using all_E_t = enable_if_t<all_E<Ps...>::value>;

  template <typename P1, typename P2, // I meant...
            typename = all_E_t<P1, P2>> // ...only ASTs!
  constexpr And<P1, P2> operator &(P1, P2)
  { return And<P1, P2>(); }

What I Mean To Say Is...

- Increasingly our designs require us to distinguish not only among predefined and user-defined conversions, but to include arbitrary constraints and properties in making compile time decisions.

- One way to look at the situation is that we’re no longer writing code just in terms of “first order” properties of types, but on design-specific, ad hoc “second order” properties.

- Some of these properties are extracted from types by the interface, some are offered to the interface by the type.
An Emergent Property of C++’s Complexity

- SFINAE is increasingly employed in modern C++ to make these decisions, and the result is that interfaces are—or can be—simpler and more natural.
- This simplicity is an emergent property of C++’s complexity.
- Newer features of the C++ language and standard library provide straightforward ways to apply SFINAE to our designs.

The End

Thanks for Coming!