SFINAE Sono Buoni

SFINAE

In attempting to use function template argument deduction to select among a number of candidate function templates, a C++ compiler may attempt an instantiation that fails on one or more of them.

```
template <typename T> void f( T );
template <typename T> void f( T * );
//…
f( 1024 ); // instantiates first f
```

Even though substitution of the integer for `T *` in the second `f` function template would have been incorrect, the attempted substitution does not give rise to an error provided that a correct substitution is found. In this case, the first `f` is instantiated, and there is no error. Thus, we have the "substitution failure is not an error" concept, dubbed SFINAE by Vandevoorde and Josuttis.[1]

SFINAE is an important property in that, without it, it would be difficult to overload function templates; the combination of argument deduction and overloading would render many uses of a set of overloaded function templates illegal. But SFINAE is also valuable as a metaprogramming technique.

SFINAE vs. Partial Specialization

Consider a simple utility that can be used to determine whether a type is a pointer type:

```
template <typename T> // T is not a pointer…
struct IsPtr
  { enum { r = false }; };

template <typename T> // …unless it's an unqualified pointer
struct IsPtr<T *>
  { enum { r = true }; };

template <typename T> // …or a const pointer
struct IsPtr<T * const>
  { enum { r = true }; };

template <typename T> // …or a volatile pointer
struct IsPtr<T * volatile>
  { enum { r = true }; };

template <typename T> // …or a const volatile pointer
```

[1] Vandevoorde and Josuttis, *C++ Templates*, Addison-Wesley 2003.

```
struct IsPtr<T * const volatile>
  { enum { r = true }; };
```

This can be used to make a compile time, metaprogrammed decision. For example, we can choose alternate container implementations based on whether the container's element type is a pointer or not.

```
template <typename T>
class SList {
  //…
  typedef typename
        Select< IsPtr<T>::r, Cptr< DePtr<T> >, T >::R ElemType;
 private:
  struct Node {
        Node *next_;
        ElemType el_;
  } *head_;
  //…
};
```

We can use SFINAE to achieve a similar result.

```
typedef True char; // sizeof(True) == 1
typedef struct { char a[2]; } False; // sizeof(False) > 1
//…
template <typename T> True isPtr( T * );
False isPtr( … );

#define is_ptr( e ) (sizeof(isPtr(e))==sizeof(True))
```

Here, we can determine whether the type of an *expression* is a pointer through a combination of function template argument deduction and SFINAE. If the expression `e` has pointer type, the compiler will match the template function `isPtr`, otherwise it will match the non-template `isPtr` function with the ellipsis formal argument. SFINAE assures us that the attempt to match the template `isPtr` with a non-pointer will not result in a compile time error.

The second bit of magic is the use of `sizeof` in the `is_ptr` macro. Notice that neither `isPtr` function is defined. This is correct, because they are never actually called. The appearance of the function call in a `sizeof` expression causes the compiler to perform argument deduction and function matching, but does not cause a function call to be generated. `sizeof` is interested only in the size of the return type of the function that *would have* been called. We can then check the size of the function's return type to determine which function was matched. If the compiler selected the function template, then the expression `e` had pointer type.

Note that we did not have to special case for const pointers, volatile pointers, and const volatile pointers as we did for the analogous `IsPtr` facility above that we implemented with class template partial specialization. As part of function template argument deduction, the compiler will ignore "first level" cv-qualifiers (const and volatile) as well as reference modifiers. (If we'd wanted to distinguish differently qualified pointer types, then we'd have declared four different template `isPtr` functions to take formal argument types of reference to pointer.) Note also that we do not have to be concerned about incorrectly identifying as a pointer type a user-defined type that has a conversion operator. The compiler employs a very restricted list of conversions on the actual arguments during function template argument deduction, and user-defined conversions are not on the list.

```
template <typename T>
class NotAPtr {
  //…
  operator T *() const; // conversion operator
};
//…
NotAPtr<int> nap;
Select< is_ptr(nap), X, Y>::R temp; //temp is of type Y
```

SFINAE Examples

Is this type a class type?

```
template <typename T>
struct IsClass {
    template <class C> static True isClass( int C::* );
    template <typename C> static False isClass( … );
    enum { r = sizeof(IsClass<T>::isClass<T>(0)) == sizeof(True) };
};
```

Is This Type a Pointer to a Class Type?

```
template <typename T>
struct IsPtrToClass {
    enum { r = IsPtr<T>::r && IsClass<typename DePtr<T>::R>::r };
};
```

Does This Class Contain The Typename `iterator`?

This is abstracted from Vandevoorde and Josuttis. Of course, this can be implemented to ask the question of any nested typename, not just `iterator`.

```
template <class C>
True hasIterator( typename C::iterator const * );
template <typename T>
False hasIterator( … );
#define has_iterator( C ) (sizeof(hasIterator<C>(0))==sizeof(True))
```

Is This a Non-Static Member Function?

This is implemented to answer the question for member functions of 0, 1, or two arguments, but can easily be extended to any fixed number.

```
template <typename R, class C>
```

```
True isMemf( R (C::*)( ) );

template <typename R, typename A, class C>
True isMemf( R (C::*)(A) );

template <typename R, typename A1, typename A2, class C>
True isMemf( R (C::*)(A1,A2) );

False isMemf( … );

#define is_member_func( f ) (sizeof( isMemf(f) ) == sizeof(True))
```

Can I Convert a T1 to a T2?

This is from Andrei Alexandrescu. Note that this mechanism will detect both predefined and user-defined conversions.

```
template <typename T1, typename T2>
struct CanConvert {
    static True canConvert( T2 );
    static False canConvert( ... );
    static T1 makeT1( );
    enum { r = sizeof(canConvert( makeT1( ) )) == sizeof(True) };
};
```

Appendix: Miscellaneous Utilities

Here's the source code for some utilities that appeared above.

Select

This implementation of `Select` is a modified form of the `Select` that appears in Andrei Alexandrescu's Loki Library. It is basically a compile time if-statement whose result type is either the second or third argument, based on the first argument.

```
template <bool, typename A, typename B>
struct Select {
    typedef A R;
};

template <typename A, typename B>
struct Select<false,A,B> {
    typedef B R;
};
```

Cptr

`Cptr` is a modified form of Nicolai Josuttis's `CountedPointer` that appeared in his *The C++ Standard Library*. `Cptr` is a "smart pointer" that reference counts and garbage collects the object to which it refers.

```
template <class T>
class Cptr {
  public:
    Cptr( T *p ) : p_( p ), c_( new long( 1 ) ) { }
    ~Cptr() { if( !--*c_ ) { delete c_; delete p_; } }
    Cptr( const Cptr &init )
         : p_( init.p_ ), c_( init.c_ ) { ++*c_; }
    Cptr &operator =( const Cptr &rhs ) {
        if( this != &rhs ) {
            if( !--*c_ ) { delete c_; delete p_; }
            p_ = rhs.p_;
            ++*(c_ = rhs.c_); // macho!
        }
        return *this;
    }
    T &operator *() const { return *p_; }
    T *operator ->() const { return p_; }
  private:
    T *p_;
    long *c_;
};
```

DePtr

`DePtr` strips away a single pointer modifier from the type used to instantiate it, if possible.

```
template <typename T>
struct DePtr {
    typedef T R; // T is not a ptr; result is same
};
template <typename T>
struct DePtr<T *> {
    typedef T R; // pointed-to type
};
template <typename T>
struct DePtr<T * const> {
    typedef T R; // pointed-to type
};
template <typename T>
struct DePtr<T * volatile> {
    typedef T R; // pointed-to type
};
template <typename T>
struct DePtr<T * const volatile> {
```

```
    typedef T R; // pointed-to type
};
```