Unrolling Expertise

Leveraging Expertise

One of the most useful aspects of generic programming is its ability to seamlessly permit the leveraging of the expertise of a group's individuals to the group as a whole. Most importantly, this sharing of expertise is accomplished without the need for general communication, so additional organizational complexity and communication overhead required to support it is minimal or non-existent. Each local improvement to a component is immediately made available to all components, invisibly and effortlessly.

Consider a simple generic sort algorithm:

```
template <typename T>
void sort( T a[], size_t n ) {
    bool changed;
    do {
        changed = false;
        for( int i = 0; i < n-1; ++i )
           if( a[i+1] < a[i] ) {
                T temp( a[i] );
                a[i] = a[i+1];
                a[i+1] = temp;
                changed = true;
           }
    } while( changed );
}
```

This is, of course, the extremely common and useful bubble sort, implemented on an array of `T` objects. However, it contains a design error. The swap of the elements `a[i]` and `a[i+1]` is written explicitly, rather than delegated to an appropriate generic algorithm. This won't cause much difficulty when sorting arrays of integers or floating point numbers, but it won't work well for all types.

Consider the effect this code will have on sorting an array of standard `strings`:

```
std::string temp( a[i] );
a[i] = a[i+1];
a[i+1] = temp;
```

This code will be much slower than the alternative presented by the `string` type itself:

```
a[i].swap(a[i+1]);
```

This is the implementation that would be used by the designer of the `string` type to implement a non-member swap for `string`, but our generic algorithm will not pick up that version. A correct implementation of the `sort` algorithm would ensure that any type-specific improvements would be picked up automatically as they appear:

```
template <typename T>
```

```
void sort( T a[], size_t n ) {
    bool changed;
    do {
        changed = false;
        for( int i = 0; i < n-1; ++i )
           if( a[i+1] < a[i] ) {
                swap(a[i], a[i+1] ); // correct!
                changed = true;
           }
    } while( changed );
}
```

Who Knew What When?

The idea is to write generic code that can make maximal use of statically available information in order to provide both flexibility and efficiency without any action or knowledge on the part of its users. The use of `swap`, above, is one example of this approach. If we can determine statically that there is a specialized version of `swap` for `T`, we'll pick it up. The important point in this approach is that the various scattered pockets of expertise related to the components of the `sort` algorithm will be collected and integrated while the user of `sort`, as well as the providers of the individual packets of expertise, remain ignorant of each other.

We can do more. Let's look at a simple sum algorithm that sums the elements of an array:

```
template <typename T>
inline T sum( T *a, int len ) {
    T s = T();
    for( int i = 0; i < len; ++i )
        s += a[i];
    return s;
}
```

This will work well under most circumstances.

```
double a1[ ] = { 1.2, 2.3, 3.4, 4.5 };
std::string a2[ ] = { "Hi", "Hej", "Hola", "Bonjour" };
//…
cout << sum(a1,4) << endl; // 11.4
cout << sum(a2,4) << endl; // HiHejHolaBonjour
```

But it doesn't work for a sequence of constants.

```
const int a3[ ] = { 1,2,3,4 };
//…
cout << sum(a3,4) << endl; // error!
```

We can do a little type manipulation to get out of this difficulty:

```
template <typename T>
```

```
inline T sum( T *a, int len ) {
    typename DeConst<T>::R s = T( );
    for( int i = 0; i < len; ++i )
        s += a[i];
    return s;
}
//…
cout << sum(a3,4) << endl; // OK
```

The use of `DeConst` is a simple piece of template metaprogramming the will remove an outer-level `const` qualifier from a type (if it's present):

```
template <typename T>
struct DeConst
  { typedef T R; };

template <typename T>
struct DeConst<const T>
  { typedef T R; };
```

This small augmentation makes the `sum` algorithm more flexible, and also saves the user of the algorithm from having to distinguish between arrays with constant and non-constant elements. But this is a rather minor "leveraging" of expertise.

Unrolling Loops

A more major leveraging would be to "unroll" the computation of the sum into straight-line code, since it's usually the case that straight-line code is faster than the equivalent iteration.

```
// not terrible…
int s = int();
for( int i = 0; i < 4; ++i )
    s += a[i];
// …but this is faster!
s = a[0]+a[1]+a[2]+a[3];
```

Let's modify our `sum` algorithm to do this unrolling for the user. The technique implements the iterative loop as a compile time recursive template instantiation.

```
template <int len, typename T>
struct Sum {
    typedef typename DeConst<T>::R Temp;
    static Temp r( T *a )
        { return *a + Sum<len-1,T>::r(a+1); }
};
```

The recursion terminates when the "loop counter" reaches one.

```
template <typename T>
    struct Sum<1,T> {
```

```
    static T r( T *a )
        { return *a; }
};
```

This will result in a sequence of inline static member function instantiations, which, in turn, will result in an unrolled loop. This works, but the user interface leaves a lot to be desired:

```
const int a3[ ] = { 1,2,3,4 };
//…
cout << Sum<4,int>::r(a) << endl;
```

To clean up the interface, we introduce a "helper" function template to perform template argument deduction and instantiate the class template:

```
template <int len, typename T>
inline typename DeConst<T>::R summer( T *a )
    { return Sum<len,T>::r(a); }
```

The compiler can deduce the type argument, but (in this case) the user has to supply the array length:

```
const int a3[ ] = { 1,2,3,4 };
//…
cout << summer<4>(a) << endl;
```

Facing Reality

Of course, we probably don't want to unroll very long loops, so we can choose between runtime iteration and compile time recursion.

```
const int maxUnroll = 7;

template <int len, typename T>
inline typename DeConst<T>::R summeropt( T *a )
    { return len>maxUnroll ? sum(a,len) : summer<len>(a); }
```

We might also want to avoid unrolling loops that involve summation of (potentially expensive) user-defined types.

```
template <int len, typename T>
inline typename DeConst<T>::R summeropt2( T *a ) {
    return len<=maxUnroll && IsPredefIntegral<T>::r
        ? summer<len>(a)
        : sum(a,len);
}
```

Note that the conditionals above are integer constant-expressions. This means that they can be evaluated at compile time, and the compiler will (unless a debugging flag is enabled) eliminate the code for the conditional and for the branch not taken, leaving a simple inline call to the remaining branch.

We're most of the way there. We've provided a lot of expertise to the user (or behind the user's back), but the user still must specify the length of the array to sum as an integer

constant-expression, whereas the original implementation of the `sum` algorithm allowed the length of the array to be any integral expression at all.

```
const int a3[ ] = { 1,2,3,4 };
//…
const int len1 = 4;
int len2 = 4;
cout << summeropt2<len1>(a) << endl; // OK
cout << summeropt2<len2>(a) << endl; // error! len2 not const-expr
```

We've traded flexibility for efficiency, essentially requiring the user of `sum` distinguish between arrays whose sizes are known at compile time and those whose sizes are not. Providing the user with such a choice will inevitably lead to inefficiency or error, since some users will simply always use the original `sum` algorithm that does not require them to make a distinction, and others will attempt to use the more efficient version when it does not apply.

Improving Reality

An improved implementation might be to allow the user to always employ the interface of the simple, original `sum` algorithm, while its implementation is special-cased depending on whether the length of the array is an integer constant-expression or not.

Unfortunately, your correspondent (me) has not been able to come up with a mechanism that can reliably distinguish between constant expressions and other expressions at compile time.[1] Here's the best I've up with so far:

```
typedef char True;
typedef struct { char x_[2]; } False;
//…
True ciexpr( const int & );
False ciexpr( int & );
#define is_probably_compile_int( e ) \
   (sizeof(ciexpr(e))==sizeof(True))
```

This approach usually works, but is not foolproof. For instance, an uninitialized constant with external linkage is not an integer constant-expression, but it will fool this implementation.

```
extern const int aValue; // const, but not a constant-expression
//…
cout << is_probably_compile_int( aValue ) << endl; // true!
```

However, this will work for most cases, and for all common cases. Let's use it to improve reality for the users of the `sum` algorithm.

[1] I mean, I've got to earn a living and all. I haven't yet found a client who's willing to hire me just to think deep thoughts. But I'm available…

First, we'll provide two versions of a template depending on whether the length of the array can be determined at compile time or not. The primary template is used in the case that the length is not a constant-expression, and will invoke the general `sum` algorithm:

```
template <bool>
struct SumImpl {
  template <int n, typename T>
  static T sum( T *a, int len ) { return ::sum(a,len); }
};
```

A specialization of `SumImpl` handles the case where the array length can be determined at compile time, and therefore has access to the all the optimizations described above.

```
template <>
struct SumImpl<true> {
  template <int n, typename T>
  static T sum( T *a, int ) { return ::summeropt2<n>(a); }
};
```

Finally, there's a bit of sleight-of-hand in invoking the correct static member function of the correct instantiation of the `SumImpl` template. First, if we want to continue to use function call syntax to invoke the algorithm, we have to use a macro. This is because a constant-expression passed as a parameter to a real function is no longer a constant-expression. (Yes, I hate macros too.)

```
#define finalsum( A, LEN ) \
  (SumImpl<is_probably_compile_int(LEN)>::sum \
       <is_probably_compile_int(LEN)?LEN:-1>( A, LEN ))
```

Second, we employ the `is_probably_compile_int` macro (sigh) to instantiate the proper version of `SumImpl`.

Third, we have to avoid attempting to instantiate the static `sum` template member function of the primary `SumImpl` template with a non-constant-expression. To this end we again employ the `is_probably_compile_int` macro to generate an unused constant-expression (in this case, `-1`, though any value would do), which is simply ignored.

Now the users of the `sum` algorithm can proceed in blithe ignorance, and still get very highly tuned results.

```
const int a2[] = { 1,2,3,4,5,6,7,8,9 };
const int length2 = sizeof(a2)/sizeof(a2[0]);
//…
cout << finalsum(a2,length2-3) << endl; // highly optimized!
```

Ignorance is Strength

Perhaps more than any other programming language, C++ requires a lot of experience and knowledge for effective use. Due to the demographics of its use and adoption, there are many more novice or casual C++ programmers than there are experts. This is in no way a pejorative comment about these inexpert programmers. Some are on their way to

becoming experts, and others are simply domain experts who have no interest or necessity in devoting the time required to become expert. Unfortunately, this implies that programmers who do not have detailed understanding of the subtleties of the language write most C++ code; as a result, much of the C++ code we see is inefficient, buggy, and difficult to maintain.

One way to improve this bleak situation is to acknowledge and strengthen the distinction between expert and novice C++ programmers. Novices should largely program in ignorance (while improving their skills, of course, to later become experts) while experts should provide toolkits and frameworks that allow the work of novices—which is the majority of C++ code—to be both efficient and correct. The simple example we've shown here is one example of how template metaprogramming can be used to accomplish this.