The technique of expression templates allows expressions to be passed to functions as an argument and inlined into the function body. Here are some examples:

```cpp
// Integrate a function from 0 to 10
DoublePlaceholder x;
double result = integrate(x/(1.0+x), 0.0, 10.0);
```
// Count the elements between 0 and 100 in a
collection of int
List<int> myCollection;
IntPlaceholder y;
int n = myCollection.count(y >= 0 && y <= 100);

// Fill a vector with a sampled sine function
Vector<double> myVec(100);
Vector<double>::placeholderType i;
myVec.fill(sin(2*M_PI*i/100));  // eqn in terms of element #

In each of these examples, the compiler produces a function instance which contains the expression inline.
The technique has a similar effect to *Jensen’s Device* in *ALGOL 60* [1]. Expression templates also solve an
important problem in the design of vector and matrix class libraries. The technique allows developers to write
code such as

Vector y(100), a(100), b(100), c(100),
d(100);
y = (a+b)/(c-d);

and have the compiler generate code to compute the result in a single pass, without using temporary vectors.

**So how does it work?** The trick is that the expression is parsed at compile time, and stored as nested
template arguments of an "expression type. Let’s work through a simplified version of the first example,
which passes a mathematical function to an integration routine. We’ll invoke a function `evaluate()` which
will evaluate f(x) at a range of points:

```cpp
int main()
{
    DExpr<DExprIdentity> x;         // Placeholder
    evaluate(x/(1.0+x), 0.0, 10.0);
    return 0;
}
```

For this example, an expression is represented by an instance of the class DExpr<A> (short for double
expression). This class is a wrapper which disguises more interesting expression types. DExpr<A> has a
template parameter (A) representing the interesting expression, which can be of type DBinExprOp (a binary
operation on two subexpressions), DExprIdentity (a placeholder for the variable x), or DExprLiteral (a
constant or literal appearing in the expression). For example, the simple subexpression "1.0+x" would be
represented by the expression type shown in Figure 1. These expression types are inferred by the compiler at
compile time.

![Figure 1. (A) Parse tree of "1.0+x", showing the similarity to the expression
type generated by the compiler (B). The
type DApAdd is an applicative template
class which represents an addition
operation. In (C), the type name is shown in
a non-expanded view. For this
implementation, type names are similar to a
postfix traversal of the expression tree.]

To write a function which accepts an expression as an argument, we include a template parameter for the
expression type. Here is the code for the `evaluate()` routine. The parenthesis operator of expr is overloaded to
evaluate the expression:

```cpp
template<class Expr>
void evaluate(DExpr<Expr> expr, double start, double end)
{
    const double step = 1.0;
```
When the above example is compiled, an instance of evaluate() is generated which contains code equivalent to the line

```cpp
cout << i/(1.0+i) << endl;
```

This happens because when the compiler encounters `x/(1.0+x)` in

```cpp
evaluate(x/(1.0+x), 0.0, 10.0);
```

it uses the return types of overloaded + and / operators to infer the expression type shown in Figure 2. Note that although the expression object is not instantiated until run time, the expression type is inferred at compile time. An instance of this expression type is passed to the evaluate() function as the first argument. When the fragment `expr(i)` is encountered, the compiler builds the expression inline, substituting `i` for the placeholder `x`. The mechanics of this are explained further on.

![Figure 2. Template instance generated by x/(1.0+x):](image)

As an example of how the return types of operators cause the compiler to infer expression types, here is an operator/() which produces a type representing the division of two subexpressions `DExpr<A>` and `DExpr<B>`:

```cpp
template<class A, class B>
DExpr<DBinExprOp<DExpr<A>, DExpr<B>, DApDivide> >
operator/(const DExpr<A>& a, const DExpr<B>& b)
{
  typedef DBinExprOp<DExpr<A>, DExpr<B>, DApDivide> ExprT;
  return DExpr<ExprT>(ExprT(a,b));
}
```

The return type contains a `DBinExprOp`, which represents a binary operation, with template parameters `DExpr<A>` and `DExpr<B>` (the two subexpressions being divided), and `DApDivide`, which is an applicative template class encapsulating the division operation. The return type is a `DBinExprOp<...>` disguised by wrapping it with a `DExpr<...>` class. If we didn’t disguise everything as `DExpr<>`, we would have to write eight different operators to handle the combinations of `DBinExprOp<>`, `DExprIdentity`, and `DExprLiteral` which can occur with `operator/()`. (More if we wanted unary operators!)

The typedef `ExprT` is the type `DBinExprOp<...>` which is to be wrapped by a `DExpr<...>`. Expression classes take constructor arguments of the embedded types; for example, `DExpr<A>` takes a const `A&` as a constructor argument. So `DExpr<ExprT>` takes a constructor argument of type `const ExprT&`. These arguments are needed so that instance data (such as literals which appear in the expression) are preserved.

The idea of applicative template classes has been borrowed from the Standard Template Library (STL) developed by Alexander Stepanov and Meng Lee, of Hewlett Packard Laboratories [2]. The STL has been accepted as part of the ISO/ANSI Standard C++ Library. In the STL, an applicative template class provides an inline operator() which applies an operation to its arguments and returns the result. For expression templates, the application function can (and should) be a static member function. Since operator() cannot be declared static, an apply() member function is used instead:

```cpp
// DApDivide -- divide two doubles
class DApDivide {
public:
  static inline double apply(double a, double b) { return a/b; }
};
```
When a DBinExprOp's operator() is invoked, it uses the applicative template class to evaluate the expression inline:

```cpp
template<class A, class B, class Op>
class DBinExprOp {
    A a_;    
    B b_;    

public:
    DBinExprOp(const A& a, const B& b)
    : a_(a), b_(b)
    { }

    double operator()(double x) const
    { return Op::apply(a_(x), b_(x)); }
};
```

It is also possible to incorporate functions such as exp() and log() into expressions, by defining appropriate functions and applicative templates. For example, an expression object representing a normal distribution can be easily constructed:

```cpp
double mean=5.0, sigma=2.0;   // mathematical constants
DExpr<DExprIdentity> x;
evaluate( 1.0/(sqrt(2*M_PI)*sigma) * exp(sqr(x-mean)/(-2*sigma*sigma)), 0.0, 10.0);
```

One of the neat things about expression templates is that mathematical constants are evaluated only once at run time, and stored as literals in the expression object. The evaluate() line above is equivalent to:

```cpp
double __t1 = 1.0/(sqrt(2*M_PI)*sigma);
double __t2 = (-2*sigma*sigma);
evaluate( __t1 * exp(sqr(x-mean)/__t2), 0.0, 10.0);
```

Another advantage is that it's very easy to pass parameters (such as the mean and standard deviation) -- they're stored as variables inside the expression object. With pointers-to- functions, these would have to be stored as globals (messy), or passed as arguments in each call to the function (expensive).

Figure 3 shows a benchmark of a simple integration function, comparing callback functions and expression templates to manually inlined code. Expression template versions ran at 95% the efficiency of hand-coded C. With larger expressions, the advantage over pointers-to-functions shrinks. This advantage is also diminished if the function using the expression (in this example, integrate()) does significant computations other than evaluating the expression.

It is possible to generalize the classes presented here to expressions involving arbitrary types (instead of just doubles). Expression templates can also be used in situations where multiple variables are needed -- such as filling out a matrix according to an equation. Since most compilers do not yet handle member function templates, a virtual function trick has to be used to pass expressions to member functions of a class. This technique is illustrated in the next example, which solves a significant outstanding problem in the design of matrix and vector class libraries.
Optimizing Vector Expressions

C++ class libraries are wonderful for scientific and engineering applications, since operator overloading makes it possible to write algebraic expressions containing vectors and matrices:

\[
\text{DoubleVec } y(1000), \ a(1000), \ b(1000), \ c(1000), \ d(1000); \\
y = (a+b)/(c-d);
\]

Until now, this level of abstraction has come at a high cost, since vector and matrix operators are usually implemented using temporary vectors [3]. Evaluating the above expression using a conventional class library generates code equivalent to:

\[
\text{DoubleVec } __t1 = a+b; \\
\text{DoubleVec } __t2 = c-d; \\
\text{DoubleVec } __t3 = __t1/__t2; \\
y = __t3;
\]

Each line in the code above is evaluated with a loop. Thus, four loops are needed to evaluate the expression \( y = (a+b)/(c-d) \). The overhead of allocating storage for the temporary vectors (__t1, __t2, __t3) can also be significant, particularly for small vectors. What we would like to do is evaluate the expression in a single pass, by fusing the four loops into one:

// double *yend is the end of the y vector. 
\[
\begin{array}{l}
\text{do } \\
\quad *yp = (*ap + *bp)/( *cp - *dp); \\
\quad ++ap; ++bp; ++cp; ++dp; \\
\text{while ( ++yp != yend);}
\end{array}
\]

By combining the ideas of expression templates and iterators, it is possible to generate this code automatically, by building an expression object which contains vector iterators rather than placeholders. Here is the declaration for a class DVec, which is a simple "vector of doubles". DVec declares a public type iterT, which is the iterator type used to traverse the vector (for this example, an iterator is a double*). DVec also declares Standard Template Library compliant begin() and end() methods to return iterators positioned at the beginning and end of the vector:

```cpp
class DVec {
private:
  double* data_; 
  int length_; 
public:
  typedef double* iterT; 
  DVec(int n) 
  : length_(n) 
  { data_ = new double[n]; } 
~DVec() 
{ delete [] data_; } 
iterT begin() const 
{ return data_; } 
iterT end() const 
{ return data_ + length_; } 

  template<class A>
  DVec& operator=(DVExpr<A>); 
};
```

Expressions involving DVec objects are of type DVExpr<A>. An instance of DVExpr<A> contains iterators which are positioned in the vectors being combined. DVExpr<A> lets itself be treated as an iterator by providing two methods: double operator*() evaluates the expression using the current elements of all iterators, and operator++() increments all the iterators:

```cpp
template<class A>
```
class DVExpr {
  private:
    A iter_; 
  
  public:
    DVExpr(const A& a) : iter_(a) {} 
    double operator*() const { return *iter_; } 
    void operator++() { ++iter_; } 
}; 

The constructor for DVExpr<A> requires a const A& argument, which contains all the vector iterators and other instance data (e.g., constants) for the expression. This data is stored in the iter_ data member. When a DVExpr<A> is assigned to a Dvec, the Dvec::operator=() method uses the overloaded * and ++ operators of DVExpr<A> to store the result of the vector operation:

template<class A>
DVec& DVec::operator=(DVExpr<A> result) {
  // Get a beginning and end iterator for the vector
  Iterator iter = begin(), endIter = end();
  // Store the result in the vector
  do {
    *iter = *result; // Inlined expression
    ++result;
  } while (++iter != endIter);
  return *this;
}

Binary operations on two subexpressions or iterators A and B are represented by a class DVBinExprOp<A,B,Op>. The operation itself (Op) is implemented by an applicative template class. The prefix ++ and unary * (dereferencing) operators of DVBinExprOp<A,B,Op> are overloaded to invoke the ++ and unary * operators of the A and B instances which it contains.

As before, operators build expression types using nested template arguments. Several versions of each operator are required to handle the combinations in which DVec, DVExpr<A>, and numeric constants can appear in expressions. For a simple example of how expressions are built, consider the code snippet:

DVec a(10), b(10);
y = a+b;

Since both 'a' and 'b' are of type DVec, the appropriate operator+() is:

DVExpr<DVBinExprOp<DVec::iterT,DVec::iterT,DAdd> >
operator+(const DVec& a, const DVec& b) {
  typedef DVBinExprOp<DVec::iterT,DVec::iterT,DAdd> ExprT;
  return DVExpr<ExprT>(ExprT(a.begin(),b.begin()));
}

All the operators which handle DVec types invoke DVec::begin() to get iterators. These iterators are bundled as part of the returned expression object. As before, it’s possible to add to this scheme so that literals can be used in expressions, as well as unary operators (such as x = -y), and functions (sin, exp). It’s also not difficult to write a templated version of DVec and the DVExpr classes, so that vectors of arbitrary types can be used. By using another template technique called "traits", it’s even possible to implement standard type promotions for vector algebra, so that adding a Vec<int> to a Vec<double> produces a Vec<double>, etc. Figure 4 shows the template instance inferred by the compiler for the expression y = (a+b)/(c-d). Although long and ugly type names result, the vector expression is evaluated entirely inline in a single pass by the assignResult() routine.
Benchmark Results

To evaluate the efficiency of the expression templates approach to evaluating vector expressions, benchmark results were obtained comparing its performance to hand-crafted C code, and that of a conventional vector class. The performance was measured for the expression \( y = a + b + c \).

Figure 5 shows the performance of the expression template technique, as compared to hand-coded C, using the Borland C++ V4.0 compiler and an 80486 processor. With a few adjustments of compiler options and tuning of the assignResult() routine, the same assembly code was generated by the expression template technique as hand coded C. The poorer performance for smaller vectors was due to the time spent in the expression object constructors. For longer vectors (eg., length 100) the benchmark results were 95-99.5% the speed of hand coded C.

Figure 6 compares the expression template technique to a conventional vector class which evaluates expressions using temporaries. For short vectors (up to a length of 20 or so), the overhead of setting up temporary vectors caused the conventional vector class to have very poor performance, and the expression template technique was 8-12 times faster. For long vectors, the performance of expression templates settles out to twice that of the conventional vector class.

Conclusions

The technique of expression templates is a powerful and convenient alternative to C style callback functions.
It allows logical and algebraic expressions to be passed to functions as arguments, and inlined directly into the function body. Expression templates also solve the problem of evaluating vector and matrix expressions in a single pass without temporaries.

References


Web Site

More information on these techniques is available from the Blitz++ Project Home Page, at "http://oonumerics.org/blitz/".