<PETE>
Portable Expression Template Engine

PETE Tutorials

Copyright © Los Alamos National Laboratory 1999

PETE Tutorials
Introduction

This document is an introduction to PETE, a library of C++ classes and templates for high-performance numerical computation. PETE, which stands for "Portable Expression Template Engine", uses a technique called *expression templates* to eliminate data copying and the creation of temporary variables. As a result, PETE-based programs can be as efficient as their C or Fortran equivalents.

PETE was designed and implemented by scientists working at the Los Alamos National Laboratory's Advanced Computing Laboratory. These scientists have written and tuned large applications on almost every kind of microprocessor built in the last two decades. PETE therefore encapsulates its authors' understanding of how to get good performance out of modern pipelined architectures and their multi-level memory hierarchies.

PETE is free for non-commercial use (i.e. your tax dollars have already paid for it). You can read its source, extend it to handle platforms or problem domains that the core distribution doesn't cater for, or integrate it with other libraries and your current application, at no cost. For more information, please see the license information included in the appendix.

Of course, nothing is perfect. As of October 1998, some C++ compilers still do not support the full ANSI/ISO C++ standard. Please see PETE's support page for a list of those that do.

A second compiler-related problem is that most compilers produce very long, and very cryptic, error messages if they encounter an error while expanding templated functions and classes, particularly if those functions and classes are nested. Since PETE uses templates extensively, it is not uncommon for a single error to result in several pages of complaints from a compiler. Programs that use templates extensively are also still sometimes slower to compile than programs that do not, and the executables produced by some compilers can be surprisingly large.

The body of this tutorial starts with a discussion of the two key concepts behind PETE: C++ templates, and parse trees. The tutorials that follow show how to apply PETE to user-defined classes, and to third-party classes such as those in the C++ Standard Template Library (STL). You may also wish to look at the PETE web site for updates, bug fixes, and general discussion. As well, an introductory article on PETE appeared in the October 1999 issue of Dr. Dobb's Journal. If you have any questions about PETE or its terms of use, or if you need help downloading or installing PETE, please mail us at `pete@acl.lanl.gov`.

[Home] [Next]

Copyright © Los Alamos National Laboratory 1999

PETE Tutorials
Background and Terminology

Contents:

Introduction

Object-oriented languages like C++ make development easier, but performance tuning harder. The same abstractions that allow programmers to express their ideas compactly also make it hard for compilers to re-order operations, predict how many times a loop will be executed, or re-use memory instead of copying values.

For example, suppose that a program uses a `Vector` class to represent vectors of floating-point values:

```
class Vector
{
  public :
    Vector();                           // default constructor

    Vector(                             // value constructor
        int size,                       // ..size of vector
        float val                       // ..initial element value
    );

    Vector(                             // copy constructor
        const Vector & v                // ..what to copy
    );

    virtual ~Vector();                  // clean up

    float getAt(                        // get an element
        int index                       // ..which element to get
    ) const;

    void setAt(                         // change an element
        int index,                      // ..which element to set
```

```
            float val                       // ..new value for element
        );

        Vector operator+(                   // add, creating a new vector
            const Vector & right            // ..thing being added
        );

        Vector operator*(                   // multiply (create result)
            const Vector & right            // ..thing being multiplied
        );

        Vector & operator=(                 // assign, returning target
            const Vector & right            // ..source
        );

      protected :
        int len_;                           // current length
        float * val_;                       // current values
    };
```

Consider what happens when the following statement is executed:

```
Vector x, a, b, c;
// variable initialization omitted
x = a + b * c;
```

`b*c` creates a new `Vector`, and fills it with the elementwise product of `b` and `c` by looping over the values that those two vectors encapsulate. The call to the addition operator creates another temporary, and executes another loop to fill it. Finally, the call to the assignment operator doesn't create a third temporary, but it does execute a third loop. Thus, this simple statement is equivalent to:

```
Vector x, a, b, c;

// ...initialization...

Vector temp_1;
for (int i=0; i<vectorLength; ++i)
{
    temp_1.setAt(i, b.getAt(i) * c.getAt(i));
}

Vector temp_2;
for (int i=0; i<vectorLength; ++i)
{
    temp_2.setAt(i, a.getAt(i) + temp_1.getAt(i));
}

for (int i=0; i<vectorLength; ++i)
{
    x.setAt(i, temp_2.getAt(i));
}
```

Clearly, if this program was written in C instead of C++, the three loops would have been combined, and the two temporary vectors eliminated:

```
Vector x, a, b, c;
// ...initialization...
for (int i=0; i<vectorLength; ++i)
{
    x.setAt(i, a.getAt(i) + b.getAt(i) * c.getAt(i));
}
```

The optimizations required to turn the three-loop version of this code into its single-loop equivalent are beyond the capabilities of existing commercial compilers. Because operations may involve aliasing---i.e., because an expression like `x=a+b*c` can assign to a vector while also reading from it---optimizers must err on the side of caution, and neither eliminate temporaries nor fuse loops. This has led many programmers to believe that C++ is intrinsically less efficient than C or Fortran 77.

Luckily, this conclusion is wrong. By using templates in a highly-structured way, PETE exposes opportunities for optimization to compilers without sacrificing readability or portability. The result is that modern C++ compilers can deliver the same performance for PETE-based programs as C or Fortran compilers do for equivalent programs written in those lower-level languages.

In order to understand how and why PETE does what it does, it is necessary to understand what C++ templates are, and how PETE (and similar libraries) use templates to encode parse trees.

Templates

Templates were a late addition to C++, but they have increased the power of the language significantly. One way to look at templates is as an improvement over macros. Suppose that you wanted to create a set of classes to store pairs of `int`s, pairs of `float`s, and so on. Without templates, you might define a macro:

```
#define DECLARE_PAIR_CLASS(name_, type_)                            \
class name_                                                         \
{                                                                   \
  public :                                                          \
    name_();                           // default constructor       \
    name_(type_ left, type_ right);    // value constructor         \
    name_(const name_ & right);        // copy constructor          \
    virtual ~name_();                  // destructor                \
    type_ & left();                    // access left element       \
    type_ & right();                   // access right element      \
                                                                    \
  protected :                                                       \
    type_ left_, right_;               // value storage             \
};
```

then use it to create each class in turn:

```
DECLARE_PAIR_CLASS(IntPair, int)
DECLARE_PAIR_CLASS(FloatPair, float)
```

A better way to do this is to declare a template class:

```
template<class DataType>
class Pair
{
  public :
    Pair();                             // default constructor
    Pair(DataType left,                 // value constructor
         DataType right);
    Pair(const Pair<DataType> & right);           // copy constructor
    virtual ~Pair();                    // destructor
    DataType & left();                  // access left element
    DataType & right();                 // access right element

  protected :
    DataType left_, right_;             // value storage
};
```

The keyword `template` tells the compiler that the class cannot be compiled right away, since it depends on an as-yet-unknown data type. When the declarations:

```
Pair<int>   pairOfInts;
Pair<float> pairOfFloats;
```

are seen, the compiler instantiates `Pair` once for each underlying data type. This happens automatically: the programmer does *not* have to create the actual pair classes explicitly by saying:

```
typedef Pair<int> IntPair;              // incorrect!
IntPair pairOfInts;
```

Templates can also be used to define generic functions, as in:

```
template<class DataType>
void swap(DataType & left, DataType & right)
{
    DataType tmp(left);
    left  = right;
    right = tmp;
}
```

Once again, this function can be called with two objects of any matching type, without any further work on the programmer's part:

```
int   i, j;
swap(i, j);

Shape back, front;
swap(back, front);
```

Note that the implementation of `swap()` depends on the actual data type of its arguments having both a copy constructor (so that `tmp` can be initialized with the value of `left`) and an assignment operator (so that `left` and `right` can be overwritten). If the actual data type does not provide either of these, the compiler will report an error.

Note also that `swap()` can be made more flexible by not requiring the two objects to have exactly the same type. The following re-definition of `swap()` will exchange the values of any two objects, provided appropriate assignment and conversion operators exist:

```
template<class LeftType, class RightType>
void swap(LeftType & left, RightType & right)
{
    LeftType tmp(left);
    left  = right;
    right = tmp;
}
```

Finally, the word `class` appears in template definitions because other values, such as integers, can also be used. The code below defines a small fixed-size vector class, but does not fix either its size or underlying data type:

```
template<class DataType, int FixedSize>
class FixedVector
{
  public :
    FixedVector();                        // default constructor
    FixedVector(DataType filler);         // value constructor
    virtual ~FixedVector();               // destructor

    FixedVector(                          // copy constructor
        const FixedVector<DataType, FixedSize> & right
    );

    FixedVector<DataType>                 // assignment
    operator=(
        const FixedVector<DataType, FixedSize> & right
    );

    DataType & operator[](int index);    // element access

  protected :
    DataType storage[FixedSize];         // fixed-size storage
};
```

It is at this point that the possible performance advantages of templated classes start to become apparent. Suppose that the copy constructor for this class is implemented as follows:

```
template<class DataType, int FixedSize>
FixedVector::FixedVector(
    const FixedVector<DataType, FixedSize> & right
){
    for (int i=0; i<FixedSize; ++i)
    {
        storage[i] = right.storage[i];
    }
}
```

When the compiler sees a use of the copy constructor, such as:

```
FixedVector<DataType, FixedSize> first;
// initialization of first vector omitted
FixedVector<DataType, FixedSize> second(first);
```

it knows the size as well as the underlying data type of the objects being manipulated, and can therefore do more optimization than it could if the size was variable.

Automatic instantiation of templates is convenient and powerful, but does have one drawback. Suppose the `Pair` class shown earlier is instantiated in one source file to create a pair of `int`s, and in another source file to create a pair of `Shape`s. The compiler and linker could:

1. treat the two instantiations as completely separate classes;
2. detect and eliminate redundant instantiations; or
3. avoid redundancy by not instantiating templates until the program as a whole was being linked.

The first of these can lead to very large programs, as a commonly-used template class may be expanded dozens of times. The second is difficult to do, as it involves patching up compiled files as they are being linked. Most recent versions of C++ compilers are therefore taking the third approach, but POOMA II users should be aware that older versions might still produce much larger executables than one would expect.

The last use of templates that is important to this discussion is template methods. Just as templated functions are instantiated for different types of arguments, so too are templated methods instantiated for a class when and as they are used. Suppose a class called `Example` is defined as follows:

```
class Example
{
  public :
    Example();                          // default constructor
    virtual ~Example();                 // destructor

    template<class T>
    void foo(T object)
    {
        // some operation on object
    }
};
```

Whenever the method `Example::foo()` is called with an object of a particular type, the compiler instantiates it for that type. Thus, both of the following calls are legal:

```
Example e;
Shape box;
e.foo(5);                               // instantiate for int
e.foo(box);                             // instantiate for Shape
```

Representing Parse Trees

Parse trees are commonly used by compilers to store the essential features of the source of a program. The leaf nodes of a parse tree consist of atomic symbols in the language, such as variable names or numerical constants. The parse tree's intermediate nodes represent ways of combining those values, such as arithmetic operators and `while` loops. For example, the expression `-B + 2 * C` could be represented by the parse tree shown in Figure 1

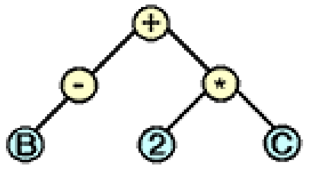

Figure 1: A Simple Parse Tree

Parse trees are often represented textually using prefix notation, in which the non-terminal combiner and its arguments are strung together in a parenthesized list. For example, the expression `-B + 2 * C` can be represented as `(+ (- B) (* 2 C))`.

What makes all of this relevant to high-performance computing is that the expression `(+ (- B) (* 2 C))` could equally easily be written `BinaryOp<Add, UnaryOp<Minus, B>, BinaryOp<Multiply, Scalar<2>, C>>`: it's just a different notation. However, this notation is very similar to the syntax of C++ templates --- so similar, in fact, that it can actually be implemented given a careful enough set of template definitions. As discussed earlier, by providing more information to the optimizer as programs are being compiled, template libraries can increase the scope for performance optimization.

Any facility for representing expressions as trees must provide:

- a representation for leaf nodes (operands);
- a way to represent operations to be performed at the leaves (i.e. functions on individual operands);
- a representation for non-leaf nodes (operators);
- a way to represent operations to be performed at non-leaf nodes (i.e. combiners);
- a way to pass information (such as the function to be performed at the leaves) downward in the tree; and
- a way to collect and combine information moving up the tree.

C++ templates were not designed with these requirements in mind, but it turns out that they can satisfy them. The central idea is to use the compiler's representation of type information in an instantiated template to store operands and operators. For example, suppose that a set of classes have been defined to represent the basic arithmetic operations:

```
struct AddOp
{
    static inline double apply(const double & left, const double & y)
    {
        return x + y;
    }
};

struct MulOp
```

```
    {
        static inline double apply(const double & left, const double & y)
        {
            return x * y;
        }
    };

    // ...and so on...
```

Note the use of the keyword `struct`; this simply signals that everything else in these classes---in particular, their default constructors and their destructors---are `public`.

Now suppose that a templated class `BinaryOp` has been defined as follows:

```
    template<class Operator, class Vector, class RHS>
    class BinaryOp
    {
      public :
        // empty constructor will be optimized away, but triggers
        // type identification needed for template expansion
        BinaryOp(
            Operator op,
            const Vector & leftArg,
            const RHS    & rightArg
        ) : left_(leftArg),
            right_(rightArg)
        {}

        // empty destructor will be optimized away
        ~BinaryOp()
        {}

        // calculate value of expression at specified index by recursing
        inline double apply(int i)
        {
            return Operator::apply(leftArg.apply(i), rightArg.apply(i));
        }

      protected :
        const Vector & left_;
        const RHS    & right_;
    };
```

If `b` and `c` have been defined as `Vector`, and if `Vector::apply()` returns the vector element at the specified index, then when the compiler sees the following expression:

```
    BinaryOp<MulOp, Vector, Vector>(MulOp(), b, c).apply(3)
```

it translates the expression into `b.apply(3) * c.apply(3)`. The creation of the intermediate instance of `BinaryOp` is optimized away completely, since all that object does is record a couple of references to arguments.

Why to go all this trouble? The answer is rather long, and requires a few seemingly-pointless steps. Consider what happens when the complicated expression above is nested inside an even more complicated expression, which adds an element of another vector `a` to the original expression's result:

```
BinaryOp< AddOp,
          Vector,
          BinaryOp< MulOp, Vector, Vector >
        >(a, BinaryOp< MulOp, Vector, Vector >(b, c)).apply(3);
```

This expression calculates `a.apply(3) + (b.apply(3) *c.apply(3))`. If the expression was wrapped in a `for` loop, and the loop's index was used in place of the constant `3`, the expression would calculate an entire vector's worth of new values:

```
BinaryOp< AddOp,
          Vector,
          BinaryOp< MulOp, Vector, Vector > >
        expr(a, BinaryOp< MulOp, Vector, Vector >(b, c));
for (int i=0; i<vectorLength; ++i)
{
  double tmp = expr.apply(i);
}
```

The possible nesting of `BinaryOp` inside itself is the reason that the `BinaryOp` template has two type parameters. The first argument to a `BinaryOp` is always a `Vector`, but the second may be either a `Vector` or an expression involving `Vector`s.

The code above is not something any reasonable person would want to write. However, having a compiler create this loop and its contained expression automatically is entirely plausible. The first step is to overload addition and multiplication for vectors, so that `operator+(Vector,Vector)` (and `operator*(Vector,Vector)`) instantiates `BinaryOp` with `AddOp` (and `MulOp`) as its first type argument, and invokes the `apply()` method of the instantiated object. The second step is to overload the assignment operator `operator=(Vector,Vector)` so that it generates the loop shown above:

```
template<class Op, T>
Vector & operator=(
    Vector & target,
    BinaryOp<Op> & expr
){
    for (int i=0; i<vectorLength; ++i)
    {
        target.set(i, expr.apply(i));
    }
    return target;
}
```

With these operator definitions in play, the simple expression:

```
Vector x, a, b, c;
// ...initialization...
x = a + b * c;
```

is automatically translated into the efficient loop shown above, rather than into the inefficient loops shown earlier. The expression on the right hand side is turned into an instance of a templated class whose type encodes the operations to be performed, while the implementation of the assignment operator causes that expression to be evaluated exactly once for each legal index. No temporaries are created, and only a single loop is executed.

Looking Ahead

Of course, an industrial-strength implementation of these ideas requires definitions that are considerably more complicated than the ones shown in the previous section. For a start, `BinaryOp` and its kin are not defined directly on any one class `Vector`. It isn't even defined for `Vector<T>`, but rather for a wrapper class. This class expects nothing from its contained class except an `apply` method capable of turning an index into a value. This allows users to integrate their own classes with PETE simply by providing the required method. Similarly, the classes that PETE defines to represent unary and binary operators are considerably more flexible than the ones shown above.

One of the idioms used by PETE that hasn't been shown above is the *tag class*. A tag class has no methods, and contains no data; its only reason for existing is as a flag to the C++ compiler during template expansion. A mutually exclusive set of tag classes is therefore the compile-time equivalent of an enumeration. PETE uses tag classes to identify operators, the way in which operands are referenced (i.e. directly or through iterators and other intermediators), and so on.

Another idiom used in PETE is the traits class, which depends on a feature of ANSI C++ called partial specialization. When a C++ compiler instantiates a template, it tries to choose the best possible match for the arguments it is given. For example, suppose that both of the following definitions are in scope when the objects `fred` and `jane` are created:

```
template<class T>
class Example
{
    enum { tag = 123; }
};

template<>
class Example<int>
{
    enum { tag = 456; }
};

Example<int>   fred;
Example<float> jane;
```

As you would expect, `fred`'s `tag` has the value 456, while `jane`'s has the generic value 123: the compiler chooses the most specific type possible.

This facility can be used to create lookup tables. For example, suppose we want to encode the types of the results of arithmetic operations involving an arbitrary mix of `int` and `double` arguments. The following definitions do the trick:

```
// generic case
template<class Left, class Right>
class TypeEncoding
```

```
    {
        // empty: no generic result possible
    };

    // int op int => int
    template<>
    class TypeEncoding<int, int>
    {
        typedef int Result_t;
    };

    // int op double => double
    template<>
    class TypeEncoding<int, double>
    {
        typedef double Result_t;
    };

    // double op int => double
    template<>
    class TypeEncoding<double, int>
    {
        typedef double Result_t;
    };

    // double op double => double
    template<>
    class TypeEncoding<double, double>
    {
        typedef double Result_t;
    };
```

We can now overcome one of the biggest shortcomings of C++ templates, and automatically generate the correct result type of a templated expression:

```
    template<class Left, class Right>
    TypeEncoding<Left, Right>::Result_t
    add(const Left & left, const Right & right)
    {
        return left + right;
    }
```

If `add()` is called with two `int` arguments, the compiler will know that that particular instantiation is going to return an `int`. If it is called with one or two `double` arguments, the compiler will know it is going to return a `double`. By specializing `TypeEncoding` for other mixes of types, a library like PETE can tell the compiler the result type of any expression over any mix of types. In particular, if a new class such as `Complex`, `Quaternion`, or `Color` is added, the compiler can be told what the result of (for example) multiplying a `Color` by a `float` is, without anything else in the library having to be changed.

`TypeEncoding` is an example of a traits class. Each specialization of the class defines a `typedef` with a particular name (in this case, `Result_t`). The class designer could also specify that `TypeEncoding`'s specializations had to define such things as constant strings:

```
    // int op double => double
    template<>
    class TypeEncoding<int, double>
    {
        typedef double Result_t;

        static const char * const Signature = "{int,double}=>double";
    };

    // other classes contain similar definitions
```

to help programs print debugging information:

```
    template<class Left, class Right>
    TypeEncoding<Left, Right>::Result_t
    add(const Left & left, const Right & right)
    {
        cout << TypeEncoding<Left, Right>::Signature << endl;
        return left + right;
    }
```

In general, if the classes in the set associated with a trait all adhere to some conventions regarding name definitions, then traits classes can be used to implement compile-time polymorphism. Another way to think of this is that each class in a set of traits classes provides different set of answers to a pre-defined set of questions.

Since writing a dozen or more specializations of classes like `TypeEncoding` and `BinaryOp` by hand would be tedious, time-consuming, and error-prone, PETE provides some simple command-line tools that can generate the required C++ code automatically. The first tutorial shows how to use these tools to integrate a simple 3-element vector class into PETE. Subsequent tutorials show the steps required to integrate more complex classes, such as the vectors and lists of the Standard Template Library (STL), and how to provide additional operators and combiners.

[Prev] [Home] [Next]

Copyright © Los Alamos National Laboratory 1999

PETE Tutorial 1
Incorporating a Simple Vector Class

Contents:

Introduction

This tutorial shows how to integrate a simple class representing 3-element vectors into PETE. The source files for this example are included in the `examples/Vec3` directory of the PETE distribution. These files are:

- `Vec3.h`: defines the `Vec3` class on which the example is based. This file also defines or specializes the template classes needed to integrate `Vec3` with PETE. These extra definitions will be discussed below.
- `Vec3Defs.in`: definitions needed to automatically generate the template classes required to integrate `Vec3` into PETE. This file is processed by the `MakeOperators` tool discussed below.
- `Vec3Operators.h`: the file generated by `MakeOperators` based on the definitions in `Vec3Defs.in`. The vector class definition file `Vec3.h` `#include`s this file, so that PETE-based programs only need to `#include` `Vec3.h`, rather than both `Vec3.h` and `Vec3Operators.h`.
- `Vec3.cpp`: a short program that shows how to construct expressions using `Vec3` and PETE together.
- `makefile`: rebuilds the example.

The Starting Point

The starting point for this tutorial is the 3-element vector class defined in `Vec3.h`. Most of this class's declaration is unremarkable. Each instance of the class contains a 3-element array of integers; the class's default constructor initializes their values to 0, while a non-default constructor can be used to give them particular initial values. A copy constructor is also provided, as are assignment operators taking either scalar or vector values. Finally, two versions of `operator[]` are provided, so that both constant and non-constant vectors can be indexed, and a `print()` method is defined for `operator<<` and other I/O routines to use. `operator<<` is overloaded further down, on lines 115-119.

If you do not understand all of the definitions on lines 22-58 and 78-94 of `Vec3.h`, you may wish to become more familiar with C++ before proceeding with these tutorials.

In order to understand the particulars of this tutorial, it is necessary to know about some of the indirection classes that PETE uses.

The most important of these is a general wrapper called `Expression<>`. It exists to wrap `UnaryNode`, `BinaryNode` and `TrinaryNode` so that they can all be captured during template expansion by a single type, namely `Expression<T>`. As we shall see below, the `Expression` template also serves to distinguish PETE expressions from other expressions, so that the compiler will not inadvertently mix PETE expressions with normal arithmetic.

PETE's second indirection class is called `MakeReturn`. For any type `T`, `MakeReturn<T>::Expression_t` is a `typedef` that produces the type of value returned by expressions on `T`. In the general case, this is simply Expression<T>, i.e. `MakeReturn` just wraps the expression produced by an operator so that it can be used inside other operators. The POOMA library overrides `MakeReturn<T>` (by specializing it explicitly) so that expressions involving POOMA arrays generate new arrays. This technique is similar to the standard C++ idiom of having a framework define one or more virtual methods with empty bodies, and call them at specified times, so that users can derive from the framework classes and override those virtual methods to insert their own code in the framework's processing stream.

Extra Definitions Required for Integration

In order to use `Vec3` with PETE, we must provide three things:

1. A description of the `Vec3` class.
2. A way to extract values from instances of `Vec3`.
3. A way to assign to a `Vec3` from a PETE expression.

These three issues are discussed in order below.

In addition, this example shows how to add new capabilities to PETE by creating a mechanism for counting the number of instances of `Vec3` involved in an expression. The same kind of mechanism can be used to do such things as check that all of the vectors in an expression have the same length before starting evaluation of that expression.

Making Leaves for the Parse Tree

PETE uses a traits class called `CreateLeaf` to record information about the leaf types on which it operates. Each specialization of `CreateLeaf` must be able to answer two questions:

1. What is the type of the leaf?
2. How can the program make an instance of this leaf?

The first question is answered by providing a `typedef` for the name `Leaf_t`. In our example, we want to have access to `Vec3`'s member functions. However, `Vec3` has deep copy semantics so we don't want to store actual `Vec3` objects at the leaves, thereby making copies and negating most of the benefit of expression templates. Instead, we store a `Vec3&`. This is accomplished by wrapping the `Vec3` class in a `Reference` wrapper. (By default, PETE stores leaves by value, which is appropriate for leaves that hold iterators. In this case we would not have to make use of the `Reference` wrapper.)

Once we have done this, the rest of PETE can be written using expressions like `CreateLeaf<T>::Leaf_t`. This allows PETE to work with classes that are added later, in the same way that making a function `virtual` allows programs that use a library to add new functionality without re-writing old code.

Making a leaf is a little bit trickier. Every specialization of `CreateLeaf` must define a `static` method called `make()` that takes something of the specialization's input type as an argument, and returns something of its leaf type. This method must be `static` so that it can be called without an instance of `CreateLeaf<Vec3>` ever having been created

As with most traits classes, the specializations of `CreateLeaf` are never instantiated, but instead exist only to answer questions. In this example, the input to `make()` is a constant reference to a `Vec3`, which the function simply returns wrapped in a `Reference<Vec3>` object. In the case of an STL list, the argument would have type `List<T>`, but the return type `Leaf_t` might be an iterator type.

```
template<>
struct CreateLeaf<Vec3>
{
  typedef Reference<Vec3> Leaf_t;
  inline static
  Leaf_t make(const Vec3 &a) { return Leaf_t(a); }
};
```

Operating on Leaves

Our next task is to provide a way for PETE expressions to extract values from `Vec3`s. This has to be done in a generic way, so that (for example) scalars can return the same value each time they are queried, while STL lists can be accessed through bidirectional iterators and `Vec3`s can be accessed by integer indexing.

PETE's solution to this problem is to require programmers to specialize the traits class `LeafFunctor` for the combination of their leaf class and a tag class called `EvalLeaf1`. `EvalLeaf1` is a simple class defined by PETE, whose only purpose is to contain the single index PETE needs in order to evaluate an expression. (`EvalLeaf2` signals that a doubly-indexed expression is being evaluated, and so on up to `EvalLeaf7`.)

The specialization of `LeafFunctor`, shown below, does two things. First, it defines the type of the result of the expression as `Type_t`. Second, it defines a `static` method called `apply()`, which uses the index stored in its `EvalLeaf1` argument and returns the corresponding element of the `Vec3`:

```
127  template<>
128  struct LeafFunctor<Vec3, EvalLeaf1>
129  {
130    typedef int Type_t;
131    static Type_t apply(const Vec3 &a, const EvalLeaf1 &f)
132      { return a[f.val1()]; }
133  };
```

Assigning to Vectors

The last step in making `Vec3` PETE-compatible is to provide a way for PETE to assign to a `Vec3` from an arbitrary expression. This is done by overloading `operator=` to take a PETE expression as input, and copy values into its owner:

```
064    template<class RHS>
065    Vec3 &operator=(const Expression<RHS> &rhs)
066    {
067      d[0] = forEach(rhs, EvalLeaf1(0), OpCombine());
068      d[1] = forEach(rhs, EvalLeaf1(1), OpCombine());
069      d[2] = forEach(rhs, EvalLeaf1(2), OpCombine());
070
071      return *this;
072    }
```

The first thing to notice about this method is that it is templated on an arbitrary class `RHS`, but its single formal parameter has type `Expression<RHS>`. This combination means that the compiler can match it against anything that is wrapped in the generic PETE template `Expression`, and *only* against things that are wrapped in that way. The compiler cannot match against `int`, `complex<short>`, or `GreatAuntJane_t`, since these do not have the form `Expression<RHS>` for some type `RHS`.

The `forEach` function is used to traverse expression trees. The first argument is the expression. The second argument is the leaf tag denoting the operation applied at the leaves. The third argument is a combiner tag, which is used to combine results at non-leaf nodes. By passing `EvalLeaf1(0)` in line 67, we are indicating that we want the `Vec3`s at the leaves to return the element at index 0. The `LeafFunctor<Scalar<T>, EvalLeaf1>` (defined inside of PETE) ensures that scalars return their value no matter the index. While `EvalLeaf1` obtains values from the leaves, `OpCombine` takes these values and combines them according to the operators present at the non-leaf nodes. The result is that line 67 evaluates the expression on the right side of the assignment operator at index 0. Line 68 does this at index 1, and so on. Once evaluation is complete, `operator=` returns the `Vec3` to which values have been assigned, in keeping with normal C++ conventions.

Counting Vectors

We could stop at this point, but in order to show off PETE's flexibility, we will finish by defining a new leaf tag that counts the number of `Vec3`s in an arbitrary expression. The required definitions, on lines 152-168 of `Vec3.h`, are:

```
141  struct CountLeaf { };
142
143  template<>
144  struct LeafFunctor<Vec3, CountLeaf>
```

```
{
  typedef int Type_t;
  static Type_t apply(const Vec3 &, const CountLeaf &)
    { return 1; }
};

template<class T>
struct LeafFunctor<T, CountLeaf>
{
  typedef int Type_t;
  static Type_t apply(const T &a, const CountLeaf &)
    { return 0; }
};
```

`CountLeaf` is an empty tag class, whose only purpose is to identify the operation we wish to carry out. `LeafFunctor` is then specialized separately for `CountLeaf` on both `Vec3` and the generic type `T`. Applying the first specialization returns 1, since it wraps a single `Vec3`. Applying the second specialization returns 0, since nothing else counts as a vector.

Making Operators

We have now provided almost everything that PETE needs in order to operate on our `Vec3` class. All that remains is several thousand lines of templated operator and function definitions. Luckily, these can be generated automatically from a few lines of information.

The file `Vec3Defs.in` stores exactly that information, and is used by PETE's `MakeOperators` tool to generate the 3740 lines of `Vec3Operators.h`. The special notation `"[n]"` is replaced by the digits 1, 2, and so on to distinguish multiple uses of the same argument. Thus, `"class T[n]"` becomes `"class T1"`, `"class T2"`, and so on as needed. The entire specification file is:

```
classes
-----
  ARG   = ""
  CLASS = "Vec3"
```

Two values are specified for each of the classes for which definitions are to be generated:

- `ARG`: how to make the template arguments needed to define instances of this class. `Vec3`'s `ARG` field is empty, since it is not a templated class. Giving `class T[n]` for the `ARG` field for a general `Expression` causes instances of `Expression` to be filled in with `class T1`, `class T2`, and so on.
- `CLASS`: the name of the class itself.

The command used to build an operator definition file from this input specification is:

```
MakeOperators --classes Vec3Defs.in --guard VEC3OPS_H --o Vec3Operators.h
```

`Vec3Defs.in` is the file shown above. The symbol `VEC3OPS_H` is copied into the output to guard against multiple inclusion, so that the output file has the form:

```
#ifndef VEC3OPS_H
#define VEC3OPS_H

// ...contents of file...

#endif // VEC3OPS_H
```

Further information about `MakeOperators` and its command-line argument can be found on its man page.

Using These Definitions

With all this out of the way, we can now write arithmetic expressions that use `Vec3`, and rely on PETE to optimize them for us. The file `Vec3.cpp` shows some of the possibilities. The simplest example is straightforward addition and assignment:

```
a = b + c;
```

which would automatically be translated into something equivalent to:

```
a[0] = b[0] + c[0];
a[1] = b[1] + c[1];
a[2] = b[2] + c[2];
```

This snippet would make use of the overloaded `operator+()` generated by the `MakeOperators` tool in the `Vec3Operators.h` file and the assignment operator defined above.

This expression could be made much more complex, and PETE would still eliminate redundant temporaries or loops. One such expression is:

```
a = sqrt(b*b + c*c);
```

which would automatically be translated into something equivalent to:

```
a[0] = sqrt(b[0]*b[0] + c[0]*c[0]);
a[1] = sqrt(b[1]*b[1] + c[1]*c[1]);
a[2] = sqrt(b[2]*b[2] + c[2]*c[2]);
```

since PETE provides appropriately-templated overloadings of the standard mathematical functions like `sqrt()` and `acos()` as well as overloadings of unary and binary operators.

The next two examples in `Vec3.cpp` make use of an expression (in this case, the addition of `b` and `c`) that has been recorded for delayed evaluation. In order to do this, the programmer must explicitly specify the type of the expression being stored, but once this has been done, that expression can be re-used any number of times. The statement that creates the expression is:

```
const Expression<BinaryNode<OpAdd, Vec3, Vec3> > &expr1 = b + c;
```

Its first use is as a source for assignment:

```
d = expr1;
```

It can also be passed to PETE's explicit evaluation function, called `forEach()`, along with the `CountLeaf` defined earlier, and PETE's built-in `SumCombine` tag class, in order to count the number of instances of `Vec3` that appear in the expression:

```
int num = forEach(expr1, CountLeaf(), SumCombine());
```

Note the parentheses after `CountLeaf` and `SumCombine`. C++ does not allow raw type names to be used to instantiate templates; instead, the program must create an unnamed instance of each tag class by invoking their default constructors. Since these classes contain no data, and their instances are not used inside `forEach()`, the compiler optimizes away all of the associated code.

The remaining examples in `Vec3.cpp` use `CountLeaf` to inspect more complicated expressions.

Summary

This tutorial has shown how to integrate a simple class into PETE's expression template framework, so that compilers can optimize expressions involving instances of that class. The steps required are:

- specializing `CreateLeaf` to tell PETE how to store instances of the user class in parse trees;
- specializing `LeafFunctor` to extract information from these leaf nodes;
- overloading `operator=` to read values from expressions and assign them to instances of the user-defined class; and
- using PETE's `MakeOperators` tool to generate specialized overloadings of C++'s unary, binary, and ternary operators to work with the user-defined class.

In addition, this tutorial showed how to extend PETE's leaf and combiner tags to calculate values on expression trees during compilation. The next tutorial will look at how to extend the PETE framework itself to synthesize new types.

Source Files

`Vec3.h`

```
#ifndef PETE_EXAMPLES_VEC3_VEC3_H
#define PETE_EXAMPLES_VEC3_VEC3_H

//-----------------------------------------------------------------------------
// Include files
//-----------------------------------------------------------------------------

#include "PETE/PETE.h"

#include <iostream.h>

//-----------------------------------------------------------------------------
//
// CLASS NAME
//    Vec3
//
// DESCRIPTION
//    A "tiny" three-element expression-template (ET) array class.
//
//-----------------------------------------------------------------------------

class Vec3
{
public:

  //---------------------------------------------------------------------------
  // Constructors and Destructor
  //---------------------------------------------------------------------------

  Vec3() { d[0] = d[1] = d[2] = 0; }

  Vec3(int i, int j, int k)
  {
    d[0] = i; d[1] = j; d[2] = k;
  }

  Vec3(const Vec3 &v)
  {
    d[0] = v.d[0];  d[1] = v.d[1];  d[2] = v.d[2];
  }

  ~Vec3() {}

  //---------------------------------------------------------------------------
  // Vec3 and scalar assigment operators
  //---------------------------------------------------------------------------

  Vec3 &operator=(const Vec3 &v)
  {
    d[0] = v.d[0];  d[1] = v.d[1];  d[2] = v.d[2];
    return *this;
  }

  Vec3 &operator=(int i)
  {
    d[0] = d[1] = d[2] = i;
    return *this;
  }

  //---------------------------------------------------------------------------
  // Assignment operator taking expression:
  //---------------------------------------------------------------------------

  template<class RHS>
  Vec3 &operator=(const Expression<RHS> &rhs)
  {
    d[0] = forEach(rhs, EvalLeaf1(0), OpCombine());
    d[1] = forEach(rhs, EvalLeaf1(1), OpCombine());
    d[2] = forEach(rhs, EvalLeaf1(2), OpCombine());

    return *this;
  }

  //---------------------------------------------------------------------------
  // Indexing operators
  //---------------------------------------------------------------------------

  int &operator[](int i)      { return d[i]; }
  int operator[](int i) const { return d[i]; }

  //---------------------------------------------------------------------------
  // Print method used by operator<< free function.
  //---------------------------------------------------------------------------

  void print(ostream &os) const
  {
    os << "{" << d[0] << "," << d[1] << "," << d[2] << "}";
  }

private:

  // The underlying complicated data structure

  int d[3];

};

//-----------------------------------------------------------------------------
// We need to specialize CreateLeaf<T> for our class, so that operators
// know what to stick in the leaves of the expression tree.
//-----------------------------------------------------------------------------

template<>
struct CreateLeaf<Vec3>
{
  typedef Reference<Vec3> Leaf_t;
  inline static
  Leaf_t make(const Vec3 &a) { return Leaf_t(a); }
};

//-----------------------------------------------------------------------------
// ostream inserter for Vec3s
```

```
//-----------------------------------------------------------------------------

ostream &operator<<(ostream &os, const Vec3 &a)
{
  a.print(os);
  return os;
}

//-----------------------------------------------------------------------------
// Specialization of LeafFunctor class for applying the EvalLeaf1
// tag to a Vec3. The apply method simply returns the array
// evaluated at the point.
//-----------------------------------------------------------------------------

template<>
struct LeafFunctor<Vec3, EvalLeaf1>
{
  typedef int Type_t;
  static Type_t apply(const Vec3 &a, const EvalLeaf1 &f)
    { return a[f.val1()]; }
};

//-----------------------------------------------------------------------------
// Specialization of LeafFunctor class for applying the CountLeaf
// tag to a Vec3. The apply method simply returns 1 for a Vec3 and 0 for
// anything else.
//-----------------------------------------------------------------------------

struct CountLeaf { };

template<>
struct LeafFunctor<Vec3, CountLeaf>
{
  typedef int Type_t;
  static Type_t apply(const Vec3 &, const CountLeaf &)
    { return 1; }
};

template<class T>
struct LeafFunctor<T, CountLeaf>
{
  typedef int Type_t;
  static Type_t apply(const T &a, const CountLeaf &)
    { return 0; }
};

// We put this include at the end because
// the operators can't be defined until after Vec3 and
// CreateLeaf<Vec3> have been defined.
// (Since Vec3 isn't templated the operators aren't just
// templates.)

#include "Vec3Operators.h"

#endif // PETE_EXAMPLES_VEC3_VEC3_H
```

Vec3Defs.in

```
classes
-----
```

```
  ARG   = ""
  CLASS = "Vec3"
```

Vec3.cpp

```
#include "Vec3.h"

int main()
{
  Vec3 a, b, c;

  c = 4;

  b[0] = -1;
  b[1] = -2;
  b[2] = -3;

  a = b + c;

  cout << a << endl;

  Vec3 d;
  const Expression<BinaryNode<OpAdd, Vec3, Vec3> > &expr1 = b + c;
  d = expr1;
  cout << d << endl;

  int num = forEach(expr1, CountLeaf(), SumCombine());
  cout << num << endl;

  const Expression<BinaryNode<OpAdd, Vec3,
    BinaryNode<OpMultiply, Scalar<int>, Vec3> > > &expr2 = b + 3 * c;
  num = forEach(expr2, CountLeaf(), SumCombine());
  cout << num << endl;

  const Expression<BinaryNode<OpAdd, Vec3,
    BinaryNode<OpMultiply, Vec3, Vec3> > > &expr3 = b + c * d;
  num = forEach(expr3, CountLeaf(), SumCombine());
  cout << num << endl;
}
```

[Prev] [Home] [Next]

Copyright © Los Alamos National Laboratory 1999

PETE Tutorial 2
Integrating with the Standard Template Library

Contents:

Introduction

This tutorial shows how to use PETE to manipulate expressions involving classes taken from pre-existing libraries---in this case, from the C++ Standard Template Library (STL). The STL's `vector` class is a generic resizeable one-dimensional array, which provides fast, constant-time element access in exchange for making extension expensive. Like the STL's other container classes, `vector` is used both as-is, and as a basis for more specialized data structures, such as fixed-size queues and stacks. This tutorial will show how to use PETE to improve the performance of elementwise expressions on numeric vectors, and how to automatically determine whether two or more vectors conform (i.e. have the same length).

The source files for this example are included in the `examples/Vector` directory of the PETE distribution. These files are:

- `Eval.h`: extends PETE's standard definitions to accommodate expressions involving `vector`s.
- `VectorDefs.in`: definitions needed to automatically generate the overloaded operators required to integrate `vector`s into PETE. Like `Vec3Defs.in` in the first tutorial, this file is processed by the `MakeOperators` tool to create a header file.
- `VectorOperators.h`: the file generated by `MakeOperators` based on the definitions in `VectorDefs.in`. The file `Eval.h` `#include`s `VectorOperators.h`, so that PETE-based programs only need to `#include` `Eval.h`, rather than both `Eval.h` and `VectorOperators.h`.
- `Vector.cpp`: a short program that shows how to use the definitions in the header files described above to create values during compilation.
- `makefile`: rebuilds the example.

Required Definitions

Most of the definitions required to integrate `vector` with PETE are generated automatically by `MakeOperators` using the information in `VectorDefs.in`. The file `Eval.h` contains the few extra definitions that must be written by hand. Of these, the most important is the function `evaluate()`, on lines 102-128. This function's arguments are:

- a `vector<T, Allocator>` (for some type `T` and some allocator `Allocator`);
- an operator specified by an instance of a PETE operator tag class; and
- a wrapped PETE expression, called `rhs`, the values of which are to be assigned to the elements of the given `vector`.

The overloaded assignment operators in `VectorOperators.h` must be able to find an `evaluate()` to match every assignment in the user's program. PETE's protocol therefore requires that every class used on the left-hand-side (LHS) in assignment expressions define a function with this name and signature.

The first thing `evaluate()` does is check that its target and expression conform, i.e. have the same length. It does this by applying the PETE function `forEach()` with a user-defined functor `SizeLeaf()` to the expression `rhs` on line 105. This functor returns `true` if the size of each `vector` at a leaf of PETE's expression tree matches the size of the LHS vector, which is passed as a contructor argument to the `SizeLeaf`. We use an `AndCombine` object to combine results at non-leaf nodes. In order for the right-hand-side (RHS) to conform, all leaves must agree. The definition of `SizeLeaf` is discussed below.

If its expression and target conform, `evaluate()` carries out the required assignment by looping over their mutual index range (line 110). For each index value, `forEach()` is used to evaluate the expression, and the given assignment operator's overloaded `operator()` method is used to transfer those values to the target vector. Note that a direct assignment is not used, since the assignment could involve `+=`, `|=`, or any one of C++'s other combine-and-assign operators.

The two other definitions that must be present for PETE to work with `vector` are specializations of `CreateLeaf` and `LeafFunctor`. The first one of these specializations, on lines 29-35, specifies that we store references to the `vector` objects themselves at the leaves of the PETE expression tree.

The specialization of `LeafFunctor` for `vector` and `EvalLeaf1` on lines 86-95 is what tells PETE how to extract elements from a `vector`. The '1' in `EvalLeaf1` indicates that the class is used to access singly-indexed structures; similar classes called `EvalLeaf2`, `EvalLeaf3`, and so on are used to access more complex classes.

Given an instance of `EvalLeaf1`, and a `vector`, this specialization of `LeafFunctor` defines a `inline static` method called `apply()`, which takes the index value stored in the `EvalLeaf1` and fetches the corresponding `vector` element. Making this method `static` means that instances of `LeafFunctor` never have to be created, while making it `inline` ensures that the compiler will replace uses of it with its body. Thus, specializations of `LeafFunctor` present container element access to compilers in a uniform way, without any efficiency cost.

Finally, the `VectorDefs.in` file, which is used to generate the standard operator overloadings for `vector`, is identical to the one used in the previous tutorial, except for a substitution of `vector<T[n]>` for `Vec3`. (Recall that the `[n]` notation is a placeholder for an automatically generated index, so that if `vector` is used as a formal parameter two or more times, the instances will be labeled `vector<T1>`, `vector<T2>`, and so on.)

Checking Conformance

Our only remaining task is to implement the conformance checking used by `evaluate()`. The first step is to write a simple functor that holds a size to compare against and contains a method to return whether an argument matches this value. This is the `SizeLeaf` class appearing in lines 43-55 of `Eval.h`.

Once we've created the functor class, we then need to tell PETE how to apply it at the leaves of the expression tree. We know that these leaves can consist of either `Scalar` or `vector` objects. We therefore need to supply two `LeafFunctor` specializations. The first, in lines 57-68 works for scalars and always returns `true` since scalars always conform. The second, in lines 70-79, uses `SizeLeaf`'s `operator()` function to compare the size of the `vector` object stored at a leaf with the reference value.

Using Vectors with PETE

The program in `Vector.cpp` shows how to use the definitions given above. The program starts by creating and initializing five vectors. It then calls PETE's `assign()` function to evaluate expressions involving vectors and scalars, and copy their values into other vectors. Note that `assign()` must be called by name because the STL pre-defines `operator=` for all of its types.

Summary

This tutorial has shown how to extend PETE so that it can handle expressions involving classes taken from a pre-existing library---in this case, the Standard Template Library. The definitions required to do this are simple and well-defined, as are the definitions required to perform other calculations (in this case, conformance checking) on those pre-defined classes.

Source Files

`Eval.h`

```
001  #ifndef PETE_EXAMPLES_VECTOR_EVAL_H
002  #define PETE_EXAMPLES_VECTOR_EVAL_H
```

```

//-----------------------------------------------------------------------------
// Includes
//-----------------------------------------------------------------------------

#include <iostream.h>
#include <vector.h>
#include "PETE/PETE.h"
#include "VectorOperators.h"

//-----------------------------------------------------------------------------
// This file contains several class definitions that are used to evaluate
// expressions containing STL vectors.  The main function defined at the end
// is evaluate(lhs,op,rhs), which allows the syntax:
// vector<int> a,b,c;
// evaluate(a,OpAssign(),b+c);
//
// evaluate() is called by all the global assignment operator functions
// defined in VectorOperators.h
//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------
// We need to specialize CreateLeaf<T> for our class, so that operators
// know what to stick in the leaves of the expression tree.
//-----------------------------------------------------------------------------

template<class T, class Allocator>
struct CreateLeaf<vector<T, Allocator> >
{
  typedef Reference<vector<T> > Leaf_t;
  inline static
  Leaf_t make(const vector<T, Allocator> &a) { return Leaf_t(a); }
};

//-----------------------------------------------------------------------------
// We need to write a functor that is capable of comparing the size of
// the vector with a stored value. Then, we supply LeafFunctor specializations
// for Scalar<T> and STL vector leaves.
//-----------------------------------------------------------------------------

class SizeLeaf
{
public:

  SizeLeaf(int s) : size_m(s) { }
  SizeLeaf(const SizeLeaf &model) : size_m(model.size_m) { }
  bool operator()(int s) const { return size_m == s; }

private:

  int size_m;

};

template<class T>
struct LeafFunctor<Scalar<T>, SizeLeaf>
{
  typedef bool Type_t;
  inline static
  bool apply(const Scalar<T> &, const SizeLeaf &)
  {
    // Scalars always conform.

    return true;
  }
};

template<class T, class Allocator>
struct LeafFunctor<vector<T, Allocator>, SizeLeaf>
{
  typedef bool Type_t;
  inline static
  bool apply(const vector<T, Allocator> &v, const SizeLeaf &s)
  {
    return s(v.size());
  }
};

//-----------------------------------------------------------------------------
// EvalLeaf1 is used to evaluate expression with vectors.
// (It's already defined for Scalar values.)
//-----------------------------------------------------------------------------

template<class T, class Allocator>
struct LeafFunctor<vector<T, Allocator>,EvalLeaf1>
{
  typedef T Type_t;
  inline static
  Type_t apply(const vector<T, Allocator>& vec,const EvalLeaf1 &f)
  {
    return vec[f.val1()];
  }
};

//-----------------------------------------------------------------------------
// Loop over vector and evaluate the expression at each location.
//-----------------------------------------------------------------------------

template<class T, class Allocator, class Op, class RHS>
inline void evaluate(vector<T, Allocator> &lhs, const Op &op,
  const Expression<RHS> &rhs)
{
  if (forEach(rhs, SizeLeaf(lhs.size()), AndCombine()))
    {
      // We get here if the vectors on the RHS are the same size as those on
      // the LHS.

      for (int i = 0; i < lhs.size(); ++i)
        {
          // The actual assignment operation is performed here.
          // PETE operator tags all define operator() to perform the operation.
          // (In this case op performs an assignment.) forEach is used
          // to compute the rhs value.  EvalLeaf1 gets the
          // values at each node using random access, and the tag
          // OpCombine tells forEach to use the operator tags in the expression
          // to combine values together.

          op(lhs[i], forEach(rhs, EvalLeaf1(i), OpCombine()));
        }
    }
  else
    {
      cerr << "Error: LHS and RHS don't conform." << endl;
      exit(1);
    }
}

#endif // PETE_EXAMPLES_VECTOR_EVAL_H
```

VectorDefs.in

```
classes
-----
  ARG   = "class T[n]"
  CLASS = "vector<T[n]>"
```

Vector.cpp

```
#include "Eval.h"

int main()
{
  int i;
  const int n = 10;
  vector<int> a, b, c, d;
  vector<double> e(n);

  for (i = 0; i < n; ++i)
  {
    a.push_back(i);
    b.push_back(2*i);
    c.push_back(3*i);
    d.push_back(i);
  }

  assign(b, 2);
  assign(d, a + b * c);
  a += where(d < 30, b, c);

  assign(e, c);
  e += e - 4 / (c + 1);

  for (i = 0;i < n; ++i)
    {
      cout << " a(" << i << ") = " << a[i]
        << " b(" << i << ") = " << b[i]
        << " c(" << i << ") = " << c[i]
        << " d(" << i << ") = " << d[i]
        << " e(" << i << ") = " << e[i]
        << endl;
    }
}
```

[Prev] [Home] [Next]

Copyright © Los Alamos National Laboratory 1999

<PETE>
Portable Expression Template Engine

PETE Tutorial 3
Synthesizing Types

Contents:

Introduction

This tutorial shows how to use PETE to perform non-arithmetic operations on a program as it is being compiled, and in particular how to synthesize new types and values by extending PETE's specialized templates and operator overloadings. The problem domain is the three primary colors---red, green, and blue---which are combined according to three simple rules:

- red and green give blue;
- red and blue give green; and
- green and blue give red.

While these rules are trivial, the techniques shown below can be used to make compilers optimize much more complex expressions on more complicated domains.

The source files for this example are included in the `examples/RGB` directory of the PETE distribution. These files are:

- `RGB.h`: defines the tag classes used in the example to represent colors, and the rules used to combine them.
- `RGBDefs.in`: definitions needed to automatically generate the overloaded operators required to integrate the color classes into PETE. Like the `Vec3Defs.in` file in the first tutorial, this file is processed by the `MakeOperators` tool to create a header file.
- `RGBOperators.h`: the file generated by `MakeOperators` based on the definitions in `RGBDefs.in`. The file `RGB.h` `#includes` `RGBOperators.h`, so that PETE-based programs only need to `#include` `RGB.h`, rather than both `RGB.h` and `RGBOperators.h`.
- `RGB.cpp`: a short program that shows how to use the definitions in the header files described above to create values during compilation.
- `makefile`: rebuilds the example.

On the Surface

PETE was created to make it easy for programmers to extend expression templates. In particular, PETE lets programmers specify a wide range of computations that are to be carried out as the program is being compiled. To do this, the programmer must provide three things: the type(s) of object(s) to be operated on, the functor(s) to be used to extract values from those objects, and the combiner(s) to be used to process those values.

In our example, the values being manipulated are the three primary colors red, green, and blue. Each color is represented by an empty tag class, whose only purpose is to act as a strongly-typed placeholder as the compiler is instantiating templates. The three classes are defined at the top of `RGB.h`:

```
struct Red { };
struct Green { };
struct Blue { };
```

Our example defines a single functor, called `GetColor`. Like `Red`, `Green`, and `Blue`, `GetColor` is an empty class, no instances of which are ever actually created by a running program.

In addition, `RGB.h` defines one more empty tag class, called `ColorCombine`. This tag is used to signal to the compiler that we are combining colors, rather than (for example) adding vectors. The two definitions are:

```
struct GetColor { };
struct ColorCombine { };
```

We have one more empty class to define in order to begin building our computational framework. The class `Operand` serves as a wrapper around color expressions; its purpose is to identify those expressions by giving them a well-known overall type. `Operand` is therefore similar to the generic PETE expression class `Expression`, which is used to distinguish PETE expressions from other types of expressions. In our case, the `Operand` class is used only to wrap up a color expression:

```
template<class ColorTag>
struct Operand
{
  // body is empty
};
```

We must now tell PETE how to store an `Operand` value in the leaf of an expression tree. The required definition is:

```
template<class ColorTag>
struct CreateLeaf<Operand<ColorTag> >
{
  typedef Operand<ColorTag> Leaf_t;
  inline static
  const Leaf_t &make(const Operand<ColorTag> &a) { return a; }
};
```

Note how the formal class name used in the template header, `ColorTag`, appears as an argument to `Operand` in the formal parameter list of the `CreateLeaf` specialization. This ensures that our definition only applies to properly-formed `Operand`s.

Note also the `typedef` inside this specialization of `CreateLeaf`. PETE's template specialization rules require every specialization of `CreateLeaf` to have a `typedef` called `Leaf_t`, which specifies the type of the leaf node. This is also the declared return type of the `static` method `make()`, which constructs a leaf node given an actual object (in this case, a color class instance).

`CreateLeaf` tells PETE how to store values in leaf nodes; specializations of `LeafFunctor` tell PETE how to apply a user-defined functor to those nodes to obtain a value. As before, we template our specialization of `LeafFunctor` on a formal parameter that will be filled in with a color class, but then nest that formal parameter inside `Operand` in order to ensure that the compiler only tries to use this specialization of `LeafFunctor` on the right kinds of expressions.

`LeafFunctor` takes a second template parameter, which is the functor that is being applied to the leaf node. In our case, we have defined only one functor on colors, namely `GetColor`. Our specialization is therefore:

```
template<class Color>
struct LeafFunctor<Operand<Color>, GetColor>
{
  typedef Color Type_t;
};
```

Unlike the `LeafFunctor` specializations in previous tutorials, notice that this version does not have an `apply()` method. The reason is that we're using this functor only for compile-time type calculations. We're never going to call `apply()` so we therefore don't need to go to the trouble of defining it.

Our last task is to define some specializations of `Combine2`, the combiner that PETE uses to operate on values in binary expressions. Six specializations are defined, for each possible ordered combination of different color values. The combiner for `Red` and `Green` is:

```
063  template<class Op>
064  struct Combine2<Red, Green, Op, ColorCombine>
065  {
066    typedef Blue Type_t;
       // not required  inline static
       // not required  Type_t combine(Red, Green, Op, ColorCombine)
       // not required  {
       // not required    return Blue();
       // not required  }
067  };
```

The generic form of `Combine2` takes four template parameters: the classes of its operands, a tag identifying the C++ operator from which the expression was created (such as `OpAdd` or `OpMultiply`), and a user-defined tag, which can be used to force the compiler to calculate a different expression than the one apparently specified. In the case of this example, the first two parameters specify the colors being combined, while the last parameter signals that these values are being combined according to our own rules. This is the only use for `ColorCombine`. The formal template parameter `Op` is not referenced anywhere in this class, so that *any* binary operation on colors, including addition, multiplication, bitwise OR, or left shift, will use the user-defined rules.

Combiners typically have two elements: a type `Type_t` that gives the type of object formed by combining the operands and a `combine` methods that takes the operands and does whatever is necessary to produce a `Type_t` object. Like the `LeafFunctor` specialization above, we don't need the `combine` method here since we're synthesizing types. However, we've shown what this function would look like if it were necessary to define it.

We can now test that our definitions do the right thing by defining three functions to print out the color of an expression involving colors. The function for `Red` takes a constant reference to a `Red` object, and prints a simple string:

```
027  inline void calc(const Red &)
028  {
029    cout << "This expression is red." << endl;
030  }
```

The other two overloadings of this function, which are defined on lines 32-40 for the tag classes `Green` and `Blue`, print out "green" and "blue" instead of "red".

Finally, the templated function `printColor()` takes an expression, evaluates it *during compilation* by forcing the compiler to expand the expression using our `GetColor` and `ColorCombine` tags, and then uses the deduced color of the expression to select a version of `calc()`, which prints out that color's name. The whole definition is:

```
048  template <class Expr>
049  void printColor(const Expression<Expr> &expr)
050  {
051    typedef typename ForEach<Expression<Expr>, GetColor, ColorCombine>::Type_t
052      DeducedColor_t;
053
054    calc(DeducedColor_t());
055  }
```

It is worth looking at this function definition closely. The expansion of `CreateLeaf<>::Leaf_t` extracts and formats the type of the expression according to our color-based evaluation rules. The expansion of the PETE-defined template `ForEach` does most of the work. During its expansion, the functor and combiner tags are passed down the parse tree. They are used to extract types from the leaves of the parse tree. These types are combined at non-leaf nodes to produce new types, which are passed up the parse tree. The result is a type---`Red`, `Green`, or `Blue`---that is labelled by `DeducedColor_t`. This type, in turn, triggers instantiation of an appropriate version of `calc()`.

Under the Hood

Let's take a closer look at exactly what happens when `printColor()` is instantiated with a color expression, as it is at the start of the test program in `RGB.cpp`:

```
  printColor(Operand<Red>() + Operand<Green>());
```

Let's start with the automatically-generated operator overloadings in `RGBOperators.h`, which are created using the `MakerOperators` tool described in the first tutorial. The overloading for `operator+` is:

```
template<class T1,class T2>
inline typename MakeReturn<BinaryNode<OpAdd,
  typename CreateLeaf<Operand<T1> >::Leaf_t,
  typename CreateLeaf<Operand<T2> >::Leaf_t,
  StoreByRefTag> >::Expression_t
operator+(const Operand<T1> & l,const Operand<T2> & r)
{
  typedef BinaryNode<OpAdd,
    typename CreateLeaf<Operand<T1> >::Leaf_t,
    typename CreateLeaf<Operand<T2> >::Leaf_t,
    StoreByRefTag> Tree_t;
  return MakeReturn<Tree_t>::make(Tree_t(
    CreateLeaf<Operand<T1> >::make(l),
    CreateLeaf<Operand<T2> >::make(r)));
}
```

Once again, there is less going on here than first meets the eye. We are trying to perform computations using C++ template notation, a job for which that notation was not designed. The first things to look at are the uses of `CreateLeaf::Leaf_t`. As we saw above, in the case of an `Operand` with a color type argument, `Leaf_t` is just the color type argument wrapped in an `Operand` type; the `CreateLeaf` indirection is provided to give programmers a hook for doing other things if they so desire. This means that we can simplify the code above as:

```
template<class T1,class T2>
inline typename MakeReturn<BinaryNode<OpAdd,
  Operand<T1>,
  Operand<T2>,
  StoreByRefTag> >::Expression_t
operator+(const Operand<T1> & l,const Operand<T2> & r)
{
  typedef BinaryNode<OpAdd,
    Operand<T1>,
    Operand<T2>,
    StoreByRefTag> Tree_t;
  return MakeReturn<Tree_t>::make(Tree_t(
    CreateLeaf<Operand<T1> >::make(l),
    CreateLeaf<Operand<T2> >::make(r)));
}
```

By referring back to the arguments of `printColor()`, we can replace `T1` with `Red`, and `T2` with `Green`:

```
inline typename MakeReturn<BinaryNode<OpAdd,
  Operand<Red>,
  Operand<Green>,
  StoreByRefTag> >::Expression_t
operator+(const Operand<Red> & l,const Operand<Green> & r)
{
  typedef BinaryNode<OpAdd,
    Operand<Red>,
    Operand<Green>,
    StoreByRefTag> Tree_t;
  return MakeReturn<Tree_t>::make(Tree_t(
    CreateLeaf<Operand<Red> >::make(l),
    CreateLeaf<Operand<Green> >::make(r)));
}
```

Looking back at `CreateLeaf` once more, we see that its `make()` method simply returns its argument. (In the case of STL containers, `make()` could return an iterator over its argument.) Our operator thus becomes simpler still:

```
inline typename MakeReturn<BinaryNode<OpAdd,
  Operand<Red>,
  Operand<Green>,
  StoreByRefTag> >::Expression_t
operator+(const Operand<Red> & l,const Operand<Green> & r)
{
  typedef BinaryNode<OpAdd,
    Operand<Red>,
    Operand<Green>,
    StoreByRefTag> Tree_t;
  return MakeReturn<Tree_t>::make(Tree_t(l, r));
}
```

To simplify this further, we must turn to the definition of `MakeReturn` in PETE's `CreateLeaf.h`:

```
template<class T>
struct MakeReturn
{
  typedef Expression<T> Expression_t;
  inline static
  Expression_t make(const T &a) { return Expression_t(a); }
};
```

The default expansion of `MakeReturn<T>::Expression_t` is simply `Expression<T>`, and `MakeReturn`'s `make()` method just returns its argument, appropriately typed. This may seem unnecessary, but as the PETE header files themselves explain:

> `MakeReturn` is used to wrap expression objects (`UnaryNode`, `BinaryNode` etc.) inside an `Expression` object. Usually this indirection is unnecessary, but the indirection allows users to specify their own approach to storing trees. By specializing `MakeReturn<UnaryNode>`, `MakeReturn<BinaryNode>`, etc. you could cause the expression trees to be stored in another format. For example, POOMA stores expressions inside `Array`s, so the result of `Array+Array` is another `Array`.

We can now expand our operator one more level:

```
operator+(const Operand<Red> & l,const Operand<Green> & r)
{
  typedef BinaryNode<OpAdd,
    Operand<Red>,
    Operand<Green>,
    StoreByRefTag> Tree_t;
  return Expression<Tree_t>(Tree_t(l, r));
}
```

Note that we are no longer bothering to show the return type of the function, since it is the same as the type of the `return` statement inside the function body.

With this in hand, let's return to `printColor()`:

```
template <class Expr>
void printColor(const Expression<Expr> &expr)
{
  typedef typename ForEach<Expression<Expr>, GetColor, ColorCombine>::Type_t
    DeducedColor_t;

  calc(DeducedColor_t());
}
```

The formal parameter `expr` is an instance of `Expression<BinaryNode<OpAdd,Operand<Red>,Operand<Green>,StoreByRefTag> >`, with an `Operand<Red>` and a `Operand<Green>` as its left and right members. This type is passed to PETE's `ForEach` class. We use this class rather than the `forEach` function because we are synthesizing types. Referring to the PETE header file `ForEach.h`, we see that the most specific matching definition is:

```
146  template<class T, class FTag, class CTag>
147  struct ForEach<Expression<T>, FTag, CTag>
148  {
149    typedef typename ForEach<T, FTag, CTag>::Type_t Type_t;
150    inline static
151    Type_t apply(const Expression<T> &expr, const FTag &f,
152                 const CTag &c)
153    {
154      return ForEach<T, FTag, CTag>::apply(expr.expression(), f, c);
155    }
156  };
```

As we've mentioned, we are synthesizing types in this example so the `apply` function (lines 150-155) will never actually be called. Therefore, in subsequent definitions, we'll omit this for brevity. The important thing to note is that the `typedef` of `Type_t` is generated by extracting the type wrapped by the `Expression` object and using that in another `ForEach`. Recall that this wrapped type in our example is `BinaryNode<OpAdd,Operand<Red>,Operand<Green>,StoreByRefTag>`. This means that the relevant `ForEach` definition is

```
107  template<class Op, class A, class B, class ST, class FTag, class CTag>
108  struct ForEach<BinaryNode<Op, A, B, ST>, FTag, CTag >
109  {
110    typedef typename ForEach<A, FTag, CTag>::Type_t TypeA_t;
111    typedef typename ForEach<B, FTag, CTag>::Type_t TypeB_t;
112    typedef typename Combine2<TypeA_t, TypeB_t, Op, CTag>::Type_t Type_t;
122  };
```

Now is a good time to perform some type substitutions. The result is

```
107  template<>
108  struct ForEach<BinaryNode<OpAdd,Operand<Red>,Operand<Green>,StoreByRefTag>,
OpAdd, GetColor, ColorCombine>
109  {
110    typedef ForEach<Operand<Red>, GetColor, ColorCombine>::Type_t TypeA_t;
111    typedef ForEach<Operand<Green>, GetColor, ColorCombine>::Type_t TypeB_t;
112    typedef Combine2<TypeA_t, TypeB_t, OpAdd, ColorCombine>::Type_t Type_t;
122  };
```

To proceed, we need to resolve the `ForEach` types in lines 110 and 111. Looking through `ForEach.h`, we see that the only partial specialization that matches is the most general version of `ForEach`:

```
074  template<class Expr, class FTag, class CTag>
075  struct ForEach
076  {
077    typedef typename LeafFunctor<Expr, FTag>::Type_t Type_t;
083  };
```

This version of `ForEach` is meant to be used for leaves. It simply passes the task to the `LeafFunctor` class. Substituting this above gives:

```
107  template<>
108  struct ForEach<BinaryNode<OpAdd,Operand<Red>,Operand<Green>,StoreByRefTag>,
OpAdd, GetColor, ColorCombine>
109  {
110    typedef LeafFunctor<Operand<Red>, GetColor>::Type_t TypeA_t;
111    typedef LeafFunctor<Operand<Green>, GetColor>::Type_t TypeB_t;
112    typedef Combine2<TypeA_t, TypeB_t, OpAdd, ColorCombine>::Type_t Type_t;
122  };
```

This is starting to look promising: we now have some invocations of `Combine2`, the combiner that was overridden in `RGB.h`, and some uses of `LeafFunctor`, which was also overridden. In fact, as we saw earlier, when `LeafFunctor` has an `Operand` as its first type argument, and the `GetColor` functor tag as its second argument, its `Type_t` definition is just its color argument. We can therefore simplify the definition of `ForEach` on binary nodes to be:

```
template<>
struct ForEach<BinaryNode<OpAdd,Operand<Red>,Operand<Green>,StoreByRefTag>, OpAdd, GetColor, ColorCombine>
{
  typedef Combine2<Red, Green, OpAdd, ColorCombine>::Type_t Type_t;
};
```

The compiler can now match the specialized definition of `Combine2` shown earlier against this code. Thus, the return type of the expansion of `ForEach` inside of `printColor` is `Blue`. This, in turn, is used to select the `calc()` function, which simply prints out the word "blue".

This may seem like a lot of work simply to print out a different word. However, it illustrates an extremely powerful capability of PETE: selecting custom algorithms at compile time based on a synthesized type. All of the type computations outlined above are performed at compile time. Also, the various `calc` functions are inlined. This means that the compiler will generate a custom `printColor` function for our expression that is equivalent to

```
template <>
void printColor(const Expression<BinaryNode<OpAdd,Operand<Red>,Operand<Green>,StoreByRefTag> > &expr)
{
  cout << "This expression is blue." << endl;
}
```

This is an example of *compile-time polymorphism*. We've used the C++ compiler to generate special code based on the types we pass into a function rather than making a run-time choice of a function to call. This can lead to the generation of extremely efficient code.

Summary

This tutorial has shown how to extend PETE to synthesize type information during compilation by performing symbolic manipulations on parse trees. The user-level definitions required are more complex than those needed to use PETE simply to optimize expression evaluation, but tracing through their operation shows how PETE exploit's the C++ compiler's pattern matching and type expansion facilities to do what it does.

Source Files

RGB.h

```
#ifndef PETE_EXAMPLES_RGB_RGB_H
#define PETE_EXAMPLES_RGB_RGB_H

//-----------------------------------------------------------------------------
// Include files
//-----------------------------------------------------------------------------

#include <iostream.h>

#include "PETE/PETE.h"

//-----------------------------------------------------------------------------
// Tag classes representing colors. Also, a functor for getting a color from
// a leaf and a combiner for combining colors.
//-----------------------------------------------------------------------------

struct Red { };
struct Green { };
struct Blue { };
struct GetColor { };
struct ColorCombine { };

//-----------------------------------------------------------------------------
// A few overloaded functions that print out color names given a type.
//-----------------------------------------------------------------------------

inline void calc(const Red &)
{
  cout << "This expression is red." << endl;
}

inline void calc(const Blue &)
{
  cout << "This expression is blue." << endl;
}

inline void calc(const Green &)
{
  cout << "This expression is green." << endl;
}

//-----------------------------------------------------------------------------
// A function that deduces a color at compile time and calls a special
// function based on the value.
//
//-----------------------------------------------------------------------------

template <class Expr>
void printColor(const Expression<Expr> &expr)
{
  typedef typename ForEach<Expression<Expr>, GetColor, ColorCombine>::Type_t
    DeducedColor_t;

  calc(DeducedColor_t());
}

//-----------------------------------------------------------------------------
// A set of combiners that produce new colors according to some arbitrary
// rules: red & green give blue, red & blue give green, blue and green give
// red.
//-----------------------------------------------------------------------------

template<class Op>
struct Combine2<Red, Green, Op, ColorCombine>
{
  typedef Blue Type_t;
};

template<class Op>
struct Combine2<Red, Blue, Op, ColorCombine>
{
  typedef Green Type_t;
};

template<class Op>
struct Combine2<Green, Blue, Op, ColorCombine>
{
  typedef Red Type_t;
};

template<class Op>
struct Combine2<Green, Red, Op, ColorCombine>
{
  typedef Blue Type_t;
};

template<class Op>
struct Combine2<Blue, Green, Op, ColorCombine>
{
  typedef Red Type_t;
};

template<class Op>
struct Combine2<Blue, Red, Op, ColorCombine>
{
  typedef Green Type_t;
};

template<class C1, class C2, class Op>
struct Combine2<C1, C2, Op, ColorCombine>
{
  typedef C1 Type_t;
};

//-----------------------------------------------------------------------------
// A class that has a single template parameter that specifies a color.
//-----------------------------------------------------------------------------

template<class ColorTag>
struct Operand
{
};

//-----------------------------------------------------------------------------
// We need to specialize CreateLeaf<T> for Operand, so that operators
// know what to stick in the leaves of the expression tree.
//-----------------------------------------------------------------------------

template<class ColorTag>
struct CreateLeaf<Operand<ColorTag> >
{
  typedef Operand<ColorTag> Leaf_t;
  inline static
  const Leaf_t &make(const Operand<ColorTag> &a) { return a; }
};

//-----------------------------------------------------------------------------
// Specialization of LeafFunctor class for applying the getting the "color"
// of an operand.
//-----------------------------------------------------------------------------

template<class Color>
struct LeafFunctor<Operand<Color>, GetColor>
{
  typedef Color Type_t;
};

#include "RGBOperators.h"

```

```
#endif // PETE_EXAMPLES_RGB_RGB_H
```

RGBDefs.in

```
classes
-----
  ARG   = "class T[n]"
  CLASS = "Operand<T[n]>"
```

RGB.cpp

```
#include "RGB.h"

int main()
{
  printColor(Operand<Red>() + Operand<Green>());
  printColor(Operand<Red>() + Operand<Green>() + Operand<Blue>());
  printColor(Operand<Red>() + (Operand<Green>() + Operand<Blue>()));
}
```

[Prev] [Home] [Next]

Copyright © Los Alamos National Laboratory 1999

PETE Tutorials
The Standard Template Library

The best-known use of templates to date has been the Standard Template Library, or STL. The STL uses templates to separate containers (such as vectors and lists) from algorithms (such as finding, merging, and sorting). The two are connected through the use of *iterators*, which are classes that know how to read or write particular containers, without exposing the actual type of those containers.

For example, consider the following code fragment, which finds the first occurrence of a particular value in a vector of floating-point numbers:

```
void findValue(vector<double> & values, double target)
{
    vector<double>::iterator loc =
        find(values.begin(), values.end(), target);
    assert(*loc == target);
}
```

The STL class `vector` declares another class called `iterator`, whose job it is to traverse a `vector`. The two methods `begin()` and `end()` return instances of `vector::iterator` marking the beginning and end of the vector. STL's `find()` function iterates from the first of its arguments to the second, looking for a value that matches the one specified. Finally, dereferencing (`operator*`) is overloaded for `vector::iterator`, so that `*loc` returns the value at the location specified by `loc`.

If we decide later to store our values in a list instead of in a vector, only the declaration of the container type needs to change, since `list` defines a nested iterator class, and `begin()` and `end()` methods, in exactly the same way as `vector`:

```
void findValue(list<double> & values, double target)
{
    list<double>::iterator loc =
        find(values.begin(), values.end(), target);
    assert(*loc == target);
}
```

If we go one step further, and use a `typedef` to label our container type, then nothing in `findValue()` needs to change at all:

```
    typedef vector<double> Storage;
    // typedef list<double> Storage;

    void findValue(Storage & values, double target)
    {
        Storage::iterator loc =
            find(values.begin(), values.end(), target);
        assert(*loc == target);
    }
```

The performance of this code will change as the storage mechanism changes, but that's the point: STL-based code can often be tuned using only minor, non-algorithmic changes.

[Prev] [Home] [Next]

Copyright © Los Alamos National Laboratory 1999

<PETE>
Portable Expression Template Engine

MakeOperators

Name

MakeOperators - make the operator functions for a given set of classes that return expression trees, so that PETE can be used with those classes. Also can construct global assignment operators and operator tag structs.

Synopsis

```
MakeOperators [--help] [--pete-help] [--classes classfile]
[--operators opfile [--pete-ops] ] [--guard INCLUDE_GUARD]
[--scalars] [--extra-classes] [--no-expression] [--assign-ops]
[--op-tags] [--no-shift-guard] [--o outputfile]
```

Description

In order to use PETE with a given set of container classes, operators such as `+, -, *, /` etc. must be defined to return expression template parse trees when given those classes. Operators must be defined for combining the container classes, `B + C`, for combining the container classes with scalars, `2 * C`, and for combining parse trees with other objects, `B + (C + D) + 2`. To generate the PETE built-in operators requires over 200 different templated operator functions to interface PETE with a single container class such as the STL vector.

Command line options are:

--help
--pete-help

Print a simple summary of the command options.

--classes classfile

Input the class definitions from the file "classfile". Omitting this option causes no operator functions to be produced, which can be useful if you only want to produce operator tags.

To understand the format of the input file, consider the STL vector. **MakeOperators** will output definitions for `operator+()` between vectors and vectors and between vectors and scalars:

```
template<class T1,class Allocator1,class T2,class Allocator2>
(parse tree return type)
operator+(const vector<T1,Allocator1> &v1,const vector<T2,Allocator2> &v2)
{
  (construct parse tree)
}
template<class T1,class T2,class Allocator2>
(parse tree return type)
operator+(const T1 &v1,const vector<T2,Allocator2> &v2)
{
  (construct parse tree)
}
template<class T1,class Allocator1,class T2>
(parse tree return type)
operator+(const vector<T1,Allocator1>& v1,const T2 &v2)
{
  (construct parse tree)
}
```

In order to construct the operator, the tool needs to know the template arguments `"class T,class Allocator"` and the templated form of the class `"vector<T,Allocator>"`. For the STL vector example the class definitions file would contain the four lines:

```
classes
-----
  ARG   = "class T[n],class Allocator[n]"
  CLASS = "vector<T[n],Allocator[n]>"
```

The string `[n]` needs to be attached to each template argument and represents a number that allows **MakeOperators** to uniquely identify each argument in binary and trinary operators. For classes with no template arguments, use `ARG = ""`. In general, the class definition definition file can look like:

```
classes
-----
  ARG   = (class 1 args)
  CLASS = (class 1 definition)
-----
  ARG   = (class 2 args)
  CLASS = (class 2 definition)
...

extraClasses
-----
  ARG   = (extra class 1 args)
  CLASS = (extra class 1 definition)
...
scalars
-----
  ARG   = (scalar 1 args)
  CLASS = (scalar 1 definition)
...
```

When multiple classes are listed, operators are produced for all combinations of those classes with each other, with scalars and with expression objects.

The second optional list starting with the word `extraClasses` is used if you want to extend a previously created file. For example, if you produced a file defining all the operators for `vector<T>` and wanted to extend your implementation to operations between vectors and `list<T>`, then you would list vector as a class and list under extraClasses and specify the option `--extra-classes`. The resulting file would define operations between lists and lists, and between lists and vectors, but omit those between vectors and vectors, so that you could include both the new file and your previously generated file. Typically, it would be better to simply create a new file with all the operators, so extraClasses should rarely be used.

The final part of this list that begins with the word `scalars` will only rarely need to be used. By the rules of partial specialization, if any class does not appear in the classes list, it will be treated as a scalar. Suppose you were to define a tensor class `Tensor<T>`, then `Tensor<T>() + vector<Tensor<T> >()` would invoke the right function: `T1 + vector<T2>` (which means treat the tensor on the left as a scalar and add it to each of the tensors in the vector of tensors). A problem arises if you also define scalar operations with tensors of the form `Tensor<T1> + T2` to represent adding a scalar to each of the tensor components. In this case `Tensor<T>() + vector<Tensor<T> >()` is ambiguous as it matches the function for adding scalars to vectors and the function for addint tensors to scalars. To resolve this case, we must explicitly define `Tensor<T> + vector<Tensor<T> >`, which will happen if we add `Tensor<T>` to the list of scalars. (So the list of scalars only needs to contain classes that act like scalars but that also define operations between themselves and classes of arbitrary type.)

--o outputfile

Send MakeOperators output to outputfile; otherwise write to stdout.

--operators opfile

Include the operator descriptions from the file "opfile". Typically this option should be omitted, in which case the set of 45 PETE built-in operators are used. See the file `src/Tools/PeteOps.in` in the PETE distribution to see operator descriptors for all the PETE built-in operators. The general format of an operator descriptor file is:

```
type1
-----
  TAG      = "tag"
  FUNCTION = "function"
  EXPR     = "expression"
-----
  TAG      = "tag"
  FUNCTION = "function"
  EXPR     = "expression"
...

type2
-----
  TAG      = "tag"
  FUNCTION = "function"
  EXPR     = "expression"
...
```

The string `"tag"` is the name of a tag class that is used in expression template nodes to differentiate between the different operators. For example, `"OpAdd"` is used for binary `operator+()`, `"OpSubtract"` is used for binary `operator-()`, and so on. The string `"function"` is the name of the operator funtion, `"operator+"` for example. The string `"expression"` contains a description of how to evaluate the operator on specific elements. The string should use the names a, b, and c to represent the arguments to the function. For example the definition of binary `operator+()` sets `EXPR`

= "(a + b)".

The headings `type1`, `type2`, etc. are operator types. Currently the following operator types are supported:

- **unaryOps** - simple unary operators whose return type is the same as their input type
- **unaryBoolOps** - unary operators that return a bool
- **unaryCastOps** - actually binary operators where the first argument is used to set the return type (like peteCast).
- **unarySpecialOp** - unary operators that use the type computation system to compute their return type. For example `real(Complex<T>)` returns `T`.
- **binaryOps** - simple binary operators that compute their return type by promotion.
- **binaryBoolOps** - binary operators that return a bool (like <).
- **binaryLeftOps** - binary operators that return the type of the left argument.
- **binarySpecialOps** - unary operators that use the type computation system to compute their return type.
- **binaryAssignOps** - assignment operators like `+=`. If operator functions are produced for these operators then they call `evaluate()` instead of returning an expression tree.
- **binaryAssignBoolOps** - assignment operators that always return a bool, like `andAssign` which emulates the mythical &&=.
- **assignOp** - `operator=()`.
- **trinaryOps** - trinary operators like `where()` which emulates `?:`.

--pete-ops

Using the `--operators` causes the tool to use operator descriptors from a file, but not to use any of the pre-defined PETE operators. If you wish produce operators for BOTH operators read from a file AND the pre-defined PETE operators, the use `--pete-ops` as well as the `--operators` option. For example, the first two commands in the POOMA example below could be simplified to produce one file:

```
MakeOperators --classes PoomaClass.in \
              --operators PoomaOps.in --pete-ops \
              --guard POOMA_ARRAY_ARRAYOPERATORS_H \
              --no-expression --o ArrayOperators.h
```

--guard INCLUDE_GUARD

The code output by **MakeOperators** includes ifdefs to guard against multiple inclusion of the form:

```
#ifndef INCLUDE_GUARD
#define INCLUDE_GUARD
 (code goes here)
#endif // INCLUDE_GUARD
```

If this option is omitted then `INCLUDE_GUARD` will default to either `GENERATED_OPERATORS_H` if the `--classes` option is present, or `OPERATOR_TAGS_H` otherwise. If you wish to omit the include guards from the output file, then use the option `--guard ""`.

--scalars

When this option is present, only operations between classes and scalars are produced. This option is useful in the situation mentioned in the description of `--classes`, where operators must be defined between containers and user defined scalars in order to resolve ambiguities. In the example of a user defined tensor class, the user would probably only define a small set of operations with general scalars, like +, -, *, and /. To produce smaller operator files, you could produce all the operators without tensors and then produce the operators between containers and tensors just for the smaller set of operators. See the example section for an example of this case.

--extra-classes

When this option is present, only operations involving extraClasses are produced. This option is useful if you want to create an operator file that extends a previously created operator file. See the `--classes` option for a description of extraClasses.

--no-expression

MakeOperators needs to define operations between parse tree objects and containers and scalars. In the expression `A + ( B + C )`, the subexpression `( B + C )` returns a parse tree object which must then be combined with the container `A`. Some users of PETE, like POOMA, wrap the result of operators inside their own container class, so there is no need to define such operators. (The sum of two POOMA arrays is an array containing an expression.) This flag turns off generation of operations with parse tree objects.

--assign-ops

Generate global assignment operators that call the function `evaluate()`.

--op-tags

Produce definitions of the operator tag classes. PETE already contains definitions of all the PETE built-in operators, so this flag only needs to be used for user defined operators.

--no-shift-guard

It is typical to define the operator `ostream << container`, which can get confused with the operator `T << container` under some circumstances. To avoid this problem, PETE only defines the shift operators between scalars and containers if the macro PETE_ALLOW_SCALAR_SHIFT is defined. If `--no-shift-guard` is selected, then the ifdefs that implement this guard are eliminated and shift operators between scalars and containers are always defined.

Examples

Here we build operators to use STL vectors with PETE. The flag `--assign-ops` is present because we cannot define the assignment member functions for STL vectors.

```
MakeOperators --classes vectorDefs.in --assign-ops > VectorOperators.h
```

For POOMA, we create the built-in PETE operators, some special POOMA operators like `real()`, and finally operators between POOMA arrays and the POOMA Vector and Tensor scalars to disambiguate them. The flag `--no-expression` is used because POOMA wraps expressions inside POOMA arrays. The flag `--assign-ops` not used because POOMA arrays define assignment member functions. In the second command, `--op-tags` is used because the POOMA operator tag classes need to be defined. In the third command, `--scalars` is used because the first command has already defined operations between POOMA arrays for the operators in `PoomaVectorOps.in` (which is a subset of the PETE operators). In the fourth command, `--o VectorOperators.h` sends output to that file rather than stdout.

```
MakeOperators --classes PoomaClass.in --guard POOMA_ARRAY_ARRAYOPERATORS_H \
              --no-expression > ArrayOperators.h

MakeOperators --classes PoomaClass.in --operators PoomaOps.in \
              --guard POOMA_POOMA_POOMAOPERATORS_H --no-expression \
              --op-tags > PoomaOperators.h

MakeOperators --classes PoomaVectorClass.in --operators PoomaVectorOps.in \
              --guard POOMA_POOMA_VECTOROPERATORS_H --no-expression --scalars \
              > VectorOperators.h

MakeOperators --classes PoomaVectorClass.in --operators PoomaVectorOps.in \
              --guard POOMA_POOMA_VECTOROPERATORS_H --no-expression --scalars \
              --o VectorOperators.h
```

[Home]

Copyright © Los Alamos National Laboratory 1999

PETE Tutorials
Legal Notice

This software and ancillary information (herein called "SOFTWARE") called PETE (Portable Expression Template Engine) is made available under the terms described here. The SOFTWARE has been approved for release with associated LA-CC Number *LA-CC-99-5*.

Unless otherwise indicated, this SOFTWARE has been authored by an employee or employees of the University of California, operator of the Los Alamos National Laboratory under Contract No. W-7405-ENG-36 with the U.S. Department of Energy. The U.S. Government has rights to use, reproduce, and distribute this SOFTWARE, and to allow others to do so. The public may copy and use this SOFTWARE, FOR NONCOMMERCIAL USE ONLY, without charge, provided that this Notice and any statement of authorship are reproduced on all copies. Neither the Government nor the University makes any warranty, express or implied, or assumes any liability or responsibility for the use of this SOFTWARE.

If SOFTWARE is modified to produce derivative works, such modified SOFTWARE should be clearly marked, so as not to confuse it with the version available from LANL.

For more information about PETE, send e-mail to pete@acl.lanl.gov, or visit the PETE web page at http://www.acl.lanl.gov/pete.

[Prev] [Home]

Copyright © Los Alamos National Laboratory 1999