Type Erasure of Expression Templates

Nevin ":-)" Liber
nevin@eviloverlord.com

Expression Templates

- Capture parse tree of a C++ expression
- Typically used for delayed evaluation
  - boost::lambda
  - boost::spirit
  - boost::proto
- How can we store an expression template?
  - Type erasure

```
struct Plus {};
struct Times {};

template<typename Op = void, typename L = void, typename R = void>
struct Calc;

template<>
struct Calc<int>
{
    Calc(int i) : i(i) {}

    int operator()() const { return i; }

    friend std::ostream& operator<<(std::ostream& os, Calc const& c)
    { return os << c.i; }

    int i;
};

template<typename L, typename R>
struct Calc<Plus, L, R>
{
    Calc(L const& lhs, R const& rhs) : lhs(lhs), rhs(rhs) {}

    int operator()() const { return lhs() + rhs(); }

    friend std::ostream& operator<<(std::ostream& os, Calc const& c)
    { return os << '(' << c.lhs << '+' << c.rhs << ')'; }

    L lhs;
    R rhs;
};

template<typename L, typename R>
struct Calc<Times, L, R>
{
    Calc(L const& lhs, R const& rhs) : lhs(lhs), rhs(rhs) {}

    int operator()() const { return lhs() * rhs(); }

    friend std::ostream& operator<<(std::ostream& os, Calc const& c)
    { return os << '(' << c.lhs << '*' << c.rhs << ')'; }

    L lhs;
    R rhs;
};

template<>
struct Calc<>
{
private:
    struct Concept
    {
        virtual ~Concept() {}
        virtual Concept* clone() const = 0;
        virtual int operator()() const = 0;
        virtual void inserter(std::ostream&) const = 0;
    };

    template<typename T>
    struct Model : Concept
    {
        Model(T const& data) : data(data) {}
        virtual Model* clone() const { return new Model(data); }
        virtual int operator()() const { return data(); }
        virtual void inserter(std::ostream& os) const { os << data; }

        T data;
    };

    boost::scoped_ptr<Concept> object;

public:
    template<typename Op, typename L, typename R>
    Calc(Calc<Op, L, R> const& data) : object(new Model< Calc<Op, L, R> >(data)) {}
    Calc(int i) : object(new Model< Calc<int> >(i)) {}
    Calc(Calc const& that) : object(that.object->clone()) {}

    friend void swap(Calc& lhs, Calc& rhs) { boost::swap(lhs.object, rhs.object); }
    Calc& operator=(Calc rhs) { swap(*this, rhs); return *this; }

    int operator()() const { return (*object)(); }

    friend std::ostream& operator<<(std::ostream& os, Calc const& c)
    { c.object->inserter(os); return os; }
};

template<typename LOp, typename LL, typename LR, typename ROp, typename RL, typename RR>
Calc< Plus, Calc<LOp, LL, LR>, Calc<ROp, RL, RR> >
operator+(Calc<LOp, LL, LR> const& lhs, Calc<ROp, RL, RR> const& rhs)
{ return Calc< Plus, Calc<LOp, LL, LR>, Calc<ROp, RL, RR> >(lhs, rhs); }

template<typename LOp, typename LL, typename LR>
Calc< Plus, Calc<LOp, LL, LR>, Calc<int> >
operator+(Calc<LOp, LL, LR> const& lhs, int i)
{ return Calc< Plus, Calc<LOp, LL, LR>, Calc<int> >(lhs, i); }

template<typename ROp, typename RL, typename RR>
Calc< Plus, Calc<int>, Calc<ROp, RL, RR> >
operator+(int i, Calc<ROp, RL, RR> const& rhs )
{ return Calc< Plus, Calc<int>, Calc<ROp, RL, RR> >(i, rhs); }

template<typename LOp, typename LL, typename LR, typename ROp, typename RL, typename RR>
Calc< Times, Calc<LOp, LL, LR>, Calc<ROp, RL, RR> >
operator*(Calc<LOp, LL, LR> const& lhs, Calc<ROp, RL, RR> const& rhs)
{ return Calc< Times, Calc<LOp, LL, LR>, Calc<ROp, RL, RR> >(lhs, rhs); }

template<typename LOp, typename LL, typename LR>
Calc< Times, Calc<LOp, LL, LR>, Calc<int> >
operator*(Calc<LOp, LL, LR> const& lhs, int i)
{ return Calc< Times, Calc<LOp, LL, LR>, Calc<int> >(lhs, i); }

template<typename ROp, typename RL, typename RR>
Calc< Times, Calc<int>, Calc<ROp, RL, RR> >
operator+(int i, Calc<ROp, RL, RR> const& rhs )
{ return Calc< Times, Calc<int>, Calc<ROp, RL, RR> >(i, rhs); }
```

```
struct Plus {};
struct Times {};

template<typename Op = void, typename L = void, typename R = void>
struct Calc;

template<>
struct Calc<int>
{
    Calc(int i) : i(i) {}

    int operator()() const { return i; }

    friend std::ostream& operator<<(std::ostream& os, Calc const& c)
    { return os << c.i; }

    int i;
};

template<typename L, typename R>
struct Calc<Plus, L, R>
{
    Calc(L const& lhs, R const& rhs) : lhs(lhs), rhs(rhs) {}

    int operator()() const { return lhs() + rhs(); }

    friend std::ostream& operator<<(std::ostream& os, Calc const& c)
    { return os << '(' << c.lhs << '+' << c.rhs << ')'; }

    L lhs;
    R rhs;
};

template<typename L, typename R>
struct Calc<Times, L, R>
{
    Calc(L const& lhs, R const& rhs) : lhs(lhs), rhs(rhs) {}

    int operator()() const { return lhs() * rhs(); }

    friend std::ostream& operator<<(std::ostream& os, Calc const& c)
    { return os << '(' << c.lhs << '*' << c.rhs << ')'; }

    L lhs;
    R rhs;
};

template<>
struct Calc<>
{
private:
    struct Concept
    {
        virtual ~Concept() {}
        virtual Concept* clone() const = 0;
        virtual int operator()() const = 0;
        virtual void inserter(std::ostream&) const = 0;
    };
```

```
struct Plus {};
struct Times {};

template<typename Op = void, typename L = void, typename R = void>
struct Calc;
```

```
struct Plus {};
struct Times {};

template<typename Op = void, typename L = void, typename R = void>
struct Calc;
```

- Tags indicating which binary operator is being captured
  - Passed in as template parameter Op

```
struct Plus {};
struct Times {};

template<typename Op = void, typename L = void, typename R = void>
struct Calc;
```

- The (templated) type of our expression template
  - Op is operator being captured
  - L is type of left hand side expression template of a binary operator
  - R is type of right hand side
  - Implementations via specializations

```
template<>
struct Calc<int>
{
    Calc(int i) : i(i) {}

    int operator()() const { return i; }

    friend std::ostream& operator<<(std::ostream& os, Calc const& c)
    { return os << c.i; }

    int i;
};
```

- Calc<int> holds a number

```
template<>
struct Calc<int>
{
    Calc(int i) : i(i) {}

    int operator()() const { return i; }

    friend std::ostream& operator<<(std::ostream& os, Calc const& c)
    { return os << c.i; }

    int i;
};
```

- operator() is eval
- operator<< prints the expression

```
template<typename L, typename R>
struct Calc<Plus, L, R>
{
    Calc(L const& lhs, R const& rhs) : lhs(lhs), rhs(rhs) {}

    int operator()() const { return lhs() + rhs(); }

    friend std::ostream& operator<<(std::ostream& os, Calc const& c)
    { return os << '(' << c.lhs << '+' << c.rhs << ')'; }

    L lhs;
    R rhs;
};
```

- Calc<Plus> holds the binary operator+ expression

```
template<typename L, typename R>
struct Calc<Plus, L, R>
{
    Calc(L const& lhs, R const& rhs) : lhs(lhs), rhs(rhs) {}

    int operator()() const { return lhs() + rhs(); }

    friend std::ostream& operator<<(std::ostream& os, Calc const& c)
    { return os << '(' << c.lhs << '+' << c.rhs << ')'; }

    L lhs;
    R rhs;
};
```

- Sums the left hand side expression with the right hand side expression

```
template<typename L, typename R>
struct Calc<Plus, L, R>
{
    Calc(L const& lhs, R const& rhs) : lhs(lhs), rhs(rhs) {}

    int operator()() const { return lhs() + rhs(); }

    friend std::ostream& operator<<(std::ostream& os, Calc const& c)
    { return os << '(' << c.lhs << '+' << c.rhs << ')'; }

    L lhs;
    R rhs;
};
```

- Outputs a textual representation of the left hand side expression '+' the right hand side expression

```
template<typename L, typename R>
struct Calc<Plus, L, R>
{
    Calc(L const& lhs, R const& rhs) : lhs(lhs), rhs(rhs) {}

    int operator()() const { return lhs() + rhs(); }

    friend std::ostream& operator<<(std::ostream& os, Calc const& c)
    { return os << '(' << c.lhs << '+' << c.rhs << ')'; }

    L lhs;
    R rhs;
};
```

- Stores an expression template of the left and side expression and the right hand side expression

```
template<typename L, typename R>
struct Calc<Times, L, R>
{
    Calc(L const& lhs, R const& rhs) : lhs(lhs), rhs(rhs) {}

    int operator()() const { return lhs() * rhs(); }

    friend std::ostream& operator<<(std::ostream& os, Calc const& c)
    { return os << '(' << c.lhs << '*' << c.rhs << ')'; }

    L lhs;
    R rhs;
};
```

- Calc<Times> is the expression template for storing a multiplication expression

```
template<>
struct Calc<>
{
private:
    struct Concept
    {
        virtual ~Concept() {}
        virtual Concept* clone() const = 0;
        virtual int operator()() const = 0;
        virtual void inserter(std::ostream&) const = 0;
    };

    template<typename T>
    struct Model : Concept
    {
        Model(T const& data) : data(data) {}
        virtual Model* clone() const { return new Model(data); }
        virtual int operator()() const { return data(); }
        virtual void inserter(std::ostream& os) const { os << data; }

        T data;
    };

    boost::scoped_ptr<Concept> object;
```

- Calc<> stores any Calc<...> template

```
template<>
struct Calc<>
{
private:
    struct Concept
    {
        virtual ~Concept() {}
        virtual Concept* clone() const = 0;
        virtual int operator()() const = 0;
        virtual void inserter(std::ostream&) const = 0;
    };

    template<typename T>
    struct Model : Concept
    {
        Model(T const& data) : data(data) {}
        virtual Model* clone() const { return new Model(data); }
        virtual int operator()() const { return data(); }
        virtual void inserter(std::ostream& os) const { os << data; }

        T data;
    };

    boost::scoped_ptr<Concept> object;
```

- Forwarding functions

```
template<>
struct Calc<>
    //...
public:
    template<typename Op, typename L, typename R>
    Calc(Calc<Op, L, R> const& data) : object(new Model< Calc<Op, L, R> >(data)) {}
    Calc(int i) : object(new Model< Calc<int> >(i)) {}
    Calc(Calc const& that) : object(that.object->clone()) {}

    friend void swap(Calc& lhs, Calc& rhs) { boost::swap(lhs.object, rhs.object); }
    Calc& operator=(Calc rhs) { swap(*this, rhs); return *this; }

    int operator()() const { return (*object)(); }

    friend std::ostream& operator<<(std::ostream& os, Calc const& c)
    { c.object->inserter(os); return os; }
};
```

- Gatekeepers
- Only store Calc<...>
  - Directly store numbers

```
template<>
struct Calc<>
    //...
public:
    template<typename Op, typename L, typename R>
    Calc(Calc<Op, L, R> const& data) : object(new Model< Calc<Op, L, R> >(data)) {}
    Calc(int i) : object(new Model< Calc<int> >(i)) {}
    Calc(Calc const& that) : object(that.object->clone()) {}

    friend void swap(Calc& lhs, Calc& rhs) { boost::swap(lhs.object, rhs.object); }
    Calc& operator=(Calc rhs) { swap(*this, rhs); return *this; }

    int operator()() const { return (*object)(); }

    friend std::ostream& operator<<(std::ostream& os, Calc const& c)
    { c.object->inserter(os); return os; }
};
```

- Forwarding functions
- Calc<> is an expression template
  - Combined with other expressions

```
template<typename LOp, typename LL, typename LR, typename ROp, typename RL, typename RR>
Calc< Plus, Calc<LOp, LL, LR>, Calc<ROp, RL, RR> >
operator+(Calc<LOp, LL, LR> const& lhs, Calc<ROp, RL, RR> const& rhs)
{ return Calc< Plus, Calc<LOp, LL, LR>, Calc<ROp, RL, RR> >(lhs, rhs); }

template<typename LOp, typename LL, typename LR>
Calc< Plus, Calc<LOp, LL, LR>, Calc<int> >
operator+(Calc<LOp, LL, LR> const& lhs, int i)
{ return Calc< Plus, Calc<LOp, LL, LR>, Calc<int> >(lhs, i); }

template<typename ROp, typename RL, typename RR>
Calc< Plus, Calc<int>, Calc<ROp, RL, RR> >
operator+(int i, Calc<ROp, RL, RR> const& rhs )
{ return Calc< Plus, Calc<int>, Calc<ROp, RL, RR> >(i, rhs); }
```

- Expressions for capturing binary operator+
  - Special cases if one side is a number

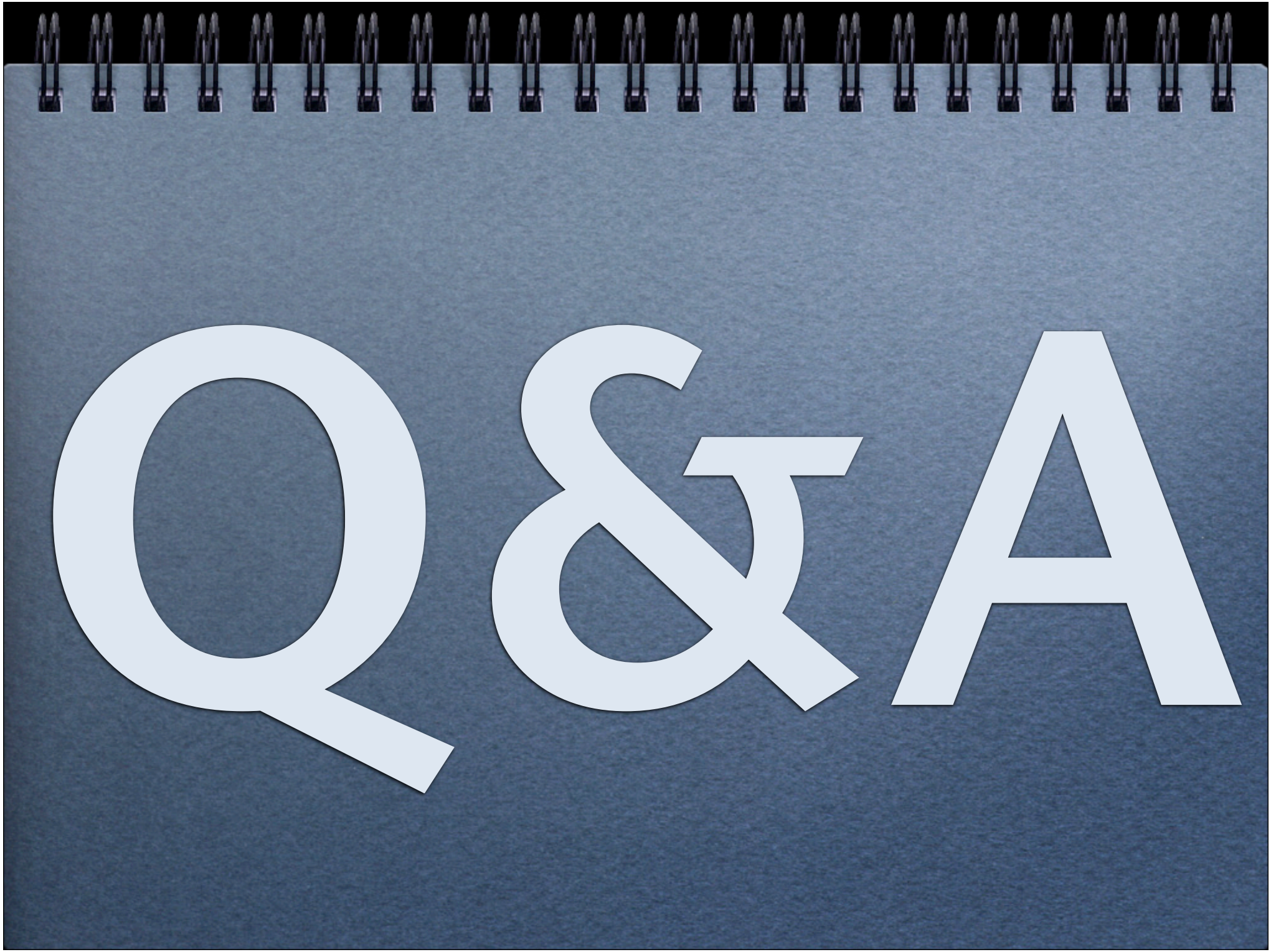
```
template<typename LOp, typename LL, typename LR, typename ROp, typename RL, typename RR>
Calc< Times, Calc<LOp, LL, LR>, Calc<ROp, RL, RR> >
operator*(Calc<LOp, LL, LR> const& lhs, Calc<ROp, RL, RR> const& rhs)
{ return Calc< Times, Calc<LOp, LL, LR>, Calc<ROp, RL, RR> >(lhs, rhs); }

template<typename LOp, typename LL, typename LR>
Calc< Times, Calc<LOp, LL, LR>, Calc<int> >
operator*(Calc<LOp, LL, LR> const& lhs, int i)
{ return Calc< Times, Calc<LOp, LL, LR>, Calc<int> >(lhs, i); }

template<typename ROp, typename RL, typename RR>
Calc< Times, Calc<int>, Calc<ROp, RL, RR> >
operator*(int i, Calc<ROp, RL, RR> const& rhs )
{ return Calc< Times, Calc<int>, Calc<ROp, RL, RR> >(i, rhs); }
```

- Similar expressions for capturing binary operator*

```
int main() {
    Calc<int> two(2);
    Calc<>    three(3);
    Calc<int> five(5);

    Calc<> seventeen(two + three * five);
    std::cout << seventeen << "==" << seventeen() << std::endl;

    Calc<> twentyfour(seventeen + 7);
    std::cout << twentyfour << "==" << twentyfour() << std::endl;

    Calc<> fourandtwenty(7 + seventeen);
    std::cout << fourandtwenty << "==" << fourandtwenty() << std::endl;
}
```

- Produces:

```
(2+(3*5))==17
((2+(3*5))+7)==24
(7+(2+(3*5)))==24
```


Q&A