Program specialization and metaprogramming

Laboratory work No. 2

Expression templates

1. Aims

To get acquainted with the expression templates technique in C++.

2. Tasks

1. To learn how to build linguistic structures representing a computation at compile time.
2. To learn how to use expression templates to achieve C++ code optimization.

3. Work

Expression Templates is a C++ technique for passing expressions as function arguments. The expression can be inlined into the function body, which results in faster and more convenient code than C-style callback functions. This technique can also be used to evaluate vector and matrix expressions in a single pass.

A common example is passing mathematical expressions to functions. The technique of expression templates allows expressions to be passed to functions as an argument and inlined into the function body. Expression templates also solve an important problem in the design of vector and matrix class libraries. The compiler produces a function instance which contains the expression inline. The technique allows developers to write code such as and have the compiler generate code to compute the result in a single pass, without using temporary vectors.

The trick is that the expression is parsed at compile time, and stored as nested template arguments of an "expression type". Let's work through a simple example, which passes a mathematical function to an integration routine.

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

```
template<class Expr>
void evaluate(DExpr<Expr> expr, double start, double end)
{
    const double step = 1.0;

    for (double i = start; i < end; i += step)
        std::cout << expr(i) << std::endl;
}
```

Next, define we DExpr class (short for double expression), which is a wrapper class for a specific type:

```
template<class A>
class DExpr {
private:
    A a_;
```

```
public:
    DExpr()
    { }

    DExpr(const A& x)
        : a_(x)
    { }

    double operator()(double x) const
    {
        return a_(x);
    }
};
```

An instance of the class DExpr<A> represents an expression. This class is a wrapper which disguises more interesting expression types. DExpr<A> has a template parameter (A) representing the interesting expression, which can be of type DBinExprOp (a binary operation on two subexpressions), DExprIdentity (a placeholder for the variable x), or DExprLiteral (a constant or literal appearing in the expression). These expression types are inferred by the compiler at compile time.

Next, define DExprIdentity, which is a placeholder for the variable in the expression.

```
class DExprIdentity {
public:
    DExprIdentity()
    { }

    double operator()(double x) const
    {
        return x;
    }
};
```

Next, define class DBinExprOp, which represents a binary operation on two expressions. Here A and B are the two expressions being combined, and Op is an applicative template representing the operation.

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

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

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

};
```

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

which can occur with operator/(). When a DBinExprOp's operator() is invoked, it uses the applicative template class to evaluate the expression inline:

Next, define sum and division operators needed to compile the expression x/(1.0+x):

```
// operator+(double, DExpr)
template<class A>
DExpr<DBinExprOp<DExprLiteral, DExpr<A>, DApAdd> >
operator+(double x, const DExpr<A>& a)
{
    typedef DBinExprOp<DExprLiteral, DExpr<A>, DApAdd> ExprT;
    return DExpr<ExprT>(ExprT(DExprLiteral(x), a));
}

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

Define class DExprLiteral, which represents a double literal which appears in the expression.

```
class DExprLiteral {
private:
    double value_;

public:
    DExprLiteral(double value)
    {
        value_ = value;
    }

    double operator()(double x) const
    {
        return value_;
    }
};
```

Define class DApAdd, which specifies the addition operation of two doubles:

```
class DApAdd {
public:
    DApAdd() { }

    static inline double apply(double a, double b)
    {
        return a + b;
    }
};
```

Define class DApDivide, which specifies the addition operation of two doubles

```
class DApDivide {
public:
    DApDivide() { }

    static inline double apply(double a, double b)
    {
```

```
        return a / b;
    }
};
```

In the main() function, call a function evaluate() which will evaluate f(x) at a range of points:

```
DExpr<DExprIdentity> x;          // Placeholder
evaluate(x / (1.0 + x), 0.0, 10.0);
```

When the above example is compiled, an instance of evaluate() is generated which contains code equivalent to the line

```
std::cout << i/(1.0+i) >> std::endl;
```

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

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

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

What is the result of program execution?

4. Challenge

1. Modify the program to incorporate more functions such as sqrt(), exp() and log() into expressions, by defining appropriate functions and applicative templates. Use these functions for implementing an expression object representing a normal distribution:

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

2. It is possible to generalize the classes presented here to expressions involving arbitrary types (instead of just doubles).

5. Report

Prepare a report of your work in this lab using a standard word processing application. The report is a single chapter of your final semester report. Final semester report will have to be uploaded to course Moodle page.

The content of report:

1. Title
2. Aims and tasks
3. Work done, described in steps and illustrated by screenshots and written or modified source code. Provide comments what you have done and what were the results (program outputs). Present answers to the questions given in the lab description.
4. Conclusions – what you have learned?

References

T. Veldhuizen, "Expression Templates," C++ Report, Vol. 7 No. 5 June 1995, pp. 26-31.