Understanding C++
Expression Templates

Angelika Langer
Training / Consulting
http://www.AngelikaLanger.com

Objectives

What are expression templates?
- an example of using templates for runtime-efficient computations
- an example of modern template programming techniques
  - template meta-programming
  - generative programming
Where it all started ...

- Erwin Unruh's prime number program ...
  - does not compile, but
  - calculates the prime numbers at compile-time and
  - emits them in error messages.
- works via recursive template evaluation
- useful for
  - evaluation of expressions (vector dot product, matrix operations)
  - calculation of constants (square root of N, prime numbers)
  - evaluation of logical expression (more readable STL functors)

Agenda

- compile-time computation of constant values
  - factorial
  - square root
- compile-time evaluation of expressions
  - dot product
  - arithmetic expression
- more examples of modern template programming
  - compile-time polymorphism
  - policy classes
  - template meta-programming
Runtime Computation of Factorial

The factorial of \( n \) is \( 1 \cdot 2 \cdot \ldots \cdot (n-1) \cdot n \), and the factorial of \( 0 \) is \( 1 \).

A recursive factorial function:

```c
int factorial (int n)
{
    return (n==0) ? 1: n*factorial(n-1); }
```

cout << factorial(4) << endl;

Compile-Time Counterpart

A class for compile time computation of the factorial:

```c
template <int n>
struct Factorial {
    enum { RET = Factorial<n-1>::RET * n }; 
};

template <>
struct Factorial<0> {
    enum { RET = 1 }; 
};
```

cout << Factorial<4>::RET << endl;
Agenda

- compile-time computation of constant values
  - factorial
  - square root
- compile-time evaluation of expressions
  - dot product
  - arithmetic expression
- more examples of modern template programming
  - compile-time polymorphism
  - policy classes
  - template meta-programming

Compile-Time Computation of Square Root

Given $N$, compute a compile-time table size $\lceil \sqrt{\sqrt{\sqrt{\sqrt{N}}}} \rceil$.

Example: `int table[Root<10>::root];`

Beginning of recursion:

```cpp
template <size_t Size, size_t Low=1, size_t High=Size>
struct Root;
```

```
Root<10> => Root<10,1,10>
```
Recursive definition of $\text{Root}::\text{root}$:

```cpp
template <size_t Size, size_t Low, size_t High>
struct Root {
    static const size_t root =
        Root<Size,(down?Low:mean+1),(down?mean:High)>::root;
    static const size_t mean = (Low+High)/2;
    static const bool down = ((mean*mean)>=Size);
};
```

Root $<10,1,10>$  =>  Root $<10,1,5>$

- mean = $(1+10)/2 = 5$
- down = $(5*5>=10) = true$

End of recursion:

```cpp
template <size_t Size, size_t Mid>
struct Root<Size,Mid,Mid> {
    static const size_t root = Mid;
};
```

Root $<10,1,5>$  =>  Root $<10,4,4>$

- mean = $(1+5)/2 = 3$
- down = $(3*3>=10) = false$

End of recursion: Root $<10,4,4>$  =>  Root $<10,4,4>$

- mean = $(4+5)/2 = 4$
- down = $(4*4>=10) = true$

End of recursion:
Given $N$, compute a compile-time table size $\lceil \sqrt[4]{\sqrt[4]{\sqrt[4]{N}} \rceil}$.

Example:

```cpp
int table[Root<10>::root];
```

### Agenda

- **compile-time computation of constant values**
  - factorial
  - square root
- **compile-time evaluation of expressions**
  - dot product
  - arithmetic expression
- more examples of modern template programming
  - compile-time polymorphism
  - policy classes
  - template meta-programming
Goal

- efficient computation of arithmetic expressions

- example:
  - dot product of 2 vectors of dimension N

```
int a[4] = {1,100,0,-1};
int b[4] = {2,2,2,2};
dot<4>(a,b);
```

- more generally:
  - arithmetic operations on multi-dimensional matrices

More Dynamic Use

- compile-time computation of integrals

```
template <class ExprT>
double integrate (ExprT e,double from,double to,size_t n)
{
    double sum=0, step=(to-from)/n;
    for (double i=from+step/2; i<to; i+=step)
        sum+=e.eval(i);
    return step*sum;
}

Identity x;
cout << integrate (x/(1.0+x),1.0,5.0,10) << endl;
```
Gauss Distribution

double sigma=2.0, mean=5.0;
const double Pi = 3.141593;
cout << integrate(
    1.0/(sqrt(2*Pi)*sigma) * exp(sqr(x-mean))/(-2*sigma*sigma))
,2.0,10.0,100) << endl;

STL Precicates

list<int> l;
Identity x;
count_if(l.begin(),l.end(), x >= 0 && x <= 100); 

... provided the logical and comparison operators are overloadded to yield expressions that evaluation to Boolean values

• another example of expression templates
  – improve readability
  – at no additional runtime cost
Agenda

- compile-time computation of constant values
  - factorial
  - square root
- compile-time evaluation of expressions
  - dot product
  - arithmetic expression
- more examples of modern template programming
  - compile-time polymorphism
  - policy classes
  - template meta-programming

Understanding Expression Templates

- try to understand template approach as an alternative to the classic OO approach
- expressions evaluation is an example of patterns in GOF
  - composite
  - interpreter
The Composite Pattern

The Composite pattern provides a way to represent a part-whole relationship where the client can ignore the difference between individual objects and compositions of objects.

- The *leaf* (terminal) defines the behavior for primitive objects in the composition.
- The *composite* (non-terminal) defines the behavior for components consisting of leaves.

Examples:
- syntax trees, expression evaluation
- aggregations and recursive structures and algorithms

```
Component
  Operation()

Leaf
  Operation()

Composite
  list<Component> g
  Operation()

for all children: g->Operation()
```
A Typical Composite Structure

Component:
– dot product of vectors of dimension N

Leaf:
– simple product of two numerical values, i.e. the dot product of a vector of dimension 1

Composite:
– consists of a leaf (dimension 1) and a composite (dimension N-1)
– calculation of the sum of the leaf’s value and the composite’s value
**OO Implementation à la GOF**

- **DotProduct**
  
  `evaluate()`

- **SimpleProduct**
  
  `float a, b`
  `evaluate()`

- **CompositeProduct**
  
  `SimpleProduct* s`
  `CompositeProduct* c`
  `evaluate()`

- `a * b`

- `s->evaluate() + c->evaluate()`

---

**Composite Structure of Dot Product**

- **DotProduct**
  
  `dimension: N`

- **SimpleProduct**
  
  `dimension: N - 1`

- **CompositeProduct**
  
  `dimension: N - 2`

- **SimpleProduct**
  
  `dimension: N - 3`
The Component

The base class that defines the interface common to leaves and composites:

template <class T>
class DotProduct {
    public:
        virtual ~DotProduct () {} 
        virtual T eval() = 0;
};

The Composite

template <class T>
class CompositeDotProduct : public DotProduct<T> {
    public:
        CompositeDotProduct (T* a, T* b, size_t dim)
            : s(new SimpleDotProduct<T>(a,b))
            , c((dim==1)?0:new CompositeDotProduct<T>(a+1,b+1,dim-1))
        {}
        virtual ~CompositeDotProduct () { delete c; delete s; }
        virtual T eval() {
            return (s->eval() + ((c)?c->eval():0)); }
    protected:
        SimpleDotProduct<T>* s;
        CompositeDotProduct<T>* c;
};
The Leaf

template <class T>
class SimpleDotProduct : public DotProduct <T> {
    public:
        SimpleDotProduct (T* a, T* b) :v1(a), v2(b) {}
        virtual T eval() { return (*v1)*(*v2); }
    private:
        T* v1; T* v2;
};

The Client

The code that initiates the recursive evaluation of the composite structure:

template <class T> T dot(T* a, T* b, size_t dim)
{ return (dim==1)
    ? SimpleDotProduct<T>(a,b).eval()
    : CompositeDotProduct<T>(a,b,dim).eval();
}

int a[4] = {1,100,0,-1};
int b[4] = {2,2,2,2};
dot(a,b,4);
A Simplified OO Implementation

Eliminate the representation of the composite and the leaf object as data members.
- Instead of passing information to the constructor and memorizing them for subsequent evaluation,
- pass them to the directly to the evaluation function.

```cpp
SimpleDotProduct<T* a, T* b> : v1(a), v2(b) {}
virtual T eval() { return (*v1)*(*v2); }
```

becomes

```cpp
T eval(T* a, T* b, size_t dim) { return (*a)*(*b); }
```

---

```cpp
template <class T>
class CompositeDotProduct : public DotProduct<T> {
public:
  virtual T eval(T* a, T* b, size_t dim)
  { return SimpleDotProduct<T>().eval(a, b, dim)
    + ((dim==1) ? 0 : CompositeDotProduct<T>().eval(a+1, b+1, dim-1));
  }
};

template <class T>
class SimpleDotProduct : public DotProduct<T> {
public:
  virtual T eval(T* a, T* b, size_t dim)
  { return (*a)*(*b); }
};
```
A Simplified OO Implementation

```
SimpleProduct().evaluate() + CompositeProduct().evaluate()
```

Implementation Using Templates

- The template solution does not need a base class.
  - Eliminate the Component base class.
- Implement the Composite as a class template using structural information as template arguments.
  - The dimension of the vector becomes a non-type template argument of the Composite.
- Implement the Leaf as a specialization of the Composite.
  - The SimpleDotProduct is a specialization of the CompositeDotProduct for dimension N = 1.
- Instead of run time recursion use compile time recursion.
  - Replace the recursive invocation of the virtual evaluation function by recursive template instantiation of a static evaluation function.
Implementation Using Templates

```cpp
template <size_t N, class T>
class DotProduct {
public:
    static T eval(T* a, T* b) {
        return DotProduct<1,T>::eval(a, b) + DotProduct<N-1,T>::eval(a+1, b+1);
    }
};
```

The Composite

```cpp
template <size_t N, class T>
class DotProduct {
public:
    static T eval(T* a, T* b) {
        return DotProduct<1,T>::eval(a, b) + DotProduct<N-1,T>::eval(a+1, b+1);
    }
};
```
A specialization of the Composite class template for dimension N = 1:

```cpp
template <class T>
class DotProduct<1,T> {
public:
    static T eval(T* a, T* b) {
        return (*a)*(*b);
    }
};
```

The Client

```cpp
template <size_t N, class T>
T dot(T* a, T* b) {
    return DotProduct<N,T>::eval(a,b);
}

int a[4] = {1, 100, 0, -1};
int b[4] = {2, 2, 2, 2};
dot<4>(a, b);
```
**Composite and Interpreter**

The dot product example is a degenerated form of a Composite because

- every Composite consists of exactly one Leaf and one Composite, and
- there is only one type of Leaf and one type of Composite.

The Interpreter pattern is a related pattern.

- The syntax tree in the Interpreter pattern is a Composite.

Let us discuss alternative implementations of the Interpreter pattern.

---

**Agenda**

- compile-time computation of constant values
  - factorial
  - square root
- compile-time evaluation of expressions
  - dot product
  - arithmetic expression
- more examples of modern template programming
  - compile-time polymorphism
  - policy classes
  - template meta-programming
The Interpreter Pattern

The Interpreter pattern provides a way to represent a language in form of an abstract syntax tree and an interpreter that uses the syntax tree to interpret language constructs.

The part-whole relationship of the Composite pattern corresponds to the relationship of an expression and its subexpressions in the Interpreter pattern.

- The leaf is a terminal expression.
- The composite is a non-terminal expression.
- Evaluation of the components is interpretation of the syntax tree and its expressions.

```
AbstractExpression
   Interpret()

TerminalExpression
   Interpret()

NonterminalExpression
   list<AbstractExpression> g
   Interpret()

for all children:
   g->Interpret()
```
Example: Arithmetic Expressions

SyntaxTree:
– the representation of an arithmetic expression such as
  \((a+1)*c\) or \(\log(\text{abs}(x-N))\)

Terminal:
– a numerical literal such as a constant of type `double`
– a reference to a variable of type `double`;
  its value might change between interpretations of the expression

NonTerminal:
– unary and binary expression consisting of one or two subexpressions;
– the expressions have different semantics such as `+`, `-`, `*`, `/`
  `++`, `--`, `exp`, `log`, `sqrt`

A Sample Syntax Tree

```
  (x+2) * 3
    /
  /    
(NonTerminal (Binary: Product))
     
  /    
(NonTerminal (Binary: Sum))  (Terminal (Literal: 3))
     
  /    
(Terminal (Literal: 2))  (Terminal (Variable: x))
```
The Abstract Expressions

The base classes that define the interface common to terminal and nonterminal expressions:

class AbstractExpr {
    public:
        virtual double eval() const = 0;
};

class TerminalExpr : public AbstractExpr {
};

class NonTerminalExpr : public AbstractExpr {
};
The Terminal Expressions

```cpp
class Literal : public TerminalExpr {
public:
    Literal(double v) : _val(v) {}    // constructor
    double eval() const { return _val; }
private:
    const double _val;
};

class Variable : public TerminalExpr {
public:
    Variable(double& v) : _val(v) {}  // constructor
    double eval() const { return _val; }
private:
    double& _val;
};
```

The Non-Terminal Expressions

```cpp
class BinaryExpr : public NonTerminalExpr {
protected:
    BinaryExpr(const AbstractExpr* e1, const AbstractExpr* e2)
        : _expr1(e1), _expr2(e2) {}   // constructor
    virtual ~BinaryExpr() {
        delete const_cast<AbstractExpr*>(_expr1);
        delete const_cast<AbstractExpr*>(_expr2);
    }
    const AbstractExpr* _expr1;
    const AbstractExpr* _expr2;
};

class Sum : public BinaryExpr {
public:
    Sum(const AbstractExpr* e1, const AbstractExpr* e2)
        : BinaryExpr(e1, e2) {}   // constructor
    double eval() const
    { return _expr1->eval() + _expr2->eval(); }
};
```
The Client

The code that creates the syntax tree and initiates its recursive interpretation:

```cpp
double x;
Product expr(new Sum(new Variable(x), new Literal(2)), new Literal(3));
cout << expr.eval() << endl;
```

Implementation Using Templates

- The template solution does not need a base class.
  - Eliminate all abstract base classes.
- Implement the NonTerminalExpression as a class template using structural information as template arguments.
  - The types of the subexpressions are type template arguments of the NonTerminalExpression.
- Parameterize the NonTerminalExpression with the type of operation.
  - The actual operation (+,-,*,/;++,-;abs,exp,log) is stored as a function object and its type is a template argument.
- Implement the TerminalExpressions as normal classes.
- Instead of run time recursion use compile time recursion.
  - Replace the recursive invocation of the virtual evaluation function by recursive template instantiation.
Implementation Using Templates

The Terminal Expressions

```cpp
class Literal {
public:
    Literal(const double v) : _val(v) {}
    double eval() const { return _val; }
private:
    const double _val;
};

class Variable {
public:
    Variable(double& v) : _val(v) {}
    double eval() const { return _val; }
private:
    double& _val;
};
```
The Non-Terminal Expressions

```cpp
template <class Expr1, class Expr2, class BinOp>
class BinaryExpr {
    public:
        BinaryExpr(Expr1 e1, Expr2 e2, BinOp op=BinOp()) : _expr1(e1), _expr2(e2), _op(op) {}
        double eval() const { return _op(_expr1.eval(), _expr2.eval()); }
    private:
        Expr1 _expr1;
        Expr2 _expr2;
        BinOp  _op;
};
```

As operations we can use the pre-defined STL function objects `plus`, `minus`, `multiplies`, `divides`, etc. or define our own functions objects as needed.

Creator Functions

A binary expression representing a sum would be of type `BinExpr<Expr1, Expr2, plus<double>>`.

For convenience, define creator functions that take advantage of automatic template argument deduction for function templates:

```cpp
template <class Expr1, class Expr2>
BinaryExpr<Expr1, Expr2, plus<double>> makeSum(Expr1 e1, Expr2 e2) {
    return BinaryExpr<Expr1, Expr2, plus<double>>(e1, e2);
}
```

```cpp
template <class Expr1, class Expr2>
BinaryExpr<Expr1, Expr2, multiplies<double>> makeProd(Expr1 e1, Expr2 e2) {
    return BinaryExpr<Expr1, Expr2, multiplies<double>>(e1, e2);
}
```
The Client

The code that creates the syntax tree and initiates its recursive interpretation:

```cpp
double x;
BinaryExpr< BinaryExpr< Variable, Literal, plus<double> >, 
           Literal, multiplies<double> >
expr = makeProd(makeSum(Variable(x), Literal(2)), 
                 Literal(3));
cout << expr.eval() << endl;
```

Operator Overloading

Often the type of the expression is not even needed:

```cpp
double x;
cout
   << makeProd(makeSum(Variable(x), Literal(2)), 
               Literal(3)).eval() 
   << endl;
```

Take advantage of operator overloading in C++ and define the creator functions as overloaded operators.

```cpp
Variable v(x);
cout << eval((v+2)*3.0) << endl;
```
First attempt

Define the creator functions as overloaded operators:

```cpp
template <class ExprT1, class ExprT2>
BinaryExpr <ExprT1, ExprT2, plus<double> >
operator+(ExprT1 e1, ExprT2 e2)
{ return BinaryExpr <ExprT1, ExprT2, plus<double> >(e1, e2); }
```

An expression such as `x+2` would evaluate to

```
BinaryExpr <double, int, plus<double> >(x, 2)
```

which is not exactly what we need. From just the type the compiler cannot distinguish between a literal and a variable of type `double`. We need a binary expression that stores a `Variable` and a `Literal` expression internally.

Requirement

Define variables as `Variable` expressions:

```cpp
double x; Variable v(x);
cout << eval((v+2)*3.0) << endl;
```

The compiler can now deduce:

```
BinaryExpr <Variable, int, plus<double> >(v, 2)
```

Inside the binary expression the variable would be stored as an expression of type `Variable` that refers to the variable `x` of type `double`. 
Solution

Inside the binary expression store constants of any numerical type as literal expressions. For this purpose define for each expression type and all numerical types what their corresponding expression type is:

```cpp
template <class ExprT> struct exprTraits {
    typedef ExprT expr_type;
};

template <> struct exprTraits<double> {
    typedef Literal expr_type;
};

template <> struct exprTraits<int> {
    typedef Literal expr_type;
};
```

Modify the BinaryExpression Class

Inside the binary expression convert expressions to their expression type as defined in the expression traits:

```cpp
template <class ExprT1, class ExprT2, class BinOp>
class BinaryExpr {
public:
    BinaryExpr(ExprT1 e1, ExprT2 e2, BinOp op=BinOp()) : _expr1(e1), _expr2(e2), _op(op) {}
    double eval() const {
        return _op(_expr1.eval(), _expr2.eval());
    }
private:
    exprTraits<ExprT1>::expr_type _expr1;
    exprTraits<ExprT2>::expr_type _expr2;
    BinOp _op;
};
```
Define a helper function:

```
template <class ExprT>
double eval(ExprT e) { return e.eval(); }
```

In the example

```
double x; Variable v(x);
cout << eval((v+2)*3.0) << endl;
```

the expression `(v+2)*3.0` evaluates to the creation of a temporary expression object and `eval()` invokes its interpretation.
**Interpretation**

```
ExprT2
 tmp2 =
  BinaryExpr<ExprT1,double,multiplies<double>>(
    tmp1, 3.0)

val( (v+2) * 3.0 )
```
Practical Applications

So far, the interpretation of the syntax tree is rather static.

- The syntax tree is created and interpreted only once.

A more dynamic usage model is possible, where a given syntax tree can be evaluated repeatedly for different input values.

```cpp
template <class ExprT>
double integrate (ExprT e, double from, double to, size_t n)
{
    double sum=0, step=(to-from)/n;
    for (double i=from+step/2; i<to; i+=step)
        sum+=e.eval(i);
    return step*sum;
}
```

```cpp
Identity x;
cout << integrate (x/(1.0+x), 1.0, 5.0, 10) << endl;
```

Minor Modifications

```cpp
class Literal {
public:
    Literal(double v) : _val(v) {} 
    double eval(double) const { return _val; }
private:
    const double _val;
};
```

```cpp
class Identity {
public:
    double eval(double d) const { return d; }
};
```

```cpp
template <class ExprT1, class ExprT2, class BinOp>
class BinExpr {
public:
    double eval(double d) const 
    { return _op(_expr1.eval(d), _expr2.eval(d)); }
};
```
Incorporating Functions

We can incorporate functions such as `sqrt()`, `exp()`, etc.:

```cpp
double sigma=2.0, mean=5.0;
const double Pi = 3.141593;
cout << integrate(
    1.0/(1.0/(1.0/(1.0/(sqrtsqrtsqrtsqrt(2*Pi)*sigma) * (2*Pi)*sigma) * (2*Pi)*sigma) * (2*Pi)*sigma) * expexpexpexp((((sqrsqrsqrsqr(x(x(x(x----mean)/(mean)/(mean)/(mean)/(----2*sigma*sigma))2*sigma*sigma))2*sigma*sigma))2*sigma*sigma))
    ,2.0,10.0,100) << endl;
```

and calculate the Gauss distribution:

```cpp
template <class ExprT>
UnaryExpr<ExprT,double(*)(double)> sqrt(const ExprT& e)
{ return UnaryExpr<ExprT,double(*)(double)>(e,::std::sqrt); }
```

These functions are provided via creator functions that yield unary operations:

```cpp
template <class ExprT, class UnaryOp>
class UnaryExpr {
public:
    UnaryExpr(ExprT e, UnaryOp op=UnaryOp())
        : _expr(e),_op(op) {}
        double eval(double d) const { return _op(_expr.eval(d)); }
private:
    exprTraits<ExprT>::expr_type _expr;
    UnaryOp _op;
};
```

```cpp
template <class ExprT>
UnaryExpr<ExprT,double(*)(double)> exp(const ExprT& e)
{ return UnaryExpr<ExprT,double(*)(double)>(e,::std::exp); }
```
Further Applications in Practice

If the `interpret()` or `eval()` function is provided as an overloaded `operator()`(), the expressions can serve as function objects for the STL.

```cpp
list<int> l;
Identity x;
count_if(l.begin(), l.end(), x >= 0 && x <= 100);
```

... provided the logical and comparison operators are overloaded to yield expressions that evaluation to Boolean values.

Agenda

- compile-time computation of constant values
  - factorial
  - square root
- compile-time evaluation of expressions
  - dot product
  - arithmetic expression
- more examples of modern template programming
  - compile-time polymorphism
  - policy classes
  - template meta-programming
Run-Time Polymorphism

Classic object-oriented approach:

```cpp
class Rotatable {
public:
    virtual void rotate(int) = 0;
};
class Ellipse : public Rotatable {
public:
    Ellipse(int x, int y)
        : Xradius(x), Yradius(y) {} 
    virtual void rotate(int);
private: int Xradius, Yradius;
};
```

Using run-time polymorphism:

```cpp
void vertical_flip(Rotatable& d) {
    d.rotate(180);
}

Ellipse ellipse(100,600);
vertical_flip(ellipse);
Rectangle rectangle(999,500);
vertical_flip(rectangle);
```
Compile-Time Polymorphism

Replace intrusive inheritance by name commonality:

```cpp
class Ellipse{
 public: Ellipse(int x, int y)
 : Xradius(x), Yradius(y) { }
     void rotate(int);
 private: int Xradius, Yradius;
};

class Rectangle {
 public: Rectangle(int x, int y)
 : Xedge(x), Yedge(y) { }
     void rotate(int);
 private: int Xedge, Yedge;
};
```

Using compile-time polymorphism:

```cpp
template <class Rotatable>
void vertical_flip(Rotatable& d)
{ d.rotate(180); }

Ellipse ellipse(100, 600);
vertical_flip(ellipse);

Rectangle rectangle(999, 500);
vertical_flip(rectangle);
```
Run-Time vs. Compile-Time Dispatch

Inheritance-based polymorphism (OOP) can be replaced by templates (GP = generic programming).

Overhead:
- GP: code bloat & compile-time overhead
- OOP: run-time overhead

Conformance to an interface:
- GP: common names
- OOP: a common base class

Integration of built-in types:
- GP: overloaded operators $\leftrightarrow$ native operators
  - function objects $\leftrightarrow$ function pointers
- OOP: not possible

Agenda

- compile-time computation of constant values
  - factorial
  - square root
- compile-time evaluation of expressions
  - dot product
  - arithmetic expression
- more examples of modern template programming
  - compile-time polymorphism
  - policy classes
  - template meta-programming
**Policy Specifications**

- classic quick sort function from C library
  - uses function pointer for sorting criterion
  ```c
  void qsort(void **base, size_t num, size_t width,
             int (*compare)(const void **elem1, const void **elem2));
  ```

- sort algorithm from the C++ library
  - uses comparator type (function pointer or function object)
  ```cpp
template<class RandomAccessIterator, class Comparator>
void sort(RandomAccessIterator first, RandomAccessIterator last,
          Comparator cmp);
```
The Strategy Pattern

Context

Strategy

Interface()

Concrete Strategy

Strategy Interface()

Concrete Strategy

Strategy Interface()

s -> Strategy Interface()

An Example - Sorting of Strings

Context:
– a class or function that sorts strings, or
– a sorted collection

Strategy:
– ASCII sorting order
– case-insensitive sorting order
– culture-sensitive, dictionary sorting order
**OO Implementation à la GOF**

```
Sorter
Comparator* _cmp
Sorter(Comparator*)
sort()
```

```
Comparator
lessthan() = 0
```

```
lessthan()
```

```
ASCII

lessthan()
```

```
case insensitive

lessthan()
```

```
culture sensitive

lessthan()
```

---

**The Context**

```cpp
class Sorter
{
public:
    explicit Sorter(Comparator* c) : _cmp(c) {}

    void sort(const StringContainer& c)
    { ... _cmp->lessthan(lhs,rhs) ... }

private:
    Comparator* _cmp;
};
```
The Strategies

```cpp
class Comparator {
public:
  virtual bool lessthan(const string&, const string&) const = 0;
};

class Ascii : public Comparator {
public:
  virtual bool lessthan(const string& lhs, const string& rhs) const {
    return lhs < rhs;
  }
};
```

Using Context and Strategies

```cpp
StringContainer cont;
Ascii asciiCmp;
CaseInsens caseInsCmp;
Sorter asciiSrt(&asciiCmp);
Sorter caseInsSrt(&caseInsCmp);
asciiSrt.sort(cont);
caseInsSrt.sort(cont);
```

- cannot tell from the type of the `Sorter` how it sorts
- need not re-compile `Sorter` when new strategies are added
- run time binding for `lessthan()` based on type of comparator
- polymorphism through “another level of indirection”
Implementation Using Templates

Sorter
Comparator _cmp
Sorter(Comparator) sort()

cmp.lessthan()

The Context

```cpp
template <typename Comparator>
class Sorter
{
public:
    explicit Sorter(const Comparator& c = Comparator())
        : _cmp(c) {}

    void sort(const StringContainer& c)
    {
        ... _cmp.lessthan(lhs, rhs) ...
    }

private:
    Comparator _cmp;
};
```
The Strategies

```cpp
class Ascii {
public:
    bool lessthan(const string& lhs, const string& rhs) const
    { return lhs < rhs; }
};

class CaseInsensitive {
public:
    static bool lessthan(const string& lhs, const string& rhs)
    {
        locale loc;
        const ctype<char> fac = use_facet<ctype<char>>(loc);
        string tmp1(lhs), tmp2(rhs);
        fac.tolower(tmp1.begin(), tmp1.end());
        fac.tolower(tmp2.begin(), tmp2.end());
        return tmp1 < tmp2;
    }
};

class CultSens {
public:
    explicit CultSens(locale l = locale()) : _loc(l) {}
    bool lessthan(const string& lhs, const string& rhs) const
    { return _loc(lhs, rhs); }
private:
    locale _loc;
};
```
Using Context and Strategies

- can tell from the name of the type of the `Sorter` how it sorts
- need to compile instantiations of `Sorter` for new strategies
- compile time binding for `sort()` and `less than()`
- polymorphism through templates

```cpp
StringContainer cont;
Sorter<Ascii>      srt1;
Sorter<CaseInsens> srt2;
Sorter<CultSens>   srt3(CultSens(locale("German")));
srt1.sort(cont);
srt2.sort(cont);
srt3.sort(cont);
```

Evaluation of Policy Specifications

- policy specification via template parameters is more flexible
  - not restricted to just one function signature
  - both function pointers and function objects allowed
  - possibility for bundling policies
Examples from Standard C++

- sorting strategy and allocation policy of containers

```cpp
template <class ElementType, 
         class Comparator = less<Key>, 
         class Allocator  = allocator<Key> >
class set;
```

- character handling and allocation policy of strings

```cpp
template<class CharacterType, 
         class CharacterTraits = char_traits<charT>, 
         class Allocator = allocator<charT> > 
class basic_string;
```

A Sorting Strategy

- comparator type `less` from the STL
  - just one functionality provided
  - via overloaded function call operator

```cpp
template <class T>
struct less : binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const;
};
```
A Policy Bundle

- the standard allocator
  - a bundle of functionalities
  - provided as non-static member functions

```cpp
template <class T> class allocator {
public:
  ...
  pointer allocate(size_t);
  void deallocate(T* p, size_t n);
  void construct(T* p, const T& val);
  void destroy(T* p);
};
```

Another Policy Bundle

- the standard character traits for type `char`
  - a bundle of static functionality

```cpp
template<>
struct char_traits<char> {
  static bool eq(const char& c1, const char& c2);
  static bool lt(const char& c1, const char& c2);
  static char* move(char* s1, const char* s2, size_t n);
  static char* copy(char* s1, const char* s2, size_t n);
  ...
};
```
Alexandrescu’s Smart Pointer

- the configurable smart pointer class template from the Loki library
  - uses not just template type parameters, but also template template parameters

```cpp
template <
  typename T,
  template <class> class OwnershipPolicy = RefCounted,
  class ConversionPolicy = DissallowConversion,
  template <class> class CheckingPolicy = AssertCheck,
  template <class> class StoragePolicy = Default SPStorage
>
class SmartPointer;
```

Alexandrescu’s Smart Pointer

- ownership policy
  - deep copy, destructive copy, no copy
  - reference counted (thread-safe or not)
- conversion policy
  - allow or disallow implicit conversion to underlying pointer type
- checking policy
  - reject null
  - no check
- storage policy
  - default storage (does delete)
  - array storage (does array delete[])
  - heap storage (calls free())
Benefits of Meta-Programming

Very good performance due to

- exclusive use of static binding; polymorphic behavior can be simulated statically.
- inlining; enables the compiler to optimize aggressively.

Problems with Meta-Programming

- Debugging. practically not possible; there are tricks though.
- Error reporting. no influence on the compiler message.
- Readability of code. it looks awkward.
- Compilation times. increases by orders of magnitude.
- Compiler limits. truncation of identifiers, loop limits exceeded.
- Portability. some compilers still not support standard C++
Expressions and template libraries

The Blitz Project
http://oonumerics.org/blitz/
A C++ class library for scientific computing which uses template techniques to achieve high performance.

PETE (Portable Expression Template Engine)
http://www.acl.lanl.gov/pete/
A portable C++ framework to easily add powerful expression templates.

POOMA (Parallel Object-Oriented Methods and Applications)
http://www.acl.lanl.gov/pooma/
- An object-oriented framework for applications in computational science requiring high-performance parallel computers.
- Considerable success in real applications including gyrokinetic particle-in-cell plasma simulation and multimaterial compressible hydrodynamics.
- Developed at Los Alamos National Laboratory, New Mexico.
Further Links

Research Centre Jülich  
http://www.fz-juelich.de/zam/cxx/

An impressive directory of C++ resources such as books, articles, FAQs, other C++ pages, compilers, libraries, etc.

See in particular the links to other C++ libraries at 
http://www.fz-juelich.de/zam/cxx/extmain.html#lib


Patterns

Design Patterns  
Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides  
Addison-Wesley, 1994
Contact Info

Angelika Langer

Training & Mentoring
Object-Oriented Software Development in C++ & Java

Munich, Germany

http://www.AngelikaLanger.com