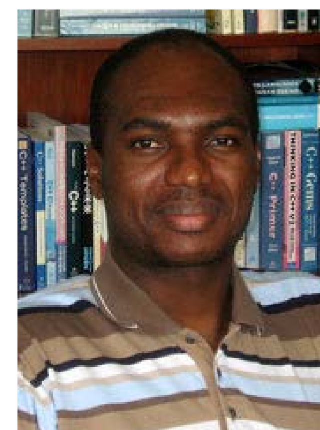
Pattern Matching for C++

Yuriy Solodkyy (Microsoft)
Gabriel Dos Reis (Microsoft)
Bjarne Stroustrup (Morgan Stanley)
All: formerly Texas A&M University



Purpose

- To start a discussion
  - Would PM be good for C++?
  - What would PM for C++ look like?
  - What are the costs?
- To give a starting point
  - Syntax, aims, semantics
  - Based on
    - the Mach7 library implementation
      - A C++11 library
    - Ideas from a variety of functional languages
      - Incl., ML, F#, Haskell, Scala, OpenAxiom

Purpose

- I want an integrated set of language features and libraries for C++
- "Multiparadigm programming" is at best a placeholder
  - I have been saying that for almost a decade (maybe more)
  - Anyone has a better term?
- Don't try to define "isolated" mini-languages within C++

Overview

- What is pattern matching?
- Why consider PM for C++?
- Syntax
- Design questions
- Summary: pros and cons

- This presents a language design based on Mach7
  - Y. Solodkyy, G. Dos Reis, and B. Stroustrup: Open Pattern Matching for C++. ACM GPCE'13.
    - http://bit.ly/Mach7 - GitHub of the project
    - http://bit.ly/Mach7CppNow - slides of the C++ Now 2014 talk
    - http://bit.ly/Mach7CppNowVideo - video of the C++ Now 2014 talk
    - http://bit.ly/AcceptNoVisitors - slides of the CppCon 2014 talk
    - http://bit.ly/AcceptNoVisitorsVideo - video of the CppCon 2014 talk
  - We have an implementation, but not a language design ☺

What is pattern matching?

- A way of picking values using a variety of criteria
  - Value
    - is x nullptr?
  - Type
    - is s a Circle?
  - Concept
    - is Iter a Random_access_iterator
  - Predicate
    - is sz less than 14
- Type safe unions
- A way of avoiding visitors for class hierarchies
- A way of decomposing objects into parts
- A way of structuring computations
- A simpler notation for some examples

Simula-inspired derived class lookup

- Use some form of RTTI to determine which derived class
  - At least one virtual function in base class
  - Could be costly (but see mach7)
  - Organizes code as lists of cases (not OO)
  - Non-intrusive
  - No access to private members

```
double area(const Shape& s)
{
    inspect (s) {
    when Circle:     return 2*pi*radius();         // not s.radius()
    when Square:     return height()*width();
    default:         error("unknown shape");
    }
}
```

Found by member lookup in Circle

Found by member lookup in Square

An alternative to visitors

- Provide a suitable public interface to classes in a hierarchy

```
class Expr  { virtual ~Expr(); };
class Value : Expr { int value; };
class Plus  : Expr { Expr& a; Expr& b; };
class Minus : Expr { Expr& a; Expr& b; };
class Times : Expr { Expr& x; Expr& y; };
class Divide: Expr { Expr& divident; Expr& divisor; };

int eval(const Expr* e) // not a virtual function, not a member
{
        inspect (e) {
        when Value:     return value;
        when Plus:      return eval(a)+eval(b);
        when Minus:     return eval(a)-eval(b);
        when Times:     return eval(x)*eval(y);
        when Divide:    return eval(dividend)/eval(divisor);
        }
}
```

Pascal-inspired discriminating union

- Have a hidden member/field/discriminant to say which union/record member is currently used
  - Type safe
  - Optimizable
  - A plain union is faster if you don't check

```
variant U { int; double; };  // needs to be distinguished from union
                             // std::variant?

istream& operator<<(istream& os, const U& u)
{
        inspect (u) {
        when {int a}:        return os << a;        // {type local-name} pair
        when {double d}:     return os << d;
        }
}
```

Predicate as discriminant?

- Select an alternative by a predicate rather than a separate stored value

```
struct string_rep {
        int sz;
        variant U (sz>12) {     // select in [0:n); false==0, true==1
                char [12];  // characters in rep itself
                {           char* p;            // characters in free store
                            int space;          // unused allocated space
                }
        };
        char* str()
        {
                inspect (*this) {
                when {0 x}:         return x;   // {value local-name} pair
                when {1 y}:         return y.p;
                }
        }
};
```

Concept-based overloading?

- Should we be able to match against concepts?

```
void advance(Iterator p, int n)
{
        inspect(Iterator) {
        when Forward_iterator:
        when Bidirectional_iterator:        while(--n>0) ++p;
        when Randomaccess_iterator:         p+=n;
}
```

- PM Is very much like overloading
- P.S. should we allow fall-through for empty patterns?

Observations

- Type safety has been maintained/guaranteed
- We don't need switch/case-style fall through
  - And won't propose it
- For class hierarchies
  - the set of alternatives is open
    - a default is needed
  - The alternatives are not disjoint
    - **when**-order matters
  - One RTTI operation: not a if-then-else chain
- For unions
  - The set of alternatives is closed
    - We can give an error if not all cases are covered
  - The alternatives are disjoint
- Doesn't look very FP
  - E.g., no algebraic data types

Patterns

- We can match several entities at once
  - We group by {} when matching more than one value
  - We need to represent: value, type, and placeholder

```
template<typename T, typename U>
void f(T& x, U xx)
{
        inspect (x,xx) {
        when {int* p,0}:                p=nullptr;
        when {_a,int}:      …           // _a is a placeholder matching everything
                                        // shorthand for auto _a
        }
}
```

- Place holders become important: what should they look like?

Selection among alternatives

- A pattern is **{ … }**
  - A single type of value doesn’t need parentheses
  - When-clauses are executed in order

```
double factorial(int n)
{
	assert(0<=n);

	inspect(n)  {
	when 0:				return 1;
	when {double m}:		return m*factorial(m−1); // m initialized by n
	}
}
```

Tuples

- Tuples are recursively defined
  - tuples have a tail (or should have)

```
template<typename T…> void print(tuple<T…>& t)
{
    inspect (t) { // for this to work, inspect must know about …
    when {}:            ;
    when {auto a}:      cout<<a;
    when {_a,_tail}:    cout<<a; print(tail);
    }
}
```

Tuples

- Tuples are recursively defined
  - tuples have a tail (or should have)

```
template<typename T…, typename U…>
bool operator==(tuple<T…>& t, tuple<U…>& u)
{
        inspect (t,u) {
        when {{},{}}:         return true;
        when {_,{}}:          return false;        // _ is the unnamed placeholder
        when {{},_}:          return false;
        default:              if (head(t)!=head(u)) return false;
                              return tail(t)==tail(u);
        // when {{tHead,tTail}, {uHead,uTail}}: return tHead==uHead && tTail==uTail;
        // when {{head, tail}, {head, tail}}: return true;
        // when {{head,tail}, {+head,+tail}}: return true;
        }
}
```

Ranges

- Ranges: vectors, lists, etc.
- A pattern is parenthesized
  - Can “list comprehension” be done with C++ containers and/or ranges?
  - C++ ranges are [b:e) not recursive (head,tail)

```
void print(Range<T> r)    // use PM?
{
        inspect(r)  {
        when {}:            ;                              // Oops! undefined
        when {_p,_q}:       cout<<*_p; print(++_p,_q);     // iterators
        }
}
```

Ranges

- A pattern is parenthesized
  - Can “list comprehension” be done with C++ containers and/or ranges?
  - C++ ranges are [b:e) not recursive (head,tail)

```
void print(Range<T> r)    // use PM?
{
        for (x : r) cout << x;
}
```

- Pattern matching will never be the only control structure

Ranges

- We can write a pattern for traversing [a:b)
  - But should we?
  - FP is just syntactic sugar
  - Iteration can be faster than recursion

```
void print(Range<T> r)              // use PM?
{
    inspect(begin(r),end(r))  {
    when {_b,_e} | _b==_e:              return;   // conditional match
    when {Iterator b, Iterator e}:      cout<<*_b; print(++_e);
    }
}
```

Balancing Red-Black Tree

```
class T{ enum color{black,red} col; T* left; K key; T* right; };

void balance(T& n)
{
  T::color col;
  const col B = T::black, R = T::red;

  inspect(n) {
  when T{B, T{R, _a, _x, _b}, _z, _d}:   // or use the | combinator
  when T{B, T{R, _a, _x, T{R, _b, _y, _c}}, _z, _d}:
  when T{B, _a, _x, T{R, T{R, _b, _y, _c}, _z, _d}}:
  when T{B, _a, _x, T{R, _b, _y, T{R, _c, _z, _d}}}:
        // modify n, *n.left, and *n.right
        n.col = R;
        *n.left = T{B,_a,_x,_b};
        n.key = y;
        *n.right = T{B,_c,_z,_d};
  when T{col, _, _, _} return;
  }
}
```

Patterns

- There are many kinds of patterns (in a variety of languages) and ways of composing them
  - Constants
  - Variables
  - Or
  - And
  - Tuple
  - Nested
  - …
- We don’t have to support them all
  - Keep simple things simple
  - Don’t make complicated things unnecessarily difficult

Patterns

- Which patterns should we be able to express?
  - Tersely?
  - Simply?
  - Elegantly?
  - Experts only?
- We need more archetypical examples
  - "We can do it is not a sufficient reason to do it"
- How do PM interact with library types?
  - std::tuple, std::pair, std::optional, std::variant
  - Concepts, such as Range?
- Lots of little syntax questions
  - What should placeholders look like?

Why consider PM for C++

- PM provides type-safe selection among alternatives
- PM provides a more general switch
- PM provides an alternative to the visitor pattern
- PM is the basic of much functional programming
  - Currently very popular
  - We get many “suggestions” to add it to C++
- PM can dramatically shorten programs
- Switch-on-type saves us from switching on enums
- PM can be efficiently implemented in C++
  - Mach7 library and paper

Why not introduce PM?

- Yet another language feature
  - To overuse
  - Stability: We have enough new stuff for C++17
- Unions are good enough
  - And if you don't check the tag unions are faster
- Switch-on-type breaks modularity
  - Code organized by function rather than by type
  - The reason C with Classes did not have **inspect**

Suggested approach

- Start with the simple cases
- Decide on place holder syntax
  - **_**, **_a**, **_1**, declare, **`a**, …
- Decide on generality of patterns
  - Mach7 supports ***a lot***
    - Variable patterns (yes)
    - n+k patterns (no)
    - equivalence patterns
    - equivalence combinators (+)
    - …

???