

Microsoft

Accept No Visitors

Yuriy Solodkyy
Microsoft

CppCon 2014
September 12, 2014, Bellevue, WA

Based on work performed in collaboration with:
Bjarne Stroustrup, Gabriel Dos Reis, Peter Pirkelbauer
at Texas A&M University

Partially supported by NSF grants:
CCF-0702765, CCF-1043084, CCF-1150055

Executive Summary

- Keep simple things simple
- Don't make complex things unnecessarily complex
- Don't make things impossible
- Constraint: Don't sacrifice performance
- C++ is expert friendly
 - and becoming more so
- C++ must not be **just** expert friendly
 - Serving novices and “occasional users” is very important
 - Most of the time, *I* don't want to be a language lawyer



Outline

- Visitor Design Pattern
 - Visitors solve real-world problems
 - but there are better abstraction mechanisms to solve the class of problems they address
- Two Alternatives
 - Pattern Matching
 - demonstrated by a library, but it is the language solution based on it we advocate for
 - Open Multi-Methods
 - demonstrated on a language extension we worked on
- Goal
 - Compare the solutions on some common examples
 - In a hope to gather more supporters for a language feature
 - Implementation details are not the goal, we have presented them before

Visitor Design Pattern

- A technique for performing an external case analysis on object structure
- Numerous implementations
 - Polymorphic classes
 - Tagged classes
 - Polymorphic exceptions
 - Templates
- We will use polymorphic classes for demonstration
 - most common when discussing VDP
 - but many of the issues are present in all of these implementations

“Represents an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.”

-- GoF

Visitor Design Pattern Use Cases

- Adding new functionality to class hierarchies
 - functionality you haven't foreseen
- Analysis in object graphs
 - knowledge graphs
 - traversals
- Interactions in games
 - Objects with objects/scene
- Customization points
- Plug-ins
- Anything you'd otherwise use virtual functions for

Motivating Example

Grammar

// Abstract syntax of boolean expressions

BoolExp ::= VarExp | ValExp | NotExp | AndExp | OrExp | '(' BoolExp ')'

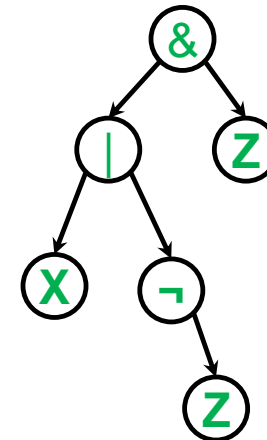
VarExp ::= 'A' | 'B' | ... | 'Z'

ValExp ::= 'true' | 'false'

NotExp ::= 'not' BoolExp

AndExp ::= BoolExp 'and' BoolExp

OrExp ::= BoolExp 'or' BoolExp



Motivating Example: Perfect Interface

```
struct BoolExp
{
    virtual void      print()                const = 0;
    virtual BoolExp*  copy()                 const = 0;
    virtual bool      eval(Context&)         const = 0;
    virtual BoolExp*  replace(const char*, const BoolExp*) = 0;
    virtual bool      equal(const BoolExp*)   const = 0;
    virtual bool      match(const BoolExp*, Assignments&) const = 0;
};
```

Note to self:

- Did we miss any?
 - unify, replace_subtree, cnf, dnf, etc.

Motivating Example: Perfect Interface

```
struct VarExp : BoolExp { ... string    name;  };
struct ValExp : BoolExp { ... bool      value; };
struct NotExp : BoolExp { ... BoolExp*  e;    };
struct AndExp : BoolExp { ... BoolExp*  e1; BoolExp*  e2; };
struct OrExp  : BoolExp { ... BoolExp*  e1; BoolExp*  e2; };

void      NotExp::print() const { std::cout << '!'; e->print(); }
BoolExp*  NotExp::copy()  const { return new NotExp(e->copy()); }
bool      NotExp::eval(Context& c) const { return !e->eval(c); }
BoolExp*  NotExp::replace(const char* n, const BoolExp* x)
                        { e = e->inplace(n,x); return this; }
```

Note to self:

- Should we group them by class or method in translation units?

Polymorphic Interfaces: Summary

Pros

- Extensibility of data
 - Adding new variant: easy - just derive it from BoolExp
- Modularity
- Encapsulation
- Works in the presence of
 - multiple inheritance
 - dynamic linking

Cons

- No extensibility of functions
 - Adding new function: hard - requires changing the interface
- No local reasoning
 - Cases can be scattered around translation units
- Non-relational
 - Inherently on a single argument

Visitor Design Pattern

- List your cases

```
struct VarExp;  
struct ValExp;  
struct NotExp;  
struct AndExp;  
struct OrExp ;
```

- Define a case analysis (visitation) interface

```
struct BoolExpVisitor  
{  
    virtual void visitVarExp(const VarExp&) {}  
    virtual void visitValExp(const ValExp&) {}  
    virtual void visitNotExp(const NotExp&) {}  
    virtual void visitAndExp(const AndExp&) {}  
    virtual void visitOrExp (const OrExp &) {}  
};
```

So how can we implement these using the Visitor Design Pattern instead?

- We assume the same class hierarchy, but none of the earlier virtual functions declared

Note to self:

- Requires foresight of cases
- Don't use overloading of visit
 - unless you need to...

Visitor Design Pattern

- Embed accept into the class hierarchy

```
struct BoolExp { virtual void accept(BoolExpVisitor&) const = 0; }
struct VarExp : BoolExp { void accept(BoolExpVisitor& v) const { v.visitVarExp(*this); } }
struct ValExp : BoolExp { void accept(BoolExpVisitor& v) const { v.visitValExp(*this); } }
struct NotExp : BoolExp { void accept(BoolExpVisitor& v) const { v.visitNotExp(*this); } }
struct AndExp : BoolExp { void accept(BoolExpVisitor& v) const { v.visitAndExp(*this); } }
struct OrExp : BoolExp { void accept(BoolExpVisitor& v) const { v.visitOrExp(*this); } }
```

- And you are ready to use it!

Note to self:

- Intrusive
- Cannot be added retroactively
- Specific to class hierarchy!

Example: eval

```
typedef std::map<std::string, bool> Context;
```

```
bool eval(Context& ctx, const BoolExp* exp)
{
```

```
    struct EvalVisitor : BoolExpVisitor
    {
        EvalVisitor(Context& c)
            : m_ctx(c), result(false) {}
```

```
        bool    result;
        Context& m_ctx;
```

```
        void visitVarExp(const VarExp& x) { result = m_ctx[x.name]; }
```

```
        void visitValExp(const ValExp& x) { result = x.value; }
```

```
        void visitNotExp(const NotExp& x) { result = !eval(m_ctx, x.e); }
```

```
        void visitAndExp(const AndExp& x) { result = eval(m_ctx, x.e1) && eval(m_ctx, x.e2); }
```

```
        void visitOrExp (const OrExp & x) { result = eval(m_ctx, x.e1) || eval(m_ctx, x.e2); }
```

```
    } evaluator(ctx);
```

```
    exp->accept(evaluator);
```

```
    return evaluator.result;
```

```
}
```

Note to self:

- Return does not return from eval
- No access to function's arguments
- Both due to control inversion:
 - Don't call us, we call you!

Example: replace

```
BoolExp* replace(BoolExp* where, const char* name, const BoolExp* with)
```

```
{
```

NOTE: non-const argument!

```
    struct ReplaceVisitor : BoolExpVisitor
```

```
    {
```

```
        ReplaceVisitor(const char* n, const BoolExp* w) : name(n), with(w), result(nullptr) {}
```

```
        BoolExp*      result;
```

```
        const char*   name;
```

```
        const BoolExp* with;
```

error C2440: '=' : cannot convert from 'const BoolExp *' to 'BoolExp *'

```
        void visitVarExp(const VarExp& x) { result = x.name == name ? copy(with) : &x; }
```

```
        void visitValExp(const ValExp& x) { result = &x; }
```

```
        void visitNotExp(const NotExp& x) { result = &x; x.e = replace(x.e, name, with); }
```

```
        void visitAndExp(const AndExp& x) { result = &x; x.e1 = replace(x.e1, name, with);  
                                             x.e2 = replace(x.e2, name, with); }
```

```
        void visitOrExp (const OrExp & x) { result = &x; x.e1 = replace(x.e1, name, with);  
                                             x.e2 = replace(x.e2, name, with); }
```

```
    } replacer(name, with);
```

```
    where->accept(replacer);
```

```
    return replacer.result;
```

```
}
```

Note to self:

- const_cast them all?
- Always pass a modifiable reference?

Example: replace

```
BoolExp* replace(BoolExp* where, const char* name, const BoolExp* with)
{
    struct ReplaceVisitor : MutableBoolExpVisitor
    {
        ReplaceVisitor(const char* n, const BoolExp* w) : name(n), with(w), result(nullptr) {}

        BoolExp*      result;
        const char*    name;
        const BoolExp* with;

        void visitVarExp(VarExp& x) { result = x.name == name ? copy(with) : &x; }
        void visitValExp(ValExp& x) { result = &x; }
        void visitNotExp(NotExp& x) { result = &x; x.e = replace(x.e, name, with); }
        void visitAndExp(AndExp& x) { result = &x; x.e1 = replace(x.e1, name, with);
                                     x.e2 = replace(x.e2, name, with); }
        void visitOrExp (OrExp & x) { result = &x; x.e1 = replace(x.e1, name, with);
                                     x.e2 = replace(x.e2, name, with); }
    } replacer(name, with);

    where->accept(replacer);
    return replacer.result;
}
```

Mutable Visitation

```
struct BoolExpVisitor
{
    virtual void visitVarExp(const VarExp&) {}
    virtual void visitValExp(const ValExp&) {}
    virtual void visitNotExp(const NotExp&) {}
    virtual void visitAndExp(const AndExp&) {}
    virtual void visitOrExp (const OrExp &) {}
};
```

```
struct MutableBoolExpVisitor
{
    virtual void visitVarExp(VarExp& x) {}
    virtual void visitValExp(ValExp& x) {}
    virtual void visitNotExp(NotExp& x) {}
    virtual void visitAndExp(AndExp& x) {}
    virtual void visitOrExp (OrExp & x) {}
};
```

```
struct BoolExp
{
    virtual void accept(      BoolExpVisitor&) const = 0; // Read-only introspection
    virtual void accept(MutableBoolExpVisitor&)      = 0; // Mutable visitation
};
```

Note to self:

- Forwarding can also be used to default-implement derived cases via base cases

Binary Methods with Visitors: equal

```
bool equal(const BoolExp* x1, const BoolExp* x2)
{
    struct EqualityVisitor : BoolExpVisitor
    {
        EqualityVisitor(const BoolExp* x2) : x2(x2), result(false) {}

        bool result;
        const BoolExp* x2;

        void visitVarExp(const VarExp& x1) { x2->accept(v); result = v.result; }
        void visitValExp(const ValExp& x1) { x2->accept(v); result = v.result; }
        void visitNotExp(const NotExp& x1) { x2->accept(v); result = v.result; }
        void visitAndExp(const AndExp& x1) { x2->accept(v); result = v.result; }
        void visitOrExp(const OrExp & x1) { x2->accept(v); result = v.result; }
    } equator(x2);

    x1->accept(equator);
    return equator.result;
}
```

Note to self:

- Factor out at least this code:
 - Create visitor
 - Accept it
 - Copy result

Binary Methods with Visitors: equal

```
bool equal(const BoolExp*, const BoolExp*);
```

```
bool eq(const BoolExp& , const BoolExp& ) { return false; }
bool eq(const VarExp& a, const VarExp& b) { return a.name == b.name; }
bool eq(const ValExp& a, const ValExp& b) { return a.value == b.value; }
bool eq(const NotExp& a, const NotExp& b) { return equal(a.e, b.e); }
bool eq(const AndExp& a, const AndExp& b) { return equal(a.e1, b.e1) && equal(a.e2, b.e2); }
bool eq(const OrExp& a, const OrExp& b) { return equal(a.e1, b.e1) && equal(a.e2, b.e2); }
```

```
template <typename Exp>
```

```
struct EqualToVisitor : BoolExpVisitor
{
```

```
    EqualToVisitor(const Exp* x) : x1(x), result(false) {}
```

```
    bool result;
    const Exp* x1;
```

```
    void visit (const Exp& x2) { result = eq(*x1,x2); } // Now generic name would have helped!
    // ... because interesting cases here are only those where both arguments have the same type
```

```
};
```

Note to self:

- 5 vtbl entries
- per each of the 5 instantiations
- plus 5 vtbl entries in EqualityVisitor
- assuming immutable visitation only

Binary Methods with Visitors: match

```
typedef std::map<std::string, const BoolExp*> Assignments;

bool match(const BoolExp*, const BoolExp*, Assignments&);

bool mc(const BoolExp& , const BoolExp& , Assignments& ctx) { return false; }
bool mc(const VarExp& a, const BoolExp& b, Assignments& ctx)
    { if (ctx[a.name] == nullptr) { ctx[a.name] = copy(&b); return true; } else { return equal(ctx[a.name], &b); } }
bool mc(const ValExp& a, const ValExp& b, Assignments& ctx) { return a.value == b.value; }
bool mc(const NotExp& a, const NotExp& b, Assignments& ctx) { return match(a.e, b.e, ctx); }
bool mc(const AndExp& a, const AndExp& b, Assignments& ctx) { return match(a.e1, b.e1, ctx) && match(a.e2, b.e2, ctx); }
bool mc(const OrExp& a, const OrExp& b, Assignments& ctx) { return match(a.e1, b.e1, ctx) && match(a.e2, b.e2, ctx); }

template <typename Exp>
struct MatchToVisitor : BoolExpVisitor
{
    MatchToVisitor(const Exp* p, Assignments& ctx) : m_p(p), m_ctx(ctx), result(false) {}

    bool result;
    const Exp* m_p;
    Assignments& m_ctx;

    void visitVarExp(const VarExp& x) { result = mc(*m_p, x, m_ctx); }
    void visitValExp(const ValExp& x) { result = mc(*m_p, x, m_ctx); }
    void visitNotExp(const NotExp& x) { result = mc(*m_p, x, m_ctx); }
    void visitAndExp(const AndExp& x) { result = mc(*m_p, x, m_ctx); }
    void visitOrExp(const OrExp& x) { result = mc(*m_p, x, m_ctx); }
};

bool match(const BoolExp* p, const BoolExp* x, Assignments& ctx)
{
    struct MatchVisitor : BoolExpVisitor
    {
        MatchVisitor(const BoolExp* x, Assignments& ctx) : x(x), ctx(ctx), result(false) {}

        bool result;
        const BoolExp* x;
        Assignments& ctx;

        void visitVarExp(const VarExp& p) { MatchToVisitor<VarExp> v(&p, ctx); x->accept(v); result = v.result; }
        void visitValExp(const ValExp& p) { MatchToVisitor<ValExp> v(&p, ctx); x->accept(v); result = v.result; }
        void visitNotExp(const NotExp& p) { MatchToVisitor<NotExp> v(&p, ctx); x->accept(v); result = v.result; }
        void visitAndExp(const AndExp& p) { MatchToVisitor<AndExp> v(&p, ctx); x->accept(v); result = v.result; }
        void visitOrExp(const OrExp& p) { MatchToVisitor<OrExp> v(&p, ctx); x->accept(v); result = v.result; }
    } matcher(x, ctx);

    p->accept(matcher);
    return matcher.result;
}
```

All this boilerplate code

← Just to do this tiny case analysis

- This doesn't include VDP declarations
- Specific to both:
 - class hierarchy
 - method
- Attempts for any reuse will
 - further complicate the code
 - make it slower

Visitor Design Pattern: Summary

Pros

- Extensibility of functions
- Speed (open world)
- Library solution

Cons

- Hard to teach
- Intrusive
- Specific to hierarchy
- Lots of boilerplate code
- Control inversion
- Hinders extensibility of classes

- Erich Gamma, Richard Helm, Ralph E. Johnson, and John M. Vlissides. 1993. Design Patterns: Abstraction and Reuse of Object-Oriented Design. In *Proceedings of the 7th European Conference on Object-Oriented Programming (ECOOP '93)*, Oscar Nierstrasz (Ed.). Springer-Verlag, London, UK, UK, 406-431.
- Daniel H. H. Ingalls. “A simple technique for handling multiple polymorphism” OOPSLA '86, pages 347–349, New York, NY, USA, 1986. ACM.

Alternative 1: Pattern Matching

- What is a pattern?
 - a term representing an immediate predicate on an implicit argument
- What is pattern matching?
 - a language feature that provides intuitive laconic syntax and an efficient decision procedure for checking the structure of data and decomposing it into subcomponents
- Examples of patterns
 - Wildcards, Variables, Values, Regular Expressions, Terms of the above, grammars etc.
- Why should I care?
 - Pattern matching has been known in other languages to drastically simplify code, making it more readable, easier to teach and understand, more maintainable and efficient
- When is it useful?
 - Whenever you need to perform an analysis of the structure of data
- We demonstrate it with a syntax of an experimental library
 - So please ignore the quirks of the syntax
 - The actual language feature would have a better one

- A library solution to pattern matching in C++
 - Implemented in standard C++ (mostly 03, but benefits from 11)
- Open to new patterns
 - All patterns are user-definable
- First-class patterns
 - Patterns can be saved in variables and passed to functions
- Type safe
 - Incorrect application is manifested at compile time
- Non-intrusive
 - Can be applied retroactively
 - Works with the existing C++ object model, including multiple inheritance
- Efficient
 - Works on top of an efficient type switch construct
 - Faster than existing alternatives to open pattern matching in C++

Working with *Mach7*

- Declare your variants

```
struct BoolExp          { virtual ~BoolExp() {} };
struct VarExp : BoolExp { std::string name;  };
struct ValExp : BoolExp { bool              value; };
struct NotExp : BoolExp { BoolExp* e; };
struct AndExp : BoolExp { BoolExp* e1; BoolExp* e2; };
struct OrExp  : BoolExp { BoolExp* e1; BoolExp* e2; };
```

Note to myself:

- Non-intrusive!
- Respects member access

- Define bindings (mapping of members to pattern-matching positions)

```
namespace mch { ///Mach7 library namespace
    template <> struct bindings<VarExp> { Members(VarExp::name); };
    template <> struct bindings<ValExp> { Members(ValExp::value); };
    template <> struct bindings<NotExp> { Members(NotExp::e); };
    template <> struct bindings<AndExp> { Members(AndExp::e1, AndExp::e2); };
    template <> struct bindings<OrExp>  { Members( OrExp::e1,  OrExp::e2); };
}
```

- Pick the patterns you'd like to use

```
using mch::C; using mch::var; using mch::_;
```

Example: eval

```
bool eval(Context& ctx, const BoolExp* exp)
{
    var<std::string> name; var<bool> value; var<const BoolExp*> e1, e2;

    Match(exp)
    {
        Case(C<VarExp>(name) )
        Case(C<ValExp>(value))
        Case(C<NotExp>(e1)    )
        Case(C<AndExp>(e1,e2))
        Case(C<OrExp >(e1,e2))
    }
    EndMatch
}
```

Note to self:

- Patterns in the LHS, values in the RHS
- No control inversion!
 - Direct access to arguments
 - Direct return from the function

Example: replace

```
BoolExp* replace(BoolExp* where, const char* what, const BoolExp* with)
{
    var<std::string> name; var<bool> value; var<BoolExp*> e1, e2;

    Match(where)
        Case(C<VarExp>(name) ) return name == what ? copy(with) : &match0;
        Case(C<ValExp>(value)) return &match0;
        Case(C<NotExp>(e1)    ) match0.e = replace(e1, what, with); return &match0;
        Case(C<AndExp>(e1,e2))
            match0.e1 = replace(e1, what, with);
            match0.e2 = replace(e2, what, with);
            return &match0;
        Case(C<OrExp >(e1,e2))
            match0.e1 = replace(e1, what, with);
            match0.e2 = replace(e2, what, with);
            return &match0;
    EndMatch
}
```

Note to self:

- Mutability of match0 is mutability of the subject!
- Forget the fall-through!

Example: match

```
bool match(const BoolExp* p, const BoolExp* x, Assignments& ctx)
{
    var<std::string> name; var<bool> value; var<const BoolExp*> p1, p2, e1, e2;

    Match( p, x
           Case(C<VarExp>(name), _
                ) if (ctx[name] == nullptr) {
                    ctx[name] = copy(x);
                    return true;
                } else
                    return equal(ctx[name],x);
           Case(C<ValExp>(value), C<ValExp>(value)) return true;
           Case(C<NotExp>(p1), C<NotExp>(e1)) return match(p1, e1, ctx);
           Case(C<AndExp>(p1,p2), C<AndExp>(e1,e2)) return match(p1, e1, ctx) && match(p2, e2, ctx);
           Case(C< OrExp>(p1,p2), C< OrExp>(e1,e2)) return match(p1, e1, ctx) && match(p2, e2, ctx);
           Otherwise() return false;
    EndMatch
}
```

Note to self:

- Relational matching!
- Pattern combinators!

Example: Nested Matching

```
BoolExp* dnf(BoolExp* exp)
{
    var<BoolExp*> e, e1, e2;

    Match(exp)
        Case(C<NotExp>(C<NotExp>(e))          ) return e;
        Case(C<AndExp>(e, C<OrExp>(e1,e2))) return new OrExp(new AndExp(e,e1), new AndExp(e,e2));
        Case(C<AndExp>(C<OrExp>(e1,e2), e)) return new OrExp(new AndExp(e1,e), new AndExp(e2,e));
        Otherwise()                          return exp;
    EndMatch
}
```

Note to self:

- Visitors are not directly suitable for nested matching!

What about boost::Variant?

```
void foo(const variant<double,float,int,complex<double>,unsigned int*>& v)
{
    var<double> a, b;
    Match(v)
    {
        Case(C<double>())           ) cout << "double " << match0; break;
        Case(C<float> ()           ) cout << "float  " << match0; break;
        Case(C<int>   ()           ) cout << "int    " << match0; break;
        Case(C<complex<double>>(a,b)) cout << a << '+' << b << 'i'; break;
        Otherwise()                ) cout << "other " << match0; break;
    }
    EndMatch
}
```

Note to self:

- Almost done, check on github soon
- Required generalization of some parts

Pattern Matching: Summary

Pros

- Intuitive, easy to teach and understand
- Direct show of intent
- Relational matching
- Nested matching
- No control inversion
- Local reasoning

Cons

- Not available as a language feature yet
- Can be abused for writing ad-hoc code, where hierarchies and virtual functions should have been normally used

<https://github.com/solodon4/Mach7>

- Y.Solodkyy, G.Dos Reis, B.Stroustrup. ["Open Pattern Matching for C++"](#) In Proceedings of the 12th international conference on Generative programming: concepts & experiences (GPCE '13). ACM, New York, NY, USA, pp. 33-42.
- Y.Solodkyy, G.Dos Reis, B.Stroustrup. ["Open and Efficient Type Switch for C++"](#) In Proceedings of the ACM international conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '12). ACM, New York, NY, USA, pp. 963-982

Alternative 2: Open Multi-Methods

- Multi-Methods + Open Class Extensions
 - Multiple Dispatch
 - The selection of a function to be invoked based on the dynamic type of two or more arguments
 - Open Class Extensions
 - Ability to introduce polymorphic functions outside of their class
- Examples of Open Multi-Methods uses
 - equality, shape intersection, object interactions in games.
- Why should I care?
 - They help retroactively introduce a virtual function into a class without changes to the interface
 - They help implement dynamic dispatch on 2 or more polymorphic arguments: e.g. equal
- When is it useful?
 - Whenever you need to perform an analysis of interaction between 2 or more given objects
- We demonstrate it with the syntax of an experimental implementation

Example: Open Class Extension

```
bool eval(Context& ctx, virtual const BoolExp* x) { return false; }
bool eval(Context& ctx, virtual const VarExp* x) { return ctx[x->name]; }
bool eval(Context& ctx, virtual const ValExp* x) { return x->value; }
bool eval(Context& ctx, virtual const NotExp* x) { return !eval(ctx, x->e); }
bool eval(Context& ctx, virtual const AndExp* x) { return eval(ctx, x->e1) && eval(ctx, x->e2); }
bool eval(Context& ctx, virtual const OrExp * x) { return eval(ctx, x->e1) || eval(ctx, x->e2); }
```

Note to myself:

- No need to foresee all the virtual functions!
- Mix of virtual and non-virtual arguments
- Hard to reason locally about
- Semi-inverted control

Example: Open Multi-Method

```
bool equal(virtual const BoolExp& , virtual const BoolExp& ) { return false; }
bool equal(virtual const VarExp& a, virtual const VarExp& b) { return a.name == b.name; }
bool equal(virtual const ValExp& a, virtual const ValExp& b) { return a.value == b.value; }
bool equal(virtual const NotExp& a, virtual const NotExp& b) { return equal(*a.e, *b.e); }
bool equal(virtual const AndExp& a, virtual const AndExp& b) { return equal(*a.e1, *b.e1)
                                                                && equal(*a.e2, *b.e2); }
bool equal(virtual const OrExp& a, virtual const OrExp& b) { return equal(*a.e1, *b.e1)
                                                                && equal(*a.e2, *b.e2); }
```

Note to myself:

- Subject to ambiguities
- Requires changes to linker and loader
- Works with current C++ object model

Open Multi-Methods

Pros

- Extensibility of functions
- Extensibility of classes
- Speed
- Easy to teach
- Non-intrusive
- General
- Breve
- Relational

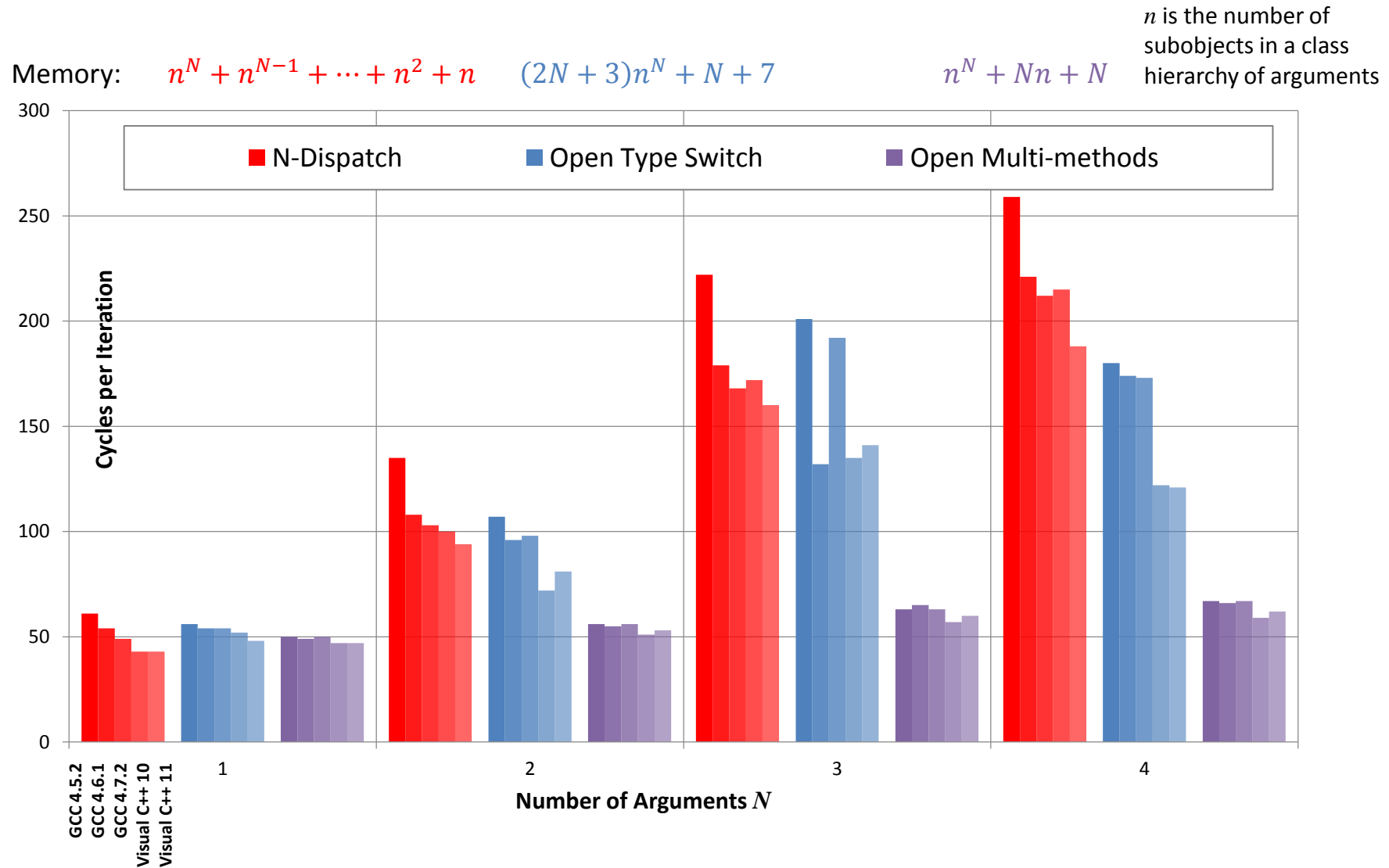
Cons

- Subject to ambiguities
- Requires changes to linker and loader
- Semi-inverted control
- No local reasoning

<https://parasol.tamu.edu/groups/pttlgroup/omm/>

- P.Pirkelbauer, Y.Solodkyy, B.Stroustrup. "[Design and evaluation of C++ open multi-methods](#)", Science of Computer Programming, 2009.
- P.Pirkelbauer, Y.Solodkyy, B.Stroustrup. "[Open multi-methods for C++](#)", In Proceedings of the 6th international conference on Generative Programming and Component Engineering, October 01-03, 2007, Salzburg, Austria

Performance Comparison



Comparison of Possibilities

	Extensibility of Functions	Extensibility of Data	Type Safe	Multiple Inheritance	Relational	Nesting	Retroactive	Local Reasoning	No Control Inversion	Redundancy Checking	Completeness Checking	Speed in Cycles
Virtual Functions	-	+	+	+	-	-	-	-	-		-	29
Visitor Design Pattern	+	*	+	+	*	-	-	+	-		-	55
Open Pattern Matching	+	+	+	+	+	+	+	+	+	*	-	70
Open Multi-methods	+	+	+	+	+	-	+	-	-		+	38

Conclusions

Visitor Design Pattern

- Unnecessarily complicates things
- Extremely hard to teach to novices
- Expert friendly

Open Pattern Matching

- Keeps simple things simple
- Does not sacrifice the performance
- Easy to teach novices
- Also available as a library solution

Open Multi-Methods

- Keeps simple things simple
- Ultimate performance
- Integrates with the rest of the language

Morgan Stanley

Executive Summary


- Keep simple things simple
- Don't make complex things unnecessarily complex
- Don't make things impossible
- Constraint: Don't sacrifice performance
- C++ is expert friendly
 - and becoming more so
- C++ must not be *just* expert friendly
 - Serving novices and “occasional users” is very important
 - Most of the time, *I* don't want to be a language lawyer

Stroustrup - Simple - Cppcon'14

Morgan Stanley

Make Simple Tasks Simple!

Bjarne Stroustrup
Morgan Stanley
Columbia University, Texas A&M University
www.stroustrup.com



Thank You!

Acknowledgements

Abe Skolnik

Bjarne Stroustrup

Gabriel Dos Reis

Jason Wilkins

Michael Lopez

Jasson Casey

Peter Pirkelbauer

Andrew Sutton

Karel Driesen

Emil 'Skeen' Madsen

Visual C++ Team



<https://github.com/solodon4/Mach7>

<http://parasol.tamu.edu/~yuriys/>

<http://parasol.tamu.edu/mach7/>

<https://parasol.tamu.edu/groups/pttlgroup/omm/>

Chicken or Egg: Double Dispatch or Visitor Design Pattern?

```
bool VarExp::equal(const BoolExp* x) const { auto p=dynamic_cast<const VarExp*>(x); return p && name == p->name; }
bool ValExp::equal(const BoolExp* x) const { auto p=dynamic_cast<const ValExp*>(x); return p && value == p->value; }
bool NotExp::equal(const BoolExp* x) const { auto p=dynamic_cast<const NotExp*>(x); return p && e->equal(p->e); }
bool AndExp::equal(const BoolExp* x) const { auto p=dynamic_cast<const AndExp*>(x); return p && e1->equal(p->e1)&&e2->equal(p->e2); }
bool OrExp::equal(const BoolExp* x) const { auto p=dynamic_cast<const OrExp*>(x); return p && e1->equal(p->e1)&&e2->equal(p->e2); }
```

```
struct BoolExp
{ ...
    virtual bool equal_to_VarExp(const VarExp*) const { return false; }
    virtual bool equal_to_ValExp(const ValExp*) const { return false; }
    virtual bool equal_to_NotExp(const NotExp*) const { return false; }
    virtual bool equal_to_AndExp(const AndExp*) const { return false; }
    virtual bool equal_to_OrExp (const OrExp *) const { return false; }
};
```

```
bool VarExp::equal(const BoolExp* x) const { return equal_to_VarExp(this); }
bool ValExp::equal(const BoolExp* x) const { return equal_to_ValExp(this); }
bool NotExp::equal(const BoolExp* x) const { return equal_to_NotExp(this); }
bool AndExp::equal(const BoolExp* x) const { return equal_to_AndExp(this); }
bool OrExp::equal(const BoolExp* x) const { return equal_to_OrExp (this); }
```

Chicken or Egg: Double Dispatch or Visitor Design Pattern?

```
struct BoolExp
{ ...
    virtual bool equal_to_VarExp(const VarExp*) const { return false; }
    virtual bool equal_to_ValExp(const ValExp*) const { return false; }
    virtual bool equal_to_NotExp(const NotExp*) const { return false; }
    virtual bool equal_to_AndExp(const AndExp*) const { return false; }
    virtual bool equal_to_OrExp (const OrExp *) const { return false; }
};
```

```
bool VarExp::equal(const BoolExp* x) const { return equal_to_VarExp(this); }
bool ValExp::equal(const BoolExp* x) const { return equal_to_ValExp(this); }
bool NotExp::equal(const BoolExp* x) const { return equal_to_NotExp(this); }
bool AndExp::equal(const BoolExp* x) const { return equal_to_AndExp(this); }
bool OrExp ::equal(const BoolExp* x) const { return equal_to_OrExp (this); }
```

```
bool VarExp::equal_to_VarExp(const VarExp* p) const { return name == p->name; }
bool ValExp::equal_to_ValExp(const ValExp* p) const { return value == p->value; }
bool NotExp::equal_to_NotExp(const NotExp* p) const { return e->equal(p->e); }
bool AndExp::equal_to_AndExp(const AndExp* p) const { return e1->equal(p->e1) && e2->equal(p->e2); }
bool OrExp ::equal_to_OrExp (const OrExp * p) const { return e1->equal(p->e1) && e2->equal(p->e2); }
```

Double Dispatch

- Doesn't have to be symmetric
 - One type presents its cases to another
- Allows us to uncover dynamic types of 2 arguments
 - Well, not necessarily the actual dynamic type
- Only 2 virtual function calls
 - Hence “double dispatch”

Daniel H. H. Ingalls. *“A simple technique for handling multiple polymorphism”*
OOPLSA '86, pages 347–349, New York, NY, USA, 1986. ACM.

Example: print

```
void print(const BoolExp* exp)
{
    struct PrintVisitor : BoolExpVisitor
    {
        void visitVarExp(const VarExp& x) { std::cout << x.name; }
        void visitValExp(const ValExp& x) { std::cout << x.value; }
        void visitNotExp(const NotExp& x) { std::cout << '!'; print(x.e); }
        void visitAndExp(const AndExp& x) { std::cout << '('; print(x.e1);
                                            std::cout << '&'; print(x.e2);
                                            std::cout << ')'; }
        void visitOrExp (const OrExp & x) { std::cout << '('; print(x.e1);
                                            std::cout << '|'; print(x.e2);
                                            std::cout << ')'; }
    } printer;

    exp->accept(printer);
}
```

Note to self:

- Return does not return from print
- No access to function's arguments

Returning Result

```
BoolExp* copy(const BoolExp* exp)
{
    struct CopyVisitor : BoolExpVisitor
    {
        BoolExp* result;

        void visitVarExp(const VarExp& x) { result = new VarExp(x.name.c_str()); }
        void visitValExp(const ValExp& x) { result = new ValExp(x.value); }
        void visitNotExp(const NotExp& x) { result = new NotExp(copy(x.e)); }
        void visitAndExp(const AndExp& x) { result = new AndExp(copy(x.e1), copy(x.e2)); }
        void visitOrExp (const OrExp & x) { result = new OrExp(copy(x.e1), copy(x.e2)); }
    } copier;

    exp->accept(copier);
    return copier.result;
}
```

Note to myself:

- Can't accept/visit return BoolExp*?
- Parameterized BoolExpVisitor<R>?
- Parameterized BoolExpVisitorImpl<R>

Example: replace

```
BoolExp* replace(const BoolExp* where, const char* what, const BoolExp* with)
{
    struct ReplaceVisitor : BoolExpVisitor
    {
        ReplaceVisitor(const char* n, const BoolExp* w) : name(n), with(w), result(nullptr) {}

        BoolExp*      result;
        const char*    name;
        const BoolExp* with;

        void visitVarExp(const VarExp& x) { result = x.name == name ? copy(with) : copy(&x); }
        void visitValExp(const ValExp& x) { result = copy(&x); }
        void visitNotExp(const NotExp& x) { result = new NotExp(replace(x.e, name, with)); }
        void visitAndExp(const AndExp& x) { result = new AndExp(replace(x.e1, name, with), replace(x.e2, name, with)); }
        void visitOrExp (const OrExp & x) { result = new OrExp(replace(x.e1, name, with), replace(x.e2, name, with)); }
    } replacer(what, with);

    where->accept(replacer);
    return replacer.result;
}
```

Note to myself:

- Bad name: creates a copy of the entire tree with applied replacements

Example: print

```
void print(const BoolExp* exp)
{
    var<std::string> name; var<bool> value; var<const BoolExp*> e1, e2;

    Match(exp)
    {
        Case(C<VarExp>(name) )
        Case(C<ValExp>(value))
        Case(C<NotExp>(e1)    )
        Case(C<AndExp>(e1,e2))

        Case(C<OrExp >(e1,e2))

    }
    EndMatch
}
```

Note to myself:

- Patterns in the LHS
- Values in the RHS