

# Defining Visitors Inline in Modern C++

Robert W. Mill

Jonathan B. Coe

August 10, 2014

The visitor pattern can be useful when type-specific handling is required and tight coupling of type-handling logic and handled types is either an acceptable cost or desirable in its own right. We've found that selective application of the visitor pattern adds strong compile-time safety, as the handling of new types needs explicit consideration in every context where type-specific handling occurs. The visitor pattern presents an inversion of control that can feel unnatural and often requires introduction of considerable non-local boilerplate code. We've found that this slows adoption of the visitor pattern especially among engineers and scientists who traditionally write their type-handling logic inline. Here we present a solution for defining visitors inline.

## The Problem

In object-oriented programming, we may need to perform a function on an object of polymorphic type, such that the behaviour of the function is specific to the derived type. Suppose that for the abstract base class `Polygon` we derive the concrete classes `Triangle` and `Square`. The free function `CountSides`, returns the number of sides in the polygon, `p`.

```
struct Polygon {};  
  
struct Triangle : Polygon  
{  
    // members  
}  
  
struct Square : Polygon  
{  
    // members  
}  
  
int CountSides(Polygon& p)  
{  
    // implementation  
}
```

`CountSides` will need the derived type of the polygon `p` to compute its result, which is problematic, because its argument is conveyed by a reference of the base class type, `Polygon`.

## Visitor pattern

The *visitor* design pattern offers a mechanism for type-specific handling using virtual dispatch. The pattern uses the `this` pointer inside the class to identify the derived type. Each derived object must *accept* a visitor interface which provides a list of `Visit` members with single argument overloaded on various derived types.

To continue our illustration, the `PolygonVisitor` is able to *visit* `Triangles` and `Squares`, and all these polygons must be able to *accept* a `PolygonVisitor`.

```

struct Triangle;
struct Square;

struct PolygonVisitor
{
    virtual ~PolygonVisitor() {}

    virtual void visit(Triangle& tr) = 0;
    virtual void visit(Square& sq) = 0;
};

struct Polygon
{
    virtual void accept(PolygonVisitor& v) = 0;
};

```

**Squares and Triangles accept the visitor as follows. Observe that the `this` pointer is used to select the appropriate overloaded function in the visitor interface.**

```

struct Triangle : Polygon
{
    void accept(PolygonVisitor&v) override
    {
        v.Visit(*this);
    }
};

struct Square : Polygon
{
    void accept(PolygonVisitor& v) override
    {
        v.Visit(*this);
    }
};

```

**A visitor object, `SideCounter`, which counts the number of sides of a polygon and stores the result, is implemented and used as follows.**

```

struct SideCounter : PolygonVisitor
{
    void visit(Square&sq) override
    {
        m_sides = 4;
    }

    void visit(Triangle& tr) override
    {
        m_sides = 3;
    }

    int m_sides = 0;
};

int CountSides(Polygon& p)
{
    SideCounter sideCounter;
    p.Accept(sideCounter);
    return sideCounter.m_sides;
}

```

## Inline visitor pattern

One potential drawback of the visitor pattern is that it requires the creation of a new visitor object type for each algorithm that operates on the derived type. In some cases, the class created will not be reused and, much like a lambda, it would be more convenient to write the visitor clauses inline. The listing below shows how this can be accomplished in a form that resembles a `switch` statement.

```
int CountSides(Polygon& p)
{
    int sides = 0;

    auto v = begin_visitor<PolygonVisitor>
        .on<Triangle>([&sides](Triangle& tr)
        {
            sides = 3;
        })
        .on<Square>([&sides](Square& sq)
        {
            sides = 4;
        })
        .end_visitor();

    p.Accept(v);
    return sides;
}
```

In Listing 1 we demonstrate generic code that permits the `begin_visitor ... end_visitor` construction to be used with any visitor base. The initial `start_visitor` call instantiates a class which defines an inner object inheriting from the visitor interface; each subsequent call of the `on` function instantiates a class whose inner class inherits from the previous inner class implementing an additional `Visit` function. Finally the `end_visitor` call returns an instance of the the inner visitor class.

### Listing 1

```
template <typename T,
          typename F,
          typename BaseInner,
          typename ArgsT>
struct ComposeVisitor
{
    struct Inner : public BaseInner
    {
        using BaseInner::Visit;

        Inner(ArgsT&& args) :
            BaseInner(move(args.second)),
            m_f(move(args.first))
        {
        }

        void Visit(T& t) final override
        {
            m_f(t);
        }

    private:
        F m_f;
    };
};
```

```

ComposeVisitor(ArgsT&& args) :
    m_args(move(args))
{
}

template <typename Tadd,
          typename Fadd>
ComposeVisitor<
    Tadd,
    Fadd,
    Inner,
    pair<Fadd, ArgsT>> on(Fadd&& f)
{
    return ComposeVisitor<
        Tadd,
        Fadd,
        Inner,
        pair<Fadd, ArgsT>> (
            make_pair(
                move(f),
                move(m_args)));
}

Inner end_visitor()
{
    return Inner(move(m_args));
}

ArgsT m_args;
};

template <typename TVisitorBase>
struct EmptyVisitor
{
    struct Inner : public TVisitorBase
    {
        using TVisitorBase::Visit;

        Inner(nullptr_t) {}
    };

    template <typename Tadd, typename Fadd>
    ComposeVisitor<
        Tadd,
        Fadd,
        Inner,
        pair<Fadd, nullptr_t>> on(Fadd&& f)
    {
        return ComposeVisitor<
            Tadd,
            Fadd,
            Inner,
            pair<Fadd, nullptr_t>> (
                make_pair(
                    move(f),
                    nullptr));
    }
};

template <typename TVisitorBase>
EmptyVisitor<TVisitorBase> begin_visitor()
{
    return EmptyVisitor<TVisitorBase>();
}

```

The consistency between the list of types used with `on` and those in the visitor base is verified at compilation time. Since the `override` qualifier is specified on the `Visit` member function, it is not possible to add a superfluous `Visit` which does not correspond to a type overload in the visitor base. Similarly, because the `final` qualifier is specified on the `Visit` member function it is not possible to define a `Visit` member function more than once.

That inline visitors cannot be constructed when clauses are missing may also be considered desirable in some contexts. For instance, if a new type `Hexagon` is derived from `Polygon`, then the code base will compile only when appropriate `Visit` functions been introduced to handle it. In large code bases, this may serve maintainability. If it is deemed that a visitor clause should have some default behaviour (e.g., no operation), a concrete visitor base can be passed into `start_visitor`.

## Performance

With optimizations turned on MSVC 2013, GCC 4.9.1 and Clang 3.4.2 compile the inline visitor without introducing any cost. GCC and Clang produce identical assembler in the case when a visitor class is explicitly written out. MSVC produces different assembler for the inline visitor and explicit visitor class; the inline visitor has been measured to run marginally faster.

Performance and convenience of the inline visitor mean that we would encourage its use where type-specific handling is logically localized.