

# An Inline Visitor Design Pattern for C++11

Robert W. Mill

Jonathan B. Coe

July 22, 2014

Object-oriented programmers often wish to perform a function on an object of polymorphic type, such that the behaviour of the function is specific to the derived type. For instance, suppose we derive from the abstract base class `Polygon` the concrete classes `Triangle` and `Square`. Now suppose that we require a function `CountSides`, which returns the number of sides in the polygon, `p`.

```
struct Polygon { };

struct Triangle : Polygon
{
    // members
}

struct Square : Polygon
{
    // members
}

int CountSides(Polygon& p);
{
    // implementation
}
```

Naturally, the algorithm used by `Angles` to compute its result depends on the derived type of `p`, 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 switching to a code path specific to the derived type. 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’ an `PolygonVisitor`.

```
struct Triangle;
struct Square;

struct PolygonVisitor
```

```
{
    virtual ~PolygonVisitor() {}

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

struct Polygon : 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
    {
        sides = 4;
    }

    void Visit(Triangle& tr) override
    {
        sides = 3;
    }

    int sides;
```

```
};

int CountSides(Polygon& p)
{
    SideCounter sideCounter;
    p.Accept(sideCounter);
    return sideCounter.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 creation of a dedicated class is excessive, and it would be more convenient to write the visitor clauses inline, in a form that resembles a switch statement, as shown in the listing below.

```
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 what follows we demonstrate generic code that permits the `begin_visitor ... end_visitor` construction to be used with any visitor base. The principle behind the pattern is simple. The initial `start_visitor` call returns an object which implements the visitor interface *abstractly*; subsequent calls of the `on` function implement a `Visit` function which takes objects of the template argument type as an argument; and the `end_visitor` call returns the *concrete* visitor object.

### Listing 1

```
template <typename T,
          typename F,
          typename BaseInner,
          typename ArgsT>
class ComposeVisitor
{
public:
    class Inner : public BaseInner
    {
```

```
public:
    using BaseInner::Visit;

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

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

private:
    F m_f;
};

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

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

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

ArgsT m_args;
};

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

        Inner(std::nullptr_t) {}
    };

    template <typename Tadd, typename Fadd>
    ComposeVisitor<
        Tadd,
```

```

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

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

```

The `start_visitor<V>` function returns an object  $t_0 = \text{EmptyVisitor}<V>$  which has the pure visitor interface `V` as a base class, but does not implement any `Visit` members.

Invoking `on<T1>` on a  $t_0$  object returns an object which implements the `Visit(T1&)` member that executes a function object with type `F1`. This object has type  $t_1 =$

```

ComposeVisitor<
    T1,
    F1,
    EmptyVisitor<T>,
    pair<F1, nullptr_t>>

```

which derives from  $t_0$ . (We omit `std::` for brevity.)

Invoking `on<T2>` on a  $t_1$  object returns an object which implements the `Visit(T2&)` member that executes a function object with type `F2`. This object has type  $t_2 =$

```

ComposeVisitor<
    T1,
    F,
    ComposeVisitor<
        T1,
        F,
        EmptyVisitor<T>,
        pair<F1, nullptr_t>>
    pair<F2, <pair<F1, nullptr_t>>

```

which derives from  $t_1$ , and so forth. These nested calls proceed until a type is available which implements all the visitor members.

The pattern is complicated by the fact that no intermediate object returned in the nested series of calls to `on` is allowed to be abstract. As a solution, we construct the visitor type as an *inner class*, and use

`end_visitor` to return an object of this inner class type at the end. Inner class types are allowed remain abstract, provided they are never constructed. If the `end_visitor` function is invoked before the inner visitor class has concrete implementations for all `Visit` members, then the compiler will signal that the programmer has attempt to construct an object with an abstract type; otherwise a visitor is successfully returned.

A few advantages of this approach are notable compile-time. Firstly, because the `override` qualifier is included on the `Visit` member function, it is not possible to include a superfluous clause, which does not correspond to a type overload in the visitor base. Secondly, because the `final` qualifier is included on the `Visit` member function it is not possible to include a visit clause more than once. (If the `final` keyword were omitted, the most recent call to `on` would be observed for a given type.) That inline visitors cannot be constructed when clauses are missing may also be considered desirable.