Object-oriented programmers often wish to perform a function on an object of polymorphic type such that the operation of the function is specific to the derived type. For instance, suppose we derive from the abstract base class `IPolygon` the concrete classes `CSquare` and `CTriangle`. Now suppose that we require a function `Angles`, which returns a vector containing the interior angles of a given polygon, `p`.

```
struct IPolygon { };

struct CSquare : public IPolygon
{
  double m_edge;
}

struct CTriangle : public IPolygon
{
  std::array<double, 3> m_edges;
}

std::vector<double> Angles(const IPolygon& 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, `IPolygon`.

Implement as a member function

The first solution is to implement functions in the derived classes of `CSquare` and `CTriangle`, and make them accessible via a pure virtual function in the interface of `IPolygon`.

```
struct IPolygon
{
  std::vector<double> Angles() const = 0;
};

struct CSquare : public IShape
{
  std::vector<double> Angles() const override
  {
    // implementation
  }

  double m_edge;
};

struct CTriangle : public IShape
{
  std::vector<double> Angles() const override
  {
    // implementation
  }

  std::array<double, 3> m_edges;
};
```

The interior angles of an `IPolygon& p` can now be returned by invoking its member function `Angles`. The principal and well-known benefit of this approach is that any program that introduces a new concrete type of `IPolygon` and neglects to implement its `Angles` interface will fail to compile.

However, if this strategy is adopted in the ongoing development of large code bases, bloated interfaces and implementations are inevitable. Types which start out with a simple, clean implementation will accrue additional member functions every time an algorithm is needed that relies on the derived type. The list of member functions in our example class `IPolygon` could grow indefinitely long, to include `Area`, `Perimeter`, `Vertices`, `IsSymmetric`, and so on.

This proliferation of members is compounded by two further disadvantages. Firstly, in some cases, the inclusion of a member function is inappropriate. For instance, consider the function `CalculateCost`, which returns the cost associated with the automatated machine-cutting of a shape. `CalculateCost` is not esssential to polygons *per se*, in the way that `Area` and `Perimeter` are. Secondly, even if `CalculateCost` were included in the interface, it is possible that at some later stage the developer may wish to provide several switchable implementations, necessitating awkward dependency injections (at best), or additional flags and switches in the argument list of `CalculateCost` (at worst).

At this juncture it should be clear that we require a method for defining functions external to the derived class definitions. Before turning to the visitor pattern, we consider the pros and cons of run-time type information (RTTI), which accomplishes precisely this.

Run-time type information

Run-time type information allows the derived type of an object to be identified during a program's execution.

The visitor pattern

to do

The inline visitor pattern

to do