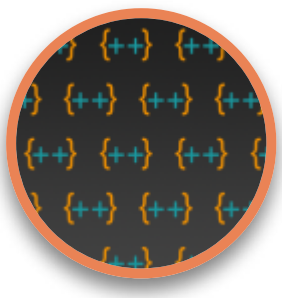


Runtime Polymorphism Without Objects or Virtual Functions

Published May 15, 2020

When thinking of polymorphism, and in particular of runtime polymorphism, the first thing that comes to mind is virtual functions.

Virtual functions are very powerful, and fit for some use cases. But before using them, it's a good thing to consider our exact need for polymorphism, and look around if there are other, more adapted tools to satisfy it.

Indeed, virtual functions create polymorphism on objects. But what if you don't need objects? What if you only need you code to behave differently depending on some conditions, but you don't need any objects involved?

In this case we can use something else that virtual functions.

Motivating example: choosing the right calculator

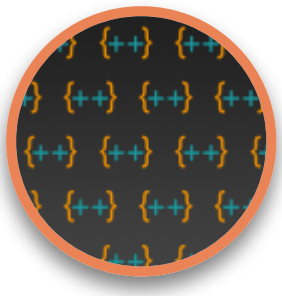
Consider the following example, which is inspired from a project I worked on. I simplified the example by stripping off everything domain related to make it easier to understand.

We have an input, and we'd like to compute an output (this is a pretty standardised example, right?). The input value looks like this:

```
struct Input
{
    double value;
};
```

And the output value looks like that:

```
    double value;
};
```



To compute the `Output` based on the `Input`, we use a calculator.

There are various types of calculators, that are designed to handle various types of inputs. To make the example simple but without losing any of its generality, let's say that there are two calculators: one that handles big inputs (with a `value` larger than 10) and one that handles small inputs (with a `value` smaller or equal to 10).

Moreover, each calculator can log some information about a given pair of input and output.

We'd like to write code that, given an `Input`,

- determines what calculator will handle it,
- launches the calculation to produce an `Output`,
- and invokes the logging of that calculator for the `Input` and the `Output`.

Implementing polymorphism

Given the above needs, we would need some interface to represent a Calculator, with the three following functions:

```
bool handles(Input const& input);

Output compute(Input const& input);

void log(Input const& input, Output const& output);
```

Those three functions define a calculator.

It would be nice to group those three functions in the same place, for example a class. But we don't need them to be member functions, they can be just regular functions. If we use a class to stick them together, we can implement them as `static` functions.

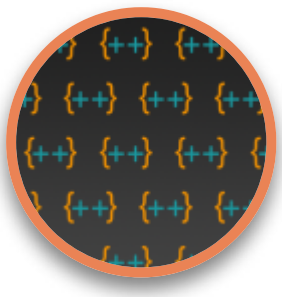
Here is then our calculator that handles big values:

```
struct BigCalculator
{
   static bool handles(Input const& input)
   {
      return input.value > 10;
   }

   static Output compute(Input const& input)
   {
      return Output{ input.value * 5 };
   }

   static void log(Input const& input, Output const& output)
   {
       std::cout << "BigCalculator took an input of " << input.value << " and produced an
   }
};
```

And this is the one that handles small values:




```
   static bool handles(Input const& input)
   {
      return input.value <= 10;
   }

   static Output compute(Input const& input)
   {
      return Output{ input.value + 2 };
   }

   static void log(Input const& input, Output const& output)
   {
       std::cout << "SmallCalculator took an input of " << input.value << " and produced a
   }
};
```

`BigCalculator` and `SmallCalculator` are two implementations of the "Calculator" interface.

Binding the implementations with the call site

Now that we have various implementations of the Calculator interface, we need to somehow bind them to a call site, in a uniform manner.

This means that the code of a given call site should be independent of the particular calculator that it uses. This is by definition what polymorphism achieves.

So far, the "Calculator" interface was implicit. Let's now create a component that embodies a Calculator, and that can behave either like a `SmallCalculator` or a `BigCalculator`.

This component must have the three functions of the Calculator interface, and execute the code of either `BigCalculator` or `SmallCalculator`. Let's add three functions pointers, that we will later assign the the `static` functions of the calculator implementations:

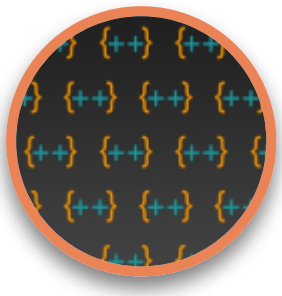
```
struct Calculator
{
   bool (*handles) (Input const& input);
   Output (*compute)(Input const& input);
   void (*log)(Input const& input, Output const& output);
};
```

To make the binding with a calculator implementation easier, let's add a helper function that assigns those function pointers to the one of a calculator:

```
struct Calculator
{
   bool (*handles) (Input const& input);
   Output (*compute)(Input const& input);
   void (*log)(Input const& input, Output const& output);

   template<typename CalculatorImplementation>
   static Calculator createFrom()
   {
      return Calculator{ &CalculatorImplementation::handles, &CalculatorImplementation::co
   }
};
```

This function is a bit like a constructor, but instead of taking values like a normal constructor, it takes a type as input.



To solve our initial problem of choosing the right calculator amongst several ones, let's instantiate and store the calculators in a collection. To do that, we'll have a collection of `Calculators` that we bind to either `BigCalculator` or `SmallCalculator`:

```
std::vector<Calculator> getCalculators()
{
   return {
        Calculator::createFrom<BigCalculator>(),
        Calculator::createFrom<SmallCalculator>()
        };
}
```

We now have a collection of calculator at the ready.

Using the calculator in polymorphic code

We can now write code that uses the `Calculator` interface, and that is independent from the individual types of calculators:

```
auto const input = Input{ 50 };

auto const calculators = getCalculators();
auto const calculator = std::find_if(begin(calculators), end(calculators),
                [&input](auto&& calculator){ return calculator.handles(input); });

if (calculator != end(calculators))
{
    auto const output = calculator->compute(input);
    calculator->log(input, output);
}
```

This code prints the following output (run the code yourself here):

```
BigCalculator took an input of 50 and produced an output of 250
```

And if we replace the first line by the following, to take a small input:

```
SmallCalculator took an input of 5 and produced an output of 7
```

We see that the code picks the correct calculator and uses it to perform the calculation and the logging.

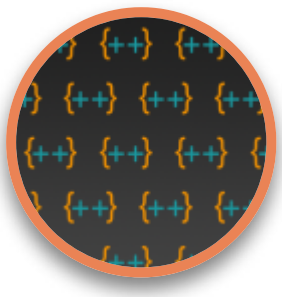
Didn't we reimplement virtual functions?

The above code doesn't contain inheritance nor the keyword `virtual`. But it uses function pointers to route the execution to an implementation in a given class, and that sounds a lot like what virtual functions and vtables do.

Did we just manually implement virtual functions? In this case, we'd be better off using the native feature of the language rather than implementing our own.

The problem we're trying to solve is indeed implementable with virtual functions. Here is the code to do this, with highlight on the significant differences with our previous code:

```
struct Input
{
    double value;
};
```




```
    double value;
};

struct Calculator
{
    virtual bool handles(Input const& input) const = 0; // virtual methods
    virtual Output compute(Input const& input) const = 0;
    virtual void log(Input const& input, Output const& output) const = 0;
    virtual ~Calculator() {};
};

struct BigCalculator : Calculator // inheritance
{
   bool handles(Input const& input) const override
   {
      return input.value > 10;
   }

   Output compute(Input const& input) const override
   {
      return Output{ input.value * 5 };
   }

   void log(Input const& input, Output const& output) const override
   {
       std::cout << "BigCalculator took an input of " << input.value << " and produced an
   }
};

struct SmallCalculator : Calculator
{
   bool handles(Input const& input) const override
   {
      return input.value <= 10;
   }

   Output compute(Input const& input) const override
   {
      return Output{ input.value + 2 };
   }

   void log(Input const& input, Output const& output) const override
   {
       std::cout << "SmallCalculator took an input of " << input.value << " and produced a
   }
};

std::vector<std::unique_ptr<Calculator>> getCalculators() // unique_ptrs
{
   auto calculators = std::vector<std::unique_ptr<Calculator>>{};
   calculators.push_back(std::make_unique<BigCalculator>());
   calculators.push_back(std::make_unique<SmallCalculator>());
   return calculators;
}

int main()
{
    auto const input = Input{ 50 };

    auto const calculators = getCalculators();
```

```
    if (calculator != end(calculators))
    {
        auto const output = (*calculator)->compute(input); // extra indirection
        (*calculator)->log(input, output);
    }
}
```

There are a few notable differences with our previous code that didn't use virtual functions:

- there is now inheritance,
- calculators are now represented as pointers,
- calculators are now allocated on the heap with `new` (in the `std::unique_ptrs`).

The structural difference between the two approaches is that the first one was using polymorphism on classes, or on code, whereas the one with virtual functions uses polymorphism on objects.

As a result, polymorphic objects are instantiated on the heap, in order to store them in a container. With polymorphism on classes, we didn't instantiate any object on the heap.

Which code is better?

Using `new` (and `delete`) can be a problem, especially for performance. Some applications are even forbidden to use heap storage for this reason.

However, if your system allows the use of `new`, it is preferable to write expressive code and optimise it only where necessary. And maybe in this part of the code calling `new` doesn't make a significant difference.

Which solution has the most expressive code then?

Our first code using polymorphism on classes has a drawback in terms of expressiveness: it uses a non-standard construct, with the `Calculator` interface handling function pointers. Virtual functions, on the other hand, use only standard features that hide all this binding, and gives less code to read.

On the other hand, virtual functions don't express our intention as precisely as polymorphism on classes does: calculators are not objects, they are functions. The solution using polymorphism with class demonstrates this, by using `static` functions instead of object methods.

In summary when it comes to expressiveness, there are pros and cons for both solutions. When it comes to the usage of `new`, one solution uses `new` and one doesn't.

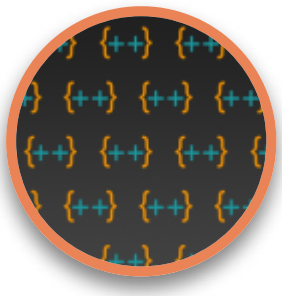
What do you think about those two solutions?

In any case, it is important to keep in mind that virtual functions are powerful as they allow polymorphism at object level, but they come at a cost: instantiation on the heap, and using pointers.

When you need polymorphism, don't rush on virtual functions. Don't rush on any design, for that matter. Think first about what you need. There may be other solutions that will match your need better.

Any feedback is appreciated.

- How to Pass a Polymorphic Object to an STL Algorithm
- The “Extract Interface” refactoring, at compile time
- FSeam: A Mocking Framework That Requires No Change In Code
- Write Your Own Dependency-Injection Container



Jonathan Boccara's blog

RECENT POSTS