Introduction

One of the most powerful language features that we get with C++17 is the compile-time if in the form of `if constexpr`. It allows you to check, at compile time, a condition and depending on the result the code is rejected from the further steps of the compilation.

In this chapter, you'll see one example of how this new feature can simplify the code.

WE'LL COVER THE FOLLOWING

The Problem

In item 18 of Effective Modern C++ Scott Meyers described a method called `makeInvestment`:

```
template<typename... Ts>
std::unique_ptr<Investment>
makeInvestment(Ts&&... params);
```

There's a factory method that creates derived classes of `Investment`, and the main advantage is that it supports a variable number of arguments!

For example, here are the proposed derived types:

```
// base type:
class Investment
{
public:
    virtual ~Investment() { }
    virtual void calcRisk() = 0;
};
class Stock : public Investment
{
public:
    explicit Stock(const std::string&) { }
    void calcRisk() override { }
};
```

```
class Bond : public Investment
{
public:
    explicit Bond(const std::string&, const std::string&, int) { }
    void calcRisk() override { }
};
class RealEstate : public Investment
{
public:
    explicit RealEstate(const std::string&, double, int) { }
    void calcRisk() override { }
};
```

The code from the book was too idealistic, and it worked until all your classes have the same number and types of input parameters:

> Scott Meyers: Modification History and Errata List for Effective Modern C++:

The `makeInvestment` interface is unrealistic because it implies that all derived object types can be created from the same types of arguments. This is especially apparent in the sample implementation code, where arguments are perfect-forwarded to all derived class constructors.

For example, if you had a constructor that needed two arguments and one constructor with three arguments, then the code might not compile:

```
// pseudo code:
Bond(int, int, int) { }
Stock(double, double) { }
make(args...)
{
  if (bond)
     new Bond(args...);
  else if (stock)
     new Stock(args...)
}
```

Now, if you write `make(bond, 1, 2, 3)` - then the `else` statement won't compile - as there no `Stock(1, 2, 3)` available! To make it work, we need a compile time if statement that rejects parts of the code that don't match a condition.

On my blog, with the help of one reader, we proposed one working solution

On my blog, with the help of one reader, we proposed one working solution (you can read more in Bartek's coding blog: Nice C++ Factory Implementation 2).

Code that could work

```
template <typename... Ts>
unique_ptr<Investment>
makeInvestment(const string &name, Ts&&... params)
{
    unique_ptr<Investment> pInv;
    if (name == "Stock")
        pInv = constructArgs<Stock, Ts...>(forward<Ts>(params)...);
        cout <<pInv <<endl;
    else if (name == "Bond")
        pInv = constructArgs<Bond, Ts...>(forward<Ts>(params)...);
    else if (name == "RealEstate")
        pInv = constructArgs<RealEstate, Ts...>(forward<Ts>(params)...);
// call additional methods to init pInv...
    return pInv;
}
```

The "magic" happens inside `constructArgs` function.

As you can see the main idea is to return `unique_ptr<Type>` when `Type` is constructible from a given set of attributes and `nullptr` when it's not.