Duck Typing, the C++ Way

How Type Erasure Bends the Rules

SARTHAK SEHGAL

Sarthak Sehgal

Quoting Tech Lead @ Maven Securities

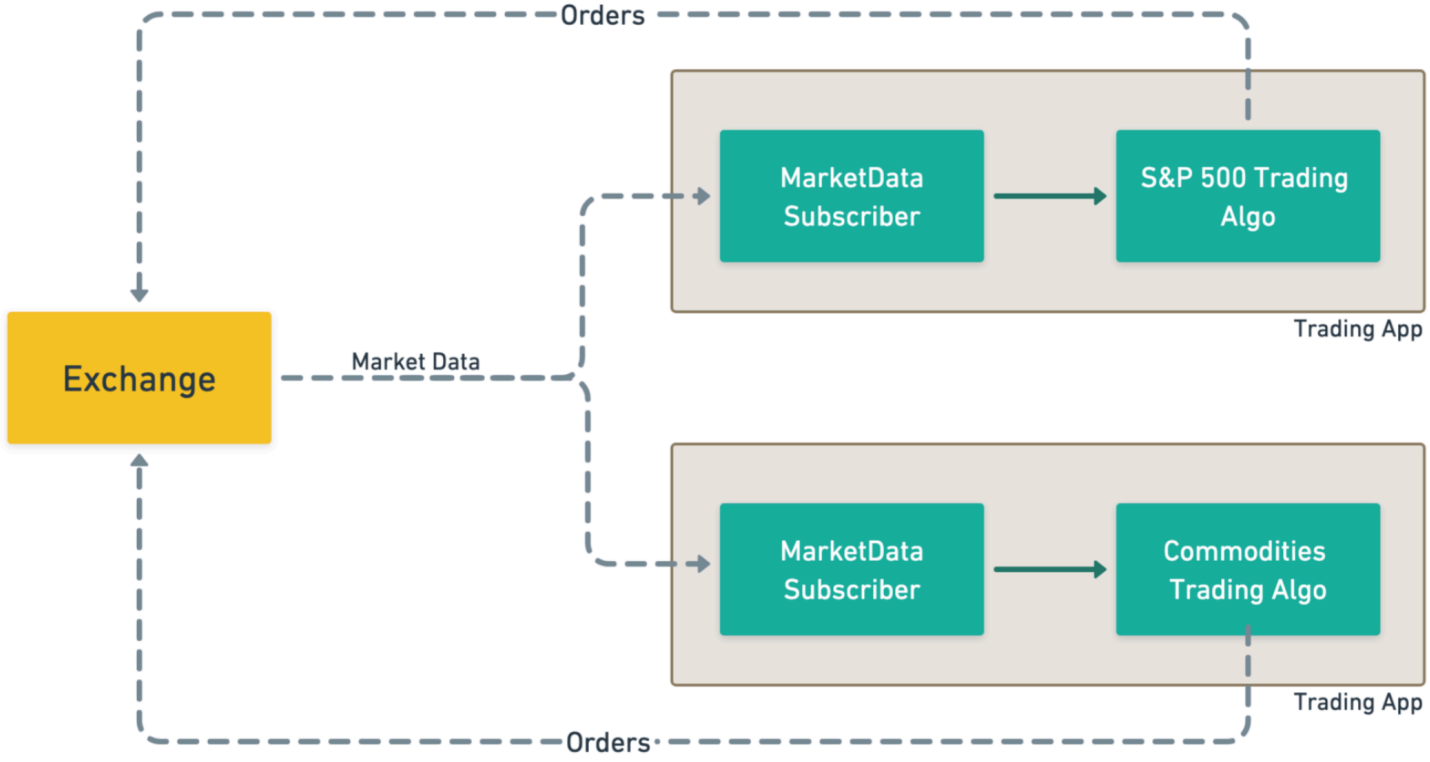
- Low latency C++ Software Engineer
- First talk at CppCon!
- Sometimes, I write about C++ at sartech.substack.com

Maven

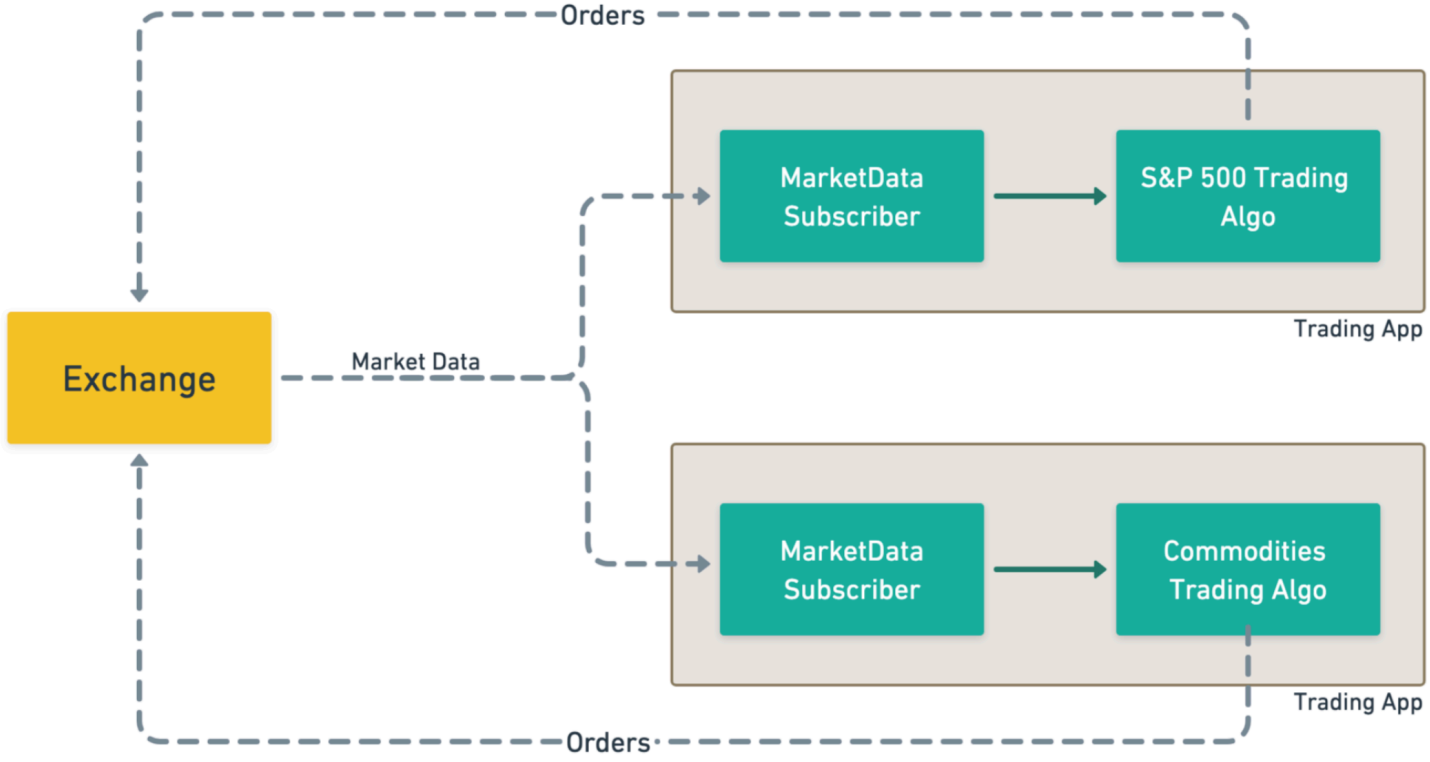
Chicago | London | Amsterdam

OVERVIEW

- Example of type erasure - why std::function
- Building our type-erased std::function
- Cost of type erased function container
- std::any - a modern void*



MarketDataSubscriber listens to a market data feed, and processes instrument definitions



MarketDataSubscriber listens to a market data feed, and processes instrument definitions

Common class that allows filtering of instruments

```
class MarketDataSubscriber
{
  MarketDataSubscriber(?? instrumentFilterFunc)
  : instrumentFilterFunc{std::move(instrumentFilterFunc)}
  {
    listenToMarketDataFeed([this](InstrumentDefinitionType const& def) {
      this->onInstrumentDefinition(def);
    });
  }

  void onInstrumentDefinition(InstrumentDefinitionType const& def)
  {
    if (not instrumentFilterFunc(def))
      return;

    // process
  }

private:
  ?? instrumentFilterFunc;
```

```
{
  MarketDataSubscriber(?? instrumentFilterFunc)
  : instrumentFilterFunc{std::move(instrumentFilterFunc)}
  {
    listenToMarketDataFeed([this](InstrumentDefinitionType const& def) {
      this->onInstrumentDefinition(def);
    });
  }

  void onInstrumentDefinition(InstrumentDefinitionType const& def)
  {
    if (not instrumentFilterFunc(def))
      return;

    // process
  }

private:
  ?? instrumentFilterFunc;
};
```

```cpp
bool isEquity(InstrumentDefinitionType const& def) { return def.isEquity(); }
MarketDataSubscriber equityMdSub(isEquity); // free function

MarketDataSubscriber futureMdSub([](InstrumentDefinitionType const& def)
{
  return def.isFuture();
}); // lambda

struct IsBondFunctor
{
  bool operator()(InstrumentDefinitionType const& def) const
  {
    return def.isBond();
  }
};
MarketDataSubscriber bondMdSub(IsBondFunctor()); // functor
```

Client should be able to pass any callable with matching signature

```cpp
bool isEquity(InstrumentDefinitionType const& def) { return def.isEquity(); }
MarketDataSubscriber equityMdSub(isEquity); // free function

MarketDataSubscriber futureMdSub([](InstrumentDefinitionType const& def)
{
  return def.isFuture();
}); // lambda

struct IsBondFunctor
{
  bool operator()(InstrumentDefinitionType const& def) const
  {
    return def.isBond();
  }
};
MarketDataSubscriber bondMdSub(IsBondFunctor()); // functor
```

Client should be able to pass any callable with matching signature

```cpp
bool isEquity(InstrumentDefinitionType const& def) { return def.isEquity(); }
MarketDataSubscriber equityMdSub(isEquity); // free function

MarketDataSubscriber futureMdSub([](InstrumentDefinitionType const& def)
{
  return def.isFuture();
}); // lambda

struct IsBondFunctor
{
  bool operator()(InstrumentDefinitionType const& def) const
  {
    return def.isBond();
  }
};
MarketDataSubscriber bondMdSub(IsBondFunctor()); // functor
```

Client should be able to pass any callable with matching signature

USING FUNCTION POINTER

```
1  class MarketDataSubscriber
2  {
3  public:
4    MarketDataSubscriber(bool (*instrumentFilter)(InstrumentDefinitionType const&))
5    : instrumentFilterFunc{instrumentFilter}
6    {}
7
8    void onInstrumentDefinition(InstrumentDefinitionType const& def)
9    {
10     if (not (*instrumentFilterFunc)(def))
11       return;
12
13     // process
14   }
15
16 private:
17   bool (*instrumentFilterFunc)(InstrumentDefinitionType const&);
18 };
```

USING FUNCTION POINTER

```
1  class MarketDataSubscriber
2  {
3  public:
4    MarketDataSubscriber(bool (*instrumentFilter)(InstrumentDefinitionType const&))
5    : instrumentFilterFunc{instrumentFilter}
6    {}
7
8    void onInstrumentDefinition(InstrumentDefinitionType const& def)
9    {
10     if (not (*instrumentFilterFunc)(def))
11       return;
12
13     // process
14   }
15
16 private:
17   bool (*instrumentFilterFunc)(InstrumentDefinitionType const&);
18 };
```

- Can pass free function, captureless lambda, static member function

- Can pass free function, captureless lambda, static member function
- Callables need to be stateless*

- Can pass free function, captureless lambda, static member function
- Callables need to be stateless*
- Pointers, and lack of value semantics

- Can pass free function, captureless lambda, static member function
- Callables need to be stateless*
- Pointers, and lack of value semantics
- Need to perform null checks*

- Can pass free function, captureless lambda, static member function
- Callables need to be stateless*
- Pointers, and lack of value semantics
- Need to perform null checks*

** C++26 introduces std::function_ref*

USING INHERITANCE

```
1  class InstrumentDefinitionCallable
2  {
3  public:
4    virtual bool operator()(InstrumentDefinitionType const&) const = 0;
5    virtual ~InstrumentDefinitionCallable() = default;
6  };
7
8  class EquityFilter : public InstrumentDefinitionCallable
9  {
10 public:
11   bool operator()(InstrumentDefinitionType const& idef) const override
12   { ... }
13 };
14
15 class BondFilter : public InstrumentDefinitionCallable
16 {
17 public:
18   bool operator()(InstrumentDefinitionType const& idef) const override
19   { ... }
20 };
```

USING INHERITANCE

```
1  class InstrumentDefinitionCallable
2  {
3  public:
4    virtual bool operator()(InstrumentDefinitionType const&) const = 0;
5    virtual ~InstrumentDefinitionCallable() = default;
6  };
7
8  class EquityFilter : public InstrumentDefinitionCallable
9  {
10 public:
11   bool operator()(InstrumentDefinitionType const& idef) const override
12   { ... }
13 };
14
15 class BondFilter : public InstrumentDefinitionCallable
16 {
17 public:
18   bool operator()(InstrumentDefinitionType const& idef) const override
19   { ... }
20 };
```

USING INHERITANCE

```
  virtual bool operator()(InstrumentDefinitionType const&) const = 0;
  virtual ~InstrumentDefinitionCallable() = default;
};

class EquityFilter : public InstrumentDefinitionCallable
{
public:
  bool operator()(InstrumentDefinitionType const& idef) const override
  { ... }
};

class BondFilter : public InstrumentDefinitionCallable
{
public:
  bool operator()(InstrumentDefinitionType const& idef) const override
  { ... }
};

class MarketDataSubscriber
{
public:
```

USING INHERITANCE

```
22 class MarketDataSubscriber
23 {
24 public:
25   MarketDataSubscriber(std::unique_ptr<InstrumentDefinitionCallable> instrumentFilter
26   : instrumentFilterFunc{std::move(instrumentFilter)}
27   {}
28
29   void onInstrumentDefinition(InstrumentDefinitionType const& def)
30   {
31     if (not instrumentFilterFunc->operator()(def))
32       return;
33
34     // process
35   }
36
37 private:
38   std::unique_ptr<InstrumentDefinitionCallable> instrumentFilterFunc;
39 };
40
41 MarketDataSubscriber equityMdSub(std::make_unique<EquityFilter>());
```

USING INHERITANCE

```
};
public:
  MarketDataSubscriber(std::unique_ptr<InstrumentDefinitionCallable> instrumentFilter
  : instrumentFilterFunc{std::move(instrumentFilter)}
  {}

  void onInstrumentDefinition(InstrumentDefinitionType const& def)
  {
    if (not instrumentFilterFunc->operator()(def))
      return;

    // process
  }

private:
  std::unique_ptr<InstrumentDefinitionCallable> instrumentFilterFunc;
};

MarketDataSubscriber equityMdSub(std::make_unique<EquityFilter>());
MarketDataSubscriber bondMdSub(std::make_unique<BondFilter>());
```

- Single concrete type holding different implementations

- Single concrete type holding different implementations
- Runtime polymorphism, can reassign the filter function

- Single concrete type holding different implementations
- Runtime polymorphism, can reassign the filter function
- All callables need to inherit the interface

- Single concrete type holding different implementations
- Runtime polymorphism, can reassign the filter function
- All callables need to inherit the interface
- Non-zero overhead at runtime

- Single concrete type holding different implementations
- Runtime polymorphism, can reassign the filter function
- All callables need to inherit the interface
- Non-zero overhead at runtime
- Not possible to pass third-party defined callables

- Single concrete type holding different implementations
- Runtime polymorphism, can reassign the filter function
- All callables need to inherit the interface
- Non-zero overhead at runtime
- Not possible to pass third-party defined callables
- Pointers, lifetime management, ...

USING TEMPLATED TYPE

```
1  template <typename T>
2    requires (std::is_invocable_r_v<bool, T, InstrumentDefinitionType const&>)
3  class MarketDataSubscriber
4  {
5  public:
6    MarketDataSubscriber(T instrumentFilter)
7    : instrumentFilterFunc{std::move(instrumentFilter)}
8    {}
9
10   void onInstrumentDefinition(InstrumentDefinitionType const& def)
11   {
12     if (not instrumentFilterFunc(def))
13       return;
14
15     // process
16   }
17
18 private:
19   T instrumentFilterFunc;
20 };
```

USING TEMPLATED TYPE

```
1  template <typename T>
2    requires (std::is_invocable_r_v<bool, T, InstrumentDefinitionType const&>)
3  class MarketDataSubscriber
4  {
5  public:
6    MarketDataSubscriber(T instrumentFilter)
7    : instrumentFilterFunc{std::move(instrumentFilter)}
8    {}
9
10   void onInstrumentDefinition(InstrumentDefinitionType const& def)
11   {
12     if (not instrumentFilterFunc(def))
13       return;
14
15     // process
16   }
17
18 private:
19   T instrumentFilterFunc;
20 };
```

- Accepts any suitable callable at construction

- Accepts any suitable callable at construction
- Zero overhead at runtime

- Accepts any suitable callable at construction
- Zero overhead at runtime
- Code bloat

- Accepts any suitable callable at construction
- Zero overhead at runtime
- Code bloat
- Creates distinct types, not easily stored in a container

std::function for the rescue

```
1  MarketDataSubscriber(
2    std::function<bool(InstrumentDefinitionType const&)> instrumentFilter)
3  {}
4
5  bool isEquity(InstrumentDefinitionType const& def) { return def.isEquity(); }
6  MarketDataSubscriber equityMdSub(isEquity); // free function
7
8  MarketDataSubscriber futureMdSub([](InstrumentDefinitionType const& def)
9  {
10   return def.isFuture();
11 }); // lambda
12
13 struct IsBondFunctor
14 {
15   bool operator()(InstrumentDefinitionType const& def) const
16   {
17     return def.isBond();
18   }
19 };
20 MarketDataSubscriber bondMdSub(IsBondFunctor()); // functor
```

std::function for the rescue

```
1  MarketDataSubscriber(
2    std::function<bool(InstrumentDefinitionType const&)> instrumentFilter)
3  {}
4
5  bool isEquity(InstrumentDefinitionType const& def) { return def.isEquity(); }
6  MarketDataSubscriber equityMdSub(isEquity); // free function
7
8  MarketDataSubscriber futureMdSub([](InstrumentDefinitionType const& def)
9  {
10   return def.isFuture();
11 }); // lambda
12
13 struct IsBondFunctor
14 {
15   bool operator()(InstrumentDefinitionType const& def) const
16   {
17     return def.isBond();
18   }
19 };
20 MarketDataSubscriber bondMdSub(IsBondFunctor()); // functor
```

std::function for the rescue

```
1  MarketDataSubscriber(
2    std::function<bool(InstrumentDefinitionType const&)> instrumentFilter)
3  {}
4
5  bool isEquity(InstrumentDefinitionType const& def) { return def.isEquity(); }
6  MarketDataSubscriber equityMdSub(isEquity); // free function
7
8  MarketDataSubscriber futureMdSub([](InstrumentDefinitionType const& def)
9  {
10   return def.isFuture();
11 }); // lambda
12
13 struct IsBondFunctor
14 {
15   bool operator()(InstrumentDefinitionType const& def) const
16   {
17     return def.isBond();
18   }
19 };
20 MarketDataSubscriber bondMdSub(IsBondFunctor()); // functor
```

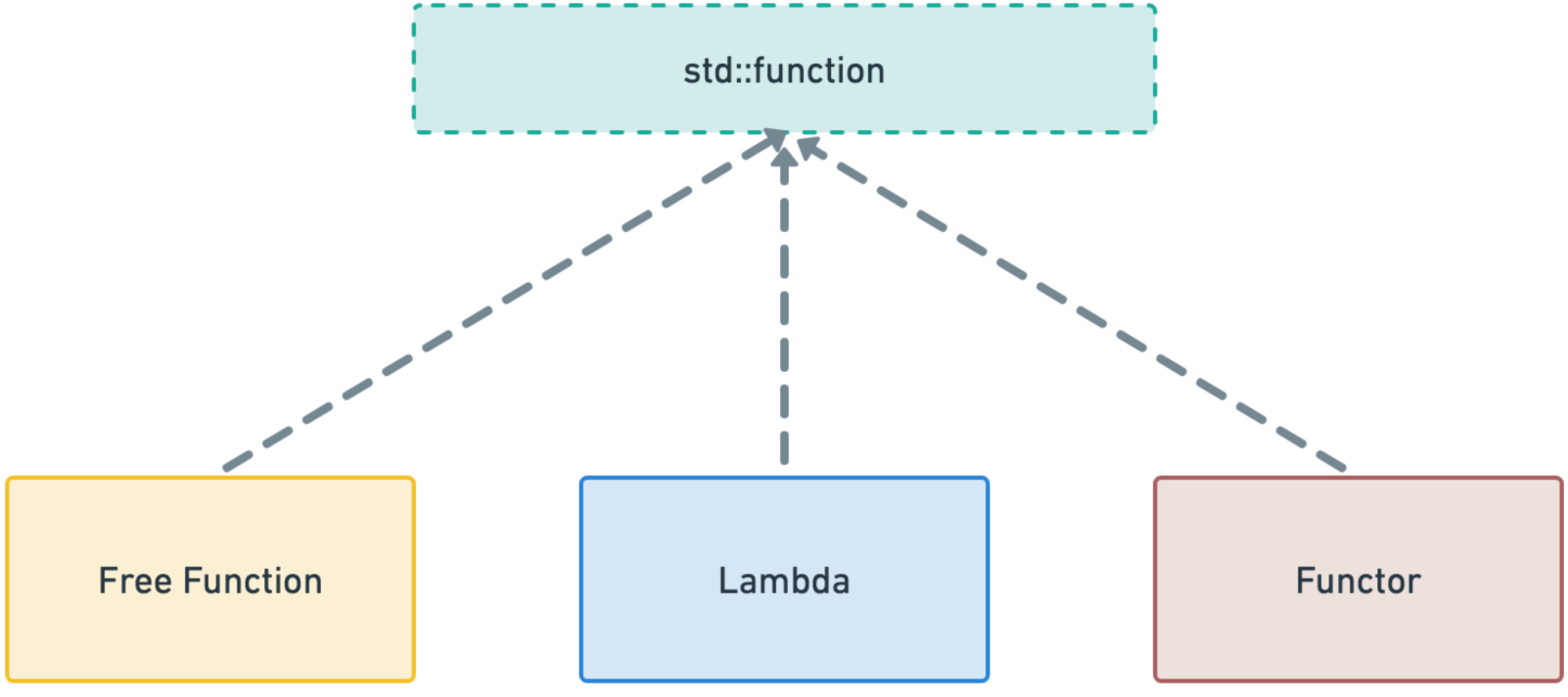
std::function for the rescue

```
1  MarketDataSubscriber(
2    std::function<bool(InstrumentDefinitionType const&)> instrumentFilter)
3  {}
4
5  bool isEquity(InstrumentDefinitionType const& def) { return def.isEquity(); }
6  MarketDataSubscriber equityMdSub(isEquity); // free function
7
8  MarketDataSubscriber futureMdSub([](InstrumentDefinitionType const& def)
9  {
10   return def.isFuture();
11 }); // lambda
12
13 struct IsBondFunctor
14 {
15   bool operator()(InstrumentDefinitionType const& def) const
16   {
17     return def.isBond();
18   }
19 };
20 MarketDataSubscriber bondMdSub(IsBondFunctor()); // functor
```

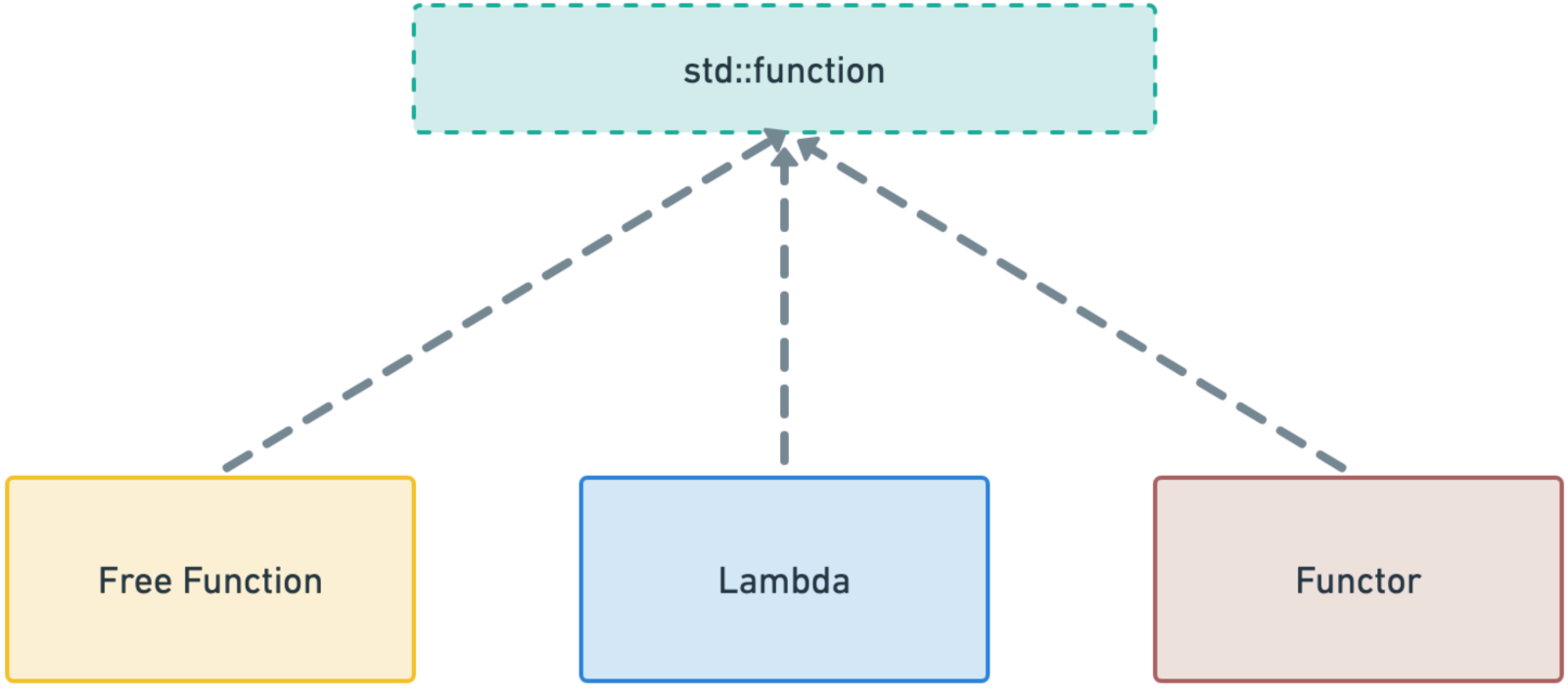
std::function for the rescue

```
1  MarketDataSubscriber(
2    std::function<bool(InstrumentDefinitionType const&)> instrumentFilter)
3  {}
4
5  bool isEquity(InstrumentDefinitionType const& def) { return def.isEquity(); }
6  MarketDataSubscriber equityMdSub(isEquity); // free function
7
8  MarketDataSubscriber futureMdSub([](InstrumentDefinitionType const& def)
9  {
10   return def.isFuture();
11 }); // lambda
12
13 struct IsBondFunctor
14 {
15   bool operator()(InstrumentDefinitionType const& def) const
16   {
17     return def.isBond();
18   }
19 };
20 MarketDataSubscriber bondMdSub(IsBondFunctor()); // functor
```

* C++26 introduces std::copyable_function (P2548R5)



Distinct concrete types with common interface assigned to a single concrete type



Distinct concrete types with common interface assigned to a single concrete type

"If it walks like a duck and quacks like a duck, then it must be a duck" 🦆 🦆

THE BIG QUESTION

How can unrelated types - *free function*, *lambda expression*, and a *functor* - be assigned to a common type `std::function<bool(InstrumentDef const&)>`

REQUIREMENTS

```
namespace cppcon {

  struct Function {};

} // namespace cppcon
```

- Constructible from unrelated types
- Common callable interface

PROS AND CONS OF INHERITANCE

- Single concrete type holding different implementations
- Runtime polymorphism, can reassign the filter function
- All callables need to inherit the interface
- Not possible to pass third-party defined callables
- Pointers, lifetime management, ...

Interfaces and third-party functions

```
1  namespace cppcon {
2    struct InstrumentFilterConcept
3    {
4      virtual bool operator()(InstrumentDef const&) const = 0;
5      virtual ~InstrumentFilterConcept() {}
6    };
7
8    struct IsStockFilter : InstrumentFilterConcept
9    {
10     bool operator()(InstrumentDef const& def) const override { return def.isEquity();
11   };
12
13   InstrumentDefList filterInstruments(InstrumentDefList instruments, InstrumentFilter
14   {
15     ...
16   }
17 }
18
19 namespace thirdparty {
20   bool isMillionDollarStock(InstrumentDef const& def) { ... }
```

Interfaces and third-party functions

```
  struct IsStockFilter : InstrumentFilterConcept
  {
    bool operator()(InstrumentDef const& def) const override { return def.isEquity();
  };

  InstrumentDefList filterInstruments(InstrumentDefList instruments, InstrumentFilte
  {
    ...
  }
}

namespace thirdparty {
  bool isMillionDollarStock(InstrumentDef const& def) { ... }
}

int main()
{
  cppcon::filterInstruments(def, /*thirdparty::isMillionDollarStock*/);
}
```

Interfaces and third-party functions

```
 8   struct IsStockFilter : InstrumentFilterConcept
 9   {
10     bool operator()(InstrumentDef const& def) const override { return def.isEquity();
11   };
12
13   InstrumentDefList filterInstruments(InstrumentDefList instruments, InstrumentFilte
14   {
15     ...
16   }
17 }
18
19 namespace thirdparty {
20   bool isMillionDollarStock(InstrumentDef const& def) { ... }
21 }
22
23 int main()
24 {
25   cppcon::filterInstruments(def, /*thirdparty::isMillionDollarStock*/);
26 }
```

```cpp
struct IsMillionDollarStockFilter : cppcon::InstrumentFilterConcept
{
  bool operator()(InstrumentDef const& def) const override {
    return thirdparty::isMillionDollarStock(def);
  }
};

int main()
{
  cppcon::filterInstruments(def, std::make_unique<IsMillionDollarStockFilter>{});
}
```

```cpp
struct IsMillionDollarStockFilter : cppcon::InstrumentFilterConcept
{
  bool operator()(InstrumentDef const& def) const override {
    return thirdparty::isMillionDollarStock(def);
  }
};

int main()
{
  cppcon::filterInstruments(def, std::make_unique<IsMillionDollarStockFilter>{});
}
```

```cpp
struct IsMillionDollarStockFilter : cppcon::InstrumentFilterConcept
{
  bool operator()(InstrumentDef const& def) const override {
    return thirdparty::isMillionDollarStock(def);
  }
};

int main()
{
  cppcon::filterInstruments(def, std::make_unique<IsMillionDollarStockFilter>{});
}
```

~~Not~~ Possible to pass third-party functions

```cpp
struct IsMillionDollarStockFilter : cppcon::InstrumentFilterConcept
{
  bool operator()(InstrumentDef const& def) const override {
    return thirdparty::isMillionDollarStock(def);
  }
};

int main()
{
  cppcon::filterInstruments(def, std::make_unique<IsMillionDollarStockFilter>{});
}
```

~~Not~~ Possible to pass third-party functions

although cumbersome..

```
1  namespace cppcon {
2    struct IsStockFilter : InstrumentFilterConcept
3    {
4      bool operator()(InstrumentDef const& def) const override
5      {
6        return def.isEquity();
7      }
8    };
9  }
10 namespace myapp {
11   struct IsMillionDollarStockFilter : cppcon::InstrumentFilterConcept
12   {
13     bool operator()(InstrumentDef const& def) const override {
14       return thirdparty::isMillionDollarStock(def);
15     }
16   };
17 }
```

```
1  namespace cppcon {
2    struct IsStockFilter : InstrumentFilterConcept
3    {
4      bool operator()(InstrumentDef const& def) const override
5      {
6        return def.isEquity();
7      }
8    };
9  }
10 namespace myapp {
11   struct IsMillionDollarStockFilter : cppcon::InstrumentFilterConcept
12   {
13     bool operator()(InstrumentDef const& def) const override {
14       return thirdparty::isMillionDollarStock(def);
15     }
16   };
17 }
```

```
1  namespace cppcon {
2    struct IsStockFilter : InstrumentFilterConcept
3    {
4      bool operator()(InstrumentDef const& def) const override
5      {
6        return def.isEquity();
7      }
8    };
9  }
10 namespace myapp {
11   struct IsMillionDollarStockFilter : cppcon::InstrumentFilterConcept
12   {
13     bool operator()(InstrumentDef const& def) const override {
14       return thirdparty::isMillionDollarStock(def);
15     }
16   };
17 }
```

Goal: A *generic* class derived from InstrumentFilterConcept, constructible from any callable

Template + Interface

```
namespace cppcon {

  template <typename FuncType>
  struct InstrumentFilterModel : InstrumentFilterConcept
  {
    InstrumentFilterModel(FuncType f)
    : mFunc{std::move(f)}
    {}

    bool operator()(InstrumentDef const& def) const override { return mFunc(i); }

    FuncType mFunc;
  };

}

namespace myapp {
  using thirdparty::isMillionDollarStock;
  using FuncType = std::decay_t<decltype(isMillionDollarStock)>;
```

Template + Interface

```
1  namespace cppcon {
2
3    template <typename FuncType>
4    struct InstrumentFilterModel : InstrumentFilterConcept
5    {
6      InstrumentFilterModel(FuncType f)
7      : mFunc{std::move(f)}
8      {}
9
10     bool operator()(InstrumentDef const& def) const override { return mFunc(i); }
11
12     FuncType mFunc;
13   };
14
15 }
16
17 namespace myapp {
18   using thirdparty::isMillionDollarStock;
19   using FuncType = std::decay_t<decltype(isMillionDollarStock)>;
20
```

Template + Interface

```
1  namespace cppcon {
2
3    template <typename FuncType>
4    struct InstrumentFilterModel : InstrumentFilterConcept
5    {
6      InstrumentFilterModel(FuncType f)
7      : mFunc{std::move(f)}
8      {}
9
10     bool operator()(InstrumentDef const& def) const override { return mFunc(i); }
11
12     FuncType mFunc;
13   };
14
15 }
16
17 namespace myapp {
18   using thirdparty::isMillionDollarStock;
19   using FuncType = std::decay_t<decltype(isMillionDollarStock)>;
20
```

Template + Interface

```
7       : mFunc{std::move(f)}
8       {}
9
10      bool operator()(InstrumentDef const& def) const override { return mFunc(i); }
11
12      FuncType mFunc;
13    };
14
15  }
16
17  namespace myapp {
18    using thirdparty::isMillionDollarStock;
19    using FuncType = std::decay_t<decltype(isMillionDollarStock)>;
20
21    cppcon::filterInstruments(input,
22      new cppcon::InstrumentFilterModel<FuncType>(isMillionDollarStock));
23
24    cppcon::filterInstruments(input,
25      new cppcon::InstrumentFilterModel(cppcon::IsStockFilter{}));
26  }
```

Template + Interface

```
7       : mFunc{std::move(f)}
8       {}
9
10      bool operator()(InstrumentDef const& def) const override { return mFunc(i); }
11
12      FuncType mFunc;
13    };
14
15  }
16
17  namespace myapp {
18    using thirdparty::isMillionDollarStock;
19    using FuncType = std::decay_t<decltype(isMillionDollarStock)>;
20
21    cppcon::filterInstruments(input,
22      new cppcon::InstrumentFilterModel<FuncType>(isMillionDollarStock));
23
24    cppcon::filterInstruments(input,
25      new cppcon::InstrumentFilterModel(cppcon::IsStockFilter{}));
26  }
```

Template + Interface

```
1  namespace cppcon {
2
3    template <typename FuncType>
4    struct InstrumentFilterModel : InstrumentFilterConcept
5    {
6      InstrumentFilterModel(FuncType f)
7      : mFunc{std::move(f)}
8      {}
9
10     bool operator()(InstrumentDef const& def) const override { return mFunc(i); }
11
12     FuncType mFunc;
13   };
14
15 }
16
17 namespace myapp {
18   using thirdparty::isMillionDollarStock;
19   using FuncType = std::decay_t<decltype(isMillionDollarStock)>;
20
```

No longer need to create functor inheriting from the concept

RECAP

RECAP

- `filterInstruments()` can be called using any callable wrapped in InstrumentFilterModel

RECAP

- `filterInstruments()` can be called using any callable wrapped in InstrumentFilterModel
- The client has to wrap function in a pointer - it's clunky

RECAP

- `filterInstruments()` can be called using any callable wrapped in InstrumentFilterModel
- The client has to wrap function in a pointer - it's clunky
- Lifetime management

```
struct Function
{
private:
  struct InstrumentFilterConcept
  {
    virtual bool operator()(InstrumentDef const&) = 0;
    virtual ~InstrumentFilterConcept() {}
  };

  template <typename FuncType>
  struct InstrumentFilterModel : InstrumentFilterConcept
  {
    explicit InstrumentFilterModel(FuncType f)
    : mFunc{std::move(f)}
    {}

    bool operator()(InstrumentDef const& def) override { return mFunc(def); }

    FuncType mFunc;
  };
public:
  template <typename FuncType>
  explicit Function(FuncType&& func)
  : mFunc{std::make_unique<FunctionModel<std::decay_t<FuncType>>>(std::forward<FuncType>(func))}
  {}

  bool operator()(InstrumentFilter const& def)
  {
    if (not mFunc)
      throw std::bad_function_call();
```

```
  template <typename FuncType>
  struct InstrumentFilterModel : InstrumentFilterConcept
  {
    explicit InstrumentFilterModel(FuncType f)
    : mFunc{std::move(f)}
    {}

    bool operator()(InstrumentDef const& def) override { return mFunc(def); }

    FuncType mFunc;
  };
public:
  template <typename FuncType>
  explicit Function(FuncType&& func)
  : mFunc{std::make_unique<FunctionModel<std::decay_t<FuncType>>>(std::forward<FuncType>(func))}
  {}

  bool operator()(InstrumentFilter const& def)
  {
    if (not mFunc)
      throw std::bad_function_call();
    mFunc->operator()(def);
  }

  std::unique_ptr<InstrumentFilterConcept> mFunc;
};

filterInstruments(input, Function(thirdparty::isMillionDollarStock));
filterInstruments(input, Function(cppcon::IsStockFilter{}));
```

```
  template <typename FuncType>
  struct InstrumentFilterModel : InstrumentFilterConcept
  {
    explicit InstrumentFilterModel(FuncType f)
    : mFunc{std::move(f)}
    {}

    bool operator()(InstrumentDef const& def) override { return mFunc(def); }

    FuncType mFunc;
  };
public:
  template <typename FuncType>
  explicit Function(FuncType&& func)
  : mFunc{std::make_unique<FunctionModel<std::decay_t<FuncType>>>(std::forward<FuncType>(func))}
  {}

  bool operator()(InstrumentFilter const& def)
  {
    if (not mFunc)
      throw std::bad_function_call();
    mFunc->operator()(def);
  }

  std::unique_ptr<InstrumentFilterConcept> mFunc;
};

filterInstruments(input, Function(thirdparty::isMillionDollarStock));
filterInstruments(input, Function(cppcon::IsStockFilter{}));
```

```
  template <typename FuncType>
  struct InstrumentFilterModel : InstrumentFilterConcept
  {
    explicit InstrumentFilterModel(FuncType f)
    : mFunc{std::move(f)}
    {}

    bool operator()(InstrumentDef const& def) override { return mFunc(def); }

    FuncType mFunc;
  };
public:
  template <typename FuncType>
  explicit Function(FuncType&& func)
  : mFunc{std::make_unique<FunctionModel<std::decay_t<FuncType>>>(std::forward<FuncType>(func))}
  {}

  bool operator()(InstrumentFilter const& def)
  {
    if (not mFunc)
      throw std::bad_function_call();
    mFunc->operator()(def);
  }

  std::unique_ptr<InstrumentFilterConcept> mFunc;
};

filterInstruments(input, Function(thirdparty::isMillionDollarStock));
filterInstruments(input, Function(cppcon::IsStockFilter{}));
```

```cpp
struct Function
{
private:
  struct InstrumentFilterConcept
  {
    virtual bool operator()(InstrumentDef const&) = 0;
    virtual ~InstrumentFilterConcept() {}
  };

  template <typename FuncType>
  struct InstrumentFilterModel : InstrumentFilterConcept
  {
    explicit InstrumentFilterModel(FuncType f)
    : mFunc{std::move(f)}
    {}

    bool operator()(InstrumentDef const& def) override { return mFunc(def); }

    FuncType mFunc;
  };
public:
  template <typename FuncType>
  explicit Function(FuncType&& func)
  : mFunc{std::make_unique<FunctionModel<std::decay_t<FuncType>>>(std::forward<FuncType>(func))}
  {}

  bool operator()(InstrumentFilter const& def)
  {
    if (not mFunc)
      throw std::bad_function_call();
```

std::move_only_function

```cpp
struct Function
{
private:
  struct InstrumentFilterConcept
  {
    virtual bool operator()(InstrumentDef const&) = 0;
    virtual ~InstrumentFilterConcept() {}
  };

  template <typename FuncType>
  struct InstrumentFilterModel : InstrumentFilterConcept
  {
    explicit InstrumentFilterModel(FuncType f)
    : mFunc{std::move(f)}
    {}

    bool operator()(InstrumentDef const& def) override { return mFunc(def); }

    FuncType mFunc;
  };
public:
  template <typename FuncType>
  explicit Function(FuncType&& func)
  : mFunc{std::make_unique<FunctionModel<std::decay_t<FuncType>>>(std::forward<FuncType>(func))}
  {}

  bool operator()(InstrumentFilter const& def)
  {
    if (not mFunc)
      throw std::bad_function_call();
```

std::move_only_function

"when in doubt, do as the ints do"
- Scott Meyers (More Effective C++)

```cpp
struct Function
{
private:
  struct InstrumentFilterConcept
  {
    virtual bool operator()(InstrumentDef const&) = 0;
    virtual std::unique_ptr<InstrumentFilterConcept> clone() const = 0;
    virtual ~InstrumentFilterConcept() {}
  };

  template <typename FuncType>
  struct InstrumentFilterModel : InstrumentFilterConcept
  {
    explicit InstrumentFilterModel(FuncType f)
    : mFunc{std::move(f)}
    {}

    std::unique_ptr<InstrumentFilterConcept> clone() const override
    {
      return std::make_unique<InstrumentFilterModel>(*this);
    }

    bool operator()(InstrumentDef const& def) override { return mFunc(def); }

    FuncType mFunc;
  };
public:
  template <typename FuncType>
  explicit Function(FuncType&& func)
  : mFunc{std::make_unique<InstrumentFilterModel<std::decay_t<FuncType>>>(std::forward<FuncType>(func))}
```

Prototype design pattern

```
    {
      return std::make_unique<InstrumentFilterModel>(*this);
    }

    bool operator()(InstrumentDef const& def) override { return mFunc(def); }

    FuncType mFunc;
  };
public:
  template <typename FuncType>
  explicit Function(FuncType&& func)
  : mFunc{std::make_unique<InstrumentFilterModel<std::decay_t<FuncType>>>(std::forward<FuncType>(func))}
  {}

  Function(Function const& other)
  : mFunc(other.mFunc ? other.mFunc->clone() : nullptr)
  {}

  bool operator()(InstrumentDef const& def)
  {
    if (not mFunc)
      throw std::bad_function_call();
    mFunc->operator()(def);
  }

  std::unique_ptr<InstrumentFilterConcept> mFunc;
};

struct IsEquityFilter
{
  IsEquityFilter() = default;
```

Prototype design pattern

```
  std::unique_ptr<InstrumentFilterConcept> mFunc;
};

struct IsEquityFilter
{
  IsEquityFilter() = default;
  IsEquityFilter(IsEquityFilter&&) = default;
  IsEquityFilter(IsEquityFilter const&)
  {
    std::cout << "IsEquityFilter ctor\n";
  }

  bool operator()(InstrumentDef const&)
  {
    std::cout << std::format("Invoked IsEquityFilter {} times\n", ++counter);
    return true;
  }

  static inline unsigned counter = 0;
};

int main()
{
  Function f1(IsEquityFilter{});
  f1(123); // Invoked IsEquityFilter 1 times

  Function f2 = f1; // IsEquityFilter ctor
  f2(123); // Invoked IsEquityFilter 2 times
}
```

Prototype design pattern

RECAP

RECAP

- `filterInstruments()` can be called using any callable

RECAP

- `filterInstruments()` can be called using any callable
- Pointers and lifetime management abstracted from the client

RECAP

- `filterInstruments()` can be called using any callable
- Pointers and lifetime management abstracted from the client
- Value semantics

RECAP

- `filterInstruments()` can be called using any callable
- Pointers and lifetime management abstracted from the client
- Value semantics
- Implicit assumptions about the callable type

RECAP

- `filterInstruments()` can be called using any callable
- Pointers and lifetime management abstracted from the client
- Value semantics
- Implicit assumptions about the callable type
- Only works for bool(InstrumentDef const&)

FINALLY...MORE TEMPLATES

```
template <typename T>
struct Function;

template <typename R, typename... Args>
struct Function<R(Args...)>
{
private:
  ...
public:
  template <typename FuncType>
    requires (std::is_invocable_r_v<R, FuncType, Args...>)
  explicit Function(FuncType&& func)
  : mFunc{std::make_unique<FunctionModel<std::decay_t<FuncType>>>(std::forward(func))}
  {}

  R operator()(Args... args)
  {
    if (not mFunc)
      throw std::bad_function_call();
    return mFunc->operator()(std::forward<Args>(args)...);
  }

  std::unique_ptr<FunctionConcept> mFunc;
};
```

FINALLY...MORE TEMPLATES

```
template <typename T>
struct Function;

template <typename R, typename... Args>
struct Function<R(Args...)>
{
private:
  ...
public:
  template <typename FuncType>
    requires (std::is_invocable_r_v<R, FuncType, Args...>)
  explicit Function(FuncType&& func)
  : mFunc{std::make_unique<FunctionModel<std::decay_t<FuncType>>>(std::forward(func))}
  {}

  R operator()(Args... args)
  {
    if (not mFunc)
      throw std::bad_function_call();
    return mFunc->operator()(std::forward<Args>(args)...);
  }

  std::unique_ptr<FunctionConcept> mFunc;
};
```

FINALLY...MORE TEMPLATES

```
template <typename T>
struct Function;

template <typename R, typename... Args>
struct Function<R(Args...)>
{
private:
  ...
public:
  template <typename FuncType>
    requires (std::is_invocable_r_v<R, FuncType, Args...>)
  explicit Function(FuncType&& func)
  : mFunc{std::make_unique<FunctionModel<std::decay_t<FuncType>>>(std::forward(func))}
  {}

  R operator()(Args... args)
  {
    if (not mFunc)
      throw std::bad_function_call();
    return mFunc->operator()(std::forward<Args>(args)...);
  }

  std::unique_ptr<FunctionConcept> mFunc;
};
```

Cost of Type Erased Class

Cost of Type Erased Class

1. Heap allocation on construction

Cost of Type Erased Class

1. Heap allocation on construction[*] small buffer optimisation (SBO)

Cost of Type Erased Class

1. Heap allocation on construction* small buffer optimisation (SBO)

2. Virtual function calls

Cost of Type Erased Class

1. Heap allocation on construction* small buffer optimisation (SBO)

2. Virtual function calls* manual virtual dispatch (MVD)

Small Buffer Optimisation

```
1  struct Function
2  {
3    using StorageModelType = FunctionModel<bool(*)(int)>;
4    inline static constexpr size_t StorageSize = sizeof(StorageModelType);
5
6    ...
7
8    FunctionConcept* mFunc;
9    alignas(StorageModelType) std::byte mStorage[StorageSize];
10 };
```

SBO - Constructor

```
1  struct Function
2  {
3    using StorageModelType = FunctionModel<bool(*)(int)>;
4    inline static constexpr size_t StorageSize = sizeof(StorageModelType);
5
6    template <typename FuncType>
7    explicit Function(FuncType&& func)
8    {
9      using T = FunctionModel<std::decay_t<FuncType>>;
10     if constexpr (sizeof(T) <= StorageSize)
11     {
12       mFunc = std::construct_at(std::addressof(mStorage), T(std::forward<FuncType>(func)));
13     }
14     else
15     {
16       mFunc = new T(std::forward<FuncType>(func));
17     }
18   }
19
20   FunctionConcept* mFunc;
21   alignas(StorageModelType) std::byte mStorage[StorageSize];
22 };
```

SBO - Clone

```
struct Function
{
private:
  template <typename T>
  struct FunctionModel : FunctionConcept
  {
    FunctionConcept* clone(StorageType& storage) const override
    {
      if constexpr (sizeof(*this) <= sizeof(storage))
      {
        return std::construct_at(std::addressof(storage), FunctionModel(*this));
      }
      else
      {
        return new FunctionModel(*this);
      }
    }
  };
public:
  ...

  Function(Function const& other)
  : mFunc{other.mFunc ? other.mFunc->clone(mStorage) : nullptr}
  {

  }

  FunctionConcept* mFunc;
  alignas(StorageModelType) std::byte mStorage[StorageSize];
};
```

std::any

The class any describes a *type-safe* container for single values of any *copy constructible* type

```cpp
std::any a = 1;
std::cout << std::format("{}: {}\n", a.type().name(), std::any_cast<int>(a)); // int: 1

a = 3.14;
std::cout << std::format("{}: {}\n", a.type().name(), std::any_cast<double>(a)); // double: 3.14

a = true;
try
{
  std::any_cast<double>(a);
}
catch (const std::bad_any_cast&)
{
  // BAD CAST!
}
```

```cpp
std::any a = 1;
std::cout << std::format("{}: {}\n", a.type().name(), std::any_cast<int>(a)); // int: 1

a = 3.14;
std::cout << std::format("{}: {}\n", a.type().name(), std::any_cast<double>(a)); // double: 3.14

a = true;
try
{
  std::any_cast<double>(a);
}
catch (const std::bad_any_cast&)
{
  // BAD CAST!
}
```

```cpp
std::any a = 1;
std::cout << std::format("{}: {}\n", a.type().name(), std::any_cast<int>(a)); // int: 1

a = 3.14;
std::cout << std::format("{}: {}\n", a.type().name(), std::any_cast<double>(a)); // double: 3.14

a = true;
try
{
  std::any_cast<double>(a);
}
catch (const std::bad_any_cast&)
{
  // BAD CAST!
}
```

```
class any
{
public:
  template <typename T>
  any(T&& obj)
  : mObj{new AnyModel{std::forward(obj)}}
  {}

  AnyConcept* mObj = nullptr;
};
```

```
class any
{
private:
  struct AnyConcept
  {
    virtual std::type_index type() const = 0;
    virtual ~AnyConcept() {}
  };

  template <typename T>
  struct AnyModel : AnyConcept
  {
    AnyModel(T obj)
    : typeInfo{typeid(T)}
    {}

    std::type_index type() const override
    {
      return typeInfo;
    }

    ~AnyModel() override {}

    std::type_index typeInfo;
  };
public:
  template <typename T>
  any(T&& obj)
  : mObj{new AnyModel{std::forward<T>(obj)}}
  {}
```

```
private:
  struct AnyConcept
  {
    virtual std::type_index type() const = 0;
    virtual ~AnyConcept() {}
  };

  template <typename T>
  struct AnyModel : AnyConcept
  {
    AnyModel(T obj)
    : typeInfo{typeid(T)}
    {}

    std::type_index type() const override
    {
      return typeInfo;
    }

    ~AnyModel() override {}

    std::type_index typeInfo;
  };
public:
  template <typename T>
  any(T&& obj)
  : mObj{new AnyModel{std::forward<T>(obj)}}
  {}

  std::type_index type() const { return mObj ? mObj->type() : typeid(void); }

```

```
  struct AnyModel : AnyConcept
  {
    AnyModel(T obj)
    : typeInfo{typeid(T)}
    {}

    std::type_index type() const override
    {
      return typeInfo;
    }

    ~AnyModel() override {}

    std::type_index typeInfo;
  };
public:
  template <typename T>
  any(T&& obj)
  : mObj{new AnyModel{std::forward<T>(obj)}}
  {}

  std::type_index type() const { return mObj ? mObj->type() : typeid(void); }

  ~any()
  {
    delete mObj;
  }

  AnyConcept* mObj = nullptr;
};
```

Type erasure without virtual

```
1  class any
2  {
3  public:
4    template <typename T>
5    any(T&& value)
6    : mValue{new T{std::forward<T>(value)}}
7    , mType{typeid(T)}
8    {}
9
10   ~any() { ... }
11
12   std::type_index type() const { return mValue ? mType : typeid(void): ; }
13
14 private:
15   void* mValue;
16   std::type_index mType;
17 };
```

Type erasure without virtual

```
1  class any
2  {
3  public:
4    template <typename T>
5    any(T&& value)
6    : mValue{new T{std::forward<T>(value)}}
7    , mType{typeid(T)}
8    {}
9
10   ~any() { ... }
11
12   std::type_index type() const { return mValue ? mType : typeid(void): ; }
13
14 private:
15   void* mValue;
16   std::type_index mType;
17 };
```

```cpp
class any
{
public:
  template <typename T>
  any(T&& value)
  : mValue{ new T{std::forward<T>(value)} }
  , mType{typeid(T)}
  , mDeleter{ [](void* ptr) { delete static_cast<T*>(ptr); } }
  {}

  ~any() { if (mValue) mDeleter(mValue); }

  std::type_index type() const { return mValue ? mType : typeid(void): ; }

private:
  void* mValue;
  std::type_index mType;
  std::function<void(void*)> mDeleter;
};
```

RECAP

RECAP

- Type Erasure enables the use of unrelated types with a uniform interface

RECAP

- Type Erasure enables the use of unrelated types with a uniform interface
- Does not require inheritance and reduces dependencies

RECAP

- Type Erasure enables the use of unrelated types with a uniform interface
- Does not require inheritance and reduces dependencies
- Prevelant in the standard library

RECAP

- Type Erasure enables the use of unrelated types with a uniform interface
- Does not require inheritance and reduces dependencies
- Prevelant in the standard library
- Achieves duck typing, common pattern in other languages

Bonus: shared_ptr and type safety

```
struct NoisyFoo
{
  NoisyFoo() { std::cout << "Constructed NoisyFoo\n"; }
  ~NoisyFoo() { std::cout << "Destructed NoisyFoo\n"; }
};

void* ptr = new NoisyFoo();
...
delete ptr;
```

```cpp
struct NoisyFoo
{
  NoisyFoo() { std::cout << "Constructed NoisyFoo\n"; }
  ~NoisyFoo() { std::cout << "Destructed NoisyFoo\n"; }
};

void* ptr = new NoisyFoo();
...
delete ptr;
```

[gcc14] warning: deleting 'void*' is undefined
[clang19] warning: cannot delete expression with pointer-to-'void' type 'void *'

```cpp
struct NoisyFoo
{
  NoisyFoo() { std::cout << "Constructed NoisyFoo\n"; }
  ~NoisyFoo() { std::cout << "Destructed NoisyFoo\n"; }
};

void* ptr = new NoisyFoo();
...
delete ptr;
```

[gcc14] warning: deleting 'void*' is undefined
[clang19] warning: cannot delete expression with pointer-to-'void' type 'void *'

An object cannot be deleted using a pointer of type void* because void is not an object type

```cpp
struct NoisyFoo
{
  NoisyFoo() { std::cout << "Constructed NoisyFoo\n"; }
  ~NoisyFoo() { std::cout << "Destructed NoisyFoo\n"; }
};

void* ptr = new NoisyFoo();
...
delete ptr;
```

[gcc14] warning: deleting 'void*' is undefined
[clang19] warning: cannot delete expression with pointer-to-'void' type 'void *'

An object cannot be deleted using a pointer of type void* because void is not an object type

```cpp
auto fooDeleter = [](void* p) {
  delete static_cast<NoisyFoo*>(p);
};
void* ptr = new NoisyFoo();
...
fooDeleter(ptr);
```

```
1 std::unique_ptr<void> ptr(new NoisyFoo{});
```

```cpp
std::unique_ptr<void> ptr(new NoisyFoo{});
```

[gcc14] error: static assertion failed: can't delete pointer to incomplete type
[clang19] error: invalid application of 'sizeof' to an incomplete type 'void'

```cpp
std::unique_ptr<void> ptr(new NoisyFoo{});
```

[gcc14] error: static assertion failed: can't delete pointer to incomplete type
[clang19] error: invalid application of 'sizeof' to an incomplete type 'void'

```cpp
template<
  class T,
  class Deleter = std::default_delete<T>
> class unique_ptr;

auto fooDeleter = [](void* p) {
  delete static_cast<NoisyFoo*>(p);
};
std::unique_ptr<void, decltype<fooDeleter>> ptr(new NoisyFoo{});

ptr = std::unique_ptr<void, decltype(barDeleter)>(new NoisyBar()); // not allowed!
```

```cpp
std::unique_ptr<void> ptr(new NoisyFoo{});
```

[gcc14] error: static assertion failed: can't delete pointer to incomplete type
[clang19] error: invalid application of 'sizeof' to an incomplete type 'void'

```cpp
template<
  class T,
  class Deleter = std::default_delete<T>
> class unique_ptr;

auto fooDeleter = [](void* p) {
  delete static_cast<NoisyFoo*>(p);
};
std::unique_ptr<void, decltype<fooDeleter>> ptr(new NoisyFoo{});

ptr = std::unique_ptr<void, decltype(barDeleter)>(new NoisyBar()); // not allowed!
```

```cpp
std::unique_ptr<void> ptr(new NoisyFoo{});
```

[gcc14] error: static assertion failed: can't delete pointer to incomplete type
[clang19] error: invalid application of 'sizeof' to an incomplete type 'void'

```cpp
template<
  class T,
  class Deleter = std::default_delete<T>
> class unique_ptr;

auto fooDeleter = [](void* p) {
  delete static_cast<NoisyFoo*>(p);
};
std::unique_ptr<void, decltype<fooDeleter>> ptr(new NoisyFoo{});

ptr = std::unique_ptr<void, decltype(barDeleter)>(new NoisyBar()); // not allowed!
```

```
1 std::shared_ptr<void> ptr = std::make_shared<NoisyFoo>();
```

```
std::shared_ptr<void> ptr = std::make_shared<NoisyFoo>();
```

```
ptr = std::make_shared<NoisyBar>(); // works!
// Constructed Foo
// Constructed Bar
// Destructed Foo
// Destructed Bar
```

```
std::shared_ptr<void> ptr = std::make_shared<NoisyFoo>();
```

```
ptr = std::make_shared<NoisyBar>(); // works!
// Constructed Foo
// Constructed Bar
// Destructed Foo
// Destructed Bar
```

```
template<
  class T,
  class Deleter = std::default_delete<T>
> class unique_ptr;

template<class T> class shared_ptr;
```

Type Erasure in shared_ptr

Type Erasure in shared_ptr

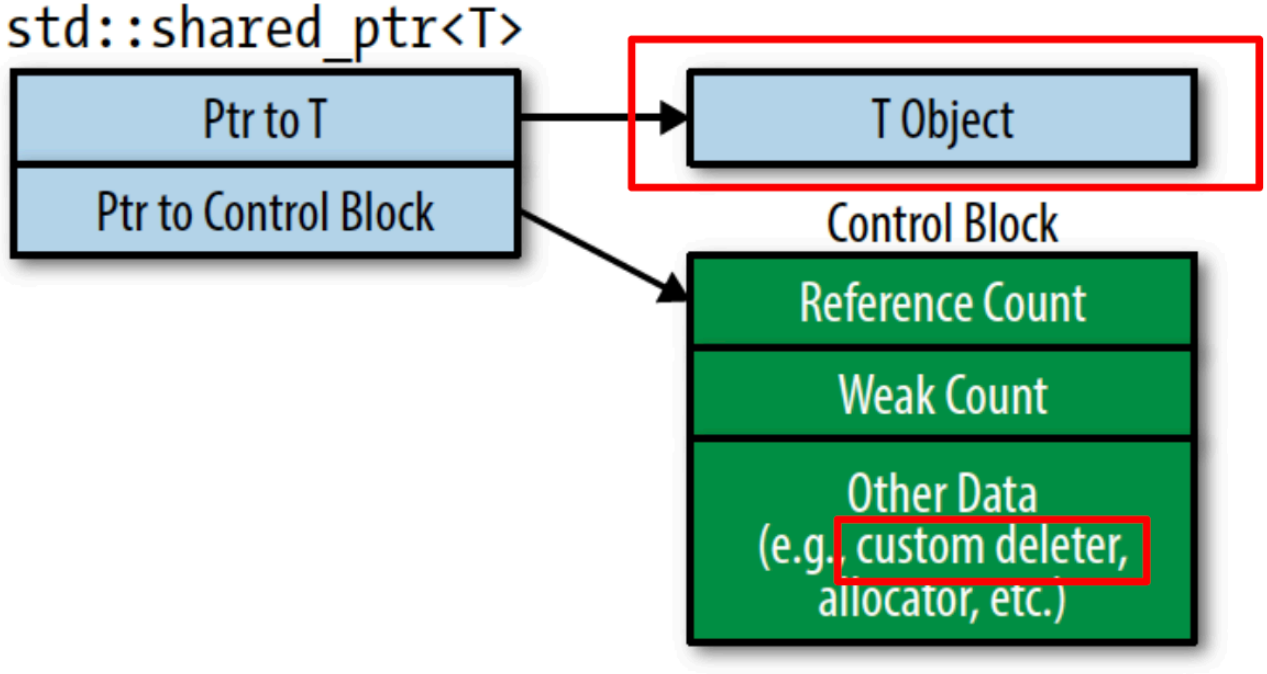
- unique_ptr<T> only stores the template type information

Type Erasure in shared_ptr

- unique_ptr<T> only stores the template type information
- shared_ptr<T> stores both template type information and the object type information

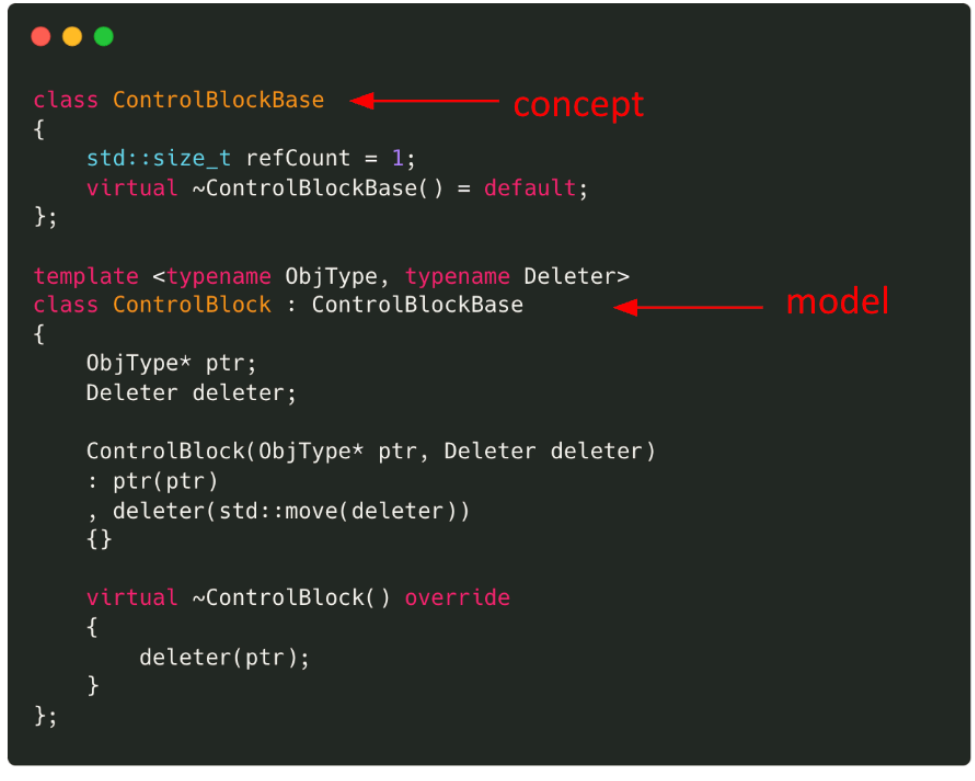
Type Erasure in shared_ptr

- unique_ptr<T> only stores the template type information
- shared_ptr<T> stores both template type information and the object type information
- shared_ptr<T> uses type erasure to call the correct destructor


std::shared_ptr<T>
Ptr to T
Ptr to Control Block
T Object
Control Block
Reference Count
Weak Count
Other Data
(e.g., custom deleter,
allocator, etc.)

```cpp
template<typename T>
class shared_ptr
{
public:
  template<typename Y> requires std::is_convertible_v<Y*, T*>
  explicit shared_ptr(Y* ptr)
  : shared_ptr{ptr, std::default_delete<Y>()}
  {}

  template<typename Y, typename Deleter> requires std::is_convertible_v<Y*, T*>
  shared_ptr(Y* ptr, Deleter deleter)
  : ptr{ptr}
  // type erase Y* ptr and deleter
  {}

private:
  T* ptr;
};
```

```cpp
template<typename T>
class shared_ptr
{
public:
  template<typename Y> requires std::is_convertible_v<Y*, T*>
  explicit shared_ptr(Y* ptr)
  : shared_ptr{ptr, std::default_delete<Y>()}
  {}

  template<typename Y, typename Deleter> requires std::is_convertible_v<Y*, T*>
  shared_ptr(Y* ptr, Deleter deleter)
  : ptr{ptr}
  // type erase Y* ptr and deleter
  {}

private:
  T* ptr;
};
```


class ControlBlockBase
concept
{
std::size_t refCount = 1;
virtual ~ControlBlockBase() = default;
};
template <typename ObjType, typename Deleter>
class ControlBlock : ControlBlockBase
model
{
ObjType* ptr;
Deleter deleter;
ControlBlock(ObjType* ptr, Deleter deleter)
: ptr(ptr)
, deleter(std::move(deleter))
{}
virtual ~ControlBlock() override
{
deleter(ptr);
}
};

THANK YOU!