Unveiling Type Erasure in C++

From std::function to std::any

Sarthak Sehgal

Sarthak Sehgal

- Low latency C++ Software Engineer
- Interested in finance, and low level programming
- Excited for my first in person talk
- Sometimes, I write about C++ at sartech.substack.com
- Email: sarthaksehgal00@gmail.com

OVERVIEW

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

```cpp
std::vector<int>
  filterRange(std::vector<int> input, FilterFunc filterOn)
{
  std::vector<int> result;
  for (int i : input)
  {
    if (filterOn(i))
    {
      result.push_back(i);
    }
  }
  return result;
}
```

Consider a generic filter function

```cpp
bool isOdd(int i) { return i & 1; }
filterRange(myVector, isOdd);

filterRange(myVector, [](int i) { return i&1; });

struct IsOddFunctor
{
  bool operator()(int i) const
  {
    return i & 1;
  }
};
filterRange(myVector, IsOddFunctor());
```

Generic callable with free function, lambda, functor, etc.

```cpp
bool isOdd(int i) { return i & 1; }
filterRange(myVector, isOdd);

filterRange(myVector, [](int i) { return i&1; });

struct IsOddFunctor
{
  bool operator()(int i) const
  {
    return i & 1;
  }
};
filterRange(myVector, IsOddFunctor());
```

Generic callable with free function, lambda, functor, etc.

```cpp
bool isOdd(int i) { return i & 1; }
filterRange(myVector, isOdd);

filterRange(myVector, [](int i) { return i&1; });

struct IsOddFunctor
{
  bool operator()(int i) const
  {
    return i & 1;
  }
};
filterRange(myVector, IsOddFunctor());
```

Generic callable with free function, lambda, functor, etc.

```cpp
bool isOdd(int i) { return i & 1; }
filterRange(myVector, isOdd);

filterRange(myVector, [](int i) { return i&1; });

struct IsOddFunctor
{
  bool operator()(int i) const
  {
    return i & 1;
  }
};
filterRange(myVector, IsOddFunctor());
```

Generic callable with free function, lambda, functor, etc.

Use Concepts

```
std::vector<int>
  filterRange(std::vector<int> input, std::invocable<bool(int)> auto&& filterOn)
{
  ...
}
```

```cpp
class MarketDataSubscriber
{
  template <std::incovable<bool(InstrumentDefinitionType)> T>
  MarketDataSubscriber(T&& instrumentFilter)
  : instrumentFilterFunc{std::forward<decltype(instrumentFilter)>(instrumentFilter)}
  {}

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

    // process
  }

private:
  ?? instrumentFilterFunc;
};
```

```cpp
class MarketDataSubscriber
{
  template <std::incovable<bool(InstrumentDefinitionType)> T>
  MarketDataSubscriber(T&& instrumentFilter)
  : instrumentFilterFunc{std::forward<decltype(instrumentFilter)>(instrumentFilter)}
  {}

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

    // process
  }

private:
  ?? instrumentFilterFunc;
};
```

Function Pointer?

```
std::vector<int>
  filterRange(std::vector<int> input, bool (*filterOn)(int))
{
  ...
}
```

Downsides on Function Pointers

Downsides on Function Pointers

- Callables need to be stateless

Downsides on Function Pointers

- Callables need to be stateless
- Pointers, and lack of value semantics

Downsides on Function Pointers

- Callables need to be stateless
- Pointers, and lack of value semantics
- Need to perform null checks

Interface and inheritance?

```
struct IFunction
{
  virtual bool operator()(int i) = 0;
};

struct IsEvenFunctor : IFunction
{
  bool operator()(int i) override { return !(i & 1); }
};

struct IsOddFunctor : IFunction
{
  bool operator()(int i) override { return i & 1; }
};

std::vector<int>
  filterRange(std::vector<int> input, IFunction* filterOn)
{
  std::vector<int> output;
  for (int i : input)
```

Interface and inheritance?

```
struct IFunction
{
  virtual bool operator()(int i) = 0;
};

struct IsEvenFunctor : IFunction
{
  bool operator()(int i) override { return !(i & 1); }
};

struct IsOddFunctor : IFunction
{
  bool operator()(int i) override { return i & 1; }
};

std::vector<int>
  filterRange(std::vector<int> input, IFunction* filterOn)
{
  std::vector<int> output;
  for (int i : input)
```

Interface and inheritance?

```cpp
  bool operator()(int i) override { return !(i & 1); }
};

struct IsOddFunctor : IFunction
{
  bool operator()(int i) override { return i & 1; }
};

std::vector<int>
  filterRange(std::vector<int> input, IFunction* filterOn)
{
  std::vector<int> output;
  for (int i : input)
  {
    if (filterOn->operator()(i))
      output.push_back(i);
  }
  return output;
}
```

Downsides of Inheritance

Downsides of Inheritance

- All callables need to inherit the interface

Downsides of Inheritance

- All callables need to inherit the interface
- Not possible to pass third-party function

Downsides of Inheritance

- All callables need to inherit the interface
- Not possible to pass third-party function
- Pointers, lifetime management, ...

std::function for the rescue

```cpp
std::vector<int> filterRange(
  std::vector<int> const& input,
  std::function<bool(int)> filterOn)
{
  ...
}

filterRange(myVector, isOdd); // free function

filterRange(myVector, [](int i) { return i&1; }); // lambda

filterRange(myVector, IsOddFunctor()); // functor
```

std::function for the rescue

```
std::vector<int> filterRange(
  std::vector<int> const& input,
  std::function<bool(int)> filterOn)
{
  ...
}

filterRange(myVector, isOdd); // free function

filterRange(myVector, [](int i) { return i&1; }); // lambda

filterRange(myVector, IsOddFunctor()); // functor
```

std::function for the rescue

```cpp
std::vector<int> filterRange(
  std::vector<int> const& input,
  std::function<bool(int)> filterOn)
{
  ...
}

filterRange(myVector, isOdd); // free function

filterRange(myVector, [](int i) { return i&1; }); // lambda

filterRange(myVector, IsOddFunctor()); // functor
```

THE BIG QUESTION

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

REQUIREMENTS

```
namespace cppnorth {

  struct Function {};

} // namespace cppnorth
```

- Constructible from unrelated types
- Callable common interface

Interfaces and third-party functions

```cpp
namespace cppnorth {
  struct FunctionConcept
  {
    virtual bool operator()(int) const = 0;
    virtual ~FunctionConcept() {}
  };

  struct IsEvenFunctor : FunctionConcept
  {
    bool operator()(int i) const override { return !(i&1); }
  };

  std::vector<int> filterRange(
    std::vector<int> const& input,
    FunctionConcept* filterOn)
  {
    ...
  }
}
```

Interfaces and third-party functions

```
  std::vector<int> filterRange(
    std::vector<int> const& input,
    FunctionConcept* filterOn)
  {
    ...
  }
}

namespace thirdparty {
  bool isPrime(int);
}

namespace myapp {
  int main()
  {
    std::vector<int> input {1, 2, 3};
    std::vector<int> output = cppnorth::filterRange(input, /*thirdparty::isPrime*/);
  }
}
```

Interfaces and third-party functions

```cpp
  std::vector<int> filterRange(
    std::vector<int> const& input,
    FunctionConcept* filterOn)
  {
    ...
  }
}

namespace thirdparty {
  bool isPrime(int);
}

namespace myapp {
  int main()
  {
    std::vector<int> input {1, 2, 3};
    std::vector<int> output = cppnorth::filterRange(input, /*thirdparty::isPrime*/);
  }
}
```

```cpp
namespace myapp {
  struct IsPrimeFunction : cppnorth::FunctionConcept
  {
    bool operator()(int i) const override { return thirdparty::isPrime(i); }
  };

  int main()
  {
    std::vector<int> input {1, 2, 3};
    std::vector<int> output = cppnorth::filterRange(input, new IsPrimeFunction{});
  }
}
```

```cpp
namespace myapp {
  struct IsPrimeFunction : cppnorth::FunctionConcept
  {
    bool operator()(int i) const override { return thirdparty::isPrime(i); }
  };

  int main()
  {
    std::vector<int> input {1, 2, 3};
    std::vector<int> output = cppnorth::filterRange(input, new IsPrimeFunction{});
  }
}
```

```cpp
namespace myapp {
  struct IsPrimeFunction : cppnorth::FunctionConcept
  {
    bool operator()(int i) const override { return thirdparty::isPrime(i); }
  };

  int main()
  {
    std::vector<int> input {1, 2, 3};
    std::vector<int> output = cppnorth::filterRange(input, new IsPrimeFunction{});
  }
}
```

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

```cpp
namespace myapp {
  struct IsPrimeFunction : cppnorth::FunctionConcept
  {
    bool operator()(int i) const override { return thirdparty::isPrime(i); }
  };

  int main()
  {
    std::vector<int> input {1, 2, 3};
    std::vector<int> output = cppnorth::filterRange(input, new IsPrimeFunction{});
  }
}
```

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

although cumbersome..

```
namespace cppnorth {
  struct IsEvenFunctor : FunctionConcept
  {
    bool operator()(int i) const override { return !(i&1); }
  };
}
namespace myapp {
  struct IsPrimeFunction : cppnorth::FunctionConcept
  {
    bool operator()(int i) const override { return thirdparty::isPrime(i); }
  };
}
```

```cpp
namespace cppnorth {
  struct IsEvenFunctor : FunctionConcept
  {
    bool operator()(int i) const override { return !(i&1); }
  };
}
namespace myapp {
  struct IsPrimeFunction : cppnorth::FunctionConcept
  {
    bool operator()(int i) const override { return thirdparty::isPrime(i); }
  };
}
```

```cpp
namespace cppnorth {
  struct IsEvenFunctor : FunctionConcept
  {
    bool operator()(int i) const override { return !(i&1); }
  };
}
namespace myapp {
  struct IsPrimeFunction : cppnorth::FunctionConcept
  {
    bool operator()(int i) const override { return thirdparty::isPrime(i); }
  };
}
```

Goal: A generic class derived from FunctionConcept, constructible from any callable

Just stick a template!

```
namespace cppnorth {

  template <typename FuncType>
  struct FunctionModel : FunctionConcept
  {
    FunctionModel(FuncType f)
    : mFunc{std::move(f)}
    {}

    bool operator()(int i) const override { return mFunc(i); }

    FuncType mFunc;
  };

}

namespace myapp {
  using cppnort::filterRange;
  using thirdparty::isPrime;
```

Just stick a template!

```
namespace cppnorth {

  template <typename FuncType>
  struct FunctionModel : FunctionConcept
  {
    FunctionModel(FuncType f)
    : mFunc{std::move(f)}
    {}

    bool operator()(int i) const override { return mFunc(i); }

    FuncType mFunc;
  };

}

namespace myapp {
  using cppnort::filterRange;
  using thirdparty::isPrime;
```

Just stick a template!

```cpp
namespace cppnorth {

  template <typename FuncType>
  struct FunctionModel : FunctionConcept
  {
    FunctionModel(FuncType f)
    : mFunc{std::move(f)}
    {}

    bool operator()(int i) const override { return mFunc(i); }

    FuncType mFunc;
  };

}

namespace myapp {
  using cppnort::filterRange;
  using thirdparty::isPrime;
```

Just stick a template!

```cpp
  {
    FunctionModel(FuncType f)
    : mFunc{std::move(f)}
    {}

    bool operator()(int i) const override { return mFunc(i); }

    FuncType mFunc;
  };

}

namespace myapp {
  using cppnort::filterRange;
  using thirdparty::isPrime;

  filterRange(input,
    new cppnorth::FunctionModel<std::decay_t<decltype(isPrime)>>(isPrime));

}
```

```cpp
namespace cppnorth {
  struct FunctionConcept
  {
    virtual bool operator()(int) const = 0;
    virtual ~FunctionConcept() {}
  };

  template <typename FuncType>
  struct FunctionModel : FunctionConcept
  {
    FunctionModel(FuncType f)
    : mFunc{std::move(f)}
    {}

    bool operator()(int i) const override { return mFunc(i); }

    FuncType mFunc;
  };

  struct IsEvenFunctor
```

```cpp
    : mFunc{std::move(f)}
    {}

    bool operator()(int i) const override { return mFunc(i); }

    FuncType mFunc;
  };

  struct IsEvenFunctor
  {
    bool operator()(int i) const { return !(i&1); }
  };
}

namespace myapp {
  using cppnort::filterRange;
  using thirdparty::isPrime;

  filterRange(input,
    new cppnorth::FunctionModel<std::decay_t<decltype(isPrime)>>(isPrime));
```

```cpp
    FuncType mFunc;
  };

  struct IsEvenFunctor
  {
    bool operator()(int i) const { return !(i&1); }
  };
}

namespace myapp {
  using cppnort::filterRange;
  using thirdparty::isPrime;

  filterRange(input,
    new cppnorth::FunctionModel<std::decay_t<decltype(isPrime)>>(isPrime));

  filterRange(input,
    new cppnorth::FunctionModel<cppnorth::IsEvenFunctor>(IsEvenFunctor{}));

}
```

RECAP

RECAP

- `filterRange()` can be called using any callable wrapped in FunctionModel

RECAP

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

RECAP

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

RECAP

- `filterRange()` can be called using any callable wrapped in FunctionModel
- The client has to wrap function in a pointer - it's clunky
- Lifetime management
- No value semantics

RECAP

- `filterRange()` can be called using any callable wrapped in FunctionModel
- The client has to wrap function in a pointer - it's clunky
- Lifetime management
- No value semantics

Let's do the dirty work...

```cpp
struct Function
{
private:
  struct FunctionConcept
  {
    virtual bool operator()(int) = 0;
    virtual ~FunctionConcept() {}
  };

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

    bool operator()(int i) override { return mFunc(i); }

    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()(int i)
  {
    if (not mFunc)
      throw std::bad_function_call();
```

```
    virtual bool operator()(int) = 0;
    virtual ~FunctionConcept() {}
  };

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

    bool operator()(int i) override { return mFunc(i); }

    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()(int i)
  {
    if (not mFunc)
      throw std::bad_function_call();
    mFunc->operator()(i);
  }

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

```
    virtual bool operator()(int) = 0;
    virtual ~FunctionConcept() {}
  };

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

    bool operator()(int i) override { return mFunc(i); }

    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()(int i)
  {
    if (not mFunc)
      throw std::bad_function_call();
    mFunc->operator()(i);
  }

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

```cpp
struct Function
{
private:
  struct FunctionConcept
  {
    virtual bool operator()(int) = 0;
    virtual ~FunctionConcept() {}
  };

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

    bool operator()(int i) override { return mFunc(i); }

    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()(int i)
  {
    if (not mFunc)
      throw std::bad_function_call();
```

```cpp
struct Function
{
private:
  struct FunctionConcept
  {
    virtual bool operator()(int) = 0;
    virtual ~FunctionConcept() {}
  };

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

    bool operator()(int i) override { return mFunc(i); }

    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()(int i)
  {
    if (not mFunc)
      throw std::bad_function_call();
```

std::move_only_function

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

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

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

    bool operator()(int i) override { return mFunc(i); }

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

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

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

  bool operator()(int i)
  {
    if (not mFunc)
      throw std::bad_function_call();
    mFunc->operator()(i);
  }

  std::unique_ptr<FunctionConcept> mFunc;
};

std::vector<int> filterRange(
  std::vector<int> const& input,
  Function filterOn)
{
  ...
}

filterRange(input, Function(isOdd));
filterRange(input, Function(IsOddFunctor{}));
```

```
struct Function
{
private:
  struct FunctionConcept
  {
    virtual bool operator()(int) = 0;
    virtual std::unique_ptr<FunctionConcept> clone() const = 0;
    virtual ~FunctionConcept() {}
  };

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

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

    bool operator()(int i) override { return mFunc(i); }

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

Prototype design pattern

RECAP

RECAP

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

RECAP

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

RECAP

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

RECAP

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

RECAP

- `filterRange()` 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(int)

FINALLY...MORE TEMPLATES

```
template <typename T>
struct Function;

template <typename R, typename... Args>
struct Function<R(Args...)>
{
private:
  ...
public:
  template <std::invocable>Args...> FuncType>
    requires requires (FuncType t, Args... args) {
      { t(std::forward<Args>(args)...) } -> std::same_as<R>;
    }
  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 <std::invocable<Args...> FuncType>
    requires requires (FuncType t, Args... args) {
      { t(std::forward<Args>(args)...) } -> std::same_as<R>;
    }
  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 <std::invocable<Args...> FuncType>
    requires requires (FuncType t, Args... args) {
      { t(std::forward<Args>(args)...) } -> std::same_as<R>;
    }
  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 *Function*

Cost of *Function*

1. Heap allocation on construction

Cost of *Function*

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

Cost of *Function*

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

2. Virtual function calls

Cost of *Function*

1. Heap allocation on construction * small buffer optimisation (SBO)
2. Virtual function calls * manual virtual dispatch (MVD)

Small Buffer Optimisation

```
struct Function
{
  using StorageModelType = FunctionModel<bool(*)(int)>;
  inline static constexpr size_t StorageSize = sizeof(StorageModelType);

  ...

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

SBO - Constructor

```cpp
struct Function
{
  using StorageModelType = FunctionModel<bool(*)(int)>;
  inline static constexpr size_t StorageSize = sizeof(StorageModelType);

  template <typename FuncType>
  explicit Function(FuncType&& func)
  {
    using T = FunctionModel<std::decay_t<FuncType>>;
    if constexpr (sizeof(*new T()) <= StorageSize)
    {
      mFunc = std::construct_at(std::addressof(mStorage), T(std::forward<FuncType>(func)));
    }
    else
    {
      mFunc = new T(std::forward<FuncType>(func));
    }
  }

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

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];
};
```

shared_ptr and type safety

```cpp
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);
```

```
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!
```

```
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'

```
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!
```

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

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

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

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

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

```cpp
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

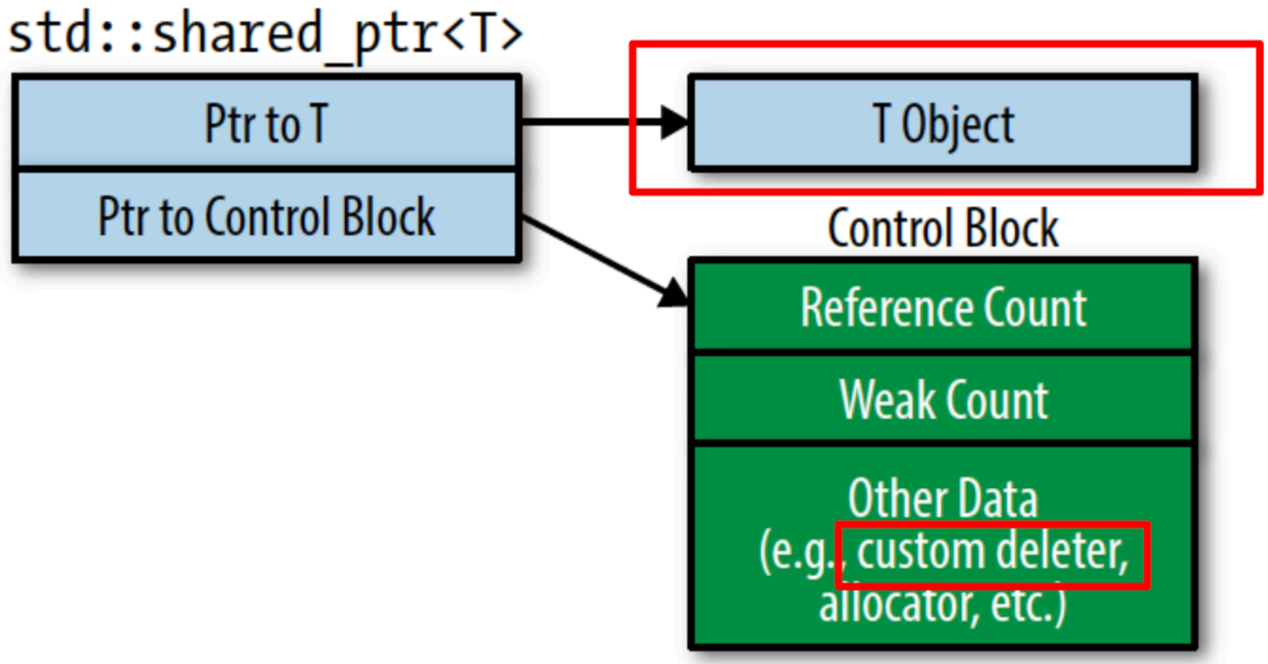
- unique_ptr<T> only stores the template type information

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;
};
```

```
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);
    }
};
```

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!
}
```

```cpp
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)}}
  {}
```

```cpp
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

```
class any
{
public:
  template <typename T>
  any(T&& value)
  : mValue{new T{std::forward<T>(value)}}
  , mType{typeid(T)}
  {}

  ~any() { ... }

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

private:
  void* mValue;
  std::type_index mType;
};
```

Type erasure without virtual

```
class any
{
public:
  template <typename T>
  any(T&& value)
  : mValue{new T{std::forward<T>(value)}}
  , mType{typeid(T)}
  {}

  ~any() { ... }

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

private:
  void* mValue;
  std::type_index mType;
};
```

```
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