Unveiling Type Erasure In C++

std::function | std::any

Sarthak Sehgal

C++ Software Engineer

- Working at a high frequency options market making firm
- Interested in finance, low level programming, and C++ under the hood
- sartech.substack.com
- LinkedIn://sarthaksehgal99

std::function

```
template<typename R, typename... Args>
class function<R(Args...)>;
```

Class template std::function is a general-purpose *polymorphic* function wrapper

```cpp
void print_num(int i)
{
    std::cout << i << '\n';
}

std::function<void(int)> f_display = print_num;
```

Free function

```
auto const display_func = [](int i){ std::cout << i; };
std::function<void(int)> f_display_lambda = display_func;
```

Lambda expression

```cpp
struct PrintNum
{
    void operator()(int i) const
    {
        std::cout << i << '\n';
    }
};

std::function<void(int)> f_display_obj = PrintNum{};
```

Function object (functor)

Million $$$ question

How are we able to assign completely unrelated types (free function, lambda expression, functor, etc) to a common type `std::function<void(int)>`

Requirements

1. Hold different types sharing a common interface ("generic")
2. Retain the type information of the assigned object ("type safe")

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

struct IsEvenFunctor : IFunction
{
    bool operator()(int a) override
    {
        return a%2 == 0;
    }
};

struct IsOddFunctor : IFunction
{
    bool operator()(int a) override
    {
        return a%2 != 0;
    }
};

void filterRange(std::vector<int>& inputVec, IFunction const& func)
{...}
```

✅ Generic

✅ Type Safe

Requirements (again)

1. Hold ~~different~~ *unrelated* (or related) types sharing a common interface ("generic")
2. Retain the type information of the assigned object ("type safe")

```cpp
// holds all function like objects implementing `void operator()(int)`
struct MagicFunctionContainer
{};

MagicFunctionContainer f1 = print_num;
MagicFunctionContainer f2 = [](int i) { std::cout << i; };
MagicFunctionContainer f3 = PrintNumFunctor{};
```

```cpp
struct MagicFunctionContainer
{
    template <typename Func>
    MagicFunctionContainer(Func&& func)
    : mVal{std::forward<Func>(func)}
    {}

    void operator()(int i) const
    {
        mVal(i);
    }

private:
    ?? mVal;
};
```

Constructible from unrelated types implementing the common interface

```cpp
struct Concept
{
    virtual void operator()(int) = 0;
    virtual ~Concept() = default;
};

template <typename T>
struct Model : Concept
{
    Model(T const& val): mVal{val} {}

    void operator()(int i) override
    {
        mVal(i);
    }
private:
    T mVal;
};
```

Any* type T implementing operator()(int) can be stored in Model<T>

Model<T> inherits from Concept

```cpp
struct Concept
{
    virtual void operator()(int) = 0;
    virtual ~Concept() = default;
};

template <typename T>
struct Model : Concept
{
    Model(T const& val): mVal{val} {}

    void operator()(int i) override
    {
        mVal(i);
    }
private:
    T mVal;
};
```

```cpp
struct MagicFunctionContainer
{
    template <typename Func>
    MagicFunctionContainer(Func&& func)
    : mFunc{new Model<Func>(func)}
    {}

    void operator()(int i)
    {
        if (not mFunc)
            throw std::bad_function_call();
        (*mFunc)(i);
    }

private:
    Concept* mFunc = nullptr;
};
```


lafuddyduddy
DRUM ROLL PLEASE

```cpp
template <typename T>
class function;

template <typename R, typename... Args>
class function<R(Args...)>
{
public:
    template <std::invocable<Args...> T>
        requires requires (T t, Args... args) {
            { t(std::forward<Args>(args)...) } -> std::same_as<R>;
        }
    function(T&& func)
    {
        mFunc = std::make_unique<Model<T, R, Args...>>(func);
    }

    R operator()(Args... args)
    {
        return (*mFunc)(std::forward<Args>(args)...);
    }

private:
    std::unique_ptr<Concept<R, Args...>> mFunc;
};
```

PS: Not standard compliant

Type Erasure

Type erasure enables you to use various concrete unrelated types through a single generic interface

Cost of std::function *

1. Heap allocation on construction
2. Virtual function calls

Requirements

1. Ability to hold unrelated types sharing a common interface ("generic")
2. Retain the type information of the assigned object ("type safe")

Type safety in type erasure

```
void* fooPtr = new NoisyFoo();
delete fooPtr;
```

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

Type safety in type erasure

```
void* fooPtr = new NoisyFoo();
delete fooPtr;
```

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

deleting void is undefined behaviour as per standard*

Type safety in type erasure

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

Type safety in type erasure

```
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'

Type safety in type erasure

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

Type safety in type erasure

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

Constructed NoisyFoo
Destructed NoisyFoo

Type safety in type erasure

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

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

```cpp
template<class T> class shared_ptr;
```

```cpp
std::shared_ptr<void> ptr1(new NoisyFoo());
std::shared_ptr<void> ptr2(new NoisyBar());
ptr1 = ptr2;
```

```cpp
template <typename T>
class shared_ptr
{
public:
    shared_ptr(T* ptr)
    : ptr{ptr}
    {};

    ~shared_ptr()
    {
        delete(ptr);
    }

private:
    T* ptr;
    int* refCount = new int(1);
};
```

```cpp
template <typename T>
class shared_ptr
{
public:
    shared_ptr(T* ptr)
    : ptr{ptr}
    {};

    ~shared_ptr()
    {
        delete(ptr);
    }

private:
    T* ptr;
    int* refCount = new int(1);
};
```

shared_ptr<Base>(new Derived())

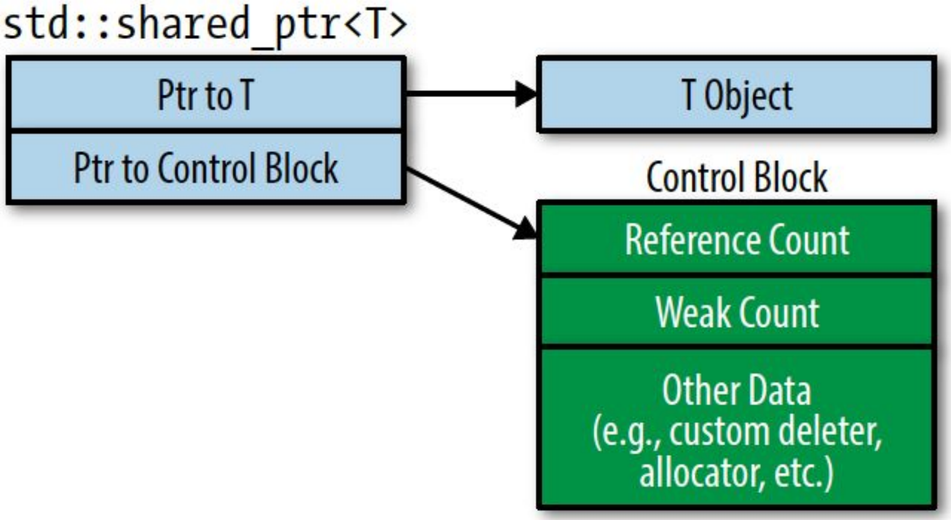
```cpp
template <typename T>
class shared_ptr
{
public:
    shared_ptr(T* ptr)
    : ptr{ptr}
    {};

    ~shared_ptr()
    {
        delete(ptr);
    }

private:
    T* ptr;
    int* refCount = new int(1);
};
```

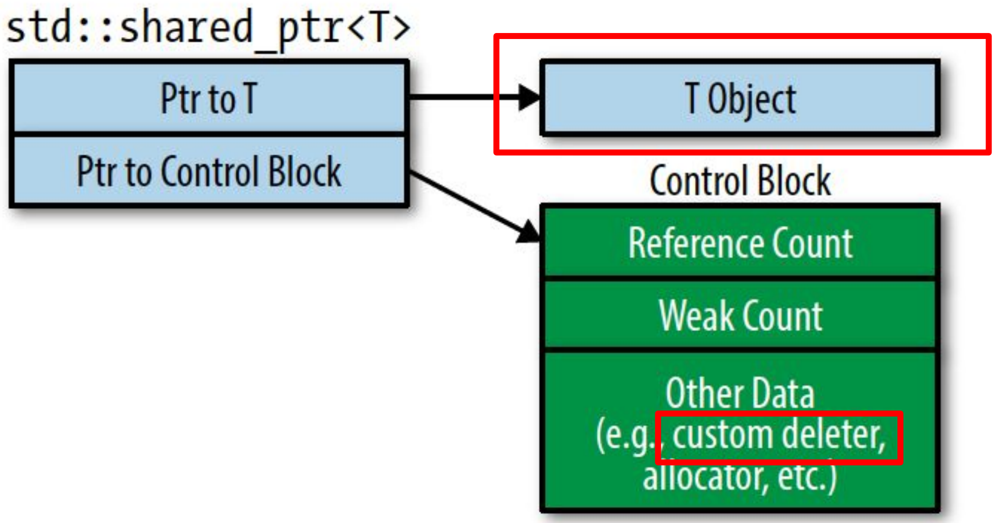
shared_ptr<Base>(new Derived())

shared_ptr needs to be aware of the actual type passed during construction in order to delete it correctly

Type Erasure in shared_ptr

- shared_ptr<T1> can store objects of any type T2 as long as T2* is convertible to T1*. It *type erases* T2
- shared_ptr will call the correct destructor, ie, the destructor corresponding to the type of object stored


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.)


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>
    shared_ptr(Y* ptr)
    : ptr{ptr}
    {};

    ~shared_ptr()
    {
        delete(ptr);
    }

private:
    T* ptr;
    int* refCount = new int(1);
};
```

```cpp
template <typename T>
struct 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>
    shared_ptr( Y* ptr, Deleter d )
    : mPtr{ptr}
    // need a way to store Y* ptr and Deleter d
    // for any types Y and Deleter
    {}

private:
    T* mPtr;
};
```

```cpp
template <typename T>
struct 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>
    shared_ptr( Y* ptr, Deleter d )
    : mPtr{ptr}
    // need a way to store Y* ptr and Deleter d
    // for any types Y and Deleter
    {}

private:
    T* mPtr;
};
```

Store *concrete* types Y* and Deleter in a *generic* interface

```cpp
template <typename T>
struct 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>
    shared_ptr( Y* ptr, Deleter d )
    : mPtr{ptr}
    , mControlBlock{new ControlBlock<Y, Deleter>{ptr, std::move(d)}}
    {}

private:
    T* mPtr;
    ControlBlockBase* mControlBlock;
};
```

```cpp
class ControlBlockBase
{
    std::size_t refCount = 1;
    virtual ~ControlBlockBase() = default;
};

template <typename ObjType, typename Deleter>
class ControlBlock : ControlBlockBase
{
    ObjType* ptr;
    Deleter deleter;

    ControlBlock(ObjType* ptr, Deleter deleter)
    : ptr(ptr)
    , deleter(std::move(deleter))
    {}

    virtual ~ControlBlock() override
    {
        deleter(ptr);
    }
};
```

shared_ptr with a type erased deleter. Notice that two shared_ptr with same managed object type but different deleter type can be assigned to each other

```cpp
template <typename T>
struct 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>
    shared_ptr( Y* ptr, Deleter d )
    : mPtr{ptr}
    , mControlBlock{new ControlBlock<Y, Deleter>{ptr, std::move(d)}}
    {}

private:
    T* mPtr;
    ControlBlockBase* mControlBlock;
};
```

```cpp
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);
    }
};
```

shared_ptr with a type erased deleter. Notice that two shared_ptr with same managed object type but different deleter type can be assigned to each other

std::any

```cpp
class any;
```

The class any describes a container for single values of any type

```cpp
std::any a = 1;
a = 3.14;
a = true;
```

Modern void*

```cpp
class any
{
private:
    void* data;

public:
    template <typename T>
    explicit any(T&& value)
    : data{new std::decay_t<T>{value}}
    {}
};
```

```
class any;
```

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

```cpp
std::any a = 1;
std::cout << a.type().name() << ": " << std::any_cast<int>(a); // int: 1

a = 3.14;
std::cout << a.type().name() << ": " << std::any_cast<double>(a); // double: 3.14

a = true;
std::cout << a.type().name() << ": " << std::any_cast<bool>(a); // bool: true
```

```cpp
// bad cast
try
{
    a = 1;
    std::cout << std::any_cast<float>(a) << '\n';
}
catch (const std::bad_any_cast& e)
{
    std::cout << e.what() << '\n';
}
```

```
// reset
a.reset();
assert(!a.has_value())
```

```cpp
class any
{
    void* data;
    std::type_index typeInfo;

public:
    template<typename T>
    explicit any(T&& value)
        : data{new std::decay_t<T>{std::forward<T>(value)}}
        , typeInfo{typeid(T)}
    {}

    template <typename T>
    any& operator=(T&& value)
    {
        data = new T{std::forward<T>(value)};
        typeInfo = typeid(T);
        return *this;
    }

    any& operator=(any const&) = default;

    template<typename T>
    friend T& any_cast(any& a);
};
```

```cpp
template<typename T>
T& any_cast(any& a)
{
    if (typeid(T) == a.typeInfo)
    {
        return *static_cast<T*>(a.data);
    }
    else
    {
        throw std::bad_any_cast{};
    }
}

any a(1);
any b(1.2);
any c("abc");

int d = any_cast<int>(a); // success
double e = any_cast<double>(a); // throws
```

```cpp
class any
{
    void* data;
    std::type_index typeInfo;

public:
    template<typename T>
    explicit any(T&& value)
        : data{new std::decay_t<T>{std::forward<T>(value)}}
        , typeInfo{typeid(T)}
    {}

    template <typename T>
    any& operator=(T&& value)
    {
        data = new T{std::forward<T>(value)};
        typeInfo = typeid(T);
        return *this;
    }

    any& operator=(any const&) = default;

    template<typename T>
    friend T& any_cast(any& a);
};
```

```cpp
template<typename T>
T& any_cast(any& a)
{
    if (typeid(T) == a.typeInfo)
    {
        return *static_cast<T*>(a.data);
    }
    else
    {
        throw std::bad_any_cast{};
    }
}

any a(1);
any b(1.2);
any c("abc");

int d = any_cast<int>(a); // success
double e = any_cast<double>(a); // throws
```

```cpp
class any
{
    using CloneFunc = std::function<void*(void*)>;
    using DeleteFunc = std::function<void(void*)>;

    void* data;
    std::type_index type_info;
    CloneFunc clone;
    DeleteFunc deleter;

public:
    template <typename T>
        requires (!std::same_as<std::decay_t<T>, any>)
    any(T&& value)
        : data{new T{std::forward<T>(value)}}
        , type_info{typeid(T)}
        , clone([](void* other) -> void* { return new T(*static_cast<T*>(other)); })
        , deleter([](void* ptr) { delete static_cast<T*>(ptr); })
    {}

    any(any const& other)
    : type_info(other.type_info)
    , clone(other.clone)
    , deleter(other.deleter)
    {
        deleter(data);
        data = other.clone(other.data);
    }

    ~any()
    {
        deleter(data);
    }
};
```

Questions

Further reading

- Andrzej's blogs on type erasure
- std::function small buffer optimisation
- void* based type erasure