C++ IDIOMS

SFINAE, PIMPL

Created by Łukasz Ziobroń

OVERLOAD RESOLUTION

```cpp
void foo(unsigned i) {
    std::cout << "unsigned " << i << "\n";
}

template <typename T>
void foo(const T& t) {
    std::cout << "template " << t << "\n";
}
```

What is the result of calling `foo(42)`?

`template 42`

Why `const int &` is a better match than `unsigned`?

Const reference can bind to r-values. No conversion is needed, there is an exact match, so this option is chosen.

OVERLOAD RESOLUTION

```
void foo(unsigned i) {
    std::cout << "unsigned " << i << "\n";
}

template <typename T>
void foo(T& t) {
    std::cout << "template " << t << "\n";
}
```

What is the result of calling `foo(42)` now?

`unsigned 42`

Why `unsigned` is a better match than `int &`?

Reference (non-const) cannot bind to r-values. There is only one function matching. Implicit conversion from int to unsigned is applied.

OVERLOAD RESOLUTION

```cpp
void foo(unsigned i) {
    std::cout << "unsigned " << i << "\n";
}

void foo(double i) {
    std::cout << "double " << i << "\n";
}
```

What is the result of calling `foo(42)`?

`error: call of overloaded ‘foo(int)’ is ambiguous`

Why?

Promotion to double and conversion to unsigned are equally viable.

SIMPLE SFINAE EXAMPLE

```
template <typename T>
void foo(T arg) {}

template <typename T>
void foo(T* arg) {}
```

Calling `foo(42)` makes a compiler to generate two functions.

```
void foo(int arg) {}
void foo(int* arg) {}
```

Compiler cannot substitute `42` as an argument to the second function. It would cause a compilation error. Therefore this overload is discarded.

There is no compilation error - this is SFINAE works.

If the first funtion will be missing, there would be a compilation error.

SFINAE

Substitution Failure Is Not An Error is a meta-programming technique.

> *"This rule applies during overload resolution of function templates: When substituting the explicitly specified or deduced type for the template parameter fails, the specialization is discarded from the overload set instead of causing a compile error."*
>
> -- *cppreference.com*

Rationale: have a universal interface without letting the caller to decide which implementation should be called. Selection of an optimal implementation is done by a compiler and is coded by a library creators.

`std::enable_if`

C++11 has a metaprogramming helper struct - `std::enable_if`. It is a compile-time switch for enabling or disabling some templates.

```
template <bool Condition, class T = void>
struct enable_if {};

template <class T>
struct enable_if<true, T> { using type = T; };
```

- If `Condition` is `true`, accessing internal type by `enable_if<Condition, T>::type` is valid.
- If `Condition` is `false`, accessing internal type by `enable_if<Condition, T>::type` is invalid and substitution is not correct - SFINAE works.

std::enable_if_t

C++14 defines a helper type:

```
template <bool B, class T = void>
using enable_if_t = typename enable_if<B, T>::type;
```

Using both is equivalent.

```
template <class T,
          typename std::enable_if<std::is_integral<T>::value,
                                  T>::type* = nullptr>
void function(T& t) {}


template <class T,
          typename std::enable_if_t<std::is_integral_v<T>,
                                    T>* = nullptr>
void function(T& t) {}
```

Why `* = nullptr`?

enable_if OCCURENCES

```
template<class T>      // #1 return type
auto construct(T* t) ->
    typename std::enable_if_t<std::has_virtual_destructor_v<T>, T>*
{ return new T{}; }
```

```
template<class T>      // #2 parameter
T* construct(
    T* t,
    typename std::enable_if_t<std::has_virtual_destructor_v<T>>* = nullptr
){ return new T{}; }
```

```
template<
    class T,           // #3 template parameter - usual choice from C++11
    typename std::enable_if_t<std::has_virtual_destructor_v<T>>* = nullptr
>
T* construct(T* t)
{ return new T{}; }
```

`enable_if` OCCURENCES

The most elegant way

```
template <typename T>
using HasVirtDtor = std::enable_if_t<std::has_virtual_destructor_v<T>>;

template<
    class T,            // the same as #3 - template parameter
    typename = HasVirtDtor<T>
>
T* construct(T* t)
{ return new T{}; }
```

Standard library header <type_traits>

This header is part of the type support library.

Classes

Helper Classes

integral_constant (C++11) **bool_constant** (C++17)	compile-time constant of specified type with specified value (class template)
true_type	`std::integral_constant<bool, true>`
false_type	`std::integral_constant<bool, false>`

Primary type categories

is_void (C++11)	checks if a type is `void` (class template)
is_null_pointer (C++14)	checks if a type is `std::nullptr_t` (class template)
is_integral (C++11)	checks if a type is an integral type (class template)
is_floating_point (C++11)	checks if a type is a floating-point type (class template)
is_array (C++11)	checks if a type is an array type (class template)
is_enum (C++11)	checks if a type is an enumeration type (class template)
is_union (C++11)	checks if a type is an union type (class template)
is_class (C++11)	checks if a type is a non-union class type (class template)
is_function (C++11)	checks if a type is a function type (class template)
is_pointer (C++11)	checks if a type is a pointer type (class template)
is_lvalue_reference (C++11)	checks if a type is a *lvalue reference* (class template)
is_rvalue_reference (C++11)	checks if a type is a *rvalue reference* (class template)
is_member_object_pointer (C++11)	checks if a type is a pointer to a non-static member object (class template)
is_member_function_pointer (C++11)	checks if a type is a pointer to a non-static member function (class template)

Composite type categories

is_fundamental (C++11)	checks if a type is a fundamental type (class template)
is_arithmetic (C++11)	checks if a type is an arithmetic type (class template)

type_traits LIBRARY

<type_traits> on cppreference.com

TASK

Write a function that allows inserting only subclasses of Shape to the collection. Other parameter types should not compile. Use SFINAE. Find proper type_traits.

Hints:

- std::is_base_of
- std::remove_reference
- std::remove_cv

PIMPL

A.K.A. COMPILATION FIREWALL

Rationale: avoiding long recompilation times, minimizing dependencies.

PIMPL stands for "pointer to implementation".

It is a dependency breaking technique, that allows to avoid recompilation of translation units utilizing our class. If internal parts of class changes very often and recompilation takes a lot of time it is worth considering PIMPL.

BEFORE

```
class Foo {                   // foo.hpp header file
    int internalData = 0;     // private part changes often
    int doWork(int);          // fields, functions signatures change
public:
    Foo();
    int interface(int);       // interface is stable, doesn't change often
};
```

```
#include "foo.hpp"

int Foo::doWork(int value) {
    internalData = value;
    return internalData;
}
Foo::Foo() {}
int Foo::interface(int value) { return doWork(value); }
```

Problem?

When a header file changes, every file that includes it, will need to be recompiled.

AFTER - PIMPL

```cpp
#include <memory>      // foo.hpp header file
class Foo {
    class Impl;
    std::unique_ptr<Impl> pimpl;
public:
    Foo();
    ~Foo();
    int interface(int);
};
```

```cpp
#include "foo.hpp"
class Foo::Impl {    // foo.cpp - implementation file
    int internalData = 0;
public:
    int doWork(int value) {
        internalData = value;
        return internalData;
    }
};
Foo::Foo() : pimpl{std::make_unique<Impl>()} {}
Foo::~Foo() = default;  // must be explicitly defined in cpp file
int Foo::interface(int value) { return pimpl->doWork(value); }
```

PIMPL - IMPLEMENTATION

- All private non-virtual functions are placed in the implementation class
- All public, protected and virtual members remain in the interface class
- If any private member needs to access a public or protected part, a reference or a pointer should be passed to the private function as a parameter.
- Use `std::unique_ptr` as a pointer to implementation
- Define a destructor in cpp file - it needs to see the complete definition of Impl class to destroy `std::unique_ptr`
- Move operations also needs to be implemented in cpp file (`= default` is usually good solution)
- Copy operations needs to be implemented by hand in cpp file

PIMPL - DISADVANTAGES

- More code
- Access overhead - one additional level of indirection
- Space overhead - one pointer in a public interface, optionally additional parameters needs to be passed to functions
- Memory management overhead - pimpl is placed on the heap, possible memory fragmentation
- Const correctness - how to propagate const to private implementation? std::experimental::propagate_const

Bartek Filipek article on PIMPL

C++ IDIOMS

- SFINAE
- Overload resolution rules
- enable_if
- enable_if_t
- type_traits
- constexpr if
- PIMPL