The Type-based Dispatch Challenge

- Assume we use many types in our program, from different libraries
  - Some implement an output stream operator for printing
  - Others have a `print_to` member function
- We want to implement a function that can deal with all of them
  - E.g. for debugging or logging

We'll investigate this problem in **06_07_dispatch.cpp**

What is this Sorcery?

```
77
78  template<typename T, class = void>
79  struct has_print_to : public std::false_type {};
80
81  template<typename T>
82  struct has_print_to<T,
83      std::void_t<decltype(std::declval<T>().print_to(std::declval<std::ostream&>()))>
84  > : public std::true_type {};
85
```

- It’s rather dense, but each component can be understood easily in isolation

std::declval<T>()

- `std::declval<T>()` returns an instance of type T
  - An r-value by default, can generate an l-value by using `declval<T&>()`
- Can even be used if the type has no (publicly available) default constructor
- *It doesn't have an implementation/definition*
  - Specifically designed to be used in **unevaluated contexts**
  - E.g. in template metaprogramming

So what is an unevaluated context?

Unevaluated Contexts

- A context in which an expression is not actually evaluated (i.e. executed)
- 4 cases:
  - **`sizeof`**`(expr)` – oldest, we know this one
  - **`noexcept`**`(expr)` – checks whether *expr* can throw an exception
  - **`typeid`**`(expr)` – yields a std::type_info object for the type of *expr*
    NOTE: only unevaluated if there is no polymorphism!
  - **`decltype`**`(expr)` – The type of the expression *expr*

Examples for decl* in **06_08_decltype_declval.cpp**

Putting it into Practice

```
77
78  template<typename T, class = void>
79  struct has_print_to : public std::false_type {};
80
81  template<typename T>
82  struct has_print_to<T,
83      std::void_t<decltype(std::declval<T>().print_to(std::declval<std::ostream&>()))>
84  > : public std::true_type {};
85
```

- We now understand this part: *in an unevaluated context, an instance of T is created, and "print_to" is called on it with an l-value of type std::ostream*
- **But why?**

The Selection Mechanism

```
77
78  template<typename T, class = void>
79  struct has_print_to : public std::false_type {};
80
81  template<typename T>
82  struct has_print_to<T,
83      std::void_t<decltype(std::declval<T>().print_to(std::declval<std::ostream&>()))>
84  > : public std::true_type {};
85
```

- We are again using the “more specialized” template mechanism to make case distinctions
- But with an additional twist: **SFINAE**

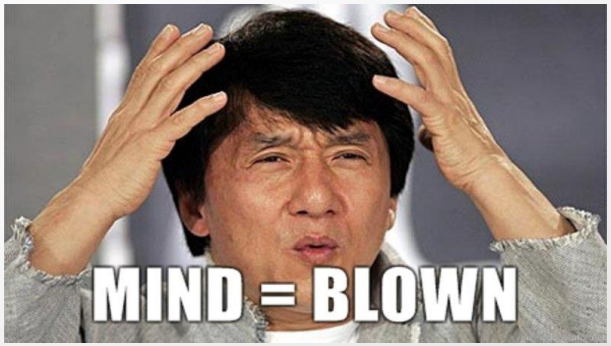
Substitution **F**ailure **I**s **N**ot **A**n **E**rror

- Principle that applies when the compiler instantiates templates (and their specializations)
- When substitution of a template Parameter leads to erroneous code, that particular specialization is *removed from the overload set*, rather than creating a compiler error
- The general usage is to **create an error on purpose in some circumstances to remove an implementation from the overload set**

Example in **06_09_sfinae.cpp**

So what is `std::void_t`?

```
template< class... >
using void_t = void;
```



"This metafunction is used in template metaprogramming to detect ill-formed types in SFINAE context" - cppreference

→ If any of the argument types is ill-formed, it's an error
→ Otherwise, it is simply a fancy way to write "void"

```
77
78  template<typename T, class = void>
79  struct has_print_to : public std::false_type {};
80
81  template<typename T>
82  struct has_print_to<T,
83      std::void_t<decltype(std::declval<T>().print_to(std::declval<std::ostream&>()))>
84  > : public std::true_type {};
85
```

Note: the second template parameter needs to default to `void` for the specialization to work

- **If** the type T has a member function with the desired signature
  - `void_t<…>` in the specialization is `void`
  - The specialization is less general and is chosen → **`true_type`**
- **otherwise**,
  - The type passed to `void_t` is ill-formed → `void_t` causes an error
  - The specialization is dropped due to *SFINAE*
  - The primary template is chosen → **`false_type`**