1 Introduction

SFINAE can be used to choose appropriate overloads of function templates based on properties of the passed in type(s).

We have to make sure that any given type can at most match one of these function templates, otherwise we will get ambiguity. In the example below both overloads of `foo` work if `T = std::string`, so the compiler cannot decide which version to use, thus giving us a compile error.

```
template <typename T>
auto foo(T&& t) -> decltype(t.size())
{
  return t.size();
}

template <typename T>
auto foo(T&& t) -> decltype(t.length())
{
  return t.length();
}

int main()
{
  string str{"Hello"};
  cout << foo(str) << endl;
}
```

A solution to this problem would be to induce some kind of priority between the different overloads. The recommended way to do this is to leverage the overload resolution rules.

Recall that whenever the compiler sees `foo(t)` it will first find all functions named `foo` which can be called with one parameter of type `T` (the type of `t`), and then decide which one to call with this priority:

1. An ordinary function that match the parameters exactly,
2. A function template that does not require any conversions,
3. An ordinary function where some type conversions are required for the parameter to match;
4. A function templates that requires conversions.

At each step the compiler will choose the *closest matching* function, i.e. the function with the least amount of conversions required. So for example `fun(0, 0)` would match `void fun(T, int)` before it matches `void fun(T, double)` since the second overload would require `0` to be converted from `int` to `double`.

We can leverage this to construct our own ordering of `foo` by adding parameters that uses conversions so the compiler chooses a closer matching overload. We can do it like this:

```
// if substituting T succeeds for this overload
// the compiler is guaranteed to pick it since the
// second parameter won't trigger a conversion
template <typename T>
auto foo_helper(T&& t, int) -> decltype(t.size())
{
  return t.size();
}

// if substituting T succeeds for this overload
// the compiler will only pick this if the first
// overload fails, since this requires 0 to be
// converted to long.
template <typename T>
auto foo_helper(T&& t, long) -> decltype(t.length())
{
  return t.length();
}

template <typename T>
auto foo(T&& t)
{
  // second argument is 0, a value of type int
  // in order to control the overload resolution
  // of foo_helper.
  return foo_helper(std::forward<T>(t), 0);
}
```

2 The Exercise

In this exercise you are to use this technique to create functions that call specific member functions of class types based on what functions exists in the type.

You are to implement a function template `get_priority` that takes one parameter `t` of type `T`. The behaviour will depend on `t`, given by:

1. If `t` has a member function named `first`, then `get_priority` should return the result of `t.first()`,
2. If `t` has a member function named `second`, then `get_priority` should return the result of `t.second()`,
3. If `t` has a member function named `third`, then `get_priority` should return the result of `t.third()`,
4. If `t` has none of the above, it should return the value `4`.

Do note that these are not disjoint conditions, `t` might fulfil more than one of these conditions at once; the numberings of the list denotes the priority, e.g. if `t` has member functions `second` and `third` then `get_priority(t)` should return the result of `t.second()`.

In `given_files/priority.cc` there are testcases given; all of which should work without having to modify anything except `get_priority` (additions to the code is however allowed).

Hint: Create a set of helper function overloads called `priority_helper` and use extra parameters and conversions to force a priority on the overloads (more than one extra parameter is probably needed).