For example, consider the task of writing a ~~generic~~ `sortRange` function template that, given a **range**, either invokes the `sort` member function of the argument (the one specifically optimized for that type) if available or falls back to the more general `std::sort`:

```
template <typename Range>
void sortRange(Range& range)
{
    sortRangeImpl(range, 0);
}
```

The client-facing `sortRange` function in the example above delegates its behavior to an **overloaded** `sortRangeImpl` function in the example below, invoking the latter with the `range` and a **disambiguator** of type **`int`**. The type of this additional **parameter**, whose value is arbitrary, is used to give priority to the `sort` member function at compile time by exploiting **overload resolution** rules in the presence of an implicit, *standard* conversion from **`int`** to **`long`**:

```
template <typename Range>
void sortRangeImpl(Range& range, long)  // low priority: standard conversion
{
    // fallback implementation
    std::sort(std::begin(range), std::end(range));
}
```

The fallback overload of `sortRangeImpl` in the code snippet above will accept a **`long`** disambiguator, requiring a **standard conversion** from **`int`**, and will simply invoke `std::sort`. The more specialized overload of `sortRangeImpl` in the code snippet below will accept an **`int`** disambiguator requiring no conversions and thus will be a better match, provided a ~~range-specific~~ sort is available:

```
template <typename Range>
void sortRangeImpl(Range& range,
                   int,                          // high priority: exact match
                   decltype(range.sort())* = 0)  // check expression validity
{
    // optimized implementation
    range.sort();
}
```

Note that, by exposing **`decltype`**`(range.sort())` as part of `sortRangeImpl`'s declaration, the more specialized overload will be discarded during template substitution if `range.sort()` is not a valid expression for the deduced `Range` type.[3]

[3]The technique of exposing a possibly unused unevaluated expression — e.g., using **`decltype`** — in a function's declaration for the purpose of expression-validity detection prior to **template instantiation** is commonly known as **expression SFINAE**, which is a restricted form of the more general, classical SFINAE, and acts exclusively on expressions visible in a function's **signature** rather than on frequently obscure template-based type computations.