Function overloading is more flexible (and more convenient) than template function specialization

devblogs.microsoft.com/oldnewthing/20250410-00/?p=111063

April 10, 2025

A colleague of mine was having trouble specializing a templated function. Here's a simplified version.

```
template<typename T, typename U>
bool same_name(T const& t, U const& u)
{
    return t.name() == u.name();
}
```

They wanted to provide a specialization for the case that the parameters are a Widget and a string literal.

```
template<>
bool same_name<Widget, const char[]>(Widget const& widget, const char name[])
{
    return strcmp(widget.descriptor().name().c_str(), name) == 0;
}
```

However, this failed to compile:

```
// msvc
error C2912: explicit specialization 'bool same_name<Widget,const char[]>(const
Widget &,const char [])' is not a specialization of a function template
```

What do you mean "not a specialization of a function template"? I mean doesn't it look like a specialization of a function template? It sure follows the correct syntax for a function template specialization.

The error message from gcc is a little more helpful:

```
error: template-id 'same_name<Widget, const char []>' for 'bool same_name(const
Widget&, const char*)' does not match any template declaration
```

Okay, so gcc recognized that it's a specialization of a function template, but it couldn't find a match. What is this "match" talking about?

The error message from clang helps even more:

```
error: no function template matches function template specialization 'same_name'
| bool same_name<Widget, const char[]>(Widget const& widget, const char name[])

note: candidate template ignored: could not match 'bool (const Widget &, const
const char (&)[])' against 'bool (const Widget &, const char *)'
```

Okay, now we're getting somewhere. The compiler is taking the specialization we provided and is unable to match it against the non-specialized version.

And that's where we see the problem. If we substitute `Widget` and `const char[]` into the original declaration of `bool same_name(T const& t, U const& u)`, we get

```
    bool same_name(Widget const& t, const char(& u)[]);
```

But this isn't the function signature of our proposed specialization. Our proposed specialization takes a `const char*` as the final parameter, since function and array parameters in parameter lists are rewritten as pointers: **[dcl.fct]**(4): "any parameter of type 'array of `T`' or of function type T is adjusted to be 'pointer to `T`'."

That's what msvc was trying to tell us when it said "is not a specialization of a function template": "What you wrote sure looks like a specialization of a function template, but it's not because the signature is wrong." Perhaps a better message would be "is not a *valid* specialization of a function template" or "does not *correspond to* a specialization of a function template."

A valid specialization would be

```
template<>
bool same_name<Widget, const char*>(Widget const& widget, const char *const& name)
{
    return strcmp(widget.descriptor().name().c_str(), name) == 0;
}
```

That sure looks clunky, but it doesn't have to be.

You don't need to do specialization at all: You can use overloading.

```
bool same_name(Widget const& widget, const char* name)
{
    return strcmp(widget.descriptor().name().c_str(), name) == 0;
}
```

The nice thing about overloading is that you don't have to be a perfect match for the original template. Here, we take the second parameter by value instead of by reference. You can even change the return value in an overload.

```
std::optional<bool> same_name(Widget const& widget, const char* name)
{
    if (!widget.is_name_known()) return std::nullopt;
    return strcmp(widget.descriptor().name().c_str(), name) == 0;
}
```

In this case, we change the return type from `bool` to `std::optional<bool>` to be able to express the “I don’t know” case.

Function templates cannot be partially specialized, but that’s okay: You can get the same effect via overloading.

```
template<typename U>
std::optional<bool> same_name(Widget const& widget, U const& u)
{
    if (!widget.is_name_known()) return std::nullopt;
    return widget.name() == u.name();
}
```

	Class	**Function**
Can template	Yes	Yes
Can specialize	Yes	Yes
Can partially specialize	Yes	No
Can overload	No	Yes

Template functions: Don’t specialize them. Overload them.