Writing a helper class for generating a particular category of C callback wrappers around C++ methods

devblogs.microsoft.com/oldnewthing/20250616-00/?p=111271

June 16, 2025

A common pattern for C callbacks is to accept a function pointer and a `void*`, and then the callback function receives that pointer in addition to the parameters specific to the callback.

```
// Hypothetical callback
typedef int (*callback_t)(
    void* context,
    int arg1, char const* arg2, double arg3);

void RegisterCallback(callback_t callback, void* context);
```

If you're writing code in C++, it's more convenient to write the callback as a class member function.

```
struct Widget
{
    int OnCallback(
        int arg1, char const* arg2, double arg3);

    static int OnCallbackStatic(void* context,
        int arg1, char const* arg2, double arg3)
    {
        auto self = (Widget*)context;
        return self->OnCallback(arg1, arg2, arg3);
    }

    void Register()
    {
        RegisterCallback(OnCallbackStatic, this);
    }
};
```

Is there a way to simplify this boilerplate?

My idea was to use the conversion operator to deduce the callback function signature, and give the wrapper function the same signature. The wrapper function then forwards the parameters to the member function and propagates the result. Doing it this way

means that the member function need not have the exact same signature, as long as the inputs and outputs are convertible.

```
template<auto F>
struct CallbackWrapperMaker
{
    template<typename Ret, typename...Args>
    static Ret callback(void* p, Args...args)
        { auto obj = (???*)p;
          return (obj->*F)((Args)args...); }

    template<typename Ret, typename...Args>
    using StaticCallback = Ret(*)(void*, Args...);

    template<typename Ret, typename...Args>
    operator StaticCallback<Ret, Args...>()
        { return callback<Ret, Args...>; }
};

template<auto F>
inline CallbackWrapperMaker<F> CallbackWrapper =
    CallbackWrapperMaker<F>();

struct Widget
{
    int OnCallback(
        int arg1, char const* arg2, double arg3);

    void Register()
    {
        RegisterCallback(
            CallbackWrapper<&Widget::OnCallback>,
            this);
    }
};
```

// usage:

The `CallbackWrapper` is an empty object whose job is merely to be convertible to a C-style callback function. We use the conversion operator to detect the signature that we need to produce, and we use those template type parameters to generate the correct version of the `callback` function.

The part we haven't written yet is the code to cast the context parameter `p` back to the original object type, like `Widget`. To do that, we use a helper traits type that can extract the parent object from a pointer to member function.

```
template<typename F> struct MemberFunctionTraits;

template<typename Ret, typename T, typename...Args>
struct MemberFunctionTraits<Ret(T::*)(Args...)>
{
    using Object = T;
};


template<auto F>
struct CallbackWrapperMaker
{
    template<typename Ret, typename...Args>
    static Ret callback(void* p, Args...args)
        { auto obj = (typename MemberFunctionTraits<decltype(F)>::
                                  Object*)p;
          return (obj->*F)((Args)args...); }

    ⟦ ... rest as before ... ⟧
};
```

The nice thing about forwarding to the member function is that the member function need not accept the parameters in the same way as the callback. For example, we could have written

```
struct Widget
{
    short OnCallback(
        long arg1, char const* arg2, double arg3);

    void Register()
    {
        RegisterCallback(
            CallbackWrapper<&Widget::OnCallback>,
            this);
    }
};
```

and the compiler will automatically apply the normal integral promotions of `arg1` from `int` to `long` and the return value from `short` to `int`.