Pointers for Function Types

Adventures in the C++ type system

Tony Thompson

tony.thompson@sig.com

SIG SUSQUEHANNA

About Me

- C++ since 2013, industry since 2016, SIG since 2017
  - High energy physics data acquisition/analysis, radar systems, monitoring, library design
- Software Developer on SMI team in Enterprise Technology
  - LibSMI (Cross-platform C++ messaging middleware library with C# and Python bindings)
  - SMI Agent (running on ~9200 SIG servers/desktops)
  - SMI AlertEngine, SMI Ganglia Adapter, SMI Router, SMI JsonAdapter, etc.
  - Maintainer/user of Jolt (STL/boost extensions for SIG), UnitTest libraries in plus toolchains
- Resident elasticsearch/Kibana power user

What integral types involve functions?

- Functions
- Pointers to functions
- References to functions
- Pointers to member functions

Why use function types?

- Custom callback logic
- Library implementations of higher level function objects
  - std::function
  - inplace_function (non-allocating std::function)
  - function_view (non-owning generic callable)
  - Boost.Signals2 (Callback management library)
  - ...
- Type traits for function types
  - std::is_function
  - std::is_member_function_pointer
  - ...

Examples

```
void
my_function(int, char);

decltype(my_function);   // void (int, char)
decltype((my_function)); // void (&) (int, char)
decltype(&my_function);  // void (*) (int, char)

struct my_struct
{
   void
   my_member_function(int);
};

// decltype(my_struct::my_member_function) won't compile!

decltype(&my_struct::my_member_function);
   // void (my_struct::*) (int, char)
```

But wait, there's more!

```
void my_function(int, char) noexcept

decltype(my_function); // void (int, char) noexcept

// Yes, variadic functions have their own type too!
decltype(std::printf) // int(const char*, ...)

struct my_struct
{
    void
    my_member_function(int, ...) const volatile & noexcept;
};

decltype(&my_struct::my_member_function);
//void (my_struct::*)(int, ...) const volatile & noexcept
```

Function qualifiers

- 5 different qualifiers
  - `const`
  - `volatile`
  - lvalue reference (`&`)
  - rvalue reference (`&&`)
  - `noexcept`
- Can't be both lvalue and rvalue reference-qualified
- 24 (2^3 * 3) different types for the same ReturnType/Parameters combo!
- Can't be deduced*, must brute force specialize to cover all cases
- cppreference's "possible implementation" of std::is_function has 48 specializations!
  - Need two for each combination of qualifiers, variadic and non-variadic
- Likely to cause developers to reach for the preprocessor to simplify their lives

Deducing `noexcept`

- `noexcept` added to language in C++11, added to type system in C++17
- Some compilers can deduce `noexcept` using a bool non-type template parameter
- According to [the standard](), *exception-specification* is not a type from which a template argument can be deduced
  - Good amount of support to get added to language
  - Implemented in the following compilers (Tested on [compiler explorer]())
    - GCC 7.1+
    - Clang 5+
  - Not supported in MSVC (Even latest 16.0 Visual Studio 2019)

```
// This function takes pointers to both noexcept and non-noexcept
// functions
template <bool IsNoExcept>
void
my_function(void(*_func)(int) noexcept(IsNoExcept))

// also possible for member functions
```

The dream? Maybe for library implementers

```
template
<
    class T
    , bool IsConst
    , bool IsVolatile
    , bool IsLvalueRef
    , bool isRvalueRef
    , bool IsNoExcept
>
void my_function(T & _object,
    void (T::* _handler)(int)
        const(IsConst)
        volatile(IsVolatile)
        &(IsLvalueRef)
        &&(IsRvalueRef)
        noexcept(IsNoExcept))
```

```cpp
// primary template
template<class>
struct is_function : std::false_type { };

// specialization for regular functions
template<class Ret, class... Args>
struct is_function<Ret(Args...)> : std::true_type {};

// specialization for variadic functions such as std::printf
template<class Ret, class... Args>
struct is_function<Ret(Args......)> : std::true_type {};

// specialization for function types that have cv-qualifiers
template<class Ret, class... Args>
struct is_function<Ret(Args...) const> : std::true_type {};
template<class Ret, class... Args>
struct is_function<Ret(Args...) volatile> : std::true_type {};
template<class Ret, class... Args>
struct is_function<Ret(Args...) const volatile> :
std::true_type {};

// etc...
```

Decomposing member function pointer types

```
template<class T>
struct is_member_function_pointer
    : std::false_type {};


template<class T, class U>
struct is_member_function_pointer<T U::*> // or T (U::*)
    : std::is_function<T> {};
// T will be ReturnType(Args...) qualifiers

static_assert(
    is_member_function_pointer
    <
        &my_struct::my_member_function
    >::value);
```

Decomposing member function pointer types

```
template <class T, class U>
void
register_callback(T & _object, U (T::*_callback))
{
    // This function will also be called for pointers to
    // data members, so we have to make sure it's a function
    static_assert(std::is_function<U>::value);
    //...
}

register_callback(instance, &my_struct::my_callback);
```

Lambda closure decays to function pointer

- Lambda closures that do not capture any variables can implicitly convert to function pointers

```
void (*my_func_pointer)(int) = [](int){};

// helpful when dealing with functions shared with C

std::atomic_bool shutdown(false); // global variable

signal(SIGINT, [](int _signal){shutdown = true; });
```

Questions?

Future additions

- What is the size of a function pointer? Member function pointer? Member data pointer? Union data pointer?