`function_ref`

(a non-owning reference to a `Callable`)

Bloomberg

Vittorio Romeo
https://vittorioromeo.info
vittorio.romeo@outlook.com

Meeting C++ 2017
11/11/2017

C++ is getting *more functional*

- In C++11, we got *lambda expressions* and `std::function`
- In C++14 we got *generic lambdas*
- In C++17 we got `constexpr` *lambdas*

Lambda expressions are syntactic sugar for the definition of *anonymous closure types*

```cpp
auto l = []{ std::cout << "hi!\n"; };
```

↓

```cpp
struct
{
    auto operator()() const
    {
        std::cout << "hi!\n";
    }
} l;
```

Even though they're just *syntactic sugar*, lambdas **changed the way we think about code**

```
const auto benchmark = [](auto f)
{
    const auto time = clock::now();
    f();
    return clock::now() - time;
};
```

```
const auto t = benchmark([]
{
    some_algorithm(/* ... */);
});
```

```cpp
synchronized<widget> sw;
sw.access([](widget& w)
{
    w.foo();
    w.bar();
});
```

- *Lambda expressions* make *higher-order functions* **viable** in C++
    - *E.g.* accepting a function as a parameter
    - *E.g.* returning a function from a function

> What options do we have to implement *higher-order functions*?

Pointers to functions

```
int operation(int(*f)(int, int))
{
    return f(1, 2);
}
```

- Works with *non-member functions* and *stateless closures*
- Doesn't work with *stateful* `Callable` *objects*
- Small run-time overhead (easily inlined in the same TU)
- Constrained, with obvious signature

Template parameters

```
template <typename T>
auto operation(F&& f) → decltype(std::forward<F>(f)(1, 2))
{
    return std::forward<F>(f)(1, 2);
}
```

- Works with *any* `FunctionObject` *or* `Callable` *with* `std::invoke`
- Zero-cost abstraction
- Hard to constrain
- Might degrade compilation time

`std::function`

```
int operation(const std::function<int(int, int)>& f)
{
    return f(1, 2);
}
```

- Works with *any* `FunctionObject` or `Callable`
- Significant run-time overhead (hard to inline/optimize)
- Constrained, with obvious signature
- Unclear semantics: can be both *owning* or *non-owning*

`function_ref`

```
int operation(function_ref<int(int, int)> f)
{
    return f(1, 2);
}
```

- Works with *any* `FunctionObject` or `Callable`
- Small run-time overhead (easily inlined in the same TU)
- Constrained, with obvious signature
- Clear *non-owning* semantics
- Lightweight - think of "`string_view` for `Callable` objects"

I proposed `function_ref` to LEWG as P0792

- Will be discussed this week at the Albuquerque ISO C++ meeting
- https://wg21.link/p0792

How does it work?

"Match" a signature though template specialization:

```
template <typename Signature>
class function_ref;

template <typename Return, typename... Args>
class function_ref<Return(Args...)>
{
    // ...
}
```

Store *pointer to* `Callable` *object* and *pointer to erased function*:

```
template <typename Return, typename... Args>
class function_ref<Return(Args...)>
{
private:
    void* _ptr;
    Return (*_erased_fn)(void*, Args...);

public:
    // ...
};
```

On construction, set the pointers:

```
template <typename F>
function_ref(F&& f) noexcept : _ptr{&f}
{
    _erased_fn = [](void* ptr, Args... xs) -> Return
    {
        return (*reinterpret_cast<F*>(ptr))(
            std::forward<Args>(xs)...);
    };
}
```

On invocation, go through `_erased_fn`:

```
Return operator()(Args... xs) const
{
    return _erased_fn(_ptr, std::forward<Args>(xs)...);
}
```

```cpp
template <typename Return, typename... Args>
class function_ref<Return(Args...)>
{
    void* _ptr;
    Return (*_erased_fn)(void*, Args...);

public:
    template <typename F, /* ...some constraints... */>
    function_ref(F&& x) noexcept : _ptr{&f}
    {
        _erased_fn = [](void* ptr, Args... xs) -> Return {
            return (*reinterpret_cast<F*>(ptr))(
                std::forward<Args>(xs)...);
        };
    }

    Return operator()(Args... xs) const noexcept(/* ... */)
    {
        return _erased_fn(_ptr, std::forward<Args>(xs)...);
    }
};
```

In the proposal (**https://wg21.link/p0792**):

- In-depth analysis of the covered techniques' pros/cons
- Synopsis and specification of `function_ref`
- Existing practice *(e.g. LLVM, Folly, `gdb`, ...)*
- Possible issues and open questions

Article on my blog (**https://vittorioromeo.info**):

- *"Passing functions to functions"*

Thanks!

https://wg21.link/p0792

https://vittorioromeo.info

vittorio.romeo@outlook.com

vromeo5@bloomberg.net

https://github.com/SuperV1234/meetingcpp2017