folly::Function

A Non-copyable Alternative to std::function

Sven Over

Software Engineer, Facebook London

tl; dr

- folly::Function is a replacement for std::function which
  - is not copyable, i.e. move-only
  - is noexcept movable
  - enforces const correctness
  - as fast or faster than std::function in terms of execution time and compile time on our relevant platforms
- 18 months of production use at Facebook
- folly is open source (github.com/facebook/folly)

Outline

- from callable types to function wrappers
- problems we found using std::function
- design decisions for folly::Function
- what we learned from migrating to folly::Function and during 18 months of production use

Callable Types in C++

- function pointers/references
- lambdas
- classes or structs that implement operator()

```cpp
int (*functionPointer)(std::string);
int (&functionReference)(std::string);

auto lambda = [](std::string s) -> int { return s.size(); }

class ComplexObject {
  static int operator()(std::string);
};
```

Stateful Callables

- lambdas that capture data
- classes or structs that implement non-static operator()

```
auto lambda = [x](std::string s) { return s.size() + x; }

class ComplexObject {
  int operator()(std::string) const;
};
```

State-Mutating Callables

- mutable lambdas
- classes or structs that implement non-static non-const operator()

```
auto lambda = [x](std::string s) mutable { return x += s.size(); }

class ComplexObject {
  int operator()(std::string);
};
```

Passing Callables

```cpp
std::string work(int x);

void workAsynchronously(int x, void (*processResult)(std::string));
```

- function pointers can only be used to pass stateless callables

Function Wrappers

```
void workAsynchronously(int x,
                        function<void(std::string)> processResult);
```

- Much simplified declaration:

```
template<typename R, typename... Args>
class function<R(Args...)> {
  void* state;
  void (*func)(void*, Args...);
  void (*destroy)(void*);

  ~function();

  R operator()(Args...);

  template<typename F>
  function(F&& f);

  template<typename F>
  function& operator=(F&& f);
};
```

std::function

std::function

- 48 bytes (libstdc++ on x86_64)
  - function pointer for invoking
  - pointer to management function (copy, delete, etc.)
  - 32 bytes to store wrapped object, or alternatively to store pointer to object on heap
- copyable (makes copy of wrapped object)
- not noexcept movable

Typical Use Cases

- passing a task to libraries for execution at a later time or in a different thread
- storing those tasks in the library implementations
- in either case, those tasks are never executed more than once
- and there is never a need to copy them

Most Popular Use Cases

(from the Facebook code base)

```
folly::Executor* executor;

executor->add(callable);
```

- folly::Executor is an interface (abstract base class) for passing tasks that need to be executed
- implementations include a thread pool which executes tasks in parallel
- std::function<void()> was used to pass tasks to the executor

Most Popular Use Cases

(from the Facebook code base)

```
folly::Future<std::string> result = someRpcCall(1, 2, 3);

result.then([&foo](std::string r) { return foo.extractNumber(r); } )
      .then([obj = std::move(obj)](int x) { obj.setNumber(x); });
```

- Future::then takes a function that is executed when the Future has a value.
- the implementation used to use std::function to store the callback

The Problem with std::function

- often want to capture move-only types (e.g. unique_ptr, folly::Promise)
- std::function can only wrap copyable objects
- this did not compile:

```
MoveOnlyType x;
executor.add([x = std::move(x)]() mutable { x.doStuff(); });
```

Workarounds

std::shared_ptr<T>

```
auto x = std::make_shared<MoveOnlyType>();

executor.add([x](){ x->doStuff(); });
```

- impacts performance:
  - extra memory allocation
  - incrementing/decrementing reference count

Workarounds

folly::MoveWrapper<T>

- wrapper type that implements the copy constructor by moving the contained object
- folly::MoveWrapper<std::unique_ptr<T>> is basically std::auto_ptr<T>
- breaks copy semantics

```
folly::MoveWrapper<MoveOnlyType> x;
executor.add([x]() mutable { x->doStuff(); });
```

The Need for a Different Function Wrapper

- requiring all callable objects to be copyable is painful
- for most of our use cases, we never need to make copies of functions
- you want this to work:

```
MoveOnlyType x;
executor.add([x = std::move(x)]() mutable { x.doStuff(); });
```

Const Correctness

- std::function's operator() is declared const
- but it invokes the wrapped object as a non-const reference
- it does not enforce const correctness

folly::Function

Non-copyable

- must be able to store non-copyable callables
- no run-time performance regression
- maintain value semantics
- therefore must be non-copyable itself

noexcept-Movable

- non-copyable types really need to be noexcept movable
- std::move_if_noexcept is used e.g. by STL containers
- classes containing folly::Function members would be rendered not-noexcept-movable if folly::Function was not noexcept-movable

Const Correct

- folly::Function comes in two flavours:
  - folly::Function<void()>
  - folly::Function<void() const>

Implementation Details

- 64 bytes (on x86_64)
  - function pointer for invoking
  - pointer to management function (move, delete, etc.)
  - 48 bytes to store wrapped objects inline
- types that are not noexcept-movable are never stored inline

Trivia

- std::function objects can be converted to folly::Function, but not vice versa

Migrating to folly::Function

- folly::Function works as a drop-in replacement for std::function
- …unless copyability is needed (surprisingly rare)
- …or lax const behaviour is needed (considered a bug)
- const variant is rarely needed, most function objects are non-const
- std::function often passed as a const&, when replaced with non-const folly::Function must be passed as & or &&

Adoption at Facebook

- folly::Function replaced std::function in folly::Future
- instant win: can use move-only callbacks
- folly::Function replaced std::function in folly::Executor
- required audit/changes to all derived classes
- where migration was non-trivial, it usually pointed to code that needed fixing anyway

std::function Vs folly::Function

- both can be useful
- std::function in APIs where copies of function objects will be needed
- folly::Function everywhere else, to be less restrictive
- use of copy-as-move wrappers (like folly::MoveWrapper) problematic as they make non-copyable things appear copyable

Benchmarks

```
============================================================================
folly/test/function_benchmark/main.cpp          relative  time/iter  iters/s
============================================================================
fn_invoke                                                  911.56ps    1.10G
fn_ptr_invoke                                                1.31ns  761.26M
std_function_invoke                                          2.28ns  437.95M
Function_invoke                                              1.96ns  510.15M
mem_fn_invoke                                                1.22ns  822.84M
fn_ptr_create_invoke                                       911.53ps    1.10G
std_function_create_invoke                                   3.04ns  329.09M
Function_create_invoke                                       2.79ns  358.66M
mem_fn_create_invoke                                       911.53ps    1.10G
============================================================================
```

(measured on Intel Core Haswell 2.5GHz, g++5 20170328, -O3)

Conclusions

- folly::Function can often be used as drop-in replacement for std::function
- it lifts the requirement for wrapped callables to be copyable
- migrating to folly::Function allowed us to get rid of ugly workarounds
- it enforces const correctness
- it is as fast or better than std::function at run-time
- after 18 months, it is widely used in the Facebook codebase

Thank you!