# Why and How to Roll Your Own std::function Implementation

Tom Poole - CppCon 2018

# This talk

The supporting material is available on GitHub:
https://github.com/tpoole/presentations

I'm a maintainer of the open source, cross-platform, application development framework JUCE.

This talk is motivated by the addition of a std::function replacement to JUCE.

- Cross-platform compatibility
- JUCE is focussed on audio applications

But first, a std:: function recap...

# What is std::function?

A type-erasing wrapper containing a callable thing with a specific signature

```
int addOneFunction (int x) {
    return x + 1;
}

struct AddOneStruct {
    int operator() (int x) const {
        return x + 1;
    }
};

std::function<int(int)> f;
f = addOneFunction;
f = AddOneStruct();
f = [](int x) { return x + 1; };
```

Callable things:

- Function pointers
- Objects with a call operator
- Lambda functions

# Why is that useful?

std::function has transformed the way many programs are written

- Asynchronous callbacks

std::function is a good fit for audio processing operations

- A lot of audio effects are successive transformations of arrays of audio samples
- Each stage has the same signature (input samples -> output samples)
- Swappable arrays of callable objects a good way to change behaviour dynamically

Example: An oscillator class which takes a std::function from a user to determine the periodic part of a waveform

# Cross platform compatibility

JUCE is cross platform: desktop, mobile, embedded

A decent number of systems lack a standard library containing std::function

- Toolchains for embedded systems
- OS X < 10.7

JUCE users want to be able to use the same code everywhere

# Realtime audio safety

Processing live audio is merciless.

The process is typically:

- You supply the audio device with a callback, with a pointer to some memory and a size
- The callback is called from the audio thread
- You fill the memory with some audio samples
- A short time later the memory is read by the sound card

If you don't finish writing to the memory you will hear a glitch.

# Predictable runtime

Glitches are bad!

- People will not use your audio software
- Potential to cause permanent hearing damage

Golden rule of audio programming:

- Do not do anything on the audio thread which can take an indeterminate amount of time

-> No locks

-> No memory allocations

# When does std::function allocate?

std::function can avoid allocations by using a small bit of stack space in each object

- Small buffer optimisation (the same principle as std::string)

If a callable thing is bigger than the stack space then std::function will get some memory from the heap

- The size of the stack space is implementation defined

# Other std::function alternatives

Game developers are also worried memory allocations

SG14 (GameDev & low latency ISO C++ working group):

- inplace_function

Facebook's Folly library (see last year's CppCon talk by Sven Over):

- folly::Function

A move-only alternative to std::function which also addresses const-correctness

# A basic implementation

Define a template class, then pass the function signature you want into that template parameter

```
template <typename>
class function;

template <typename Result, typename... Arguments>
class function<Result (Arguments...)>
```

# Type-erasing

```
using invokePtr_t = Result(*)(void*, Arguments&&...);
using createPtr_t = void(*)(void*, void*);
using destroyPtr_t = void(*)(void*);

invokePtr_t invokePtr;
createPtr_t createPtr;
destroyPtr_t destroyPtr;

std::unique_ptr<char[]> storage;
```

```
template <typename Functor>
static Result invoke (Functor* f, Arguments&&... args)
{
    return (*f)(std::forward<Arguments>(args)...);
}

template <typename Functor>
static void create (Functor* destination, Functor* source)
{
    new (destination) Functor (*source);
}

template <typename Functor>
static void destroy (Functor* f)
{
    f->~Functor();
}
```

# Constructor, destructor, call operator

```
template <typename Functor>
function (Functor f)
    : invokePtr  (reinterpret_cast<invokePtr_t>  (invoke<Functor>)),
      createPtr  (reinterpret_cast<createPtr_t>  (create<Functor>)),
      destroyPtr (reinterpret_cast<destroyPtr_t> (destroy<Functor>)),
      storage (new char[sizeof (Functor)])
{
    createPtr (storage.get(), std::addressof (f));
}

~function()
{
    destroyPtr (storage.get());
}

Result operator() (Arguments&&... args) const
{
    return invokePtr (storage.get(), std::forward<Arguments> (args)...);
}
```

# Copying, assigning and null/empty state

```
function() = default;

function (const function& other)
{
    if (other.storage != nullptr)
    {
        invokePtr  = other.invokePtr;
        createPtr  = other.createPtr;
        destroyPtr = other.destroyPtr;

        storageSize = other.storageSize;
        storage.reset (new char[storageSize]);

        createPtr (storage.get(), other.storage.get());
    }
}
```

```
size_t storageSize;
std::unique_ptr<char[]> storage;
```

```
function& operator= (function const& other)
{
    if (storage != nullptr)
    {
        destroyPtr (storage.get());
        storage.reset();
    }

    if (other.storage != nullptr)
    {
        invokePtr = other.invokePtr;
        createPtr = other.createPtr;
        destroyPtr = other.destroyPtr;

        storageSize = other.storageSize;
        storage.reset (new char[storageSize]);

        createPtr (storage.get(), other.storage.get());
    }

    return *this;
}
```

# Benchmark against std::function

```
int addOne (int x)
{
    return x + 1;
}

template <typename FunctionType>
static int doWork()
{
    std::array<FunctionType, 24> functions;

    for (auto& f : functions)
        f = addOne;

    int sum = 0;
    for (auto& f : functions)
        sum += f (4);

    return sum;
}
```

# Benchmark against std::function

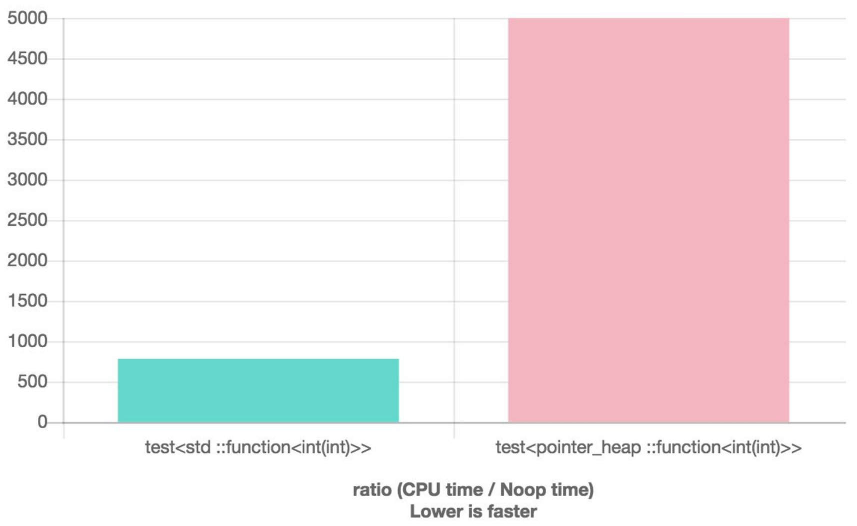
```cpp
int addOne (int x)
{
    return x + 1;
}

template <typename FunctionType>
static int doWork()
{
    std::array<FunctionType, 24> functions;

    for (auto& f : functions)
        f = addOne;

    int sum = 0;
    for (auto& f : functions)
        sum += f (4);

    return sum;
}
```




Courtesy of [http://quick-bench.com/](http://quick-bench.com/)

You can run my code yourself (see final slide).

# The small buffer optimisation (SBO)

The std::function in all the common standard libraries contains a small area of memory in which it can store small callable things.

- Good: avoids allocation
- Bad: non-configurable implementation defined size

But we still want this for compatibility in JUCE!

# std::function feature parity

```
template <typename Functor>
function (Functor f)
    : invokePtr  (reinterpret_cast<invokePtr_t>  (invoke<Functor>)),
      createPtr  (reinterpret_cast<createPtr_t>  (create<Functor>)),
      destroyPtr (reinterpret_cast<destroyPtr_t> (destroy<Functor>))
{
    if (sizeof (Functor) <= sizeof (stack))
    {
        storagePtr = std::addressof (stack);
    }
    else
    {
        heapSize = sizeof (Functor);
        storagePtr = std::malloc (heapSize);
    }

    createPtr (storagePtr, std::addressof (f));
}
```

```
typename std::aligned_storage<24>::type stack;
int heapSize;
void* storagePtr = nullptr;
```

```
function& operator= (function const& other)
{
    if (storagePtr != nullptr)
    {
        destroyPtr (storagePtr);

        if (storagePtr != std::addressof (stack))
            std::free (storagePtr);

        storagePtr = nullptr;
    }

    if (other.storagePtr != nullptr)
    {
        invokePtr  = other.invokePtr;
        createPtr  = other.createPtr;
        destroyPtr = other.destroyPtr;

        if (other.storagePtr == std::addressof (other.stack))
        {
            storagePtr = std::addressof (stack);
        }
        else
        {
            heapSize = other.heapSize;
            storagePtr = std::malloc (heapSize);
        }

        createPtr (storagePtr, other.storagePtr);
    }

    return *this;
}
```

# Benchmark against std::function

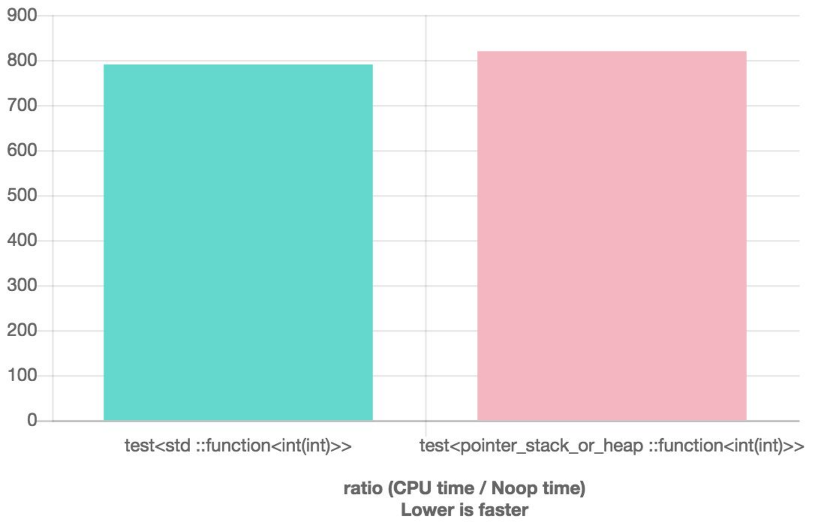
```cpp
int addOne (int x)
{
    return x + 1;
}

template <typename FunctionType>
static int doWork()
{
    std::array<FunctionType, 24> functions;

    for (auto& f : functions)
        f = addOne;

    int sum = 0;
    for (auto& f : functions)
        sum += f (4);

    return sum;
}
```




Courtesy of [http://quick-bench.com/](http://quick-bench.com/)

# Type-erasing function pointer members (again)

```cpp
template <typename Functor>
static Result invoke (Functor* f, Arguments&&... args)
{
    return (*f)(std::forward<Arguments>(args)...);
}

template <typename Functor>
static void create (Functor* destination, Functor* source)
{
    new (destination) Functor (*source);
}

template <typename Functor>
static void destroy (Functor* f)
{
    f->~Functor();
}
```

```cpp
using invokePtr_t = Result(*)(const void*, Arguments&&...);
using createPtr_t = void(*)(void*, const void*);
using destroyPtr_t = void(*)(void*);

invokePtr_t invokePtr;
createPtr_t createPtr;
destroyPtr_t destroyPtr;

typename std::aligned_storage<24>::type stack;
int heapSize;
void* storagePtr = nullptr;
```

# Type-erasing with inheritance

```
template <typename ReturnType, typename... Args>
struct FunctorHolderBase
{
    virtual ~FunctorHolderBase() {}
    virtual ReturnType operator()(Args&&...) = 0;
    virtual void copyInto (void*) const = 0;
    virtual FunctorHolderBase<Result, Arguments...>* clone() const = 0;
};
```

```
typename std::aligned_storage<32>::type stack;
FunctorHolderBase<Result, Arguments...>* functorHolderPtr = nullptr;
```

```
template <typename Functor, typename ReturnType, typename... Args>
struct FunctorHolder final : FunctorHolderBase<Result, Arguments...>
{
    FunctorHolder (Functor func) : f (func) {}

    ReturnType operator()(Args&&... args) override
    {
        return f (std::forward<Arguments> (args)...);
    }

    void copyInto (void* destination) const override
    {
        new (destination) FunctorHolder (f);
    }

    FunctorHolderBase<Result, Arguments...>* clone() const override
    {
        return new FunctorHolder (f);
    }

    Functor f;
};
```

# Benchmark against std::function

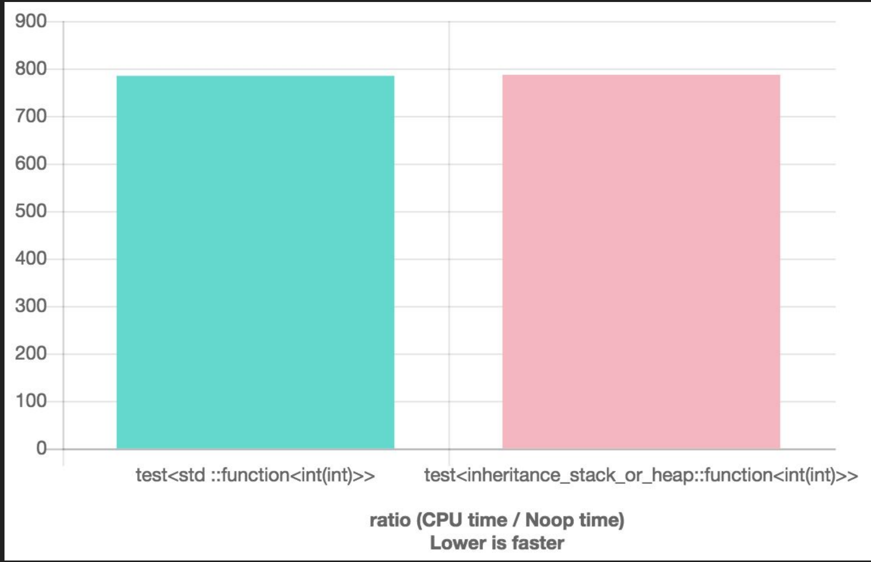
```
int addOne (int x)
{
    return x + 1;
}

template <typename FunctionType>
static int doWork()
{
    std::array<FunctionType, 24> functions;

    for (auto& f : functions)
        f = addOne;

    int sum = 0;
    for (auto& f : functions)
        sum += f (4);

    return sum;
}
```




Courtesy of [http://quick-bench.com/](http://quick-bench.com/)

# Replacing std::function

```
namespace std {
    template <typename T>
    using function = juce::function<T>;
}
```

Adding declarations to the std namespace is undefined behaviour (apart from some template specialisations)

...

but adding function (carefully guarded by lots of preprocessor checks to make sure we are not going to collide with a preexisting definition) has not caused any issues that we know about.

- Only done on systems which lack a std::function implementation
- JUCE is distributed as source code
- No ABI issues

# Simple stack function

```
template <typename Functor>
function (Functor f)
{
    static_assert (sizeof (FunctorHolder<Functor, Result, Arguments...>) <= sizeof (stack), "Too big!");
    functorHolderPtr = (FunctorHolderBase<Result, Arguments...>*) std::addressof (stack);
    new (functorHolderPtr) FunctorHolder<Functor, Result, Arguments...> (f);
}
```

```
Result operator() (Arguments&&... args) const
{
    return (*functorHolderPtr) (std::forward<Arguments>(args)...);
}
```

```
typename std::aligned_storage<24>::type stack;
FunctorHolderBase<Result, Arguments...>* functorHolderPtr = nullptr;
```

# Benchmark against std::function

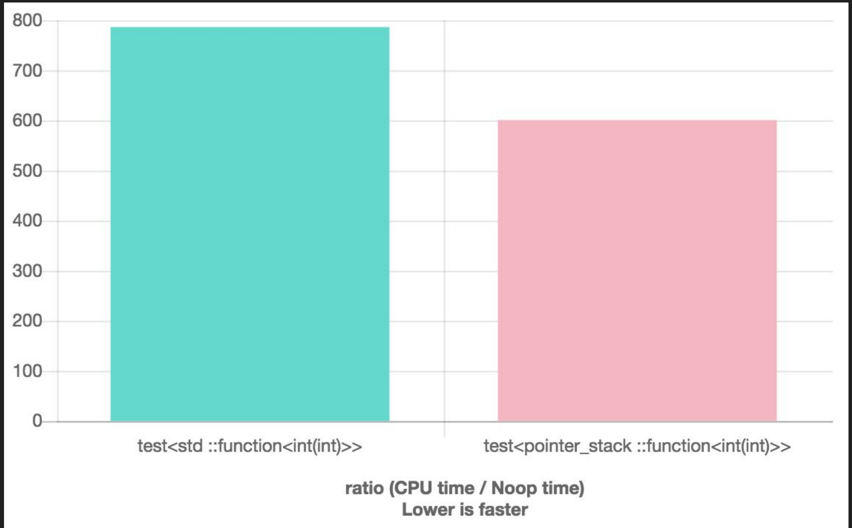
```
int addOne (int x)
{
    return x + 1;
}

template <typename FunctionType>
static int doWork()
{
    std::array<FunctionType, 24> functions;

    for (auto& f : functions)
        f = addOne;

    int sum = 0;
    for (auto& f : functions)
        sum += f (4);

    return sum;
}
```




Courtesy of [http://quick-bench.com/](http://quick-bench.com/)

# Memory management

```
template <typename, class Allocator = std::allocator<char>>
class function;

template <typename Result, typename... Arguments, class Allocator>
class function<Result (Arguments...), Allocator>
{
public:
    template <typename Functor>
    function (Functor f, const Allocator& alloc = Allocator())
```

Delegate allocation behaviour to an allocator

- Allocators are awkward in C++11, better ones available in C++17

A better approach: Louis Dionne's 2017 CppCon talk - "Runtime Polymorphism: Back to the Basics"

# More indirection

```
template <typename>
class function;

template <typename Result, typename... Arguments>
class function<Result (Arguments...)>
{
public:
    virtual ~function() {}
    virtual Result operator() (Arguments&&...) const = 0;
};

template <typename, size_t>
class StackFunction;

template <size_t stackSize, typename Result, typename... Arguments>
class StackFunction<Result (Arguments...), stackSize> final : public function<Result (Arguments...)>
```

# Non-type erasing function

```
template <typename Functor>
function (Functor f)
    : functionPtr  (f)
{}

function() = default;

Result operator() (Arguments&&... args) const
{
    return functionPtr (std::forward<Arguments>(args)...);
}

Result(*functionPtr)(Arguments...) = nullptr;
```

# Benchmark against std::function

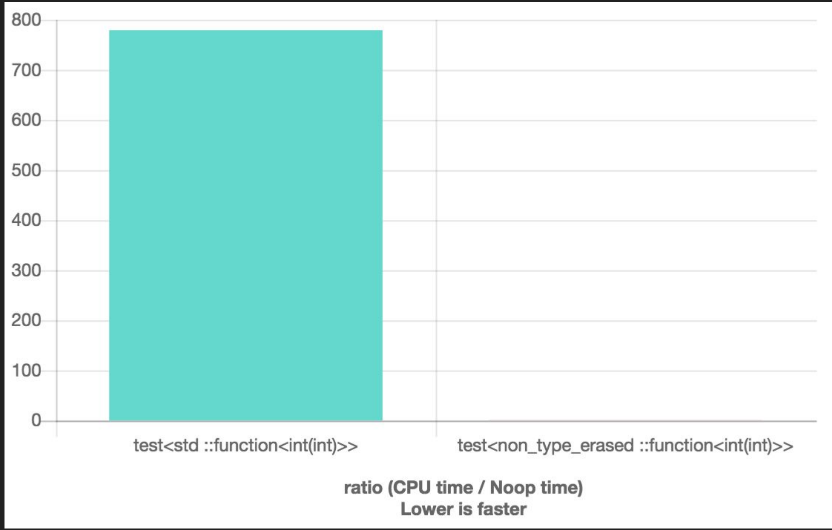
```cpp
int addOne (int x)
{
    return x + 1;
}

template <typename FunctionType>
static int doWork()
{
    std::array<FunctionType, 24> functions;

    for (auto& f : functions)
        f = addOne;

    int sum = 0;
    for (auto& f : functions)
        sum += f (4);

    return sum;
}
```




Courtesy of [http://quick-bench.com/](http://quick-bench.com/)

# Caveats

Only use a type-erased callable wrapper if:

- You don't know the memory layout of the callables you need to store
- You need to store different callables in the same container

Capturing function arguments with a templated type and calling that type is often a better approach.

# Conclusions

Rolling your own version of std::function is not difficult

Replace the internal memory management with your own scheme

- Enforce realtime safety at compile time
- Increase runtime performance
- Use more adventurous memory management

If you can restrict your callable objects to those with a certain memory layout then do not use a type-erasing wrapper!

# Links

Code from the slides: https://github.com/tpoole/presentations

JUCE: https://juce.com/

SG14 inplace_function: https://github.com/WG21-SG14/SG14

folly::Function: https://github.com/facebook/folly

folly::Function CppCon talk: https://www.youtube.com/watch?v=SToaMS3jNH0

Quick Bench: http://quick-bench.com/

Allocation strategies: https://www.youtube.com/watch?v=gVGtNFg4ay0