Document number: D0419R0V0
Date: 2016-07-31
Project: Library Working Group, SG14
Reply-to: Carl Cook, carlcook@optiver.com, Nicolas Fleury, nidoizo@gmail.com

Non-allocating standard functions

Abstract

This paper introduces `inplace_function`, a non-allocating general-purpose polymorphic function wrapper, designed to be a drop-in, zero-cost replacement for `std::function`. This proposal has been put forward due to many members of SG14 implementing this concept within games and low latency C++ systems. The key idea is to avoid the several costs of memory allocation when constructing a `std::function`, instead embedding the buffer for the target within the `inplace_function` itself.

In typical usage of `std::function`, the target is called before the function goes out of scope (or is copied), meaning heap allocation only solves the problem of not knowing how much space to reserve for the target. With `inplace_function`, we pass the responsibility of specifying an adequate buffer size to the caller at compile time (verified by a static assert), allowing users to avoid all unnecessary memory allocations - something important to developers of low latency C++ systems.

I. Table of Contents

II. Introduction

This paper is to outline the motivation for adding a non-allocating standard function to the standard library. A non-allocating standard function is designed to be a drop-in replacement for `std::function`, but with all storage internal to the inplace_function object itself, rather than externally allocated memory.

III. Motivation and Scope

The introduction of `std::function`, a general-purpose polymorphic wrapper over callable targets, has been widely appreciated by C++ users. It gives the ability to assign from several callable target types, pass functions by value, and invoke targets with the familiar function call syntax.

`std::function` generally incurs a dynamic allocation on assignment of the target function (the exception being the small object optimization for function pointers and std::reference_wrappers). For performance critical software, this overhead, while seemingly low, is unacceptable.

Within the SG14 reflector, so far we have found six implementations of non allocating functions that are used in commercial games and high frequency trading applications.[1] This suggests that the problem of dynamic allocation is real, and that a standardised non-allocating function would be of use.

We present a full reference implementation and test suite for inspection. We expect that such a function is useful within games and low latency C++ development, where a general-purpose function is useful, but `std::function` can't be used due to its expensive and unpredictable performance characteristics.

IV. Impact on the standard

This proposal is a pure library extension. It does not require changes to any standard classes, functions or headers, and it does not affect the application binary interface.

It can be implemented by C++11 compilers, and relies only on standard libraries.

---

[1] See Existing Implementations

V. Design decisions

Name

The name static_function is something that could first come up when thinking of an embedded buffer, however with the meaning of “static function” in C++, it would sound confusing. So far the name suggested is inplace_function, as it implies the buffer is embedded, whatever the size of the function. Since a lambda could end up with multiple closures, this is a detail important to be understood as a programmer has to explicitly increase the template size argument. It could make sense to adopt the same nomenclature of proposals like inline_vector, so inline_function (or inplace_vector), to have a common suffix for different standard utilities with embedded buffers.

Relation with std::function

The first discussion on SG14 was about adding a base class to std::function (or make std::function a template typedef) that is more flexible to prevent heap usage. However as discussion evolved, the conclusion is that is what is wanted is another class, std::inplace_function, dedicated to being allocation-less.

For that new class, sharing a base class with `std::function` was discussed, to be able to pass function objects by reference without dependence on how it’s stored. However, that might not be worth the burden in implementation restrictions, and would break the ABI with the existing `std::function`. Instead, std::`inplace_function` class can prioritize performance without compromise, and still conform to the `std::function` interface.

Copying from `std::inplace_function` to `std::function` of the same function signature should be supported, as `std::function` supports any function size. However, so far, copying from `std::function` to `std::inplace_function` would not be allowed, as it risks breaking the compile-time guarantees of `std::inplace_function` (an option here is to throw a runtime exception if the target buffer is too small).

It might be worth noting that a codebase preferring `std::inplace_function` to `std::function` will probably always prefer it.

Class signature

```
template<typename Signature, size_t Capacity = /*default-capacity*/,
size_t Alignment = /*default-alignment*/>
class inplace_function;
```

- `Capacity` is the size of the internal buffer
- `Alignment` is the largest supported alignment of assigned functions
- `Default-capacity` is implementation-defined
- `Default-alignment` is implementation-defined

Use of standard allocators

Allocator support was part of the original proposal for std::function, but was dropped as part of C++17 (see P0302R1). The same concerns for allocator support are held by us, such as what should be the expected behaviour when an `inplace_function` is copied, and what are the semantics for type erasure v.s. allocators.

A more pressing concern, however, is the negative performance impact a custom allocator is likely to introduce. For example, if the internal buffer is allocated elsewhere, this means that the management information (such as the pointer to the buffer) are located inplace, but the actual buffer itself is elsewhere.

We believe that custom allocators are not a good match for inplace_function. We prefer to allow the implementor of inplace_function to determine the exact layout of the object's memory, and it is usually best to hold all such memory internal to the inplace_function (i.e. allocation free).

Relationship to llvm::function_ref

The LLVM API includes a utility function named `llvm::function_ref`, which provides a reference to a callable object: http://llvm.org/docs/ProgrammersManual.html#the-function-ref-class-template. This is semantically identical to a `std::function` that contains a reference wrapper, i.e. a non allocating handle to a callable object that can itself be passed by reference or by value.

While this is comparable to `inplace_function`, a copy of the callable object is made by `inplace_function`, meaning there are no issues with temporaries going out of scope before the `inplace_function` is invoked. For example, if a lambda has references to local variables, the compiler is free to clean these up before the `llvm::function_ref` is invoked, leading to undefined behavior.

Possibility of std::make_inplace_function

This would be useful in some cases, but given that `std::function` does not provide a `make_function` (due to ambiguities in terms of type resolution), `make_inplace_function` won't be provided either.

Compilation-time guarantee

Since the buffer size and alignment is known at compilation-time, then assigned functions are validated at compilation-time to be of proper size and alignment. The function size can be at most the buffer size, and function alignment can be at most the alignment. Internal to `std::inplace_function`, `static_assert` should be used for these validations.

The only run-time error inside `std::inplace_function` itself is when calling it without any function assigned.

Default buffer size and default alignment size

In our implementation, a default buffer size of 32 bytes, with alignment of 16 bytes, seems to be a reasonable choice. This gives a total object size of 48 bytes (i.e. less than a L1 cache line on x86), and is still large enough to capture most callable objects in the codebases tested on so far. With 16 byte alignment, this allows us to store the two internal pointers with no padding, followed by the buffer, on the same cache line, with no padding (assuming the `inplace_function` object itself is cache aligned).

Copy/move/destruction

Proper copy, move and destruction are all supported for the embedded function. Again, the exact mechanics of this are left to the implementation.

Generated code

An optimizing compiler should generate a minimum amount of overhead when constructing and invoking an `inplace_function`. For example, to construct an `inplace_function` with a simple lambda, such as:

```
inplace_function<void()> fn = [&]{ locallyScopedVar += rand(); };
```

all recent versions of gcc will generate two move instructions (one for the address of the management function, and one for the address of the invocation function), and then a single call instruction against the invocation function.

Memory layout

Memory layout is left to implementation, however we can note that all implementations we have found so far have taken the same approach of storing function pointers directly as members to avoid the indirection of type-erasing using a vtable, as well as a properly aligned buffer to store the function.

The function pointers are used for four things: calling, copying, moving and destroying. The same function can be used for multiple tasks. However, since calling performance is the most important and has a unique signature, the function pointer for calling should probably be dedicated to that task.

The buffer storing the function will be used for calling, but its last bytes may have a high chance of not being used. So optimal memory layout depends on `Alignment`, as follows:

If `Alignment <= sizeof(void*)` then it is optimal to store the members in this order:

1. CallerFctPtr
2. Buffer
3. ManagementFctPtr

If `Alignment == 2*sizeof(void*)` then you can avoid wasted padding in the first cache line, and members should be stored in this order:

1. CallerFctPtr
2. ManagementFctPtr
3. Buffer

Otherwise the same logic applies if the implementation would use more than two function pointers:

1. CallerFctPtr
2. DestructionFctPtr
3. CopyAndMoveFctPtr
4. Buffer

Overall we tend to think it's better to put the Destroy, Copy and Move routines inside the same management function, similar to gcc's implementation of `std::function`.

Trivial/non trivial classes split

An additional `std::inplace_trivial_function` class could be provided to avoid storing function pointers to management routines that are not used. However, the flexible member layout that can be used depending on alignment reduces this need, by storing members in terms of optimal cache locality.

Base class without size

A base class `std::inplace_function_base` without the `Capacity` template argument could be added, to allow passing a `std::inplace_function` object of any capacity as an argument. It would contain the caller function pointer. However to be fully functional it would need to pass the `this` pointer to the caller function or have an additional template argument with alignment to be able to perform a proper down cast upon invocation.

The base class would require deleted or protected copy constructors to avoid object slicing, meaning a solution like proposal of `std::unique_function`[2] could be used instead. A proposal like `std::unique_function` sounds more powerful for this kind of need, by allowing wrapping of any `callable` type.

Swapping

We have seen some implementations with support for swapping. However, we have seen some implementations that would not properly support certain functor types. For example, suppose the buffer is implemented by the following member:

```
std::aligned_storage<CapacityT, AlignmentT> _M_data;
```

You cannot do something as simple as this in the swap function:

---

[2] See Related Work

```
std::swap(_M_data, other._M_data);
```

Since the two buffers can contain different types (functors), swapping must be done through three different moves and would only work for two buffers of same size:

```
std::aligned_storage<Capacity, Alignment> tempData;
std::move(_M_data, tempData);
std::move(other._M_data, this->_M_data);
std::move(tempData, other._M_data);
```

VI. Technical specifications

A full implementation and tests can be found on the SG14 github repository, please see item [1] in the references section for details.

```
template <typename Signature, size_t Capacity =
/*InplaceFunctionDefaultCapacity*/, size_t Alignment =
/*InplaceFunctionDefaultAlignment*/>
class inplace_function;

template <typename R, typename... Args, size_t Capacity, size_t Alignment>
class inplace_function<R(Args...), Capacity, Alignment>
{
Public:
      // Creates an empty function
      inplace_function();

      // Destroys the inplace_function. If the stored callable is valid, it
is destroyed also
      ~inplace_function();

      // Creates an implace function, copying the target of other within the
internal buffer
      // If the callable is larger than the internal buffer, a compile-time
error is issued
      // May throw any exception encountered by the constructor when copying
the target object
      template<typename Callable>
      inplace_function(const Callable& target);

      // Moves the target of  an implace function, storing the callable
within the internal buffer
      // If the callable is larger than the internal buffer, a compile-time
error is issued
      // May throw any exception encountered by the constructor when moving
the target object
      template<typename Callable>
      inplace_function(Callable&& target);
```

```
	// Copy construct an inplace_function, storing a copy of other's
target internally
	// May throw any exception encountered by the constructor when copying
the target object
	inplace_function(const inplace_function& other);

	// Move construct an inplace_function, moving the other's target to
this inplace_function's internal buffer
	// May throw any exception encountered by the constructor when moving
the target object
	inplace_function(inplace_function&& other);

	// Allows for copying from inplace_function object of the same type,
but with a smaller buffer
	// May throw any exception encountered by the constructor when copying
the target object
	// If OtherCapacity is greater than Capacity, a compile-time error is
issued
	template<size_t OtherCapacity>
	inplace_function(const inplace_function<R(Args...), OtherCapacity>&
other);

	// Allows for moving an inplace_function object of the same type, but
with a smaller buffer
	// May throw any exception encountered by the constructor when moving
the target object
	// If OtherCapacity is greater than Capacity, a compile-time error is
issue
	template<size_t OtherCapacity>
	inplace_function(inplace_function<R(Args...), OtherCapacity>&& other);

	// Assigns a copy of other's target
	// May throw any exception encountered by the assignment operator when
copying the target object
	inplace_function& operator=(const inplace_function& other);

	// Assigns the other's target by way of moving
	// May throw any exception encountered by the assignment operator when
moving the target object
	inplace_function& operator=(inplace_function&& other);

	// Allows for copy assignment of an inplace_function object of the
same type, but with a smaller buffer
	// If the copy constructor of target object throws, this is left in
uninitialized state
	// If OtherCapacity is greater than Capacity, a compile-time error is
issued
	template<size_t OtherCapacity>
	inplace_function& operator=(const inplace_function<R(Args...),
OtherCapacity>& other);
```

```
	// Allows for move assignment of an inplace_function object of the
same type, but with a smaller buffer
	// If the move constructor of target object throws, this is left in
uninitialized state
	// If OtherCapacity is greater than Capacity, a compile-time error is
issued
	template<size_t OtherCapacity>
	inplace_function& operator=(inplace_function<R(Args...),
OtherCapacity>&& other);

	// Assign a new target
	// If the copy constructor of target object throws, this is left in
uninitialized state
	template<typename Callable>
	inplace_function& operator=(const Callable& target);

	// Assign a new target by way of moving
	// If the move constructor of target object throws, this is left in
uninitialized state
	template<typename Callable>
	inplace_function& operator=(Callable&& target);

	// Compares this inplace function with a null pointer
	// Empty functions compare equal, non-empty functions compare unequal
	bool operator==(std::nullptr_t);

	// Compares this inplace function with a null pointer
	// Empty functions compare equal, non-empty functions compare unequal
	bool operator!=(std::nullptr_t);

	// Converts to 'true' if assigned
	explicit operator bool() const throw();

	// Invokes the target
	// Throws std::bad_function_call if not assigned
	R operator () (Args... args) const;

	// Swap two targets
	void swap(inplace_function& other);
};
```

Sample use

```
#include <iostream>

// simple functor type
struct Functor
{
	Functor() {}
	Functor(const Functor&) { std::cout << "copy functor" << std::endl; }
```

```
	Functor(Functor&&) { std::cout << "move functor" << std::endl; }
	void operator()()
	{
		std::cout << "functor operator()" << std::endl;
	}
};

// simple free function
void Foo()
{
	std::cout << "foo()" << std::endl;
}

// exercise either a standard_function or inplace_function
template <typename T>
void FunctionTest()
{
	// construct function from lambda and invoke
	T func1 = [] { std::cout << "lambda invoked" << std::endl; };
	func1();

	// assign to function from free function and invoke
	func1 = &Foo;
	func1();

	// construct function from functor and invoke
	T func2 = Functor();
	func2();

	// swap two compatible functions
	func.swap(func2);
}

int main()
{
	FunctionTest<std::inplace_function<void()>>();
	FunctionTest<std::function<void()>>();
}
```

VII. Acknowledgements

The authors would like to thank Maciej Gajewski from Optiver B.V. and Edward Catmur from Maven Securities, for contributing their reference implementations, and for their insightful comments.

Existing implementations

1. Optiver B.V.

a. Non allocating function which has a user specified capacity. Static_assert is used to detect buffer overflows. Lambdas record destructors and constructors

2. Maven Securities:
   a. Non allocating function which supports only trivial types, meaning no pointer to constructors or destructors is required (only the buffer and an invocation pointer). A user defined capacity of N bytes, with static_asserts for overflow
   b. Non allocating function which supports copying, moving and destructing of callable targets. A user defined capacity of N bytes.
3. Ubisoft
   a. Non allocating function that was a wrapper over std::function using TLS to work with specific stateless allocator. Was working with VS2012 but with variadic templates it's now much simpler to make a custom type without wrapping std::function.
4. Wargaming Seattle
   a. Sean to comment here?
5. Erik Ringtorp
   a. Erik, feel like commenting here?
6. https://github.com/rukkal/static-stl/blob/master/include/sstl/function.h

VIII. References

- [1] https://github.com/carlcook/SG14/SG14/inplace_function.h
- [2] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4543.pdf