P0984R0 - All (*)()-Pointers Replaced by Ideal Lambdas Functions Objects Obsoleted by Lambdas

Peter Sommerlad

2018-04-01

Document Number:	P0984R0
Date:	2018-04-01
Project:	Programming Language C++
Audience:	EWG/LEWG
Target:	C++20

Abstract

Ban non-overloaded functions and `<functional>` function object class templates from the standard = simpler syntax

1 Motivation

Recent discussions around abbreviated concept syntax not needing braces or parentheses (P0956) or angle brackets, as well as the ban to take function addresses of standard functions (P0551) and passing them to algorithms, and the ugliness of `std::multiplies<>` as an algorithm argument led me to propose this change to the standard.

In addition it addresses issues many programmers have with ADL and allows for safer code, because it is easier to employ those functions in functional code without the ugliness or uncertainty one faces when using a function name that might degenerate to a function pointer (or not in generic code accepting references).

As a third issue, I am addressing that the standard committee often failed to "eat their own dog food" when specifying the next revision of the C++ standard, leading to holes in the library or even the language.

1.1 Inability to take the address of a function defined in the standard library

Please note the following example taken from P0052R7. It shows how to sidestep taking the address of a standard library function (`&::fclose`) by wrapping it in a (unnecessarily) generic lambda: [*Example:* The following example shows its use to avoid calling `fclose` when `fopen` fails

```
auto file = make_unique_resource_checked(
  ::fopen("potentially_nonexistent_file.txt", "r"),
  nullptr, [](auto fptr){::fclose(fptr);});
```

— *end example*]

Almost all examples on stack overflow are wrong with the current wording, e.g. `https://stackoverflow.com/questions/26360916/using-custom-deleter-with-unique-ptr` that claims `unique_ptr<FILE, int(*)(FILE*)>(fopen("file.txt", "rt"), &fclose);` is the way to employ `unique_ptr` for `FILE *`.

1.2 Ugly syntax when using standard function objects

Teaching a functional approach has become a treat with the introduction of lambdas in C++11, especially the relaxed rules for lambda bodies and generic lambdas of more recent standard revisions cry out for simplifying the syntax. Just consider the burden and teachability of the accumulate call in multiply_vector versus the multiply_vector_new.

```
void multiply_vector(){
        std::vector v{1,2,3,4,5,6};
        auto res=accumulate(begin(v),end(v),1,std::multiplies<>{});
        ASSERT_EQUAL(720,res);
}
```

```
void multiply_vector_new(){
        std::vector v{1,2,3,4,5,6};
        auto res=accumulate(begin(v),end(v),1,Times);
        ASSERT_EQUAL(720,res);
}
```

In addition the standard library functional forgot to provide corresponding function object class templates for a whole bunch of operators. Also using C++17 fold expressions one could do even more by providing n-ary function objects for the corresponding binary operators. This will also add to the versatility of the corresponding function objects.

2 April's Impact on the standard

Core language change:

2.1 8.4.5.1 Closure Types [expr.prim.lambda.closure]

In [expr.prim.lambda.closure] change paragraph 7 as follows

1 The closure type for a non-generic *lambda-expression* with no *lambda-capture* whose constraints (if any) are satisfied is called a *Ideal Lambda*. An Ideal Lambda has a conversion function to pointer to function with C++ language linkage(10.5) having the same parameter and return types as the closure type's function call operator. The conversion is to "pointer to `noexcept` function" if the function call operator has a non-throwing exception specification. The value returned by this conversion function is the address of a function `F` that, when invoked, has the same effect as invoking the closure type's function call operator. `F` is a constexpr function if the function call operator is a constexpr function. For a generic lambda with no *lambda-capture*, the closure type

has a conversion function template to pointer to function. The conversion function template has the same invented template parameter list, and the pointer to function has the same parameter types, as the function call operator template. The return type of the pointer to function shall behave as if it were a *decltype-specifier* denoting the return type of the corresponding function call operator template specialization. An Ideal Lambda furthermore defines an overload for the unary `operator&()` that returns the result of the said conversion to function pointer. [*Note:* That operator overload guarantees that existing code bases that invalidly take the address of a standard library function continue to work as expected. — *end note*]

2.2 20.5.1.2 Headers [headers]

Change paragraph 6 of section [headers] as follows:

Names that are defined as functions in C shall be defined as ~~functions~~`constexpr inline auto` variables initialized from an Ideal Lambda in the C++ standard library, unless the C++ standard defines overloads of said function. In that case the names defined as functions in C shall be defined as functions. [1]

2.3 20.5.2.3 Linkage [using.linkage]

Change paragraph 2 as follows:

~~Whether a~~A name from the C standard library declared with external linkage has ~~`extern "C"` or~~ `extern "C++"` linkage ~~is implementation-defined. It is recommended that an implementation use `extern "C++"` linkage for this purpose.~~[2]

2.4 20.5.5.4 Non-member functions [global.functions]

Change paragraph 1 of [global.functions] as follows

1 ~~It is unspecified whether any~~ All non-overloaded non-template non-member functions in the C++ standard library shall be defined as `constexpr inline auto` variables initialized from an Ideal Lambda. For the purpose of this standard these variables are called *FOOL* (Function ObsOleted by Lambda). [*Note:* This mechanism allows many wrong programs that take the address of a standard library function to conform to this standard. — *end note*] It is unspecified wether any overloaded or templated non-member functions are defined as inline(10.1.6).

2.5 20.5.5.6 Constexpr functions and constructors [constexpr.functions]

Extend paragraph 1 of [constexpr.functions] as follows:

This document explicitly requires that certain standard library functions are `constexpr`(10.1.5). An implementation shall not declare any standard library function signature as `constexpr` except for those where it is explicitly required. Within any header that provides any non-defining declarations

1) This disallows the practice, allowed in C, of providing a masking macro in addition to the function prototype. The only way to achieve equivalent inline behavior in C++ is to provide a definition ~~as an extern inline function~~within the Ideal Lambda's body.

2) The only reliable way to declare an object or function signature from the C standard library is by including the header that declares it, notwithstanding the latitude granted in 7.1.4 of the C Standard.

of constexpr functions or constructors an implementation shall provide corresponding definitions. A FOOL's ([global.functions]) call operator shall be defined as `constexpr` or not accordingly.

3 Fool's Impact on the standard

Note my plan is to deprecate all existing function object class templates and replace them by the generic version of a lambda which is not only the syntactical convenient thing, but also the only thing that should work^{TM}. Names of the lambda variables are uppercased at the moment to avoid confusion, but that is LEWG's bike shedding's call.

As a side effect, disregarding the space for the deprecated features section 23.14 becomes significantly shorter. One could also attempt to deprecate `bind`, but I leave that with LEWG and do not propose it right now.

3.1 23.14.1 Header <functional> synopsis [functional.syn]

Adjust the synopsis of [functional.syn] as follows. Move all removed definitions and declarations to appendix D.

```
namespace std {
  // ??, invoke
  template<class F, class... Args>
    invoke_result_t<F, Args...> invoke(F&& f, Args&&... args)
      noexcept(is_nothrow_invocable_v<F, Args...>);

  // ??, reference_wrapper
  template<class T> class reference_wrapper;

  template<class T> reference_wrapper<T> ref(T&) noexcept;
  template<class T> reference_wrapper<const T> cref(const T&) noexcept;
  template<class T> void ref(const T&&) = delete;
  template<class T> void cref(const T&&) = delete;

  template<class T> reference_wrapper<T> ref(reference_wrapper<T>) noexcept;
  template<class T> reference_wrapper<const T> cref(reference_wrapper<T>) noexcept;
  // unary operators
  constexpr inline auto Deref = see below ; // unary *
  constexpr inline auto Address = see below ; // unary &
  constexpr inline auto Negate = see below ; // unary -
  constexpr inline auto Posate = see below ; // unary +
  constexpr inline auto Not = see below ; // ! not
  constexpr inline auto Cmpl = see below ; // ~ cmpl


  // left associative binary operators
  constexpr inline auto PtrMemb = see below ; // ->*
  constexpr inline auto RefMemb = see below ; // .*

  constexpr inline auto Plus = see below ; // +
constexpr inline auto Minus = see below ; // -
constexpr inline auto Times = see below ; // *
constexpr inline auto Divide = see below ; // /
constexpr inline auto Remainder = see below ; // &

constexpr inline auto Equal = see below ; // ==
constexpr inline auto Not_eq = see below ; // !=
constexpr inline auto Bigger = see below ; // >
constexpr inline auto Smaller = see below ; // <
constexpr inline auto Maybe_bigger = see below ; // >=
constexpr inline auto Sometimes_smaller = see below ; // <=
constexpr inline auto Spaceship = see below ; // <=>

constexpr inline auto And = see below ; // && and
constexpr inline auto Or = see below ; // || or

constexpr inline auto Bitand = see below ; // & bit_and
constexpr inline auto Bitor = see below ; // | bit_or
constexpr inline auto Xor = see below ; // ^ xor
constexpr inline auto Lshift = see below ; // «
constexpr inline auto Rshift = see below ; // »

// right associative binary operators
constexpr inline auto Assign = see below ; // =
constexpr inline auto PlusAssign = see below ; // +=
constexpr inline auto MinusAssign = see below ; // -=
constexpr inline auto TimesAssign = see below ; // *=
constexpr inline auto DivideAssign = see below ; // /=
constexpr inline auto RemainderAssign = see below ; // %=
constexpr inline auto AndAssign = see below ; // &&=
constexpr inline auto OrAssign = see below ; // ||=
constexpr inline auto And_eq = see below ; // &=
constexpr inline auto Or_eq = see below ; // |=
constexpr inline auto Xor_eq = see below ; // ^=
constexpr inline auto LshiftAssign = see below ; // <<=
constexpr inline auto RrhiftAssign = see below ; // >>=

// ternary operator
constexpr inline auto Wtf = see below ; // ?:

// ??, arithmetic operations
template<class T = void> struct plus;
template<class T = void> struct minus;
template<class T = void> struct multiplies;
template<class T = void> struct divides;
template<class T = void> struct modulus;
template<class T = void> struct negate;
template<> struct plus<void>;
template<> struct minus<void>;
template<> struct multiplies<void>;
template<> struct divides<void>;
```

```
template<> struct modulus<void>;
template<> struct negate<void>;

// ??, comparisons
template<class T = void> struct equal_to;
template<class T = void> struct not_equal_to;
template<class T = void> struct greater;
template<class T = void> struct less;
template<class T = void> struct greater_equal;
template<class T = void> struct less_equal;
template<> struct equal_to<void>;
template<> struct not_equal_to<void>;
template<> struct greater<void>;
template<> struct less<void>;
template<> struct greater_equal<void>;
template<> struct less_equal<void>;

// ??, logical operations
template<class T = void> struct logical_and;
template<class T = void> struct logical_or;
template<class T = void> struct logical_not;
template<> struct logical_and<void>;
template<> struct logical_or<void>;
template<> struct logical_not<void>;

// ??, bitwise operations
template<class T = void> struct bit_and;
template<class T = void> struct bit_or;
template<class T = void> struct bit_xor;
template<class T = void> struct bit_not;
template<> struct bit_and<void>;
template<> struct bit_or<void>;
template<> struct bit_xor<void>;
template<> struct bit_not<void>;


// ??, function template not_fn
template<class F> unspecified not_fn(F&& f);

// ??, bind
template<class T> struct is_bind_expression;
template<class T> struct is_placeholder;

template<class F, class... BoundArgs>
  unspecified bind(F&&, BoundArgs&&...);
template<class R, class F, class... BoundArgs>
  unspecified bind(F&&, BoundArgs&&...);

namespace placeholders {
  // M is the implementation-defined number of placeholders
  see below _1;
```

```
    see below _2;
              .
              .
              .
    see below _M;
  }

  // ??, member function adaptors
  template<class R, class T>
    unspecified mem_fn(R T::*) noexcept;

  // ??, polymorphic function wrappers
  class bad_function_call;

  template<class> class function; // not defined
  template<class R, class... ArgTypes> class function<R(ArgTypes...)>;

  template<class R, class... ArgTypes>
    void swap(function<R(ArgTypes...)>&, function<R(ArgTypes...)>&) noexcept;

  template<class R, class... ArgTypes>
    bool operator==(const function<R(ArgTypes...)>&, nullptr_t) noexcept;
  template<class R, class... ArgTypes>
    bool operator==(nullptr_t, const function<R(ArgTypes...)>&) noexcept;
  template<class R, class... ArgTypes>
    bool operator!=(const function<R(ArgTypes...)>&, nullptr_t) noexcept;
  template<class R, class... ArgTypes>
    bool operator!=(nullptr_t, const function<R(ArgTypes...)>&) noexcept;

  // ??, searchers
  template<class ForwardIterator, class BinaryPredicate = equal_to<>>
    class default_searcher;

  template<class RandomAccessIterator,
           class Hash = hash<typename iterator_traits<RandomAccessIterator>::value_type>,
           class BinaryPredicate = equal_to<>>
    class boyer_moore_searcher;

  template<class RandomAccessIterator,
           class Hash = hash<typename iterator_traits<RandomAccessIterator>::value_type>,
           class BinaryPredicate = equal_to<>>
    class boyer_moore_horspool_searcher;

  // ??, hash function primary template
  template<class T>
    struct hash;

  // ??, function object binders
  template<class T>
    inline constexpr bool is_bind_expression_v = is_bind_expression<T>::value;
  template<class T>
    inline constexpr int is_placeholder_v = is_placeholder<T>::value;
}
```

Move the sections 23.14.6 [arithmetic.operations], 23.14.7 [comparisons], 23.14.8 [logical.operations], and 23.14.9 [bitwise.operations]j to Appendix D and replace them by the following new section 23.14.6.

3.2 23.14.6 Function Objects for Operators as Lambdas [functional.fool]

According to Table 1 provide the corresponding definition of the constexpr variable by initializing it with a lambda expression. For each unary operator `@` in the Table 1 with the name *`Name`* define the variable initialized with a unary lambda expression as follows:

```
  constexpr inline auto Name  =
    [](auto&& x) constexpr
      noexcept(noexcept( @ std::declval<decltype(x)>()))
      -> decltype(@ std::forward<decltype(x)>(x))
    {
      return @ std::forward<decltype(x)>(x);
    };
```

Table 1 — Unary operation function objects

name	operator
`Deref`	`*`
`Address`	`&`
`Negate`	`-`
`Posate`	`+`
`Not`	`!`
`Cmpl`	`~`

According to Table 2 provide the corresponding definition of the constexpr variable by initializing it with a lambda expression. For each binary operator `@` in Table 2 with the name *`Name`* define the variable initialized with a lambda expression as follows:

```
  constexpr inline auto Name =
    [](auto&& ... x) constexpr
      noexcept(noexcept(( ... @ std::declval<decltype(x)>()))
      -> decltype((... @ std::forward<decltype(x)>(x)))
    {
      return ( ... @ std::forward<decltype(x)>(x));
    };
```

Table 2 — Binary left associative operation function objects

name	operator
`PtrMemb`	`->*`
`RefMemb`	`.*`
`Plus`	`+`
`Minus`	`-`
`Times`	`*`
`Divide`	`/`
`Remainder`	`%`
`Equal`	`==`
`Unequal`	`!=`
`Bigger`	`>`
`Smaller`	`<`
`Maybe_bigger`	`>=`
`Sometimes_smaller`	`<=`
`Spaceship`	`<=>`
`And`	`&&`
`Or`	`||`
`Bitand`	`&`
`Bitor`	`|`
`Xor`	`^`
`Lshift`	`<<`
`Rshift`	`>>`

According to Table 3 provide the corresponding definition of the constexpr variable by initializing it with a lambda expression. For each binary operator *@* in Table 3 with the name *Name* define the variable initialized with a lambda expression as follows:

```
constexpr inline auto Name =
  [](auto&& ... x) constexpr
    noexcept(noexcept((std::declval<decltype(x)>() @  ... ))
    -> decltype((std::forward<decltype(x)>(x) @ ... ))
  {
    return (std::forward<decltype(x)>(x) @ ...);
  };
```

Table 3 — Binary right associative operation function objects

name	operator
`Assign`	`=`
`PlusAssign`	`+=`
`MinusAssign`	`-=`
`TimesAssign`	`*=`
`DivideAssign`	`/=`
`RemainderAssign`	`%=`
`AndAssign`	`&&=`
`OrAssign`	`||=`
`And_eq`	`&=`
`Or_eq`	`|=`
`Xor_eq`	`^=`
`LshiftAssign`	`<<=`
`RshiftAssign`	`>>=`

Define the ternary operator's lambda object as follows:

```
constexpr auto Wtf=[](auto&&c, auto&& l, auto&& r) constexpr
noexcept(noexcept(std::declval<decltype(c)>() ? std::declval<decltype(l)>() : std::declval<decltype(r)>
-> decltype(std::declval<decltype(c)>() ? std::declval<decltype(l)>() : std::declval<decltype(r)>())
{
        return std::forward<decltype(c)>(c) ?
                        std::forward<decltype(l)>(l) : std::forward<decltype(r)>(r);
};
```