From Functions to Lambdas:

How Do C++ Callables *Really* Work?

WALTER E. BROWN, PH.D.

< webrown.cpp @ gmail.com >

A little about me

- B.A. (math's); M.S., Ph.D. (computer science).
- Professional programmer for over 50 years, programming in C++ since 1982.
- Experienced in industry, academia, consulting, and research:
  - Founded a Computer Science Dept.; served as Professor and Dept. Head; taught and mentored at all levels.
  - Managed and mentored the programming staff for a reseller.
  - Lectured internationally as a software consultant and commercial trainer.
  - Retired from the Scientific Computing Division at Fermilab, specializing in C++ programming and in-house consulting.
- Not dead — still doing training & consulting. (Email me!)

 2

Emeritus participant in C++ standardization

- Written >160 papers for WG21, proposing such now-standard C++ library features as gcd/lcm, cbegin/cend, common_type, and void_t, as well as all of headers <random> and <ratio>.
- Influenced such core language features as *alias templates*, *contextual conversions*, and *variable templates*; recently worked on *requires-expressions*, operator<=>, and more!
- Conceived and served as Project Editor for *Int'l Standard on Mathematical Special Functions in C++* (ISO/IEC 29124), now incorporated into C++17's <cmath>.
- Be forewarned: Based on my training and experience, I hold some rather strong opinions about computer software and programming methodology — these opinions are not shared by all programmers, but they should be! ☺

 3

In today's talk

- The Big Picture
- Callable Types and Their Call Syntax
- Callbacks
- Function Objects and Their Types
- Higher-Order Functions
- Function Objects in Disguise
- An Advanced Example (as time permits)

 4

The Big Picture

Our theme

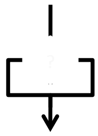
- ( ):
  - Usually termed the call/invoke operator.
  - Distinct from parentheses used to group (an operator w/ its operands) or to punctuate (a comma-separated list).
- g(···) is the commonest syntax to apply the operator:
  - Intending to yield control (of the CPU) to entity g, and …
  - Expecting to resume control when g returns (finishes).
  - Akin to a news anchor who calls upon a field reporter, later resuming when that reporter says "Back to you."

 6

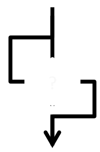
Comparing basic flow of control (many variations; not in this talk)

- Sequential

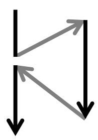
- Conditional



- Iterative



- Subroutine



Example

```
• void second( ) {
    puts( "Second\n" );
    return;
  }
• int main( ) {
    puts( "First\n" );
    second( );
    puts( "Third\n" );
    return 0;
  }
```

- Fctn main calls upon fctn second via a call expression:
  - So main is the caller/client …
  - And second is the callee/server.
- When executed, emits:

```
First
Second
Third
```


Such control flow has many variations

- *E.g.*, if you have multiple CPUs:
  - Could give each a separate task to do, while you keep going with your own, and all rendezvous at the end.
  - Such independent executions are a form of parallelism.
- *E.g.*, if you have multiple threads on a single CPU:
  - Akin to parallelism, but now tasks can synchronize with each other to share common resources.
  - Such cooperative executions are a form of concurrency.
- Exploring these (and more!) would need many hours:
  - So today we'll focus on the classical essentials only, …
  - And leave such special topics to other talks/speakers.

Making a call is like hiring someone

- *E.g.*, imagine hiring an attorney:
  - The hiree (att'y) may specify certain documents needed from the hirer (you) to provide the requested service.
- Similarly, when applying a call operator:
  - Hirer/hiree ~ caller/callee ~ client/server relationship.
  - The callee may specify certain values needed from the caller to provide the requested service.
- The needed items are termed parameters, while the corresponding supplied items are the arguments:
  - Establishing the correspondence, usually early in a call, is termed parameter passage.

Parameter syntax

- In C++, the type of each callee param encodes both:
  - The kind of value required from the caller, and …
  - How that corresponding arg will be used.

Syntax	Nomenclature	Callee's promise
T	Call by (copy) value	Use a copy of the arg
T const &	Call by const ref	Only inspect the arg
T &, T &&	Call by non-const ref	May modify the arg

How is a call carried out? (since C++17; see [expr.call])

- A call expression ***L*** (***R***) evaluates ***L*** before ***R*** :
  - The left operand ***L*** is a *postfix-expression* (often just a name) that denotes a callable entity (often a function).
  - The right operand ***R*** is a sequence of *initializer-clause*s that will be evaluated in some non-overlapping order.
- To invoke/call ***L*** with arguments ***R*** means to apply the ( ) operator to those evaluated operands:
  - Each call yields control to ***L*** via a prolog, and …
  - Each call resumes CPU control via an epilog.
  - The details of this calling convention vary per platform.

Prolog: a call is initiated in the context of the caller/client

- Sufficient storage (a stack frame) is reserved (allocated) for all of the callee's automatic lifetime needs:
  - CPU registers may be allocated, too (but their contents may need first to be saved/spilled for restoration upon return).
  - This storage is used for parameters (explicitly declared + implicit/hidden), as well as for other fctn-local variables.
- Examples of implicit parameters:
  - A return address for resuming control.
  - this, a mbr fctn's pointer to its invoking object (*this).
  - A ptr/ref to storage for the call's result (unless void).

 13

More prolog: can't yet yield control

- Parameter passage now takes place by initializing each param's storage with the corresponding arg:
  - Each param p (of type P) is init'd from corresp arg a just as in a declaration P p = a; *// recall:* P *may be a ref type .*
  - (Recall that if a's type is not P, a temporary may be created to hold a's promoted/decayed/converted value.)
  - Each such init (arg eval + any side effects) is indeterminately sequenced with respect to any other param's init.
- The implicit parameters' storage is likewise init'd:
  - With arg's arranged/provided by the compiler.
  - But (of course) not yet the call's result.

 14

What can go wrong?

- An exception might arise during an init:
  - The prolog discontinues further parameter passage, ...
  - Destroys each successfully init'd param, …
  - Then reclaims all the stack frame storage.
- Now process the exception:
  - Search for a matching exception handler, starting in the caller (since we never yielded control to the callee), …
  - Init the handler's param with (a copy of) the exception object, …
  - Finally, yield control to the handler.

 15

Still not ready for the callee to have control (C++23?)

- Preconditions [[pre: ⋯]] are part of the contract between the caller and callee functions:
  - Each precondition must be satisfied (true) "immediately before starting [execution] of the [callee] function …."
  - Otherwise, the program cannot trust its state, so the contract violation handler function will be called.
- A violation handler typically ends program execution:
  - Why? Because a callee is not obliged to behave sanely when a caller fails to satisfy callee preconditions.
  - In the standard library, "Violation of any preconditions … results in undefined behavior" (*see* [res.on.required]).

 16

"There and Back Again" (apologies to J. R. R. Tolkien)

- Only now can control be yielded (flow) to the callee:
  - The callee begins execution with its 1st statement, ...
  - And continues with its own internal control flow …
  - Possibly making calls of its own along the way …
  - Until it executes a return statement to start the epilog.
- "The result of a fctn call is the result of the operand [if any] of the evaluated return stmt…" (*see* [expr.call]/9):
  - That result then initializes the implicit result parameter.
  - Certain optimizations (*e.g.*, RVO, NRVO, …) can reduce (sometimes significantly) the costs of that initialization.

 17

More epilog to conclude a call normally

- Before leaving the callee, d'tors are invoked for non-trivial locals (*i.e.*, for var's and temp's, not yet param's):
  - Happens in the reverse order of construction.
- Then the caller regains control (per the return address hidden parameter):
  - Now back in the caller's context (where they were constructed), param's (and their temp's) are destroyed.
    - Usually happens promptly for param's, but delayed until the end of the call's enclosing *full-expression* for temp's.
  - Finally, registers are restored, the stack frame storage is reclaimed, and the call is considered complete.

 18

Postconditions (C++23?)

- Postconditions are also part of the contract between the caller and callee functions:
  - "A postcondition is checked by evaluating its predicate immediately before returning control to the caller …
  - [By which time the] lifetime of local variables and temporaries [but not of parameters] has ended."
- If false, the program cannot trust its state, so the contract violation handler function will be called:
  - The caller is not obliged to behave sanely if the callee fails to satisfy its postconditions.
  - If the violation handler doesn't abort the program, whatever happens afterwards is undefined behavior.

 19

std:: fctns can conclude a call abnormally (decl'd in various hdrs)

- [[noreturn]] void exit(int exit_code);
  - Causes normal program termination w/ partial cleanup.
  - Similar to returning from main.
- [[noreturn]] void _Exit(int exit_code) noexcept;
  - Causes normal program termination w/out cleanup.
- [[noreturn]] void abort( ) noexcept;
  - Causes abnormal program termination w/out cleanup.
- [[noreturn]] void quick_exit(int exit_code) noexcept;
  - Causes quick program termination w/ partial cleanup.
- [[noreturn]] void terminate( ) noexcept;
  - Calls the current terminate_handler (abort by default).
  - Is implicitly called when exception handling fails.

 20

Callable Types and Their Call Syntax

We can invoke/call an entity of type F (or F& or F&&) …

① If F is a *fctn* type R (⋯) (also noexcept(⋯) since C++17).

② If F is a *ptr-to-fctn* type R (∗) (⋯).

③ If F is a *ptr-to-mbr-fctn* type R (C :: ∗) (⋯).

④ If F is a *class* type defining a non-static member that is a fctn call operator R F :: operator ( ) (⋯).

⑤ If F is a *class* type …
- With an implicit conversion fctn F :: operator G ( ) …
- Where G is a { *ptr*, *ref*, *or ref-to-ptr* } *-to-fctn* type.

 22

Speaking of function types …

- Can a fctn type be *cv*- and/or ref-qualified?
  - *E.g.*, using bizarre_t = int ( ) const &&; *// valid type?*
  - Syntactically yes, but (despite appearances) such a fctn type is neither a *cv*-qualified type nor a ref-qualified type!
- So std::is_const_v< F const > is false if F is a fctn type!
  - (Or if F is a reference type.)
- Such abominable types are rarely useful:
  - Except perhaps in torture-testing a compiler ...
  - And in implementing the std :: is_function type trait.

 23

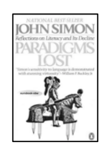
It is said …

"In the beginning was the word.

"But by the time
the second word was added to it,
there was trouble.

"For with it came syntax.…"

— John Simon,
1981



 24

Call syntax depends mostly on the callee's type

- If an entity g has (ref to) *fctn*, *ptr-to-fctn*, or *class* type, use traditional call operator syntax g(a_1 , a_2 , ⋯ , a_N).
- But if C::g is a non-static mbr fctn, the hidden this param must be init'd to point to the invoking object:
  - ① When decltype(a_1) is (or inherits from) type C, use call syntax a_1 . g (a_2 , ⋯ , a_N).
  - ② Otherwise (*e.g.*, decltype(a_1) is a ptr/smart-ptr type), use call syntax ((∗a_1) . g) (a_2 , ⋯ , a_N).
- Similarly, if g has *ptr-to-mbr-fctn* type:
  - Use call syntax (a_1 . ∗ g) (a_2 , ⋯ , a_N) for case ①, and …
  - Use call syntax ((∗a_1) . ∗ g) (a_2 , ⋯ , a_N) for case ②.

Copyability of callables

If decltype(g) is a …	Then g is a …	Is g copyable?
fctn type	function	no, but will decay
ptr-to-fctn type or *ptr-to-mbr-fctn* type	variable	yes, natively so
class type such that g(⋯) is a valid expr	variable, a.k.a. function object	depends on the type's copy/move mbr fctns
reference type	variable that's not reseatable	no, but can initialize another variable

But why do we care about copyability?

Callbacks

Example

- Let's use this function:
  ```
  bool is_even( int k ) { return k % 2 == 0; }
  ```
- … in the following context:
  ```
  int a[N] = { ⋯ };
  std::partition( a + 0, a + N, is_even ); // ⇒ &is_even
  ```
- … in order to rearrange the array's contents, placing all the even values ahead of all the odd values:
  - For each array item a[k], the partition algorithm calls is_even(a[k]) to decide where the item belongs.
  - Executable code (*e.g.*, a function) that is passed as an arg to other code (*e.g.*, to an algorithm) is termed a callback.

How do callbacks work in C++?

- While functions can't be copied, they decay into ptr-to-fctn values that can be copied (and called, too).
- *E.g.*, here's an int-specific partition algorithm:
  ```
  void partition( int ∗ b, int ∗ e   // e is past-the-last
                , bool belongs_in_front( int ) // bool(∗)(int)
                ) {
    for( ; ; ) { // do find_if_not, then do rfind_if
      while( b != e and     belongs_in_front(∗ b) ) ++b;
      while( b != e and not belongs_in_front(∗ −−e) ) ;
      if ( b == e ) return;
      std::swap_iter( b++, e );
    }
  }
  ```


Equivalently, using named helper algorithms …

- Given helpers find_if_not and rfind_if, our int-specific partition algorithm gains a much simpler structure:
  ```
  void partition( int ∗ b, int ∗ e
                , bool belongs_in_front( int )
                ) {
    while( b = find_if_not( b, e, belongs_in_front )
         , e = rfind_if( b, e, belongs_in_front )
         , b != e )
      swap_iter( b++, e );
  }
  ```
- Note that partition forwarded its callback to those helper algorithms for their use.

Possible generic helpers (C++20-style)

- Algorithm find_if_not:
  - template< InputIterator Iter, Predicate<⋯> Pred >
    Iter find_if_not(Iter b, Iter e, Pred p) { *// e is past-the-last*
    for(; b != e and p(∗ b); ++b) ;
    return b;
    }
- Algorithm rfind_if:
  - template< BidirectionalIterator Iter, Predicate<⋯> Pred >
    Iter rfind_if(Iter b, Iter e, Pred p) { *// e is past-the-last*
    while(b != e and not p(∗ −−e)) ;
    return b;
    }

 31

So what characterizes a callback?

- A callback is an entity such that:
  - Its type F satisfies invocable<F, ⋯> (C++20).
  - It is supplied to an algorithm, usually as an argument of known or deduced type.
  - The algorithm then invokes the callback as needed, …
  - Thereby tailoring the algorithm's behavior with respect to the client's data.
- The callback thus serves as the go-between:
  - It has/provides the client-specific details …
  - That the general algorithm lacks.

 32

C++ callbacks have a long history

- *E.g.*, in the C library headers <stdlib.h>/<cstdlib>:
  - void* qsort(void ∗ ptr, size_t count, size_t size
    , int comp(void const ∗, void const ∗));
  - void* bsearch(void const ∗ key
    , void const ∗ ptr, size_t count, size_t size
    , int comp(void const ∗, void const ∗));
- *E.g.*, in the C++ library header <exception>:
  - using terminate handler = void (∗) ( );
  - terminate handler
    set_terminate(terminate handler h) noexcept;

 33

INVOKE and std::invoke (C++11/C++17, resp.)

- For specification purposes, the std lib denotes the call-syntax selection algorithm as *INVOKE(g, a_1 , ⋯, a_N)*:
  - Provides a uniform interface to all callables/invokables.
- *INVOKE* exists only notionally, but std::invoke is real:
  - template< class G, class... A >
    constexpr invoke_result_t< G, A... > invoke(G&& g, A&&... a)
    noexcept(is nothrow_invocable_v<G, A...>);
- Each also correctly handles ref types for *g* and for a_1:
  - To decide the correct call syntax, ref types F&/F&& and std::reference_wrapper<F> types are treated as type F.
  - And if a_1 has a std::reference_wrapper< ⋯ > type, it is unwrapped (via .get( )) in the selected call syntax.

 34

The std::invocable concepts (C++20)

- template< class G, class... A >
  concept invocable = requires(G&& g, A&&... a) {
  invoke(forward<G>(g) , forward<A>(a)...);
  *// **not** required to be equality-preserving*
  };
- template< class G, class... A >
  concept regular_invocable = invocable<G, A…>;
  *// **is** required to be equality-preserving*
- An expression is said to be equality-preserving iff:
  - Multiple evaluations with identical inputs/operands …
  - Always give the same outputs/results/side effects.
  - *E.g.*, a URBG satisfies invocable, not regular_invocable.

 35

Function Objects and Their Types

Characteristics of C++ function objects

- Any function object (*e.g.*, a ptr-to-fctn) is used both:
  - As a value (construct/copy/destroy it), and …
  - As a callable entity (invoke it to get a result + side effects).
- Thus, a function object of class type typically has:
  ① A default c'tor if it's stateless (no data members), else a c'tor that initializes its state (data members), …
  ② A copy c'tor (and maybe also a move c'tor) , …
  ③ A d'tor, and …
  ④ At least one member fctn (could be a fctn template) named operator ( ). (A suitable conv op will do, too.)

Example (C++20)

- A (stateful) function object type:
  - ```
    class is_less {
       int n;
    public:
       is_less( int n = 0 ) : n{ n } { } // serves as default c'tor
       // implicitly copyable and destructible
       bool operator( ) ( int val ) const noexcept
       { return val < n; }
    };
    ```
- A function object, instantiated for use as a callback:
  - ```
    int a[N] = { ··· };
    std::partition( a + 0, a + N, is_less{ 42 } );
    ```


Function object types in the standard library

- Header <functional> provides these fctn object types:
  - plus, minus, multiplies, divides, modulus, negate;
  - equal_to, not_equal_to;
  - greater, less, greate_equal, less_equal;
  - logical_and, logical_or, logical_not;
  - bit_and, bit_or, bit_xor, bit_not;
- These are often instantiated for use with <algorithm>:
  - ```
    std::greater<> gt;
    std::sort( from, upto, gt );     // sort will call fctn object gt
    ```
  - ```
    std::sort( from, upto, std::greater<>( ) );     // prefer { }
          // sort will call the (anonymous, temporary) fctn object
    ```


How to convert a fctn obj type to a ptr-to-fctn type

- ```
  struct Doubler {
     using fctn_t = int ( int ); // convenient alias, not essential
     constexpr int operator ( ) ( int k ) const noexcept
        { return 2 * k; }
     static constexpr int decayed( int k ) noexcept
        { return Doubler{ }( k ); }
     constexpr operator fctn_t * ( ) const noexcept
        { return &decayed; }
  };
  ```
- ```
  int (*fp) ( int ) = Doubler{ }; // uses conversion operator
  ··· fp( 3 ) ··· // yields 6
  ```


std::function as a universal callable handle

- ```
  template< class > class function; // not defined in general
  ```
- ```
  template< class R, class... P >
  class function< R ( P... ) > {       // template arg is a fctn type
     R operator( ) ( P... ) const;     // behaves as INVOKE does
     ⋮
  };
  ```
- Think of it as a generalized ptr-to-fctn type:
  - Because it can be initialized from any callable that …
  a) Obtains a result of (or convertible to) type R, and …
  b) Takes arg's that can init param's of types P... .

Example: using std::function as a callback parameter

- Revising the earlier int-specific partition algorithm:
  - ```
    void partition( int * b, int * e // e is past-the-last
                  , bool belongs_in_front( int ) // bool(*)(int)
                  , std::function<bool( int )> belongs_in_front
                  ) {
       ⋮ // algorithm's body is unchanged
    }
    ```
- This still accepts a callback arg of ptr-to-fctn type:
  - But an arg of any fctn obj type is equally accepted, …
  - As long as that arg is int-taking and bool-returning.
    ```

Another example: a container of callables

- void fctn( ) { cout << "I'm an ordinary function.\n"; }
- struct obj {
    void operator( ) ( ) const
    { cout << "I'm a stateless function object.\n"; }
  };
- struct state {
    int n = 0; *// stateful*
    void operator( ) ( ) const
    { cout << "Times I've been called: " << ++n << ".\n"; }
  };
- *// a container of callables, each taking/returning nothing:*
  vector< function<void( )> > v{ &fctn, obj{ }, stateful{ } };
  for(auto&& g : v) g( ); *// invoke each callable in* v

 43

Proposed for C++next

- std::function_ref:
  - Designed as a (non-owning) reference to a Callable.
  - Interface very much like that of std::function.
- Like any reference (or handle or view) type:
  - Has the classical coordination of lifetime issue arises.
  - Must keep the Callable alive while the function_ref lives.
  - Temporary Callables are especially vulnerable.

 44

Callbacks Generalized: Higher-Order Functions

Characteristics

- A higher-order function is a ~~function~~ callable that:
  - Has at least one callback parameter, and/or …
  - Has a result of a callable type.
- Mathematics is full of higher-order functions:
  - Prominent examples include differentiation and integration operations, each of which is a function …
  - That produces a function as its result, having taken a function as its parameter.
- In computer programming languages, higher-order functions have been around since LISP (1958):
  - Modern LISP is based on lambda calculus (Church, 1936).

 46

Example: composition of callables

- using fctn_t = std::function<int(int)>;
- struct Composer {
    fctn_t f1, f2;
    Composer(fctn_t f1, fctn_t f2) f1(f1), f2(f2) { }
    int operator( ) (int k) { return f1(f2(k)); }
  };
- int double(int k) { return 2 ∗ k; }
  int triple(int k) { return 3 ∗ k; }
- static_assert(Composer{ double, triple} (5)
      == 30); // == 2 ∗ (3 ∗ 5)

 47

Function Objects in Disguise: Lambda Expressions and Their Closures

C++ lambda expression syntax

- The fundamental syntax is [⋯] (⋯) { ⋯ } :
  - Brackets provide for captures. (Stay tuned!)
  - Parentheses provide for function parameters.
  - Braces provide for a function body.
- C++20's full syntax also permits such extra items as:
  - A return type and/or an exception specification, …
  - Constrained function parameters, …
  - Template parameters (optionally constrained), …
  - And more.

 49

What can we do with a C++ lambda?

- We can initialize a variable with one:
  - *E.g.*, auto greater = [ ] (int a, int b) { return a > b; } ;
    bool b = greater(1, 2); *// later*
- We can pass one as a callback:
  - *E.g.*, std::sort(from, upto
    , [ ] (int a, int b) { return a > b; }) ;
- We can return one from a function (*e.g.*, factory-style):
  - *E.g.*, auto make( )
    { return [ ] (int a, int b) { return a > b; } ; }
    auto greater = make( );
- We can even invoke one immediately:
  - *E.g.*, bool b = [ ] (int a, int b) { return a > b; } (1, 2);

 50

A lambda is a kind of expression, nothing more

- Like any expression, a *lambda-expression* is evaluated:
  - Such evaluation produces a temporary function object termed a closure (closure object).
  - *I.e.*, it has class type, with a public operator ( ) member and most/all of the usual special member functions.
  - Thus, a closure has both value and callable semantics.
- It's always the closure that serves:
  - As the initializer in a declaration, or …
  - As the callback (argument) in a function call, or …
  - As the callee in a call expression, or …
  - As the result in a return statement.

 51

A closure's type

- What's the type of this lambda's closure?
  - auto greet = [ ] ( ) { std::cout << "Hello!"; };
  - struct { *// 1st approximation; more detail forthcoming*
    inline void operator( ) ( ) const { std::cout << "Hello!"; }
    } greet;
- Each closure has a uniquely named type:
  - That type's true name is known to the compiler only.
  - The type is nonetheless deducible (as usual) via auto or via a template type parameter.
  - Can also use decltype(greet), if there's a need.

 52

Calling a closure

- A variable of a closure type is called exactly as we would call any other function object:
  - ⋯ greet ( ) ⋯
- An anonymous closure is called only immediately:
  - ⋯ [ ] ( ) { std::cout << "Hello!"; } ( ) ⋯
- Either can be passed as a callback argument:
  - template< class G >
    void emit (G g) { g ( ); }
  - ⋯ emit(greet) ⋯
  - ⋯ emit([ ] ( ) { std::cout << "Hello!"; }) ⋯

 53

Return type?

- The lambda's return type can often be deduced:
  - Exactly as is done for a function with auto return type.
- No deduction if there's a late-specified return type:
  - Placed after the parameters') and before the body's {.
  - Exactly as is done for a function with auto return type.
- Example:
  - auto greet = [ ] ( )
    -> void
    { std::cout << "Hello!"; };
  - More items syntactically permitted there.

 54

The body of a lambda

- C++11 limited the body to a single stmt, a return.
  - Since, most limitations have been lifted.
  - But there remains an inherent scope/lifetime issue.
- Although a closure starts out as a temporary:
  - Like any copyable value, it can be saved for later use, …
  - Possibly from an entirely different scope, ...
  - Thus outliving any nonlocal variables used in its body!
- When a closure outlives the lifetime of variables it uses when called, Bad Things usually ensue.

Problematic example (doesn't compile)

- Note the lifetimes of parameter pi and closure L :
  - ```
    auto get_work( double pi ) {  // automatic lifetime …
        auto L = [ ] ( ) { return pi; } ;  // … leads to …
        return L;
    }
    ```
  - ```
    auto work = get_work( 3.14 );
    auto answer2 = work( );        // … undefined behavior!
    ```
- C++ forbids a lambda to use any automatic-lifetime variable from an enclosing scope:
  - Unless it is first captured, by value or by reference, …
  - Much like a function argument is passed.

Captures

- The contents of a *lambda-introducer* (the introductory brackets) specify the lambda's captures:
  - For each captured variable, the compiler injects a corresponding data member into the closure type.
  - If the brackets are empty, the lambda has no captures and its closure is described as stateless.
  - Otherwise, the closure is described as stateful.
- Captures can be specified individually or in bulk:
  - If individually, there will be 1 data member per capture.
  - If in bulk, the compiler will determine how many data members are needed.

Capture syntax for stateful closures (partial list; most common only)

Syntax (between [ ])	**Will inject** (into the closure) …	**Since**
=		
Identifier	A copy of the spec'd automatic var	C++11
&		
&		
&		

Example of captures (C++20)

- ```
  auto echo( auto x ) {
      auto C1 = [ = ] ( )  // by-value capture in bulk
                { return x; };
      auto C2 = [ y = C1( ) ] ( )  // single init-capture
                { return y; };
      return C2( );  // just returns x
  }
  ```
- ```
  int main( ) {
      return echo( 6 );  // tedious form of exit status 6
  }
  ```


Closure types' special member functions have evolved

- Recall that C++'s special member functions are those class c'tors, d'tor, and assignment op's that a compiler can sometimes generate for us.
-

Spec mbr fctn	C++11	C++14/17, C++20 stateful	C++20 stateless
destructor	= default	= default	= default
default c'tor	= delete	*none*	= default
copy c'tor, move c'tor	"implicitly declared"	= default	= default
copy assign	= delete	= delete	= default
move assign	*not declared*	*not declared*	= default

An Advanced Example

Scenario

- Suppose I will want to make a call w(a_1, ⋯, a_n):
  - Of the *n* arg's, I already know a_1, ⋯, a_k now, but …
  - Only later will I know the remaining a_{k+1}, ⋯, a_n.
- I want to combine (bind) w with the *k* known arg's:
  - Producing an equivalent new callable (say, b) that has the known a_1, ⋯, a_k hard-wired/built-in.
  - *I.e.*, auto b = combiner(w, a_1, ⋯, a_k);
  - Later, once the remaining a_{k+1}, ⋯, a_n are known, I can call b(a_{k+1}, ⋯, a_n) and have the effect of calling w(a_1, ⋯, a_n).
  - C++20 has such a combiner, std::bind_front.

A form of currying (C++20)

```
template< class callable_t, class… bound_t >
auto my_bind_front( callable_t && c, bound_t &&… b ) {
  return [   c = c
         , … b = b
         ]
         ( auto && … free )  // mutable? noexcept?
         -> decltype(auto)
         { return c( b …
                   , free …
                   ) ;
         } ;
}
```


A form of currying ② (fwd ⇒ std::forward) (C++20)

```
template< class callable_t, class… bound_t >
auto my_bind_front( callable_t && c, bound_t &&… b ) {
  return [   c = fwd<callable_t>( c )
         , … b = fwd<bound_t>( b )
         ]
         ( auto && … free )  // mutable? noexcept?
         -> decltype(auto)
         { return c( fwd<bound_t>( b ) …
                   , fwd<decltype(free)>( free ) …
                   ) ;
         } ;
}
```


A form of currying ③ (invoke ⇒ std::invoke) (C++20)

```
template< class callable_t, class… bound_t >
auto my_bind_front( callable_t && c, bound_t &&… b ) {
  return [   c = fwd<callable_t>( c )
         , … b = fwd<bound_t>( b )
         ]
         ( auto && … free )  // mutable? noexcept?
         -> decltype(auto)
         { return invoke( fwd<callable_t>( c )
                        , fwd<bound_t>( b ) …
                        , fwd<decltype(free)>( free ) …
                        ) ;
         } ;
}
```


From Functions to Lambdas:

How Do C++ Callables *Really* Work?

FIN

WALTER E. BROWN, PH.D.

< webrown.cpp @ gmail.com >