WIKIPEDIA

Closure (computer programming)

In programming languages, a **closure**, also **lexical closure** or **function closure**, is a technique for implementing lexically scoped name binding in a language with first-class functions. Operationally, a closure is a record storing a function[a] together with an environment.[1] The environment is a mapping associating each free variable of the function (variables that are used locally, but defined in an enclosing scope) with the value or reference to which the name was bound when the closure was created.[b] Unlike a plain function, a closure allows the function to access those *captured variables* through the closure's copies of their values or references, even when the function is invoked outside their scope.

Contents

History and etymology

The concept of closures was developed in the 1960s for the mechanical evaluation of expressions in the λ-calculus and was first fully implemented in 1970 as a language feature in the PAL programming language to support lexically scoped first-class functions.[2]

Peter J. Landin defined the term *closure* in 1964 as having an *environment part* and a *control part* as used by his SECD machine for evaluating expressions.[3] Joel Moses credits Landin with introducing the term *closure* to refer to a lambda expression whose open bindings (free variables) have been closed by (or bound in) the lexical environment, resulting in a *closed expression*, or closure.[4][5] This usage was subsequently adopted by Sussman and Steele when they defined Scheme in 1975,[6] a lexically scoped variant of Lisp, and became widespread.

Sussman and Abelson also use the term *closure* in the 1980s with a second, unrelated meaning: the property of an operator that adds data to a data structure to also be able to add nested data structures. This usage of the term comes from the mathematics usage rather than the prior usage in computer science. The authors consider this overlap in terminology to be "unfortunate."[7]

Anonymous functions

The term *closure* is often used as a synonym for anonymous function, though strictly, an anonymous function is a function literal without a name, while a closure is an instance of a function, a value, whose non-local variables have been bound either to values or to storage locations (depending on the language; see the lexical environment section below).

For example, in the following Python code:

```python
def f(x):
    def g(y):
        return x + y
    return g  # Return a closure.

def h(x):
    return lambda y: x + y  # Return a closure.

# Assigning specific closures to variables.
a = f(1)
b = h(1)

# Using the closures stored in variables.
assert a(5) == 6
assert b(5) == 6

# Using closures without binding them to variables first.
assert f(1)(5) == 6  # f(1) is the closure.
assert h(1)(5) == 6  # h(1) is the closure.
```

the values of `a` and `b` are closures, in both cases produced by returning a nested function with a free variable from the enclosing function, so that the free variable binds to the value of parameter `x` of the enclosing function. The closures in `a` and `b` are functionally identical. The only difference in implementation is that in the first case we used a nested function with a name, `g`, while in the second case we used an anonymous nested function (using the Python keyword `lambda` for creating an anonymous function). The original name, if any, used in defining them is irrelevant.

A closure is a value like any other value. It does not need to be assigned to a variable and can instead be used directly, as shown in the last two lines of the example. This usage may be deemed an "anonymous closure".

The nested function definitions are not themselves closures: they have a free variable which is not yet bound. Only once the enclosing function is evaluated with a value for the parameter is the free variable of the nested function bound, creating a closure, which is then returned from the enclosing function.

Lastly, a closure is only distinct from a function with free variables when outside of the scope of the non-local variables, otherwise the defining environment and the execution environment coincide and there is nothing to distinguish these (static and dynamic binding cannot be distinguished because the names resolve to the same values). For example, in the below program, functions with a free variable `x` (bound to the non-local variable `x` with global scope) are executed in the same environment where `x` is defined, so it is immaterial whether these are actually closures:

```
x = 1
nums = [1, 2, 3]

def f(y):
    return x + y

map(f, nums)
map(lambda y: x + y, nums)
```

This is most often achieved by a function return, since the function must be defined within the scope of the non-local variables, in which case typically its own scope will be smaller.

This can also be achieved by variable shadowing (which reduces the scope of the non-local variable), though this is less common in practice, as it is less useful and shadowing is discouraged. In this example `f` can be seen to be a closure because `x` in the body of `f` is bound to the `x` in the global namespace, not the `x` local to `g`:

```
x = 0

def f(y):
    return x + y

def g(z):
    x = 1  # local x shadows global x
    return f(z)

g(1)  # evaluates to 1, not 2
```

Applications

The use of closures is associated with languages where functions are first-class objects, in which functions can be returned as results from higher-order functions, or passed as arguments to other function calls; if functions with free variables are first-class, then returning one creates a closure. This includes functional programming languages such as Lisp and ML, as well as many modern, multi-paradigm languages, such as Python and Rust. Closures are also frequently used with callbacks, particularly for event handlers, such as in JavaScript, where they are used for interactions with a dynamic web page.

Closures can also be used in a continuation-passing style to hide state. Constructs such as objects and control structures can thus be implemented with closures. In some languages, a closure may occur when a function is defined within another function, and the inner function refers to local variables of the outer function. At run-time, when the outer function executes, a closure is formed, consisting of the inner function's code and references (the upvalues) to any variables of the outer function required by the closure.

First-class functions

Closures typically appear in languages with first-class functions—in other words, such languages enable functions to be passed as arguments, returned from function calls, bound to variable names, etc., just like simpler types such as strings and integers. For example, consider the following Scheme function:

```
; Return a list of all books with at least THRESHOLD copies sold.
(define (best-selling-books threshold)
  (filter
    (lambda (book)
      (>= (book-sales book) threshold))
    book-list))
```

In this example, the lambda expression `(lambda (book) (>= (book-sales book) threshold))` appears within the function `best-selling-books`. When the lambda expression is evaluated, Scheme creates a closure consisting of the code for the lambda expression and a reference to the `threshold` variable, which is a free variable inside the lambda expression.

The closure is then passed to the `filter` function, which calls it repeatedly to determine which books are to be added to the result list and which are to be discarded. Because the closure itself has a reference to `threshold`, it can use that variable each time `filter` calls it. The function `filter` itself might be defined in a completely separate file.

Here is the same example rewritten in JavaScript, another popular language with support for closures:

```
// Return a list of all books with at least 'threshold' copies sold.
function bestSellingBooks(threshold) {
  return bookList.filter(
      function (book) { return book.sales >= threshold; }
    );
}
```

The `function` keyword is used here instead of `lambda`, and an `Array.filter` method[8] instead of a global `filter` function, but otherwise the structure and the effect of the code are the same.

A function may create a closure and return it, as in the following example:

```
// Return a function that approximates the derivative of f
// using an interval of dx, which should be appropriately small.
function derivative(f, dx) {
  return function (x) {
    return (f(x + dx) - f(x)) / dx;
  };
}
```

Because the closure in this case outlives the execution of the function that creates it, the variables `f` and `dx` live on after the function `derivative` returns, even though execution has left their scope and they are no longer visible. In languages without closures, the lifetime of an automatic local variable coincides with the execution of the stack frame where that variable is declared. In languages with closures, variables must continue to exist as long as any existing closures have references to them. This is most commonly implemented using some form of garbage collection.

State representation

A closure can be used to associate a function with a set of "private" variables, which persist over several invocations of the function. The scope of the variable encompasses only the closed-over function, so it cannot be accessed from other program code. These are analogous to private variables in object-oriented programming, and in fact closures are analogous to a type of object, specifically function objects, with a single public method (function call), and possibly many private variables (the closed-over variables).

In stateful languages, closures can thus be used to implement paradigms for state representation and information hiding, since the closure's upvalues (its closed-over variables) are of indefinite extent, so a value established in one invocation remains available in the next. Closures used in this way no longer have referential transparency, and are thus no longer pure functions; nevertheless, they are commonly used in impure functional languages such as Scheme.

Other uses

Closures have many uses:

- Because closures delay evaluation—i.e., they do not "do" anything until they are called—they can be used to define control structures. For example, all of Smalltalk's standard control structures, including branches (if/then/else) and loops (while and for), are defined using objects whose methods accept closures. Users can easily define their own control structures also.
- In languages which implement assignment, multiple functions can be produced that close over the same environment, enabling them to communicate privately by altering that environment. In Scheme:

```
(define foo #f)
(define bar #f)

(let ((secret-message "none"))
  (set! foo (lambda (msg) (set! secret-message msg)))
  (set! bar (lambda () secret-message)))

(display (bar)) ; prints "none"
(newline)
(foo "meet me by the docks at midnight")
(display (bar)) ; prints "meet me by the docks at midnight"
```

- Closures can be used to implement object systems.[9]

Note: Some speakers call any data structure that binds a lexical environment a closure, but the term usually refers specifically to functions.

Implementation and theory

Closures are typically implemented with a special data structure that contains a pointer to the function code, plus a representation of the function's lexical environment (i.e., the set of available variables) at the time when the closure was created. The referencing environment binds the non-local names to the corresponding variables in the lexical environment at the time the closure is created, additionally extending their lifetime to at least as long as the lifetime of the closure itself. When the closure is entered at a later time, possibly with a different lexical environment, the function is executed with its non-local variables referring to the ones captured by the closure, not the current environment.

A language implementation cannot easily support full closures if its run-time memory model allocates all automatic variables on a linear stack. In such languages, a function's automatic local variables are deallocated when the function returns. However, a closure requires that the free variables it references survive the enclosing function's execution. Therefore, those variables must be allocated so that they persist until no longer needed, typically via heap allocation, rather than on the stack, and their lifetime must be managed so they survive until all closures referencing them are no longer in use.

This explains why, typically, languages that natively support closures also use garbage collection. The alternatives are manual memory management of non-local variables (explicitly allocating on the heap and freeing when done), or, if using stack allocation, for the language to accept that certain use cases will lead to undefined behaviour, due to dangling pointers to freed automatic variables, as in lambda expressions in C++11[10] or nested functions in GNU C.[11] The funarg problem (or "functional argument" problem) describes the difficulty of implementing functions as first class objects in a stack-based programming language such as C or C++. Similarly in D version 1, it is assumed that the programmer knows what to do with delegates and automatic local variables, as their references will be invalid after return from its definition scope (automatic local variables are on the stack) – this still permits many useful functional patterns, but for complex cases needs explicit heap allocation for variables. D version 2 solved this by detecting which variables must be stored on the heap, and performs automatic allocation. Because D uses garbage collection, in both versions, there is no need to track usage of variables as they are passed.

In strict functional languages with immutable data (*e.g.* Erlang), it is very easy to implement automatic memory management (garbage collection), as there are no possible cycles in variables' references. For example, in Erlang, all arguments and variables are allocated on the heap, but references to them are additionally stored on the stack. After a function returns, references are still valid. Heap cleaning is done by incremental garbage collector.

In ML, local variables are lexically scoped, and hence define a stack-like model, but since they are bound to values and not to objects, an implementation is free to copy these values into the closure's data structure in a way that is invisible to the programmer.

Scheme, which has an ALGOL-like lexical scope system with dynamic variables and garbage collection, lacks a stack programming model and does not suffer from the limitations of stack-based languages. Closures are expressed naturally in Scheme. The lambda form encloses the code, and the free variables of its environment persist within the program as long as they can possibly be accessed, and so they can be used as freely as any other Scheme expression.

Closures are closely related to Actors in the Actor model of concurrent computation where the values in the function's lexical environment are called *acquaintances*. An important issue for closures in concurrent programming languages is whether the variables in a closure can be updated and, if so, how these updates can be synchronized. Actors provide one solution.[12]

Closures are closely related to function objects; the transformation from the former to the latter is known as defunctionalization or lambda lifting; see also closure conversion.

Differences in semantics

Lexical environment

As different languages do not always have a common definition of the lexical environment, their definitions of closure may vary also. The commonly held minimalist definition of the lexical environment defines it as a set of all bindings of variables in the scope, and that is also what closures in any language have to capture.

However the meaning of a variable binding also differs. In imperative languages, variables bind to relative locations in memory that can store values. Although the relative location of a binding does not change at runtime, the value in the bound location can. In such languages, since closure captures the binding, any operation on the variable, whether done from the closure or not, are performed on the same relative memory location. This is often called capturing the variable "by reference". Here is an example illustrating the concept in ECMAScript, which is one such language:

```
// ECMAScript
var f, g;
function foo() {
  var x;
  f = function() { return ++x; };
  g = function() { return --x; };
  x = 1;
  alert('inside foo, call to f(): ' + f());
}
foo();  // 2
alert('call to g(): ' + g());  // 1 (--x)
alert('call to g(): ' + g());  // 0 (--x)
alert('call to f(): ' + f());  // 1 (++x)
alert('call to f(): ' + f());  // 2 (++x)
```

Function `foo` and the closures referred to by variables `f` and `g` all use the same relative memory location signified by local variable `x`.

In some instances the above behaviour may be undesirable, and it is necessary to bind a different lexical closure. Again in ECMAScript, this would be done using the `Function.bind()`.

Example 1: Reference to an unbound variable

[13]

```
var module = {
  x: 42,
  getX: function() {return this.x; }
}
var unboundGetX = module.getX;
console.log(unboundGetX()); // The function gets invoked at the global scope
// emits undefined as 'x' is not specified in global scope.

var boundGetX = unboundGetX.bind(module); // specify object module as the closure
console.log(boundGetX()); // emits 42
```

Example 2: Accidental reference to a bound variable

For this example the expected behaviour would be that each link should emit its id when clicked; but because the variable 'e' is bound the scope above, and lazy evaluated on click, what actually happens is that each on click event emits the id of the last element in 'elements' bound at the end of the for loop.[14]

```
var elements= document.getElementsByTagName('a');
//Incorrect: e is bound to the function containing the 'for' loop, not the closure of
"handle"
for (var e of elements){ e.onclick=function handle(){ alert(e.id);} }
```

Again here variable `e` would need to be bound by the scope of the block using `handle.bind(this)` or the `let` keyword.

On the other hand, many functional languages, such as ML, bind variables directly to values. In this case, since there is no way to change the value of the variable once it is bound, there is no need to share the state between closures—they just use the same values. This is often called capturing the variable "by value". Java's local and anonymous classes also fall into this category—they require captured local variables to be `final`, which also means there is no need to share state.

Some languages enable you to choose between capturing the value of a variable or its location. For example, in C++11, captured variables are either declared with `[&]`, which means captured by reference, or with `[=]`, which means captured by value.

Yet another subset, lazy functional languages such as Haskell, bind variables to results of future computations rather than values. Consider this example in Haskell:

```
-- Haskell
foo :: Fractional a => a -> a -> (a -> a)
foo x y = (\z -> z + r)
          where r = x / y

f :: Fractional a => a -> a
f = foo 1 0

main = print (f 123)
```

The binding of `r` captured by the closure defined within function `foo` is to the computation `(x / y)`—which in this case results in division by zero. However, since it is the computation that is captured, and not the value, the error only manifests itself when the closure is invoked, and actually attempts to use the captured binding.

Closure leaving

Yet more differences manifest themselves in the behavior of other lexically scoped constructs, such as `return`, `break` and `continue` statements. Such constructs can, in general, be considered in terms of invoking an escape continuation established by an enclosing control statement (in case of `break` and `continue`, such interpretation requires looping constructs to be considered in terms of recursive function calls). In some languages, such as ECMAScript, `return` refers to the continuation established by the closure lexically innermost with respect to the statement—thus, a `return` within a closure transfers control to the code that called it. However, in Smalltalk, the superficially similar operator `^` invokes the escape continuation established for the method invocation, ignoring the escape continuations of any intervening nested closures. The escape continuation of a particular closure can only be invoked in Smalltalk implicitly by reaching the end of the closure's code. The following examples in ECMAScript and Smalltalk highlight the difference:

```
"Smalltalk"
foo
  | xs |
  xs := #(1 2 3 4).
  xs do: [:x | ^x].
  ^0
bar
  Transcript show: (self foo printString) "prints 1"
```

```
// ECMAScript
function foo() {
  var xs = [1, 2, 3, 4];
  xs.forEach(function (x) { return x; });
  return 0;
}
alert(foo()); // prints 0
```

The above code snippets will behave differently because the Smalltalk `^` operator and the JavaScript `return` operator are not analogous. In the ECMAScript example, `return x` will leave the inner closure to begin a new iteration of the `forEach` loop, whereas in the Smalltalk example, `^x` will abort the loop and return from the method `foo`.

Common Lisp provides a construct that can express either of the above actions: Lisp `(return-from foo x)` behaves as Smalltalk `^x`, while Lisp `(return-from nil x)` behaves as JavaScript `return x`. Hence, Smalltalk makes it possible for a captured escape continuation to outlive the extent in which it can be successfully invoked. Consider:

```
"Smalltalk"
foo
    ^[ :x | ^x ]
bar
    | f |
    f := self foo.
    f value: 123 "error!"
```

When the closure returned by the method `foo` is invoked, it attempts to return a value from the invocation of `foo` that created the closure. Since that call has already returned and the Smalltalk method invocation model does not follow the spaghetti stack discipline to facilitate multiple returns, this operation results in an error.

Some languages, such as Ruby, enable the programmer to choose the way `return` is captured. An example in Ruby:

```
# Ruby

# Closure using a Proc
def foo
  f = Proc.new { return "return from foo from inside proc" }
  f.call # control leaves foo here
  return "return from foo"
end

# Closure using a lambda
def bar
  f = lambda { return "return from lambda" }
  f.call # control does not leave bar here
  return "return from bar"
end

puts foo # prints "return from foo from inside proc"
puts bar # prints "return from bar"
```

Both `Proc.new` and `lambda` in this example are ways to create a closure, but semantics of the closures thus created are different with respect to the `return` statement.

In Scheme, definition and scope of the `return` control statement is explicit (and only arbitrarily named 'return' for the sake of the example). The following is a direct translation of the Ruby sample.

```
; Scheme
(define call/cc call-with-current-continuation)

(define (foo)
    (call/cc
     (lambda (return)
       (define (f) (return "return from foo from inside proc"))
       (f) ; control leaves foo here
       (return "return from foo"))))

(define (bar)
  (call/cc
   (lambda (return)
     (define (f) (call/cc (lambda (return) (return "return from lambda"))))
     (f) ; control does not leave bar here
     (return "return from bar"))))

(display (foo)) ; prints "return from foo from inside proc"
(newline)
(display (bar)) ; prints "return from bar"
```

Closure-like constructs

Some languages have features which simulate the behavior of closures. In languages such as Java, C++, Objective-C, C#, VB.NET, and D, these features are the result of the language's object-oriented paradigm.

Callbacks (C)

Some C libraries support callbacks. This is sometimes implemented by providing two values when registering the callback with the library: a function pointer and a separate `void*` pointer to arbitrary data of the user's choice. When the library executes the callback function, it passes along the data pointer. This enables the callback to maintain state and to refer to information captured at the time it was registered with the library. The idiom is similar to closures in functionality, but not in syntax. The `void*` pointer is not type safe so this C idiom differs from type-safe closures in C#, Haskell or ML.

Callbacks are extensively used in GUI Widget toolkits to implement Event-driven programming by associating general functions of graphical widgets (menus, buttons, check boxes, sliders, spinners, etc.) with application-specific functions implementing the specific desired behavior for the application.

Nested function and function pointer (C)

With a gcc extension, a nested function (https://gcc.gnu.org/onlinedocs/gcc/Nested-Functions.html) can be used and a function pointer can emulate closures, providing the function does not exit the containing scope. The following example is invalid because `adder` is a top-level definition (depending by compiler version, it could produce a correct result if compiled without optimization, i.e. at `-O0`):

```
#include <stdio.h>

typedef int (*fn_int_to_int)(int); // type of function int->int

fn_int_to_int adder(int number) {
  int add (int value) { return value + number; }
  return &add; // & operator is optional here because the name of a function in C is a
pointer pointing on itself
}

int main(void) {
  fn_int_to_int add10 = adder(10);
  printf("%d\n", add10(1));
  return 0;
}
```

But moving `adder` (and, optionally, the `typedef`) in `main` makes it valid:

```
#include <stdio.h>

int main(void) {
  typedef int (*fn_int_to_int)(int); // type of function int->int

  fn_int_to_int adder(int number) {
    int add (int value) { return value + number; }
    return add;
  }

  fn_int_to_int add10 = adder(10);
  printf("%d\n", add10(1));
  return 0;
}
```

If executed this now prints `11` as expected.

Local classes and lambda functions (Java)

Java enables classes to be defined inside methods. These are called *local classes*. When such classes are not named, they are known as *anonymous classes* (or anonymous *inner* classes). A local class (either named or anonymous) may refer to names in lexically enclosing classes, or read-only variables (marked as `final`) in the lexically enclosing method.

```
class CalculationWindow extends JFrame {
    private volatile int result;
    // ...
    public void calculateInSeparateThread(final URI uri) {
        // The expression "new Runnable() { ... }" is an anonymous class implementing the
'Runnable' interface.
        new Thread(
            new Runnable() {
                void run() {
                    // It can read final local variables:
                    calculate(uri);
                    // It can access private fields of the enclosing class:
                    result = result + 10;
                }
            }
        ).start();
    }
}
```

The capturing of `final` variables enables you to capture variables by value. Even if the variable you want to capture is non-`final`, you can always copy it to a temporary `final` variable just before the class.

Capturing of variables by reference can be emulated by using a `final` reference to a mutable container, for example, a single-element array. The local class will not be able to change the value of the container reference itself, but it will be able to change the contents of the container.

With the advent of Java 8's lambda expressions,[15] the closure causes the above code to be executed as:

```
class CalculationWindow extends JFrame {
    private volatile int result;
    // ...
    public void calculateInSeparateThread(final URI uri) {
        // The code () -> { /* code */ } is a closure.
        new Thread(() -> {
            calculate(uri);
            result = result + 10;
        }).start();
    }
}
```

Local classes are one of the types of inner class that are declared within the body of a method. Java also supports inner classes that are declared as *non-static members* of an enclosing class.[16] They are normally referred to just as "inner classes".[17] These are defined in the body of the enclosing class and have full access to instance variables of the enclosing class. Due to their binding to these instance variables, an inner class may only be instantiated with an explicit binding to an instance of the enclosing class using a special syntax.[18]

```
public class EnclosingClass {
    /* Define the inner class */
    public class InnerClass {
        public int incrementAndReturnCounter() {
            return counter++;
        }
    }

    private int counter;
    {
        counter = 0;
    }

    public int getCounter() {
        return counter;
    }

    public static void main(String[] args) {
        EnclosingClass enclosingClassInstance = new EnclosingClass();
        /* Instantiate the inner class, with binding to the instance */
        EnclosingClass.InnerClass innerClassInstance =
            enclosingClassInstance.new InnerClass();

        for (int i = enclosingClassInstance.getCounter();
             (i = innerClassInstance.incrementAndReturnCounter()) < 10;
             /* increment step omitted */) {
            System.out.println(i);
        }
    }
}
```

Upon execution, this will print the integers from 0 to 9. Beware to not confuse this type of class with the nested class, which is declared in the same way with an accompanied usage of the "static" modifier; those have not the desired effect but are instead just classes with no special binding defined in an enclosing class.

As of Java 8, Java supports functions as first class objects. Lambda expressions of this form are considered of type `Function<T,U>` with T being the domain and U the image type. The expression can be called with its `.apply(T t)` method, but not with a standard method call.

```
public static void main(String[] args) {
    Function<String, Integer> length = s -> s.length();

    System.out.println( length.apply("Hello, world!") ); // Will print 13.
}
```

Blocks (C, C++, Objective-C 2.0)

Apple introduced blocks, a form of closure, as a nonstandard extension into C, C++, Objective-C 2.0 and in Mac OS X 10.6 "Snow Leopard" and iOS 4.0. Apple made their implementation available for the GCC and clang compilers.

Pointers to block and block literals are marked with `^`. Normal local variables are captured by value when the block is created, and are read-only inside the block. Variables to be captured by reference are marked with `__block`. Blocks that need to persist outside of the scope they are created in may need to be copied.[19][20]

```
typedef int (^IntBlock)();

IntBlock downCounter(int start) {
     __block int i = start;
     return [[ ^int() {
         return i--;
     } copy] autorelease];
}

IntBlock f = downCounter(5);
NSLog(@"%d", f());
NSLog(@"%d", f());
NSLog(@"%d", f());
```

Delegates (C#, VB.NET, D)

C# anonymous methods and lambda expressions support closure:

```
var data = new[] {1, 2, 3, 4};
var multiplier = 2;
var result = data.Select(x => x * multiplier);
```

Visual Basic .NET, which has many language features similar to those of C#, also supports lambda expressions with closures:

```
Dim data = {1, 2, 3, 4}
Dim multiplier = 2
Dim result = data.Select(Function(x) x * multiplier)
```

In D, closures are implemented by delegates, a function pointer paired with a context pointer (e.g. a class instance, or a stack frame on the heap in the case of closures).

```
auto test1() {
    int a = 7;
    return delegate() { return a + 3; }; // anonymous delegate construction
}

auto test2() {
    int a = 20;
    int foo() { return a + 5; } // inner function
    return &foo;  // other way to construct delegate
}

void bar() {
    auto dg = test1();
    dg();    // =10   // ok, test1.a is in a closure and still exists

    dg = test2();
    dg();    // =25   // ok, test2.a is in a closure and still exists
}
```

D version 1, has limited closure support. For example, the above code will not work correctly, because the variable a is on the stack, and after returning from test(), it is no longer valid to use it (most probably calling foo via dg(), will return a 'random' integer). This can be solved by explicitly allocating the variable 'a' on heap, or using structs or class to store all needed closed variables and construct a delegate from a method implementing the same code. Closures can be passed to other functions, as long as they are only used while the referenced values are still valid (for example calling another function with a closure as a callback parameter), and are useful for writing generic data processing code, so this limitation, in practice, is often not an issue.

This limitation was fixed in D version 2 - the variable 'a' will be automatically allocated on the heap because it is used in the inner function, and a delegate of that function can escape the current scope (via assignment to dg or return). Any other local variables (or arguments) that are not referenced by delegates or that are only referenced by delegates that do not escape the current scope, remain on the stack, which is simpler and faster than heap allocation. The same is true for inner's class methods that reference a function's variables.

Function objects (C++)

C++ enables defining function objects by overloading `operator()`. These objects behave somewhat like functions in a functional programming language. They may be created at runtime and may contain state, but they do not implicitly capture local variables as closures do. As of the 2011 revision, the C++ language also supports closures, which are a type of function object constructed automatically from a special language construct called *lambda-expression*. A C++ closure may capture its context either by storing copies of the accessed variables as members of the closure object or by reference. In the latter case, if the closure object escapes the scope of a referenced object, invoking its `operator()` causes undefined behavior since C++ closures do not extend the lifetime of their context.

```
void foo(string myname) {
    int y;
    vector<string> n;
    // ...
    auto i = std::find_if(n.begin(), n.end(),
               // this is the lambda expression:
               [&](const string& s) { return s != myname && s.size() > y; }
             );
    // 'i' is now either 'n.end()' or points to the first string in 'n'
    // which is not equal to 'myname' and whose length is greater than 'y'
}
```

Inline agents (Eiffel)

Eiffel includes inline agents defining closures. An inline agent is an object representing a routine, defined by giving the code of the routine in-line. For example, in

```
ok_button.click_event.subscribe (
    agent (x, y: INTEGER) do
        map.country_at_coordinates (x, y).display
    end
)
```

the argument to `subscribe` is an agent, representing a procedure with two arguments; the procedure finds the country at the corresponding coordinates and displays it. The whole agent is "subscribed" to the event type `click_event` for a certain button, so that whenever an instance of the event type occurs on that button – because a user has clicked the button – the procedure will be executed with the mouse coordinates being passed as arguments for `x` and `y`.

The main limitation of Eiffel agents, which distinguishes them from closures in other languages, is that they cannot reference local variables from the enclosing scope. This design decision helps in avoiding ambiguity when talking about a local variable value in a closure - should it be the latest value of the variable or the value captured when the agent is created? Only `Current` (a reference to current object, analogous to `this` in Java), its features, and arguments of the agent itself can be accessed from within the agent body. The values of the outer local variables can be passed by providing additional closed operands to the agent.

C++Builder __closure reserved word

Embarcadero C++Builder provides the reserve word __closure to provide a pointer to a method with a similar syntax to a function pointer.[21]

In standard C you could write a `typedef` for a pointer to a function type using the following syntax:

```
typedef void (*TMyFunctionPointer)( void );
```

In a similar way you can declare a `typedef` for a pointer to a method using the following syntax:

```
typedef void (__closure *TMyMethodPointer)();
```

See also

- Anonymous function
- Blocks (C language extension)
- Command pattern
- Continuation
- Currying
- Funarg problem
- Lambda calculus
- Lazy evaluation
- Partial application
- Spaghetti stack
- Syntactic closure
- Value-level programming

Notes

a. The function may be stored as a reference to a function, such as a function pointer.
b. These names most frequently refer to values, mutable variables, or functions, but can also be other entities such as constants, types, classes, or labels.

References

1. Sussman and Steele. "Scheme: An interpreter for extended lambda calculus". "... a data

structure containing a lambda expression, and an environment to be used when that lambda expression is applied to arguments." (Wikisource)

2. David A. Turner (2012). "Some History of Functional Programming Languages" (http://www.cs.kent.ac.uk/people/staff/dat/tfp12/tfp12.pdf). Trends in Functional Programming '12. Section 2, note 8 contains the claim about M-expressions.
3. P. J. Landin (1964), *The mechanical evaluation of expressions*
4. Joel Moses (June 1970), *The Function of FUNCTION in LISP, or Why the FUNARG Problem Should Be Called the Environment Problem*, hdl:1721.1/5854 (https://hdl.handle.net/1721.1%2F5854), AI Memo 199, "A useful metaphor for the difference between FUNCTION and QUOTE in LISP is to think of QUOTE as a porous or an open covering of the function since free variables escape to the current environment. FUNCTION acts as a closed or nonporous covering (hence the term "closure" used by Landin). Thus we talk of "open" Lambda expressions (functions in LISP are usually Lambda expressions) and "closed" Lambda expressions. [...] My interest in the environment problem began while Landin, who had a deep understanding of the problem, visited MIT during 1966–67. I then realized the correspondence between the FUNARG lists which are the results of the evaluation of "closed" Lambda expressions in LISP and ISWIM's Lambda Closures."
5. Åke Wikström (1987). *Functional Programming using Standard ML*. ISBN 0-13-331968-7. "The reason it is called a "closure" is that an expression containing free variables is called an "open" expression, and by associating to it the bindings of its free variables, you close it."
6. Gerald Jay Sussman and Guy L. Steele, Jr. (December 1975), *Scheme: An Interpreter for the Extended Lambda Calculus*, AI Memo 349
7. Abelson, Harold; Sussman, Gerald Jay; Sussman, Julie (1996). *Structure and Interpretation of Computer Programs* (https://mitpress.mit.edu/sites/default/files/sicp/full-text/book/book.html). Cambridge, MA: MIT Press. p. 98–99. ISBN 0262510871.
8. "array.filter" (https://developer.mozilla.org/en/Core_JavaScript_1.5_Reference/Global_Objects/Array/filter). *Mozilla Developer Center*. 10 January 2010. Retrieved 9 February 2010.
9. "Re: FP, OO and relations. Does anyone trump the others?" (https://web.archive.org/web/20081226055307/http://okmij.org/ftp/Scheme/oop-in-fp.txt). 29 December 1999. Archived from the original (http://okmij.org/ftp/Scheme/oop-in-fp.txt) on 26 December 2008. Retrieved 23 December 2008.
10. *Lambda Expressions and Closures (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2550.pdf)* C++ Standards Committee. 29 February 2008.
11. GCC Manual, 6.4 Nested Functions (https://gcc.gnu.org/onlinedocs/gcc/Nested-Functions.html), "If you try to call the nested function through its address after the containing function exits, all hell breaks loose. If you try to call it after a containing scope level exits, and if it refers to some of the variables that are no longer in scope, you may be lucky, but it's not wise to take the risk. If, however, the nested function does not refer to anything that has gone out of scope, you should be safe."
12. *Foundations of Actor Semantics (https://dspace.mit.edu/handle/1721.1/6935)* Will Clinger. MIT Mathematics Doctoral Dissertation. June 1981.
13. "Function.prototype.bind()" (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_objects/Function/bind). *MDN Web Docs*. Retrieved 20 November 2018.
14. "Closures" (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures#Creating_closures_in_loops_A_common_mistake). *MDN Web Docs*. Retrieved 20 November 2018.
15. "Lambda Expressions (The Java Tutorials)" (http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html).
16. "Nested, Inner, Member, and Top-Level Classes" (https://blogs.oracle.com/darcy/entry/nested_inner_member_and_top).
17. "Inner Class Example (The Java Tutorials > Learning the Java Language > Classes and Objects)" (http://java.sun.com/docs/books/tutorial/java/javaOO/innerclasses.html).

18. "Nested Classes (The Java Tutorials > Learning the Java Language > Classes and Objects)" (http://java.sun.com/docs/books/tutorial/java/javaOO/nested.html).
19. Apple Inc. "Blocks Programming Topics" (https://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/Blocks/Articles/00_Introduction.html). Retrieved 8 March 2011.
20. Joachim Bengtsson (7 July 2010). "Programming with C Blocks on Apple Devices" (https://web.archive.org/web/20101025034928/http://thirdcog.eu/pwcblocks/). Archived from the original (http://thirdcog.eu/pwcblocks/) on 25 October 2010. Retrieved 18 September 2010.
21. Full documentation can be found at http://docwiki.embarcadero.com/RADStudio/Rio/en/Closure

External links

- Original "Lambda Papers" (https://web.archive.org/web/20160510140804/http://library.readscheme.org/page1.html): A classic series of papers by Guy Steele and Gerald Sussman discussing, among other things, the versatility of closures in the context of Scheme (where they appear as *lambda expressions*).
- Neal Gafter (28 January 2007). "A Definition of Closures" (http://gafter.blogspot.com/2007/01/definition-of-closures.html).
- Gilad Bracha, Neal Gafter, James Gosling, Peter von der Ahé. "Closures for the Java Programming Language (v0.5)" (http://www.javac.info/closures-v05.html).
- Closures (http://martinfowler.com/bliki/Closure.html): An article about closures in dynamically typed imperative languages, by Martin Fowler.
- Collection closure methods (http://martinfowler.com/bliki/CollectionClosureMethod.html): An example of a technical domain where using closures is convenient, by Martin Fowler.