Function object

From Wikipedia, the free encyclopedia
Jump to: navigation, search
This article is about the computer programming concept. For functors in category theory, see Functor.

A **function object**[a] is a computer programming construct allowing an object to be invoked or called as if it were an ordinary function, usually with the same syntax (a function parameter that can also be a function).

Description

A typical use of a function object is in writing callback functions. A callback in procedural languages, such as C, may be performed by using function pointers.[2] However it can be difficult or awkward to pass a state into or out of the callback function. This restriction also inhibits more dynamic behavior of the function. A function object solves those problems since the function is really a façade for a full object, carrying its own state.

Many modern (and some older) languages, e.g. C++, Eiffel, Groovy, Lisp, Smalltalk, Perl, PHP, Python, Ruby, Scala, and many others, support first-class function objects and may even make significant use of them.[3] Functional programming languages additionally support closures, i.e. first-class functions that can 'close over' variables in their surrounding environment at creation time. During compilation, a transformation known as lambda lifting converts the closures into function objects.

In C and C++

Consider the example of a sorting routine that uses a callback function to define an ordering relation between a pair of items. A C program using function pointers may appear as:

```
#include <stdlib.h>

/* qsort() callback function, returns < 0 if a < b, > 0 if a > b, 0 if a == b */
int compareInts(const void* a, const void* b)
{
    return (*(int *)a > *(int *)b) - (*(int *)a < *(int *)b);
}
...
// prototype of qsort is
// void qsort(void *base, size_t nel, size_t width, int (*compar)(const void *, const void *));
...
int main(void)
{
    int items[] = { 4, 3, 1, 2 };
    qsort(items, sizeof(items) / sizeof(items[0]), sizeof(items[0]), compareInts);
    return 0;
}
```

In C++, a function object may be used instead of an ordinary function by defining a class that overloads the function call operator by defining an `operator()` member function. In C++, this may appear as follows:

```
// comparator predicate: returns true if a < b, false otherwise
struct IntComparator
{
  bool operator()(const int &a, const int &b) const
  {
    return a < b;
  }
};

int main()
{
    std::vector<int> items { 4, 3, 1, 2 };
    std::sort(items.begin(), items.end(), IntComparator());
    return 0;
}
```

Notice that the syntax for providing the callback to the `std::sort()` function is identical, but an object is passed instead of a function pointer. When invoked, the callback function is executed just as any other member function, and therefore has full access to the other members (data or functions) of the object. Of course, this is just a trivial example. To understand what power a functor provides more than a regular function, consider the common use case of sorting objects by a particular field. In the following example, a functor is used to sort a simple employee database by each employee's ID number.

```
struct CompareBy
{
    const std::string SORT_FIELD;
    CompareBy(const std::string& sort_field="name")
      : SORT_FIELD(sort_field)
    {
        /* validate sort_field */
    }

    bool operator()(const Employee& a, const Employee& b)
    {
        if (SORT_FIELD == "name")
            return a.name < b.name;
        else if (SORT_FIELD == "age")
            return a.age < b.age;
        else if (SORT_FIELD == "idnum")
            return a.idnum < b.idnum;
        else
```

```
            /* throw exception or something */
    }
};

int main()
{
    std::vector<Employee> emps;

    /* code to populate database */

    // Sort the database by employee ID number
    std::sort(emps.begin(), emps.end(), CompareBy("idnum"));

    return 0;
}
```

In C++11, the lambda expression provides a more succinct way to do the same thing.

```
int main()
{
    std::vector<Employee> emps;
    /* code to populate database */
    const std::string sort_field = "idnum";
    std::sort(emps.begin(), emps.end(), [&sort_field](const Employee& a, const Employee& b){ /* code to select and compare field */ });
    return 0;
}
```

It is possible to use function objects in situations other than as callback functions. In this case, the shortened term *functor* is normally *not* used about the function object. Continuing the example,

```
  IntComparator cpm;
  bool result = cpm(a, b);
```

In addition to class type functors, other kinds of function objects are also possible in C++. They can take advantage of C++'s member-pointer or template facilities. The expressiveness of templates allows some functional programming techniques to be used, such as defining function objects in terms of other function objects (like function composition). Much of the C++ Standard Template Library (STL) makes heavy use of template-based function objects.

Maintaining state

Another advantage of function objects is their ability to maintain a state that affects `operator()` between calls. For example, the following code defines a generator counting from 10 upwards and is invoked 11 times.

```
#include <iostream>
#include <iterator>
#include <algorithm>

class CountFrom {
private:
  int &count;
public:
  CountFrom(int &n) : count(n) {}
  int operator()() { return count++; }
};

int main()
{
  int state(10);
  std::generate_n(std::ostream_iterator<int>(std::cout, "\n"), 11, CountFrom(state));
  return 0;
}
```

In C#

In C#, function objects are declared via delegates. A delegate can be declared using a named method or a lambda expression. Here is an example using a named method.

```
using System;
using System.Collections.Generic;

public class ComparisonClass1 {
    public static int CompareFunction(int x, int y) {
        return x - y;
    }

    public static void Main() {
        List<int> items = new List<int> { 4, 3, 1, 2 };

        Comparison<int> del = CompareFunction;

        items.Sort(del);
    }
}
```

Here is an example using a lambda expression.

```
using System;
using System.Collections.Generic;

public class ComparisonClass2 {
    public static void Main() {
        List<int> items = new List<int> { 4, 3, 1, 2 };
        items.Sort((x, y) => x - y);
```

```
    }
}
```

In D

D provides several ways to declare function objects: Lisp/Python-style via closures or C#-style via delegates, respectively:

```
bool find(T)(T[] haystack, bool delegate(T) needle_test) {
  foreach (straw; haystack) {
    if (needle_test(straw))
      return true;
  }
  return false;
}

void main() {
    int[] haystack = [345, 15, 457, 9, 56, 123, 456];
    int   needle = 123;
    bool needleTest(int n) {
      return n == needle;
    }
    assert(
      find(haystack, &needleTest)
    );
}
```

The difference between a delegate and a closure in D is automatically and conservatively determined by the compiler. D also supports function literals, that allow a lambda-style definition:

```
void main() {
    int[] haystack = [345, 15, 457, 9, 56, 123, 456];
    int   needle = 123;
    assert(
       find(haystack, (int n) { return n == needle; })
   );
}
```

To allow the compiler to inline the code (see above), function objects can also be specified C++-style via operator overloading:

```
bool find(T, F)(T[] haystack, F needle_test) {
  foreach (straw; haystack) {
    if (needle_test(straw))
      return true;
  }
  return false;
}

void main() {
    int[] haystack = [345, 15, 457, 9, 56, 123, 456];
    int   needle = 123;
    class NeedleTest {
      int needle;
      this(int n) { needle = n; }
      bool opCall(int n) {
        return n == needle;
      }
    }
    assert(
      find(haystack, new NeedleTest(needle))
   );
}
```

In Eiffel

In the Eiffel software development method and language, operations and objects are seen always as separate concepts. However, the agent mechanism facilitates the modeling of operations as runtime objects. Agents satisfy the range of application attributed to function objects, such as being passed as arguments in procedural calls or specified as callback routines. The design of the agent mechanism in Eiffel attempts to reflect the object-oriented nature of the method and language. An agent is an object that generally is a direct instance of one of the two library classes, which model the two types of routines in Eiffel: `PROCEDURE` and `FUNCTION`. These two classes descend from the more abstract `ROUTINE`.

Within software text, the language keyword `agent` allows agents to be constructed in a compact form. In the following example, the goal is to add the action of stepping the gauge forward to the list of actions to be executed in the event that a button is clicked.

```
            my_button.select_actions.extend (agent my_gauge.step_forward)
```

The routine `extend` referenced in the example above is a feature of a class in a graphical user interface (GUI) library to provide event-driven programming capabilities.

In other library classes, agents are seen to be used for different purposes. In a library supporting data structures, for example, a class modeling linear structures effects universal quantification with a function `for_all` of type `BOOLEAN` that accepts an agent, an instance of `FUNCTION`, as an argument. So, in the following example, `my_action` is executed only if all members of `my_list` contain the character '!':

```
    my_list: LINKED_LIST [STRING]
        ...
            if my_list.for_all (agent {STRING}.has ('!')) then
                my_action
            end
        ...
```

When agents are created, the arguments to the routines they model and even the target object to which they are applied can be either *closed* or left *open*. Closed arguments and targets are given values at agent creation time. The assignment of values for open arguments and targets is deferred until some point after the agent is created. The routine `for_all` expects as an argument an agent representing a function with one open argument or target that conforms to actual generic parameter for the structure (`STRING` in this example.)

When the target of an agent is left open, the class name of the expected target, enclosed in braces, is substituted for an object reference as shown in the text `agent {STRING}.has ('!')` in the example above. When an argument is left open, the question mark character ('?') is coded as a placeholder for the open argument.

The ability to close or leave open targets and arguments is intended to improve the flexibility of the agent mechanism. Consider a class that contains the following procedure to print a string on standard output after a new line:

```
    print_on_new_line (s: STRING)
            -- Print `s' preceded by a new line
        do
            print ("%N" + s)
        end
```

The following snippet, assumed to be in the same class, uses `print_on_new_line` to demonstrate the mixing of open arguments and open targets in agents used as arguments to the same routine.

```
    my_list: LINKED_LIST [STRING]
        ...
            my_list.do_all (agent print_on_new_line (?))
            my_list.do_all (agent {STRING}.to_lower)
            my_list.do_all (agent print_on_new_line (?))
        ...
```

This example uses the procedure `do_all` for linear structures, which executes the routine modeled by an agent for each item in the structure.

The sequence of three instructions prints the strings in `my_list`, converts the strings to lowercase, and then prints them again.

Procedure `do_all` iterates across the structure executing the routine substituting the current item for either the open argument (in the case of the agents based on `print_on_new_line`), or the open target (in the case of the agent based on `to_lower`).

Open and closed arguments and targets also allow the use of routines that call for more arguments than are required by closing all but the necessary number of arguments:

```
            my_list.do_all (agent my_multi_arg_procedure (closed_arg_1, ?, closed_arg_2, closed_arg_3)
```

The Eiffel agent mechanism is detailed in the Eiffel ISO/ECMA standard document.

In Java

Java has no first-class functions, so function objects are usually expressed by an interface with a single method (most commonly the `Callable` interface), typically with the implementation being an anonymous inner class, or, starting in Java 8, a lambda.

For an example from Java's standard library, `java.util.Collections.sort()` takes a `List` and a functor whose role is to compare objects in the List. Without first-class functions, the function is part of the Comparator interface. This could be used as follows.

```
List<String> list = Arrays.asList("10", "1", "20", "11", "21", "12");

Comparator<String> numStringComparator = new Comparator<String>() {
    public int compare(String str1, String str2) {
        return Integer.valueOf(str1).compareTo(Integer.valueOf(str2));
    }
};

Collections.sort(list, numStringComparator);
```

In Java 8+, this can be written as:

```
List<String> list = Arrays.asList("10", "1", "20", "11", "21", "12");

Comparator<String> numStringComparator = (str1, str2) -> Integer.valueOf(str1).compareTo(Integer.valueOf(str2));

Collections.sort(list, numStringComparator);
```

In JavaScript

In JavaScript, functions are first class objects. JavaScript also supports closures.

Compare the following with the subsequent Python example.

```
function Accumulator(start) {
  var current = start;
  return function (x) {
    return current += x;
  };
}
```

An example of this in use:

```
var a = Accumulator(4);
var x = a(5);   // x has value 9
x = a(2);       // x has value 11
```

```
var b = Accumulator(42);
x = b(7);       // x has value 49 (current = 42 in closure b)
x = a(7);       // x has value 18 (current = 11 in closure a)
```

In Lisp and Scheme

In Lisp family languages such as Common Lisp, Scheme, and others, functions are objects, just like strings, vectors, lists, and numbers. A closure-constructing operator creates a *function object* from a part of the program: the part of code given as an argument to the operator is part of the function, and so is the lexical environment: the bindings of the lexically visible variables are *captured* and stored in the function object, which is more commonly called a closure. The captured bindings play the role of *member variables*, and the code part of the closure plays the role of the *anonymous member function*, just like operator () in C++.

The closure constructor has the syntax `(lambda (parameters ...) code ...)`. The `(parameters ...)` part allows an interface to be declared, so that the function takes the declared parameters. The `code ...` part consists of expressions that are evaluated when the functor is called.

Many uses of functors in languages like C++ are simply emulations of the missing closure constructor. Since the programmer cannot directly construct a closure, they must define a class that has all of the necessary state variables, and also a member function. Then, construct an instance of that class instead, ensuring that all the member variables are initialized through its constructor. The values are derived precisely from those local variables that ought to be captured directly by a closure.

A function-object using the class system, no use of closures:

```
(defclass counter ()
  ((value :initarg :value :accessor value-of)))

(defmethod functor-call ((c counter))
  (incf (value-of c)))

(defun make-counter (initial-value)
  (make-instance 'counter :value initial-value))

;;; use the counter:
(defvar *c* (make-counter 10))
(functor-call *c*) --> 11
(functor-call *c*) --> 12
```

Since there is no standard way to make funcallable objects in Lisp, we fake it by defining a generic function called FUNCTOR-CALL. This can be specialized for any class whatsoever. The standard FUNCALL function is not generic; it only takes function objects.

It is this FUNCTOR-CALL generic function that gives us function objects, which are *a computer programming construct allowing an object to be invoked or called as if it were an ordinary function, usually with the same syntax.* We have *almost* the same syntax: FUNCTOR-CALL instead of FUNCALL. Some Lisps provide *funcallable* objects as a simple extension. Making objects callable using the same syntax as functions is a fairly trivial business. Making a function call operator work with different kinds of *function things*, whether they be class objects or closures is no more complicated than making a + operator that works with different kinds of numbers, such as integers, reals or complex numbers.

Now, a counter implemented using a closure. This is much more brief and direct. The INITIAL-VALUE argument of the MAKE-COUNTER factory function is captured and used directly. It does not have to be copied into some auxiliary class object through a constructor. It *is* the counter. An auxiliary object is created, but that happens *behind the scenes*.

```
(defun make-counter (value)
  (lambda () (incf value)))

;;; use the counter
(defvar *c* (make-counter 10))
(funcall *c*) ; --> 11
(funcall *c*) ; --> 12
```

Scheme makes closures even simpler, and Scheme code tends to use such higher-order programming somewhat more idiomatically.

```
(define (make-counter value)
  (lambda () (set! value (+ value 1)) value))
;;; use the counter
(define c (make-counter 10))
(c) ; --> 11
(c) ; --> 12
```

More than one closure can be created in the same lexical environment. A vector of closures, each implementing a specific kind of operation, can quite faithfully emulate an object that has a set of virtual operations. That type of single dispatch object-oriented programming can be done fully with closures.

Thus there exists a kind of tunnel being dug from both sides of the proverbial mountain. Programmers in OOP languages discover function objects by restricting objects to have one *main* function to *do* that object's functional purpose, and even eliminate its name so that it looks like the object is being called! While programmers who use closures are not surprised that an object is called like a function, they discover that multiple closures sharing the same environment can provide a complete set of abstract operations like a virtual table for single dispatch type OOP.

In Objective-C

In Objective-C, a function object can be created from the `NSInvocation` class. Construction of a function object requires a method signature, the target object, and the target selector. Here is an example for creating an invocation to the current object's `myMethod`:

```
// Construct a function object
SEL sel = @selector(myMethod);
NSInvocation* inv = [NSInvocation invocationWithMethodSignature:
                     [self methodSignatureForSelector:sel]];
[inv setTarget:self];
[inv setSelector:sel];

// Do the actual invocation
[inv invoke];
```

An advantage of `NSInvocation` is that the target object can be modified after creation. A single `NSInvocation` can be created and then called for each of any number of targets, for instance from an observable object. An `NSInvocation` can be created from only a protocol, but it is not straightforward. See here.

In Perl

In Perl, a function object can be created either from a class's constructor returning a function closed over the object's instance data, blessed into the class:

```
package Acc1;
sub new {
    my $class = shift;
    my $arg = shift;
    my $obj = sub {
        my $num = shift;
        $arg += $num;
    };
    bless $obj, $class;
}
1;
```

or by overloading the &{} operator so that the object can be used as a function:

```
package Acc2;
use overload
    '&{}' =>
        sub {
            my $self = shift;
            sub {
                my $num = shift;
                $self->{arg} += $num;
            }
        };

sub new {
    my $class = shift;
    my $arg = shift;
    my $obj = { arg => $arg };
    bless $obj, $class;
}
1;
```

In both cases the function object can be used either using the dereferencing arrow syntax *$ref->(@arguments)*:

```
use Acc1;
my $a = Acc1->new(42);
print $a->(10), "\n";    # prints 52
print $a->(8), "\n";     # prints 60
```

or using the coderef dereferencing syntax *&$ref(@arguments)*:

```
use Acc2;
my $a = Acc2->new(12);
print &$a(10), "\n";     # prints 22
print &$a(8), "\n";      # prints 30
```

In PHP

PHP 5.3+ has first-class functions that can be used e.g. as parameter to the usort() function:

```
$a = array(3, 1, 4);
usort($a, function ($x, $y) { return $x - $y; });
```

It is also possible in PHP 5.3+ to make objects invokable by adding a magic __invoke() method to their class:[4]

```
class Minus {
    public function __invoke($x, $y) {
        return $x - $y;
    }
}

$a = array(3, 1, 4);
usort($a, new Minus());
```

In PowerShell

In the Windows PowerShell language, a script block is a collection of statements or expressions that can be used as a single unit. A script block can accept arguments and return values. A script block is an instance of a Microsoft .NET Framework type System.Management.Automation.ScriptBlock.

```
Function Get-Accumulator($x) {
    {
```

```
        param($y)
        return $script:x += $y
    }.GetNewClosure()
}

PS C:\> $a = Get-Accumulator 4
PS C:\> & $a 5
9
PS C:\> & $a 2
11
PS C:\> $b = Get-Accumulator 32
PS C:\> & $b 10
42
```

In Python

In Python, functions are first-class objects, just like strings, numbers, lists etc. This feature eliminates the need to write a function object in many cases. Any object with a `__call__()` method can be called using function-call syntax.

An example is this accumulator class (based on Paul Graham's study on programming language syntax and clarity):[5]

```
class Accumulator(object):
    def __init__(self, n):
        self.n = n

    def __call__(self, x):
        self.n += x
        return self.n
```

An example of this in use (using the interactive interpreter):

```
>>> a = Accumulator(4)
>>> a(5)
9
>>> a(2)
11
>>> b = Accumulator(42)
>>> b(7)
49
```

Since functions are objects, they can also be defined locally, given attributes, and returned by other functions ,[6] as demonstrated in the following two examples:

```
def Accumulator(n):
    def inc(x):
        inc.n += x
        return inc.n
    inc.n = n
    return inc
```

Function object creation using a closure referencing a non-local variable in Python 3:

```
def Accumulator(n):
    def inc(x):
        nonlocal n
        n += x
        return n
    return inc
```

In Ruby

In Ruby, several objects can be considered function objects, in particular Method and Proc objects. Ruby also has two kinds of objects that can be thought of as semi-function objects: UnboundMethod and block. UnboundMethods must first be bound to an object (thus becoming a Method) before they can be used as a function object. Blocks can be called like function objects, but to be used in any other capacity as an object (e.g. passed as an argument) they must first be converted to a Proc. More recently, symbols (accessed via the literal unary indicator `:`) can also be converted to `Proc`s. Using Ruby's unary `&` operator—equivalent to calling `to_proc` on an object, and assuming that method exists—the Ruby Extensions Project created a simple hack.

```
class Symbol
  def to_proc
    proc { |obj, *args| obj.send(self, *args) }
  end
end
```

Now, method `foo` can be a function object, i.e. a `Proc`, via `&:foo` and used via `takes_a_functor(&:foo)`. `Symbol.to_proc` was officially added to Ruby on June 11, 2006 during RubyKaiga2006. [1]

Because of the variety of forms, the term Functor is not generally used in Ruby to mean a Function object. Just a type of dispatch delegation introduced by the Ruby Facets project is named as Functor. The most basic definition of which is:

```
class Functor
  def initialize(&func)
    @func = func
  end
  def method_missing(op, *args, &blk)
    @func.call(op, *args, &blk)
  end
end
```

This usage is more akin to that used by functional programming languages, like ML, and the original mathematical terminology.

Other meanings

In a more theoretical context a *function object* may be considered to be any instance of the class of functions, especially in languages such as Common Lisp in which functions are first-class objects.

The functional programming languages ML and Haskell use the term *functor* to represent a mapping from modules to modules, or from types to types and is a technique for reusing code. Functors used in this manner are analogous to the original mathematical meaning of functor in category theory, or to the use of generic programming in C++, Java or Ada.

In Prolog and related languages, functor is a synonym for function symbol.

See also

- Callback (computer science)
- Closure (computer science)
- Function pointer
- Higher-order function
- Command pattern
- Currying

Notes

1. ^ In C++, a **functionoid** is an object that has one major method, and a **functor** is a special case of a functionoid.[1] They are similar to a function object, *but not the same*.

References

1. ^ What's the difference between a functionoid and a functor?
2. ^ *Silan Liu. "C++ Tutorial Part I - Basic: 5.10 Function pointers are mainly used to achieve call back technique, which will be discussed right after.". TRIPOD: Programming Tutorials Copyright © Silan Liu 2002. Retrieved 2012-09-07. "Function pointers are mainly used to achieve call back technique, which will be discussed right after."*
3. ^ *Paweł Turlejski (2009-10-02). "C++ Tutorial Part I - Basic: 5.10 Function pointers are mainly used to achieve call back technique, which will be discussed right after.". Just a Few Lines. Retrieved 2012-09-07. "PHP 5.3, along with many other features, introduced closures. So now we can finally do all the cool stuff that Ruby / Groovy / Scala / any_modern_language guys can do, right? Well, we can, but we probably won't... Here's why."*
4. ^ PHP Documentation on Magic Methods
5. ^ Accumulator Generator
6. ^ Python reference manual - Function definitions

Further reading

- David Vandevoorde & Nicolai M Josuttis (2006). *C++ Templates: The Complete Guide*, ISBN 0-201-73484-2: Specifically, chapter 22 is devoted to function objects.

External links

- Description from the Portland Pattern Repository
- C++ Advanced Design Issues - Asynchronous C++ by Kevlin Henney
- The Function Pointer Tutorials by Lars Haendel (2000/2001)
- Article "Generalized Function Pointers" by Herb Sutter
- Generic Algorithms for Java
- PHP Functors - Function Objects in PHP
- What the heck is a functionoid, and why would I use one? (C++ FAQ)

Categories:

- Object (computer science)
- Subroutines
- Articles with example Ruby code

Hidden categories:

- Articles needing additional references from February 2009
- All articles needing additional references
- Articles with example C code
- Articles with example C++ code
- Articles with example Java code
- Articles with example Perl code
- Articles with example Python code
- Pages using ISBN magic links

Navigation menu

Personal tools