WIKIPEDIA

# Function pointer

A **function pointer**, also called a **subroutine pointer** or **procedure pointer**, is a pointer that points to a function. As opposed to referencing a data value, a function pointer points to executable code within memory. Dereferencing the function pointer yields the referenced function, which can be invoked and passed arguments just as in a normal function call. Such an invocation is also known as an "indirect" call, because the function is being invoked *indirectly* through a variable instead of *directly* through a fixed identifier or address.

Function pointers can be used to simplify code by providing a simple way to select a function to execute based on run-time values.

Function pointers are supported by third-generation programming languages (such as PL/I, COBOL, Fortran,[1] dBASE dBL, and C) and object-oriented programming languages (such as C++ and D).[2]

## Contents

## Simple function pointers

The simplest implementation of a function (or subroutine) pointer is as a variable containing the address of the function within executable memory. Older third-generation languages such as PL/I and COBOL, as well as more modern languages such as Pascal and C generally implement function pointers in this manner.[3]

### Example in C

The following C program illustrates the use of two function pointers:

- *func1* takes one double-precision (double) parameter and returns another double, and is assigned to a function which converts centimetres to inches.
- *func2* takes a pointer to a constant character array as well as an integer and returns a pointer to a character, and is assigned to a C string handling function which returns a pointer to the first

occurrence of a given character in a character array.

```
#include <stdio.h>  /* for printf */
#include <string.h> /* for strchr */

double cm_to_inches(double cm) {
    return cm / 2.54;
}

// "strchr" is part of the C string handling (i.e., no need for declaration)
// See https://en.wikipedia.org/wiki/C_string_handling#Functions

int main(void) {
    double (*func1)(double) = cm_to_inches;
    char * (*func2)(const char *, int) = strchr;
    printf("%f %s", func1(15.0), func2("Wikipedia", 'p'));
    /* prints "5.905512 pedia" */
    return 0;
}
```

The next program uses a function pointer to invoke one of two functions (`sin` or `cos`) indirectly from another function (`compute_sum`, computing an approximation of the function's Riemann integration). The program operates by having function `main` call function `compute_sum` twice, passing it a pointer to the library function `sin` the first time, and a pointer to function `cos` the second time. Function `compute_sum` in turn invokes one of the two functions indirectly by dereferencing its function pointer argument `funcp` multiple times, adding together the values that the invoked function returns and returning the resulting sum. The two sums are written to the standard output by `main`.

```
#include <math.h>
#include <stdio.h>

// Function taking a function pointer as an argument
double compute_sum(double (*funcp)(double), double lo, double hi) {
    double sum = 0.0;

    // Add values returned by the pointed-to function '*funcp'
    int i;
    for (i = 0; i <= 100; i++) {
        // Use the function pointer 'funcp' to invoke the function
        double x = i / 100.0 * (hi - lo) + lo;
        double y = funcp(x);
        sum += y;
    }
    return sum / 101.0 * (hi - lo);
}

double square(double x) {
     return x * x;
}

int main(void) {
    double  sum;

    // Use standard library function 'sin()' as the pointed-to function
    sum = compute_sum(sin, 0.0, 1.0);
    printf("sum(sin): %g\n", sum);

    // Use standard library function 'cos()' as the pointed-to function
    sum = compute_sum(cos, 0.0, 1.0);
    printf("sum(cos): %g\n", sum);

    // Use user-defined function 'square()' as the pointed-to function
    sum = compute_sum(square, 0.0, 1.0);
    printf("sum(square): %g\n", sum);

    return 0;
}
```

## Functors

Functors, or function objects, are similar to function pointers, and can be used in similar ways. A functor is an object of a class type that implements the function-call operator, allowing the object to be used within expressions using the same syntax as a function call. Functors are more powerful than simple function pointers, being able to contain their own data values, and allowing the programmer to emulate closures. They are also used as callback functions if it is necessary to use a member function as a callback function.[4]

Many "pure" object-oriented languages do not support function pointers. Something similar can be implemented in these kinds of languages, though, using references to interfaces that define a single method (member function). CLI languages such as C# and Visual Basic .NET implement type-safe function pointers with delegates.

In other languages that support first-class functions, functions are regarded as data, and can be passed, returned, and created dynamically directly by other functions, eliminating the need for function pointers.

Extensively using function pointers to call functions may produce a slow-down for the code on modern processors, because branch predictor may not be able to figure out where to branch to (it depends on the value of the function pointer at run time) although this effect can be overstated as it is often amply compensated for by significantly reduced non-indexed table lookups.

## Method pointers

C++ includes support for object-oriented programming, so classes can have methods (usually referred to as member functions). Non-static member functions (instance methods) have an implicit parameter (the *this* pointer) which is the pointer to the object it is operating on, so the type of the object must be included as part of the type of the function pointer. The method is then used on an object of that class by using one of the "pointer-to-member" operators: `.*` or `->*` (for an object or a pointer to object, respectively).

Although function pointers in C and C++ can be implemented as simple addresses, so that typically `sizeof(Fx)==sizeof(void *)`, member pointers in C++ are sometimes implemented as "fat pointers", typically two or three times the size of a simple function pointer, in order to deal with virtual methods and virtual inheritance.

## In C++

In C++, in addition to the method used in C, it is also possible to use the C++ standard library class template `std::function`, of which the instances are function objects:

```
#include <iostream>
#include <functional>

static double derivative(const std::function<double(double)> &f, double x0, double eps) {
    double eps2 = eps / 2;
    double lo = x0 - eps2;
    double hi = x0 + eps2;
    return (f(hi) - f(lo)) / eps;
}

static double f(double x) {
    return x * x;
}

int main() {
    double x = 1;
    std::cout << "d/dx(x ^ 2) [@ x = " << x << "] = " << derivative(f, x, 1e-5) << std::endl;
    return 0;
}
```

### Pointers to member functions in C++

This is how C++ uses function pointers when dealing with member functions of classes or structs. These are invoked using an object pointer or a this call. They are type safe in that you can only call members of that class (or derivatives) using a pointer of that type. This example also demonstrates the use of a typedef for the pointer to member function added for simplicity. Function pointers to static member functions are done in the traditional 'C' style because there is no object pointer for this call required.

```
#include <iostream>
using namespace std;

class Foo {

public:
    int add(int i, int j) {
        return i+j;
    }
    int mult(int i, int j) {
        return i*j;
    }
    static int negate(int i) {
        return -i;
    }
};

int bar1(int i, int j, Foo* pFoo, int(Foo::*pfn)(int,int)) {
    return (pFoo->*pfn)(i,j);
}

typedef int(Foo::*Foo_pfn)(int,int);

int bar2(int i, int j, Foo* pFoo, Foo_pfn pfn) {
    return (pFoo->*pfn)(i,j);
}

typedef int(*PFN)(int);

int bar3(int i, PFN pfn) {
    return pfn(i);
}

int main() {
    Foo foo;
    cout << "Foo::add(2,4) = " << bar1(2,4, &foo, &Foo::add) << endl;
    cout << "Foo::mult(3,5) = " << bar2(3,5, &foo, &Foo::mult) << endl;
    cout << "Foo::negate(6) = " << bar3(6, &Foo::negate) << endl;
    return 0;
}
```

## Alternate C and C++ Syntax

The C and C++ syntax given above is the canonical one used in all the textbooks - but it's difficult to read and explain. Even the above `typedef` examples use this syntax. However, every C and C++ compiler supports a more clear and concise mechanism to declare function pointers: use `typedef`, but *don't* store the pointer as part of the definition. Note that the only way this kind of `typedef` can actually be used is with a pointer - but that highlights the pointer-ness of it.

### C and C++

```
// This declares 'F', a function that accepts a 'char' and returns an 'int'. Definition is
elsewhere.
int F(char c);

// This defines 'Fn', a type of function that accepts a 'char' and returns an 'int'.
typedef int Fn(char c);
```

```
// This defines 'fn', a variable of type pointer-to-'Fn', and assigns the address of 'F' to it.
Fn *fn = &F;      // Note '&' not required - but it highlights what is being done.

// This calls 'F' using 'fn', assigning the result to the variable 'a'
int a = fn('A');

// This defines 'Call', a function that accepts a pointer-to-'Fn', calls it, and returns the
result
int Call(Fn *fn, char c) {
   return fn(c);
} // Call(fn, c)

// This calls function 'Call', passing in 'F' and assigning the result to 'call'
int call = Call(&F, 'A');   // Again, '&' is not required

// LEGACY: Note that to maintain existing code bases, the above definition style can still be
used first;
// then the original type can be defined in terms of it using the new style.

// This defines 'PFn', a type of pointer-to-type-Fn.
typedef Fn *PFn;

// 'PFn' can be used wherever 'Fn *' can
PFn pfn = F;
int CallP(PFn fn, char c);
```

## C++

These examples use the above definitions. In particular, note that the above definition for `Fn` can be used in pointer-to-member-function definitions:

```
// This defines 'C', a class with similar static and member functions,
// and then creates an instance called 'c'
class C {
public:
static int Static(char c);
int Member(char c);
} c; // C

// This defines 'p', a pointer to 'C' and assigns the address of 'c' to it
C *p = &c;

// This assigns a pointer-to-'Static' to 'fn'.
// Since there is no 'this', 'Fn' is the correct type; and 'fn' can be used as above.
fn = &C::Static;

// This defines 'm', a pointer-to-member-of-'C' with type 'Fn',
// and assigns the address of 'C::Member' to it.
// You can read it right-to-left like all pointers:
// "'m' is a pointer to member of class 'C' of type 'Fn'"
Fn C::*m = &C::Member;

// This uses 'm' to call 'Member' in 'c', assigning the result to 'cA'
int cA = (c.*m)('A');

// This uses 'm' to call 'Member' in 'p', assigning the result to 'pA'
int pA = (p->*m)('A');

// This defines 'Ref', a function that accepts a reference-to-'C',
// a pointer-to-member-of-'C' of type 'Fn', and a 'char',
// calls the function and returns the result
int Ref(C &r, Fn C::*m, char c) {
   return (r.*m)(c);
} // Ref(r, m, c)

// This defines 'Ptr', a function that accepts a pointer-to-'C',
// a pointer-to-member-of-'C' of type 'Fn', and a 'char',
// calls the function and returns the result
int Ptr(C *p, Fn C::*m, char c) {
   return (p->*m)(c);
} // Ptr(p, m, c)

// LEGACY: Note that to maintain existing code bases, the above definition style can still be
used first;
// then the original type can be defined in terms of it using the new style.
// This defines 'FnC', a type of pointer-to-member-of-class-'C' of type 'Fn'
typedef Fn C::*FnC;

// 'FnC' can be used wherever 'Fn C::*' can
FnC fnC = &C::Member;
int RefP(C &p, FnC m, char c);
```

## See also

- Delegation (computing)
- Function object
- Higher-order function
- Procedural parameter

## References

1. Andrew J. Miller. "Fortran Examples" (http://www.esm.psu.edu/~ajm138/fortranexamples.html#ex1). http://www.esm.psu.edu/~ajm138/fortranexamples.html. Retrieved 2013-09-14.
2. "The Function Pointer Tutorials" (http://www.newty.de/fpt/intro.html#what). http://www.newty.de/: logo. Retrieved 2011-04-13. "Function Pointers are pointers, i.e. variables, which point to the address of a function"
3. "The Function Pointer Tutorials" (http://www.newty.de/fpt/intro.html#top). http://www.newty.de/: logo. Retrieved 2011-04-13. "Important note: A function pointer always points to a function with a specific signature! Thus all functions, you want to use with the same function pointer, must have the same parameters and return-type!"
4. "Expertise: Intermediate Language: C++: Use Functor for Callbacks in C++" (http://www.devx.com/tips/Tip/27126). http://www.devx.com/: DevX.com. 2005-01-31. Retrieved 2011-04-13. "If you want to use a member function as a callback function, then the member function needs to be associated with an object of the class before it can be called. In this case, you can use functor [with an example on this page]."

## External links

- FAQ on Function Pointers (https://web.archive.org/web/20041013202445/http://www.parashift.com/c++-faq-lite/pointers-to-members.html#faq-33.12), things to avoid with function pointers, some information on using function objects
- Function Pointer Tutorials (http://www.newty.de/fpt/), a guide to C/C++ function pointers, callbacks, and function objects (functors)
- Member Function Pointers and the Fastest Possible C++ Delegates (http://www.codeproject.com/KB/cpp/FastDelegate.aspx), CodeProject article by Don Clugston
- Pointer Tutorials (http://www.cplusplus.com/doc/tutorial/pointers.html), C++ documentation and tutorials
- C pointers explained (http://www.onlinecomputerteacher.net/pointers-in-c.html) a visual guide of pointers in C
- Secure Function Pointer and Callbacks in Windows Programming (http://www.codeproject.com/KB/security/Securefunctionpointer.aspx), CodeProject article by R. Selvam
- The C Book (http://publications.gbdirect.co.uk/c_book/chapter5/function_pointers.html), Function Pointers in C by "The C Book"
- Function Pointers in dBASE dBL (http://www.dbase.com/help/2_80/Language_Definition/IDH_LDEF_FUNCPOINTERS.htm), Function Pointer in dBASE dBL