# Function Pointer in C

In C, like normal data pointers (int \*, char \*, etc), we can have pointers to functions. Following is a simple example that shows declaration and function call using function pointer.

#include <stdio.h>

// A normal function with an int parameter

// and void return type

void fun(int a)

{

    printf("Value of a is %d\n", a);

}

int main()

{

    // fun\_ptr is a pointer to function fun()

    void (\*fun\_ptr)(int) = &fun;

    /\* The above line is equivalent of following two

       void (\*fun\_ptr)(int);

       fun\_ptr = &fun;

    \*/

    // Invoking fun() using fun\_ptr

    (\*fun\_ptr)(10);

    return 0;

}

Output:

Value of a is 10

**Why do we need an extra bracket around function pointers like fun\_ptr in above example?**  
If we remove bracket, then the expression “void (\*fun\_ptr)(int)” becomes “void \*fun\_ptr(int)” which is declaration of a function that returns void pointer.

**Following are some interesting facts about function pointers:**

1. Unlike normal pointers, a function pointer points to code, not data. Typically a function pointer stores the start of executable code.
2. Unlike normal pointers, we *do not allocate de-allocate memory* using function pointers.
3. A function’s name can also be used to get functions’ address. For example, in the below program, we have removed address operator ‘&’ in assignment. We have also changed function call by removing \*, the program still works.

|  |
| --- |
| #include <stdio.h>  // A normal function with an int parameter  // and void return type  void fun(int a)  {      printf("Value of a is %d\n", a);  }    int main()  {      void (\*fun\_ptr)(int) = **fun**;  // & removed    **fun\_ptr(10);**  // \* removed        return 0;  } |

Output:

Value of a is 10

1. Like normal pointers, we can have **an array of function pointers**. Below example in point 5 shows syntax for array of pointers.
2. Function pointer can be **used in place of switch case**. For example, in below program, user is asked for a choice between 0 and 2 to do different tasks.

|  |
| --- |
| #include <stdio.h>  void add(int a, int b)  {      printf("Addition is %d\n", a+b);  }  void subtract(int a, int b)  {      printf("Subtraction is %d\n", a-b);  }  void multiply(int a, int b)  {      printf("Multiplication is %d\n", a\*b);  }    int main()  {      // fun\_ptr\_arr is an array of function pointers  **void (\*fun\_ptr\_arr[])(int, int) = {add, subtract, multiply};**      unsigned int ch, a = 15, b = 10;        printf("Enter Choice: 0 for add, 1 for subtract and 2 "              "for multiply\n");      scanf("%d", &ch);        if (ch > 2) return 0;    **(\*fun\_ptr\_arr[ch])(a, b);**        return 0;  }  Enter Choice: 0 for add, 1 for subtract and 2 for multiply  2  Multiplication is 150 |

1. Like normal data pointers, **a function pointer can be passed as an argument and can also be returned from a function.**  
   For example, consider the following C program where wrapper() receives a void fun() as parameter and calls the passed function.

|  |
| --- |
| // A simple C program to show function pointers as parameter  #include <stdio.h>    // Two simple functions  void fun1() { printf("Fun1\n"); }  void fun2() { printf("Fun2\n"); }    // A function that receives a simple function  // as parameter and calls the function  void wrapper(void (\*fun)())  {      fun();  }    int main()  {      wrapper(fun1);      wrapper(fun2);      return 0;  } |

This point in particular is very useful in C. In C, we can use function pointers to avoid code redundancy. For example a simple [qsort()](http://www.cplusplus.com/reference/cstdlib/qsort/) function can be used to sort arrays in ascending order or descending or by any other order in case of array of structures. Not only this, with function pointers and void pointers, it is possible to use qsort for any data type.

|  |
| --- |
| // An example for qsort and comparator  #include <stdio.h>  #include <stdlib.h>    // A sample comparator function that is used  // for sorting an integer array in ascending order.  // To sort any array for any other data type and/or  // criteria, all we need to do is write more compare  // functions.  And we can use the same qsort()  int compare (const void \* a, const void \* b)  {    return ( \*(int\*)a - \*(int\*)b );  }    int main ()  {    int arr[] = {10, 5, 15, 12, 90, 80};    int n = sizeof(arr)/sizeof(arr[0]), i;      qsort (arr, n, sizeof(int), compare);      for (i=0; i<n; i++)       printf ("%d ", arr[i]);    return 0;  } |

Output:

5 10 12 15 80 90

Similar to qsort(), we can write our own functions that can be used for any data type and can do different tasks without code redundancy.

Below is an example search function that can be used for any data type. In fact we can use this search function to find close elements (below a threshold) by writing a customized compare function.

|  |
| --- |
| #include <stdio.h>  #include <stdbool.h>    // A compare function that is used for searching an integer  // array  bool compare (const void \* a, const void \* b)  {    return ( \*(int\*)a == \*(int\*)b );  }    // General purpose search() function that can be used  // for searching an element \*x in an array arr[] of  // arr\_size. Note that void pointers are used so that  // the function can be called by passing a pointer of  // any type.  ele\_size is size of an array element  int search(void \*arr, int arr\_size, int ele\_size, void \*x,             bool compare (const void \* , const void \*))  {      // Since char takes one byte, we can use char pointer      // for any type/ To get pointer arithmetic correct,      // we need to multiply index with size of an array      // element ele\_size      char \*ptr = (char \*)arr;        int i;      for (i=0; i<arr\_size; i++)          if (compare(ptr + i\*ele\_size, x))             return i;        // If element not found      return -1;  }    int main()  {      int arr[] = {2, 5, 7, 90, 70};      int n = sizeof(arr)/sizeof(arr[0]);      int x = 7;      printf ("Returned index is %d ", search(arr, n,                                 sizeof(int), &x, compare));      return 0;  } |

Output:

Returned index is 2

The above search function can be used for any data type by writing a separate customized compare().

<http://www.cplusplus.com/reference/cstdlib/qsort/>

From the Linux man pages, we have the following declaration for qsort (from stdlib.h):

void qsort(void \*base, size\_t nmemb, size\_t size,

int(\*compar)(const void \*, const void \*));

Note the use of void\*s to allow qsort to operate on any kind of data (in C++, you'd normally use [templates](http://www.cprogramming.com/tutorial/templates.html) for this task, but C++ also allows the use of void\* pointers) because void\* pointers can point to anything. Because we don't know the size of the individual elements in a void\* array, we must give qsort the number of elements, nmemb, of the array to be sorted, base, in addition to the standard requirement of giving the length, size, of the input.   
  
But what we're really interested in is the compar argument to qsort: it's a function pointer that takes two void \*s and returns an int. This allows anyone to specify how to sort the elements of the array base without having to write a specialized sorting algorithm. Note, also, that compar returns an int; the function pointed to should return -1 if the first argument is less than the second, 0 if they are equal, or 1 if the second is less than the first.   
  
For instance, to sort an array of numbers in ascending order, we could write code like this:

#include <stdlib.h>

int int\_sorter( const void \*first\_arg, const void \*second\_arg )

{

int first = \*(int\*)first\_arg;

int second = \*(int\*)second\_arg;

if ( first < second )

{

return -1;

}

else if ( first == second )

{

return 0;

}

else

{

return 1;

}

}

int main()

{

int array[10];

int i;

/\* fill array \*/

for ( i = 0; i < 10; ++i )

{

array[ i ] = 10 - i;

}

qsort( array, 10 , sizeof( int ), int\_sorter );

for ( i = 0; i < 10; ++i )

{

printf ( "%d\n" ,array[ i ] );

}

}

## **Using Polymorphism and Virtual Functions Instead of Function Pointers (C++)**

You can often avoid the need for explicit function pointers by using virtual functions. For instance, you could write a sorting routine that takes a pointer to a class that provides a virtual function called compare:

class Sorter

{

public:

virtual int compare (const void \*first, const void \*second);

};

// cpp\_qsort, a qsort using C++ features like virtual functions

void cpp\_qsort(void \*base, size\_t nmemb, size\_t size, Sorter \*compar);

inside cpp\_qsort, whenever a comparison is needed, compar->compare should be called. For classes that override this virtual function, the sort routine will get the new behavior of that function. For instance:

class AscendSorter : public Sorter

{

virtual int compare (const void\*, const void\*)

{

int first = \*(int\*)first\_arg;

int second = \*(int\*)second\_arg;

if ( first < second )

{

return -1;

}

else if ( first == second )

{

return 0;

}

else

{

return 1;

}

}

};

and then you could pass in a pointer to an instance of the AscendSorter to cpp\_qsort to sort integers in ascending order.

# What is the point of function pointers?

Most examples boil down to **callbacks**: You call a function f() passing the address of another function g(), and f() calls g() for some specific task. If you pass f() the address of h()instead, then f() will call back h() instead.

Basically, this is a way to **parametrize** a function: Some part of its behavior is not hard-coded into f(), but into the callback function. Callers can make f() behave differently by passing different callback functions. A classic is qsort() from the C standard library that takes its sorting criterion as a pointer to a comparison function.

In C++, this is often done using **function objects** (also called functors). These are objects that overload the function call operator, so you can call them as if they were a function.

Example:

class functor {

public:

void operator()(int i) {std::cout << "the answer is: " << i << '\n';}

};

functor f;

f(42);

The idea behind this is that, unlike a function pointer, a function object can carry not only an algorithm, but also data:

class functor {

public:

functor(const std::string& prompt) : prompt\_(prompt) {}

void operator()(int i) {std::cout << prompt\_ << i << '\n';}

private:

std::string prompt\_;

};

functor f("the answer is: ");

f(42);

Another advantage is that it is sometimes easier to inline calls to function objects than calls through function pointers. This is a reason why sorting in C++ is sometimes faster than sorting in C.

**Another Answer:**

I used function pointers recently to create an **abstraction layer**.

I have a program written in pure C that runs on embedded systems. It supports multiple hardware variants. Depending on the hardware I am running on, it needs to call different versions of some functions.

At initialization time, the program figures out what hardware it is running on and populates the function pointers. All of the higher-level routines in the program just call the functions referenced by pointers. I can add support for new hardware variants without touching the higher-level routines.

I used to use switch/case statements to select the proper function versions, but this became impractical as the program grew to support more and more hardware variants. I had to add case statements all over the place.

I also tried intermediate function layers to figure out which function to use, but they didn't help much. I still had to update case statements in multiple places whenever we added a new variant. With the function pointers, I only have to change the initialization function.

***What is a Callback function?***  
In simple terms, a **Callback function** is one that is not called explicitly by the programmer. Instead, there is some mechanism that continually waits for events to occur, and it will call selected functions in response to particular events.  
This mechanism is typically used when a operation (function) can take long time for execution and the caller of the function does not want to wait till the operation is complete, but does wish to be intimated of the outcome of the operation. Typically, Callback functions help implement such an **asynchronous mechanism**, wherein the caller registers to get intimated about the result of the time consuming processing and continuous other operations while at a later point of time, the caller gets informed of the result.

***An practical example:***  
Windows event processing:  
virtually all windows programs set up an event loop, that makes the program respond to particular events (eg button presses, selecting a check box, window getting focus) by calling a function. The handy thing is that the programmer can specify what function gets called when (say) a particular button is pressed, even though it is not possible to specify when the button will be pressed. The function that is called is referred to as a callback.

**An source Code Illustration:**

//warning: Mind compiled code, intended to illustrate the mechanism

#include <map>

typedef void (\*Callback)();

std::map<int, Callback> callback\_map;

void RegisterCallback(int event, Callback function)

{

callback\_map[event] = function;

}

bool finished = false;

int GetNextEvent()

{

static int i = 0;

++i;

if (i == 5) finished = false;

}

void EventProcessor()

{

int event;

while (!finished)

{

event = GetNextEvent();

std::map<int, Callback>::const\_iterator it = callback\_map.find(event);

if (it != callback\_map.end()) // if a callback is registered for event

{

Callback function = \*it;

if (function)

{

(\*function)();

}

else

{

std::cout << "No callback found\n";

}

}

}

}

void Cat()

{

std::cout << "Cat\n";

}

void Dog()

{

std::cout << "Dog\n";

}

void Bird()

{

std::cout << "Bird\n";

}

int main()

{

RegisterCallBack(1, Cat);

RegisterCallback(2, Dog);

RegisterCallback(3, Cat);

RegisterCallback(4, Bird);

RegisterCallback(5, Cat);

EventProcessor();

return 0;

}

The above would output the following:

Cat

Dog

Cat

Bird

Cat

For more details: <http://www.newty.de/fpt/index.html>

# Where Function pointers are stored ?

#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'*

for (int i = 0; i <= 100; i++)

{

double x, y;

*// Use the function pointer 'funcp' to invoke the function*

x = i/100.0 \* (hi - lo) + lo;

y = (\*funcp)(x);

sum += y;

}

return sum;

}

int main(void)

{

double (\*fp)(double); *// Function pointer*

double sum;

*// Use 'sin()' as the pointed-to function*

fp = &sin;

sum = compute\_sum(fp, 0.0, 1.0);

printf("sum(sin): %f**\n**", sum);

*// Use 'cos()' as the pointed-to function*

sum = compute\_sum(&cos, 0.0, 1.0);

printf("sum(cos): %f**\n**", sum);

return 0;

}

It depends on where you declare it. In the code above, it is stored on the stack, just like sum or any other local variables, like if you had a char \* or int \* variable. You could dynamically allocate space for a function pointer, and have it be stored on the heap, or declare it as a global or static local, and have it be stored with the other global and static local variables, but that is less common.  
  
Remember, a function pointer is just a pointer, so all it contains is the address. It doesn't store the function itself (i.e. no code or variables).