How do function pointers in C work?

Asked 14 years, 7 months ago Modified 1 year, 1 month ago Viewed 995k times

1489

I had some experience lately with function pointers in C.

So going on with the tradition of answering your own questions, I decided to make a small summary of the very basics, for those who need a quick dive-in to the subject.

c function-pointers

Share Edit Follow Flag asked May 8, 2009 at 15:49 community wiki Yuval Adam

42 Also: For a bit of an in-depth analysis of C pointers, see blogs.oracle.com/ksplice/entry/the_ksplice_pointer_challenge. Also, Programming from the Ground Up shows how they work on the machine level. Understanding C's "memory model" is very useful for understanding how C pointers work. – Abbafei May 22, 2013 at 11:17

10 Great info. By the title though, I would have expected to really see an explanation of how "function pointers work", not how they are coded :) – Bogdan Alexandru Aug 28, 2014 at 12:39

2 The following answer is shorter and a lot much easier to understand: stackoverflow.com/a/142809/2188550 – user2188550 Jan 8, 2021 at 22:31

12 Answers Sorted by: Highest score (default)

1733

Function pointers in C

Let's start with a basic function which we will be *pointing to*:

```
int addInt(int n, int m) {
    return n+m;
}
```

First thing, let's define a pointer to a function which receives 2 `int` s and returns an `int` :

```
int (*functionPtr)(int,int);
```

Now we can safely point to our function:

```
functionPtr = &addInt;
```

Now that we have a pointer to the function, let's use it:

```
int sum = (*functionPtr)(2, 3); // sum == 5
```

Passing the pointer to another function is basically the same:

```
int add2to3(int (*functionPtr)(int, int)) {
    return (*functionPtr)(2, 3);
}
```

We can use function pointers in return values as well (try to keep up, it gets messy):

```
// this is a function called functionFactory which receives parameter n
// and returns a pointer to another function which receives two ints
// and it returns another int
int (*functionFactory(int n))(int, int) {
    printf("Got parameter %d", n);
    int (*functionPtr)(int,int) = &addInt;
    return functionPtr;
}
```

But it's much nicer to use a `typedef`:

```
typedef int (*myFuncDef)(int, int);
// note that the typedef name is indeed myFuncDef

myFuncDef functionFactory(int n) {
    printf("Got parameter %d", n);
    myFuncDef functionPtr = &addInt;
    return functionPtr;
}
```

Share Edit Follow Flag

edited Dec 29, 2016 at 22:23

Tom Zych
13.4k 9 36 53

answered May 8, 2009 at 15:49

Yuval Adam
162k 94 307 398

26 Thanks for the great info. Could you add some insight on where function pointers are used or happen to be particularly useful? – Rich.Carpenter May 8, 2009 at 15:55

374 "functionPtr = &addInt;" can also be written (and often is) as " functionPtr = addInt;" which is also valid since the standard says that a function name in this context is converted to the address of the function. – hlovdal May 9, 2009 at 14:39

26 hlovdal, in this context it's interesting to explain that this is what enables one to write functionPtr = **********addInt; – Johannes Schaub - litb May 10, 2009 at 17:54

126 @Rich.Carpenter I know this is 4 years too late, but I figure other people might benefit from this: **Function pointers are useful for passing functions as parameters to other functions**. It took a lot of searching for me to find that answer for some odd reason. So basically, it gives C pseudo first-class functionality. – giant91 Oct 13, 2013 at 2:28

30 @Rich.Carpenter: function pointers are nice for runtime CPU detection. Have multiple versions of some functions to take advantage of SSE, popcnt, AVX, etc. At startup, set your function pointers to the best version of each function for the current CPU. In your other code, just call through the

function pointer instead of having conditional branches on the CPU features everywhere. Then you can do complicated logic about deciding that well, even though this CPU supports `pshufb`, it's slow, so the earlier implementation is still faster. x264/x265 use this extensively, and are open source. – Peter Cordes Aug 30, 2015 at 2:22

|

340

Function pointers in C can be used to perform object-oriented programming in C.

For example, the following lines is written in C:

```
String s1 = newString();
s1->set(s1, "hello");
```

Yes, the `->` and the lack of a `new` operator is a dead give away, but it sure seems to imply that we're setting the text of some `String` class to be `"hello"`.

By using function pointers, **it is possible to emulate methods in C**.

How is this accomplished?

The `String` class is actually a `struct` with a bunch of function pointers which act as a way to simulate methods. The following is a partial declaration of the `String` class:

```
typedef struct String_Struct* String;

struct String_Struct
{
    char* (*get)(const void* self);
    void (*set)(const void* self, char* value);
    int (*length)(const void* self);
};

char* getString(const void* self);
void setString(const void* self, char* value);
int lengthString(const void* self);

String newString();
```

As can be seen, the methods of the `String` class are actually function pointers to the declared function. In preparing the instance of the `String`, the `newString` function is called in order to set up the function pointers to their respective functions:

```
String newString()
{
    String self = (String)malloc(sizeof(struct String_Struct));

    self->get = &getString;
    self->set = &setString;
    self->length = &lengthString;

    self->set(self, "");
```

```
    return self;
}
```

For example, the `getString` function that is called by invoking the `get` method is defined as the following:

```
char* getString(const void* self_obj)
{
    return ((String)self_obj)->internal->value;
}
```

One thing that can be noticed is that there is no concept of an instance of an object and having methods that are actually a part of an object, so a "self object" must be passed in on each invocation. (And the `internal` is just a hidden `struct` which was omitted from the code listing earlier -- it is a way of performing information hiding, but that is not relevant to function pointers.)

So, rather than being able to do `s1->set("hello");`, one must pass in the object to perform the action on `s1->set(s1, "hello")`.

With that minor explanation having to pass in a reference to yourself out of the way, we'll move to the next part, which is **inheritance in C**.

Let's say we want to make a subclass of `String`, say an `ImmutableString`. In order to make the string immutable, the `set` method will not be accessible, while maintaining access to `get` and `length`, and force the "constructor" to accept a `char*`:

```
typedef struct ImmutableString_Struct* ImmutableString;

struct ImmutableString_Struct
{
    String base;

    char* (*get)(const void* self);
    int (*length)(const void* self);
};

ImmutableString newImmutableString(const char* value);
```

Basically, for all subclasses, the available methods are once again function pointers. This time, the declaration for the `set` method is not present, therefore, it cannot be called in a `ImmutableString`.

As for the implementation of the `ImmutableString`, the only relevant code is the "constructor" function, the `newImmutableString`:

```
ImmutableString newImmutableString(const char* value)
{
    ImmutableString self = (ImmutableString)malloc(sizeof(struct
ImmutableString_Struct));

    self->base = newString();
```

```
    self->get = self->base->get;
    self->length = self->base->length;

    self->base->set(self->base, (char*)value);

    return self;
}
```

In instantiating the `ImmutableString`, the function pointers to the `get` and `length` methods actually refer to the `String.get` and `String.length` method, by going through the `base` variable which is an internally stored `String` object.

The use of a function pointer can achieve inheritance of a method from a superclass.

We can further continue to **polymorphism in C**.

If for example we wanted to change the behavior of the `length` method to return `0` all the time in the `ImmutableString` class for some reason, all that would have to be done is to:

1. Add a function that is going to serve as the overriding `length` method.
2. Go to the "constructor" and set the function pointer to the overriding `length` method.

Adding an overriding `length` method in `ImmutableString` may be performed by adding an `lengthOverrideMethod`:

```
int lengthOverrideMethod(const void* self)
{
    return 0;
}
```

Then, the function pointer for the `length` method in the constructor is hooked up to the `lengthOverrideMethod`:

```
ImmutableString newImmutableString(const char* value)
{
    ImmutableString self = (ImmutableString)malloc(sizeof(struct
ImmutableString_Struct));

    self->base = newString();

    self->get = self->base->get;
    self->length = &lengthOverrideMethod;

    self->base->set(self->base, (char*)value);

    return self;
}
```

Now, rather than having an identical behavior for the `length` method in `ImmutableString` class as the `String` class, now the `length` method will refer to the behavior defined in the `lengthOverrideMethod` function.

I must add a disclaimer that I am still learning how to write with an object-oriented programming style in C, so there probably are points that I didn't explain well, or may just be off mark in terms of how best to implement OOP in C. But my purpose was to try to illustrate one of many uses of function pointers.

For more information on how to perform object-oriented programming in C, please refer to the following questions:

- Object-Orientation in C?
- Can you write object oriented code in C?

Share Edit Follow Flag edited May 23, 2017 at 12:18 community wiki
3 revs, 2 users 99%
coobird

31 This answer is horrible! Not only it implies that OO somehow depends on dot notation, it also encourages putting junk into your objects! – Alexei Averchenko Sep 16, 2012 at 14:30

34 This is OO all right, but not anywhere near the C-style OO. What you have brokenly implemented is Javascript-style prototype-based OO. To get C++/Pascal-style OO, you'd need to: 1. Have a const struct for a virtual table of each *class* with virtual members. 2. Have pointer to that struct in polymorphic objects. 3. Call virtual methods via the virtual table, and all other methods directly -- usually by sticking to some `ClassName_methodName` function naming convention. Only then you get the same runtime and storage costs as you do in C++ and Pascal.
– Kuba hasn't forgotten Monica Mar 18, 2013 at 21:53

21 Working OO with a language that is not intended to be OO is always a bad idea. If you want OO and still have C just work with C++. – rbaleksandar Jul 4, 2013 at 15:21

29 @rbaleksandar Tell that to the Linux kernel developers. "*always a bad idea*" is strictly your opinion, with which I firmly disagree. – Jonathon Reinhart Apr 30, 2015 at 12:31

8 I like this answer but don't cast malloc – cat Sep 29, 2016 at 14:41

262

The guide to getting fired: How to abuse function pointers in GCC on x86 machines by compiling your code by hand:

These string literals are bytes of 32-bit x86 machine code. `0xC3` is an x86 `ret` instruction.

You wouldn't normally write these by hand, you'd write in assembly language and then use an assembler like `nasm` to assemble it into a flat binary which you hexdump into a C string literal.

1. Returns the current value on the EAX register

```
int eax = ((int(*)())("\xc3 <- This returns the value of the EAX register"))();
```

2. Write a swap function

```
int a = 10, b = 20;
((void(*)
```

```
(int*,int*))"\x8b\x44\x24\x04\x8b\x5c\x24\x08\x8b\x00\x8b\x1b\x31\xc3\x31\xd8\x31\xc
<- This swaps the values of a and b")(&a,&b);
```

3. Write a for-loop counter to 1000, calling some function each time

```
((int(*)
())"\x66\x31\xc0\x8b\x5c\x24\x04\x66\x40\x50\xff\xd3\x58\x66\x3d\xe8\x03\x75\xf4\xc3
(&function); // calls function with 1->1000
```

4. You can even write a recursive function that counts to 100

```
const char* lol =
"\x8b\x5c\x24\x4\x3d\xe8\x3\x0\x0\x7e\x2\x31\xc0\x83\xf8\x64\x7d\x6\x40\x53\xff\xd3\
<- Recursively calls the function at address lol.";
i = ((int(*)())(lol))(lol);
```

Note that compilers place string literals in the `.rodata` section (or `.rdata` on Windows), which is linked as part of the text segment (along with code for functions).

The text segment has Read+Exec permission, so casting string literals to function pointers works without needing `mprotect()` or `VirtualProtect()` system calls like you'd need for dynamically allocated memory. (Or `gcc -z execstack` links the program with stack + data segment + heap executable, as a quick hack.)

To disassemble these, you can compile this to put a label on the bytes, and use a disassembler.

```
// at global scope
const char swap[] =
"\x8b\x44\x24\x04\x8b\x5c\x24\x08\x8b\x00\x8b\x1b\x31\xc3\x31\xd8\x31\xc3\x8b\x4c\x2
<- This swaps the values of a and b";
```

Compiling with `gcc -c -m32 foo.c` and disassembling with `objdump -D -rwC -Mintel`, we can get the assembly, and find out that this code violates the ABI by clobbering EBX (a call-preserved register) and is generally inefficient.

```
00000000 <swap>:
   0:   8b 44 24 04             mov    eax,DWORD PTR [esp+0x4]   # load int *a
arg from the stack
   4:   8b 5c 24 08             mov    ebx,DWORD PTR [esp+0x8]   # ebx = b
   8:   8b 00                   mov    eax,DWORD PTR [eax]       # dereference:
eax = *a
   a:   8b 1b                   mov    ebx,DWORD PTR [ebx]
   c:   31 c3                   xor    ebx,eax                # pointless xor-
swap
   e:   31 d8                   xor    eax,ebx                # instead of just
storing with opposite registers
  10:   31 c3                   xor    ebx,eax
  12:   8b 4c 24 04             mov    ecx,DWORD PTR [esp+0x4]  # reload a from
the stack
  16:   89 01                   mov    DWORD PTR [ecx],eax     # store to *a
  18:   8b 4c 24 08             mov    ecx,DWORD PTR [esp+0x8]
```

```
  1c:   89 19                   mov    DWORD PTR [ecx],ebx
  1e:   c3                      ret

  not shown: the later bytes are ASCII text documentation
  they're not executed by the CPU because the ret instruction sends execution
back to the caller
```

This machine code will (probably) work in 32-bit code on Windows, Linux, OS X, and so on: the default calling conventions on all those OSes pass args on the stack instead of more efficiently in registers. But EBX is call-preserved in all the normal calling conventions, so using it as a scratch register without saving/restoring it can easily make the caller crash.

Share Edit Follow Flag edited Oct 9, 2018 at 13:58 community wiki 4 revs, 3 users 50% Lee

9 Note: this doesn't work if Data Execution Prevention is enabled (e.g. on Windows XP SP2+), because C strings are not normally marked as executable. – SecurityMatt Feb 12, 2013 at 5:53

5 Hi Matt! Depending on the optimization level, GCC will often inline string constants into the TEXT segment, so this will work even on newer version of windows provided that you don't disallow this type of optimization. (IIRC, the MINGW version at the time of my post over two years ago inlines string literals at the default optimization level) – Lee Jan 2, 2014 at 6:20

11 could someone please explain what's happening here? What are those weird looking string literals? – ajay Jan 20, 2014 at 10:17

65 @ajay It looks like he's writing raw hexidecimal values (for instance '\x00' is the same as '/0', they're both equal to 0) into a string, then casting the string into a C function pointer, then executing the C function pointer because he's the devil. – ejk314 Feb 21, 2014 at 21:27

4 hi FUZxxl, I think it might vary based on the compiler and the operating system version. The above code seems to run fine on codepad.org; codepad.org/FMSDQ3ME – Lee Mar 13, 2014 at 0:48

125

One of my favorite uses for function pointers is as cheap and easy iterators -

```
#include <stdio.h>
#define MAX_COLORS  256

typedef struct {
    char* name;
    int red;
    int green;
    int blue;
} Color;

Color Colors[MAX_COLORS];


void eachColor (void (*fp)(Color *c)) {
    int i;
    for (i=0; i<MAX_COLORS; i++)
        (*fp)(&Colors[i]);
}

void printColor(Color* c) {
```

```
    if (c->name)
        printf("%s = %i,%i,%i\n", c->name, c->red, c->green, c->blue);
}

int main() {
    Colors[0].name="red";
    Colors[0].red=255;
    Colors[1].name="blue";
    Colors[1].blue=255;
    Colors[2].name="black";

    eachColor(printColor);
}
```

Share Edit Follow Flag edited Jul 24, 2012 at 16:40 community wiki
2 revs, 2 users 99%
Nick

10 You should also pass a pointer to user-specified data if you want to somehow extract any output from iterations (think closures). – Alexei Averchenko Sep 16, 2012 at 14:32

3 Agreed. All of my iterators look like this: `int (*cb)(void *arg, ...)` . The return value of the iterator also lets me stop early (if nonzero). – Jonathon Reinhart Apr 30, 2015 at 12:35

26

Function pointers become easy to declare once you have the basic declarators:

- id: `ID` : *ID is a*
- Pointer: `*D` : *D pointer to*
- Function: `D(<parameters>)` : *D function taking < parameters > returning*

While D is another declarator built using those same rules. In the end, somewhere, it ends with `ID` (see below for an example), which is the name of the declared entity. Let's try to build a function taking a pointer to a function taking nothing and returning int, and returning a pointer to a function taking a char and returning int. With type-defs it's like this

```
typedef int ReturnFunction(char);
typedef int ParameterFunction(void);
ReturnFunction *f(ParameterFunction *p);
```

As you see, it's pretty easy to build it up using typedefs. Without typedefs, it's not hard either with the above declarator rules, applied consistently. As you see i missed out the part the pointer points to, and the thing the function returns. That's what appears at the very left of the declaration, and is not of interest: It's added at the end if one built up the declarator already. Let's do that. Building it up consistently, first wordy - showing the structure using `[` and `]` :

```
function taking
    [pointer to [function taking [void] returning [int]]]
returning
    [pointer to [function taking [char] returning [int]]]
```

As you see, one can describe a type completely by appending declarators one after each other. Construction can be done in two ways. One is bottom-up, starting with the very right thing (leaves) and working the way through up to the identifier. The other way is top-down, starting at the identifier, working the way down to the leaves. I'll show both ways.

Bottom Up

Construction starts with the thing at the right: The thing returned, which is the function taking char. To keep the declarators distinct, i'm going to number them:

```
D1(char);
```

Inserted the char parameter directly, since it's trivial. Adding a pointer to declarator by replacing `D1` by `*D2`. Note that we have to wrap parentheses around `*D2`. That can be known by looking up the precedence of the `*`-operator and the function-call operator `()`. Without our parentheses, the compiler would read it as `*(D2(char p))`. But that would not be a plain replace of D1 by `*D2` anymore, of course. Parentheses are always allowed around declarators. So you don't make anything wrong if you add too much of them, actually.

```
(*D2)(char);
```

Return type is complete! Now, let's replace `D2` by the function declarator *function taking* `<parameters>` *returning*, which is `D3(<parameters>)` which we are at now.

```
(*D3(<parameters>))(char)
```

Note that no parentheses are needed, since we *want* `D3` to be a function-declarator and not a pointer declarator this time. Great, only thing left is the parameters for it. The parameter is done exactly the same as we've done the return type, just with `char` replaced by `void`. So i'll copy it:

```
(*D3(   (*ID1)(void)))(char)
```

I've replaced `D2` by `ID1`, since we are finished with that parameter (it's already a pointer to a function - no need for another declarator). `ID1` will be the name of the parameter. Now, i told above at the end one adds the type which all those declarator modify - the one appearing at the very left of every declaration. For functions, that becomes the return type. For pointers the pointed to type etc... It's interesting when written down the type, it will appear in the opposite order, at the very right :) Anyway, substituting it yields the complete declaration. Both times `int` of course.

```
int (*ID0(int (*ID1)(void)))(char)
```

I've called the identifier of the function `ID0` in that example.

Top Down

This starts at the identifier at the very left in the description of the type, wrapping that declarator as we walk our way through the right. Start with *function taking < parameters > returning*

```
ID0(<parameters>)
```

The next thing in the description (after "returning") was *pointer to*. Let's incorporate it:

```
*ID0(<parameters>)
```

Then the next thing was *functon taking < parameters > returning*. The parameter is a simple char, so we put it in right away again, since it's really trivial.

```
(*ID0(<parameters>))(char)
```

Note the parentheses we added, since we again want that the `*` binds first, and *then* the `(char)`. Otherwise it would read *function taking < parameters > returning function ...*. Noes, functions returning functions aren't even allowed.

Now we just need to put < parameters >. I will show a short version of the deriveration, since i think you already by now have the idea how to do it.

```
pointer to: *ID1
... function taking void returning: (*ID1)(void)
```

Just put `int` before the declarators like we did with bottom-up, and we are finished

```
int (*ID0(int (*ID1)(void)))(char)
```

The nice thing

Is bottom-up or top-down better? I'm used to bottom-up, but some people may be more comfortable with top-down. It's a matter of taste i think. Incidentally, if you apply all the operators in that declaration, you will end up getting an int:

```
int v = (*ID0(some_function_pointer))(some_char);
```

That is a nice property of declarations in C: The declaration asserts that if those operators are used in an expression using the identifier, then it yields the type on the very left. It's like that for arrays too.

Hope you liked this little tutorial! Now we can link to this when people wonder about the strange declaration syntax of functions. I tried to put as little C internals as possible. Feel free

to edit/fix things in it.

Share Edit Follow Flag edited May 9, 2009 at 13:38 community wiki
3 revs
Johannes Schaub - litb

25

Another good use for function pointers: Switching between versions painlessly

They're very handy to use for when you want different functions at different times, or different phases of development. For instance, I'm developing an application on a host computer that has a console, but the final release of the software will be put on an Avnet ZedBoard (which has ports for displays and consoles, but they are not needed/wanted for the final release). So during development, I will use `printf` to view status and error messages, but when I'm done, I don't want anything printed. Here's what I've done:

version.h

```
// First, undefine all macros associated with version.h
#undef DEBUG_VERSION
#undef RELEASE_VERSION
#undef INVALID_VERSION


// Define which version we want to use
#define DEBUG_VERSION       // The current version
// #define RELEASE_VERSION  // To be uncommented when finished debugging

#ifndef __VERSION_H_      /* prevent circular inclusions */
    #define __VERSION_H_  /* by using protection macros */
    void board_init();
    void noprintf(const char *c, ...); // mimic the printf prototype
#endif

// Mimics the printf function prototype. This is what I'll actually
// use to print stuff to the screen
void (* zprintf)(const char*, ...);

// If debug version, use printf
#ifdef DEBUG_VERSION
    #include <stdio.h>
#endif

// If both debug and release version, error
#ifdef DEBUG_VERSION
#ifdef RELEASE_VERSION
    #define INVALID_VERSION
#endif
#endif

// If neither debug or release version, error
#ifndef DEBUG_VERSION
#ifndef RELEASE_VERSION
    #define INVALID_VERSION
```

```
#endif
#endif

#ifdef INVALID_VERSION
    // Won't allow compilation without a valid version define
    #error "Invalid version definition"
#endif
```

In `version.c` I will define the 2 function prototypes present in `version.h`

version.c

```
#include "version.h"

/*****************************************************************************/
/**
* @name board_init
*
* Sets up the application based on the version type defined in version.h.
* Includes allowing or prohibiting printing to STDOUT.
*
* MUST BE CALLED FIRST THING IN MAIN
*
* @return    None
*
*****************************************************************************/
void board_init()
{
    // Assign the print function to the correct function pointer
    #ifdef DEBUG_VERSION
        zprintf = &printf;
    #else
        // Defined below this function
        zprintf = &noprintf;
    #endif
}

/*****************************************************************************/
/**
* @name noprintf
*
* simply returns with no actions performed
*
* @return   None
*
*****************************************************************************/
void noprintf(const char* c, ...)
{
    return;
}
```

Notice how the function pointer is prototyped in `version.h` as

```
void (* zprintf)(const char *, ...);
```

When it is referenced in the application, it will start executing wherever it is pointing, which has yet to be defined.

In `version.c`, notice in the `board_init()` function where `zprintf` is assigned a unique function (whose function signature matches) depending on the version that is defined in `version.h`

`zprintf = &printf;` zprintf calls printf for debugging purposes

or

`zprintf = &noprint;` zprintf just returns and will not run unnecessary code

Running the code will look like this:

mainProg.c

```
#include "version.h"
#include <stdlib.h>
int main()
{
    // Must run board_init(), which assigns the function
    // pointer to an actual function
    board_init();

    void *ptr = malloc(100); // Allocate 100 bytes of memory
    // malloc returns NULL if unable to allocate the memory.

    if (ptr == NULL)
    {
        zprintf("Unable to allocate memory\n");
        return 1;
    }

    // Other things to do...
    return 0;
}
```

The above code will use `printf` if in debug mode, or do nothing if in release mode. This is much easier than going through the entire project and commenting out or deleting code. All that I need to do is change the version in `version.h` and the code will do the rest!

Share Edit Follow Flag edited Jun 11, 2013 at 15:36 community wiki 2 revs Zack Sheffield

7 U stand to lose a lot of performance time. Instead you could use a macro that enables and disables a section of code based on Debug / Release. – AlphaGoku May 11, 2018 at 13:41

24

Function pointer is usually defined by `typedef`, and used as param & return value.

Above answers already explained a lot, I just give a full example:

```c
#include <stdio.h>

#define NUM_A 1
#define NUM_B 2

// define a function pointer type
typedef int (*two_num_operation)(int, int);

// an actual standalone function
static int sum(int a, int b) {
    return a + b;
}

// use function pointer as param,
static int sum_via_pointer(int a, int b, two_num_operation funp) {
    return (*funp)(a, b);
}

// use function pointer as return value,
static two_num_operation get_sum_fun() {
    return &sum;
}

// test - use function pointer as variable,
void test_pointer_as_variable() {
    // create a pointer to function,
    two_num_operation sum_p = &sum;
    // call function via pointer
    printf("pointer as variable:\t %d + %d = %d\n", NUM_A, NUM_B, (*sum_p)(NUM_A,
NUM_B));
}

// test - use function pointer as param,
void test_pointer_as_param() {
    printf("pointer as param:\t %d + %d = %d\n", NUM_A, NUM_B,
sum_via_pointer(NUM_A, NUM_B, &sum));
}

// test - use function pointer as return value,
void test_pointer_as_return_value() {
    printf("pointer as return value:\t %d + %d = %d\n", NUM_A, NUM_B,
(*get_sum_fun())(NUM_A, NUM_B));
}

int main() {
    test_pointer_as_variable();
    test_pointer_as_param();
    test_pointer_as_return_value();

    return 0;
}
```

Share Edit Follow Flag

edited Jun 8, 2019 at 6:56

community wiki
2 revs, 2 users 99%
Eric Wang

20

Starting from scratch function has Some Memory Address From Where They start executing. In Assembly Language They Are called as (call "function's memory address").Now come back to C If function has a memory address then they can be manipulated by Pointers in C.So By the rules of C

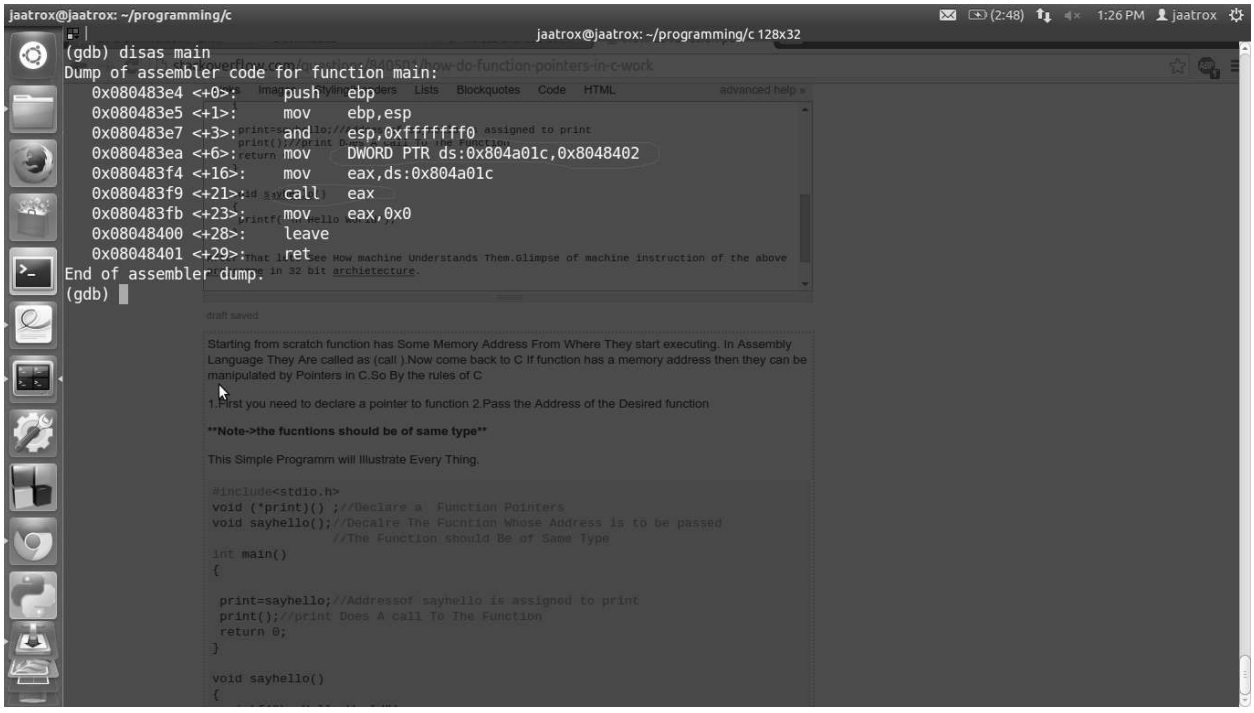
1.First you need to declare a pointer to function 2.Pass the Address of the Desired function

****Note->the functions should be of same type****

This Simple Programme will Illustrate Every Thing.

```
#include<stdio.h>
void (*print)() ;//Declare a  Function Pointers
void sayhello();//Declare The Function Whose Address is to be passed
                //The Functions should Be of Same Type
int main()
{
 print=sayhello;//Addressof sayhello is assigned to print
 print();//print Does A call To The Function
 return 0;
}

void sayhello()
{
 printf("\n Hello World");
}
```



After That lets See How machine Understands Them.Glimpse of machine instruction of the above programme in 32 bit architecture.

The red mark area is showing how the address is being exchanged and storing in eax. Then their is a call instruction on eax. eax contains the desired address of the function.

Share Edit Follow Flag

edited Jun 8, 2019 at 6:56

community wiki
2 revs, 2 users 97%
Mohit Dabas

How do I use a function pointer returned from a method? `something()` seems to just crash the program. I have some context and failed code here: stackoverflow.com/questions/67152106 – Aaron Franke Apr 18, 2021 at 18:41

18

A function pointer is a variable that contains the address of a function. Since it is a pointer variable though with some restricted properties, you can use it pretty much like you would any other pointer variable in data structures.

The only exception I can think of is treating the function pointer as pointing to something other than a single value. Doing pointer arithmetic by incrementing or decrementing a function pointer or adding/subtracting an offset to a function pointer isn't really of any utility as a function pointer only points to a single thing, the entry point of a function.

The size of a function pointer variable, the number of bytes occupied by the variable, may vary depending on the underlying architecture, e.g. x32 or x64 or whatever.

The declaration for a function pointer variable needs to specify the same kind of information as a function declaration in order for the C compiler to do the kinds of checks that it normally does. If you don't specify a parameter list in the declaration/definition of the function pointer, the C compiler will not be able to check the use of parameters. There are cases when this lack of checking can be useful however just remember that a safety net has been removed.

Some examples:

```
int func (int a, char *pStr);     // declares a function

int (*pFunc)(int a, char *pStr);  // declares or defines a function pointer

int (*pFunc2) ();                 // declares or defines a function pointer, no
parameter list specified.

int (*pFunc3) (void);             // declares or defines a function pointer, no
arguments.
```

The first two declararations are somewhat similar in that:

- `func` is a function that takes an `int` and a `char *` and returns an `int`
- `pFunc` is a function pointer to which is assigned the address of a function that takes an `int` and a `char *` and returns an `int`

So from the above we could have a source line in which the address of the function `func()` is assigned to the function pointer variable `pFunc` as in `pFunc = func;` .

Notice the syntax used with a function pointer declaration/definition in which parenthesis are used to overcome the natural operator precedence rules.

```
int *pfunc(int a, char *pStr);    // declares a function that returns int pointer
int (*pFunc)(int a, char *pStr);  // declares a function pointer that returns an
int
```

Several Different Usage Examples

Some examples of usage of a function pointer:

```
int (*pFunc) (int a, char *pStr);    // declare a simple function pointer
variable
int (*pFunc[55])(int a, char *pStr); // declare an array of 55 function pointers
int (**pFunc)(int a, char *pStr);    // declare a pointer to a function pointer
variable
struct {                             // declare a struct that contains a function
pointer
    int x22;
    int (*pFunc)(int a, char *pStr);
} thing = {0, func};                 // assign values to the struct variable
char * xF (int x, int (*p)(int a, char *pStr));  // declare a function that has a
function pointer as an argument
char * (*pxF) (int x, int (*p)(int a, char *pStr));  // declare a function
pointer that points to a function that has a function pointer as an argument
```

You can use variable length parameter lists in the definition of a function pointer.

```
int sum (int a, int b, ...);
int (*psum)(int a, int b, ...);
```

Or you can not specify a parameter list at all. This can be useful but it eliminates the opportunity for the C compiler to perform checks on the argument list provided.

```
int  sum ();      // nothing specified in the argument list so could be anything
or nothing
int (*psum)();
int  sum2(void);  // void specified in the argument list so no parameters when
calling this function
int (*psum2)(void);
```

C style Casts

You can use C style casts with function pointers. However be aware that a C compiler may be lax about checks or provide warnings rather than errors.

```
int sum (int a, char *b);
int (*psplsum) (int a, int b);
psplsum = sum;               // generates a compiler warning
psplsum = (int (*)(int a, int b)) sum;   // no compiler warning, cast to function
pointer
psplsum = (int *(int a, int b)) sum;     // compiler error of bad cast generated,
parenthesis are required.
```

Compare Function Pointer to Equality

You can check that a function pointer is equal to a particular function address using an `if` statement though I am not sure how useful that would be. Other comparison operators would seem to have even less utility.

```
static int func1(int a, int b) {
    return a + b;
}

static int func2(int a, int b, char *c) {
```

```
    return c[0] + a + b;
}

static int func3(int a, int b, char *x) {
    return a + b;
}

static char *func4(int a, int b, char *c, int (*p)())
{
    if (p == func1) {
        p(a, b);
    }
    else if (p == func2) {
        p(a, b, c);      // warning C4047: '==': 'int (__cdecl *)()' differs in
levels of indirection from 'char *(__cdecl *)(int,int,char *)'
    } else if (p == func3) {
        p(a, b, c);
    }
    return c;
}
```

An Array of Function Pointers

And if you want to have an array of function pointers each of the elements of which the argument list has differences then you can define a function pointer with the argument list unspecified (not `void` which means no arguments but just unspecified) something like the following though you may see warnings from the C compiler. This also works for a function pointer parameter to a function:

```
int(*p[])() = {       // an array of function pointers
    func1, func2, func3
};
int(**pp)();          // a pointer to a function pointer


p[0](a, b);
p[1](a, b, 0);
p[2](a, b);      // oops, left off the last argument but it compiles anyway.

func4(a, b, 0, func1);
func4(a, b, 0, func2);  // warning C4047: 'function': 'int (__cdecl *)()' differs
in levels of indirection from 'char *(__cdecl *)(int,int,char *)'
func4(a, b, 0, func3);

    // iterate over the array elements using an array index
for (i = 0; i < sizeof(p) / sizeof(p[0]); i++) {
    func4(a, b, 0, p[i]);
}
    // iterate over the array elements using a pointer
for (pp = p; pp < p + sizeof(p)/sizeof(p[0]); pp++) {
    (*pp)(a, b, 0);          // pointer to a function pointer so must dereference
it.
    func4(a, b, 0, *pp);     // pointer to a function pointer so must dereference
it.
}
```

C style `namespace` Using Global `struct` with Function Pointers

You can use the `static` keyword to specify a function whose name is file scope and then assign this to a global variable as a way of providing something similar to the `namespace`

functionality of C++.

In a header file define a struct that will be our namespace along with a global variable that uses it.

```
typedef struct {
   int (*func1) (int a, int b);             // pointer to function that returns
an int
   char *(*func2) (int a, int b, char *c);  // pointer to function that returns a
pointer
} FuncThings;

extern const FuncThings FuncThingsGlobal;
```

Then in the C source file:

```
#include "header.h"

// the function names used with these static functions do not need to be the
// same as the struct member names. It's just helpful if they are when trying
// to search for them.
// the static keyword ensures these names are file scope only and not visible
// outside of the file.
static int func1 (int a, int b)
{
    return a + b;
}

static char *func2 (int a, int b, char *c)
{
    c[0] = a % 100; c[1] = b % 50;
    return c;
}

const FuncThings FuncThingsGlobal = {func1, func2};
```

This would then be used by specifying the complete name of global struct variable and member name to access the function. The `const` modifier is used on the global so that it can not be changed by accident.

```
int abcd = FuncThingsGlobal.func1 (a, b);
```

Application Areas of Function Pointers

A DLL library component could do something similar to the C style `namespace` approach in which a particular library interface is requested from a factory method in a library interface which supports the creation of a `struct` containing function pointers.. This library interface loads the requested DLL version, creates a struct with the necessary function pointers, and then returns the struct to the requesting caller for use.

```
typedef struct {
    HMODULE  hModule;
    int (*Func1)();
    int (*Func2)();
    int(*Func3)(int a, int b);
```

```
} LibraryFuncStruct;

int  LoadLibraryFunc LPCTSTR  dllFileName, LibraryFuncStruct *pStruct)
{
    int  retStatus = 0;   // default is an error detected

    pStruct->hModule = LoadLibrary (dllFileName);
    if (pStruct->hModule) {
        pStruct->Func1 = (int (*)()) GetProcAddress (pStruct->hModule, "Func1");
        pStruct->Func2 = (int (*)()) GetProcAddress (pStruct->hModule, "Func2");
        pStruct->Func3 = (int (*)(int a, int b)) GetProcAddress(pStruct->hModule,
"Func3");
        retStatus = 1;
    }

    return retStatus;
}

void FreeLibraryFunc (LibraryFuncStruct *pStruct)
{
    if (pStruct->hModule) FreeLibrary (pStruct->hModule);
    pStruct->hModule = 0;
}
```

and this could be used as in:

```
LibraryFuncStruct myLib = {0};
LoadLibraryFunc (L"library.dll", &myLib);
//  ....
myLib.Func1();
//  ....
FreeLibraryFunc (&myLib);
```

The same approach can be used to define an abstract hardware layer for code that uses a particular model of the underlying hardware. Function pointers are filled in with hardware specific functions by a factory to provide the hardware specific functionality that implements functions specified in the abstract hardware model. This can be used to provide an abstract hardware layer used by software which calls a factory function in order to get the specific hardware function interface then uses the function pointers provided to perform actions for the underlying hardware without needing to know implementation details about the specific target.

Function Pointers to create Delegates, Handlers, and Callbacks

You can use function pointers as a way to delegate some task or functionality. The classic example in C is the comparison delegate function pointer used with the Standard C library functions `qsort()` and `bsearch()` to provide the collation order for sorting a list of items or performing a binary search over a sorted list of items. The comparison function delegate specifies the collation algorithm used in the sort or the binary search.

Another use is similar to applying an algorithm to a C++ Standard Template Library container.

```
void * ApplyAlgorithm (void *pArray, size_t sizeItem, size_t nItems, int (*p)
(void *)) {
    unsigned char *pList = pArray;
    unsigned char *pListEnd = pList + nItems * sizeItem;
```

```
    for ( ; pList < pListEnd; pList += sizeItem) {
        p (pList);
    }

    return pArray;
}

int pIncrement(int *pI) {
    (*pI)++;

    return 1;
}

void * ApplyFold(void *pArray, size_t sizeItem, size_t nItems, void * pResult,
int(*p)(void *, void *)) {
    unsigned char *pList = pArray;
    unsigned char *pListEnd = pList + nItems * sizeItem;
    for (; pList < pListEnd; pList += sizeItem) {
        p(pList, pResult);
    }

    return pArray;
}

int pSummation(int *pI, int *pSum) {
    (*pSum) += *pI;

    return 1;
}

// source code and then lets use our function.
int intList[30] = { 0 }, iSum = 0;

ApplyAlgorithm(intList, sizeof(int), sizeof(intList) / sizeof(intList[0]),
pIncrement);
ApplyFold(intList, sizeof(int), sizeof(intList) / sizeof(intList[0]), &iSum,
pSummation);
```

Another example is with GUI source code in which a handler for a particular event is registered by providing a function pointer which is actually called when the event happens. The Microsoft MFC framework with its message maps uses something similar to handle Windows messages that are delivered to a window or thread.

Asynchronous functions that require a callback are similar to an event handler. The user of the asynchronous function calls the asynchronous function to start some action and provides a function pointer which the asynchronous function will call once the action is complete. In this case the event is the asynchronous function completing its task.

Share Edit Follow Flag answered Aug 20, 2018 at 5:47 community wiki Richard Chambers

---

18

One of the big uses for function pointers in C is to call a function selected at run-time. For example, the C run-time library has two routines, `qsort` and `bsearch`, which take a pointer to a function that is called to compare two items being sorted; this allows you to sort or search, respectively, anything, based on any criteria you wish to use.

A very basic example, if there is one function called `print(int x, int y)` which in turn may require to call a function (either `add()` or `sub()`, which are of the same type) then what we will do, we will add one function pointer argument to the `print()` function as shown below:

```
#include <stdio.h>

int add()
{
   return (100+10);
}

int sub()
{
   return (100-10);
}

void print(int x, int y, int (*func)())
{
    printf("value is: %d\n", (x+y+(*func)()));
}

int main()
{
    int x=100, y=200;
    print(x,y,add);
    print(x,y,sub);

    return 0;
}
```

The output is:

```
value is: 410
value is: 390
```

Share Edit Follow Flag edited Mar 13, 2019 at 17:09 community wiki
3 revs, 3 users 76%
Vamsi

1

Pointers to functions are useful because, as "The C Programming Language" book says, functions in C are not variables. This means,

```
// Say you have add function
int add(int x, int y){
    return x + y;
}

// Say you have another add function
int another_add(int x, int y){
    return y + x;
}


int main(){
    // Although the types of another_add and add are same
    // You can't do
```

```
    another_add = add

    // You have a compute function that takes a function of int's signature
    int (*compute)(int, int);

    // You won't even be able to pass functions to other functions
    // (Although when you do, C is just passing the pointer to that function)
    // So, compute(add) is really compute(&add)
    // But you can create a pointer to functions that are variables
    // you can assign to and/or pass to other functions

    int (*operation)(int, int);
    // Now you can do
    operation = &add;
    // You could also do, the following to do the same thing
    // When a function is passed in right hand side of assignment,
    // C knows that you mean pointer, and you don't need explicit &
    operation = add;
}
```

Similarly, an array is also not a variable in C. You can make up a similar example as above and test out.

Share Edit Follow Flag answered Oct 18, 2022 at 14:51 community wiki Suman Chapai

---

1

Since function pointers are often typed callbacks, you might want to have a look at type safe callbacks. The same applies to entry points, etc of functions that are not callbacks.

C is quite fickle and forgiving at the same time :)

Share Edit Follow Flag answered May 9, 2009 at 13:56 community wiki Tim Post

Link is broken. – julaine Nov 24 at 14:30